

МАШИННОЕ ОБУЧЕНИЕ С PYTORCH И SCIKIT-LEARN

Разрабатывайте модели машинного
и глубокого обучения с помощью Python

Предисловие:
Дмитро Джулгаков,
ведущий специалист PyTorch Core

Себастьян Рашка
Юси (Хэйден) Лю
Вахид Мирджалили

Machine Learning with PyTorch and Scikit-Learn

Develop machine learning and deep learning models with Python

Sebastian Raschka

Yuxi (Hayden) Liu

Vahid Mirjalili

Packt

BIRMINGHAM—MUMBAI

“Python” and the Python Logo are trademarks of the Python Software Foundation.

Себастьян Рашка
Юси (Хэйден) Лю
Вахид Мирджалили

МАШИННОЕ ОБУЧЕНИЕ С PYTORCH И SCIKIT-LEARN

УДК 004.43

ББК 32.973.26-018.1

Р28

Рашка, С.

P28 Машинное обучение с PyTorch и Scikit-Learn: Пер. с англ. / С. Рашка, Ю. Лю, В. Мирджалили. — Астана: Фолиант, 2024. — 688 с.: ил.

ISBN 978-601-11-0034-2

Исчерпывающее руководство по машинному (МО) и глубокому обучению с использованием языка программирования Python, фреймворка PyTorch и библиотеки Scikit-Learn. Рассмотрены основы МО, алгоритмы для задач классификации, классификаторы на основе Scikit-Learn, предварительная обработка и сжатие данных, современные методы оценки моделей и объединение различных моделей для ансамблевого обучения. Рассказано о применении МО для анализа текста и прогнозирования непрерывных целевых переменных с помощью регрессионного анализа, кластерном анализе и обучении без учителя, показано построение многослойной искусственной нейронной сети с нуля. Раскрыты продвинутые возможности PyTorch для решения сложных задач. Описано применение глубоких сверточных и рекуррентных нейронных сетей, трансформеров, генеративных состязательных и графовых нейронных сетей. Особое внимание удалено обучению с подкреплением для систем принятия решений в сложных средах.

Электронный архив содержит цветные иллюстрации и коды всех примеров.

Для программистов в области машинного обучения

УДК 004.43

ББК 32.973.26-018.1

© 2024 Foliant Publishing House LLP

© Packt Publishing 2022. First published in the English language under the title ‘Machine Learning with PyTorch and Scikit-Learn’ – (9781801819312)’

Впервые опубликовано на английском языке под названием ‘Machine Learning with PyTorch and Scikit-Learn – (9781801819312)’

Подписано в печать 06.05.24.

Формат 70×100¹/16. Печать офсетная. Усл. печ. л. 55,47.

Тираж 1500 экз. Заказ № 9700.

ТОО «Издательство «Фолиант».

Республика Казахстан, 010000, г. Астана, ул. Ш. Айманова, 13.

Отпечатано с готового оригинал-макета

ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-80181-931-2 (англ.)

ISBN 978-601-11-0034-2 (каз.)

© Packt Publishing, 2022

© Издание на русском языке.

ТОО «Издательство «Фолиант», 2024

Оглавление

https://t.me/it_boooks/2

Вступительное слово	15
Об авторах.....	17
Предисловие	19
Для кого эта книга?	20
О чем рассказывает книга?	20
Как получить максимальную отдачу от этой книги?	22
Скачивание примеров кода	23
Скачивание цветных изображений	23
Соглашения и условные обозначения	24
Глава 1. Как помочь компьютеру учиться на данных?.....	25
1.1. Интеллектуальные машины для преобразования данных в знания	25
1.2. Три типа машинного обучения	26
1.2.1. Предсказание будущего при помощи обучения с учителем	27
Прогнозирование меток классов в задаче классификации.....	28
Прогнозирование непрерывного значения в задаче регрессии	29
1.2.2. Применение обучения с подкреплением для решения интерактивных задач	30
1.2.3. Обнаружение скрытых структур с помощью обучения без учителя.....	31
Поиск подгрупп путем кластеризации.....	32
Уменьшение размерности для сжатия данных.....	32
1.3. Введение в основную терминологию и обозначения	33
1.3.1. Соглашения и условные обозначения, используемые в этой книге	33
1.3.2. Терминология машинного обучения	35
1.4. Общие принципы построения систем машинного обучения	36
1.4.1. Предварительная обработка: приведение данных к нужному виду	36
1.4.2. Обучение и выбор прогностической модели	37
1.4.3. Оценка моделей и прогнозирование незнакомых экземпляров данных	38
1.5. Использование Python для машинного обучения.....	38
1.5.1. Установка Python и пакетов из Python Package Index	39
1.5.2. Использование дистрибутива Python Anaconda и менеджера пакетов	39
1.5.3. Пакеты для научных вычислений, науки о данных и машинного обучения	40
1.6. Заключение.....	42
Глава 2. Простые алгоритмы машинного обучения для задач классификации	43
2.1. Искусственные нейроны: краткий экскурс в историю машинного обучения	43
2.1.1. Формальное определение искусственного нейрона	44
2.1.2. Правило обучения персептрона	46

2.2. Реализация алгоритма обучения персептрона на Python	49
2.2.1. Объектно-ориентированный API персептрона.....	49
2.2.2. Обучение модели персептрона на наборе данных Iris.....	52
2.3. Адаптивные линейные нейроны и сходимость обучения.....	57
2.3.1. Минимизация функции потерь с помощью градиентного спуска.....	58
2.3.2. Реализация Adaline на Python	61
2.3.3. Улучшение градиентного спуска за счет масштабирования признаков.....	65
2.3.4. Крупномасштабное машинное обучение и стохастический градиентный спуск	67
2.4. Заключение.....	71
Глава 3. Знакомство с классификаторами машинного обучения на основе scikit-learn	73
3.1. Выбор алгоритма классификации.....	73
3.2. Первые шаги с scikit-learn: обучение персептрона	74
3.3. Моделирование вероятностей классов с помощью логистической регрессии	79
3.3.1. Логистическая регрессия и условные вероятности	79
3.3.2. Изучение весов модели с помощью логистической функции потерь.....	83
3.3.3. Преобразование Adaline в алгоритм логистической регрессии	86
3.3.4. Обучение модели логистической регрессии с помощью scikit-learn.....	89
3.3.5. Борьба с переобучением путем регуляризации.....	92
3.4. Классификация по наибольшему отступу с помощью метода опорных векторов.....	95
3.4.1. Смысл максимизации зазора.....	95
3.4.2. Работа с нелинейно разделимым случаем при использовании резервных переменных	96
3.4.3. Альтернативные реализации алгоритмов в scikit-learn	98
3.5. Решение нелинейных задач с использованием ядерного варианта SVM	99
3.5.1. Ядерные методы для линейно неразделимых данных	99
3.5.2. Использование ядерного трюка для поиска разделяющих гиперплоскостей в многомерном пространстве	101
3.6. Обучение дерева решений.....	104
3.6.1. Максимизация IG: получение наибольшей отдачи от затраченных усилий	105
3.6.2. Построение дерева решений.....	110
3.6.3. Объединение нескольких деревьев решений с помощью случайных лесов.....	112
3.7. К-ближайшие соседи: «ленивый» алгоритм обучения	115
3.8. Заключение.....	118
Глава 4. Предварительная обработка данных для создания качественных обучающих наборов	120
4.1. Как поступать с отсутствующими данными?	120
4.1.1. Выявление пропущенных значений в табличных данных	121
4.1.2. Исключение обучающих записей или признаков с пропущенными значениями	122
4.1.3. Подстановка пропущенных значений	123
4.1.4. API оценивателя scikit-learn	124
4.2. Работа с категориальными данными	125
4.2.1. Категориальное кодирование данных при помощи pandas	126
4.2.2. Сопоставление порядковых признаков	126
4.2.3. Кодирование меток классов	127
4.2.4. Позиционное кодирование номинальных признаков	128
4.2.5. Кодирование порядковых признаков	131
4.3. Разделение набора данных на обучающие и тестовые наборы	131
4.4. Приведение признаков к одному масштабу	134
4.5. Выбор значимых признаков	136
4.5.1. Регуляризация L1 и L2 как штраф за сложность модели	136
4.5.2. Геометрическая интерпретация регуляризации L2	137
4.5.3. Разреженные решения с регуляризацией L1	139
4.5.4. Последовательные алгоритмы отбора признаков.....	142

4.6. Оценка важности признаков с помощью случайных лесов	147
4.7. Заключение.....	150
Глава 5. Сжатие данных путем уменьшения размерности.....	151
5.1. Уменьшение размерности без учителя с помощью метода главных компонент	151
5.1.1. Основные этапы метода главных компонент.....	152
5.1.2. Пошаговый процесс извлечения основных компонент	154
5.1.3. Общая и объясненная дисперсия	156
5.1.4. Преобразование признаков	157
5.1.5. Анализ основных компонент в scikit-learn.....	160
5.1.6. Оценка вклада признаков.....	163
5.2. Сжатие данных с учителем с помощью линейного дискриминантного анализа	165
5.2.1. Сравнение методов PCA и LDA.....	165
5.2.2. Как устроен алгоритм LDA?	166
5.2.3. Вычисление матриц разброса	167
5.2.4. Выбор линейных дискриминантов для нового подпространства признаков	169
5.2.5. Проектирование точек данных на новое функциональное пространство.....	171
5.2.6. Реализация LDA при помощи scikit-learn	171
5.3. Нелинейное уменьшение размерности и визуализация.....	173
5.3.1. Зачем рассматривать нелинейное уменьшение размерности?	174
5.3.2. Визуализация данных с помощью алгоритма t-SNE.....	175
5.4. Заключение.....	178
Глава 6. Современные методы оценки моделей и настройки гиперпараметров.....	179
6.1. Оптимизация рабочих процессов с помощью конвейеров	179
6.1.1. Загрузка набора данных по раку молочной железы в Висконсине.....	179
6.1.2. Объединение преобразователей и оценивателей в конвейер	181
6.2. Использование k-кратной перекрестной проверки для оценки производительности модели	183
6.2.1. Перекрестная проверка с отложенными данными	183
6.2.2. k-кратная перекрестная проверка	184
6.3. Алгоритмы отладки с использованием кривых обучения и валидации.....	188
6.3.1. Диагностика смещения и дисперсии с помощью кривых обучения.....	188
6.3.2. Устранение переобучения и недообучения с помощью кривых валидации.....	191
6.4. Точная настройка моделей с помощью поиска по сетке	193
6.4.1. Настройка гиперпараметров с помощью поиска по сетке	193
6.4.2. Изучение обширных конфигураций гиперпараметров с помощью рандомизированного поиска	195
6.4.3. Поиск гиперпараметров методом последовательного деления пополам	197
6.4.4. Выбор алгоритма методом вложенной перекрестной проверки	199
6.5. Обзор различных показателей оценки эффективности.....	201
6.5.1. Чтение матрицы несоответствий	201
6.5.2. Оптимизация правильности и полноты модели классификации	203
6.5.3. Построение рабочей характеристики приемника.....	205
6.5.4. Метрики оценки многоклассовой классификации	208
6.5.5. Борьба с дисбалансом классов	209
6.6. Заключение.....	211
Глава 7. Объединение различных моделей для ансамблевого обучения	212
7.1. Обучение ансамблей моделей	212
7.2. Объединение классификаторов по методу большинства голосов	216
7.2.1. Реализация простого мажоритарного классификатора.....	216
7.2.2. Использование принципа мажоритарного голосования для прогнозирования	220
7.2.3. Оценка и настройка ансамблевого классификатора	223

7.3. Бэггинг: создание ансамбля классификаторов из исходного набора данных	228
7.3.1. Коротко о бэггинге	229
7.3.2. Применение бэггинга для классификации экземпляров набора данных Wine	230
7.4. Использование «слабых учеников» в механизме аддитивного бустинга	234
7.4.1. Как работает аддитивный бустинг?	234
7.4.2. Применение AdaBoost с помощью scikit-learn	238
7.5. Градиентный бустинг: обучение ансамбля на основе градиентов потерь	241
7.5.1. Сравнение AdaBoost с градиентным бустингом	241
7.5.2. Описание обобщенного алгоритма градиентного бустинга	242
7.5.3. Применение алгоритма градиентного бустинга для классификации	243
7.5.4. Иллюстрация применения градиентного бустинга для классификации	245
7.5.5. Использование XGBoost	247
7.6. Заключение	249
Глава 8. Применение машинного обучения для смыслового анализа текста	250
8.1. Подготовка набора данных с обзорами фильмов на IMDb	250
8.1.1. Получение набора данных с обзорами фильмов	251
8.1.2. Преобразование набора данных в более удобный формат	251
8.2. Знакомство с моделью мешка слов	253
8.2.1. Преобразование слов в векторы признаков	253
8.2.2. Оценка релевантности слов с помощью частоты термина и обратной частоты документа	255
8.2.3. Очистка текстовых данных	257
8.2.4. Получение токенов из документов	259
8.3. Обучение модели логистической регрессии для классификации документов	260
8.4. Работа с большими данными: онлайн-алгоритмы и внешнее обучение	263
8.5. Моделирование тем с использованием скрытого распределения Дирихле	266
8.5.1. Разбор текстовых документов с помощью LDA	267
8.5.2. Реализация LDA в библиотеке scikit-learn	267
8.6. Заключение	270
Глава 9. Прогнозирование непрерывных целевых переменных с помощью регрессионного анализа	272
9.1. Знакомство с линейной регрессией	272
9.1.1. Простая линейная регрессия	273
9.1.2. Множественная линейная регрессия	273
9.2. Изучение набора данных Ames Housing	274
9.2.1. Загрузка набора данных Ames Housing в DataFrame	275
9.2.2. Визуализация важных характеристик набора данных	277
9.2.3. Просмотр отношений с помощью матрицы корреляции	278
9.3. Реализация обычной модели линейной регрессии методом наименьших квадратов	280
9.3.1. Нахождение регрессии для параметров регрессии с градиентным спуском	281
9.3.2. Оценка коэффициента регрессионной модели с помощью scikit-learn	284
9.4. Обучение устойчивой регрессионной модели с использованием RANSAC	286
9.5. Оценка производительности моделей линейной регрессии	289
9.6. Использование методов регуляризации для регрессии	293
9.7. Переход от прямой линии к кривой: полиномиальная регрессия	294
9.7.1. Добавление полиномиальных членов с помощью scikit-learn	295
9.7.2. Моделирование нелинейных отношений в наборе данных Ames Housing	296
9.8. Моделирование нелинейных отношений с использованием случайных лесов	299
9.8.1. Регрессия на основе алгоритма дерева решений	299
9.8.2. Регрессия на основе случайного леса	301
9.9. Заключение	303

Глава 10. Работа с неразмечеными данными: кластерный анализ	304
10.1. Группировка объектов по сходству с использованием k-средних	304
10.1.1. Кластеризация методом k-средних с использованием scikit-learn	304
10.1.2. Более разумный способ размещения начальных центроидов: алгоритм k-средних++	309
10.1.3. Жесткая и мягкая кластеризация	310
10.1.4. Использование метода локтя для нахождения оптимального количества кластеров	312
10.1.5. Количественная оценка качества кластеризации с помощью силуэтных графиков	313
10.2. Организация кластеров в виде иерархического дерева	318
10.2.1. Группировка кластеров снизу вверх	318
10.2.2. Выполнение иерархической кластеризации с матрицей расстояний	319
10.2.3. Прикрепление дендрограмм к тепловой карте	323
10.2.4. Агglomerативная кластеризация с помощью scikit-learn	324
10.3. Обнаружение областей высокой плотности с помощью DBSCAN	325
10.4. Заключение	330
Глава 11. Построение многослойной искусственной нейронной сети с нуля	331
11.1. Моделирование сложных функций с помощью искусственных нейросетей	331
11.1.1. Вкратце об однослойной нейронной сети	332
11.1.2. Знакомство с архитектурой многослойной нейронной сети	334
11.1.3. Активация нейронной сети прямого распространения	336
11.2. Классификация рукописных цифр	338
11.2.1. Получение и подготовка набора данных MNIST	339
11.2.2. Реализация многослойного персептрона	342
11.2.3. Обучающий цикл нейронной сети	346
11.2.4. Оценка производительности нейронной сети	351
11.3. Обучение искусственной нейронной сети	354
11.3.1. Вычисление функции потерь	354
11.3.2. Подробнее о механизме обратного распространения	356
11.3.3. Обучение нейронных сетей с помощью обратного распространения	358
11.4. О сходимости в нейронных сетях	361
11.5. Несколько заключительных слов о реализации нейронной сети	362
11.6. Заключение	363
Глава 12. Глубокое обучение нейронных сетей на основе PyTorch	364
12.1. PyTorch и производительность обучения	364
12.1.1. Проблемы с быстродействием	364
12.1.2. Что такое PyTorch?	366
12.1.3. Как мы будем изучать PyTorch?	367
12.2. Первые шаги с PyTorch	367
12.2.1. Установка PyTorch	367
12.2.2. Создание тензоров в PyTorch	369
12.2.3. Управление типом данных и формой тензора	369
12.2.4. Применение к тензорам математических операций	370
12.2.5. Разделение, стекирование и конкатенация тензоров	371
12.3. Построение конвейеров ввода в PyTorch	373
12.3.1. Создание объекта <i>DataLoader</i> из существующих тензоров	373
12.3.2. Объединение двух тензоров в совместный набор данных	374
12.3.3. Перемешивание, группировка и повторение	375
12.3.4. Создание набора данных из файлов на локальном диске	377
12.3.5. Получение доступных наборов данных из библиотеки <i>torchvision.datasets</i>	380
12.4. Построение нейросетевой модели в PyTorch	384
12.4.1. Модуль нейронной сети PyTorch (<i>torch.nn</i>)	384
12.4.2. Построение модели линейной регрессии	384

12.4.3. Обучение модели с помощью модулей <i>torch.nn</i> и <i>torch.optim</i>	388
12.4.4. Построение многослойного персептрана для классификации цветков в наборе данных <i>Iris</i>	389
12.4.5. Оценка обученной модели на тестовом наборе данных	391
12.4.6. Сохранение и повторная загрузка обученной модели	392
12.5. Выбор функций активации для многослойных нейронных сетей	393
12.5.1. Несколько слов о логистической функции	393
12.5.2. Оценка вероятностей классов в мультиклассовой классификации с помощью функции <i>softmax</i>	395
12.5.3. Расширение выходного спектра с использованием гиперболического тангенса	396
12.5.4. Спрямленная линейная активация	398
12.6. Заключение	399
Глава 13. Углубленное знакомство с PyTorch.....	400
13.1. Основные возможности PyTorch.....	401
13.2. Граф вычислений PyTorch.....	401
13.2.1. Вкратце о графе вычислений	401
13.2.2. Создание графа в PyTorch	402
13.3. Тензорные объекты PyTorch для хранения и обновления параметров модели	403
13.4. Вычисление градиентов с помощью автоматического дифференцирования	405
13.4.1. Вычисление градиентов потерь по обучаемым переменным	405
13.4.2. Как работает автоматическое дифференцирование?	406
13.4.3. Состязательные примеры	407
13.5. Упрощение реализации популярных архитектур с помощью модуля <i>torch.nn</i>	407
13.5.1. Реализация моделей на основе <i>nn.Sequential</i>	407
13.5.2. Выбор функции потерь	409
13.5.3. Решение задачи классификации XOR	409
13.5.4. Более гибкое построение моделей с помощью <i>nn.Module</i>	414
13.5.5. Создание в PyTorch пользовательских слоев	416
13.6. Проект № 1: прогнозирование расхода топлива автомобиля	420
13.6.1. Работа со столбцами признаков	420
13.6.2. Обучение регрессионной модели DNN	423
13.7. Проект № 2: классификация рукописных цифр из набора MNIST	425
13.8. API PyTorch более высокого уровня: краткое введение в PyTorch-Lightning	428
13.8.1. Настройка модели PyTorch Lightning	428
13.8.2. Настройка загрузчиков данных для Lightning	431
13.8.3. Обучение модели с помощью класса PyTorch Lightning Trainer	432
13.8.4. Оценка модели с помощью TensorBoard	433
13.9. Заключение	436
Глава 14. Классификация изображений с помощью глубоких сверточных нейронных сетей.....	437
14.1. Функциональные блоки CNN	437
14.1.1. Устройство CNN и понятие иерархии признаков	438
14.1.2. Выполнение дискретных сверток	439
Дискретные свертки в одном измерении	440
Заполнение входных данных для управления размером выходных карт признаков	442
Определение размера вывода свертки	443
Вычисление дискретной двумерной свертки	445
14.1.3. Слои подвыборки	448
14.2. Практическая реализация CNN	449
14.2.1. Работа с несколькими входными или цветовыми каналами	450
14.2.2. Регуляризация L2 и прореживание	452
14.2.3. Функции потерь для задач классификации	456

14.3. Реализация глубокой CNN с использованием PyTorch	458
14.3.1. Архитектура многослойной CNN	458
14.3.2. Загрузка и предварительная обработка данных	459
14.3.3. Реализация CNN с использованием модуля torch.nn	460
Настройка слоев CNN в PyTorch.....	460
Создание CNN в PyTorch.....	461
14.4. Классификация улыбающихся лиц с помощью CNN	466
14.4.1. Загрузка набора данных CelebA.....	466
14.4.2. Преобразование изображений и дополнение данных.....	467
14.4.3. Обучение классификатора улыбки на основе CNN	473
14.5. Заключение.....	479

Глава 15. Моделирование последовательных данных с помощью

рекуррентных нейронных сетей	480
15.1. Знакомство с понятием последовательных данных	480
15.1.1. Моделирование последовательных данных: порядок имеет значение	481
15.1.2. Данные временных рядов — особый тип последовательных данных.....	481
15.1.3. Способы представления последовательностей	481
15.1.4. Различные категории моделирования последовательности	482
15.2. RNN для моделирования последовательностей.....	484
15.2.1. Определение потока данных в RNN	484
15.2.2. Вычисление активаций в RNN	486
15.2.3. Скрытая и выходная рекуррентность.....	488
15.2.4. Проблемы изучения дальних взаимодействий	491
15.2.5. Ячейки долгой краткосрочной памяти	492
15.3. Реализация RNN для моделирования последовательностей в PyTorch	494
15.3.1. Проект № 1: предсказание эмоциональной окраски отзывов на фильмы IMDb	494
Подготовка данных отзывов на фильмы	495
Слои встраивания для кодирования предложений	499
Построение модели RNN	501
Построение модели RNN для задачи анализа тональности	502
Подробнее о двунаправленной RNN	504
15.3.2. Проект № 2: моделирование языка на уровне символов в PyTorch	506
Предварительная обработка набора данных	507
Построение модели RNN символьного уровня	511
Этап оценки: создание новых отрывков текста	513
15.4. Заключение.....	517

Глава 16. Трансформеры: улучшение обработки естественного языка с помощью

механизмов внимания.....	518
16.1. Добавляем к RNN механизм внимания	519
16.1.1. Как внимание помогает RNN извлекать информацию?	519
16.1.2. Оригинальный механизм внимания для RNN	520
16.1.3. Обработка входных данных с использованием двунаправленной RNN.....	520
16.1.4. Генерация выходных данных из векторов контекста	522
16.1.5. Вычисление весов внимания	522
16.2. Знакомство с механизмом самовнимания	523
16.2.1. Начнем с базовой формы самовнимания.....	524
16.2.2. Параметризация механизма самовнимания: взвешенное скалярное произведение	527
16.3. Внимание — это все, что нам нужно: знакомство с архитектурой Transformer	530
16.3.1. Кодирование контекстных встраиваний с помощью многоголового внимания	531
16.3.2. Обучение языковой модели: декодер и маскированное многоголовое внимание	535
16.3.3. Детали реализации: позиционное кодирование и нормализация слоев	536

16.4. Построение больших языковых моделей с использованием неразмеченные данных	538
16.4.1. Предварительное обучение и тонкая настройка моделей трансформеров	539
16.4.2. Использование неразмеченные данных с помощью GPT	541
16.4.3. Использование GPT-2 для создания нового текста	545
16.4.4. Двунаправленное предварительное обучение модели BERT	547
16.4.5. Лучшее из двух миров: BART	550
16.5. Тонкая настройка модели BERT в PyTorch	553
16.5.1. Загрузка набора данных обзора фильмов IMDb	554
16.5.2. Токенизация набора данных	555
16.5.3. Загрузка и тонкая настройка предварительно обученной модели BERT	557
16.5.4. Удобная тонкая настройка трансформера с помощью API Trainer	560
16.6. Заключение	564
Глава 17. Генеративно-состязательные сети и синтез новых данных	566
17.1. Знакомство с генеративно-состязательными сетями	566
17.1.1. Начнем с автокодировщиков	567
17.1.2. Генеративные модели для синтеза новых данных	569
17.1.3. Генерация новых экземпляров данных с помощью GAN	570
17.1.4. Функции потерь сетей генератора и дискриминатора в модели GAN	572
17.2. Строим собственную GAN с нуля	574
17.2.1. Обучение моделей GAN в Google Colab	574
17.2.2. Реализация сетей генератора и дискриминатора	577
17.2.3. Определение набора обучающих данных	580
17.2.4. Обучение модели GAN	582
17.3. Улучшение качества синтезированных изображений с помощью сверточной GAN и метрики Вассерштейна	588
17.3.1. Транспонированная свертка	588
17.3.2. Пакетная нормализация	590
17.3.3. Реализация генератора и дискриминатора	592
17.3.4. Меры несходства между двумя распределениями	598
17.3.5. Практическое применение ЕМ-расстояния для построения GAN	602
17.3.6. Штраф за градиент	603
17.3.7. Реализация WGAN-GP для обучения модели DCGAN	603
17.3.8. Схлопывание мод распределения	606
17.4. Другие применения GAN	608
17.5. Заключение	609
Глава 18. Графовые нейронные сети: выявление зависимостей в структурированных графовых данных	610
18.1. Введение в графовые данные	611
18.1.1. Неориентированные графы	611
18.1.2. Ориентированные графы	612
18.1.3. Помеченные графы	612
18.1.4. Представление молекул в виде графов	613
18.2. Что такое графовая свертка?	614
18.2.1. Причина использования графовых сверток	614
18.2.2. Реализация базовой свертки графа	616
18.3. Построение GNN в PyTorch с нуля	620
18.3.1. Определение модели <i>NodeNetwork</i>	620
18.3.2. Реализация слоя графовой свертки <i>NodeNetwork</i>	622
18.3.3. Добавочный слой глобального объединения для работы с графиками разного размера	623
18.3.4. Подготовка загрузчика данных <i>DataLoader</i>	625
18.3.5. Использование <i>NodeNetwork</i> для прогнозирования	627

18.4. Построение GNN с использованием геометрической библиотеки PyTorch	628
18.5. Другие слои GNN и новейшие разработки	634
18.5.1. Спектральные графовые свертки	634
18.5.2. Объединение	635
18.5.3. Нормализация	637
18.5.4. Дополнительная литература по графовым нейронным сетям	639
18.6. Заключение	640
Глава 19. Обучение с подкреплением для принятия решений в сложных условиях	641
19.1. Введение в обучение на собственном опыте	642
19.1.1. Ключевые идеи обучения с подкреплением	642
19.1.2. Определение интерфейса агент-среда в системе обучения с подкреплением	643
19.2. Теоретические основы RL	645
19.2.1. Марковские процессы принятия решений	645
Математическая основа марковских процессов принятия решений	645
Визуальное представление марковского процесса	647
19.2.2. Эпизодические и непрерывные задачи	648
19.2.3. Терминология RL: отдача, стратегия и функция ценности	648
Отдача	648
Стратегия	650
Функция ценности	650
19.2.4. Уравнение Беллмана в динамическом программировании	652
19.3. Алгоритмы обучения с подкреплением	653
19.3.1. Динамическое программирование	653
Оценка стратегии: прогнозирование функции ценности с помощью динамического программирования	654
Улучшение стратегии с помощью известной функции ценности	655
Итерация стратегии	655
Итерация ценности	655
19.3.2. Обучение с подкреплением по методу Монте-Карло	656
Нахождение функции ценности состояния с использованием МС	656
Нахождение функции ценности действия с использованием МС	657
Поиск оптимальной стратегии с помощью МС-управления	657
Улучшение стратегии: вычисление жадной стратегии на основе функции ценности действия	657
19.3.3. Обучение на временных разностях	658
Прогнозирование ценности с TD	658
Алгоритм on-policy TD (SARSA)	660
Алгоритм off-policy TD (Q-обучение)	660
19.4. Реализация первого алгоритма RL	661
19.4.1. Знакомство с набором инструментов OpenAI Gym	661
Работа с готовыми средами в OpenAI Gym	661
Пример клетчатого мира	663
Реализация среды клетчатого мира в OpenAI Gym	664
19.4.2. Решение задачи клетчатого мира с помощью Q-обучения	670
19.5. Обзор глубокого Q-обучения	673
19.5.1. Применение алгоритма Q-обучения в моделях DQN	674
Глобальная память	674
Определение целей для расчета потерь	676
19.5.2. Реализация алгоритма глубокого Q-обучения	677
19.6. Краткое содержание главы и книги	680
Предметный указатель	684

Вступительное слово

В последние годы методы машинного обучения с их способностью анализировать огромные объемы данных и автоматизировать принятие решений нашли широкое применение в здравоохранении, робототехнике, биологии, физике, производстве потребительских товаров, интернет-услугах и многих других отраслях.

Наука обычно совершает гигантские рывки вперед благодаря сочетанию мощных идей и отличных инструментов. Машинное обучение не является исключением. Успех методов обучения на основе данных опирается на гениальные идеи тысяч талантливых исследователей за 60-летнюю историю этой области. Но недавний взрыв популярности машинного обучения стал возможен благодаря появлению масштабируемых и общедоступных аппаратных и программных решений. Экосистема замечательных библиотек для числовых вычислений, анализа данных и машинного обучения на основе Python, таких как NumPy и scikit-learn, получила широкое распространение в научных исследованиях и промышленности. Во многом вследствие этого Python стал самым популярным языком программирования.

Хорошим примером слияния теории и практики могут служить удивительные достижения в области компьютерного зрения, обработки естественного языка и других задач, вызванные недавним появлением методов глубокого обучения. Теория нейронных сетей, разработанная в течение последних четырех десятилетий, начала замечательно работать в сочетании с графическими процессорами и высокооптимизированными вычислительными процедурами.

Занимаясь разработкой PyTorch в течение последних пяти лет, мы стремились предоставить исследователям наиболее гибкий инструмент для реализации алгоритмов глубокого обучения, взяв на себя труд решить базовые инженерные задачи. Мы воспользовались превосходной экосистемой Python. В свою очередь, нам посчастливилось увидеть, как сообщество очень талантливых людей создает передовые модели глубокого обучения в различных областях на основе PyTorch. Среди них были и авторы этой книги.

Я знаю Себастьяна как активного члена этого сплоченного сообщества, уже несколько лет. Он обладает непревзойденным талантом доходчиво объяснять новые понятия и делать сложное доступным. Себастьян участвовал в разработке многих широко распространенных пакетов и библиотек для машинного обучения и является автором десятков отличных руководств по глубокому обучению и визуализации данных.

Для применения машинного обучения на практике необходимо овладеть как идеями, так и инструментами. Путь, на который вы собираетесь ступить, выглядит на первый

взгляд устрашающее — ведь вам предстоит сделать очень многое, начиная с изучения теоретических основ и заканчивая выяснением того, какие программные пакеты нужно установить.

К счастью, книга, которую вы держите в руках, прекрасно сочетает в себе изложение теории машинного обучения и практические примеры, которые помогут вам пройти этот путь. Вас ждет восхитительное путешествие от азов обучения на основе данных до знакомства с самыми передовыми архитектурами глубокого обучения. В каждой главе вы найдете примеры кода, демонстрирующие применение изученных методов на практике.

Когда в 2015 году вышло первое издание этой книги, оно установило очень высокую планку качества для книг в области машинного обучения и программирования на Python. Но авторы не остановились на достигнутом. С каждым изданием Себастьян и его команда обновляли и совершенствовали материал по мере того, как революция глубокого обучения охватывала все новые отрасли. В этом — обновленном — издании книги про PyTorch вы найдете новые главы об архитектуре Transformer и графовых нейронных сетях. Эти подходы находятся на переднем крае глубокого обучения и за последние два года совершили прорыв в области понимания текста и молекулярных структур. Вы сможете применить их на практике, используя новые, но очень популярные программные пакеты, такие как Hugging Face, PyTorch Lightning и PyTorch Geometric.

Благодаря сочетанию передовых научных знаний и прикладного опыта авторы этой книги достигли превосходного баланса теории и практики. Себастьян Рашка и Вахид Мирджалили опираются на свой опыт в исследованиях глубокого обучения для компьютерного зрения и вычислительной биологии. Юси (Хэйден) Лю обладает опытом применения методов машинного обучения для прогнозирования событий, рекомендательных систем и других задач. Все авторы искренне увлекаются преподаванием и смогли организовать плавный переход от простого материала к сложному.

Я уверен, что эта книга станет для вас бесценным источником теоретических знаний в области машинного обучения и сокровищницей практических идей. Я надеюсь, что она вдохновит вас на новые потрясающие достижения, независимо от того, какие задачи вы решаете.

*Дмитро Джулгаков,
ведущий разработчик PyTorch Core*

Об авторах

Доктор **Себастьян Рашка** преподает на кафедре статистики Висконсинского университета в Мадисоне и специализируется на машинном и глубоком обучении. Его последние исследования сосредоточены на обобщенных задачах — таких как многошаговое обучение для работы с ограниченными данными и разработка глубоких нейронных сетей для порядковых целей. Себастьян также является активным участником проектов с открытым исходным кодом, и в своей новой роли ведущего преподавателя ИИ в Grid.ai он намерен продолжать заниматься любимым делом и помогать людям осваивать машинное обучение и ИИ.

Большое спасибо Цзитянь Чжоу и Бену Кауфману, с которыми я имел удовольствие работать над новыми главами о трансформерах и графовых нейронных сетях. Я также очень благодарен Хэйдену и Вахиду за помощь — без вас я бы не написал эту книгу. Наконец, я хочу поблагодарить Андреа Паницу, Тони Гиттера и Адама Бельски за полезные советы и замечания при обсуждении рукописи.

Юси (Хэйден) Лю — инженер-программист, специалист по машинному обучению в Google, а также в различных областях, связанных с обработкой данных, и автор серии книг по машинному обучению. Его первая книга «Python Machine Learning By Example» заняла первое место в своей категории на Amazon в 2017 и 2018 годах и была переведена на многие языки¹. Он также написал такие популярные книги, как «R Deep Learning Projects», «Hands-On Deep Learning Architectures with Python» и «PyTorch 1.x Reinforcement Learning Cookbook»².

Я хочу поблагодарить всех замечательных людей, с которыми я работал, особенно моих соавторов, редакторов в издательстве Packt и рецензентов. Без них эту книгу было бы труднее читать и применять для решения реальных задач. Наконец, я хочу поблагодарить всех читателей за их поддержку, которая побудила меня написать новую книгу, посвященную применению PyTorch в машинном обучении.

Доктор **Вахид Мирджалили** — исследователь глубокого обучения, специализирующийся на приложениях компьютерного зрения. Вахид получил докторскую степень в области машиностроения и компьютерных наук в Мичиганском государственном

¹ На русском языке она не издавалась. — Прим. пер.

² В 2020 г. в издательстве «ДМК Пресс» вышел перевод этой книги на русский язык под названием «Обучение с подкреплением на PyTorch: сборник рецептов». — Прим.пер.

университете. Во время подготовки докторской диссертации он разработал новые алгоритмы компьютерного зрения для решения прикладных задач и опубликовал несколько исследовательских статей, которые широко цитируют в сообществе компьютерного зрения.

Соавторы

Бенджамин Кауфман — соискатель степени доктора философии в области биомедицинских наук о данных в Висконсинском университете в Мадисоне. Его исследования сосредоточены на разработке и применении методов машинного обучения для создания лекарств. Работы Кауфмана в этой области дают более глубокое понимание графовых нейронных сетей.

Цзитянь Чжао — аспирант Висконсинского университета в Мадисоне, где она заинтересовалась большими языковыми моделями. Она увлечена разработкой как реальных приложений, так и теоретической базы глубокого обучения.

Хочу поблагодарить родителей за поддержку. Они побуждали меня всегда следовать своей мечте и быть хорошим человеком.

Рецензент

Роман Тезиков — промышленный инженер-исследователь и энтузиаст глубокого обучения с более чем четырехлетним опытом работы в области передового компьютерного зрения, NLP и MLOps. Как один из создателей сообщества ML-REPA, он организовал несколько семинаров и встреч, посвященных воспроизведимости машинного обучения и автоматизации конвейера обучения моделей. Одно из направлений его текущей работы связано с использованием компьютерного зрения в индустрии моды. Роман также был основным разработчиком Catalyst — платформы PyTorch для ускоренного глубокого обучения.

Предисловие

Благодаря новостям и социальным сетям вы, вероятно, знакомы с тем фактом, что машинное обучение стало одной из самых захватывающих технологий нашего времени. Крупные компании, такие как Microsoft, Google, Apple, Amazon, IBM и многие другие, неспроста вкладывают огромные средства в исследования и разработку приложений машинного обучения. И хотя может показаться, что машинное обучение стало лишь модным термином последних лет, это вовсе не пустая шумиха. Захватывающие исследования в области машинного обучения открыли доступ к новым возможностям, которые уже стали незаменимыми в нашей повседневной жизни. Интеллектуальные голосовые помощники в смартфонах, рекомендации покупок в интернет-магазинах, предотвращение мошенничества с кредитными картами, фильтрация спама в почтовых ящиках, обнаружение и диагностика медицинских заболеваний — области применения машинного обучения можно перечислять очень долго.

Если вы хотите стать практиком в области машинного обучения, найти новые решения своих задач или даже задумались о карьере исследователя, то эта книга — для вас! Однако новичку теория, заложенная в основу машинного обучения, может показаться довольно сложной. Тем не менее в последние годы опубликовано много книг, которые помогут вам начать работу с машинным обучением.

Знакомство с практическими примерами кода и работа с приложениями — отличный способ погрузиться в область машинного обучения. Конкретные примеры иллюстрируют более широкие понятия и позволяют моментально применить на деле изученный материал. Однако помните, что с большой властью приходит и большая ответственность! В дополнение к практическому опыту машинного обучения с использованием Python и описанию библиотек на основе Python эта книга также знакомит читателя с математическими основами алгоритмов машинного обучения, что необходимо для успешного использования машинного обучения. В этом состоит главное ее отличие от чисто прикладной книги: здесь рассмотрены важные теоретические аспекты машинного обучения, предложены интуитивно понятные, но информативные объяснения того, как работают алгоритмы машинного обучения, как их использовать и, самое главное, как избежать наиболее распространенных ошибок.

С этой книгой в руках вы отправитесь в захватывающее путешествие в мир машинного обучения, последовательно познакомившись со всеми необходимыми темами. Если вы почувствуете, что нуждаетесь в дополнительных знаниях, то найдете в этой книге ссылки на множество полезных ресурсов, которые следует регулярно посещать, чтобы следить за важными достижениями в области машинного обучения.

Для кого эта книга?

Эта книга послужит идеальным учебником для тех, кто собирается применять машинное и глубокое обучение к широкому кругу задач и наборов данных. Если вы программист и хотите идти в ногу с передовыми технологиями, она определенно для вас. Кроме того, если вы студент или думаете о смене карьеры, она станет для вас и введением в мир машинного обучения, и подробным путеводителем по нему.

О чём рассказывает книга?

Глава 1. Как помочь компьютеру учиться на данных? — знакомит вас с основными областями, в которых машинное обучение применяется для решения различных задач. Кроме того, в ней представлены основные этапы типичного процесса построения модели машинного обучения. Знание этого процесса пригодится вам в следующих главах.

Глава 2. Простые алгоритмы машинного обучения для задач классификации — обращается к истокам машинного обучения и знакомит с классификаторами на основе бинарного персептрана и адаптивными линейными нейронами. Эта глава представляет собой краткое введение в основы классификации образов и посвящена взаимодействию алгоритмов оптимизации и машинного обучения.

Глава 3. Знакомство с классификаторами машинного обучения на основе scikit-learn — описывает основные алгоритмы машинного обучения для задач классификации и содержит практические примеры использования scikit-learn — одной из самых популярных и всеобъемлющих библиотек машинного обучения с открытым исходным кодом.

Глава 4. Предварительная обработка данных для создания качественных обучающих наборов — рассказывает о том, как справляться с наиболее распространенными недостатками необработанных наборов данных (такими как отсутствующие данные). В ней также представлены различные подходы к выявлению наиболее информативных признаков в наборах данных и продемонстрировано использование переменных разных типов в качестве корректных входных данных для алгоритмов машинного обучения.

Глава 5. Сжатие данных путем уменьшения размерности — содержит описание основных методов сокращения количества признаков в наборе данных при сохранении большей части полезной отличительной информации. В ней представлен стандартный подход к уменьшению размерности с помощью метода главных компонент в сравнении с методами управляемого и нелинейного преобразования.

Глава 6. Современные методы оценки моделей и настройки гиперпараметров — знакомит с рекомендациями и ограничениями при оценке качества предсказательных моделей. Кроме того, в ней обсуждаются различные показатели для измерения точности и быстродействия моделей и методы тонкой настройки алгоритмов машинного обучения.

Глава 7. Объединение различных моделей для ансамблевого обучения — знакомит читателей с различными способами эффективного объединения нескольких алгоритмов обучения. В ней показано, как создавать ансамбли экспертов, помогающие обходить недостатки отдельных обучаемых моделей, что позволяет получать более точные и надежные прогнозы.

Глава 8. Применение машинного обучения для смыслового анализа текста — рассказывает о принципах преобразования текстовых данных в понятные для алгоритмов машинного обучения представления, дающие возможность предсказывать настроение людей на основе написанного ими текста.

Глава 9. Прогнозирование непрерывных целевых переменных с помощью регрессионного анализа — представляет основные методы моделирования линейных взаимосвязей между целевыми переменными и переменными отклика для получения непрерывных прогнозов. Познакомив читателей с различными линейными моделями, она расскажет о полиномиальной регрессии и древовидных моделях.

Глава 10. Работа с неразмеченными данными: кластерный анализ — основное внимание уделяет другой области машинного обучения — обучению без учителя. В ней говорится о трех основных семействах алгоритмов кластеризации, которые позволяют найти группы объектов, имеющих определенную степень сходства.

Глава 11. Построение многослойной искусственной нейронной сети с нуля — раскрывает принцип оптимизации на основе градиента, представленный в главе 2, и описывает процесс создания мощных многослойных нейронных сетей с учетом популярного алгоритма обратного распространения ошибки.

Глава 12. Глубокое обучение нейронных сетей на основе PyTorch — опирается на знания, полученные в предыдущей главе, и содержит практическое руководство по более эффективному обучению нейронных сетей. В ней основное внимание уделяется PyTorch — библиотеке Python с открытым исходным кодом, позволяющей использовать несколько ядер современных графических процессоров и создавать глубокие нейронные сети из стандартных блоков с помощью удобного и гибкого API.

Глава 13. Углубленное знакомство с PyTorch — начинается с того места, на котором остановилась предыдущая глава, и знакомит с более продвинутыми возможностями PyTorch. PyTorch — чрезвычайно обширная и сложная библиотека, и в этой главе вы познакомитесь с такими понятиями, как динамические графы вычислений и автоматическое дифференцирование. Вы также узнаете, как использовать объектно-ориентированный API PyTorch для реализации сложных нейронных сетей и как PyTorch Lightning помогает свести к минимуму шаблонный код.

Глава 14. Классификация изображений с помощью глубоких сверточных нейронных сетей — представляет вниманию читателей сверточные нейронные сети (Convolutional Neural Networks, CNN). CNN — это особая разновидность архитектуры глубокой нейронной сети, которая особенно хорошо подходит для работы с изображениями. Благодаря эффективности, превосходящей традиционные подходы, CNN теперь широко применяются в компьютерном зрении для решения самых сложных и разнообразных задач распознавания изображений. В этой главе вы узнаете, как можно использовать сверточные слои в качестве мощных экстракторов признаков для классификации изображений.

Глава 15. Моделирование последовательных данных с помощью рекуррентных нейронных сетей — знакомит читателей с еще одной популярной архитектурой глубокой нейронной сети, которая особенно хорошо подходит для работы с текстом и иными типами последовательных данных, а также данными временных рядов. Глава начинается с описания рекуррентных нейронных сетей, предназначенных для прогнозирования эмоциональной окраски рецензий на фильмы, после чего в ней будет показано, как можно на-

учить рекуррентные сети извлекать информацию из книг, чтобы генерировать совершенно новый текст.

Глава 16. Трансформеры: улучшение обработки естественного языка с помощью механизмов внимания — посвящена новейшим тенденциям в обработке естественного языка и объясняет, как механизмы внимания помогают моделировать сложные отношения в длинных последовательностях. В частности, в этой главе говорится о знаменитой архитектуре Transformer и современных моделях-трансформерах — таких как BERT и GPT.

Глава 17. Генеративно-состязательные сети и синтез новых данных — раскрывает принцип состязательного обучения нейронных сетей, лежащий в основе создания новых, реалистично выглядящих изображений. Глава начинается с краткого введения в автокодировщики — особый тип архитектуры нейронной сети, который можно использовать для сжатия данных. Далее в главе показано, как декодирующую часть автокодировщика объединяют со второй нейронной сетью, которая обучается различать реальные и синтезированные изображения. Заставив две нейронные сети соревноваться друг с другом в состязательном подходе к обучению, вы создадите генеративную состязательную сеть, которая генерирует новые рукописные цифры.

Глава 18. Графовые нейронные сети: выявление зависимостей в структурированных графовых данных — предоставляет вам информацию, расширяющую рамки работы с наборами табличных данных, изображениями и текстом. Она рассказывает о нейронных сетях, работающих с данными, структурированными в виде графов, — такими как связи в социальных сетях и молекулы. Начав с введения в основы графовых сверток, глава продолжается руководством по созданию прогнозирующей модели для молекулярных данных.

Глава 19. Обучение с подкреплением для принятия решений в сложных условиях — охватывает подкатегорию машинного обучения, предназначенную для обучения роботов и других автономных систем. Глава начинается с введения в основы обучения с подкреплением (Reinforcement Learning, RL), знакомящего читателей с взаимодействием агента и среды, процессом вознаграждения в системах RL и концепцией обучения на основе опыта. Изучив основные категории RL, вы создадите и обучите агента, который сможет перемещаться по координатной сетке, используя алгоритм Q-обучения. В завершение в этой главе описан алгоритм глубокого Q-обучения, представляющий собой вариант Q-обучения, использующий глубокие нейронные сети.

Как получить максимальную отдачу от этой книги?

В идеальном случае читатель должен иметь навыки программирования на Python, чтобы разобраться в примерах кода, которые иллюстрируют применение различных алгоритмов и моделей. Желательно также хорошо знать общепринятые математические обозначения.

Обычного ноутбука или настольного компьютера вполне достаточно для выполнения большей части примеров кода из этой книги, а в первой главе мы приводим инструкции по настройке среды Python. В последующих главах будет рассказано о дополнительных

библиотеках и даны рекомендации по их установке, когда в этом возникнет необходимость.

Наличие в компьютере современного графического процессора ускорит выполнение кода, представленного в заключительных главах. Однако использование общедоступных облачных ресурсов, о которых мы также расскажем в этой книге, позволяет обойтись без дорогостоящего графического процессора.

Скачивание примеров кода

Все примеры кода, приведенные в книге, доступны для скачивания на GitHub по адресу: <https://github.com/rasbt/machine-learning-book>, а также (в виде файлового ZIP-архива) по ссылке: <https://tinyurl.com/mpfr785d>.

У авторов книги есть и другие пакеты кода в богатом каталоге книг и видео, доступном по адресу: <https://github.com/PacktPublishing/>. Посмотрите и их!

Присоединяйтесь к сообществу Discord, где вы сможете найти единомышленников и обсуждать вопросы машинного обучения вместе с более чем двумя тысячами его участников. Его адрес: <https://packt.link/MLwPyTorch>.



Хотя для интерактивного выполнения кода мы рекомендуем использовать Jupyter Notebook, все примеры кода из книги представлены как в виде скриптов Python (например, файл ch02/ch02.py), так и в формате Jupyter Notebook (например, файл ch02/ch02.ipynb). Кроме того, рекомендуется также просматривать файл README.md, сопровождающий примеры кода каждой главы, для получения дополнительной информации и сведений об исправлениях и обновлениях.

Скачивание цветных изображений

Читатели могут скачать PDF-файл с цветными изображениями снимков экрана, графиков и диаграмм, приведенными в этой книге, по адресу: https://static.packt-cdn.com/downloads/9781801819312_ColorImages. Кроме того, цветные изображения с более низким разрешением размещены в соответствующих папках с примерами кода из этой книги.

Примечание от русской редакции

Указанная в этом разделе ссылка на момент подготовки русского издания не работала. PDF-файл с полным комплектом цветных иллюстраций из книги, снабженных переведенными на русский язык подписями, включен в файловый архив с примерами кода, который можно скачать по ссылке: <https://tinyurl.com/mpfr785d> или по приведенному QR-коду



Соглашения и условные обозначения

В этой книге используется ряд соглашений об обозначениях.

- ◆ Фрагменты и отдельные слова кода в тексте показаны монодириным шрифтом — например: «Установленные пакеты можно обновить с помощью флага `--upgrade`».
- ◆ Блоки кода обозначены так:

```
def __init__(self, eta=0.01, n_iter=50, random_state=1):  
    self.eta = eta  
    self.n_iter = n_iter  
    self.random_state = random_state
```

- ◆ Любой ввод в интерпретаторе Python обозначен следующим образом (обратите внимание на символы `>>>`). Ожидаемый результат (вывод) будет показан без символов `>>>:`

```
>>> v1 = np.array([1, 2, 3])  
>>> v2 = 0.5 * v1  
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *  
... np.linalg.norm(v2)))  
0.0
```

- ◆ Любой ввод или вывод командной строки обозначен следующим образом:

```
pip install gym==0.20
```

- ◆ Новые термины и важные слова выделены в тексте *курсивом*.
- ◆ Надписи, которые вы видите на экране — например, в меню или диалоговых окнах, показаны в тексте так: «Нажатие кнопки **Next** (Далее) приводит к переходу на следующий экран».
- ◆ Имена файлов, каталоги и пути к ним обозначены рубленым шрифтом — например, так: D:\Machine Learning with PyTorch and Scikit-Learn\ eBook\.



Этим символом обозначены предупреждения или важные примечания



Этим символом обозначены советы и рекомендации

1

Как помочь компьютеру учиться на данных?

https://t.me/it_boooks/2

На мой взгляд, *машинное обучение* — наука об алгоритмах, которые анализируют данные, — это самая захватывающая область информатики! Мы живем в эпоху, когда огромные объемы данных поступают к нам отовсюду, и, используя самообучающиеся алгоритмы машинного обучения, мы можем превратить эти данные в знания. В последние годы были разработаны различные удобные библиотеки с открытым исходным кодом, и сейчас, пожалуй, самое подходящее время, чтобы заняться освоением машинного обучения и научиться использовать мощные алгоритмы для выявления закономерностей в данных и прогнозирования будущих событий.

В этой главе вы узнаете об основных концепциях и различных типах машинного обучения. Мы начнем со знакомства с основными терминами и заложим прочную теоретическую основу для успешного использования методов машинного обучения при решении практических задач.

Таким образом, здесь будут рассмотрены следующие темы:

- ◆ общие концепции машинного обучения;
- ◆ три типа обучения и основная терминология;
- ◆ компоненты для успешного построения систем машинного обучения;
- ◆ установка и настройка Python для анализа данных и машинного обучения.

1.1. Интеллектуальные машины для преобразования данных в знания

В наш век современных технологий есть один ресурс, которого у нас в изобилии: огромное количество структурированных и неструктурированных данных. Во второй половине XX века машинное обучение обзавелось алгоритмами самообучения, которые извлекают из данных знания для прогнозирования, и стало частью более обширной отрасли *искусственного интеллекта* (ИИ).

Машинное обучение избавляет людей от необходимости вручную выводить правила и строить модели на основе анализа больших объемов данных, предлагая более эффективную альтернативу в виде извлечения знаний из данных для постепенного повыше-

ния производительности¹ прогностических моделей и принятия решений на основе данных.

Сегодня машинное обучение — не только предмет исследований в области компьютерных наук, оно также играет важную роль в нашей повседневной жизни. Благодаря машинному обучению, у нас есть надежные фильтры спама в электронной почте, удобные приложения для распознавания текста и голоса, качественные поисковые системы, рекомендации по просмотру развлекательных фильмов, безопасные мобильные платежи, расчет времени доставки еды и многое другое. Надеюсь, скоро мы добавим в этот список безопасные и комфортные беспилотные автомобили. Кроме того, машинное обучение добилось заметного прогресса в области медицины — например, исследователи продемонстрировали, что модели глубокого обучения могут обнаруживать рак кожи с точностью, близкой к показателям опытного эксперта-человека². Еще одну важную веху недавно миновали исследователи из DeepMind, которые использовали глубокое обучение для предсказания трехмерных белковых структур, значительно превзойдя подходы, основанные на традиционной физике³. И хотя точное предсказание трехмерной структуры белка играет важную роль в биологических и фармацевтических исследованиях, в последнее время машинное обучение нашло множество других важных применений в здравоохранении. Например, исследователи разработали системы для прогнозирования потребности пациентов с COVID-19 в кислороде за четыре дня, чтобы помочь больницам спланировать распределение ресурсов⁴. Одной из самых больших и критических проблем нашего времени является изменение климата. И сейчас множество усилий направлено на разработку интеллектуальных систем для борьбы с глобальным потеплением⁵. Одним из многих подходов к решению проблемы изменения климата является развивающаяся область точного земледелия. Здесь исследователи стремятся разработать системы машинного обучения на основе компьютерного зрения, чтобы оптимизировать расход ресурсов и свести к минимуму использование удобрений.

1.2. Три типа машинного обучения

В этом разделе мы рассмотрим три типа машинного обучения: *обучение с учителем* (supervised learning), *обучение без учителя* (unsupervised learning) и *обучение с подкреплением* (reinforcement learning). Вы узнаете о фундаментальных различиях между этими тремя разными типами обучения и, благодаря наглядным примерам, получите понимание прикладных областей, в которых они могут применяться (рис. 1.1).

¹ Термин *performance*, который обычно переводят как «производительность», на самом деле очень многозначен. В машинном обучении в зависимости от контекста он может означать точность модели, ее быстродействие, стабильность работы, а иногда все одновременно, т. е. совокупный уровень качества. В этой книге мы тоже используем перевод «производительность», подразумевая под ним свойство модели, зависящее от контекста. — Прим. пер.

² См. <https://www.nature.com/articles/nature21056>.

³ См. <https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology>.

⁴ См. <https://ai.facebook.com/blog/new-ai-research-to-help-predict-covid-19-resource-needs-from-a-series-of-x-rays/>.

⁵ См. <https://www.forbes.com/sites/roboews/2021/06/20/these-are-the-startups-applying-ai-to-tackle-climate-change>.

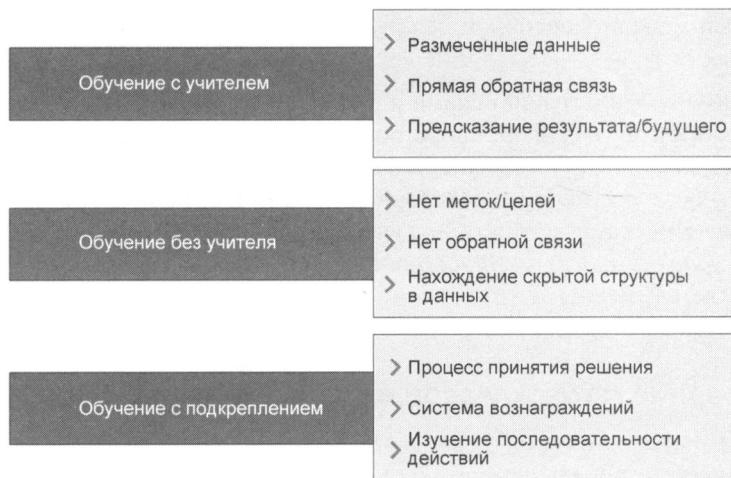


Рис. 1.1. Три типа машинного обучения

1.2.1. Предсказание будущего при помощи обучения с учителем

Основная цель обучения с учителем — обучить на помеченных данных модель, которая позволит нам делать прогнозы относительно незнакомых или будущих данных. Здесь определение «с учителем» относится к набору обучающих примеров (входных данных), для которых желаемые выходные сигналы (метки) уже известны. Фактически обучение с учителем представляет собой процесс моделирования взаимосвязи между входными данными и метками. Следовательно, обучение с учителем можно рассматривать как «обучение по меткам».

На рис. 1.2 показан типичный рабочий процесс обучения с учителем, в котором размеченные обучающие данные передаются алгоритму машинного обучения для подбора



Рис. 1.2. Процесс обучения с учителем

прогностической модели, способной делать прогнозы на основе новых неразмеченных входных данных.

Возьмем, к примеру, фильтрацию спама в электронной почте. Мы можем обучить модель с помощью алгоритма машинного обучения с учителем на пакете электронных писем, которые правильно помечены экспертами как «спам» или «не спам», чтобы предсказать, относится ли новое электронное письмо к одной из этих двух категорий. Задача обучения с метками дискретных классов, как в примере фильтрации спама, также называется *задачей классификации* (*classification task*). Другой категорией обучения с учителем является *регрессия* (*regression*), где выходной сигнал модели является непрерывным значением.

Прогнозирование меток классов в задаче классификации

Классификация — это категория задач обучения с учителем, целью которых является прогнозирование категориальных меток классов новых экземпляров или точек данных на основе предыдущих наблюдений. Эти метки классов представляют собой дискретные неупорядоченные значения, которые можно рассматривать как указатели на принадлежность точек данных к определенным группам (классам). Упомянутый ранее пример обнаружения спама в электронной почте представляет собой типичный пример задачи *бинарной классификации* (*binary classification*), где алгоритм машинного обучения изучает набор правил, чтобы научить модель различать два возможных класса: «спам» и «не спам».

Рис. 1.3 иллюстрирует смысл задачи бинарной классификации для 30 обучающих примеров. Здесь 15 обучающих примеров помечены как класс A, и 15 обучающих примеров — как класс B. В этом сценарии наш набор данных является *двумерным*, т. е. каж-

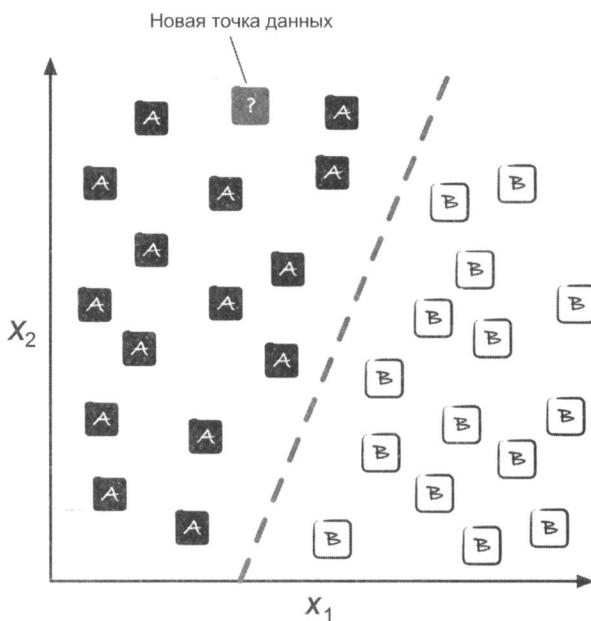


Рис. 1.3. Классификация новой точки данных

дый экземпляр имеет два связанных с ним значения: x_1 и x_2 . Теперь мы можем использовать алгоритм машинного обучения с учителем, чтобы изучить *правило* — разделяющую границу, представленную пунктирной линией, — которое может разделить эти два класса и отнести новые точки данных к одной из этих двух категорий исходя из значений x_1 и x_2 .

Однако набор меток классов не обязательно должен быть бинарным. Прогностическая модель, полученная с помощью алгоритма обучения с учителем, может назначать любую метку класса, представленную в обучающем наборе данных, новой непомеченной точке данных или экземпляру.

Типичным примером задачи *многоклассовой классификации* (multiclass classification) является распознавание рукописных символов. Мы можем собрать обучающий набор данных, состоящий из нескольких рукописных примеров каждой буквы алфавита. Буквы (А, В, С и т. д.) будут представлять в нем различные неупорядоченные категории, или метки классов, которые мы хотим предсказать. Теперь, если пользователь введет новый рукописный символ через устройство ввода, наша прогностическая модель сможет предсказать правильную букву алфавита с определенной точностью. Однако она не сможет правильно распознать, например, ни одну из цифр от 0 до 9, если они не были частью обучающего набора данных.

Прогнозирование непрерывного значения в задаче регрессии

В предыдущем разделе вы узнали, что задача классификации заключается в присвоении экземплярам категориальных неупорядоченных меток. Второй тип обучения с учителем — это прогнозирование непрерывных результатов, которое также называется *регрессионным анализом* (regression analysis). В регрессионном анализе нам дается ряд *предикторов* (predictor, объясняющая переменная) и непрерывная переменная *отклика* (outcome, результат), и мы пытаемся найти взаимосвязь между этими переменными, на основании которой сможем предсказывать результат.

Следует заметить, что в области машинного обучения переменные-предикторы обычно называются *признаками* (feature), а переменные отклика — *целевыми переменными* (target variable). Мы будем использовать эти термины на протяжении всей книги.

Например, предположим, что нас интересует прогнозирование результатов SAT по математике среди учащихся⁶. Если существует связь между временем, затраченным на подготовку к тесту, и окончательными оценками, мы могли бы использовать этот факт для обучения модели, предсказывающей результаты тестов будущих студентов на основании времени, затраченного ими на подготовку к тесту.



Регрессия к среднему

Термин «регрессия» был в 1886 году предложен Фрэнсисом Гальтоном в его статье «Регрессия к среднему при наследовании роста». Гальтон описал биологический феномен, заключающийся в том, что разница в росте внутри популяции не увеличивается с течением времени.

Он заметил, что рост родителей не передается их детям по наследству — вместо этого рост детей регрессирует к среднему значению роста в популяции.

⁶ SAT — это стандартизованный тест, часто используемый при поступлении в колледжи в Соединенных Штатах.

Рис. 1.4 иллюстрирует понятие линейной регрессии. Имея переменную признака x и целевую переменную y , мы подгоняем к этим данным прямую линию, которая минимизирует расстояние — чаще всего среднеквадратичное — между точками данных и этой линией.

Теперь мы можем использовать точку пересечения и наклон линии, полученные из известных данных, чтобы предсказать целевую переменную для новой точки данных.

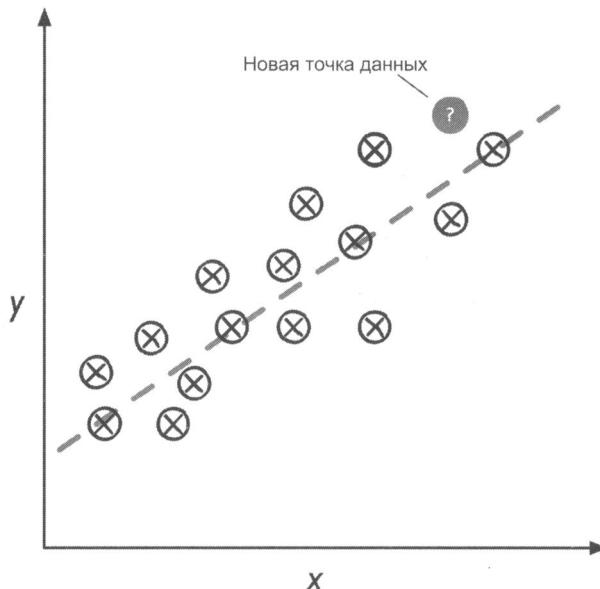


Рис. 1.4. Пример линейной регрессии

1.2.2. Применение обучения с подкреплением для решения интерактивных задач

Еще один распространенный тип машинного обучения — *обучение с подкреплением* (reinforcement learning). Цель обучения с подкреплением заключается в том, чтобы разработать *агента*, т. е. систему, которая улучшает качество своих действий на основе взаимодействия с окружающей средой. Поскольку информация о текущем состоянии среды обычно также содержит так называемый *сигнал вознаграждения* (reward signal), мы можем рассматривать обучение с подкреплением как особую разновидность обучения с учителем. Однако в обучении с подкреплением эта обратная связь является не правильной меткой или значением истинности, а мерой того, насколько высокую оценку получило действие агента от функции вознаграждения. Благодаря взаимодействию с окружающей средой, агент может использовать обучение с подкреплением, чтобы выучить серию действий, которые максимизируют это вознаграждение. Агент делает это с помощью исследовательского подхода (метод проб и ошибок) или *совещательного планирования* (deliberative planning).

Популярным примером обучения с подкреплением является шахматная программа. Здесь агент выбирает серию ходов в зависимости от состояния доски (среды), а награда может быть определена как выигрыш или проигрыш в конце игры (рис. 1.5).

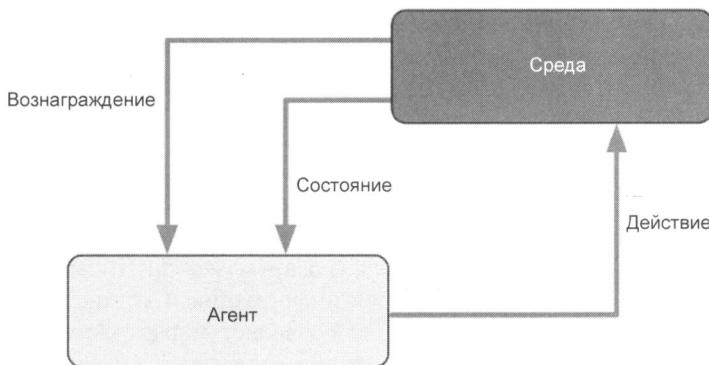


Рис. 1.5. Процесс обучения с подкреплением

Существует множество разновидностей обучения с подкреплением. Однако общая идея такова, что агент в обучении с подкреплением пытается максимизировать вознаграждение посредством серии взаимодействий с окружающей средой. Каждое состояние может быть связано с положительным или отрицательным вознаграждением, а вознаграждение может быть определено как достижение общей цели, — допустим, выигрыш или проигрыш в игре в шахматы. Например, в шахматах результат каждого хода можно рассматривать как новое состояние окружающей среды.

Продолжая рассмотрение примера с шахматами, давайте подумаем о том, что создание определенных конфигураций на шахматной доске связано с состояниями, которые с большей вероятностью приведут к победе, — например, удаление с доски шахматной фигуры противника или угроза его ферзю. Однако другие позиции связаны с состояниями, которые с большей вероятностью приведут к проигрышу в игре, — например, проигрыш шахматной фигуры противнику на следующем ходу. Важно, что в шахматной партии вознаграждение (как положительное за победу, так и отрицательное за поражение) не будет выдано до конца игры. Кроме того, итоговое вознаграждение будет зависеть еще и от тактики соперника. Например, он может пожертвовать ферзя, но в итоге выиграет партию.

Таким образом, обучение с подкреплением сводится к умению агента выбирать последовательность действий, дающих наибольшее общее вознаграждение, доступное либо сразу после совершения действия, либо с помощью *отсроченной обратной связи* (delayed feedback).

1.2.3. Обнаружение скрытых структур с помощью обучения без учителя

В обучении с учителем мы — когда обучаем модель — заранее знаем правильный ответ (метку или целевую переменную), а в обучении с подкреплением сами назначаем меру вознаграждения за определенные действия, выполняемые агентом. Однако при обучении без учителя мы имеем дело с непомеченными данными или данными неизвестной структуры. Используя методы обучения без учителя, мы можем получить значимую информацию о структуре данных, не применяя известные переменные результата или функцию вознаграждения.

Поиск подгрупп путем кластеризации

Кластеризация (*clustering*) — это метод исследовательского анализа данных или обнаружения закономерностей, который позволяет нам распределить большой объем неупорядоченных данных по *кластерам* (*cluster*), не зная заранее о принадлежности точек данных к определенным группам. Каждый кластер, возникающий в ходе анализа, определяет группу объектов, имеющих определенную степень сходства между собой, но относительно непохожих на объекты в других кластерах, поэтому кластеризацию также иногда называют *классификацией без учителя* (*unsupervised classification*). Кластеризация — отличный метод для структурирования информации и установления значимых взаимосвязей между данными. Например, она позволяет маркетологам обнаруживать группы клиентов на основе их интересов, чтобы разрабатывать отдельные маркетинговые программы.

На рис. 1.6 показано, как можно применить кластеризацию для организации непомеченных данных в три отдельные группы или кластеры (A, B и C в произвольном порядке) на основе сходства их характеристик x_1 и x_2 .

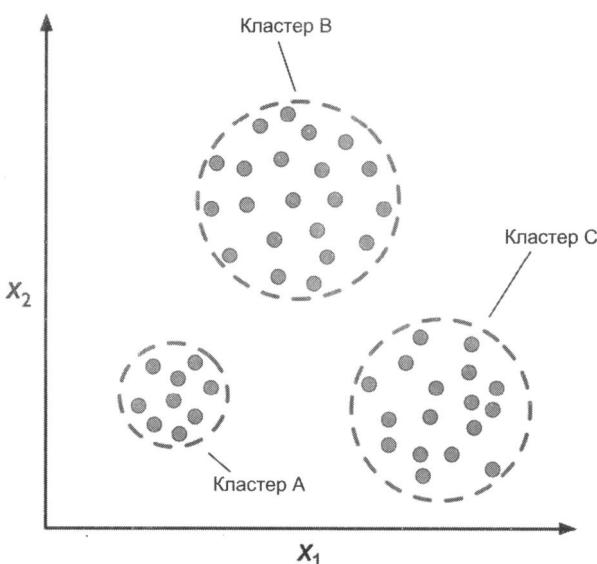


Рис. 1.6. Кластеризация

Уменьшение размерности для сжатия данных

Еще одной областью применения обучения без учителя является *уменьшение размерности* (*dimensionality reduction*). Часто мы работаем с данными высокой размерности — когда каждое наблюдение сопровождается большим количеством измерений — что может представлять проблему в случае ограниченной памяти для хранения и низкой вычислительной эффективности алгоритмов машинного обучения. Уменьшение размерности при помощи обучения без учителя — это широко используемый подход в предварительной обработке признаков для удаления из данных шума, который может ухудшить прогностическую эффективность некоторых алгоритмов. Уменьшение раз-

мерности сжимает данные в подпространство меньшего размера, сохраняя при этом большую часть релевантной информации.

Иногда уменьшение размерности необходимо для визуализации данных — например, многомерный набор признаков можно спроектировать на одно-, двух- или трехмерные пространства признаков, чтобы визуализировать его с помощью двух- или трехмерных диаграмм рассеяния или гистограмм. На рис. 1.7 показан пример, в котором нелинейное уменьшение размерности применяется для сжатия трехмерной фигуры «швейцарский рулет» в новое подпространство двухмерных объектов.

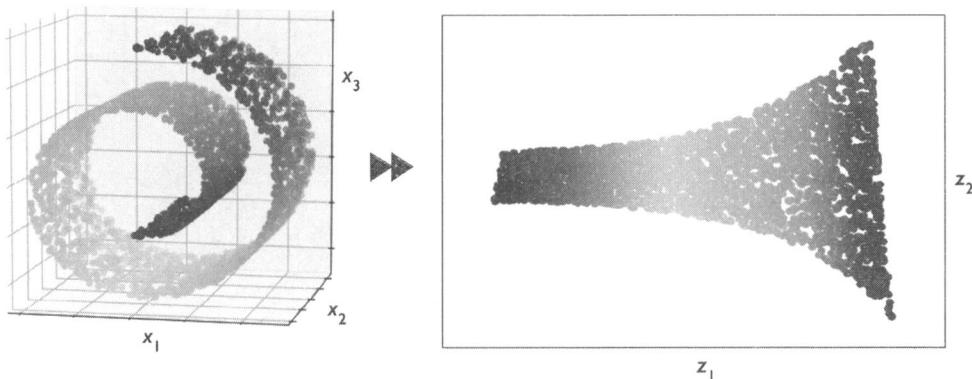


Рис. 1.7. Пример уменьшения размерности с трех измерений до двух

1.3. Введение в основную терминологию и обозначения

Теперь, когда мы обсудили три обширные категории машинного обучения: с учителем, без учителя и обучение с подкреплением, — настало время познакомиться с основными терминами, которые вы встретите в этой книге. В разд. 1.3.1 представлены общие термины, которые мы будем использовать при описании различных аспектов наборов данных, а также соответствующие математические обозначения, которые помогут вам избежать неоднозначности.

Поскольку машинное обучение — это область, в которой пересекаются различные дисциплины, вы обязательно рано или поздно столкнетесь с ситуацией, когда разные термины обозначают одни и те же понятия. В разд. 1.3.2 собраны многие из наиболее часто используемых терминов, встречающихся в литературе по машинному обучению. Этот раздел может послужить вам справочником при чтении публикаций, касающихся этой темы.

1.3.1. Соглашения и условные обозначения, используемые в этой книге

На рис. 1.8 показан фрагмент набора данных Iris, который представляет собой классический пример в области машинного обучения⁷. Набор данных Iris содержит данные

⁷ Дополнительную информацию про этот набор можно найти по адресу: <https://archive.ics.uci.edu/ml/datasets/iris>.

измерений 150 цветков ирисов трех разных видов: ирис щетинистый (*Iris setosa*), ирис виргинский (*Iris virginica*) и ирис разноцветный (*Iris versicolor*).

Каждому экземпляру цветка соответствует одна строка в наборе данных, а размеры цветка в сантиметрах представлены в виде столбцов, которые также называют *признаками набора данных*.

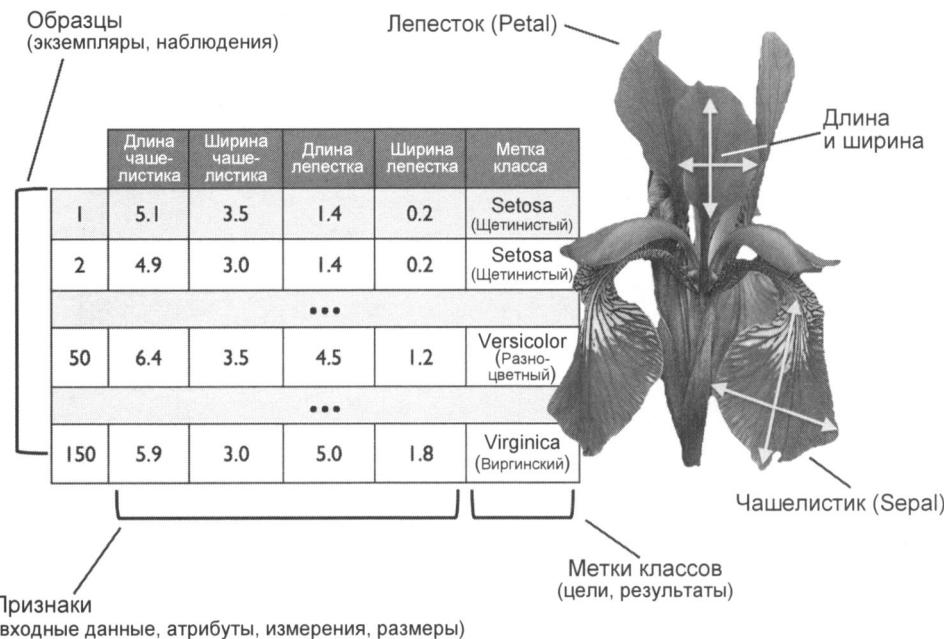


Рис. 1.8. Набор данных Iris

Чтобы обозначения и реализация были простыми, но эффективными, мы обратимся к основам линейной алгебры и в следующих главах для обозначения данных будем использовать матричную запись, следуя при этом общепринятому соглашению о представлении каждого экземпляра данных в виде отдельной строки в матрице признаков X , где каждый признак представлен в виде отдельного столбца.

Набор данных Iris, состоящий из 150 экземпляров и четырех признаков, можно записать в виде матрицы 150×4 , формально обозначаемой как $X \in \mathbb{R}^{150 \times 4}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}.$$



Условные обозначения

В большинстве глав этой книги, если не указано иное, мы будем использовать:

- верхний индекс i — для обозначения i -го обучающего экземпляра, а нижний индекс j — для обозначения j -го измерения обучающего набора данных;

- строчные жирные буквы для обозначения векторов: $x \in \mathbb{R}^{n \times 1}$ и прописные жирные буквы для обозначения матриц: $X \in \mathbb{R}^{n \times m}$.

Для обозначения отдельных элементов в векторе или матрице мы будем выделять буквы курсивом: $x^{(n)}$ или $x_m^{(n)}$ соответственно.

Например, $x_1^{(150)}$ обозначает первый размер экземпляра цветка 150 — т. е. длину наружной доли его чашелистика, или, как их еще называют, *околоцветника* (*sepal length*). При этом каждая строка в матрице X представляет один экземпляр цветка и может быть записана как четырехмерный вектор-строка $x^{(i)} \in \mathbb{R}^{1 \times 4}$:

$$x^{(i)} = [x_1^{(i)} \ x_2^{(i)} \ x_3^{(i)} \ x_4^{(i)}].$$

А каждое измерение признака представляет собой 150-мерный вектор-столбец $X^{(i)} \in \mathbb{R}^{150 \times 1}$, например:

$$x_j = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \dots \\ x_j^{(150)} \end{bmatrix}.$$

Аналогичным образом мы можем представить целевые переменные (здесь метки классов) как 150-мерный вектор-столбец:

$$y = \begin{bmatrix} y^{(1)} \\ \dots \\ y^{(150)} \end{bmatrix}, \text{ where } y^{(i)} \in \{\text{Setosa, Versicolor, Virginica}\}.$$

1.3.2. Терминология машинного обучения

Машинное обучение — это обширная междисциплинарная область, поскольку она объединяет ученых, работающих в разных отраслях науки. Так случилось, что многие термины и определения были в ней предложены заново или переопределены и, возможно, некоторые понятия уже знакомы вам под другими именами. Для вашего удобства в следующем списке представлена подборка часто используемых терминов и их синонимов, которые могут пригодиться при чтении этой книги и литературы по машинному обучению в целом:

- ◆ **обучающий пример:** строка в таблице, представляющая набор данных. Это синоним *наблюдения, записи, экземпляра или выборки* (в большинстве случаев выборка относится к набору обучающих примеров);
- ◆ **учение:** подгонка модели к данным, аналогичная оценке параметров параметрических моделей;
- ◆ **признак** (обозначается символом x) — столбец в таблице или матрице данных. Синоним *предиктора, переменной, входных данных, атрибута или ковариата*;
- ◆ **цель** (обозначается символом y) — синоним *результата, выхода, переменной отклика, зависимой переменной, метки (класса) и эталона*.
- ◆ **функция потерь:** часто используется как синоним *функции стоимости или затрат*. Иногда функцию потерь также называют функцией *ошибки*. В ряде источни-

ков термин «потери» относится к потере, измеренной для одной точки данных, а стоимость — это измерение, которое вычисляет убыток (средний или суммарный) по всему набору данных.

1.4. Общие принципы построения систем машинного обучения

Ранее мы обсудили основные концепции машинного обучения и три различных типа обучения. В этом разделе мы расскажем о других важных составляющих системы машинного обучения, включая алгоритм обучения.

На рис. 1.9 показан типичный рабочий процесс использования машинного обучения в прогнозном моделировании, которое мы обсудим далее.

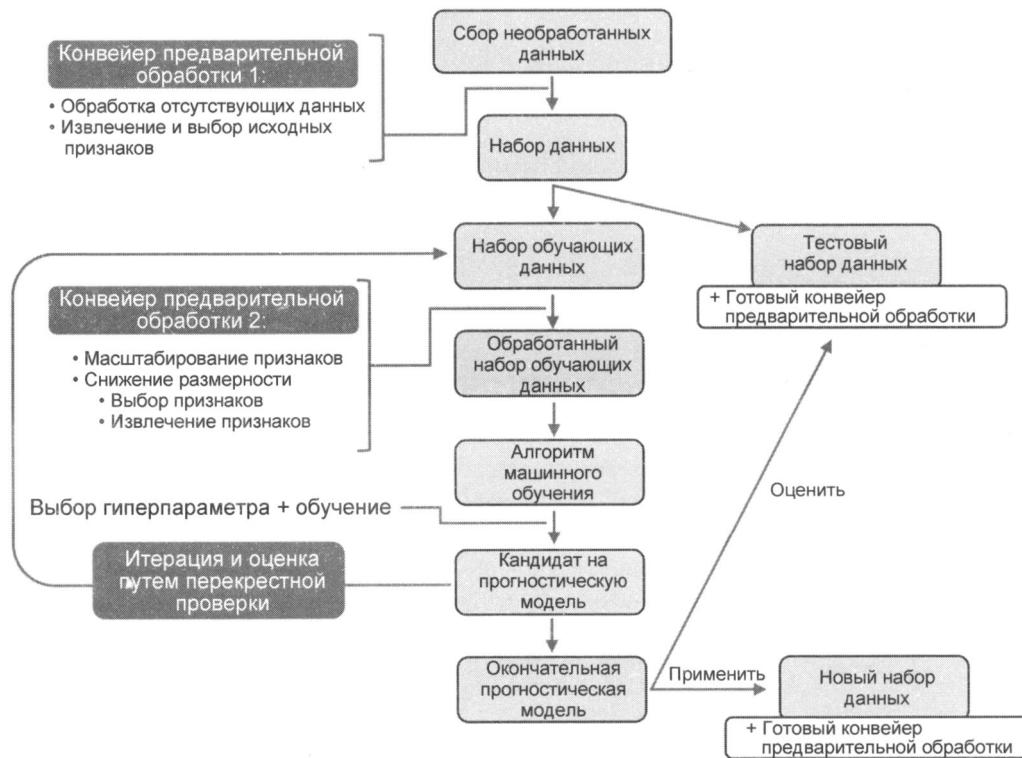


Рис. 1.9. Обобщенная схема процесса прогнозного моделирования

1.4.1. Предварительная обработка: приведение данных к нужному виду

Необработанные данные редко поступают в той форме, которая необходима для оптимальной работы алгоритма обучения. Поэтому предварительная обработка данных является одним из наиболее важных шагов в любом приложении машинного обучения.

Если мы возьмем для примера набор данных о цветках ириса из предыдущего раздела, то в роли необработанных данных будет выступать набор изображений цветков, из которых мы хотим извлечь значимые признаки. В качестве таких признаков могут быть приняты цвет лепестков или высота, длина и ширина цветков.

Многие алгоритмы машинного обучения также требуют, чтобы выбранные признаки имели одинаковый масштаб для достижения оптимальной производительности модели, чего часто добиваются путем приведения значений признаков к диапазону $[0, 1]$ или к стандартному нормальному распределению с нулевым средним значением и единичной дисперсией, как вы увидите в последующих главах.

Некоторые из выбранных признаков могут быть сильно коррелированы и, следовательно, в определенной степени избыточны. В этих случаях полезно применить методы уменьшения размерности для сжатия объектов в подпространство более низкого измерения. Уменьшение размерности пространства признаков дает то преимущество, что требуется меньше места для хранения данных, и алгоритм обучения может работать намного быстрее. В некоторых случаях уменьшение размерности также может улучшить прогностическую эффективность модели, если набор данных содержит большое количество нерелевантных признаков (или шума), — иными словами, если набор данных имеет низкое соотношение сигнал/шум.

Чтобы убедиться, что наш алгоритм машинного обучения не только хорошо работает с обучающим набором данных, но и успешно обобщает новые данные, необходимо случайным образом разделить основной набор данных на два отдельных набора: обучающий и тестовый. Мы используем обучающий набор данных для обучения и оптимизации нашей модели и сохраняем тестовый набор данных в неприкасновенности до самого конца, чтобы оценить окончательную модель.

1.4.2. Обучение и выбор прогностической модели

Как вы увидите в следующих главах, для решения разных задач было разработано множество различных алгоритмов машинного обучения. Важный вывод, который можно сделать из знаменитой теоремы Дэвида Вулперта об отсутствии бесплатных завтраков⁸, заключается в том, что мы не можем получить обучение «бесплатно»⁹. Здесь уместна и другая знаменитая фраза: «Если единственный инструмент, который у вас есть, — это молоток, все вокруг будет казаться вам гвоздями» (Абраам Маслоу, 1966). Например, у каждого алгоритма классификации есть присущие ему искажения, и ни одна модель классификации не имеет априорного преимущества, если мы не делаем никаких предположений о задаче. Поэтому на практике важно сравнивать хотя бы несколько различных алгоритмов обучения, чтобы обучить и выбрать наилучшую модель. Но прежде чем мы сможем сравнивать разные модели, нам сначала нужно выбрать измеримый показатель качества. Одним из часто используемых показателей является точность

⁸ Согласно Википедии, крылатая фраза «Бесплатных завтраков не бывает» (англ. There ain't no such thing as a free lunch) подразумевает, что получение какой-либо выгоды всегда связано с затратами, даже если эти затраты на первый взгляд не видны.

⁹ См. «The Lack of A Priori Distinctions Between Learning Algorithms», D. H. Wolpert, 1996 и «No free lunch theorems for optimization», D. H. Wolpert and W. G. Macready, 1997.

классификации (*classification accuracy*), которая определяется как доля правильно классифицированных экземпляров.

Возникает закономерный вопрос: как узнать, какая модель хорошо работает с окончательным набором тестовых данных и с реальными данными, если мы не применяем набор тестовых данных для выбора модели, а сохраняем его для окончательной оценки? Ответом на этот вопрос служат различные методы под общим названием *перекрестная проверка* (*cross-validation*). При перекрестной проверке мы дополнительно делим набор данных на обучающие и проверочные (валидационные) подмножества, чтобы оценить эффективность обобщения модели.

Наконец, мы не можем ожидать, что параметры по умолчанию различных алгоритмов обучения, представленных программными библиотеками, оптимальны для каждой конкретной задачи. Поэтому в последующих главах мы будем часто использовать методы оптимизации гиперпараметров, которые помогут нам точно настроить производительность нашей модели.

Гиперпараметры удобно рассматривать как параметры модели, которые не извлекаются из данных, а представляют собой «ручки настройки» модели, которые можно вращать, чтобы улучшить ее производительность. Это определение станет намного понятнее в последующих главах, когда вы увидите реальные примеры.

1.4.3. Оценка моделей и прогнозирование незнакомых экземпляров данных

Выбрав модель, которая успешно прошла обучение, мы можем воспользоваться тестовым набором данных, чтобы оценить, насколько хорошо она работает с этими незнакомыми данными, и определить так называемую *ошибку обобщения* (*generalization error*). Если мы удовлетворены полученным результатом, то можем использовать эту модель для прогнозирования будущих данных. Важно отметить, что параметры ранее упомянутых процедур, таких как масштабирование признаков и уменьшение размерности, получаются исключительно из обучающего набора данных, и те же параметры позже повторно применяются для преобразования тестового набора данных, а также любых новых экземпляров данных — в противном случае производительность, измеренная на тестовых данных, может быть чрезмерно оптимистичной.

1.5. Использование Python для машинного обучения

Python — один из самых популярных языков программирования, применяемых в науке о данных, и благодаря высокой активности его разработчиков и обширному сообществу его открытого исходного кода разработано большое количество полезных библиотек для научных вычислений и машинного обучения.

Хотя быстродействие интерпретируемых языков, таких как Python, в задачах с интенсивными вычислениями уступает низкоуровневым языкам программирования, существуют быстродействующие библиотеки расширений, такие как NumPy и SciPy, написанные на низкоуровневых языках Fortran и С и предназначенные для выполнения быстрых векторных операций над многомерными массивами.

Занимаясь программированием для задач машинного обучения, мы в основном будем использовать библиотеку scikit-learn, которая в настоящее время является одной из самых популярных и доступных библиотек машинного обучения с открытым исходным кодом. В последующих главах, когда мы сосредоточимся на разновидности машинного обучения, называемой *глубоким обучением* (deep learning), мы станем ориентироваться на последнюю версию библиотеки PyTorch, которая специализируется на очень эффективном обучении моделей так называемых *глубоких нейронных сетей* (deep neural network) с использованием графических видеокарт.

1.5.1. Установка Python и пакетов из Python Package Index

Python доступен для всех трех основных операционных систем: Microsoft Windows, macOS и Linux — а его установщик и документацию можно загрузить с официального веб-сайта Python: <https://www.python.org>.

Примеры кода, представленные в этой книге, написаны и протестированы для версии Python 3.9, но мы рекомендуем вам использовать самую последнюю доступную версию Python 3. Часть кода также может быть совместима с Python 2.7, но поскольку официальная поддержка Python 2.7 закончилась в 2019 году, да и большинство библиотек с открытым исходным кодом уже прекратили поддержку Python 2.7 (см. <https://python3statement.org>), мы настоятельно советуем вам использовать Python 3.9 или новее.

Вы можете узнать свою версию Python, выполнив команду:

```
python --version
```

или

```
python3 --version
```

в вашем терминале командной строки (или PowerShell, если вы используете Windows).

Дополнительные пакеты, которые мы будем использовать в этой книге, можно установить с помощью установщика pip, который является частью стандартной библиотеки Python, начиная с версии Python 3.3. Дополнительную информацию о pip можно найти по адресу: <https://docs.python.org/3/installing/index.html>.

После завершения установки Python можно выполнить pip из терминала, чтобы установить дополнительные пакеты (вместо *SomePackage* подставьте имя нужного вам пакета):

```
pip install SomePackage
```

Уже установленные пакеты можно обновить с помощью флага `--upgrade`:

```
pip install SomePackage --upgrade
```

1.5.2. Использование дистрибутива Python Anaconda и менеджера пакетов

Очень популярной системой управления пакетами с открытым исходным кодом для установки Python с целью научных вычислений является пакет conda от Continuum Analytics, бесплатно распространяемый под лицензией с открытым исходным кодом. Его цель — помочь с установкой и управлением версиями пакетов Python в разных

операционных системах специалистам в областях науки о данных, математики и инженерных знаний. Если вы хотите использовать conda, то имейте в виду, что пакет этот поставляется в трех разных вариантах — а именно: Anaconda, Miniconda и Miniforge:

- ◆ Anaconda поставляется с предустановленными пакетами для научных вычислений. Установщик Anaconda можно загрузить по адресу: <https://docs.anaconda.com/anaconda/install/>, а краткое руководство по Anaconda доступно по адресу: <https://docs.anaconda.com/anaconda/user-guide/>;
- ◆ Miniconda — более компактная альтернатива Anaconda (<https://docs.conda.io/en/latest/miniconda.html>). По сути, она похожа на Anaconda, но без предустановленных пакетов, что нравится многим пользователям (включая авторов этой книги);
- ◆ Miniforge похож на Miniconda, но поддерживается сообществом и использует другой репозиторий пакетов (conda-forge), отличный от Miniconda и Anaconda. Мы обнаружили, что Miniforge — отличная альтернатива Miniconda. Инструкции по загрузке и установке Miniforge можно найти в репозитории GitHub по адресу: <https://github.com/conda-forge/miniforge>.

После успешной установки conda через Anaconda, Miniconda или Miniforge мы можем установить новые пакеты Python, используя следующую команду (вместо *SomePackage* подставьте имя нужного вам пакета):

```
conda install SomePackage
```

Существующие пакеты можно обновить с помощью команды:

```
conda update SomePackage
```

Пакеты, недоступные с использованием официального канала conda, могут быть доступны через поддерживаемый сообществом проект conda-forge (<https://conda-forge.org>), который можно указать через флаг `--channel` в conda-forge. Например:

```
conda install SomePackage --channel conda-forge
```

Пакеты, недоступные через канал conda по умолчанию или conda-forge, можно установить через pip, как было показано ранее — например, так:

```
pip install SomePackage
```

1.5.3. Пакеты для научных вычислений, науки о данных и машинного обучения

В первой половине этой книги для хранения данных и управления ими мы будем в основном использовать многомерные массивы NumPy. При этом время от времени мы будем задействовать pandas — построенную на основе NumPy библиотеку, содержащую дополнительные инструменты для работы с данными, написанные на языке высокого уровня и делающие работу с табличными данными еще более удобной. Чтобы расширить ваш опыт работы с глубоким обучением и визуализировать количественные данные, что часто чрезвычайно полезно для их понимания, мы будем применять библиотеку Matplotlib, обладающую возможностями гибкой настройки.

Основной библиотекой машинного обучения, используемой в этой книге, является scikit-learn (см. главы с 3 по 11). Затем в главе 12 будет представлена библиотека PyTorch для глубокого обучения.

Номера версий основных пакетов Python, использованных при написании этой книги, указаны в следующем списке. Пожалуйста, убедитесь, что номера версий установленных у вас пакетов совпадают (в идеале) с номерами из этого списка, — тогда примеры кода будут работать правильно:

- ◆ NumPy 1.21.2;
- ◆ SciPy 1.7.0;
- ◆ Scikit-learn 1.0;
- ◆ Matplotlib 3.4.3;
- ◆ pandas 1.3.2.

После установки указанных пакетов вы можете перепроверить установленные версию, импортировав пакет в Python и обратившись к его атрибуту `_version_`, например, так:

```
>>> import numpy  
>>> numpy._version_  
'1.21.2'
```

Для вашего удобства мы включили скрипт `python-environment-check.py` в бесплатный репозиторий кода этой книги, расположенный по адресу: <https://github.com/rasbt/machine-learning-book>, чтобы вы могли проверить как свою версию Python, так и версии пакетов, просто запустив этот скрипт.

Для некоторых глав потребуются дополнительные пакеты — в этом случае вам там будет предоставлена информация об их установке. Например, сейчас пока вы можете не беспокоиться об установке PyTorch, — в главе 12, когда вам понадобятся советы и инструкции, вы их получите.

Если вы столкнулись с ошибками при выполнении, даже если ваш код точно совпадает с кодом, приведенным в главе, мы рекомендуем вам сначала проверить номера версий базовых пакетов, прежде чем тратить время на отладку или обращение к издателю или авторам. Иногда в более новые версии библиотек внесены изменения, несовместимые со старыми версиями, что служит причиной ошибок.

Если вы не хотите менять версию своего базового пакета Python, мы рекомендуем вам для установки пакетов, используемых в этой книге, задействовать виртуальную среду. Если вы работаете с Python без диспетчера conda, то для создания новой виртуальной среды можете применить библиотеку `venv`. Например, создать и активировать виртуальную среду можно с помощью следующих двух команд:

```
python3 -m venv /Users/sebastian/Desktop/pyml-book  
source /Users/sebastian/Desktop/pyml-book/bin/activate
```

Обратите внимание, что вам будет необходимо активировать виртуальную среду каждый раз, когда вы открываете новый терминал или Power-Shell. Дополнительную информацию о `venv` можно найти по адресу: <https://docs.python.org/3/library/venv.html>.

Если вы используете Anaconda с менеджером пакетов conda, то можете создать и активировать виртуальную среду следующим образом:

```
conda create -n pyml python=3.9  
conda activate pyml
```

1.6. Заключение

В этой главе мы окинули взглядом машинное обучение и познакомили вас с общей картиной и основными понятиями, которые более подробно представлены в следующих главах. Вы узнали, что обучение с учителем делится на два важных типа: классификация и регрессия. Классифицирующие модели позволяют нам распределять объекты по известным классам, а регрессионный анализ применяется для прогнозирования значений непрерывных целевых переменных. Обучение без учителя способно не только выявлять характерные структуры в неразмеченных данных, но и служит для сжатия данных на этапах предварительной обработки признаков.

Мы кратко рассмотрели общие принципы применения машинного обучения к типичным задачам, которые послужат основой для более глубокого обсуждения и практических примеров в следующих главах. Наконец, мы настроили среду Python, установили и обновили необходимые пакеты, и теперь готовы увидеть машинное обучение в действии.

Далее в этой книге, помимо самого машинного обучения, мы рассмотрим различные методы предварительной обработки набора данных, которые помогут вам добиться максимальной производительности от различных алгоритмов машинного обучения. Хотя в книге достаточно подробно представлены алгоритмы классификации, мы не оставим без внимания различные методы регрессионного анализа и кластеризации.

Впереди нас ждет захватывающее путешествие, в ходе которого мы исследуем множество мощных методов в обширной области машинного обучения. Однако мы будем подходить к машинному обучению шаг за шагом, постепенно наращивая свои знания с каждой новой главой этой книги. В следующей главе мы начнем с реализации одного из самых первых алгоритмов машинного обучения для классификации и подготовимся перейти к *главе 3*, где изучим более продвинутые алгоритмы машинного обучения, используя библиотеку машинного обучения с открытым исходным кодом *scikit-learn*.

2

Простые алгоритмы машинного обучения для задач классификации

В этой главе мы рассмотрим два наиболее известных базовых алгоритма машинного обучения для задач классификации: персепtron и адаптивные линейные нейроны. Мы начнем с пошаговой реализации персептрана на Python и обучим его классифицировать различные виды цветков в наборе данных Iris. Это поможет понять идеи, лежащие в основе алгоритмов машинного обучения для задач классификации и освоить приемы эффективной реализации алгоритмов на языке Python.

Знание основ оптимизации с использованием адаптивных линейных нейронов послужит фундаментом для реализации более сложных классификаторов с помощью библиотеки машинного обучения scikit-learn в главе 3.

В этой главе будут рассмотрены следующие темы:

- ◆ построение алгоритмов машинного обучения;
- ◆ использование pandas, NumPy и Matplotlib для чтения, обработки и визуализации данных;
- ◆ реализация линейных классификаторов для задач с двумя классами на Python.

2.1. Искусственные нейроны: краткий экскурс в историю машинного обучения

Прежде чем приступить к подробному изучению персептрана и связанных с ним алгоритмов, давайте совершим краткое путешествие к истокам машинного обучения. Пытаясь понять, как работает биологический мозг, исследователи в области *искусственного интеллекта* (ИИ), Уоррен МакКаллох и Уолтер Питтс в 1943 году опубликовали первую концепцию упрощенной клетки мозга — так называемого нейрона МакКаллоха — Питтса (MCP)¹.

Биологические нейроны — это взаимосвязанные нервные клетки в головном мозге, участвующие в обработке и передаче химических и электрических сигналов. Их устройство показано на рис. 2.1.

¹ См.: «A Logical Calculus of the Ideas Immanent in Nervous Activity», W. S. McCulloch and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115–133, 1943 (Логическое исчисление идей, имманентных нервной деятельности. В. МакКаллок и У. Питтс, Бюллетень математической биофизики, 5 (4): 115–133, 1943).

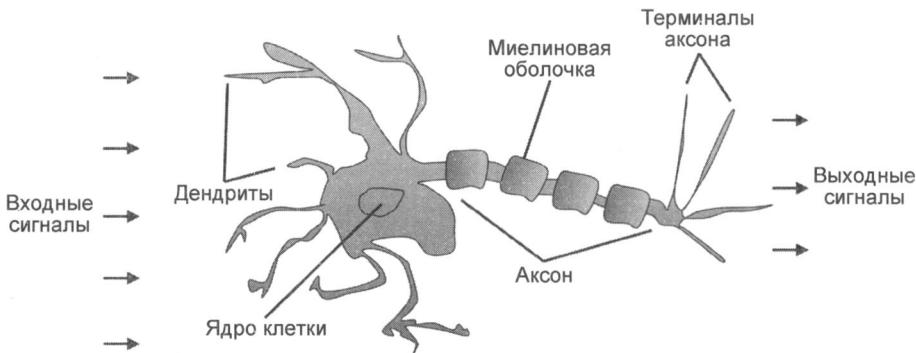


Рис. 2.1. Нейрон, обрабатывающий химические и электрические сигналы

Маккаллох и Питтс описали такую нервную клетку как простой логический вентиль с бинарными выходами; к дендритам поступает несколько сигналов, которые суммируются в теле клетки, и если суммарный сигнал превышает определенный порог, генерируется выходной сигнал, передаваемый дальше через аксон.

Всего несколько лет спустя Фрэнк Розенблatt предложил идею обучения персептрана на основе модели нейрона MCP². Исходя из правила персептрана, Розенблatt предложил алгоритм, автоматически определяющий оптимальные весовые коэффициенты, которые затем перемножаются с входными признаками, чтобы принять решение о том, возбуждается ли нейрон (передает сигнал дальше) или нет. В контексте классификации обученную при помощи такого алгоритма модель можно затем использовать для прогнозирования принадлежности новой точки данных к тому или иному классу.

2.1.1. Формальное определение искусственного нейрона

Более формально мы можем рассматривать идею искусственных нейронов с точки зрения задачи бинарной классификации с двумя классами: 0 и 1. Мы можем определить решающую функцию $\sigma(z)$, которая принимает линейную комбинацию определенных входных значений x и соответствующий весовой вектор w , где z — так называемый *фактический вход* (net input) $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}.$$

Если фактический вход конкретного экземпляра $x^{(i)}$ превышает определенный порог θ , мы предсказываем класс 1, в противном случае предсказываем класс 0. В алгоритме персептрана решающая функция $\sigma(\cdot)$ представляет собой разновидность *единичной ступенчатой функции* (unit step function):

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{в ином случае} \end{cases}.$$

² «The Perceptron: A Perceiving and Recognizing Automaton» by F. Rosenblatt, Cornell Aeronautical Laboratory, 1957.

Чтобы упростить последующую реализацию алгоритма в виде кода, выполним несколько простых преобразований. Сначала перенесем пороговое значение θ в левую часть уравнения:

$$\begin{aligned} z &\geq \theta, \\ z - \theta &\geq 0. \end{aligned}$$

Затем определим так называемое *смещение* (bias unit) как $b = -\theta$ и сделаем его частью фактического входа:

$$z = w_1x_1 + \dots + w_mx_m + b = \mathbf{w}^T \mathbf{x} + b.$$

И наконец, учитывая наличие смещения и выполненное переопределение фактического входа z , мы можем переопределить решающую функцию следующим образом:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{в ином случае} \end{cases}.$$



Основы линейной алгебры: скалярное произведение и транспонирование матриц

На протяжении всей книги мы будем часто прибегать к основным обозначениям из линейной алгебры. Например, применять сокращенную запись суммы произведений значений \mathbf{x} и \mathbf{w} , используя векторно-скалярное произведение, где верхний индекс T означает транспонирование, которое представляет собой операцию, преобразующую вектор-столбец в вектор-строку и наоборот. Например, предположим, что у нас есть следующие два вектор-столбца:

$$\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}.$$

Мы можем записать операцию транспонирования вектора \mathbf{a} как $\mathbf{a}^T = [a_1 \ a_2 \ a_3]$ и представить скалярное произведение следующим образом:

$$\mathbf{a}^T \mathbf{b} = \sum_i a_i b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3.$$

Кроме того, операцию транспонирования также можно применить к матрицам, чтобы отразить их по диагонали, — например:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}.$$

Заметим, что операция транспонирования строго определена только для матриц, однако когда в контексте машинного обучения мы говорим «вектор», то имеем в виду матрицы $n \times 1$ или $1 \times m$.

В этой книге мы будем использовать только самые базовые понятия линейной алгебры, но тем не менее если вам нужно быстро освежить свои знания, взгляните на превосходный обзор и справочник по линейной алгебре Зико Колтера, который находится в свободном доступе по адресу: http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf.

На рис. 2.2 показано, как фактический вход $z = \mathbf{w}^T \mathbf{x} + b$ сжимается в двоичный выход (0 или 1) решающей функцией персептрана (слева) и как его можно использовать для различия двух классов, разделяемых линейной решающей границей (справа).

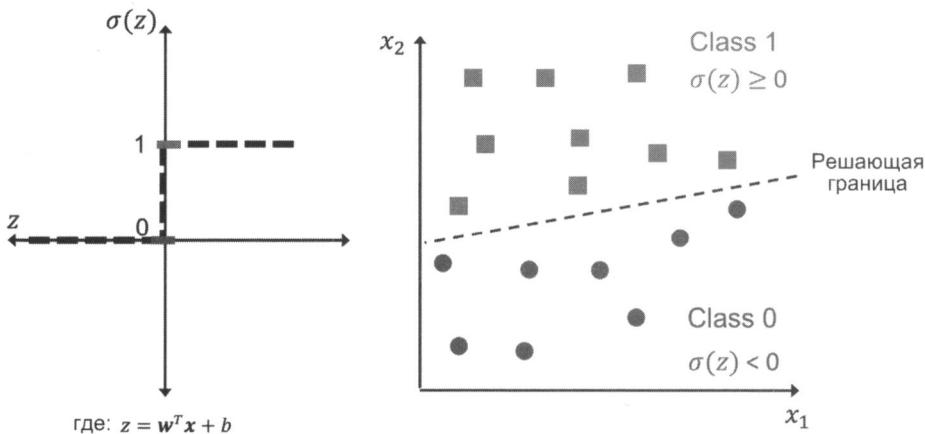


Рис. 2.2. Пороговая функция, определяющая линейную границу решения для задачи бинарной классификации

2.1.2. Правило обучения персептрана

Идея, лежащая в основе нейрона MCP и *пороговой* модели персептрана Розенблатта, заключается в использовании весьма упрощенного подхода для имитации того, как работает отдельный нейрон в мозгу: он либо срабатывает, либо нет. Поэтому классическое правило персептрана Розенблатта довольно простое, и соответствующий алгоритм можно свести к следующим шагам:

1. Инициализируем компоненты веса и смещения нулями или небольшими случайными числами.
2. Для каждого обучающего образца $x^{(i)}$:
 - Вычисляем выходное значение $\hat{y}^{(i)}$.
 - Обновляем компоненты веса и смещения.

Здесь выходным значением является метка класса, предсказанная единичной ступенчатой функцией, которую мы определили ранее, а одновременное обновление смещения и каждого веса w_j в векторе весов w можно более формально записать как:

$$w_j := w_j + \Delta w_j,$$

и $b := b + \Delta b$.

Значения обновления («дэльты») вычисляются следующим образом:

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

и $\Delta b = \eta(y^{(i)} - \hat{y}^{(i)})$.

Обратите внимание, что, в отличие от смещения, каждый вес w_j соответствует признаку x_j в наборе данных, участвующему в вычислении выполненного обновления Δw_j .

Кроме того, η — это скорость обучения (обычно константа от 0.0 до 1.0), $y^{(i)}$ — истинная метка класса i -го обучающего образца, а $\hat{y}^{(i)}$ — прогнозируемая метка класса. Важно отметить, что смещение и все веса в векторе весов обновляются одновременно, т. е. мы не пересчитываем спрогнозированную метку $\hat{y}^{(i)}$ до обновления всех весов и смещения с помощью значений Δw_j и Δb соответственно. В частности, для двумерного набора данных мы запишем обновление следующим образом:

$$\begin{aligned}\Delta w_1 &= \eta(y^{(i)} - \text{output}^{(i)})x_1^{(i)}; \\ \Delta w_2 &= \eta(y^{(i)} - \text{output}^{(i)})x_2^{(i)}; \\ \Delta b &= \eta(y^{(i)} - \text{output}^{(i)}).\end{aligned}$$

Прежде чем реализовать правило персептрана на языке Python, давайте проведем небольшой мысленный эксперимент, чтобы увидеть, насколько оно простое на самом деле. В двух сценариях, где персептрон правильно предсказывает метку класса, единица смещения и веса остаются неизменными, поскольку значения обновления равны 0:

$$(1) \quad y^{(i)} = 0, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(0 - 0)x_j^{(i)} = 0, \quad \Delta b = \eta(0 - 0) = 0$$

$$(2) \quad y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1 - 1)x_j^{(i)} = 0, \quad \Delta b = \eta(1 - 1) = 0$$

Однако в случае неправильного прогноза возникает положительный или отрицательный сдвиг весов в сторону целевого класса:

$$(3) \quad y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(1 - 0)x_j^{(i)} = \eta x_j^{(i)}, \quad \Delta b = \eta(1 - 0) = \eta$$

$$(4) \quad y^{(i)} = 0, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(0 - 1)x_j^{(i)} = -\eta x_j^{(i)}, \quad \Delta b = \eta(0 - 1) = -\eta$$

Чтобы лучше понять значение признака как мультипликативного фактора $x_j^{(i)}$, давайте рассмотрим другой простой пример, где

$$y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \eta = 1.$$

Предположим, что $x_j^{(i)} = 1.5$, и мы обнаружили, что ошибочно классифицируем этот пример как *класс 0*. В этом случае мы увеличим соответствующий вес в общей сложности на 2.5, так что фактический вход $z = x_j^{(i)} \times w_j + b$ будет более положительным в следующий раз, когда мы столкнемся с этим примером, и, следовательно, с большей вероятностью превысит порог единичной ступенчатой функции, что позволит классифицировать этот пример как *класс 1*:

$$\Delta w_j = (1 - 0)1.5 = 1.5, \quad \Delta b = (1 - 0) = 1.$$

Обновление веса Δw_j пропорционально значению $x_j^{(i)}$. Например, если у нас есть другой пример: $x_j^{(i)} = 2$, что неправильно классифицируется как *класс 0*, мы сдвинем границу решения еще сильнее, чтобы правильно классифицировать этот пример в следующий раз:

$$\Delta w_j = (1 - 0)2 = 2, \quad \Delta b = (1 - 0) = 1.$$

Важно отметить, что сходимость персептрана гарантируется только в том случае, если два класса *линейно разделимы*, т. е. могут быть полностью разделены линейной гра-

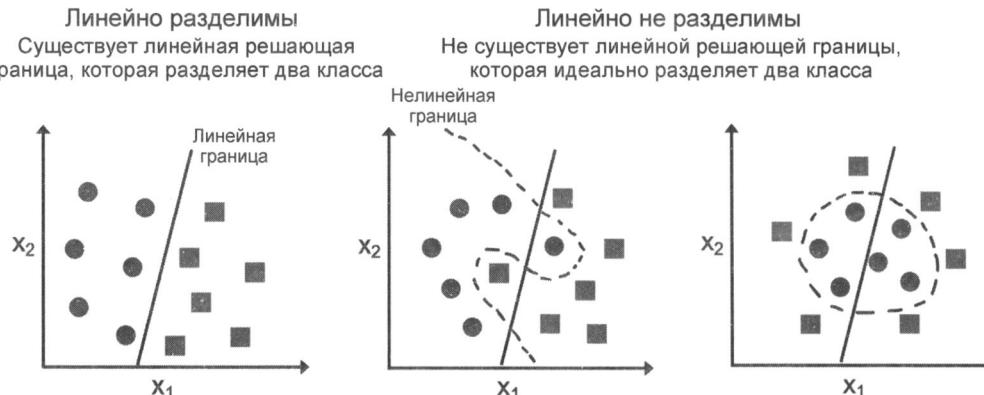


Рис. 2.3. Примеры линейно разделимых и линейно неразделимых классов

ницией³. На рис. 2.3 показаны наглядные примеры линейно разделимых и линейно неразделимых классов.

Если есть подозрение, что два класса не могут быть разделены линейной границей, следует установить максимальное количество проходов по набору обучающих данных (количество эпох) и/или пороговое значение для количества допустимых ошибочных классификаций — в противном случае персептрон никогда не перестанет обновлять веса. Позже в этой главе мы рассмотрим алгоритм Adaline, который создает линейные разделяющие границы и сходится, даже если классы не являются идеально линейно разделимыми. А в главе 3 вы узнаете об алгоритмах, которые могут строить нелинейные разделяющие границы.

Все, о чем здесь говорилось, можно обобщить на простой схеме, иллюстрирующей общий принцип работы персептрана (рис. 2.4), где показано, что персептрон получает

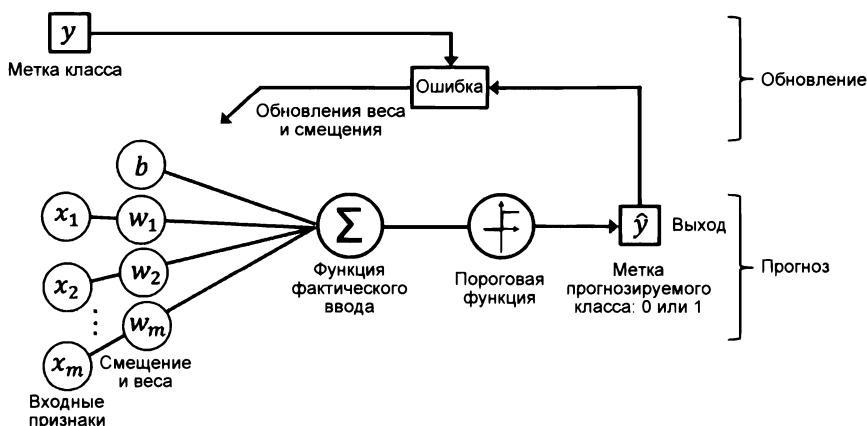


Рис. 2.4. Веса и смещения модели обновляются с помощью функции ошибок

³ Занинтересованные читатели могут найти доказательство сходимости в моих конспектах лекций:
https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf.

входные данные образца (x) и объединяет их со смещением (b) и весами (w) для вычисления фактического входа. Затем этот вход передается пороговой функции, которая генерирует двоичный вывод 0 или 1 — прогнозируемую метку класса для текущего примера. На этапе обучения этот вывод используется для вычисления ошибки прогноза и обновления весов и смещения.

2.2. Реализация алгоритма обучения персептрона на Python

В предыдущем разделе вы узнали, как работает правило обучения персептрона Розенблатта. Теперь настало время реализовать его на Python и применить к набору данных Iris, с которым вы познакомились в главе 1.

2.2.1. Объектно-ориентированный API персептрона

Мы воспользуемся объектно-ориентированным подходом к определению интерфейса персептрона в виде класса Python, что позволит нам инициализировать новые объекты Perceptron, которые могут обучаться на основе данных с помощью метода `fit` и делать прогнозы с помощью отдельного метода `predict`. По соглашению мы добавляем символ подчеркивания «`_`» к атрибутам, если они не создаются при инициализации объекта, но когда это происходит при вызове других методов объекта — например: `self.w_`.



Дополнительные ресурсы Python для научных вычислений

Если вы еще не знакомы с научными библиотеками Python или вам нужно освежить знания, рекомендую посетить следующие сайты:

- NumPy: <https://sebastianraschka.com/blog/2020/numpy-intro.html>;
- pandas: https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html;
- Matplotlib: <https://matplotlib.org/stable/tutorials/introductory/usage.html>.

Далее приведен код реализации персептрона на Python:

```
import numpy as np
class Perceptron:
    """Персептронный классификатор.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0)
    n_iter : int
        Кол-во проходов по обучающему набору.
    random_state : int
        Опорное значение генератора случайных чисел для инициализации весов.

    Атрибуты
    -----
    w_ : 1d-array
        Веса после подгонки.
```

```
b_ : Scalar
    Смещение после подгонки.
errors_ : list
    Количество неправильных классификаций (обновлений) в каждой эпохе.

"""
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state

def fit(self, X, y):
    """ Соответствие тренировочным данным.

Параметры
-----
X : {array-like}, shape = [n_examples, n_features]
    Обучающий вектор, где n_examples – это количество образцов,
    а n_features – количество признаков.
y : array-like, shape = [n_examples]
    Целевые значения.

Возвращаемые значения
-----
self : object

"""
    rgen = np.random.RandomState(self.random_state)
    self.w_ = rgen.normal(loc=0.0, scale=0.01,
                          size=X.shape[1])
    self.b_ = np.float_(0.)
    self.errors_ = []

    for _ in range(self.n_iter):
        errors = 0
        for xi, target in zip(X, y):
            update = self.eta * (target - self.predict(xi))
            self.w_ += update * xi
            self.b_ += update
            errors += int(update != 0.0)
        self.errors_.append(errors)
    return self

def net_input(self, X):
    """Вычисление фактического входа"""
    return np.dot(X, self.w_) + self.b_

def predict(self, X):
    """Возвращает метки класса после шага"""
    return np.where(self.net_input(X) >= 0.0, 1, 0)
```

Используя эту реализацию персептрона, мы теперь можем инициализировать новые объекты `Perceptron` с заданной скоростью обучения `eta` (η) и количеством эпох `n_iter` (проходит через набор обучающих данных).

С помощью метода `fit` мы инициализируем смещение `self.b_` начальным значением 0, а веса в `self.w_` — вектором \mathbb{R}^m , где m обозначает количество измерений (признаков) в наборе данных.

Обратите внимание, что начальный вектор весов содержит небольшие случайные числа, взятые из нормального распределения со стандартным отклонением 0.01 через `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, где `rgen` — генератор случайных чисел NumPy, который мы инициализировали заданным пользователем случайнм начальным числом, чтобы при желании можно было бы воспроизвести предыдущие результаты.

Технически мы могли бы инициализировать веса нулями (на самом деле так и сделано в исходном алгоритме персептрона). Однако если бы мы это сделали, то скорость обучения η (этота) не повлияла бы на разделяющую границу. Если все веса инициализированы нулем, параметр скорости обучения `eta` влияет только на масштаб вектора весов, а не на направление. Возьмем, к примеру, вектор $v1 = [1 \ 2 \ 3]$, где угол между $v1$ и вектором $v2 = 0.5 \times v1$ будет точно равен нулю, как показано в следующем фрагменте кода:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...                           np.linalg.norm(v2)))
0.0
```

Здесь `np.arccos` — тригонометрический арккосинус, а `np.linalg.norm` — функция, вычисляющая длину вектора. (Наше решение взять случайные числа из нормального распределения — например, вместо равномерного распределения, — и использовать стандартное отклонение 0.01 было произвольным. Помните, нас просто интересуют небольшие случайные значения, чтобы избежать нулевых начальных векторов, как обсуждалось ранее.)

В качестве дополнительного упражнения после прочтения этой главы вы можете изменить `self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])` на `self.w_ = np.zeros(X.shape[1])` и запустить обучающий код персептрона, представленный в следующем разделе, с разными значениями `eta`. Вы заметите, что граница решения не меняется.



Индексация массива NumPy

Индексация NumPy для одномерных массивов работает аналогично спискам Python с использованием нотации с квадратными скобками «`[]`». В случае двумерных массивов первый индекс ссылается на номер строки, а второй — на номер столбца. Например, для выбора третьей строки и четвертого столбца двумерного массива `x` следует использовать `x[2, 3]`.

После инициализации весов метод `fit` перебирает все отдельные примеры в обучающем наборе данных и обновляет веса в соответствии с правилом обучения персептрона, которое мы обсуждали в предыдущем разделе.

За предсказание метки класса отвечает метод `predict`, который вызывается в методе `fit` во время обучения, чтобы получить метку класса для обновления веса, но `predict` также

можно использовать для предсказания меток классов новых данных после обучения модели. Кроме того, мы также собираем количество ошибочных классификаций в течение каждой эпохи в списке `self.errors_`, чтобы позже можно было проанализировать, насколько хорошо наш персептрон работал во время обучения. Функция `np.dot`, используемая в методе `net_input`, просто вычисляет векторно-скалярное произведение $w^T x + b$.



Векторизация: замена циклов `for` векторизованным кодом

Вместо использования NumPy для вычисления векторного скалярного произведения между двумя массивами `a` и `b` при помощи `a.dot(b)` или `np.dot(a, b)` мы могли бы также выполнить вычисление, используя обычный цикл Python `sum([i * j for i, j in zip(a, b)])`. Однако преимущество использования библиотеки NumPy по сравнению с классическим циклом `for` языка Python заключается в том, что ее арифметические операции векторизованы. *Векторизация* означает, что элементарная арифметическая операция автоматически применяется ко всем элементам массива. Формулируя наши арифметические операции как последовательность инструкций, применяемых к массиву, а не выполняя набор операций для каждого элемента за раз, мы более оптимально используем современные архитектуры *центрального процессора* (ЦП) с поддержкой SIMD (Single Instruction, Multiple Data, одна инструкция, много данных). Кроме того, NumPy использует высокооптимизированные библиотеки линейной алгебры, такие как *базовые подпрограммы линейной алгебры* (Basic Linear Algebra Subprograms, BLAS) и пакет линейной алгебры (Linear Algebra Package, LAPACK), написанные на С или Fortran. Наконец, NumPy позволяет нам писать более компактный и интуитивно понятный код, используя основные понятия линейной алгебры, такие как векторные и матричные скалярные произведения.

2.2.2. Обучение модели персептрана на наборе данных Iris

Занимаясь тестированием нашей реализации персептрана в оставшихся примерах этой главы, мы ограничимся двумя переменными признаков (измерениями). Хотя правило персептрана не ограничивается двумя измерениями, рассмотрение только двух признаков: длины чашелистика и длины лепестка — позволит нам визуализировать области принятия решений обученной модели на диаграмме рассеяния для большей наглядности.

Обратите внимание, что мы также рассмотрим только два класса цветков из набора данных Iris: `setosa` и `versicolor` — исходя из простого практического соображения: персептрон представляет собой бинарный классификатор. Однако алгоритм персептрана может быть расширен до многоклассовой классификации — например, метода «один против всех» (One-versus-All, OvA).



Многоклассовая классификация при помощи метода OvA

Метод OvA, называемый иногда также «один против остальных» (One-versus-Rest, OvR), позволяет нам расширить любой двоичный классификатор до задач с несколькими классами. Используя OvA, мы можем обучать один классификатор для каждого класса, где определенный класс рассматривается как положительный класс, а примеры из всех других классов считаются отрицательными классами. Если нам нужно классифицировать новый непомеченный экземпляр данных, мы используем *n* классификаторов, где *n* — количество меток класса, и присваиваем

экземпляру, который мы хотим классифицировать, метку класса, обладающего наибольшей достоверностью. В случае с персепtronом мы использовали бы ОвА для выбора метки класса, связанной с наибольшим абсолютным фактическим входным значением.

Мы начнем с использования библиотеки pandas для загрузки набора данных Iris из репозитория машинного обучения UCI в объект DataFrame и распечатаем последние пять строк с помощью метода tail, чтобы проверить правильность загрузки данных:

```
>>> import os
>>> import pandas as pd
>>> s = 'https://archive.ics.uci.edu/ml/' \
...      'machine-learning-databases/iris/iris.data'
>>> print('From URL:', s)
From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
data
>>> df = pd.read_csv(s,
...                     header=None,
...                     encoding='utf-8')
>>> df.tail()
```

После выполнения этого кода вы должны увидеть на экране последние пять строк набора данных Iris (рис. 2.5).

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Рис. 2.5. Последние пять строк набора данных Iris



Загрузка набора данных Iris

Вы можете найти копию этого набора данных (и всех других наборов данных, используемых в книге) в ее файловом архиве, когда работаете в автономном режиме, или если сервер UCI по адресу: <https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data>, временно недоступен. Например, чтобы загрузить набор данных Iris из локального каталога, нужно заменить строки приведенного ранее кода:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/' \
    'machine-learning-databases/iris/iris.data',
    header=None, encoding='utf-8')
```

на следующие:

```
df = pd.read_csv(
    'your/local/path/to/iris.data',
    header=None, encoding='utf-8')
```

Далее мы извлекаем первые 100 меток классов, которые соответствуют 50 цветкам Iris-setosa и 50 цветкам Iris-versicolor, и преобразуем метки классов в две целочисленные метки: 1 (versicolor) и 0 (setosa), которые мы присваиваем вектору y , где метод values кадра данных pandas DataFrame дает соответствующее представление NumPy.

Точно так же мы извлекаем первый столбец признаков (длина чашелистика) и третий столбец признаков (длина лепестков) из этих 100 обучающих образцов и присваиваем их матрице признаков X , которую визуализируем с помощью двумерной диаграммы рассеяния:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # выбираем setosa и versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', 0, 1)
>>> # извлекаем длину чашелистика и длину лепестка
>>> X = df.iloc[0:100, [0, 2]].values
>>> # отображаем данные
>>> plt.scatter(X[:50, 0], X[:50, 1],
...                 color='red', marker='o', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...                 color='blue', marker='s', label='Versicolor')
>>> plt.xlabel('Длина чашелистика [см]')
>>> plt.ylabel('Длина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Выполнив этот пример кода, вы получите диаграмму рассеяния (рис. 2.6), где показано распределение образцов цветов в наборе данных Iris по двум осям признаков: длина лепестков и длина чашелистиков (измеренных в сантиметрах). Судя по двумерному

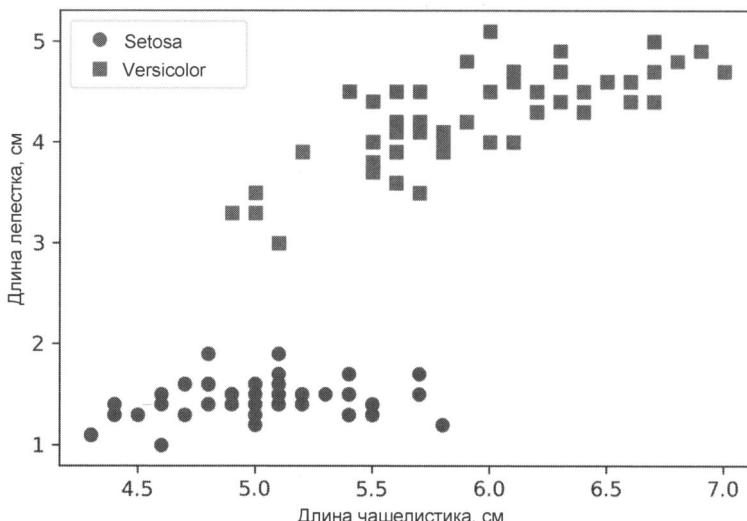


Рис. 2.6. Диаграмма рассеяния цветков setosa и versicolor по длине чашелистиков и лепестков

пространству признаков, линейной разделяющей границы должно быть достаточно, чтобы отделить цветки setosa от цветков versicolor. Следовательно, линейный классификатор, такой как персептрон, должен идеально классифицировать цветки в этом наборе данных.

Теперь пришло время обучить наш алгоритм персептрана на подмножестве данных Iris, которое мы только что извлекли. Кроме того, мы построим график ошибки неправильной классификации для каждой эпохи, чтобы проверить, сошелся ли алгоритм и нашел ли он границу, которая разделяет два класса цветов ириса:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
...           ppn.errors_, marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Количество обновлений')
>>> plt.show()
```

Обратите внимание, что количество ошибок неправильной классификации и количество обновлений одинаково, поскольку веса и смещение персептрана обновляются каждый раз, когда он ошибочно классифицирует образец. Выполнив этот код, вы получите график зависимости ошибок классификации от количества эпох (рис. 2.7).

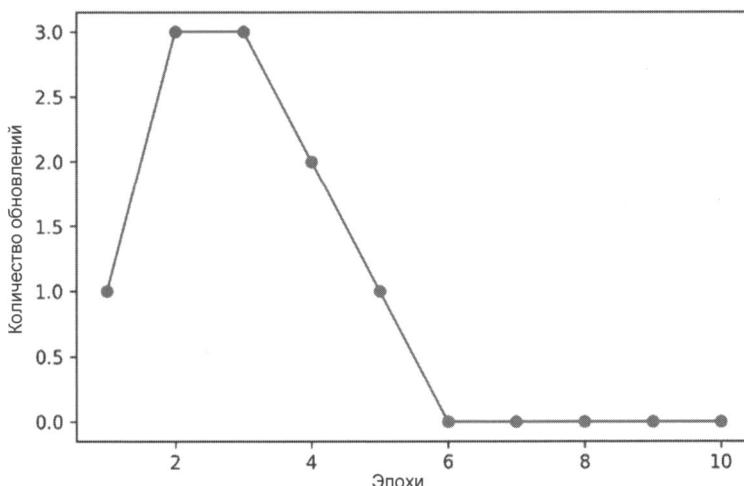


Рис. 2.7. График зависимости количества ошибок классификации от количества эпох

Как можно видеть, наш персептрон сошелся после шестой эпохи и теперь должен идеально классифицировать обучающие примеры. Давайте напишем небольшую удобную функцию визуализации решающих границ для двумерных наборов данных:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # Настройка генератора меток и цветовой карты
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
```

```

# построение решающей поверхности
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                       np.arange(x2_min, x2_max, resolution))
lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
lab = lab.reshape(xx1.shape)
plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
plt.xlim(xx1.min(), xx1.max())
plt.ylim(xx2.min(), xx2.max())

# построение образцов класса
for idx, cl in enumerate(np.unique(y)):
    plt.scatter(x=X[y == cl, 0],
                y=X[y == cl, 1],
                alpha=0.8,
                c=colors[idx],
                marker=markers[idx],
                label=f'Class {cl}',
                edgecolor='black')

```

Сначала мы здесь определяем количество `colors` и `markers` и создаем палитру из списка цветов при помощи `ListedColormap`. Затем определяем минимальное и максимальное значения для двух признаков и используем эти векторы признаков для создания с помощью функции NumPy `meshgrid` пары матричных массивов: `xx1` и `xx2`. Поскольку мы обучили наш классификатор персептрона на двух измерениях признаков, нам нужно получить плоские матричные массивы и создать матрицу с тем же количеством столбцов, что и обучающее подмножество Iris, — чтобы мы могли использовать метод `predict` для прогнозирования меток классов `lab` соответствующих узлов сетки.

Выполнив преобразование меток прогнозируемых классов `lab` в сетку с теми же размерами, что и `xx1` и `xx2`, теперь мы можем нарисовать контурный график с помощью функции `contourf` библиотеки `Matplotlib`, которая отображает различные области решений в разные цвета для каждого прогнозируемого класса в матричном массиве:

```

>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Длина чашелистика [см]')
>>> plt.ylabel('Длина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.show()

```

Выполнив приведенный пример кода, вы должны получить графическое представление областей принятия решений (рис. 2.8).

Как мы видим на графике, персептрон нашел разделяющую границу, которая может идеально классифицировать все примеры цветков в обучающем подмножестве Iris.



Сходимость персептрана

Хотя персептрон идеально классифицировал два класса цветков ириса, сходимость — одна из самых больших проблем персептрана. Розенблatt математически доказал, что правило обучения персептрана сходится, если два класса можно разде-

лить линейной гиперплоскостью. Однако, если классы не могут быть идеально разделены такой линейной границей, веса никогда не перестанут обновляться, если мы не установим максимальное количество эпох. Заинтересованные читатели могут найти краткое изложение доказательства в моих конспектах лекций по адресу: https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf.

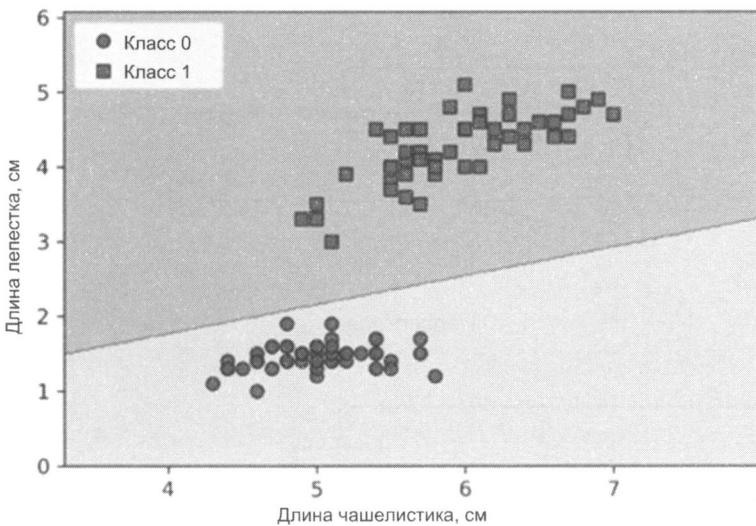


Рис. 2.8. График областей принятия решения перцептроном

2.3. Адаптивные линейные нейроны и сходимость обучения

В этом разделе мы рассмотрим еще один тип однослойной нейронной сети — Adaline (сокращение от Adaptive Linear Neuron, адаптивный линейный нейрон). Описание Adaline было опубликовано Бернардом Видроу и его докторантом Теддом Хоффом всего через несколько лет после публикации алгоритма персептрана Розенблatta, и его можно считать усовершенствованным вариантом последнего⁴.

Алгоритм Adaline особенно интересен тем, что он иллюстрирует ключевые концепции определения и минимизации непрерывных функций потерь. Это закладывает основу для понимания других алгоритмов машинного обучения, таких как логистическая регрессия, машины опорных векторов и многослойные нейронные сети, а также модели линейной регрессии, которые мы обсудим в следующих главах.

Ключевое различие между правилом Adaline (также известным как *правило Видроу — Хоффа*) и персептраном Розенблatta заключается в том, что веса обновляются на основе линейной функции активации, а не единичной ступенчатой функции, как в персеп-

⁴ См.: «An Adaptive “Adaline” Neuron Using Chemical “Memistors”», Technical Report Number 1553-2 by B. Widrow and colleagues, Stanford Electron Labs, Stanford, CA, October 1960.

троне. В Adaline эта линейная функция активации $\sigma(z)$ является просто тождественной функцией сетевого входа, так что $\sigma(z) = z$.

В то время как линейная функция активации служит для изучения весов, для окончательного прогноза по-прежнему применяется пороговая функция, аналогичная единичной ступенчатой функции.

Основные различия между персептроном и алгоритмом Adaline показаны на рис. 2.9.

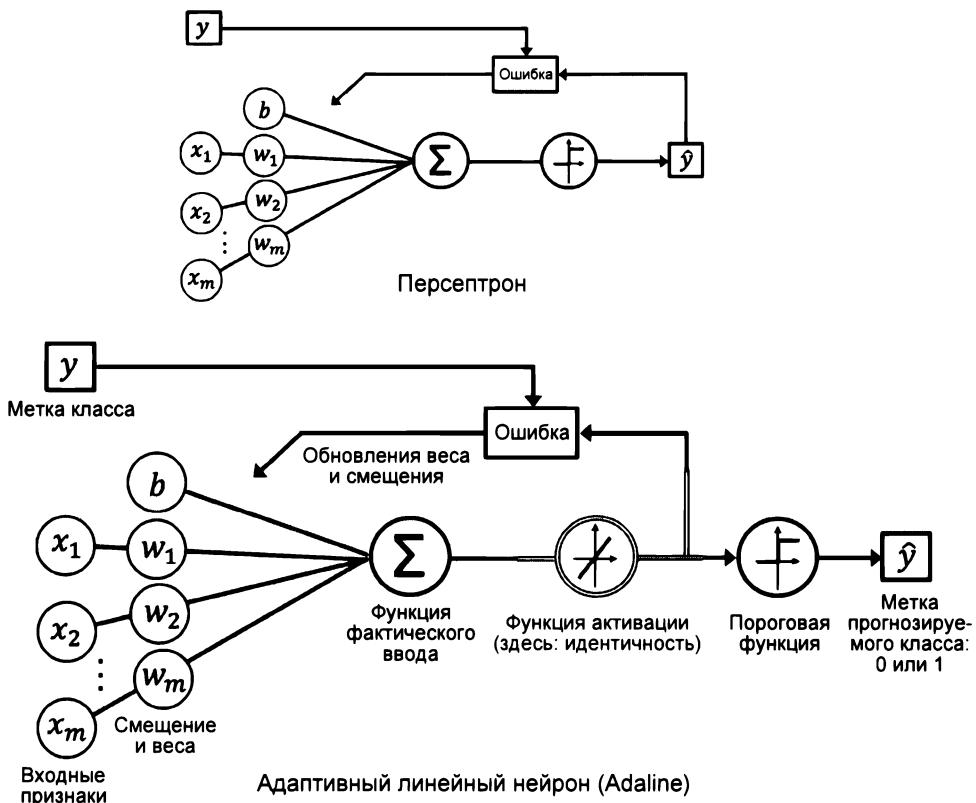


Рис. 2.9. Сравнение персептрана и алгоритма Adaline

Как можно видеть, алгоритм Adaline сравнивает истинные метки классов с непрерывным выходным значением линейной функции активации, чтобы вычислить ошибку модели и обновить веса. Напротив, персептрон сравнивает истинные метки классов с предсказанными метками классов.

2.3.1. Минимизация функции потерь с помощью градиентного спуска

Одним из ключевых компонентов алгоритмов машинного обучения с учителем является определенная *целевая функция* (objective function), которую необходимо оптимизировать в процессе обучения. Эта целевая функция часто представляет собой функцию

потерь или затрат, которую следует минимизировать. В случае Adaline мы можем определить функцию потерь L как *среднеквадратичную ошибку* (Mean Squared Error, MSE) между вычисленным выходным значением и истинной меткой класса:

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \sigma(z^{(i)}))^2.$$

Основное преимущество этой непрерывной линейной функции активации, в отличие от единичной ступенчатой функции, состоит в том, что функция потерь становится дифференцируемой. Еще одним приятным свойством этой функции потерь является то, что она выпуклая, — благодаря этому сочетанию качеств мы можем использовать очень простой, но мощный алгоритм оптимизации, называемый *градиентным спуском* (gradient descent), чтобы найти веса, которые минимизируют нашу функцию потерь для классификации примеров в наборе данных Iris.

Как показано на рис. 2.10, мы можем описать основную идею градиентного спуска как *спуск с холма*, длящийся до тех пор, пока не будет достигнут локальный или глобальный минимум потерь. На каждой итерации мы делаем шаг в направлении, противоположном градиенту, где размер шага определяется значением скорости обучения, а также наклоном градиента (для простоты на рисунке этот процесс показан только для одиночного веса w).

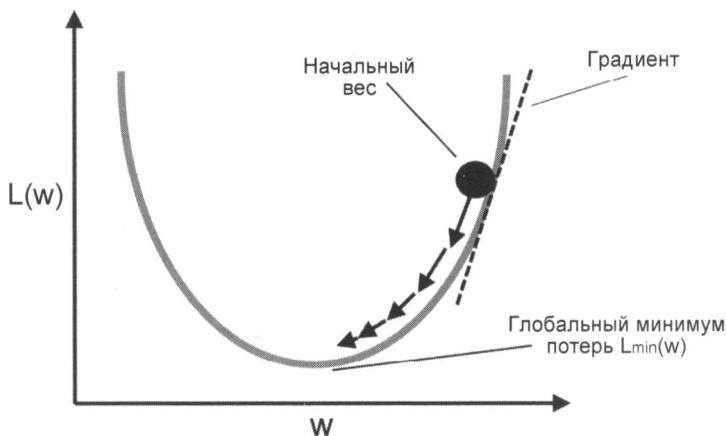


Рис. 2.10. Так работает градиентный спуск

Используя градиентный спуск, мы теперь можем обновить параметры модели, сделав шаг в направлении, противоположном градиенту $\nabla L(w, b)$ нашей функции потерь $L(w, b)$:

$$w := w + \Delta w, \quad b := b + \Delta b.$$

Изменения параметров Δw и Δb определяются как отрицательный градиент, умноженный на скорость обучения η :

$$\Delta w = -\eta \nabla_w L(w, b), \quad \Delta b = -\eta \nabla_b L(w, b).$$

Чтобы получить градиент функции потерь, нам нужно вычислить частную производную функции потерь по каждому весу w_i :

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}.$$

Аналогичным образом мы вычисляем частную производную потерь по смещению:

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})).$$

Обратите внимание, что 2 в числителе приведенной формулы — это просто постоянный коэффициент масштабирования, и его исключение никак не влияет на алгоритм. Удаление коэффициента масштабирования имеет тот же эффект, что и изменение скорости обучения в 2 раза. В следующей врезке объясняется, откуда возникает этот коэффициент масштабирования.

Теперь мы можем записать обновление веса и смещения следующим образом:

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \quad \text{и} \quad \Delta b = -\eta \frac{\partial L}{\partial b}.$$

Поскольку мы обновляем все параметры одновременно, наше правило обучения Adaline приобретает вид:

$$w := w + \Delta w, \quad b := b + \Delta b$$



Производная среднеквадратичной ошибки

Если вы знакомы с основами дифференциального исчисления, то знаете, что частную производную функции потерь MSE по j -му весу можно получить следующим образом:

$$\begin{aligned} \frac{\partial L}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))^2 = \frac{1}{n} \frac{\partial}{\partial w_j} \sum_i (y^{(i)} - \sigma(z^{(i)}))^2 = \\ &= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) \frac{\partial}{\partial w_j} (y^{(i)} - \sigma(z^{(i)})) = \\ &= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_j (w_j x_j^{(i)} + b) \right) = \\ &= \frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) (-x_j^{(i)}) = -\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}. \end{aligned}$$

Тот же подход можно использовать для нахождения частной производной $\frac{\partial L}{\partial b}$, за

исключением того, что $\frac{\partial}{\partial b} (y^{(i)} - \sum_j (w_j x_j^{(i)} + b))$ равно -1 , и поэтому последний шаг упрощается до $-\frac{2}{n} \sum_i (y^{(i)} - \sigma(z^{(i)}))$.

Хотя правило обучения Adaline выглядит идентично правилу персептрана, мы должны отметить, что $\sigma(z^{(i)})$, где $z^{(i)} = w^T x^{(i)} + b$, является действительным числом, а не целочисленной меткой класса. Кроме того, обновление веса вычисляется на основе всех образцов в наборе обучающих данных (вместо постепенного обновления параметров после каждого обучающего образца), поэтому этот подход также называется *пакетным*.

градиентным спуском (batch gradient descent). Для точности и во избежание путаницы при обсуждении связанных понятий далее в книге мы будем называть этот процесс *полным пакетным градиентным спуском* (full batch gradient descent).

2.3.2. Реализация Adaline на Python

Поскольку правило персептрана и Adaline очень похожи, мы возьмем реализацию персептрана, которую определили ранее, и изменим метод `fit`, чтобы параметры веса и смещения теперь обновлялись путем минимизации функции потерь по методу градиентного спуска:

```
class AdalineGD:  
    """Классификатор на адаптивных линейных нейронах.  
  
    Параметры  
    -----  
    eta : float  
        Скорость обучения (между 0.0 и 1.0)  
    n_iter : int  
        Количество проходов по обучающему набору.  
    random_state : int  
        Начальное значение для случайной инициализации весов.  
  
    Атрибуты  
    -----  
    w_ : Одномерный массив  
        Веса после обучения.  
    b_ : Скаляр  
        Смещение после обучения.  
    losses_ : list  
        Значения среднеквадратичной функции потерь после каждой эпохи.  
    ....  
    def __init__(self, eta=0.01, n_iter=50, random_state=1):  
        self.eta = eta  
        self.n_iter = n_iter  
        self.random_state = random_state  
  
    def fit(self, X, y):  
        """ Подгонка к обучающим данным.  
  
    Параметры  
    -----  
    X : {array-like}, shape = [n_examples, n_features]  
        Обучающие векторы, где n_examples -  
        количество образцов и  
        n_features - количество признаков.  
    y : array-like, shape = [n_examples]  
        Целевые переменные.
```

```

Возвращаемые значения
-----
self : object

"""
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01,
                      size=X.shape[1])
self.b_ = np.float_(0.)
self.losses_ = []

for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
    self.b_ += self.eta * 2.0 * errors.mean()
    loss = (errors**2).mean()
    self.losses_.append(loss)
return self

def net_input(self, X):
    """Вычисление фактического ввода"""
    return np.dot(X, self.w_) + self.b_

def activation(self, X):
    """Вычисление линейной активации"""
    return X

def predict(self, X):
    """Возвращаем метку класса"""
    return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)

```

Вместо обновления весов после оценки на каждом отдельном обучающем образце, как в персептроне, мы вычисляем градиент на основе всего обучающего набора данных. Для смещения это делается при помощи:

```
self.eta * 2.0 * errors.mean()
```

где errors — это массив, содержащий значения частных производных $\frac{\partial}{\partial b}$. Аналогичным образом мы обновляем веса. Однако обратите внимание, что обновления весов через частные производные $\frac{\partial L}{\partial w_j}$, включающие значения признаков x_j , вычисляют

перемножением errors и каждого значения признака для каждого веса:

```
for w_j in range(self.w_.shape[0]):
    self.w_[w_j] += self.eta *
        (2.0 * (X[:, w_j]*errors)).mean()
```

Чтобы реализовать более эффективное обновление весов без использования цикла `for`, можно воспользоваться операцией умножения матрицы на вектор и перемножить нашу матрицу признаков и вектор ошибки:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```

Заметим, что метод `activation` не влияет на код, т. к. это просто тождественная функция. Здесь мы добавили функцию активации (вычисленную с помощью метода `activation`), чтобы проиллюстрировать общий порядок прохождения информации через однослойную нейронную сеть: признаки из входных данных, фактический ввод, активация и вывод.

В следующей главе вы узнаете о классификаторе логистической регрессии, который использует нетождественную нелинейную функцию активации, и увидите, что модель логистической регрессии тесно связана с моделью Adaline и отличается лишь функциями активации и потерь.

Далее, как и в предыдущей реализации персептрана, мы собираем значения потерь в список `self.losses_`, чтобы проверить, сошелся ли алгоритм после обучения.



Матричное умножение

Выполнение матричного умножения похоже на вычисление векторно-скалярного произведения, где каждая строка в матрице рассматривается как один вектор-строка. Этот векторизованный подход представляет собой более компактную запись и позволяет выполнять более эффективные вычисления с использованием NumPy, например:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}.$$

В предыдущем уравнении мы умножали матрицу на вектор, который математически не определен. Однако следует помнить о нашем соглашении, что этот вектор рассматривается как матрица 3×1 .

На практике часто требуется провести несколько экспериментов, чтобы найти хорошую скорость обучения η для оптимальной сходимости. Итак, для начала выберем две разные скорости обучения: $\eta = 0.1$ и $\eta = 0.0001$ — и построим график функций потерь в зависимости от количества эпох, чтобы увидеть, насколько хорошо наша реализация алгоритма Adaline учится на обучающих данных.



Гиперпараметры

Скорость обучения η (`eta`), а также количество эпох (`n_iter`) являются так называемыми *гиперпараметрами* (или параметрами настройки) алгоритмов обучения персептрана и Adaline. В главе 6 мы рассмотрим различные методы автоматического поиска значений различных гиперпараметров, обеспечивающих оптимальную производительность модели классификации.

Теперь построим график потерь в зависимости от количества эпох для двух разных скоростей обучения:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.losses_) + 1),
...             np.log10(ada1.losses_), marker='o')
```

```

>>> ax[0].set_xlabel('Эпохи')
>>> ax[0].set_ylabel('log(Среднеквадратичная ошибка)')
>>> ax[0].set_title('Adaline – скорость обучения 0.1')
>>> ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.losses_) + 1),
...             ada2.losses_, marker='o')
>>> ax[1].set_xlabel('Эпохи')
>>> ax[1].set_ylabel('Среднеквадратичная ошибка')
>>> ax[1].set_title('Adaline – скорость обучения 0.0001')
>>> plt.show()

```

Как можно видеть на построенных этим кодом графиках функции потерь (рис. 2.11), мы столкнулись с двумя разными проблемами. На левом графике показано, что может произойти, если мы выберем слишком большую скорость обучения: вместо достижения минимума функции потерь, MSE становится больше с каждой эпохой, потому что мы проскочили глобальный минимум. На правом графике заметно, что потери уменьшаются, но выбранная скорость обучения $\eta = 0.0001$ настолько мала, что алгоритму требуется очень большое количество эпох, чтобы сойтись к глобальному минимуму потерь.

На рис. 2.12 показано, что может произойти при неправильном выборе гиперпараметра и попытке минимизировать функцию потерь L . Левый график демонстрирует пример

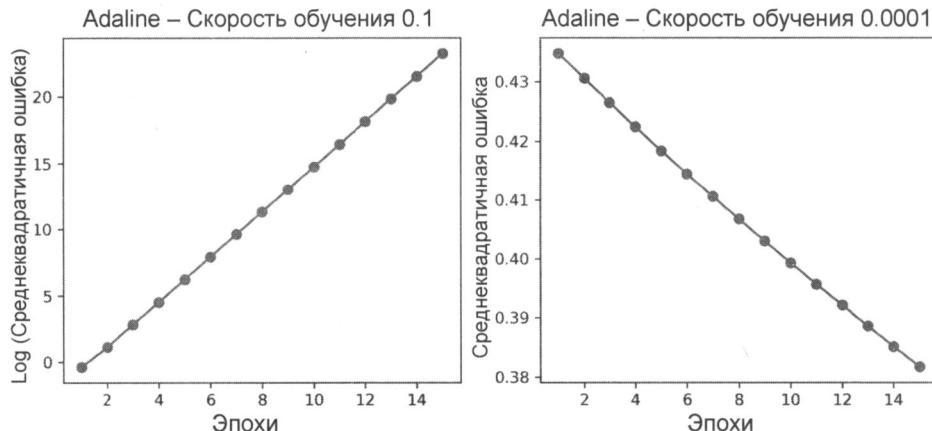


Рис. 2.11. Графики ошибок для неоптимальных скоростей обучения

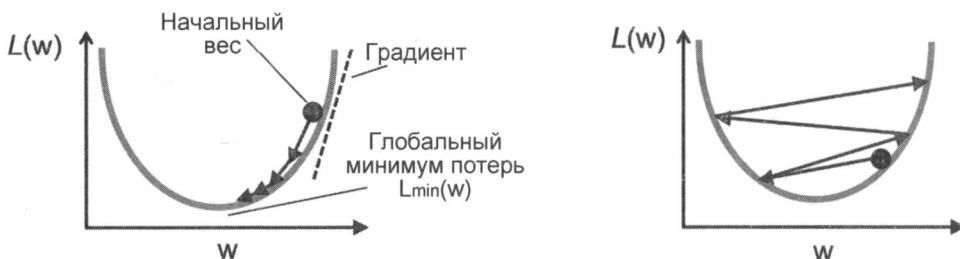


Рис. 2.12. Сравнение правильно выбранной (слева) и слишком большой скорости обучения (справа)

правильно выбранной скорости обучения, когда потери постепенно уменьшаются при движении в направлении глобального минимума, а правый показывает, что произойдет, если выбрать слишком большую скорость обучения, — мы проскочим глобальный минимум.

2.3.3. Улучшение градиентного спуска за счет масштабирования признаков

Многие алгоритмы машинного обучения, с которыми мы столкнемся в этой книге, для достижения оптимальной производительности требуют определенного масштабирования признаков (мы обсудим этот вопрос более подробно в главах 3 и 4). Градиентный спуск — один из многих алгоритмов, которые выигрывают от масштабирования признаков. В этом разделе мы применим метод масштабирования признаков, называемый *стандартизацией* (standardization). Указанная процедура нормализации помогает обучению градиентного спуска сходиться быстрее, но не делает исходный набор данных нормально распределенным. Стандартизация сдвигает среднее значение каждого признака так, чтобы оно было сосредоточено на нуле, и каждый признак имел стандартное отклонение, равное 1 (единичная дисперсия). Например, чтобы стандартизировать j -й признак, достаточно вычесть среднее значение выборки μ_j из каждого обучающего образца и разделить разность на стандартное отклонение σ_j :

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}.$$

Здесь x_j — вектор, состоящий из значений j -го признака всех обучающих образцов n , и этот метод стандартизации применяется к каждому признаку j в наборе данных.

Одна из причин, по которой стандартизация облегчает обучение с градиентным спуском, заключается в том, что с ее помощью проще найти скорость обучения, которая хорошо работает для всех весов (и смещения). Если масштаб признаков слишком сильно различается, скорость обучения, которая хорошо работает для обновления одного веса, может быть слишком большой или слишком маленькой для обновления другого веса. В целом использование стандартизованных признаков может стабилизировать обучение, так что оптимизатору понадобится меньше шагов, чтобы найти хорошее или оптимальное решение (глобальный минимум потерь). На рис. 2.13 показаны возможные обновления градиента с немасштабированными признаками (*слева*) и стандартизованными признаками (*справа*). Концентрические окружности представляют поверхность потерь как функцию двух весов модели в задаче двумерной классификации.

Стандартизация легко выполняется с помощью встроенных методов NumPy `mean` и `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

После стандартизации мы снова обучим Adaline и убедимся, что теперь модель сходится через небольшое количество эпох при скорости обучения $\eta = 0.5$:

```
>>> ada_gd = AdalineGD(n_iter=20, eta=0.5)
>>> ada_gd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
```

```

>>> plt.title('Adaline - градиентный спуск')
>>> plt.xlabel('Длина чашелистика [стандартизирована]')
>>> plt.ylabel('Длина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_gd.losses_) + 1),
...           ada_gd.losses_, marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Среднеквадратичная ошибка')
>>> plt.tight_layout()
>>> plt.show()

```

Выполнив этот код, вы получите визуальное представление областей принятия решений (рис. 2.4, слева), а также график снижения потерь (рис. 2.14, справа).

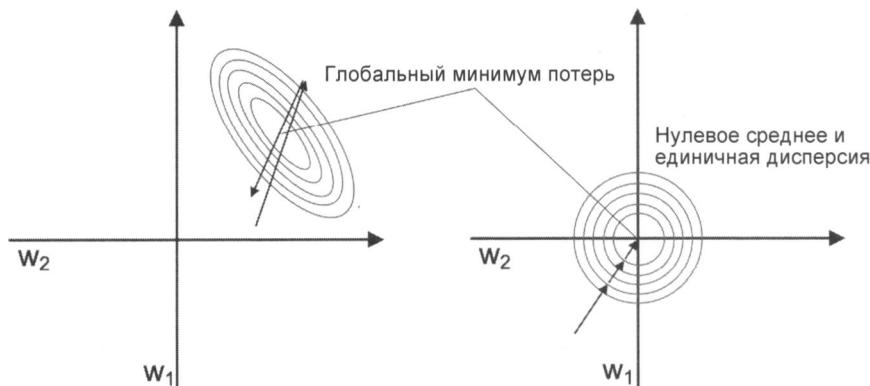


Рис. 2.13. Сравнение немасштабированных (слева) и стандартизованных (справа) признаков при обновлении градиента

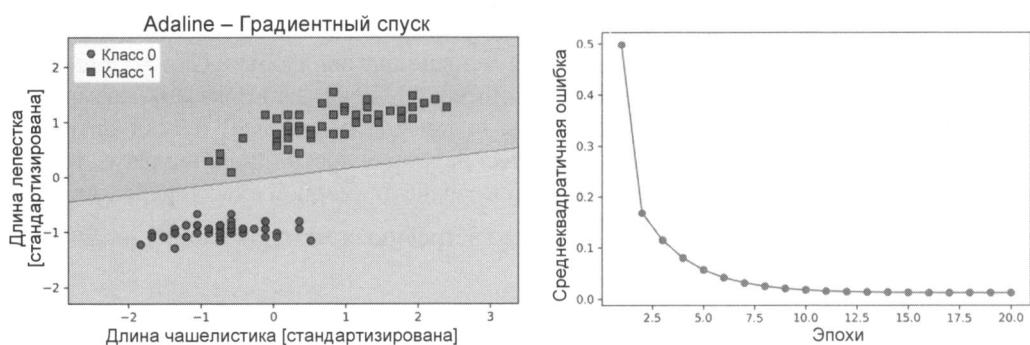


Рис. 2.14. Визуализация областей принятия решений Adaline и среднеквадратичной ошибки (MSE) по количеству эпох

Как следует из приведенных графиков, Adaline сходится после обучения на стандартизованных признаках. Тем не менее значение MSE остается ненулевым, хотя все примеры цветков были классифицированы правильно.

2.3.4. Крупномасштабное машинное обучение и стохастический градиентный спуск

В предыдущем разделе вы узнали, как минимизировать функцию потерь, пошагово двигаясь в направлении, противоположном градиенту потерь, который рассчитывается на основе всего набора обучающих данных; вот почему этот метод обучения иногда также называют *полным пакетным градиентным спуском*. Теперь представьте, что у нас есть очень большой набор данных с миллионами записей (примеров), что нередко встречается во многих приложениях машинного обучения. В этом случае применение пакетного градиентного спуска может сопровождаться огромными вычислительными затратами, поскольку нам придется заново вычислять функцию потерь на полном наборе обучающих данных каждый раз, когда мы делаем один шаг к глобальному минимуму.

Популярной альтернативой алгоритму пакетного градиентного спуска является *стохастический градиентный спуск* (Stochastic Gradient Descent, SGD), который иногда также называют *итеративным*, или *последовательным*, градиентным спуском. Вместо обновления весов исходя из суммы накопленных ошибок по всем обучающим образцам $x^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n} \sum_i (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}$$

мы пошагово обновляем параметры для каждого обучающего примера — например:

$$\Delta w_j = \eta (y^{(i)} - \sigma(z^{(i)})) x_j^{(i)}, \quad \Delta b = \eta (y^{(i)} - \sigma(z^{(i)})).$$

Хотя SGD можно рассматривать как аппроксимацию градиентного спуска, обычно он достигает сходимости намного быстрее из-за более частых обновлений веса. Поскольку каждый градиент рассчитывается на основе одного обучающего примера, поверхность ошибки более зашумлена, чем при градиентном спуске, что фактически является преимуществом, позволяющим SGD легче избегать мелких локальных минимумов, если мы работаем с нелинейными функциями потерь (подробнее об этом рассказано в *главе 11*). Чтобы получить удовлетворительные результаты с помощью SGD, важно представлять обучающие данные в случайном порядке. Кроме того, следует перетасовывать набор обучающих данных для каждой эпохи, чтобы предотвратить циклы.



Регулировка скорости обучения «на ходу»

В практических реализациях SGD фиксированная скорость обучения η часто заменяется адаптивной скоростью, которая со временем уменьшается, — например:

$$\frac{c_1}{[\text{количество итераций}] + c_2},$$

где c_1 и c_2 — константы. Обратите внимание, что SGD не достигает глобального минимума потерь, а находится в области, очень близкой к нему, так что, используя адаптивную скорость обучения, мы можем добиться дальнейшего приближения к минимуму потерь.

Еще одним преимуществом метода SGD является то, что мы можем использовать его для так называемого *последовательного обучения* (online learning). При последователь-

ном обучении наша модель обучается «на лету» по мере поступления новых обучающих данных. Это особенно полезно, если мы постепенно накапливаем большие объемы данных, — например, данные о клиентах в веб-приложениях. Благодаря последовательному обучению, система может немедленно адаптироваться к изменениям, а обучающие данные могут быть удалены после обновления модели, если возникает проблема с местом для хранения.



Мини-пакетный градиентный спуск

Компромиссом между полным пакетным градиентным спуском и SGD может стать так называемый *мини-пакетный градиентный спуск* (mini-batch gradient descent). Его можно рассматривать как применение полнопакетного градиентного спуска к меньшим подмножествам обучающих данных — например, к 32 обучающим образцам за раз. Преимущество мини-пакетного градиентного спуска по сравнению с полнопакетным заключается в более быстрой сходимости из-за более частых обновлений веса. Кроме того, мини-пакетное обучение позволяет нам заменить цикл `for` для перебора обучающих образцов в SGD векторизованными операциями, использующими приемы линейной алгебры (например, реализация взвешенной суммы через скалярное произведение), что может еще больше повысить вычислительную эффективность алгоритма обучения.

Поскольку мы уже реализовали правило обучения Adaline с использованием градиентного спуска, нам осталось лишь внести несколько изменений в код, чтобы адаптировать его для обновления весов по методу SGD. Внутри метода `fit` мы теперь будем обновлять веса после каждого обучающего образца. Кроме того, мы реализуем дополнительный метод `partial_fit` для постепенного обучения, который не выполняет повторную инициализацию весов. Чтобы проверить, сошелся ли наш алгоритм после обучения, мы рассчитаем потери как средние потери на обучающих образцах в каждую эпоху. Кроме того, мы добавим возможность перемешивать обучающие данные перед каждой эпохой, чтобы избежать повторяющихся циклов при оптимизации функции потерь. А с помощью параметра `random_state` можно указывать начальное случайное число для последующей воспроизводимости эксперимента:

```
class AdalineSGD:  
    """Классификатор на аддитивных линейных нейронах.  
  
    Параметры  
    -----  
    eta : float  
        Скорость обучения (между 0.0 и 1.0).  
    n_iter : int  
        Количество проходов по обучающему набору.  
    shuffle : bool (default: True)  
        Перемешивание обучающих данных каждую эпоху, если задано True,  
        для предотвращения возникновения циклов.  
    random_state : int  
        Затравка генератора случайных чисел для инициализации весов  
        случайными значениями.  
    """
```

Атрибуты

w_ : Одномерный массив
Веса после обучения.

b_ : Scalar
Смещение после обучения.

losses_ : list
Значения среднеквадратичной функции потерь после каждой эпохи.

"""

```
def __init__(self, eta=0.01, n_iter=10,  
            shuffle=True, random_state=None):  
    self.eta = eta  
    self.n_iter = n_iter  
    self.w_initialized = False  
    self.shuffle = shuffle  
    self.random_state = random_state
```

```
def fit(self, X, y):  
    """ Подгонка к обучающим данным.
```

Параметры

X : {array-like}, shape = [n_examples, n_features]
Обучающие векторы, где n_examples - количество образцов,
а n_features - количество признаков.

y : array-like, shape = [n_examples]
Целевые переменные.

Возвращаемые значения

self : object

"""

```
    self._initialize_weights(X.shape[1])  
    self.losses_ = []  
    for i in range(self.n_iter):  
        if self.shuffle:  
            X, y = self._shuffle(X, y)  
        losses = []  
        for xi, target in zip(X, y):  
            losses.append(self._update_weights(xi, target))  
        avg_loss = np.mean(losses)  
        self.losses_.append(avg_loss)  
    return self
```

```
def partial_fit(self, X, y):  
    """Подгонка к обучающим данным без повторной инициализации весов"""  
    if not self.w_initialized:  
        self._initialize_weights(X.shape[1])
```

```

if y.ravel().shape[0] > 1:
    for xi, target in zip(X, y):
        self._update_weights(xi, target)
else:
    self._update_weights(X, y)
return self

def _shuffle(self, X, y):
    """Перемешивание обучающих данных"""
    r = self.rgen.permutation(len(y))
    return X[r], y[r]

def _initialize_weights(self, m):
    """Инициализация весов небольшими обучающими данными"""
    self.rgen = np.random.RandomState(self.random_state)
    self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=m)
    self.b_ = np.float_(0.)
    self.w_initialized = True

def _update_weights(self, xi, target):
    """Применение правила Adaline для обновления весов"""
    output = self.activation(self.net_input(xi))
    error = (target - output)
    self.w_ += self.eta * 2.0 * xi * (error)
    self.b_ += self.eta * 2.0 * error
    loss = error**2
    return loss

def net_input(self, X):
    """Вычисление фактического ввода"""
    return np.dot(X, self.w_) + self.b_

def activation(self, X):
    """Вычисление линейной активации"""
    return X

def predict(self, X):
    """Возвращаем метку класса"""
    return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)

```

Метод `_shuffle`, который мы сейчас задействуем в классификаторе `AdalineSGD`, работает следующим образом: с помощью функции `permutation` в `np.random` мы генерируем случайную последовательность уникальных чисел в диапазоне от 0 до 100. Затем эти числа можно использовать в качестве индексов для перемешивания нашей матрицы признаков и вектора метки класса.

Далее мы поключаем метод `fit` — для обучения классификатора `AdalineSGD` и `plot_decision_regions` — для построения графика результатов обучения:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
```

```
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - стохастический градиентный спуск')
>>> plt.xlabel('Длина чашелистика [стандартизирована]')
>>> plt.ylabel('Длина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

>>> plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_, marker='o')
>>> plt.xlabel('Эпохи')
>>> plt.ylabel('Средняя потеря')
>>> plt.tight_layout()
>>> plt.show()
```

Два графика, полученные в результате выполнения этого примера кода, показаны на рис. 2.15.

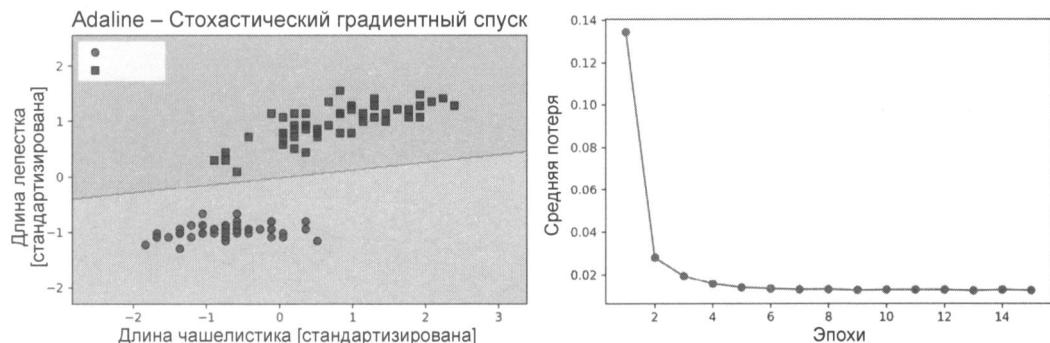


Рис. 2.15. Области принятия решений и график средних потерь после обучения модели Adaline с использованием SGD

Как можно видеть, средние потери снижаются довольно быстро, а окончательная разделяющая граница после 15 эпох выглядит аналогично полученной методом пакетного градиентного спуска Adaline. Если нам нужно обновить модель, например в сценарии постепенного обучения на потоковых данных, мы можем просто вызвать метод `partial_fit` для отдельных обучающих примеров — например: `ada_sgd.partial_fit(X_std[0, :], y[0])`.

2.4. Заключение

В этой главе вы получили представление об основных принципах работы линейных классификаторов, обучаемых с учителем. Следом за реализацией персептрона вы увидели примеры эффективного обучения модели на основе адаптивных линейных нейронов с помощью векторизованной реализации градиентного спуска и постепенного обучения с помощью SGD.

Теперь, когда вы знаете, как работают простые классификаторы на Python, настало время перейти к следующей главе, где мы применим scikit-learn — библиотеку машин-

ного обучения Python, чтобы получить доступ к более продвинутым и мощным классификаторам машинного обучения, которые широко используются в научных кругах, а также в промышленности.

Объектно-ориентированный подход, к которому мы прибегали для реализации алгоритмов персептрона и Adaline, поможет вам понять API scikit-learn, реализованный на основе тех же ключевых методов, что мы использовали в этой главе: `fit` и `predict`. Опираясь на них, мы расскажем о логистической регрессии для моделирования распределения вероятностей классов и методе опорных векторов для работы с нелинейными решающими границами. Кроме того, мы представим другой класс алгоритмов обучения с учителем — древовидные алгоритмы, которые обычно объединяются в робастные ансамблевые классификаторы.

3

Знакомство с классификаторами машинного обучения на основе scikit-learn

https://t.me/it_boooks/2

В этой главе вы познакомитесь с некоторыми популярными и мощными алгоритмами машинного обучения, которые широко используются как в научных кругах, так и в промышленности. Изучая различия между несколькими алгоритмами обучения с учителем для задач классификации, мы также будем оценивать их индивидуальные сильные и слабые стороны. Кроме того, вы начнете работать с библиотекой scikit-learn, которая предлагает удобный и последовательный интерфейс для эффективного и продуктивного использования этих алгоритмов.

Таким образом, здесь будут рассмотрены следующие темы:

- ◆ введение в надежные и популярные алгоритмы классификации — такие как логистическая регрессия, метод опорных векторов, деревья решений и k-ближайшие соседи;
- ◆ примеры и пояснения с использованием библиотеки машинного обучения scikit-learn, которая предоставляет широкий спектр алгоритмов машинного обучения на основе удобного API Python;
- ◆ обсуждение сильных и слабых сторон классификаторов с линейными и нелинейными разделяющими границами.

3.1. Выбор алгоритма классификации

Выбор подходящего алгоритма классификации для конкретной прикладной задачи требует практики и опыта, поскольку каждый алгоритм имеет свои особенности и основан на определенных предположениях. Перефразируя теорему Дэвида Вулпера об отсутствии бесплатных завтраков, можно сказать, что ни один классификатор не работает лучше всех в каждом возможном сценарии¹. На практике всегда рекомендуется сравнивать производительность хотя бы нескольких различных алгоритмов обучения, чтобы выбрать лучшую модель для конкретной задачи, — они могут различаться количеством признаков или обучающих образцов, уровнем шума в наборе данных и линейной разделимостью классов.

В конце концов, производительность классификатора — как вычислительная, так и предсказательная — сильно зависит от исходных обучающих данных. Пять основных

¹ См.: «The Lack of A Priori Distinctions Between Learning Algorithms», D. H. Wolpert, 1996, «No free lunch theorems for optimization», D. H. Wolpert and W. G. Macready, 1997.

этапов обучения алгоритма машинного обучения с учителем можно кратко сформулировать следующим образом:

1. Выбор признаков и сбор помеченных обучающих образцов данных.
2. Выбор показателя производительности.
3. Выбор алгоритма обучения и обучение модели.
4. Оценка производительности модели.
5. Изменение настроек алгоритма и точная настройка модели.

Поскольку подход этой книги состоит в том, чтобы шаг за шагом накапливать знания о машинном обучении, в этой главе мы сосредоточимся на ключевых аспектах различных алгоритмов и вернемся к таким темам, как выбор признаков и предварительная обработка данных, метрики производительности и настройка гиперпараметров, а к более детальному обсуждению этих тем перейдем в следующих главах.

3.2. Первые шаги с scikit-learn: обучение персептрона

В главе 2 вы узнали о двух связанных алгоритмах обучения для задач классификации: *правиле персептрана* и *Adaline*, которые мы самостоятельно реализовали в Python и NumPy. Теперь мы рассмотрим API scikit-learn, который, как уже упоминалось, сочетает в себе удобный и последовательный интерфейс с оптимизированной быстродействующей реализацией нескольких алгоритмов классификации. Библиотека scikit-learn предлагает не только большое разнообразие алгоритмов обучения, но и множество удобных функций для предварительной обработки данных, а также для точной настройки и оценки моделей. Мы рассмотрим эти вопросы более подробно в главах 4 и 5.

Чтобы начать работу с библиотекой scikit-learn, мы обучим модель персептрана, аналогичную той, которую реализовали в главе 2. Для простоты мы воспользуемся уже знакомым нам набором данных Iris. Весьма удобно, что этот простой, но популярный набор, часто применяемый для тестирования и экспериментов с алгоритмами, доступен через scikit-learn. Как и в предыдущей главе, для большей наглядности мы задействуем из набора данных Iris только два признака: присвоим длину и ширину лепестков 150 образцов цветков матрице признаков x , а соответствующие метки классов видов цветков — векторному массиву y :

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
>>> print('Метки класса:', np.unique(y))
Метки класса: [0 1 2]
```

Функция `np.unique(y)` вернула три уникальных метки класса, хранящиеся в `iris.target`, и, как мы видим, имена классов цветков: Iris-setosa, Iris-versicolor и Iris-virginica — уже сохранены как целые числа (здесь: 0, 1, 2). Хотя многие функции scikit-learn и ме-

тоды класса также работают с метками классов в строковом формате, рекомендуется использовать целочисленные метки, что помогает избежать технических сбоев и повысить быстродействие вычислений за счет меньшего объема привлеченной памяти. Кроме того, кодирование меток классов в виде целых чисел является распространенным соглашением среди большинства библиотек машинного обучения.

Чтобы оценить, насколько хорошо обученная модель работает с незнакомыми данными, мы дополнительно разделим набор данных на непересекающиеся обучающие и тестовые наборы. В *главе 6* мы более подробно обсудим лучшие способы оценки модели, а пока, используя функцию `train_test_split` из модуля `scikit-learn model_selection`, случайным образом разделим массивы `x` и `y` на 30% тестовых данных (45 примеров) и 70% обучающих данных (105 примеров):

```
>>> from sklearn.model_selection import train_test_split  
>>> X_train, X_test, y_train, y_test = train_test_split(  
... X, y, test_size=0.3, random_state=1, stratify=y  
... )
```

Обратите внимание, что функция `train_test_split` выполняет перемешивание обучающих данных перед разделением, — в противном случае все примеры из классов 0 и 1 оказались бы в обучающих наборах данных, а тестовый набор данных состоял бы из 45 примеров класса 2. С помощью параметра `random_state` мы передали фиксированное случайное начальное число (`random_state=1`) во внутренний генератор псевдослучайных чисел, который служит для перетасовки наборов данных перед разделением. Применение фиксированного значения `random_state` гарантирует воспроизводимость результатов.

Наконец, воспользуемся встроенной поддержкой стратификации, задав параметр `stratify=y`. В нашем случае стратификация означает, что метод `train_test_split` возвращает обучающие и тестовые поднаборы, которые имеют такое же соотношение меток классов, что и входной набор данных. Мы можем прибегнуть к функции `bincount` библиотеки NumPy, которая подсчитывает количество вхождений каждого значения в массиве, чтобы убедиться, что это действительно так:

```
>>> print('Количество меток в y:', np.bincount(y))  
Количество меток в y: [50 50 50]  
>>> print('Количество меток в y_train:', np.bincount(y_train))  
Количество меток в y_train: [35 35 35]  
>>> print(' Количество меток в y_test:', np.bincount(y_test))  
Количество меток в y_test: [15 15 15]
```

Многие алгоритмы машинного обучения и оптимизации также требуют масштабирования признаков для наилучшей производительности, как вы видели в примере с градиентным спуском в *главе 2*. Стандартизуем признаки, используя класс `StandardScaler` из модуля `preprocessing scikit-learn`:

```
>>> from sklearn.preprocessing import StandardScaler  
>>> sc = StandardScaler()  
>>> sc.fit(X_train)  
>>> X_train_std = sc.transform(X_train)  
>>> X_test_std = sc.transform(X_test)
```

С помощью этого кода мы загрузили класс `StandardScaler` из модуля `preprocessing` и инициализировали новый объект `StandardScaler`, который присвоили переменной `sc`. Используя метод `fit`, `StandardScaler` оценил параметры μ (выборочное среднее) и σ (стандартное отклонение) для каждого измерения признака из обучающих данных. Вызвав метод `transform`, мы затем стандартизировали обучающие данные, применив указанные оценочные параметры μ и σ . Не забывайте, что необходимо задействовать такие же параметры масштабирования для стандартизации тестового набора данных, чтобы значения в обучающем и тестовом наборах данных были сопоставимы друг с другом.

После стандартизации обучающих данных можно приступить к обучению модели персептрона. Большинство алгоритмов в `scikit-learn` по умолчанию поддерживают многоклассовую классификацию с помощью метода «один против остальных» (One-versus-Rest, OvR), позволяющего нам сразу передать персепtronу все три класса цветков. Код выглядит следующим образом:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

Интерфейс `scikit-learn` напоминает реализацию персептрона в главе 2. После загрузки класса `Perceptron` из модуля `linear_model` мы инициализируем новый объект `Perceptron` и обучаем модель с помощью метода `fit`. Здесь параметр модели `eta0` эквивалентен скорости обучения `eta`, которую мы использовали в собственной реализации персептрона.

Как вы помните из главы 2, подходящую скорость обучения необходимо подбирать опытным путем: если скорость обучения слишком велика, алгоритм может проскочить глобальный минимум потерь, а если она слишком мала, алгоритму потребуется намного больше эпох, чтобы сойтись, что может замедлить обучение, особенно на больших наборах данных. Кроме того, мы задействовали параметр `random_state`, чтобы обеспечить воспроизводимость начальной перетасовки обучающего набора данных после каждой эпохи.

Обучив модель в `scikit-learn`, мы можем делать прогнозы с помощью метода `predict` точно так же, как в нашей реализации персептрона в главе 2. Код выглядит следующим образом:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print(' Ошибочно классифицированы: %d' % (y_test != y_pred).sum())
Ошибочно классифицированы: 1
```

Выполнив этот код, мы увидим, что персептрон неправильно классифицирует 1 из 45 примеров цветов. Следовательно, ошибка неправильной классификации в тестовом наборе данных составляет приблизительно 0.022, или 2.2% ($1/45 \approx 0.022$).



Ошибка классификации или точность?

Вместо ошибки неправильной классификации многие специалисты по машинному обучению сообщают о точности (`accuracy`) классификации модели, которая просто рассчитывается следующим образом:

$$1 - \text{ошибка} = 0.978, \text{ или } 97.8\%.$$

Что лучше использовать, ошибку классификации или точность, — это лишь вопрос предпочтения.

Стоит отметить, что scikit-learn также предлагает реализацию большого количества различных показателей производительности, доступных через модуль `metrics`. Например, мы можем рассчитать точность классификации персептрона на тестовом наборе данных следующим образом:

```
>>> from sklearn.metrics import accuracy_score  
>>> print('Точность: %.3f' % accuracy_score(y_test, y_pred))  
Точность: 0.978
```

Здесь `y_test` — это истинные метки классов, а `y_pred` — метки классов, которые мы предсказали ранее. Кроме того, у каждого классификатора в scikit-learn есть метод `score`, который вычисляет точность предсказания классификатора, комбинируя вызов методов `predict` и `accuracy_score`:

```
>>> print('Точность: %.3f' % ppn.score(X_test_std, y_test))  
Точность: 0.978
```



Переобучение

В этой главе мы будем оценивать производительность моделей на основе тестового набора данных. В главе 6 вы узнаете о полезных методах, включающих графический анализ, — таких как кривые обучения для обнаружения и предотвращения переобучения. *Переобучение*, к которому мы вернемся позже в этой главе, означает, что модель хорошо выявляет скрытые закономерности в обучающих данных, но не может хорошо сделать то же самое с незнакомыми данными.

Наконец, воспользуемся нашей функцией `plot_decision_regions` из главы 2, чтобы построить *области решений* (`decision region`) нашей недавно обученной модели персептрона и визуально продемонстрировать, насколько хорошо она разделяет различные образцы цветков. Однако сначала внесем в функцию небольшую доработку, чтобы выделить экземпляры из тестового набора данных маленькими кружками:

```
from matplotlib.colors import ListedColormap  
import matplotlib.pyplot as plt  
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):  
    # настройка генератора меток и цветовой карты  
    markers = ('o', 's', '^', 'v', '<')  
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')  
    cmap = ListedColormap(colors[:len(np.unique(y))])  
  
    # строим поверхность решений  
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1  
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1  
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),  
                          np.arange(x2_min, x2_max, resolution))  
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)  
    lab = lab.reshape(xx1.shape)  
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)  
    plt.xlim(xx1.min(), xx1.max())  
    plt.ylim(xx2.min(), xx2.max())  
  
    # отображаем образцы класса  
    for idx, cl in enumerate(np.unique(y)):
```

```

plt.scatter(x=X[y == cl, 0],
            y=X[y == cl, 1],
            alpha=0.8,
            c=colors[idx],
            marker=markers[idx],
            label=f'Class {cl}',
            edgecolor='black')

# выделяем тестовые образцы
if test_idx:
    # отобразить все образцы
    X_test, y_test = X[test_idx, :], y[test_idx]

plt.scatter(X_test[:, 0], X_test[:, 1],
            c='none', edgecolor='black', alpha=1.0,
            linewidth=1, marker='o',
            s=100, label='Test set')

```

После внесения изменений в функцию `plot_decision_regions` мы можем указать индексы образцов, которые мы хотим отметить на результирующих графиках. Код выглядит следующим образом:

```

>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                         y=y_combined,
...                         classifier=ppn,
...                         test_idx=range(105, 150))
>>> plt.xlabel('Длина лепестка [стандартизованная]')
>>> plt.ylabel('Ширина лепестка [стандартизованная]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()

```

На полученном графике хорошо видно, что три класса цветков не могут быть идеально разделены линейной границей (рис. 3.1).

Как вы помните, в главе 2 было сказано, что алгоритм персептрона никогда не сходится на наборах данных, которые не обладают идеальной линейной разделимостью, поэтому использование алгоритма персептрона на практике обычно не рекомендуется. Далее в этой главе мы рассмотрим более мощные линейные классификаторы, которые сходятся к минимуму потерь, даже если классы не являются идеально линейно разделимыми.



Дополнительные настройки персептрана

Класс `Perceptron`, как и другие функции и классы scikit-learn, часто имеет дополнительные параметры, которые мы опускаем для простоты. Вы можете узнать больше об этих параметрах, используя функцию справки в Python (например, `help(Perceptron)`) или просмотрев отличную онлайн-документацию scikit-learn по адресу: <http://scikit-learn.org/stable/>.

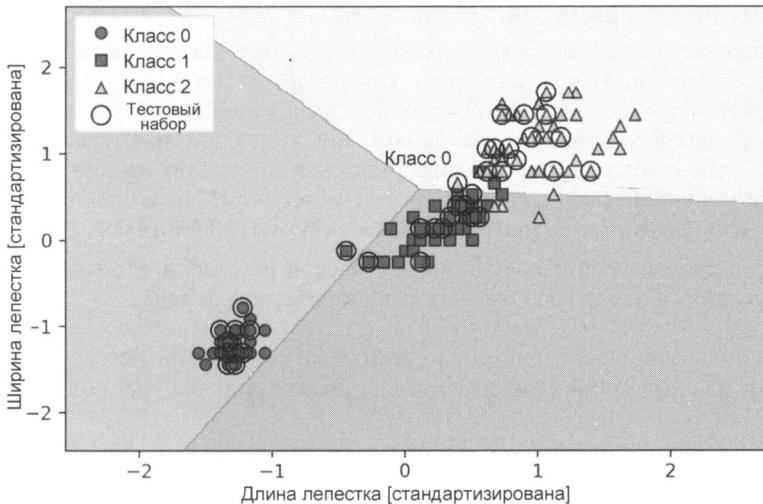


Рис. 3.1. Разделяющие границы многоклассовой модели персептрана, обученной на наборе данных Iris

3.3. Моделирование вероятностей классов с помощью логистической регрессии

Хотя правило персептрана представляет собой хорошее и простое введение в алгоритмы машинного обучения для задач классификации, его самый большой недостаток заключается в том, что персептрон никогда не сходится, если классы не обладают идеальной линейной разделимостью. Примером такого сценария может служить задача классификации, рассмотренная в предыдущем разделе. Причина отсутствия сходимости заключается в постоянном обновлении весов модели, поскольку в каждой эпохе всегда присутствует как минимум один неправильно классифицированный обучающий образец. Конечно, вы можете изменить скорость обучения и увеличить количество эпох, но имейте в виду, что персептрон никогда не сойдется на этом наборе данных.

Чтобы не терять напрасно время, мы сейчас рассмотрим еще один простой, но более мощный алгоритм для задач линейной и бинарной классификации: *логистическую регрессию* (logistic regression). Важное замечание — несмотря на свое название, логистическая регрессия является моделью для задач классификации, а не регрессии.

3.3.1. Логистическая регрессия и условные вероятности

Логистическая регрессия — это модель классификаций, которую очень легко реализовать, и при этом она прекрасно работает на линейно разделимых классах. Это один из наиболее широко используемых алгоритмов классификации в промышленности. Подобно персептрану и Adaline, модель логистической регрессии, о которой мы расскажем в этой главе, также является линейной моделью для бинарной классификации.



Логистическая регрессия для нескольких классов

Обобщение логистической регрессии на задачи с несколькими классами известно как *мультиномиальная логистическая регрессия* (*multinomial logistic regression*), или *регрессия softmax*. Более подробное рассмотрение мультиномиальной логистической регрессии выходит за рамки этой книги, но заинтересованный читатель может найти дополнительную информацию в конспектах лекций по адресу: https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L08_logistic_slides.pdf или здесь: <https://www.youtube.com/watch?v=L0FU8NFpx4E>.

Другой способ использования логистической регрессии для многоклассовой классификации — это метод OvR, который мы обсуждали ранее.

Чтобы объяснить основные принципы работы логистической регрессии как вероятностной модели для бинарной классификации, давайте сначала рассмотрим понятие *шанса* как вероятности в пользу определенного события. Шансы можно записать как $\frac{p}{1-p}$,

где p — вероятность положительного события. Термин «положительное событие» не обязательно означает что-то «хорошее» — он просто относится к событию, которое мы хотим предсказать. Например, это может быть вероятность того, что у пациента есть определенное заболевание при определенных симптомах; мы можем рассматривать положительное событие как метку класса $y = 1$, а симптомы — как признаки x . Следовательно, для краткости мы можем определить вероятность p как $p := p(y = 1|x)$ — условную вероятность того, что конкретный пример принадлежит к определенному классу 1 при наличии признаков x .

Затем мы можем дополнительно определить *логит-функцию* (или просто *логит*, *logit*), которая представляет собой логарифм отношения вероятностей:

$$\text{logit}(p) = \log \frac{p}{(1-p)}.$$

Здесь *log* означает *натуральный логарифм*, поскольку это общепринятое соглашение в информатике. Логит-функция принимает входные значения в диапазоне от 0 до 1 и преобразует их в значения, лежащие во всем диапазоне действительных чисел.

В рамках логистической модели мы предполагаем, что существует линейная зависимость между взвешенными входными данными (мы называли их *фактическими* входными данными в главе 2) и логарифмом отношения вероятностей:

$$\text{logit}(p) = w_1x_1 + \dots + w_mx_m + b = \sum_{i=1}^m w_i x_i + b = \mathbf{w}^T \mathbf{x} + b.$$

В то время как это уравнение описывает наше предположение о линейной зависимости между логарифмом отношения вероятностей и фактическими входными данными, на самом деле нас интересует p — вероятность принадлежности образца к определенному классу при заданных признаках. Поскольку логит-функция отображает вероятность в диапазон действительных чисел, мы можем взять обратную функцию, чтобы отобразить диапазон действительных чисел обратно — в диапазон [0, 1] для вероятности p .

Функцию, обратную по отношению к логит-функции, обычно называют *логистической сигмоидной функцией* (*logistic sigmoid function*). Иногда это название сокращают до *сигмоидной функции* из-за ее характерной S-образной формы:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Здесь z — это фактические входные данные, представляющие собой линейную комбинацию весов и входных данных (т. е. признаки, связанные с обучающими примерами):

$$z = \mathbf{w}^T \mathbf{x} + b.$$

Теперь давайте просто построим сигмоидную функцию для некоторых значений в диапазоне от -7 до 7 , чтобы увидеть, как она выглядит:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> sigma_z = sigmoid(z)
>>> plt.plot(z, sigma_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\sigma(z)$')
>>> # деления оси у и линии сетки
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

В результате выполнения этого кода вы должны увидеть S-образную (сигмоидную) кривую (рис. 3.2).

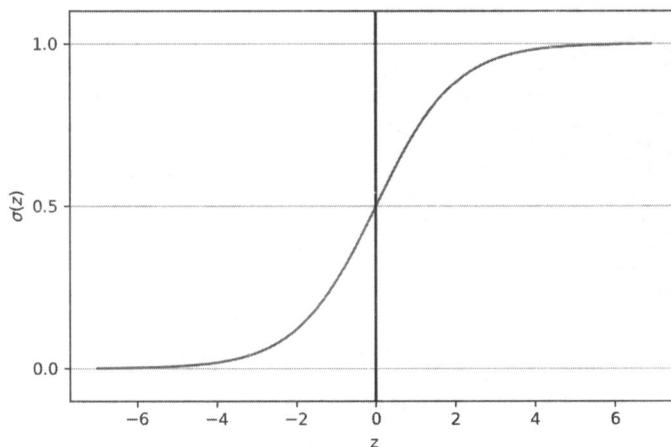


Рис. 3.2. График логистической сигмоидной функции

Здесь мы можем видеть, что $\sigma(z)$ приближается к 1 , если z стремится к бесконечности ($z \rightarrow \infty$), поскольку e^{-z} становится очень малым при больших значениях z . Точно так же $\sigma(z)$ стремится к 0 при $z \rightarrow -\infty$ в результате увеличения знаменателя. Следовательно, мы можем заключить, что эта сигмоидная функция принимает на вход вещественные числа и преобразует их в значения в диапазоне $[0, 1]$ с пересечением в точке $\sigma(0) = 0.5$.

Чтобы получить некоторое представление о модели логистической регрессии, нужно вспомнить, о чем мы говорили в главе 2. Так, в Adaline мы использовали тождественную функцию $\sigma(z) = z$ в качестве функции активации. В логистической регрессии эта функция активации просто становится сигмоидной функцией, о которой мы только что говорили.

Различие между Adaline и логистической регрессией показано на рис. 3.3, где единственная разница — это функция активации.

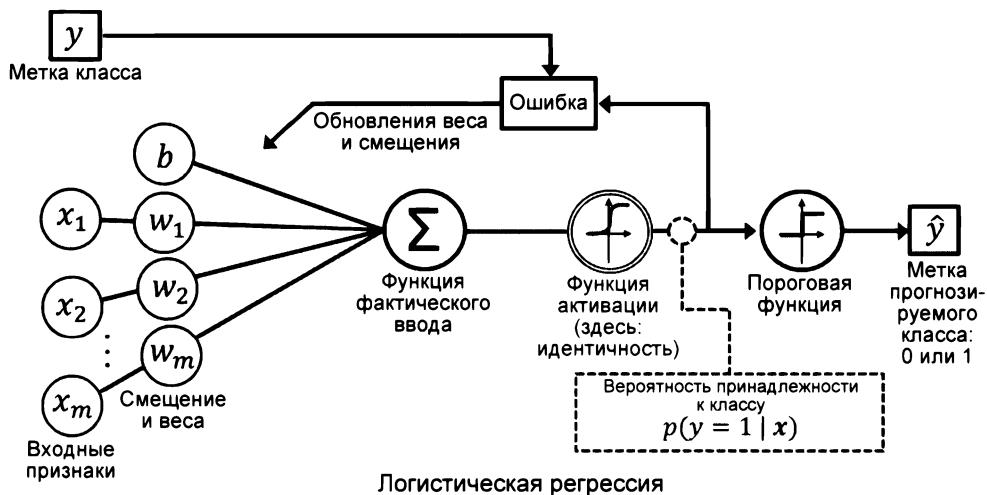
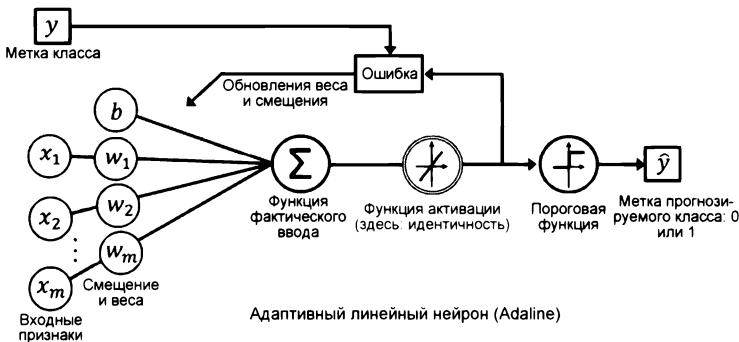


Рис. 3.3. Логистическая регрессия по сравнению с Adaline

Выход сигмоидной функции затем интерпретируется как вероятность того, что конкретный образец принадлежит классу 1, $\sigma(z) = p(y = 1 | x; w, b)$ с учетом его признаков x и параметризации весами w и смещением b . Например, если мы нашли, что для конкретного образца цветка $\sigma(z) = 0.8$, это означает, что вероятность принадлежности этого образца к классу Iris-versicolor составляет 80%. Следовательно, вероятность того, что этот цветок относится к классу Iris-setosa, можно рассчитать как $p(y = 0 | x; w, b) = 1 - p(y = 1 | x; w, b) = 0.2$, или 20%.

Предсказанная вероятность затем может быть просто преобразована в бинарный вывод с помощью пороговой функции:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{в ином случае} \end{cases}.$$

Если мы посмотрим на предыдущий график сигмоидной функции, это эквивалентно следующему условному выражению:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{в ином случае} \end{cases}.$$

На самом деле, есть много приложений, где нас интересуют не только предсказанные метки классов, но и важна оценка вероятности принадлежности к классу (вывод сигмоидной функции до применения пороговой функции). Логистическая регрессия используется в прогнозировании погоды, например не только для того, чтобы предсказать, будет ли дождь в конкретный день, но и сообщить о вероятности дождя. Аналогичным образом логистическую регрессию можно применять для прогнозирования вероятности того, что у пациента есть определенное заболевание, исходя из наличия тех или иных симптомов, поэтому логистическая регрессия пользуется большой популярностью в области медицины.

3.3.2. Изучение весов модели с помощью логистической функции потерь

Вы узнали, как модель логистической регрессии применяют для прогнозирования вероятностей и меток классов, теперь мы кратко поговорим о подгонке параметров модели — например, весов и смещения w и b . В предыдущей главе мы определили функцию потерь, вычисляемую через среднеквадратичную ошибку, следующим образом:

$$L(w, b | \mathbf{x}) = \sum_i \frac{1}{2} (\sigma(z^{(i)}) - y^{(i)})^2.$$

Мы минимизировали эту функцию, чтобы узнать параметры нашей модели классификации Adaline. Чтобы объяснить, как получают функцию потерь для логистической регрессии, давайте сначала определим *правдоподобие* \mathcal{L} , которое мы стремимся максимизировать при построении модели логистической регрессии, предполагая, что отдельные экземпляры в нашем наборе данных независимы друг от друга. Формула выглядит следующим образом:

$$\mathcal{L}(w, b | \mathbf{x}) = p(y | \mathbf{x}; w, b) = \prod_{i=1}^n p(y^{(i)} | \mathbf{x}^{(i)}; w, b) = \prod_{i=1}^n (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}.$$

На практике проще максимизировать (натуральный) логарифм этого уравнения, который называется *функцией логарифмического правдоподобия* (log-likelihood function):

$$l(w, b | \mathbf{x}) = \log \mathcal{L}(w, b | \mathbf{x}) = \sum_{i=1}^n [y^{(i)} \log(\sigma(z^{(i)})) + (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))].$$

Во-первых, применение логарифмической функции снижает вероятность *машинного обнуления* (numerical underflow), которое может произойти, если вероятность очень мала. Во-вторых, мы можем преобразовать произведение множителей в сумму множите-

лей, что упрощает получение производной этой функции с помощью приема сложения в соответствии с основами математического анализа.



Вывод функции правдоподобия

Мы можем получить выражение для вычисления правдоподобия модели при заданных данных $\mathcal{L}(w, b | x)$ следующим образом. Если нам дана задача двоичной классификации с метками классов 0 и 1, мы можем рассматривать метку 1 как переменную Бернулли — она может принимать два значения: 0 и 1, где вероятность p того, что это 1, следующая: $Y \sim Bern(p)$. Для одиночной точки данных мы можем написать эту вероятность следующим образом:

$$P(Y = 1 | X = x^{(i)}) = \sigma(z^{(i)}) \text{ и } P(Y = 0 | X = x^{(i)}) = 1 - \sigma(z^{(i)}).$$

Объединив эти два выражения и используя сокращение:

$$P(Y = y^{(i)} | X = x^{(i)}) = p(y^{(i)} | x^{(i)}),$$

получим функцию распределения вероятностей переменной Бернулли:

$$p(y^{(i)} | x^{(i)}) = (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}.$$

Мы можем записать вероятность обучающих меток, исходя из предположения, что все обучающие примеры независимы, и используя правило умножения вероятностей для вычисления вероятности того, что все события произойдут, следующим образом:

$$\mathcal{L}(w, b | x) = \prod_{i=1}^n p(y^{(i)} | x^{(i)}; w, b).$$

Далее, подставляя функцию распределения вероятностей переменной Бернулли, приходим к выражению правдоподобия, которое пытаемся максимизировать, изменяя параметры модели:

$$\mathcal{L}(w, b | x) = \prod_{i=1}^n (\sigma(z^{(i)}))^{y^{(i)}} (1 - \sigma(z^{(i)}))^{1-y^{(i)}}.$$

Теперь можно использовать алгоритм оптимизации — такой как градиентный подъем, чтобы максимизировать эту функцию логарифмического правдоподобия. (Градиентный подъем работает точно так же, как градиентный спуск, описанный в главе 2, за исключением того, что градиентный подъем максимизирует функцию, а не минимизирует ее.) В качестве альтернативы давайте перепишем логарифмическую вероятность как функцию потерь L , которую можно минимизировать с помощью градиентного спуска, как в главе 2:

$$L(w, b) = \sum_{i=1}^n [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))].$$

Чтобы лучше разобраться с этой функцией потерь, рассчитаем потери для одного обучающего образца:

$$L(\sigma(z), y; w, b) = -y \log(\sigma(z)) - (1 - y) \log(1 - \sigma(z)).$$

Из этого уравнения следует, что первый член становится равным нулю, если $y = 0$, а второй член становится равным нулю, если $y = 1$:

$$L(\sigma(z), y; w, b) = \begin{cases} -\log(\sigma(z)) & \text{if } y = 1 \\ -\log(1 - \sigma(z)) & \text{if } y = 0 \end{cases}$$

Напишем короткий блок кода для построения графика, иллюстрирующего зависимость потери при классификации одного обучающего образца от разных значений $\sigma(z)$:

```
>>> def loss_1(z):
...     return - np.log(sigmoid(z))
>>> def loss_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> sigma_z = sigmoid(z)
>>> c1 = [loss_1(x) for x in z]
>>> plt.plot(sigma_z, c1, label='L(w, b) if y=1')
>>> c0 = [loss_0(x) for x in z]
>>> plt.plot(sigma_z, c0, linestyle='--', label='L(w, b) if y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\sigma(z)$')
>>> plt.ylabel('L(w, b)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Полученный график отображает значения сигмоидной функции активации по оси x в диапазоне от 0 до 1 (входными данными для сигмоидной функции были значения z в диапазоне от -10 до 10) и соответствующие логистические потери по оси y (рис. 3.4).

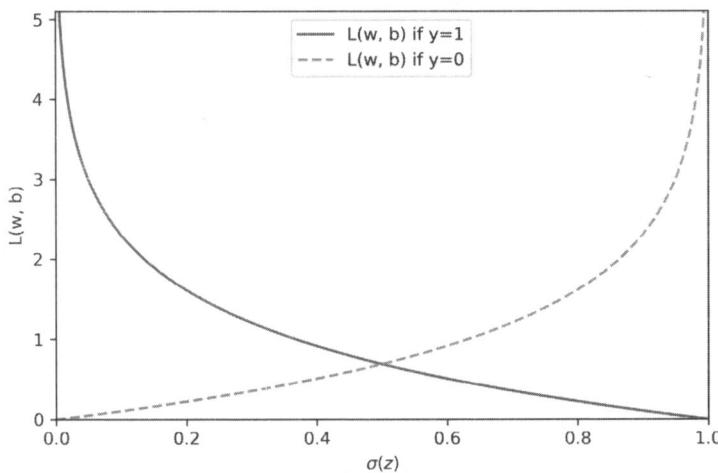


Рис. 3.4. График функции потерь, используемой в логистической регрессии

На графике мы видим, что потери приближаются к 0 (*сплошная линия*), если мы правильно предсказываем, что образец относится к классу 1. Точно так же потери приближаются к 0, если мы правильно предсказываем $y = 0$ (*пунктирная линия*). Однако если

прогноз неверен, потери стремятся к бесконечности. Дело в том, что мы наказываем неправильные прогнозы все более крупными потерями.

3.3.3. Преобразование Adaline в алгоритм логистической регрессии

Если бы нам пришлось самостоятельно заниматься реализацией логистической регрессии, то было бы достаточно просто заменить функцию потерь L в нашей реализации Adaline из главы 2 новой функцией потерь:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))].$$

Воспользуемся этим выражением для вычисления потерь классификации всех обучающих примеров за эпоху. Нам также нужно поменять местами линейную функцию активации и сигмоиду. Если мы внесем эти изменения в код Adaline, то получим работающую реализацию логистической регрессии. Далее приведена реализация для полнопакетного градиентного спуска (но обратите внимание, что те же изменения могут быть внесены и в версию стохастического градиентного спуска):

```
class LogisticRegressionGD:
    """Классификатор по методу логистической регрессии на основе градиентного спуска.

    Параметры
    -----
    eta : float
        Скорость обучения (между 0.0 и 1.0).
    n_iter : int
        Количество проходов по обучающему набору.
    random_state : int
        Затравка генератора случайных чисел для инициализации весов
        случайными значениями.

    Атрибуты
    -----
    w_ : одномерный массив
        Веса после обучения.
    b_ : Scalar
        Смещение после обучения.
    losses_ : list
        Значения среднеквадратичной функции потерь после каждой эпохи.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
        """Подгонка к обучающим данным.
```

```
Параметры
-----
X : {array-like}, shape = [n_examples, n_features]
    Обучающие векторы, где n_examples - количество образцов,
    а n_features - количество признаков.
y : array-like, shape = [n_examples]
    Целевые переменные.

Возвращаемые значения
-----
self : Экземпляр LogisticRegressionGD

"""
rgen = np.random.RandomState(self.random_state)
self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
self.b_ = np.float_(0.)
self.losses_ = []

for i in range(self.n_iter):
    net_input = self.net_input(X)
    output = self.activation(net_input)
    errors = (y - output)
    self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
    self.b_ += self.eta * 2.0 * errors.mean()
    loss = (-y.dot(np.log(output))
            - ((1 - y).dot(np.log(1 - output)))
            / X.shape[0])
    self.losses_.append(loss)
return self

def net_input(self, X):
    """ Вычисление фактического ввода """
    return np.dot(X, self.w_) + self.b_

def activation(self, z):
    """Вычисление логистической сигмоидной активации"""
    return 1. / (1. + np.exp(-np.clip(z, -250, 250)))

def predict(self, X):
    """Возвращаем метку класса"""
    return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

Собираясь обучить модель логистической регрессии, мы должны помнить, что она работает лишь для задач бинарной классификации.

Поэтому мы рассмотрим только цветки setosa и versicolor (классы 0 и 1) и убедимся, что наша реализация логистической регрессии работает:

```
>>> X_train_01_subset = X_train_std[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
```

```
>>> lrgd = LogisticRegressionGD(eta=0.3,
...                                n_iter=1000,
...                                random_state=1)
>>> lrgd.fit(X_train_01_subset,
...             y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                         y=y_train_01_subset,
...                         classifier=lrgd)
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Получившийся график области принятия решений показан на рис. 3.5.

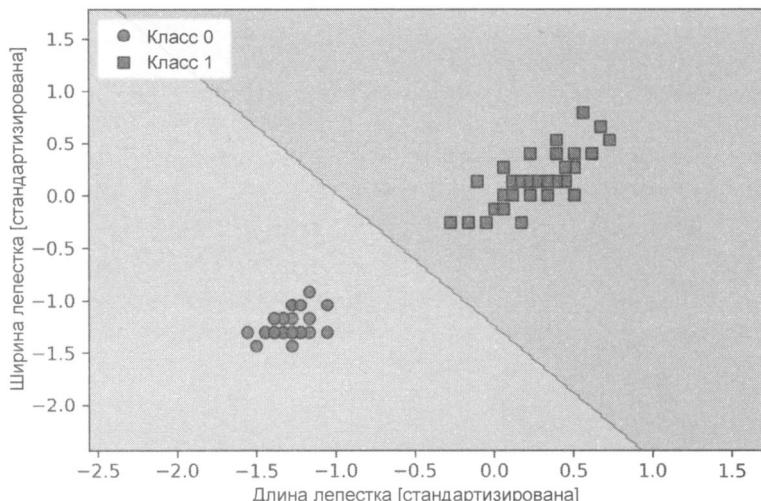


Рис. 3.5. График области принятия решений для модели логистической регрессии



Алгоритм обучения методом градиентного спуска для логистической регрессии

Если вы сравните `LogisticRegressionGD` в приведенном только что коде с кодом `AdalineGD` из главы 2, то сможете заметить, что правила обновления веса и смещения остались неизменными (за исключением коэффициента масштабирования 2). Можно легко показать, что механизмы обновления параметров с помощью градиентного спуска у логистической регрессии и `Adaline` действительно похожи. Впрочем, следующий вывод правила обучения методом градиентного спуска предназначен для читателей, которые интересуются математической теорией, лежащей в основе указанного правила. Так что не обязательно знать этот вывод, чтобы продолжить чтение главы.

На рис. 3.6 показано, как происходит вычисление частной производной логарифмической функции правдоподобия по j -му весу.

$$\frac{\partial L}{\partial w_j} = \underbrace{\frac{\partial L}{\partial a} \frac{\partial a}{\partial z} \frac{\partial z}{\partial w_j}}_{\text{Применить цепное правило}}, \quad \text{где } a = \sigma(z) = \frac{1}{1 + e^{-z}}$$

1. Находим производные по отдельности:

$$\left. \begin{aligned} \frac{\partial L}{\partial a} &= \frac{a - y}{a \cdot (1 - a)} \\ \frac{\partial a}{\partial z} &= \frac{e^{-z}}{(1 + e^{-z})^2} = a \cdot (1 - a) \\ \frac{\partial z}{\partial w_j} &= x_j \end{aligned} \right\}$$

2. Применяем цепное правило и упрощаем:

$$\left. \begin{aligned} \frac{\partial L}{\partial z} &= a - y \\ \frac{\partial L}{\partial w_j} &= (a - y)x_j \\ &= -(y - a)x_j \end{aligned} \right\}$$

Рис. 3.6. Расчет частной производной логарифмической функции правдоподобия

Обратите внимание, что для краткости мы опустили усреднение по обучающим примерам.

В главе 2 было сказано, что мы делаем шаги в направлении, противоположном градиенту. Следовательно, мы переворачиваем $\frac{\partial L}{\partial w_j} = -(y - a)x_j$, и обновляем j -й вес следующим образом, включая скорость обучения η :

$$w_j := w_j + \eta(y - a)x_j$$

Хотя частная производная функции потерь по смещению не показана, вывод смещения основан на том же подходе с использованием цепного правила, что приводит нас к следующему правилу обновления:

$$b := b + \eta(y - a)x_j.$$

Обновления веса и смещения такие же, как у Adaline в главе 2.

3.3.4. Обучение модели логистической регрессии с помощью scikit-learn

Итак, в предыдущем разделе вы выполнили полезные упражнения по программированию и математике, которые иллюстрируют концептуальные различия между Adaline и логистической регрессией. Теперь давайте узнаем, как использовать более оптимизированную реализацию логистической регрессии scikit-learn, которая изначально поддерживает многоклассовую классификацию. Обратите внимание, что в последних версиях scikit-learn метод многоклассовой классификации — мультиномиальный или OvR — выбирается автоматически. В следующем примере кода мы применим класс `sklearn.linear_model.LogisticRegression`, а также знакомый метод `fit` для обучения модели на всех трех классах цветков, входящих в стандартизированный обучающий набор. Кроме того, для наглядности мы установили параметр `multi_class='ovr'`. В качестве самостоятельного упражнения вы можете сравнить результаты с `multi_class='multinomial'`. Учтите, что параметр `multinomial` теперь является выбором по умолчанию в классе `LogisticRegression`.

библиотеки scikit-learn и рекомендуется на практике для взаимоисключающих классов, таких как классы цветков в наборе данных Iris. Здесь «взаимоисключающий» означает, что каждый обучающий образец может принадлежать только одному классу (в отличие от классификации с несколькими метками, где обучающий образец может быть членом нескольких классов).

Теперь выполним следующий пример кода:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, solver='lbfgs',
...                         multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                        y_combined,
...                        classifier=lr,
...                        test_idx=range(105, 150))
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

После обучения модели на имеющемся наборе данных этот код выводит на график области принятия решений обучающие и тестовые образцы, как показано на рис. 3.7.

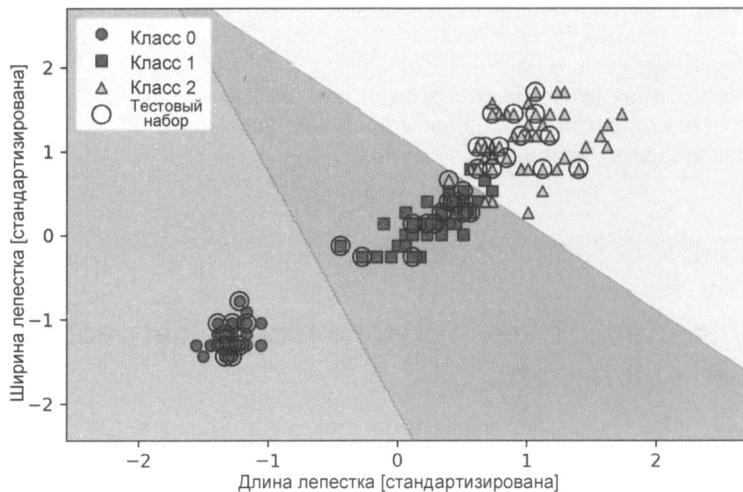


Рис. 3.7. Области принятия решений для модели многоклассовой логистической регрессии scikit-learn



Алгоритмы выпуклой оптимизации

Отметим, что существует много различных алгоритмов решения задач оптимизации. Для минимизации выпуклых функций потерь, таких как потери логистической регрессии, рекомендуется использовать более продвинутые подходы, чем обычный стохастический градиентный спуск. На самом деле scikit-learn поддерживает целый ряд подобных алгоритмов оптимизации, которые можно выбрать через параметр решателя, а именно: 'newton-cg', 'lbfgs', 'liblinear', 'sag' и 'saga'.

Хотя функция потерь логистической регрессии является выпуклой, большинство алгоритмов оптимизации должны легко сходиться к минимуму глобальных потерь. Однако существуют определенные преимущества использования одного алгоритма перед другим. Так, в предыдущих версиях scikit-learn (например, 0.21) по умолчанию применялся алгоритм 'liblinear', который не может обрабатывать мультиномиальные потери и ограничен схемой OvR для многоклассовой классификации. Однако в scikit-learn версии 0.22 параметр решателя по умолчанию был изменен на 'lbfgs', что означает применение алгоритма Бродейна — Флетчера — Голдфарба — Шанно (Broyden — Fletcher — Goldfarb — Shanno, BFGS) с ограниченной памятью (https://en.wikipedia.org/wiki/Limited-memory_BFGS), который является более гибким в этом отношении.

Если вы внимательно изучили код, который мы использовали для обучения модели LogisticRegression, у вас мог возникнуть вопрос: «Что это за загадочный параметр c?» Мы обсудим этот параметр в следующем разделе, где введем понятия переобучения и регуляризации. Однако прежде чем перейти к этим темам, давайте закончим обсуждение вероятностей членства в классе.

Вероятность того, что обучающие образцы принадлежат определенному классу, можно вычислить с помощью метода predict_proba. Например, мы можем спрогнозировать вероятности первых трех примеров в тестовом наборе данных следующим образом:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

Эта строка кода возвращает следующий массив:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

В приведенном выводе на экран первая строка содержит вероятности принадлежности к классу первого цветка, вторая строка содержит вероятности принадлежности к классу второго цветка и т. д. Обратите внимание, что сумма по столбцам в каждой строке равна 1, как и ожидалось. (Вы можете убедиться в этом, выполнив строку кода lr.predict_proba(X_test_std[:3, :]).sum(axis=1).)

Наибольшее значение в первой строке составляет примерно 0.85. Это означает, что первый образец цветка относится к классу 3 (Iris-virginica) с прогнозируемой вероятностью 85%. Возможно, вы уже догадались, что мы можем получить предсказанные метки классов, определив самый большой столбец в каждой строке, например, с помощью функции argmax библиотеки NumPy:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

Далее показаны возвращенные индексы классов (они соответствуют Iris-virginica, Iris-setosa и Iris-versicolor):

```
array([2, 0, 0])
```

В предыдущем примере кода мы вычислили условные вероятности и преобразовали их в метки классов вручную с помощью функции argmax NumPy. На практике более удобным способом получения меток классов при использовании scikit-learn является прямой вызов метода predict:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

Наконец, если вы хотите предсказать метку класса для одного образца, то имейте в виду, что в качестве входных данных библиотека scikit-learn ожидает двумерный массив, — следовательно, вы должны сначала преобразовать срез одной строки в нужный формат. Один из способов преобразования односторочной записи в двумерный массив данных — использовать метод `reshape` NumPy для добавления нового измерения, как показано в следующем примере кода:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

3.3.5. Борьба с переобучением путем регуляризации

Переобучение — распространенная проблема в машинном обучении, проявляющаяся в том, что модель хорошо работает на обучающих данных, но плохо обобщается на незнакомые (тестовые) данные. Если модель страдает от переобучения, также говорят, что модель *имеет высокую дисперсию*, которая может быть вызвана наличием слишком большого количества параметров, — т. е. модель слишком сложна по отношению к исходным данным. Точно так же наша модель может страдать от *недообучения* (*underfitting*), когда она недостаточно сложна, чтобы хорошо выявлять закономерности в обучающих данных, и, следовательно, тоже обладает низкой обобщающей способностью на незнакомых данных.

Хотя до сих пор мы имели дело только с линейными моделями классификации, проблемы переобучения и недообучения лучше всего проиллюстрировать, сравнив линейную разделяющую границу с более сложными, нелинейными границами, как показано на рис. 3.8.

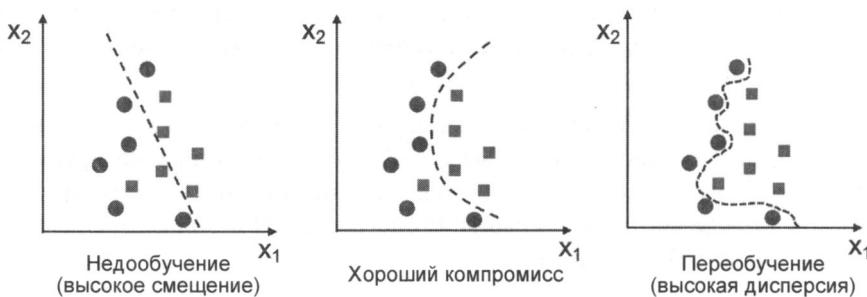


Рис. 3.8. Примеры недообученных, хорошо обученных и переобученных моделей



Компромисс между систематической ошибкой и дисперсией

Часто исследователи используют термины *систематическая ошибка* (*bias*) и *дисперсия* (*variance*) или *компромисс между смещением и дисперсией* для описания производительности модели. Вы можете встретить доклады, книги или статьи, в которых люди говорят, что модель имеет «высокую дисперсию» или «высокую систематическую ошибку». Что это значит? В целом можно сказать, что высокая дис-

персия пропорциональна переобучению, а высокая систематическая ошибка пропорциональна недообучению.

В контексте моделей машинного обучения дисперсия отражает изменчивость прогноза модели при классификации конкретного экземпляра, если мы повторно обучим модель несколько раз, например на разных подмножествах обучающего набора данных. Можно сказать, что модель чувствительна к случайной составляющей обучающих данных. Напротив, систематическая ошибка показывает, насколько далеки прогнозы от правильных значений в целом, если мы заново обучим модель на различающихся наборах данных. То есть систематическая ошибка — это мера, которая не связана со случайностью.

Если вас интересуют технические определения и происхождение терминов «систематическая ошибка» и «дисперсия», скачайте конспекты моих лекций по адресу: https://sebastianraschka.com/pdf/lecture-notes/stat451fs20/08-model-eval-1-intro_notes.pdf.

Один из способов найти хороший компромисс между систематической ошибкой и дисперсией — точная настройка сложности модели с помощью регуляризации. *Регуляризация* — это очень полезный метод для устранения коллинеарности (высокой корреляции между признаками), фильтрации шума из данных и в конечном итоге предотвращения переобучения.

Идея регуляризации заключается в том, чтобы ввести дополнительную информацию для штрафа за экстремальные значения параметров (весов). Наиболее распространенной формой регуляризации является так называемая *L2-регуляризация* (иногда также называемая *L2-усадкой*, или *затуханием веса*), которую можно записать следующим образом:

$$\frac{\lambda}{2n} \|\mathbf{w}\|^2 = \frac{\lambda}{2n} \sum_{j=1}^m w_j^2.$$

Здесь λ — так называемый *параметр регуляризации* (regularization parameter). Обратите внимание, что 2 в знаменателе — это просто коэффициент масштабирования, поэтому он отсутствует при вычислении градиента потерь. Размер выборки n добавляется для масштабирования члена регуляризации.



Регуляризация и нормализация признаков

Регуляризация — еще одна причина, по которой важно масштабирование признаков, такое как стандартизация. Чтобы регуляризация работала должным образом, нам нужно убедиться, что все наши признаки имеют сопоставимые числовые масштабы.

Функцию потерь для логистической регрессии можно регуляризовать, добавив простой член регуляризации, который уменьшит веса во время обучения модели:

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n [-y^{(i)} \log(\sigma(z^{(i)})) - (1 - y^{(i)}) \log(1 - \sigma(z^{(i)}))] + \frac{\lambda}{2n} \|\mathbf{w}\|^2.$$

Частная производная нерегуляризованных потерь определяется как

$$\frac{\partial L(\mathbf{w}, b)}{\partial w_j} = \left(\frac{1}{n} \sum_{i=1}^n (\sigma(\mathbf{w}^T \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \right).$$

Добавление члена регуляризации к потерям приводит частную производную к следующему виду:

$$\frac{\partial L(\mathbf{w}, b)}{\partial w_j} = \left(\frac{1}{n} \sum_{i=1}^n (\sigma(\mathbf{w}^T \mathbf{x}^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{n} w_j.$$

Теперь с помощью параметра регуляризации λ можно задавать, насколько точно мы подгоняем модель к обучающим данным, сохраняя при этом небольшие веса. Увеличивая значение λ , мы увеличиваем значимость члена регуляризации. Заметим, что систематическая ошибка, которая по сути представляет собой член пересечения в уравнении прямой, или отрицательный порог, как вы узнали из главы 2, обычно не регулируется.

Параметр C введен в класс `LogisticRegression` библиотеки `scikit-learn` на основании соглашения о методе опорных векторов, которое будет темой следующего раздела. Член с обратно пропорционален параметру регуляризации λ . Следовательно, уменьшение значения обратного параметра регуляризации с означает, что мы увеличиваем силу регуляризации, которую можем визуализировать, построив путь регуляризации L2 для двух весовых коэффициентов:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c,
...                             multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...            label='Длина лепестка')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...            label='Ширина лепестка')
>>> plt.ylabel('Весовой коэффициент')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

Выполнив приведенный код, мы обучили 10 моделей логистической регрессии с различными значениями обратного параметра регуляризации c . Для иллюстрации мы собрали только весовые коэффициенты класса 1 (здесь второй класс в наборе данных: `Iris-versicolor`) по сравнению со всеми классификаторами — помните, что мы используем метод OvR для многоклассовой классификации.

Как можно видеть на рис. 3.9, весовые коэффициенты уменьшаются, если мы уменьшаем параметр c , т. е. если увеличиваем влияние регуляризации.

Поскольку увеличение влияния регуляризации снижает риск переобучения, у вас может возникнуть вопрос, почему все модели не подвергают регуляризации по умолчанию. Причина в том, что при настройке регуляризации необходимо соблюдать осторожность. Например, если регуляризация слишком сильна, а весовые коэффициенты приближаются к нулю, модель может работать очень плохо из-за недообучения, как показано на рис. 3.8.

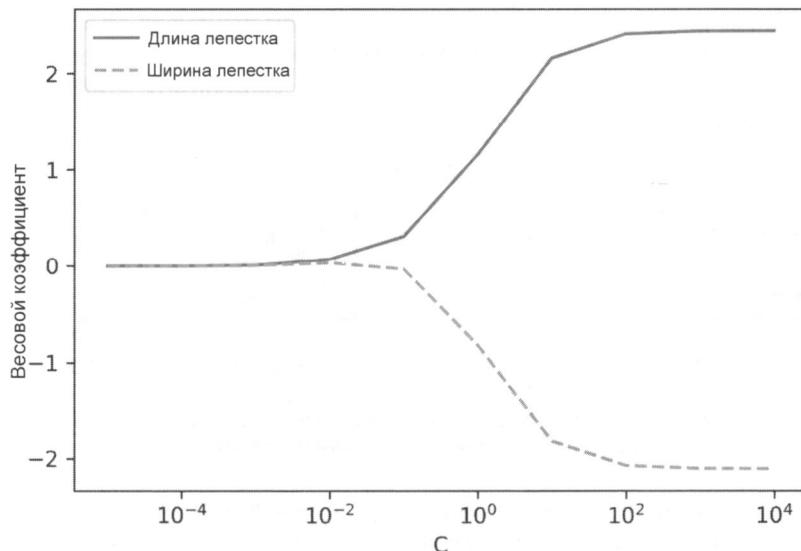


Рис. 3.9. Влияние параметра обратной регуляризации C на весовые коэффициенты регуляризованной модели L2



Дополнительный ресурс по логистической регрессии

Поскольку подробное описание отдельных алгоритмов классификации выходит за рамки этой книги, мы отсылаем читателей, которые хотят узнать больше о логистической регрессии, к книге «Logistic Regression: From Introductory to Advanced Concepts and Applications» Dr. Scott Menard, Sage Publications, 2009 (на русском языке не издавалась).

3.4. Классификация по наибольшему отступу с помощью метода опорных векторов

Еще одним мощным и широко используемым алгоритмом обучения является *метод опорных векторов* (Support Vector Machine, SVM), который можно считать расширением персептрона. Используя алгоритм персептрана, мы минимизировали ошибки неправильной классификации. Однако в SVM наша цель оптимизации состоит в том, чтобы максимизировать *отступ* (зазор, margin). Отступ определяется как расстояние между разделяющей гиперплоскостью (границей решения) и ближайшими к этой гиперплоскости точками обучающих данных, которые являются так называемыми *опорными векторами* (support vector). Этот подход показан на рис. 3.10.

3.4.1. Смысл максимизации зазора

Идея использования разделяющих границ с наибольшими отступами заключается в том, что они, как правило, имеют меньшую ошибку обобщения, тогда как модели с малыми отступами более склонны к переобучению.

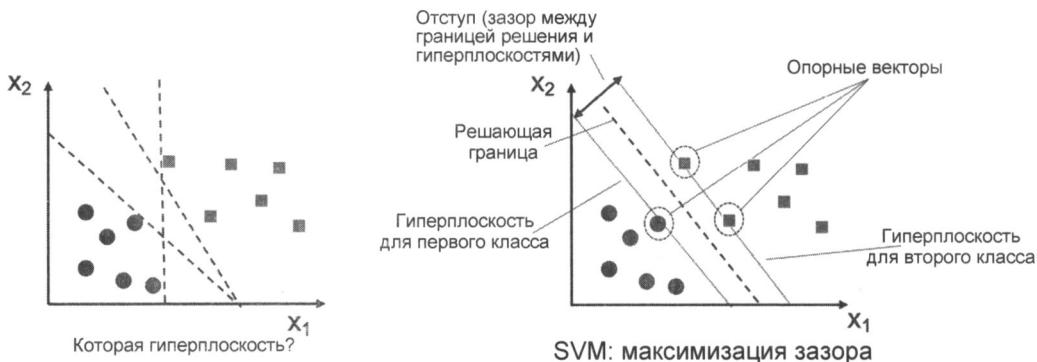


Рис. 3.10. Метод SVM основан на поиске наибольшего расстояния между решающей границей и точками обучающих данных

К сожалению, в то время как ключевая идея, лежащая в основе SVM, относительно проста, математические основы довольно сложны и требуют хороших знаний в области оптимизации с ограничениями.

Детальное описание метода оптимизации путем максимизации зазора в SVM выходит за рамки этой книги. Заинтересованным читателям, желающим узнать больше, мы рекомендуем следующие ресурсы:

- ◆ отличное объяснение Криса Дж. К. Берджеса (Chris J. C. Burges) в публикации «A Tutorial on Support Vector Machines for Pattern Recognition» (Data Mining and Knowledge Discovery, 2(2): 121–167, 1998);
- ◆ книга Владимира Вапника (Vladimir Vapnik) «The Nature of Statistical Learning Theory» (Springer Science+Business Media, 2000);
- ◆ очень подробные конспекты лекций Эндрю Йида (Andrew Ng), доступные по адресу: <https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf>.

3.4.2. Работа с нелинейно разделимыми случаем при использовании резервных переменных

Хотя мы не планируем углубляться в более сложные математические понятия, лежащие в основе классификации с наибольшим отступом, стоит вкратце упомянуть так называемую *переменную невязки* (slack variable, переменная ослабления), которая была введена Владимиром Вапником в 1995 году и привела к появлению так называемой *классификации с мягким отступом* (soft-margin classification). Основная причина введения переменной невязки заключалась в том, что линейные ограничения в цели оптимизации SVM необходимо ослабить для случая нелинейно разделимых данных, чтобы обеспечить сходимость оптимизации при наличии ошибочных классификаций и соответствующем штрафе за потери.

Использование переменной невязки, в свою очередь, привело к появлению переменной, которую обычно обозначают как C , если речь идет об использовании SVM. Мы можем рассматривать C как гиперпараметр для управления величиной штрафа за неправильную классификацию. Большие значения C соответствуют большим штрафам за ошибки

классификации, в то время как меньшие значения C влекут за собой меньшие штрафы. Мы можем использовать параметр C для управления величиной отступа и, следовательно, настроить компромисс между смещением и дисперсией, как показано на рис. 3.11.

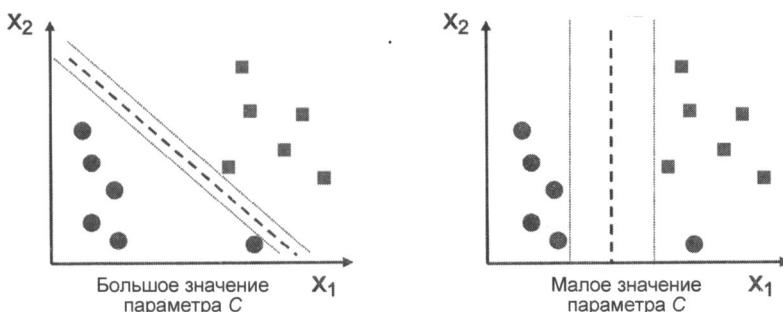


Рис. 3.11. Влияние больших и малых значений обратного коэффициента регуляризации C на классификацию

Эти рассуждения имеют прямое отношение к регуляризации, которую мы обсуждали в предыдущем разделе в контексте регуляризованной регрессии, когда уменьшение значения C увеличивает систематическое смещение (недообучение) и снижает дисперсию (переобучение) модели.

На этом мы закончим изучение основ линейного SVM и перейдем к обучению модели SVM для классификации различных цветков в нашем наборе данных Iris:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined,
...                         classifier=svm,
...                         test_idx=range(105, 150))
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Выполнив приведенный фрагмент кода, вы должны увидеть три области принятия решений SVM, сформированные после обучения классификатора на наборе данных Iris (рис. 3.12).



Логистическая регрессия или SVM?

В прикладных задачах классификации линейная логистическая регрессия и линейные SVM часто дают очень похожие результаты. Логистическая регрессия пытается максимизировать условную функцию правдоподобия на обучающих данных, что делает ее более склонной к выбросам, чем метод SVM, который в основном заботится о точках, ближайших к границе решения (опорных векторах). С другой стороны, логистическая регрессия имеет то преимущество, что она является более

простой моделью, ее легче реализовать и проще объяснить с математической точки зрения. Кроме того, модели логистической регрессии можно легко обновлять, что удобно при работе с потоковыми данными.

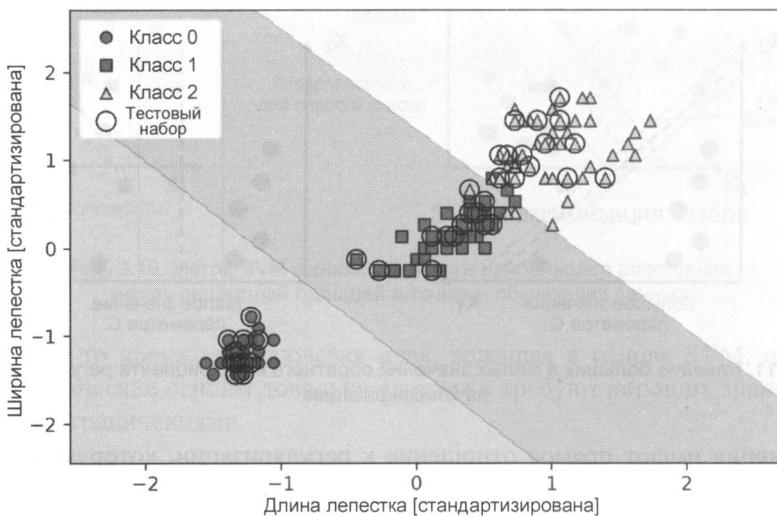


Рис. 3.12. Области принятия решений SVM

3.4.3. Альтернативные реализации алгоритмов в scikit-learn

Класс `LogisticRegression` библиотеки `scikit-learn`, который мы использовали в предыдущих разделах, может задействовать библиотеку LIBLINEAR, если установить параметр `solver='liblinear'`. LIBLINEAR — это высокооптимизированная библиотека C/C++, разработанная в Национальном университете Тайваня (<http://www.csie.ntu.edu.tw/~cjlin/liblinear/>).

Точно так же класс `svc`, который мы использовали для обучения модели SVM, задает LIBSVM — эквивалентную библиотеку C/C++, предназначенную для SVM (<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>).

Преимущество использования библиотек LIBLINEAR и LIBSVM по сравнению, например, с прямыми реализациями на Python заключается в том, что они позволяют очень быстро обучать большое количество линейных классификаторов. Однако иногда наши наборы данных слишком велики, чтобы уместиться в памяти компьютера. Поэтому `scikit-learn` также предлагает альтернативные реализации через класс `SGDClassifier`, который поддерживает онлайн-обучение с помощью метода `partial_fit`. Класс `SGDClassifier` основан на уже знакомом вам принципе стохастического градиента, который мы реализовали в главе 2 для Adaline.

Мы можем инициализировать SGD-версию персептрона (`loss='perceptron'`), логистическую регрессию (`loss='log'`) и SVM с параметрами по умолчанию (`loss='hinge'`) следующим образом:

```
>>> from sklearn.linear_model import SGDClassifier  
>>> ppn = SGDClassifier(loss='perceptron')  
>>> lr = SGDClassifier(loss='log')  
>>> svm = SGDClassifier(loss='hinge')
```

3.5. Решение нелинейных задач с использованием ядерного варианта SVM

Еще одна причина, по которой различные реализации SVM обрели большую популярность среди специалистов по машинному обучению, заключается в том, что их можно легко использовать для решения задач нелинейной классификации. Прежде чем обсудить основную идею наиболее распространенного варианта SVM — так называемого *ядерного SVM* (kernel SVM), сначала создадим синтетический набор данных, чтобы увидеть, как может выглядеть типичная проблема нелинейной классификации.

3.5.1. Ядерные методы для линейно неразделимых данных

Используя представленный далее код, мы создадим при помощи функции `logical_xor` из NumPy простой набор данных в форме элемента XOR, где 100 записям будет присвоена метка класса 1, а 100 другим записям — метка класса -1:

```
>>> import matplotlib.pyplot as plt  
>>> import numpy as np  
>>> np.random.seed(1)  
>>> X_xor = np.random.randn(200, 2)  
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,  
...                           X_xor[:, 1] > 0)  
>>> y_xor = np.where(y_xor, 1, 0)  
>>> plt.scatter(X_xor[y_xor == 1, 0],  
...               X_xor[y_xor == 1, 1],  
...               c='royalblue', marker='s',  
...               label='Class 1')  
>>> plt.scatter(X_xor[y_xor == 0, 0],  
...               X_xor[y_xor == 0, 1],  
...               c='tomato', marker='o',  
...               label='Class 0')  
>>> plt.xlim([-3, 3])  
>>> plt.ylim([-3, 3])  
>>> plt.xlabel('Признак 1')  
>>> plt.ylabel('Признак 2')  
>>> plt.legend(loc='best')  
>>> plt.tight_layout()  
>>> plt.show()
```

После выполнения этого кода мы получим набор данных XOR со случайным шумом, как показано на рис. 3.13.

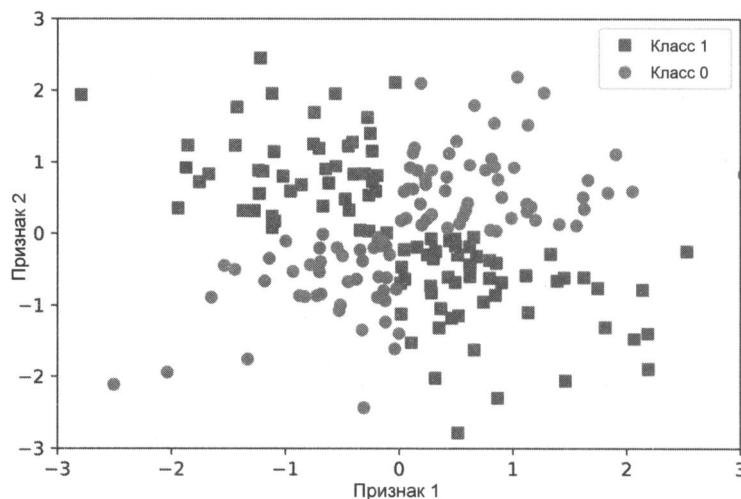


Рис. 3.13. Диаграмма распределения точек данных набора XOR

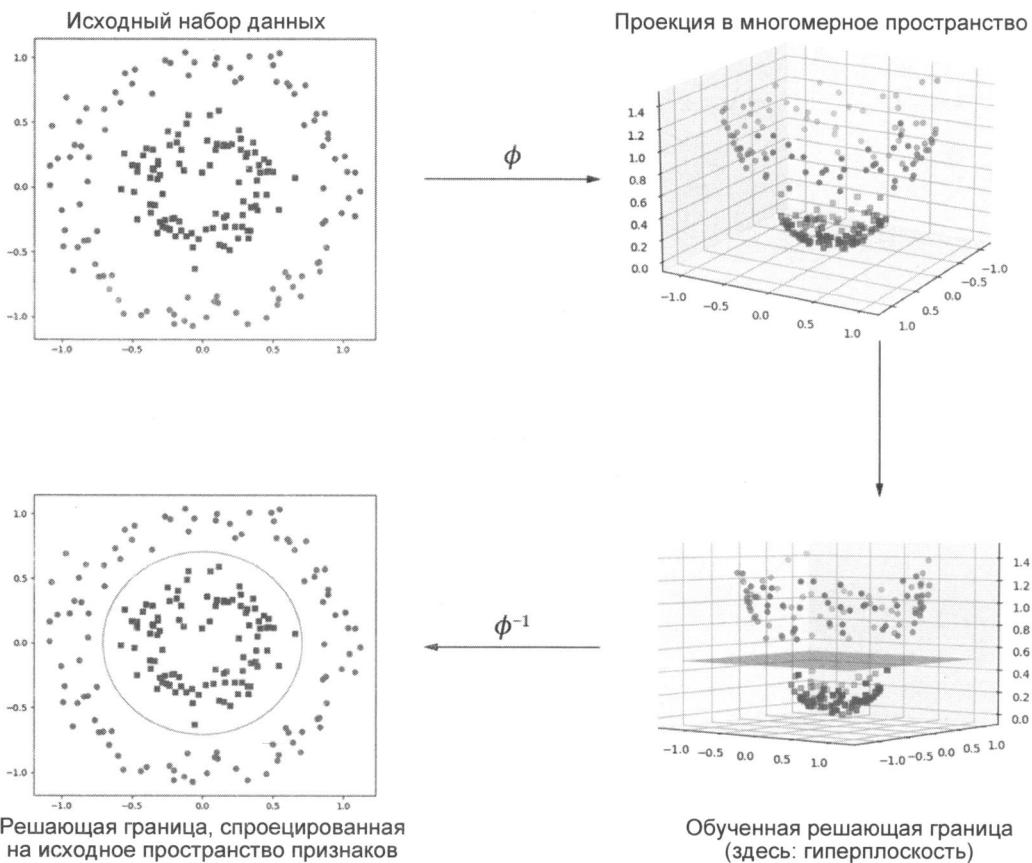


Рис. 3.14. Процесс нелинейного разделения классов с использованием ядерных методов

Очевидно, что мы не сможем хорошо разделить экземпляры положительного и отрицательного класса, используя в качестве разделяющей границы линейную гиперплоскость, — т. е. с помощью линейной логистической регрессии или линейной модели SVM, которые мы обсуждали в предыдущих разделах.

Основная идея ядерных методов (kernel method) для работы с такими линейно неразделимыми данными заключается в создании нелинейных комбинаций исходных признаков для проецирования их в многомерное пространство с помощью функции отображения ϕ , где данные становятся линейно разделимыми. Как показано на рис. 3.14, мы можем преобразовать двумерный набор данных в новое трехмерное пространство признаков, в котором классы становятся разделимыми посредством следующей проекции:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2).$$

Это позволяет нам разделить видимые на графике два класса с помощью линейной гиперплоскости, которая становится нелинейной решающей границей, если мы спроектируем ее обратно на исходное пространство признаков, что и продемонстрировано на примере набора данных в виде концентрических кругов.

3.5.2. Использование ядерного трюка для поиска разделяющих гиперплоскостей в многомерном пространстве

Чтобы решить нелинейную задачу, используя SVM, мы должны перенести обучающие данные в многомерное пространство признаков с помощью функции отображения ϕ и обучить линейную модель SVM классифицировать данные в этом новом пространстве признаков. Затем мы можем применить ту же функцию отображения ϕ для преобразования новых (незнакомых) данных и классифицировать их с помощью линейной модели SVM.

Однако проблема с отображением данных в многомерное пространство заключается в том, что извлечение новых признаков требует очень больших вычислительных ресурсов, особенно если мы имеем дело с многомерными данными. Здесь в игру вступает так называемый ядерный трюк (kernel trick).

Не вдаваясь в подробности решения задачи квадратичного программирования для обучения SVM, заметим, что на практике нам просто нужно заменить скалярное произведение $x^{(i)T} x^{(j)}$ на $\phi(x^{(i)})^T \phi(x^{(j)})$. Чтобы избежать дорогостоящего шага явного вычисления этого скалярного произведения между двумя точками, определим так называемую *керн-функцию* (ядерную функцию, kernel function):

$$\kappa(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)}).$$

Одним из наиболее широко используемых ядер является ядро *радиальной базисной функции* (Radial Basis Function, RBF), которое можно просто назвать *гауссовым ядром* (Gaussian kernel):

$$\kappa(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|^2}{2\sigma^2}\right).$$

Оно часто упрощается до вида:

$$\kappa(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp(-\gamma \|\mathbf{x}^{(i)} - \mathbf{x}^{(j)}\|^2),$$

где $\gamma = \frac{1}{2\sigma^2}$ является свободным параметром для оптимизации.

Говоря упрощенно, термин «ядро» можно интерпретировать как *функцию подобия* (similarity function) между парой экземпляров. Знак «минус» инвертирует меру расстояния, превращая ее в оценку сходства, и, благодаря экспоненциальному члену, результирующая оценка сходства будет находиться в диапазоне от 1 (для точно совпадающих экземпляров) до 0 (для очень непохожих экземпляров).

Итак, мы в общих чертах рассмотрели ядерный трюк, а теперь попробуем обучить ядерную версию SVM, способную найти нелинейную решающую границу, которая хорошо разделяет данные XOR. В следующем фрагменте кода мы просто используем класс `SVC` из `scikit-learn`, который импортировали ранее, и заменяем параметр `kernel='linear'` на `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.1, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

График на рис. 3.15 свидетельствует о том, что ядерная версия SVM относительно хорошо разделяет данные XOR.

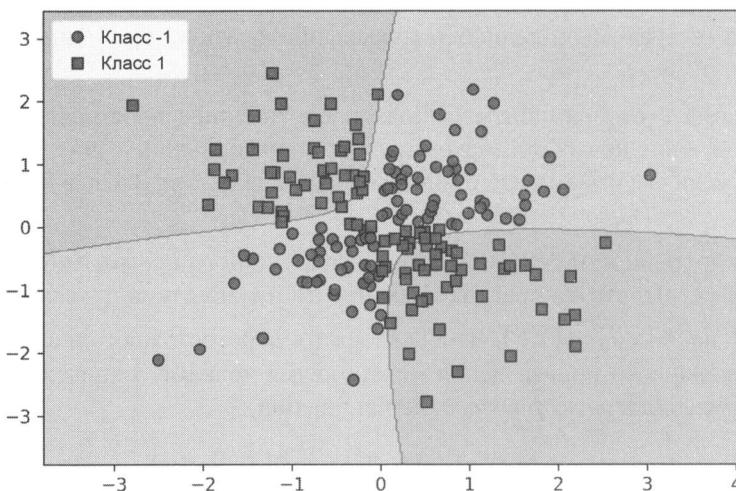


Рис. 3.15. Решающая граница для данных XOR, полученная с использованием ядерного метода

Параметр γ , для которого мы установили значение `gamma=0.1`, можно понимать как параметр отсечки для гауссовой сферы. Выбрав слишком большое значение γ , мы увеличим влияние или охват обучающих примеров, что приведет к более жесткой и неровной

разделяющей границе. Чтобы лучше понять γ , давайте применим SVM с ядром RBF к набору данных цветков ириса:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105, 150))
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Поскольку мы выбрали относительно небольшое значение для γ , полученная разделяющая граница SVM-модели с ядром RBF будет относительно мягкой, как показано на рис. 3.16.

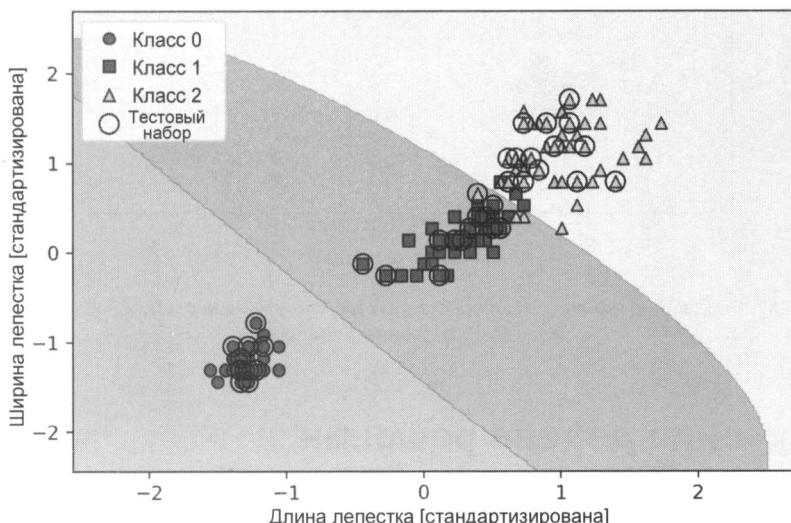


Рис. 3.16. Разделяющие границы в наборе данных Iris, полученные с помощью модели SVM (ядро RBF, небольшое значение γ)

Теперь давайте увеличим значение γ и посмотрим, как это повлияет на границы:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                         y_combined, classifier=svm,
...                         test_idx=range(105, 150))
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Теперь мы можем видеть, что при относительно большом значении γ разделяющая граница вокруг классов 0 и 1 становится намного более узкой (рис. 3.17).

Хотя модель очень точно соответствует обучающему набору данных, такой классификатор, вероятно, будет иметь высокую ошибку обобщения на незнакомых данных. Это говорит о том, что параметр γ также играет важную роль в борьбе с переобучением или дисперсией, когда алгоритм слишком чувствителен к колебаниям в обучающем наборе данных.

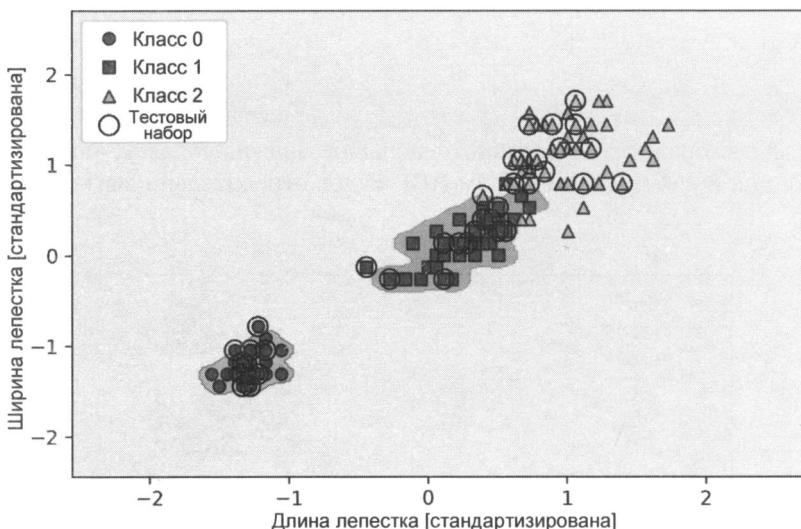


Рис. 3.17. Решающие грани в наборе данных Iris, полученные с помощью модели SVM (ядро RBF, значение γ увеличено)

3.6. Обучение дерева решений

Если мы заботимся об интерпретируемости модели, хорошим выбором будут классификаторы на основе *дерева решений*. Как следует из названия, «дерево решений» можно рассматривать как классификацию данных путем принятия последовательных решений, исходя из серии вопросов.

Давайте рассмотрим простой пример, в котором мы используем дерево решений для выбора действия в определенный день (рис. 3.18).

Основываясь на признаках обучающего набора данных, модель дерева решений изучает последовательности вопросов, позволяющие вывести метки классов экземпляров. Хотя на рис. 3.18 показано дерево решений, построенное на категориальных переменных, та же концепция применяется, если признаки являются действительными числами, как в наборе данных Iris. Например, мы могли бы просто определить пороговое значение вдоль оси признака *ширины чашелистика* и задать бинарный вопрос: «Ширина чашелистика $\geq 2,8$ см?»

Используя алгоритм принятия решения, мы начинаем с корня дерева и разделяем данные по признаку, который приводит к наибольшему *приросту информации* (Information

Gain, IG) — мы раскроем это понятие в следующем разделе. Затем мы итеративно повторяем эту процедуру разделения на каждом дочернем узле, пока листья дерева не станут чистыми (pure). Это означает, что все обучающие экземпляры в каждом узле принадлежат одному и тому же классу. На практике может получиться очень глубокое дерево с множеством узлов, что легко приводит к переобучению. Именно по этой причине дерево обычно обрезают (prune), устанавливая ограничение на максимальную глубину дерева.

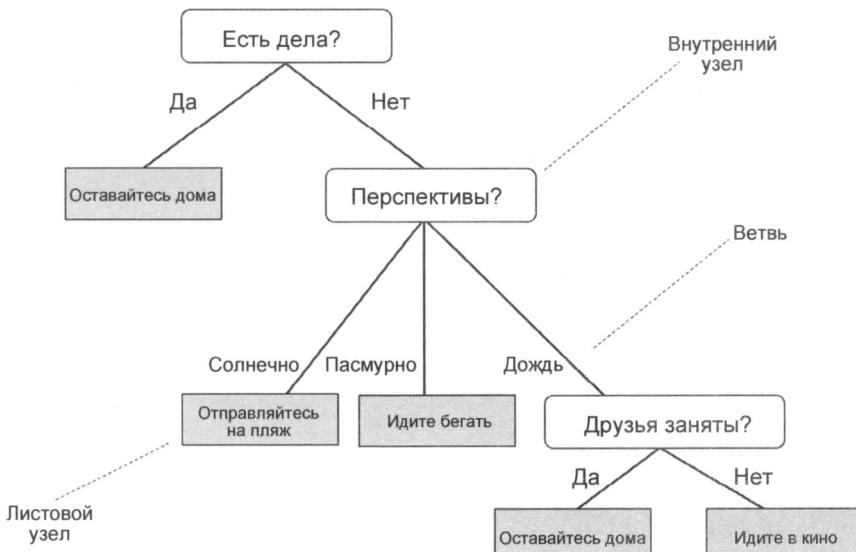


Рис. 3.18. Пример дерева решений

3.6.1. Максимизация IG: получение наибольшей отдачи от затраченных усилий

Чтобы разделить узлы по наиболее информативным признакам, нам нужно определить целевую функцию, подлежащую оптимизации с помощью алгоритма обучения дерева решений. В таком случае наша цель состоит в том, чтобы каждое разделение данных давало максимально возможный прирост информации, который мы определяем следующим образом:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^m \frac{N_j}{N_p} I(D_j).$$

Здесь f — функция для выполнения разделения; D_p и D_j — наборы данных родительского и j -го дочерних узлов; I — мера примеси (impurity); N_p — общее количество обучающих примеров на родительском узле; N_j — количество примеров в j -м дочернем узле. Вы можете видеть, что прирост информации — это просто разница между примесью родительского узла и суммой примесей дочерних узлов, т. е. чем ниже примеси дочерних узлов, тем больше прирост информации. Однако для простоты и сокращения пространства комбинаторного поиска большинство библиотек (включая scikit-learn)

реализуют *бинарные деревья решений*. Это означает, что каждый родительский узел разделяется строго на два дочерних узла — D_{left} и D_{right} :

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}).$$

Три меры примеси, или критерии разделения, которые обычно используются в бинарных деревьях решений, — это *примесь Джини* (*Gini Impurity*, I_G), *энтропия* (*Entropy*, I_H) и *ошибка классификации* (*Classification Error*, IE). Начнем с определения энтропии для всех *непустых* классов ($p(i|t) \neq 0$):

$$I_H(t) = - \sum_{i=1}^c p(i|t) \log_2 p(i|t).$$

Здесь $p(i|t)$ — доля экземпляров, принадлежащих классу i для конкретного узла t . Таким образом, энтропия равна 0, если все примеры в узле принадлежат к одному и тому же классу, и энтропия максимальна, если у нас есть равномерное распределение классов. Например, в бинарном классе энтропия равна 0, если $p(i=1|t) = 1$ или $p(i=0|t) = 0$. Если классы распределены равномерно с $p(i=1|t) = 0.5$ и $p(i=0|t) = 0.5$, энтропия равна 1. Следовательно, можно сказать, что критерий энтропии направлен на достижение максимальной взаимной информации в дереве.

Чтобы закрепить понимание, давайте построим кривую энтропии для различных распределений классов с помощью следующего кода:

```
>>> def entropy(p):
...     return - p * np.log2(p) - (1 - p) * np.log2((1 - p))
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> plt.ylabel('Энтропия')
>>> plt.xlabel('Вероятность принадлежности к классу p(i=1)')
>>> plt.plot(x, ent)
>>> plt.show()
```

На рис. 3.19 показан график, созданный этим кодом.

Примесь Джини можно рассматривать как критерий минимизации вероятности ошибочной классификации:

$$I_G(t) = \sum_{i=1}^c p(i|t) (1 - p(i|t)) = 1 - \sum_{i=1}^c p(i|t)^2.$$

Подобно энтропии примесь Джини максимальна, если классы полностью перемешаны, — например, в случае бинарных классов ($c = 2$):

$$I_G(t) = 1 - \sum_{i=1}^c 0.5^2 = 0.5.$$

Однако на практике как примесь Джини, так и энтропия, обычно дают очень похожие результаты, и часто не стоит тратить много времени на оценку деревьев с использованием разных критериев примеси, вместо того, чтобы экспериментировать с различными пороговыми значениями. На самом деле, как вы увидите позже на рис. 3.21, и примесь Джини, и энтропия имеют одинаковую форму.

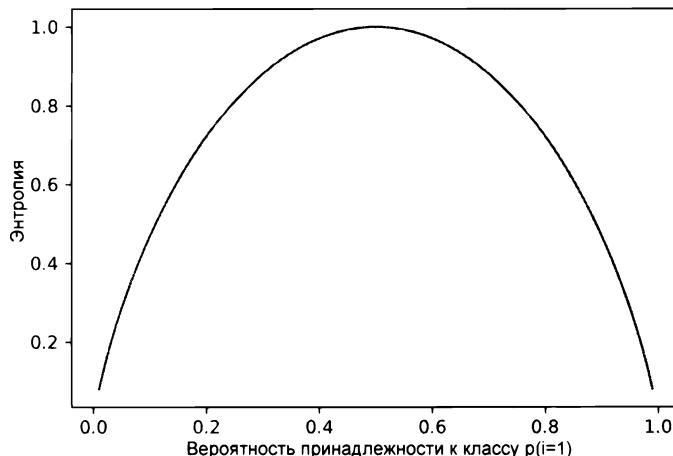


Рис. 3.19. Значения энтропии для различных вероятностей принадлежности к классу

Еще одной мерой примеси является ошибка классификации:

$$I_E(t) = 1 - \max\{p(i|t)\}.$$

Это полезный критерий для обрезки дерева, но его не рекомендуется применять при построении дерева решений, поскольку он менее чувствителен к изменениям вероятностей классов узлов. Мы можем проиллюстрировать это, рассмотрев два возможных сценария разделения (рис. 3.20).



Рис. 3.20. Разделение данных дерева решений

Мы начинаем с набора данных D_p в родительском узле, который состоит из 40 экземпляров класса 1 и 40 экземпляров класса 2, которые мы разделяем на два набора данных, D_{left} и D_{right} . При использовании ошибки классификации в качестве критерия разделения прирост информации будет одинаковым ($IG_E = 0.25$) в обоих сценариях, A и B:

$$I_E(D_p) = 1 - 0.5 = 0.5$$

$$A: \quad I_E(D_{left}) = 1 - \frac{3}{4} = 0.25$$

$$A: \quad I_E(D_{right}) = 1 - \frac{3}{4} = 0.25$$

$$A: \quad IG_E = 0.5 - \frac{4}{8} \cdot 0.25 - \frac{4}{8} \cdot 0.25 = 0.25$$

$$B: \quad I_E(D_{left}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B: \quad I_E(D_{right}) = 1 - 1 = 0$$

$$B: \quad IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

Однако примесь Джини будет способствовать разделению по сценарию B ($IG_G = 0.16$) который действительно чище по сравнению со сценарием A ($IG_G = 0.125$):

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A: \quad I_G(D_{left}) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2 \right) = \frac{3}{8} = 0.375$$

$$A: \quad I_G(D_{right}) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2 \right) = \frac{3}{8} = 0.375$$

$$A: \quad IG_G = 0.5 - \frac{4}{8} 0.375 - \frac{4}{8} 0.375 = 0.125$$

$$B: \quad I_G(D_{left}) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2 \right) = \frac{4}{9} = 0.\bar{4}$$

$$B: \quad I_G(D_{right}) = 1 - (1^2 + 0^2) = 0$$

$$B: \quad IG_G = 0.5 - \frac{6}{8} 0.\bar{4} - 0 = 0.1\bar{6}$$

Критерий энтропии тоже будет отдавать предпочтение сценарию B ($IG_H = 0.31$) по сравнению со сценарием A ($IG_H = 0.19$):

$$I_H(D_p) = -(0.5 \log_2(0.5) + 0.5 \log_2(0.5)) = 1$$

$$A: \quad I_H(D_{left}) = -\left(\frac{3}{4} \log_2\left(\frac{3}{4}\right) + \frac{1}{4} \log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: \quad I_H(D_{right}) = -\left(\frac{1}{4} \log_2\left(\frac{1}{4}\right) + \frac{3}{4} \log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: \quad IG_H = 1 - \frac{4}{8} 0.81 - \frac{4}{8} 0.81 = 0.19$$

$$B: \quad I_H(D_{left}) = -\left(\frac{2}{6} \log_2\left(\frac{2}{6}\right) + \frac{4}{6} \log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B: \quad I_H(D_{right}) = 0$$

$$B: \quad IG_H = 1 - \frac{6}{8} 0.92 - 0 = 0.31$$

Для более наглядного сравнения трех различных критериев примесей, которые мы обсуждали ранее, построим график индексов примесей для диапазона вероятностей $[0, 1]$ для класса 1. Обратите внимание, что мы добавляем на график масштабированную версию энтропии (энтропия/2) чтобы показать, что примесь Джини является промежуточной мерой между энтропией и ошибкой классификации:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
```

```

>>> def gini(p):
...     return p*(1 - p) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                           ['Энтропия', 'Энтропия (масшт.)',
...                            'Примесь Джини',
...                            'Ошибка классификации'],
...                           ['-', '--', '-.', '-.'],
...                           ['black', 'lightgray',
...                            'red', 'green', 'cyan'])):
...     line = ax.plot(x, i, label=lab, linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...            ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('индекс примеси')
>>> plt.show()

```

График, созданный этим примером кода, представлен на рис. 3.21.

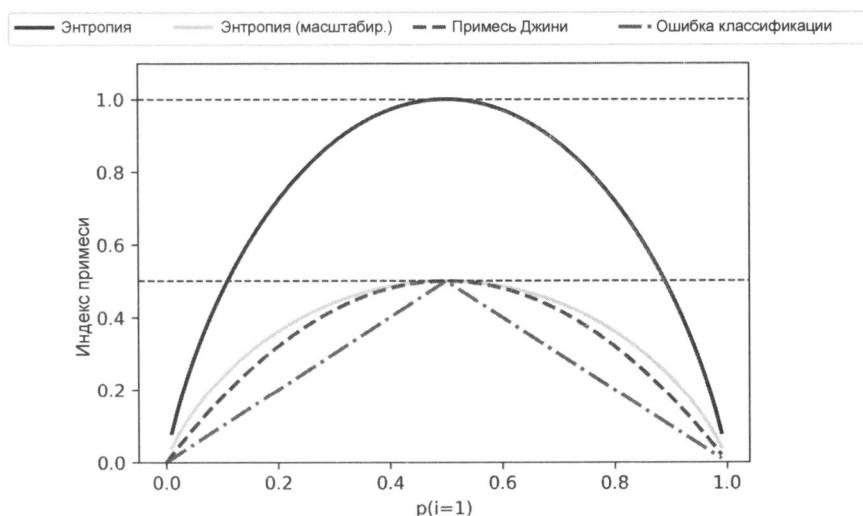


Рис. 3.21. Различные индексы примесей для различных вероятностей принадлежности к классу в интервале от 0 до 1

3.6.2. Построение дерева решений

Деревья решений могут создавать сложные решающие граници, разделяя пространство признаков на прямоугольники. Однако необходимо соблюдать осторожность, т. к. чем глубже дерево решений, тем сложнее становится решающая граница, что может легко привести к переобучению. Используя библиотеку scikit-learn, мы будем обучать дерево решений с максимальной глубиной 4, используя в качестве критерия примесь Джини.

Хотя масштабирование признаков может пригодиться для более наглядной визуализации, оно не является обязательным условием при использовании алгоритмов дерева решений. Код выглядит следующим образом:

```
>>> from sklearn.tree import DecisionTreeClassifier  
>>> tree_model = DecisionTreeClassifier(criterion='gini',  
...                                         max_depth=4,  
...                                         random_state=1)  
>>> tree_model.fit(X_train, y_train)  
>>> X_combined = np.vstack((X_train, X_test))  
>>> y_combined = np.hstack((y_train, y_test))  
>>> plot_decision_regions(X_combined,  
...                         y_combined,  
...                         classifier=tree_model,  
...                         test_idx=range(105, 150))  
>>> plt.xlabel('Длина лепестка [см]')  
>>> plt.ylabel('Ширина лепестка [см]')  
>>> plt.legend(loc='upper left')  
>>> plt.tight_layout()  
>>> plt.show()
```

Выполнив этот код, мы получим типичные параллельные оси границ дерева решений (рис. 3.22).

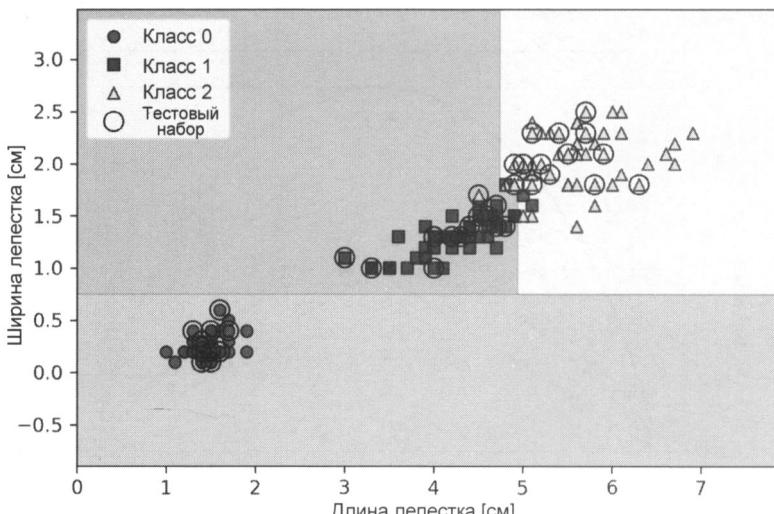


Рис. 3.22. Решающие граници для данных Iris, полученные с помощью дерева решений

Приятной особенностью библиотеки scikit-learn является то, что она позволяет нам легко визуализировать модель дерева решений после обучения с помощью следующего кода (рис. 3.23):

```
>>> from sklearn import tree
>>> feature_names = ['Sepal length', 'Sepal width',
...                   'Petal length', 'Petal width']
>>> tree.plot_tree(tree_model,
...                  feature_names=feature_names,
...                  filled=True)
>>> plt.show()
```

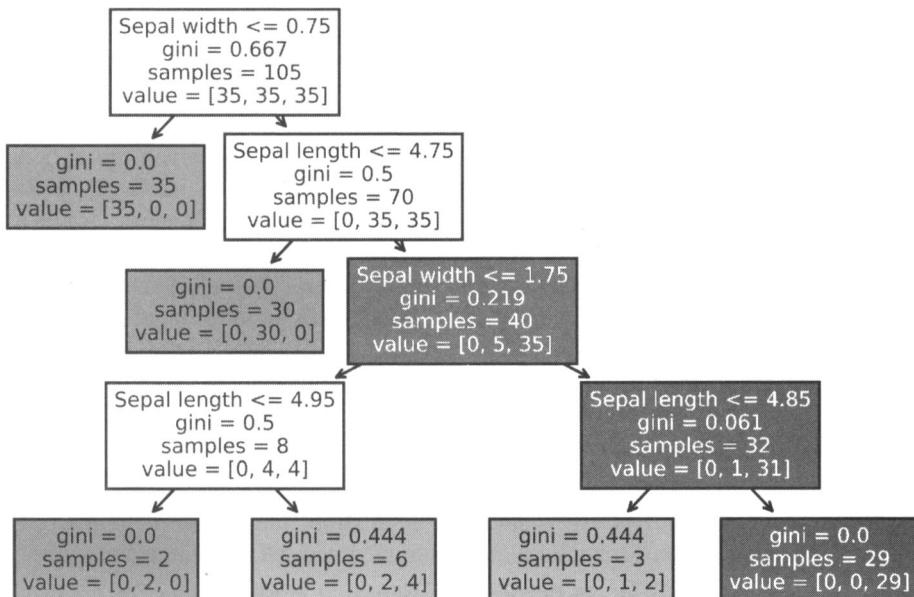


Рис. 3.23. Модель дерева решений, соответствующая набору данных Iris

Установив параметр `filled=True` в функции `plot_tree`, мы вызывали цвета узлов по метке большинства классов в этом узле. Для этой функции доступно множество дополнительных опций, которые вы можете найти в документации по адресу: https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html.

Глядя на схему, показанную на рис. 3.23, мы теперь можем хорошо проследить разбиения, которые дерево решений определило из нашего набора обучающих данных. Что касается критерия разделения признаков в каждом узле, то имейте в виду, что ветви слева соответствуют исходу `True` (Истина), а ветви справа — `False` (Ложь).

Расположенный вверху корневой узел начинается со 105 экземпляров. В первом ветвлении используется пороговый критерий «`sepal width` (ширина чашелистника) ≤ 0.75 см», который делит данные корневого узла на два дочерних узла с 35 экземплярами (слева) и 70 экземплярами (справа). Уже после первого ветвления мы видим, что левый дочерний узел чист от примесей и содержит только экземпляры класса `Iris-setosa` (примесь

Джини = 0). Дальнейшие ветвления в правой части используются для разделения экземпляров классов Iris-versicolor и Iris-virginica.

Рассматривая схему дерева решений и диаграмму решающих областей, можно сделать вывод, что дерево решений очень хорошо разделяет классы цветков. К сожалению, scikit-learn в настоящее время не предоставляет возможности ручной обрезки дерева решений после обучения. Однако мы могли бы вернуться к нашему предыдущему примеру кода, изменить параметр `max_depth` нашего дерева решений на 3 и сравнить результат с текущей моделью. Мы оставим эту модификацию как самостоятельное упражнение для заинтересованного читателя.

В качестве альтернативы scikit-learn предлагает автоматическую процедуру постобрезки деревьев решений по критерию вычислительных затрат².

3.6.3. Объединение нескольких деревьев решений с помощью случайных лесов

Методы объединения моделей в ансамбли за последнее десятилетие приобрели огромную популярность в приложениях машинного обучения из-за их хорошего качества классификации и устойчивости к переобучению. В главе 7 мы рассмотрим различные ансамблевые методы, в том числе бэггинг (bagging) и бустинг (boosting), а сейчас обсудим алгоритм *случайного леса* (random forest) на основе дерева решений, который известен своей хорошей масштабируемостью и простотой использования. Случайный лес можно рассматривать как ансамбль деревьев решений. Идея случайного леса состоит в том, чтобы усреднить несколько деревьев решений (обычно глубоких), которые по отдельности страдают от высокой дисперсии, чтобы построить более надежную модель, имеющую лучшую обобщающую способность, но при этом менее подверженную переобучению. Алгоритм случайного леса можно описать четырьмя простыми шагами:

1. Извлеките случайную *бутстрэп-выборку* (bootstrap sample) размером n (случайным образом выберите n экземпляров из обучающего набора данных с заменой).
2. Вырастите дерево решений из бутстрэп-выборки. В каждом узле:
 - случайно выберите d признаков без возвращения;
 - разделите узел, используя признак, который обеспечивает наилучшее разделение в соответствии с целевой функцией, — например, максимизируя прирост информации.
3. Повторите шаги 1 и 2 k раз.
4. Агрегируйте прогноз по каждому дереву, чтобы присвоить метку класса большинством голосов. О механизме голосования большинством голосов (majority vote) мы расскажем более подробно в главе 7.

Нужно отметить одну небольшую модификацию на *шаге 2*, когда мы обучаем отдельные деревья решений: вместо оценки всех признаков для определения наилучшего разделения в каждом узле мы рассматриваем только случайное их подмножество.

² Заинтересованные читатели могут найти дополнительную информацию по этой более сложной теме в онлайн-руководстве по следующему адресу:

https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html.



Выборка с возвращением и без возвращения

Если вы не знакомы с терминами «выборка с возвращением» (*sampling with replacement*) и «выборка без возвращения» (*sampling without replacement*)³, давайте проведем простой мысленный эксперимент. Предположим, что мы играем в лотерею, в которой случайным образом вытаскиваем числа из корзины. В начале игры корзина содержит пять уникальных чисел: 0, 1, 2, 3 и 4, и мы вынимаем ровно одно число за один раунд. В первом раунде шанс вынуть определенное число из урны будет равен $\frac{1}{5}$. При выборке *без возвращения* мы не кладем число обратно в урну после каждого раунда. Следовательно, вероятность выпадения определенного числа из набора оставшихся чисел в следующем раунде зависит от предыдущего раунда. Например, если в корзине остались числа 0, 1, 2 и 4, шанс вытянуть в следующем раунде число 0 станет равным $\frac{1}{4}$.

Однако при случайной выборке *с возвращением* мы всегда возвращаем извлеченное число в корзину, чтобы вероятность выпадения того или иного числа на каждом ходу не менялась, — т. е. мы можем извлечь одно и то же число более одного раза. Другими словами, при выборке *с возвращением* выпавшие варианты независимы и имеют нулевую ковариацию. Например, результаты пяти раундов розыгрыша случайных чисел могут выглядеть так:

- случайная выборка без возвращения: 2, 1, 3, 4, 0;
- случайная выборка с возвращением: 1, 3, 3, 4, 1.

Хотя случайные леса не обеспечивают такой же уровень интерпретируемости, как деревья решений, большое преимущество случайных лесов заключается в том, что нам не приходится так сильно беспокоиться о выборе хороших значений гиперпараметров. Обычно нам не нужно обрезать случайный лес, поскольку ансамблевая модель достаточно устойчива к шуму за счет усреднения прогнозов отдельных деревьев решений. Единственный параметр, который нам нужно учитывать на практике, — это количество деревьев k (*шаг 3*), которые мы выбираем для случайного леса. Как правило, чем больше количество деревьев, тем выше производительность классификатора случайного леса за счет увеличения вычислительных затрат.

Хотя это менее распространено на практике, есть и другие гиперпараметры классификатора случайного леса, которые можно оптимизировать — с использованием методов, которые мы обсудим в главе 6, — это размер n бутстрэп-выборки (*шаг 1*) и количество признаков d , которые выбираются случайным образом для каждого ветвления (*шаг 2a*) соответственно. С помощью размера n бутстрэп-выборки мы управляем компромиссом между систематической ошибкой и дисперсией случайного леса.

Уменьшение размера бутстрэп-выборки увеличивает разнообразие отдельных деревьев, поскольку вероятность того, что конкретный обучающий пример будет включен в бутстрэп-выборку, снижается. Следовательно, уменьшение размера бутстрэп-выборок может увеличить *случайность* леса и уменьшить эффект переобучения. Однако уменьшение бутстрэп-выборок обычно приводит к снижению общей производительности случайного леса и небольшому разрыву между производительностью обучения и тестирования, а в целом — к низкой производительности тестирования. И наоборот, увеличение размера выборки может увеличить риск переобучения. Поскольку бут-

³ Иногда говорят «выборка с заменой» и «выборка без замены» соответственно.

стрэп-выборки и, следовательно, отдельные деревья решений становятся более похожими друг на друга, они обучаются более точно соответствовать исходному набору обучающих данных.

В большинстве реализаций, включая `RandomForestClassifier` в `scikit-learn`, размер бутстрэп-выборки выбирают равным количеству обучающих экземпляров в исходном наборе обучающих данных, что обычно обеспечивает хороший компромисс смещения и дисперсии. Количество признаков d при каждом ветвлении должно быть меньше, чем общее количество признаков в обучающем наборе данных. Разумным значением по умолчанию, используемым в `scikit-learn` и других реализациях, является $d = \sqrt{m}$, где m — количество признаков в обучающем наборе данных.

Удобно, что нам не нужно самостоятельно создавать классификатор случайного леса из отдельных деревьев решений, потому что в `scikit-learn` уже есть реализация, которую мы можем использовать:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                        classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('Длина лепестка [см]')
>>> plt.ylabel('Ширина лепестка [см]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

После выполнения этого кода мы должны увидеть области принятия решений, сформированные ансамблем деревьев в случайном лесу (рис. 3.24).

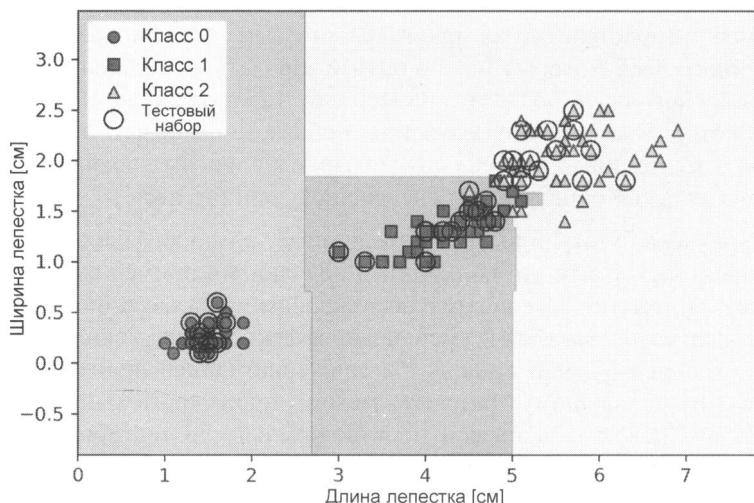


Рис. 3.24. Области решений для набора данных Iris, полученные с использованием случайного леса

Выполнив этот код, мы обучили случайный лес из 25 деревьев решений с помощью параметра `n_estimators`. По умолчанию он задействует меру примеси Джини в качестве критерия для ветвления в узлах. Хотя мы выращиваем очень маленький случайный лес из очень небольшого набора обучающих данных, мы применили в демонстрационных целях параметр `n_jobs`, который позволяет распараллелить обучение модели с использованием нескольких ядер нашего компьютера (здесь — двух ядер). Если вы столкнулись с ошибками при запуске этого кода, возможно, ваш компьютер не поддерживает многопроцессорное выполнение. В таком случае вы можете опустить параметр `n_jobs` или задать для него значение `n_jobs=None`.

3.7. К-ближайшие соседи: «ленивый» алгоритм обучения

Последний алгоритм обучения с учителем, который мы хотим обсудить в этой главе, — это классификатор по методу *k*-ближайших соседей (k-Nearest Neighbor, kNN). Он для нас особенно интересен, поскольку принципиально отличается от алгоритмов обучения, рассмотренных нами до сих пор.

kNN — типичный пример *ленивого обучателя* (lazy learner). Он называется ленивым не из-за его кажущейся простоты, а потому что не строит разграничительную функцию путем старательного изучения набора обучающих данных, а вместо этого просто запоминает набор.



Параметрические и непараметрические модели

Алгоритмы машинного обучения можно разделить на параметрические и непараметрические модели. Используя параметрические модели, мы оцениваем параметры из набора обучающих данных, чтобы получить функцию, способную классифицировать новые точки данных, не нуждаясь более в обучающем наборе. Типичными примерами параметрических моделей являются персептрон, логистическая регрессия и линейный SVM. Напротив, непараметрические модели не могут быть охарактеризованы фиксированным набором параметров, и количество параметров у них меняется в зависимости от количества обучающих данных. Два примера непараметрических моделей, которые мы упоминали, — это классификатор на основе дерева решений / случайного леса и ядерный (но не линейный) SVM.

kNN принадлежит к подкатегории непараметрических моделей, обучаемых на основе экземпляров. Моделям, обучаемым таким образом, свойственно запоминание обучающего набора данных, а ленивое обучение — это частный случай обучения на основе экземпляров, связанный с отсутствием затрат (нулевыми затратами) в процессе обучения.

Сам алгоритм kNN довольно прост и может быть представлен в виде следующих шагов:

1. Выберите количество k и метрику расстояния.
2. Найдите k ближайших соседей записи данных, которую хотите классифицировать.
3. Присвойте текущей записи метку класса большинством голосов.

На рис. 3.25 показано, как новой точке данных (?) присваивается метка класса «треугольник» на основе голосования большинства пяти ее ближайших соседей.

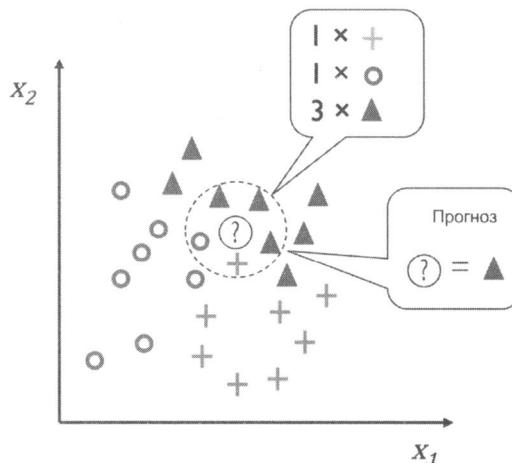


Рис. 3.25. Принцип работы алгоритма k -ближайших соседей

На основе выбранной метрики расстояния алгоритм kNN находит k экземпляров в обучающем наборе данных, которые наиболее близки (наиболее похожи) к точке, которую мы хотим классифицировать. Затем метку класса интересующей нас точки данных определяют большинством голосов среди k ближайших соседей.



Преимущества и недостатки подходов на основе запоминания

Основное преимущество подхода, основанного на запоминании, заключается в том, что классификатор непрерывно адаптируется по мере сбора новых обучающих данных. Однако недостатком является то, что вычислительная сложность для классификации новых экземпляров растет линейно с количеством записей в обучающем наборе данных в худшем случае, если только набор данных не имеет очень мало измерений (признаков), а алгоритм не был реализован с использованием специальной структуры хранения данных для более эффективной обработки запросов к набору. К таким структурам хранения данных относятся k - d tree (https://en.wikipedia.org/wiki/K-d_tree) и ball tree (https://en.wikipedia.org/wiki/Ball_tree) — обе они поддерживаются в scikit-learn. Кроме того, наряду с вычислительными затратами на запрос данных большие наборы также могут вызывать проблемы с точки зрения ограниченной емкости хранилища.

Однако во многих случаях, когда мы работаем с наборами данных относительно небольшого или среднего размера, методы на основе запоминания могут обеспечить хорошую производительность прогнозирования и вычислений и, таким образом, являются хорошим выбором для решения многих реальных проблем. К современным примерам эффективного использования методов ближайших соседей можно отнести прогнозирование свойств мишней фармацевтических препаратов («Machine Learning to Identify Flexibility Signatures of Class A GPCR Inhibition», Biomolecules, 2020, Joe Bemister-Buffington, Alex J. Wolf, Sebastian Raschka, and Leslie A. Kuhn, <https://www.mdpi.com/2218-273X/10/3/454>) и современные языковые модели (Efficient Nearest Neighbor Language Models, 2021, Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick, <https://arxiv.org/abs/2109.04212>).

Выполнив следующий код, мы реализуем модель kNN в scikit-learn с использованием евклидовой метрики расстояния:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                             metric='minkowski')
...
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                        classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('Длина лепестка [стандартизирована]')
>>> plt.ylabel('Ширина лепестка [стандартизирована]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Указав пять соседей в модели kNN для этого набора данных, мы получим относительно плавную решающую границу (рис. 3.26).

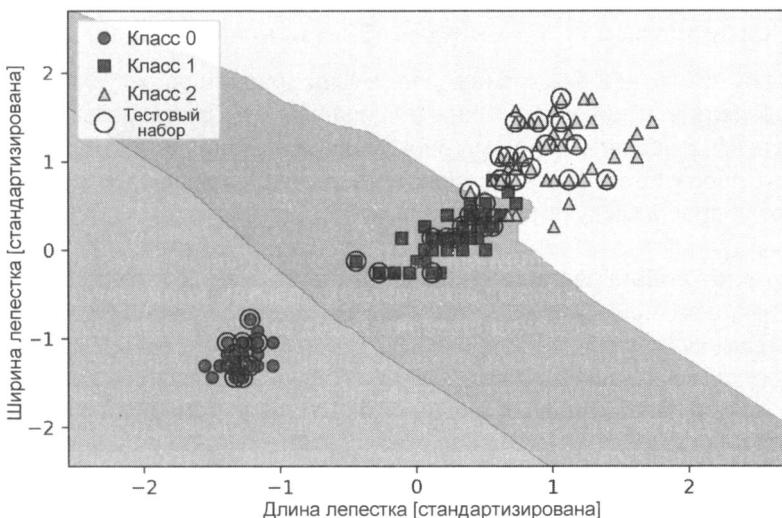


Рис. 3.26. Решающие границы, полученные методом k-ближайших соседей для набора данных Iris



Устранение неоднозначности при равенстве голосов

В случае равенства голосов реализация алгоритма kNN в scikit-learn предпочтет классификацию по соседям с более близким расстоянием до записи данных. Если и соседи имеют одинаковые расстояния, алгоритм выберет метку класса, которая идет первой в наборе обучающих данных.

Правильный выбор значения k имеет решающее значение для нахождения хорошего баланса между переобучением и недообучением. Мы также должны убедиться, что выбрали метрику расстояния, подходящую для признаков в наборе данных. Часто на практике применяется простая евклидова мера расстояния — например, признаки цветков в нашем наборе данных Iris измеряются в сантиметрах. Однако, если мы используем евклидову меру расстояния, также важно стандартизировать данные, чтобы каждый признак вносил одинаковый вклад в расстояние. Расстояние Минковского minkovsky,

использованное нами в предыдущем коде, является просто обобщением евклидова и манхэттенского расстояний, которое можно записать следующим образом:

$$d(x^{(i)}, x^{(j)}) = \sqrt[p]{\sum_k |x_k^{(i)} - x_k^{(j)}|^p}.$$

Оно становится евклидовым расстоянием, если мы устанавливаем параметр $p=2$ или манхэттенским, если $p=1$. В scikit-learn доступны многие другие метрики расстояния — их можно указать в параметре `metric`⁴.

Наконец, важно отметить, что алгоритм kNN очень сильно подвержен переобучению из-за *проклятия размерности* (curse of dimensionality). Проклятие размерности описывает ситуацию, когда пространство признаков становится все более разреженным по мере увеличения числа измерений обучающего набора данных фиксированного размера. К нему можно отнести и случаи, когда даже самые близкие соседи находятся слишком далеко в многомерном пространстве, чтобы дать точную оценку.

В разделе о логистической регрессии мы называли концепцию регуляризации одним из способов избежать переобучения. Однако в моделях, где регуляризация неприменима, таких как деревья решений и kNN, мы можем избежать проклятия размерности, применяя методы выбора признаков и уменьшения размерности. Более подробно мы рассмотрим этот вопрос в следующих двух главах.



Альтернативные реализации машинного обучения с поддержкой GPU

При работе с большими наборами данных запуск алгоритма k-ближайших соседей или подгонка случайных лесов с большим количеством оценок может потребовать значительных вычислительных ресурсов и времени обработки. Если у вас есть компьютер с графическим процессором NVIDIA, совместимым с последними версиями библиотеки NVIDIA CUDA, мы рекомендуем обратить внимание на экосистему RAPIDS (<https://docs.rapids.ai/api>). Например, в библиотеке cuML от RAPIDS (<https://docs.rapids.ai/api/cuml/stable/>) реализованы многие алгоритмы машинного обучения scikit-learn с поддержкой графического процессора для ускорения обработки. Вы можете найти введение в cuML по адресу: https://docs.rapids.ai/api/cuml/stable/estimator_intro.html. Если вам интересно узнать больше об экосистеме RAPIDS, рекомендуем прочесть статью, которую мы написали в сотрудничестве с командой RAPIDS: «Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence» (<https://www.mdpi.com/2078-2489/11/4/193>).

3.8. Заключение

В этой главе вы узнали о различных алгоритмах машинного обучения, которые применяются для решения линейных и нелинейных задач. Вы увидели, что деревья решений особенно привлекательны, если мы заботимся об интерпретируемости, а логистическая

⁴ С полным списком метрик можно ознакомиться по адресу:

<https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html>.

регрессия — это не только полезная модель для онлайн-обучения с помощью SGD, но и возможность предсказать вероятность конкретного события.

Хотя SVM являются мощными линейными моделями, которые можно расширить до нелинейных задач с помощью ядерного трюка, у них есть много параметров, требующих правильно их настроить, чтобы делать хорошие прогнозы. Напротив, ансамблевые методы, такие как случайные леса, не требуют тщательной настройки параметров и не так подвержены переобучению, как деревья решений, что делает их привлекательными моделями для решения многих прикладных задач. Классификатор kNN предлагает альтернативный подход к классификации с помощью ленивого обучения, позволяющий нам делать прогнозы без обучения модели, но с более затратным в вычислительном отношении этапом прогнозирования.

Однако даже более важным фактором, чем выбор подходящего алгоритма обучения, является качество данных в обучающем наборе. Ни один алгоритм не сможет делать хорошие прогнозы без информативных различительных признаков.

В следующей главе мы обсудим важные темы, касающиеся предварительной обработки данных, выбора признаков и уменьшения размерности, в связи с чем нам потребуется создавать мощные модели машинного обучения. Позже, в главе 6, вы научитесь оценивать и сравнивать производительность своих моделей и познакомитесь с полезными приемами точной настройки различных алгоритмов.

4

Предварительная обработка данных для создания качественных обучающих наборов

Качество данных и объем содержащейся в них полезной информации являются ключевыми факторами, определяющими, насколько хорошо алгоритм машинного обучения может обучаться. Поэтому крайне важно тщательно изучить набор данных и подвергнуть его предварительной обработке, прежде чем передать алгоритму машинного обучения. Здесь мы обсудим основные методы предварительной обработки данных, которые помогут нам создавать хорошие модели машинного обучения.

В этой главе будут рассмотрены следующие темы:

- ◆ удаление данных из набора и подстановка отсутствующих значений;
- ◆ приведение категорийных данных к форме, подходящей для алгоритмов машинного обучения;
- ◆ выбор подходящих признаков для построения модели.

4.1. Как поступать с отсутствующими данными?

В реальной жизни обучающие данные не идеальны, и нередко в них по разным причинам отсутствуют значения некоторых полей. В процессе сбора данных могла произойти ошибка, некоторые измерения могли давать недопустимые значения, или определенные поля могли быть просто оставлены пустыми, например в ходе социологического опроса. Обычно пропущенные значения имеют в таблице данных вид пробелов или строк-заполнителей — таких как `NaN`, что означает «not a number» (не число), или `null` (традиционный указатель неизвестных значений в реляционных базах данных). К сожалению, большинство вычислительных алгоритмов не в состоянии обработать такие пропущенные значения и будут давать непредсказуемые результаты, если мы просто их проигнорируем. Поэтому крайне важно позаботиться об этих недостающих значениях, прежде чем мы приступим к дальнейшему анализу.

В этом разделе мы рассмотрим несколько практических методов решения проблемы пропущенных значений путем удаления неполных записей из набора данных или подстановки пропущенных значений из других обучающих экземпляров и признаков.

4.1.1. Выявление пропущенных значений в табличных данных

Прежде чем обсуждать способы работы с отсутствующими значениями, давайте создадим простой пример объекта DataFrame из файла разделенных запятыми значений (Comma-Separated Values, CSV), чтобы лучше понять проблему:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,''''
>>> # Если вы используете Python 2.7, то должны
>>> # конвертировать строку в Unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
   A    B    C    D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 12.0  NaN
```

Используя этот код, мы прочитали данные в формате CSV в DataFrame библиотеки pandas с помощью функции `read_csv` и обнаружили, что две отсутствующие ячейки были заменены значением `NaN`. Функция `StringIO` в этом примере кода использовалась просто для иллюстрации. Она позволила нам прочитать строку, присвоенную `csv_data`, в DataFrame pandas, как если бы это был обычный файл CSV на нашем жестком диске.

В больших объектах DataFrame поиск отсутствующих значений вручную может быть утомительным — в этом случае мы можем применить метод `isnull`, возвращающий DataFrame с логическими значениями, которые указывают, содержит ли ячейка числовое значение (`False`), или данные отсутствуют (`True`). Затем, используя метод `sum`, мы можем вернуть количество пропущенных значений в столбце следующим образом:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

Так мы можем подсчитать количество пропущенных значений в столбце, а в следующих подразделах мы рассмотрим различные стратегии работы с этими отсутствующими данными.



Удобная обработка данных с помощью DataFrame от pandas

Хотя scikit-learn изначально разрабатывали для работы только с массивами NumPy, иногда бывает удобнее выполнять предварительную обработку данных с помощью DataFrame от pandas. В настоящее время большинство функций scikit-learn поддер-

живают объекты DataFrame в качестве входных данных, но поскольку обработка массивов NumPy реализована в API scikit-learn более зрело, рекомендуется по возможности использовать массивы NumPy. Заметьте, что вы всегда можете получить доступ к базовому массиву NumPy DataFrame через атрибут values, прежде чем передать его в оцениватель scikit-learn:

```
>>> df.values
array([[ 1.,   2.,   3.,   4.],
       [ 5.,   6.,   nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

4.1.2. Исключение обучающих записей или признаков с пропущенными значениями

Один из самых простых способов справиться с отсутствующими данными — просто полностью удалить соответствующие признаки (столбцы) или обучающие записи (строки) из набора данных.

Строки с отсутствующими значениями можно легко удалить с помощью метода dropna:

```
>>> df.dropna(axis=0)
      A      B      C      D
0  1.0  2.0  3.0  4.0
```

Точно так же мы можем удалить столбцы, содержащие хотя бы один NaN в любой строке, задав для аргумента axis значение 1:

```
>>> df.dropna(axis=1)
      A      B
0  1.0  2.0
1  5.0  6.0
2 10.0 11.0
```

Метод dropna поддерживает несколько дополнительных параметров, которые могут пригодиться:

```
>>> # удаляет только строки, для которых все столбцы равны NaN
>>> # (возвращает массив, в котором нет строк, содержащих только значения NaN)
>>> df.dropna(how='all')
      A      B      C      D
0  1.0  2.0  3.0  4.0
1  5.0  6.0  NaN  8.0
2 10.0 11.0 12.0  NaN
>>> # удаляет строки, в которых менее 4 реальных значений
>>> df.dropna(thresh=4)
      A      B      C      D
0  1.0  2.0  3.0  4.0
>>> # удаляет только строки, в которых NaN содержится
>>> # только в заданных столбцах (здесь столбец 'C')
>>> df.dropna(subset=['C'])
      A      B      C      D
0  1.0  2.0  3.0  4.0
2 10.0 11.0 12.0  NaN
```

Хотя удаление отсутствующих данных кажется удобным подходом, оно имеет определенные недостатки — например, мы можем в конечном итоге удалить слишком много записей, что сделает невозможным надежный анализ. Или, если удалить слишком много столбцов признаков, мы рискуем потерять ценную информацию, которая нужна нашему классификатору для различения классов. В следующем разделе мы рассмотрим одну из наиболее часто используемых альтернатив для работы с пропущенными значениями: методы интерполяции.

4.1.3. Подстановка пропущенных значений

Зачастую удаление обучающих записей или целых столбцов признаков просто невозможно, потому что мы теряем слишком много ценных данных. В этом случае можно использовать различные методы интерполяции для вычисления недостающих значений из других обучающих записей в нашем наборе данных. Одним из наиболее распространенных методов интерполяции является *подстановка среднего* (mean imputation), когда мы просто заменяем отсутствующее значение средним значением всего столбца признаков. Удобный способ сделать это — использовать класс `SimpleImputer` из `scikit-learn`, как показано в следующем коде:

```
>>> from sklearn.impute import SimpleImputer
>>> import numpy as np
>>> imr = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imr = imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[ 1.,  2.,  3.,  4.],
       [ 5.,  6.,  7.5,  8.],
       [ 10., 11., 12.,  6.]])
```

Здесь мы заменили каждое значение `NaN` соответствующим средним значением, которое рассчитывается отдельно для каждого столбца признаков. Другими вариантами параметра `strategy` являются `median` или `most_frequent`, где последний заменяет отсутствующие значения наиболее часто встречающимся значением. Это полезно для подстановки значений категориальных признаков — например, столбца признаков, в котором хранится кодировка названий цветов, таких как красный, зеленый и синий. Мы встретимся с примерами таких данных позже в этой главе.

Альтернативным и еще более удобным способом подстановки отсутствующих значений является применение метода `fillna` библиотеки `pandas` и передача метода подстановки в качестве аргумента. Например, используя `pandas`, мы могли бы добиться подстановки того же среднего значения непосредственно в объекте `DataFrame` с помощью следующей команды:

```
>>> df.fillna(df.mean())
```

Результат выполнения этой команды показан на рис. 4.1.



Дополнительные методы подстановки отсутствующих данных

Читателям, заинтересованным в изучении дополнительных способов подстановки, включая `KNNImputer`, основанный на методе *k*-ближайших соседей, мы рекомендуем ознакомиться с документацией `scikit-learn` о подстановке по адресу <https://scikit-learn.org/stable/modules/impute.html>.

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

Рис. 4.1. Замена отсутствующих значений в данных средним значением

4.1.4. API оценивателя scikit-learn

В предыдущем разделе мы использовали класс `SimpleImputer` из `scikit-learn` для подстановки отсутствующих значений в наш набор данных. Класс `SimpleImputer` является частью так называемого *API преобразователя* (*transformer API*) в `scikit-learn`, который применяется для реализации классов Python, связанных с преобразованием данных. (Будьте внимательны: API преобразователя `scikit-learn` не следует путать с архитектурой `Transformer`, используемой в обработке естественного языка, которую мы более подробно рассмотрим в главе 16.) Два основных метода этого оценивателя: `fit` и `transform`. Метод `fit` служит для изучения параметров обучающих данных, а метод `transform` использует эти параметры для преобразования данных. Массив данных, который необходимо преобразовать, должен иметь то же количество признаков, что и массив данных, который применялся для обучения модели.

На рис. 4.2 показано, как экземпляр преобразователя `scikit-learn`, обученный на данных, используется для преобразования обучающего набора данных, а также нового набора тестовых данных.

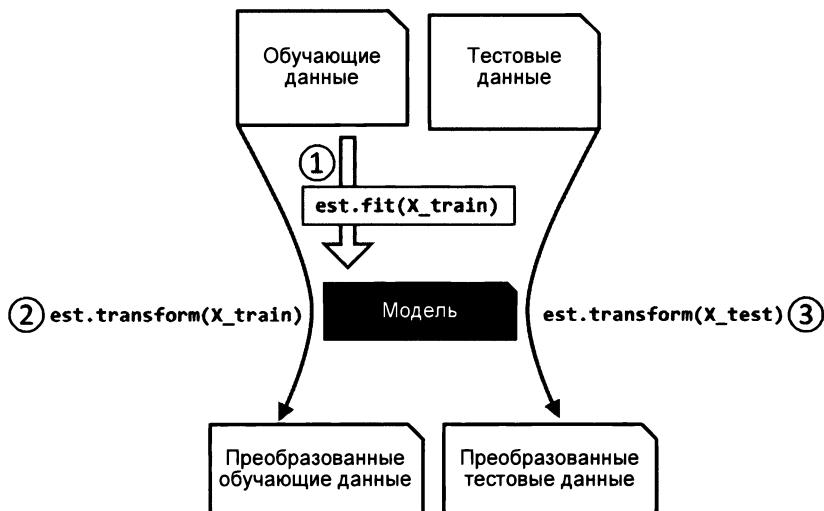


Рис. 4.2. Использование API `scikit-learn` для преобразования данных

Классификаторы, которые мы использовали в главе 3, относятся к так называемым *оценывателям* (*estimator*) в `scikit-learn` с API, который концептуально очень похож на API преобразователя `scikit-learn`. Оценыватели имеют метод `predict`, но также могут иметь и

метод `transform`, с которым мы встретимся далее в этой главе. Как вы, возможно, помните, мы также использовали метод `fit` для подгонки параметров модели, когда обучали эти оцениватели выполнять классификацию. Однако в задачах обучения с учителем мы дополнительно предоставляем метки классов для обучения модели, которую затем можно использовать для прогнозирования новых неразмеченных примеров данных с помощью метода `predict`, как показано на рис. 4.3.

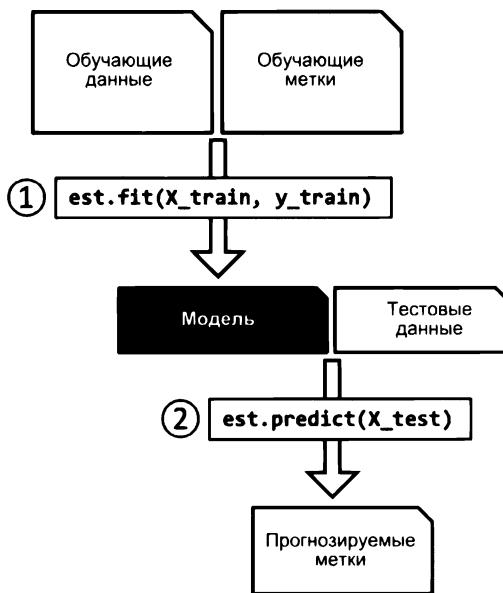


Рис. 4.3. Использование API scikit-learn для прогностических моделей, таких как классификаторы

4.2. Работа с категориальными данными

До сих пор мы работали только с числовыми значениями. Однако реальные наборы данных нередко содержат один или несколько столбцов категориальных признаков. В этом разделе мы рассмотрим простые, но наглядные примеры, и научимся работать с категориальными данными в библиотеках численных методов.

Когда мы говорим о категориальных данных, мы должны дополнительно различать *порядковые*¹ (*ordinal*) и *номинальные* (*nominal*) признаки. Порядковые признаки можно рассматривать как категориальные значения, которые можно сортировать или упорядочивать. Например, размер футболки будет порядковым признаком, потому что мы можем определить порядок: $XL > L > M$. Напротив, номинальные признаки не подразумевают никакого порядка — продолжая предыдущий пример, мы можем рассматривать цвет футболки как номинальный признак, поскольку обычно не имеет смысла говорить, например, что красный больше синего.

¹ Иногда их также называют *ординальными*. — Прим. пер.

4.2.1. Категориальное кодирование данных при помощи pandas

Прежде чем приступить к изучению методов обработки категориальных данных, давайте создадим новый DataFrame с демонстрационными данными:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...     ['green', 'M', 10.1, 'class2'],
...     ['red', 'L', 13.5, 'class1'],
...     ['blue', 'XL', 15.3, 'class2']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
   color  size  price classlabel
0  green    M    10.1      class2
1    red    L    13.5      class1
2   blue   XL    15.3      class2
```

Как можно видеть из вывода этого кода, вновь созданный DataFrame содержит номинальный признак (`color`, цвет), порядковый признак (`size`, размер) и числовой признак (`price`, цена). Метки классов (при условии, что мы создали набор данных для задачи обучения с учителем) хранятся в последнем столбце. Алгоритмы обучения для задач классификации, которые мы рассматриваем в этой книге, не используют порядковую информацию в метках классов.

4.2.2. Сопоставление порядковых признаков

Чтобы убедиться, что алгоритм обучения правильно интерпретирует порядковые признаки, нам нужно преобразовать категориальные строковые значения в целые числа. К сожалению, нет удобной функции, которая могла бы автоматически вывести правильный порядок меток нашего признака `size`, поэтому нам придется выполнить сопоставление вручную. В следующем простом примере предположим, что нам известна числовая разница между признаками — например: $XL = L + 1 = M + 2$:

```
>>> size_mapping = {'XL': 3,
...                  'L': 2,
...                  'M': 1}
>>> df['size'] = df['size'].map(size_mapping)
>>> df
   color  size  price classlabel
0  green    1    10.1      class2
1    red    2    13.5      class1
2   blue    3    15.3      class2
```

Если на более позднем этапе понадобится преобразовать целочисленные значения обратно в исходное строковое представление, достаточно определить словарь с обратным отображением:

```
inv_size_mapping = {v: k for k, v in size_mapping.items()}
```

который затем можно применить с помощью метода `map` библиотеки `pandas` к преобразованному столбцу признаков аналогично словарю `size_mapping`, который мы использовали ранее. Далее показан пример использования:

```
>>> inv_size_mapping = {v: k for k, v in size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0    M
1    L
2   XL
Name: size, dtype: object
```

4.2.3. Кодирование меток классов

Многие библиотеки машинного обучения требуют, чтобы метки классов были закодированы как целочисленные значения. Хотя большинство оценивателей для классификации в `scikit-learn` внутренне преобразуют метки классов в целые числа, считается хорошим тоном представлять метки классов в виде целочисленных массивов, чтобы избежать технических сбоев. Чтобы кодировать метки классов, мы можем использовать подход, аналогичный описанному ранее сопоставлению порядковых признаков. Нам нужно лишь помнить, что метки классов *не* являются порядковыми и не имеет значения, какое целое число мы присваиваем конкретной метке строки. Следовательно, мы можем просто перечислить метки классов, начиная с 0:

```
>>> import numpy as np
>>> class_mapping = {label: idx for idx, label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

а затем использовать словарь сопоставлений для преобразования меток классов в целые числа:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
   color  size  price  classlabel
0   green     1   10.1          1
1     red     2   13.5          0
2    blue     3   15.3          1
```

Мы можем также поменять местами элементы пар ключ-значение в словаре, чтобы сопоставить преобразованные метки классов обратно с исходным строковым представлением:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
   color  size  price  classlabel
0   green     1   10.1      class2
1     red     2   13.5      class1
2    blue     3   15.3      class2
```

В качестве альтернативного способа можно воспользоваться удобным классом `LabelEncoder`, непосредственно реализованным в `scikit-learn`:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([1, 0, 1])
```

Метод `fit_transform` — это просто ярлык для вызова методов `fit` и `transform`, и мы можем использовать метод `inverse_transform` для преобразования целочисленных меток класса обратно в их исходное строковое представление:

```
>>> class_le.inverse_transform(y)
array(['class2', 'class1', 'class2'], dtype=object)
```

4.2.4. Позиционное кодирование номинальных признаков

В разд. 4.2.3 мы использовали простой подход словарного отображения для преобразования функции порядкового признака `size` в целые числа. Поскольку оцениватели scikit-learn для задач классификации обрабатывают метки классов как категориальные данные, которые не подразумевают никакого порядка (номинальные), мы применили удобный метод `LabelEncoder` для кодирования меток строк в целые числа. Мы могли бы прибегнуть к аналогичному подходу для преобразования номинального столбца `color` в нашем наборе данных следующим образом:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

После выполнения этого кода первый столбец `X` массива NumPy теперь содержит новые значения `color`, которые кодируются следующим образом:

- ◆ blue = 0;
- ◆ green = 1;
- ◆ red = 2.

Если мы остановимся на этом месте и отправим массив в классификатор, то совершим одну из самых распространенных ошибок при работе с категориальными данными. Можете ли вы сказать, в чем проблема? Хотя значения цвета не располагаются в каком-либо определенном порядке, обычные модели-классификаторы, такие как рассмотренные в предыдущих главах, теперь будут предполагать, что зеленый больше синего, а красный больше зеленого. Хотя это ошибочное предположение, классификатор все же может дать полезные результаты. Однако эти результаты не будут оптимальными.

Распространенным решением указанной проблемы является использование метода, называемого *прямым позиционным кодированием*, или *унитарным кодированием* (*one-hot encoding*). Идея этого подхода заключается в создании нового фиктивного признака для каждого уникального значения в номинальном столбце признаков. Продолжая работу

с нашим примером, преобразуем признак `color` в три новых признака: `blue`, `green` и `red`. Затем применим двоичные значения для указания конкретного цвета образца — например, синий цвет футболки может быть закодирован так: `blue = 1`, `green = 0`, `red = 0`. Чтобы выполнить это преобразование, воспользуемся кодировщиком `OneHotEncoder`, реализованным в модуле `preprocessing` библиотеки `scikit-learn`:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> X = df[['color', 'size', 'price']].values
>>> color_ohe = OneHotEncoder()
>>> color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

Обратите внимание, что мы применили `OneHotEncoder` только к одному столбцу (`X[:, 0].reshape(-1, 1)`) — чтобы избежать изменения двух других столбцов в массиве. Если необходимо выборочно преобразовать столбцы в массиве из нескольких признаков, используйте `ColumnTransformer`, который принимает список кортежей (`name`, `transformer`, `column(s)`) следующим образом:

```
>>> from sklearn.compose import ColumnTransformer
>>> X = df[['color', 'size', 'price']].values
>>> c_transf = ColumnTransformer([
...     ('onehot', OneHotEncoder(), [0]),
...     ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
array([[0.0, 1.0, 0.0, 1, 10.1],
       [0.0, 0.0, 1.0, 2, 13.5],
       [1.0, 0.0, 0.0, 3, 15.3]])
```

В этом примере кода с помощью аргумента `'passthrough'` мы указали, что хотим изменить только первый столбец и оставить два других столбца нетронутыми.

Еще более удобный способ создания фиктивных признаков с помощью позиционного кодирования — использовать метод `get_dummies`, реализованный в `pandas`. Применительно к `DataFrame` метод `get_dummies` будет преобразовывать только строковые столбцы и оставлять все остальные столбцы без изменений:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
   price  size  color_blue  color_green  color_red
0    10.1     1          0           1           0
1    13.5     2          0           0           1
2    15.3     3          1           0           0
```

Применяя наборы данных с позиционным кодированием, нужно помнить, что им присуща мультиколлинеарность, которая может быть проблемой для определенных методов (например, методов, требующих инверсии матриц). Если признаки сильно коррелированы, матрицы сложно инвертировать с вычислительной точки зрения, что может привести к численно нестабильным оценкам. Чтобы уменьшить корреляцию между переменными, достаточно удалить один столбец признаков из массива с позиционным кодированием признаков. Обратите внимание, что удаление столбца признаков в таком

случае не ведет к утрате важной информации — например, если мы удалим столбец `color_blue`, информация об объектах все равно сохранится, — ведь если мы видим признаки `color_green=0` и `color_red=0`, это однозначно свидетельствует, что футболка синяя.

Используя функцию `get_dummies`, мы можем удалить первый столбец, передав аргумент `True` параметру `drop_first`, как показано в следующем примере кода:

```
>>> pd.get_dummies(df[['price', 'color', 'size']],  
...                  drop_first=True)  
   price  size  color_green  color_red  
0    10.1     1            1            0  
1    13.5     2            0            1  
2    15.3     3            0            0
```

Чтобы удалить избыточный столбец через `OneHotEncoder`, нам нужно задать параметры `drop='first'` и `category='auto'` следующим образом:

```
>>> color_ohe = OneHotEncoder(categories='auto', drop='first')  
>>> c_transf = ColumnTransformer([  
...           ('onehot', color_ohe, [0]),  
...           ('nothing', 'passthrough', [1, 2])  
... ])  
>>> c_transf.fit_transform(X).astype(float)  
array([[ 1. ,  0. ,  1. , 10.1],  
       [ 0. ,  1. ,  2. , 13.5],  
       [ 0. ,  0. ,  3. , 15.3]])
```



Дополнительные схемы кодирования номинальных признаков

Хотя прямое позиционное кодирование является наиболее распространенным способом кодирования неупорядоченных категориальных переменных, существует несколько альтернативных методов. Некоторые из этих методов могут быть полезны при работе с категориальными признаками, имеющими большое количество элементов (большое количество уникальных меток категорий). Примерами таких методов могут служить:

- *двоичное кодирование* (binary encoding), при котором создается несколько двоичных признаков, аналогичных позиционному кодированию, но требуется меньшее количество столбцов признаков, т. е. $\log_2(K)$ вместо $K - 1$, где K — количество уникальных категорий. В двоичном кодировании числа сначала преобразуются в двоичные представления, а затем каждая позиция двоичного числа образует новый столбец признаков;
- *подсчет* (count) или *частотное кодирование* (frequency encoding), при котором метка каждой категории заменяется количеством или частотой ее появления в обучающем наборе.

Эти методы, а также дополнительные схемы категориального кодирования доступны в совместимой со `scikit-learn` библиотеке `category_encoders`, которую можно скачать по адресу: https://contrib.scikit-learn.org/category_encoders/.

Хотя не гарантируется, что эти методы будут работать лучше, чем позиционное кодирование с точки зрения производительности модели, мы можем рассматривать выбор схемы категориального кодирования в качестве дополнительного «гиперпараметра» для улучшения производительности модели.

4.2.5. Кодирование порядковых признаков

Если нет уверенности в числовых различиях между категориями порядковых признаков, или разница между двумя порядковыми значениями не определена, мы можем за- кодировать их, используя *пороговое кодирование* со значениями 0/1. Например, можно разделить признак `size` со значениями M, L и XL на два новых признака: $x > M$ и $x > L$. Рассмотрим исходный DataFrame:

```
>>> df = pd.DataFrame([['green', 'M', 10.1,
...                      'class2'],
...                      ['red', 'L', 13.5,
...                      'class1'],
...                      ['blue', 'XL', 15.3,
...                      'class2']])
>>> df.columns = ['color', 'size', 'price',
...                 'classlabel']
>>> df
```

Мы можем применить метод `apply` объекта `DataFrame` `pandas` для написания пользовательских лямбда-выражений, кодирующих эти переменные с использованием порогового значения:

```
>>> df['x > M'] = df['size'].apply(  
...     lambda x: 1 if x in ('L', 'XL') else 0)  
>>> df['x > L'] = df['size'].apply(  
...     lambda x: 1 if x == 'XL' else 0)  
>>> del df['size']  
>>> df
```

4.3. Разделение набора данных на обучающие и тестовые наборы

В главах 1 и 3 мы кратко упомянули о разделении исходного набора данных на отдельные наборы: для обучения и для тестирования. Прежде чем мы доверим модели работу в реальном мире, необходимо строго и беспристрастно оценить ее качество, сравнивая прогнозы с реальными метками тестового набора. В этом разделе мы подготовим новый набор данных Wine. А после предварительной обработки набора данных мы изучим различные методы выбора признаков, чтобы уменьшить размерность набора данных.

Набор данных Wine — это еще один набор данных с открытым исходным кодом, доступный в репозитории машинного обучения UCI (<https://archive.ics.uci.edu/ml/datasets/Wine>). Он состоит из 178 образцов вин с 13 характеристиками, описывающими их различные химические свойства.

Используя библиотеку `pandas`, мы будем напрямую читать набор данных Wine с открытым исходным кодом из репозитория машинного обучения UCI:

```
>>> df_wine.columns = ['Class label', 'Alcohol',
...                     'Malic acid', 'Ash',
...                     'Alcalinity of ash', 'Magnesium',
...                     'Total phenols', 'Flavanoids',
...                     'Nonflavanoid phenols',
...                     'Proanthocyanins',
...                     'Color intensity', 'Hue',
...                     'OD280/OD315 of diluted wines',
...                     'Proline']
>>> print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```



Получение набора данных Wine

Вы можете найти копию этого набора данных (и всех других наборов данных, используемых в книге) в ее файловом архиве, когда работаете в автономном режиме, или если сервер UCI по адресу: <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> временно недоступен. Например, чтобы загрузить набор данных Wine из локального каталога, нужно заменить в приведенном коде строки:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

на следующие:

```
df = pd.read_csv(
    'your/local/path/to/wine.data', header=None
)
```

Фрагмент набора данных Wine, который содержит 13 различных признаков, описывающих химические свойства 178 образцов вина, представлен на рис. 4.4.

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Рис. 4.4. Фрагмент набора данных Wine

Записи в наборе относятся к одному из трех разных классов: 1, 2 и 3, которые, в свою очередь, обозначают три разновидности винограда, выращенные в одном и том же регионе Италии, но представляющие три разных ботанических сорта, как описано в сводке набора данных (<https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names>).

Для разделения этого набора на обучающие и тестовые данные удобно использовать функцию `train_test_split` из подмодуля `scikit-learn model_selection`:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                     test_size=0.3,
...                     random_state=0,
...                     stratify=y)
```

Сначала мы присвоили представление столбцов признаков 1–13 в виде массива NumPy переменной `x` и метки классов из первого столбца переменной `y`. Затем использовали функцию `train_test_split` для случайного разделения `x` и `y` на отдельные обучающие и тестовые наборы данных.

Установив параметр `test_size=0.3`, мы присвоили 30 процентов записей исходного набора Wine массивам `X_test` и `y_test`, а оставшиеся 70 процентов были назначены массивам `X_train` и `y_train` соответственно. Передача массива меток класса `y` в качестве аргумента для стратификации гарантирует, что и обучающий, и тестовый набор данных имеет те же пропорции классов, что и исходный набор данных.



Выбор правильного соотношения при разделении набора данных

Следует помнить, что, разделяя исходный набор данных на части, мы скрываем ценную информацию, которая может быть полезна для обучения модели. Поэтому не следует выделять слишком много данных для тестового набора. С другой стороны, чем меньше тестовый набор, тем грубее оценка ошибки обобщения модели. Разделение на обучающий и тестовый наборы — это всегда поиск компромисса. На практике чаще всего используют разделение в пропорции 60:40, 70:30 или 80:20 в зависимости от размера исходного набора данных. Однако для больших наборов данных также распространены и уместны разбиения 90:10 или даже 99:1. Например, если набор данных содержит более 100 тыс. записей, для тестирования можно выделить только 10 тыс. из них, и все равно получить точную оценку обобщающей способности модели. Дополнительную информацию и иллюстрации можно найти в первом разделе моей статьи «*Model evaluation, model selection, and algorithm selection in machine learning*» («Оценка модели, выбор модели и выбор алгоритма в машинном обучении»), которая находится в свободном доступе по адресу: <https://arxiv.org/pdf/1811.12808.pdf>. Кроме того, мы вернемся к теме оценки модели и обсудим ее более подробно в главе 6.

Надо также отметить, что обычно принято не отбрасывать выделенные тестовые данные после обучения и оценки модели, а выполнять повторное обучение классификатора уже на полном наборе данных, поскольку это может улучшить прогностическую эффективность модели. Впрочем, хотя такой подход часто рекомендуют, он может привести к снижению качества обобщения модели, если набор данных мал, а тестовый набор данных содержит, например, выбросы. Кроме того, после обучения модели на полном наборе данных у нас не останется независимых данных для оценки ее производительности.

4.4. Приведение признаков к одному масштабу

Масштабирование признаков (feature scaling) — важный шаг в нашем конвейере предварительной обработки, о котором легко можно забыть. *Дерево решений* (decision tree) и *случайный лес* (random forest) — два из очень немногих алгоритмов машинного обучения, в которых нам не нужно беспокоиться о масштабировании признаков. Эти алгоритмы инвариантны к масштабу. Однако большинство алгоритмов машинного обучения и оптимизации ведут себя намного лучше, если признаки находятся в одном масштабе, как вы видели в главе 2, когда рассматривали пример алгоритма оптимизации по методу градиентного спуска.

Важность масштабирования признаков можно проиллюстрировать на простом примере. Предположим, что у нас есть два признака, один из которых измеряется по шкале от 1 до 10, а второй — по шкале от 1 до 100 000 соответственно.

Если в этой ситуации мы применим функцию квадрата ошибки Adaline из главы 2, то алгоритм будет в основном занят оптимизацией весов в соответствии с большими ошибками второго признака. В качестве еще одного примера можно привести алгоритм *k*-ближайших соседей (*kNN*) с евклидовой мерой расстояния — вычисленные расстояния между точками данных тоже будут определяться второй осью признаков.

Существуют два распространенных подхода к согласованию масштабов различных признаков: *нормализация* (normalization) и *стандартизация* (standardization). Эти термины часто применяются в разных областях, и их значение следует понимать с учетом контекста. Чаще всего нормализация означает приведение масштаба признаков к диапазону [0, 1], что является частным случаем *минимаксного масштабирования* (min-max scaling). Чтобы нормализовать данные этим способом, мы применяем минимаксное масштабирование к каждому столбцу признаков, где новое значение $x_{norm}^{(i)}$ записи $x^{(i)}$ можно рассчитать следующим образом:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}.$$

Здесь $x^{(i)}$ — определенная точка данных, x_{min} — наименьшее значение в столбце признаков, а x_{max} — наибольшее значение.

Процедура минимаксного масштабирования реализована в scikit-learn и может применяться следующим образом:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Хотя нормализация с помощью минимаксного масштабирования является широко распространенным методом, который полезен, когда нам нужны значения в ограниченном интервале, для многих алгоритмов машинного обучения, особенно для алгоритмов оптимизации, таких как градиентный спуск, более предпочтительной может оказаться стандартизация. Причина в том, что многие линейные модели, такие как логистическая регрессия и SVM, рассмотренные в главе 3, инициализируют веса нулями или небольшими случайными значениями, близкими к 0. Используя стандартизацию, мы центрируем столбцы признаков по среднему значению 0 со стандартным отклонением 1, так

что столбцы признаков имеют те же параметры, что и стандартное нормальное распределение (нулевое среднее значение и единичная дисперсия), что упрощает обучение весов. Однако подчеркнем, что стандартизация не меняет формы распределения и не превращает ненормально распределенные данные в нормально распределенные. В дополнение к масштабированию данных таким образом, чтобы они имели нулевое среднее значение и единичную дисперсию, стандартизация сохраняет полезную информацию о выбросах и делает алгоритм менее чувствительным к ним, в отличие от масштабирования, которое сводит данные до ограниченного диапазона значений.

Процедура стандартизации может быть выражена следующим уравнением:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}.$$

Здесь μ_x — выборочное среднее для определенного столбца признаков, а σ_x — соответствующее стандартное отклонение.

В табл. 4.1 показана разница между двумя широко используемыми методами масштабирования объектов: стандартизацией и нормализацией — на простом примере набора данных, состоящего из чисел от 0 до 5.

Таблица 4.1. Сравнение между стандартизацией и минимаксной нормализацией

Вход	Стандартизация	Минимаксная нормализация
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

Показанные в таблице стандартизацию и нормализацию можно выполнить при помощи следующего кода:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('стандартизованы:', (ex - ex.mean()) / ex.std())
стандартизованы: [-1.46385011 -0.87831007 -0.29277002 0.29277002
0.87831007 1.46385011]
>>> print('нормализованы:', (ex - ex.min()) / (ex.max() - ex.min()))
нормализованы: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Подобно классу `MinMaxScaler`, `scikit-learn` также реализует класс для стандартизации:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Опять же, важно подчеркнуть, что мы настраиваем класс `StandardScaler` только один раз — по обучающим данным — и в дальнейшем используем эти параметры для преобразования набора тестовых данных или любой новой точки данных.

В scikit-learn доступны и другие, более продвинутые, методы масштабирования признаков — например, `RobustScaler`. Этот метод особенно полезен и рекомендуется к применению, если мы работаем с небольшими наборами данных, которые содержат много выбросов. Аналогично `RobustScaler` может стать хорошим выбором, если алгоритм машинного обучения, применяемый к такому набору данных, склонен к переобучению. Работая с каждым столбцом признаков независимо, `RobustScaler` удаляет медианное значение и масштабирует набор данных в соответствии с 1-м и 3-м квартилями набора данных (т. е. 25-м и 75-м квентилями соответственно), так что самые экстремальные значения и выбросы становятся менее выраженными. Заинтересованный читатель может найти дополнительную информацию о `RobustScaler` в официальной документации scikit-learn по адресу: <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html>.

4.5. Выбор значимых признаков

Если вы заметили, что модель работает намного лучше на обучающем наборе данных, чем на тестовом наборе, это явный признак переобучения. Как было отмечено в главе 3, переобучение означает, что модель слишком точно подстроилась под параметры конкретных экземпляров в обучающем наборе данных, но плохо обобщает новые данные. В этом случае мы говорим, что модель *имеет высокую дисперсию*. Причина переобучения в том, что наша модель слишком сложна для обучающих данных. Есть ряд распространенных способов борьбы с переобучением:

- ◆ собрать больше обучающих данных;
- ◆ ввести штраф за сложность через регуляризацию;
- ◆ выбрать более простую модель с меньшим количеством параметров;
- ◆ уменьшить размерность данных.

Сбор большего количества обучающих данных часто невозможен. В главе 6 вы узнаете о методе проверки полезности дополнительных обучающих данных. В следующих разделах мы рассмотрим распространенные способы противодействия переобучению путем регуляризации и уменьшения размерности за счет правильного выбора признаков, что приводит к более простым моделям, требующим меньшего количества параметров для соответствия данным. А в главе 5 познакомимся с дополнительными методами извлечения признаков.

4.5.1. Регуляризация L1 и L2 как штраф за сложность модели

В главе 3 вы узнали, что *регуляризация L2* — это один из подходов к снижению сложности модели путем начисления штрафа за большие индивидуальные веса. Мы определили квадрат нормы L2 нашего вектора весов w следующим образом:

$$L2: \|w\|_2^2 = \sum_{j=1}^m w_j^2.$$

Другим подходом к снижению сложности модели является *регуляризация L1*:

$$L1: \quad \|w\|_1 = \sum_{j=1}^m |w_j|.$$

Здесь мы просто заменили квадрат весов суммой абсолютных значений весов. В отличие от регуляризации L2, регуляризация L1 обычно дает разреженные векторы признаков, и вес большинства признаков будет равен нулю. Разреженность может быть полезна на практике, если у нас есть многомерный набор данных со многими нерелевантными признаками, особенно в тех случаях, когда у нас больше нерелевантных измерений, чем обучающих экземпляров данных. В этом смысле регуляризацию L1 можно рассматривать как метод выбора признаков.

4.5.2. Геометрическая интерпретация регуляризации L2

Как отмечалось в предыдущем разделе, регуляризация L2 добавляет штрафной член к функции потерь, что приводит к менее экстремальным значениям весов по сравнению с моделью, обученной с нерегуляризованной функцией потерь.

Чтобы лучше понять, как регуляризация L1 способствует разреженности, давайте вернемся на шаг назад и рассмотрим геометрическую интерпретацию регуляризации, для чего построим контуры выпуклой функции потерь для двух весовых коэффициентов: w_1 и w_2 .

Функцию потерь мы определим здесь в форме *среднеквадратичной ошибки* (Mean Squared Error, MSE), которую использовали для Adaline в главе 2. Эта функция вычисляет квадраты расстояний между истинными и предсказанными метками классов: y и \hat{y} , усредненные по всем N записям в обучающем наборе. Поскольку MSE имеет сферическую форму, ее легче визуализировать, чем функцию потерь логистической регрессии, однако при этом применяются те же принципы. Помните, что наша цель — найти комбинацию весовых коэффициентов, при которых функция потерь на обучающих данных минимальна, как показано на рис. 4.5 (точка в центре эллипсов).

Мы можем рассматривать регуляризацию как добавление штрафного члена к функции потерь для поощрения меньших весов — другими словами, мы наказываем большие веса модели. Таким образом, увеличивая силу регуляризации с помощью параметра регуляризации λ , мы смещаем веса в сторону нуля и уменьшаем зависимость нашей модели от обучающих данных. Рис. 4.6 иллюстрирует эту идею применительно к слагаемому штрафа L2. Член квадратичной регуляризации L2 представлен здесь заштрихованным шаром.

Наши весовые коэффициенты не могут превышать бюджет регуляризации — т. е. комбинация весовых коэффициентов не может выходить за пределы заштрихованной области. С другой стороны, мы по-прежнему стремимся минимизировать функцию потерь. При наличии штрафного ограничения наша лучшая попытка состоит в том, чтобы выбрать точку, в которой шар L2 пересекается с контурами функции потерь без штрафа. Чем больше становится значение параметра регуляризации λ , тем быстрее растет штрафной компонент потери, что приводит к более узкому шару L2. Например, если мы увеличим параметр регуляризации до бесконечности, весовые коэффициенты станут фактически равными нулю, что соответствует центру шара L2. По сути, наша цель

состоит в том, чтобы минимизировать сумму чистой потери и штрафного члена, что можно рассматривать как добавление смещения в сторону выбора более простой модели для уменьшения дисперсии при отсутствии достаточного объема обучающих данных.

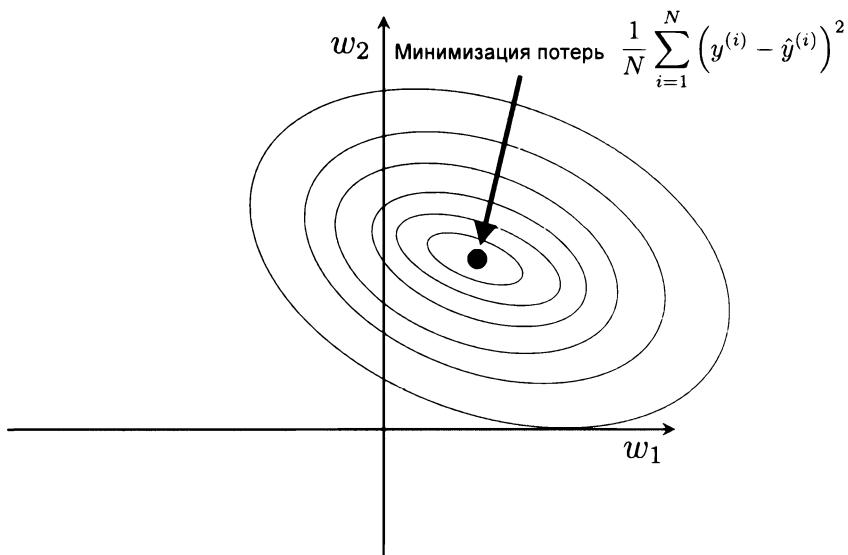


Рис. 4.5. Минимизация функции потерь среднеквадратичной ошибки

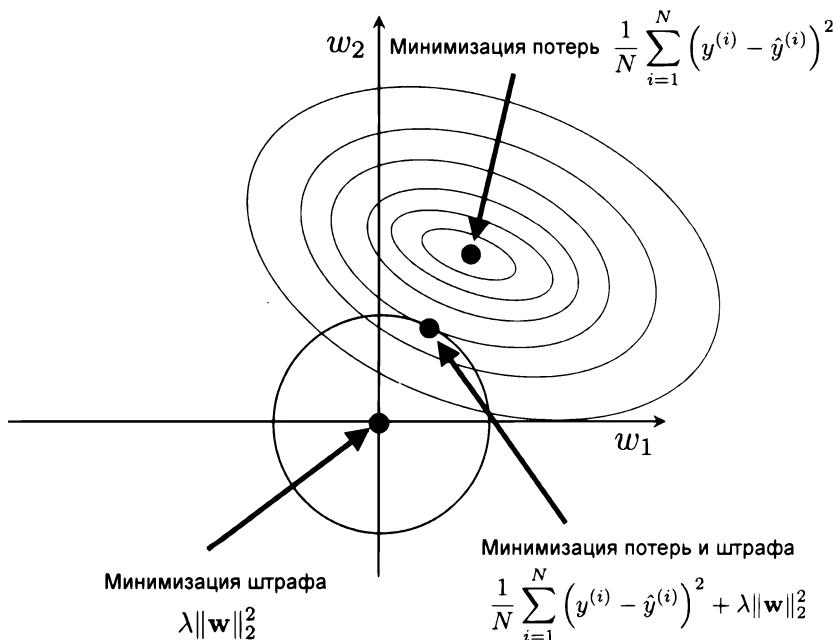


Рис. 4.6. Применение регуляризации L2 к функции потерь

4.5.3. Разреженные решения с регуляризацией L1

Теперь давайте рассмотрим регуляризацию L1 и вытекающую из нее разреженность. Базовая идея регуляризации L1 аналогична той, что мы обсуждали в предыдущем разделе. Однако, поскольку штраф L1 представляет собой сумму абсолютных весовых коэффициентов (помните, что член L2 квадратичен), мы можем представить его в виде ромбовидного бюджета (рис. 4.7).

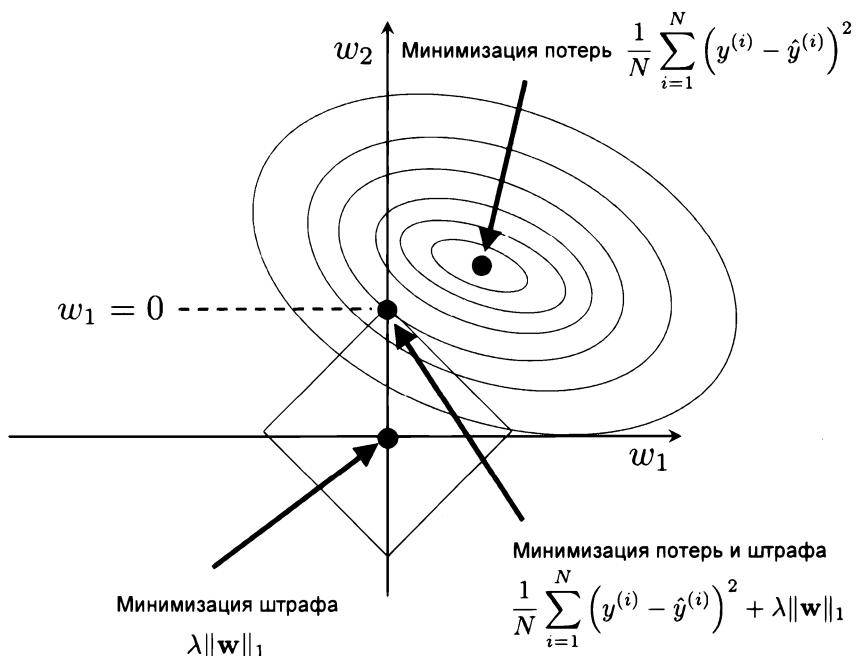


Рис. 4.7. Применение регуляризации L1 к функции потерь

Как можно здесь видеть, контур функции потерь касается ромба L1 при $w_1 = 0$. Поскольку контуры регуляризованной системы L1 резкие, более вероятно, что оптимум, т. е. пересечение между эллипсом функции потерь и границей ромба L1, расположен на осях, что способствует разреженности.



Регуляризация L1 и разреженность

Математическое обоснование того, почему регуляризация L1 может привести к разреженным решениям, выходит за рамки этой книги. Если вам интересно, отличное объяснение регуляризации L2 по сравнению с L1 можно найти в разд. 3.4 книги «The Elements of Statistical Learning» by Trevor Hastie, Robert Tibshirani, Jerome Friedman, Springer Science+Business Media, 2009.

Для регуляризованных моделей в scikit-learn, поддерживающих регуляризацию L1, достаточно установить для параметра штрафа значение '11', чтобы получить разреженное решение:

```
>>> from sklearn.linear_model import LogisticRegression  
>>> LogisticRegression(penalty='l1',  
...                      solver='liblinear',  
...                      multi_class='ovr')
```

Обратите внимание, что нам также необходимо выбрать другой алгоритм оптимизации (например, `solver='liblinear'`), поскольку '`lbfgs`' в настоящее время не поддерживает оптимизацию потерь с регуляризацией L1. Применительно к стандартизованным данным Wine L1-регуляризованная логистическая регрессия даст следующее разреженное решение:

```
>>> lr = LogisticRegression(penalty='l1',  
...                          C=1.0,  
...                          solver='liblinear',  
...                          multi_class='ovr')  
>>> # Значение C=1.0 по умолчанию. Вы можете увеличить или  
>>> # уменьшить его, чтобы сделать эффект регуляризации  
>>> # сильнее или слабее соответственно.  
>>> lr.fit(X_train_std, y_train)  
>>> print('Точность при обучении:', lr.score(X_train_std, y_train))  
Точность при обучении: 1.0  
>>> print('Точность при тестировании:', lr.score(X_test_std, y_test))  
Точность при тестировании: 1.0
```

Высокие значения точности на обучающих и тестовых данных (оба 100 процентов) показывают, что наша модель отлично справляется с обоими наборами данных. Когда мы обращаемся к членам точки пересечения через атрибут `lr.intercept_`, то видим, что массив возвращает три значения:

```
>>> lr.intercept_  
array([-1.26317363, -1.21537306, -2.37111954])
```

Поскольку мы обучаем объект `LogisticRegression` на многоклассовом наборе данных с помощью метода *один против всех* (One-versus-Rest, OvR), первое пересечение принадлежит модели, которая обучена классу 1 в противовес классам 2 и 3, второе значение — это пересечение модели, которая обучена классу 2 в противовес классам 1 и 3, а третье значение — это пересечение модели, которая обучена классу 3 в противовес классам 1 и 2:

```
>>> lr.coef_  
array([[ 1.24647953,  0.18050894,  0.74540443, -1.16301108,  
        0.          ,  0.          ,  1.16243821,  0.          ,  
        0.          ,  0.          ,  0.          ,  0.55620267,  
        2.50890638],  
       [-1.53919461, -0.38562247, -0.99565934,  0.36390047,  
        -0.05892612,  0.          ,  0.66710883,  0.          ,  
        0.          , -1.9318798 ,  1.23775092,  0.          ,  
        -2.23280039],  
       [ 0.13557571,  0.16848763,   0.35710712,  0.          ,  
        0.          ,  0.          , -2.43804744,  0.          ,  
        0.          ,  1.56388787, -0.81881015, -0.49217022,  
        0.         ]])
```

Массив весов, доступ к которому мы получили через атрибут `lr.coef_`, содержит три строки весовых коэффициентов — по одному вектору весов для каждого класса. Каждая строка состоит из 13 весов, где каждый вес умножается на соответствующий признак в 13-мерном наборе данных Wine для расчета входных данных сети:

$$z = w_1x_1 + \dots + w_mx_m + b = \sum_{j=1}^m x_jw_j + b = \mathbf{w}^T\mathbf{x} + b.$$



Получение смещений и параметров весов оценивателей scikit-learn

В библиотеке scikit-learn параметр `intercept_` соответствует смещению, а `coef_` соответствует значениям w_j .

В результате регуляризации L1, которая, как уже упоминалось, служит методом выбора признаков, мы только что обучили модель, устойчивую к потенциально нерелевантным признакам в этом наборе данных. Однако, строго говоря, векторы весов из предыдущего примера не обязательно разрежены, потому что они содержат больше ненулевых, чем нулевых элементов. Но мы могли бы обеспечить разреженность (больше нулевых элементов), еще больше увеличив влияние регуляризации, т. е. выбрав более низкие значения для параметра C.

В последнем примере регуляризации в этой главе мы будем менять влияние регуляризации и построим путь регуляризации — весовые коэффициенты различных признаков для разных уровней регуляризации:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...             'magenta', 'yellow', 'black',
...             'pink', 'lightgreen', 'lightblue',
...             'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4., 6.):
...     lr = LogisticRegression(penalty='l1', C=10.**c,
...                             solver='liblinear',
...                             multi_class='ovr', random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...               label=df_wine.columns[column + 1],
...               color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('Весовой коэффициент')
>>> plt.xlabel('C (обратный уровень регуляризации)')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
```

```
>>> ax.legend(loc='upper center',
...             bbox_to_anchor=(1.38, 1.03),
...             ncol=1, fancybox=True)
>>> plt.show()
```

Полученный график (рис. 4.8) дает нам дополнительное представление о поведении регуляризации L1. Как следует из графика, все веса признаков будут равны нулю, если мы штрафуем модель сильным параметром регуляризации ($C < 0.01$). Здесь C является обратной величиной по отношению к параметру регуляризации λ .

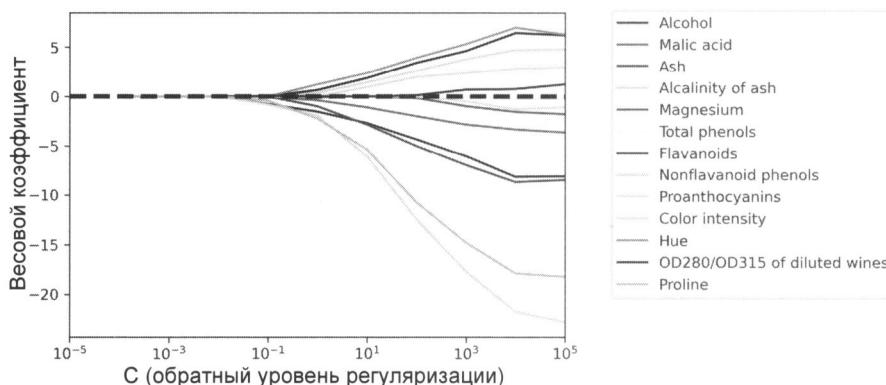


Рис. 4.8. Влияние гиперпараметра величины регуляризации C

4.5.4. Последовательные алгоритмы отбора признаков

Альтернативный способ уменьшить сложность модели и избежать переобучения — уменьшить ее размерность за счет выбора признаков, что особенно полезно для нерегуляризованных моделей. Существуют две основные категории методов уменьшения размерности: *отбор признаков* (feature selection) и *извлечение признаков* (feature extraction). В первом случае мы выбираем подмножество исходных признаков, тогда как во втором случае извлекаем информацию из набора признаков для построения нового подпространства признаков.

В этом разделе мы рассмотрим классическое семейство алгоритмов отбора признаков, а из следующей главы вы узнаете о различных методах извлечения признаков для сжатия набора данных в подпространство признаков более низкой размерности.

Последовательные алгоритмы отбора признаков представляют собой семейство жадных алгоритмов поиска, которые используются для сведения начального d -мерного пространства признаков к k -мерному подпространству признаков, где $k < d$. Ключевая идея алгоритмов отбора признаков заключается в автоматическом выборе такого подмножества признаков, наиболее релевантных задаче, при котором повышается эффективность вычислений или уменьшается ошибка обобщения модели за счет удаления нерелевантных признаков или шума. Это может быть полезно для алгоритмов, которые не поддерживают регуляризацию.

Классический последовательный алгоритм отбора признаков — это *последовательное пошаговое исключение* (Sequential Backward Selection, SBS), целью которого является

уменьшение размерности начального подпространства признаков с минимальным снижением производительности классификатора для повышения эффективности вычислений. В некоторых случаях SBS может даже улучшить прогностическую способность модели, если она страдает от переобучения.



Жадные алгоритмы поиска

Жадные алгоритмы делают локально оптимальный выбор на каждом этапе задачи комбинаторного поиска и обычно дают субоптимальное решение задачи, в отличие от алгоритмов полного перебора, которые оценивают все возможные комбинации и гарантированно находят оптимальное решение. Однако на практике полный перебор часто неосуществим с вычислительной точки зрения, тогда как жадные алгоритмы позволяют найти менее сложное и более эффективное с вычислительной точки зрения решение.

Идея алгоритма SBS довольно проста: он последовательно удаляет признаки из полного подмножества признаков до тех пор, пока новое подпространство признаков не будет содержать желаемое количество признаков. Чтобы определить, какой признак должен быть удален на текущем этапе, необходимо задать критериальную функцию J , которую мы должны минимизировать.

Критерием, вычисляемым по значениям целевой функции, может быть просто разница в производительности классификатора до и после удаления того или иного признака. Тогда признак, подлежащий удалению на каждом этапе, может быть просто определен как признак, максимизирующий этот критерий, — проще говоря, на каждом этапе мы устранием признака, который вызывает наименьшую потерю производительности после удаления. Основываясь на этом определении SBS, мы можем составить алгоритм, состоящий из четырех простых шагов:

1. Инициализируем алгоритм с $k = d$, где d — размерность полного пространства признаков X_d .
2. Находим признак x^- , который максимизирует критериальную функцию: $x^- = \operatorname{argmax} J(X_k - x)$, где $x \in X_k$.
3. Удаляем признак x^- из набора признаков: $X_{k-1} = X_k - x^-$; $k = k - 1$.
4. Завершаем выполнение, если k равно количеству желаемых признаков, в противном случае переходим к шагу 2.



Обзор последовательных алгоритмов отбора признаков

Подробный обзор нескольких алгоритмов последовательных признаков представлен в работе «Comparative Study of Techniques for Large-Scale Feature Selection» by F. Ferri, P. Pudil, M. Hatef, J. Kittler, p. 403–413, 1994.

Чтобы попрактиковаться в программировании и приобрести навык реализации собственных алгоритмов, давайте разработаем код алгоритма отбора на Python с нуля:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
```

```
class SBS:  
    def __init__(self, estimator, k_features,  
                 scoring=accuracy_score,  
                 test_size=0.25, random_state=1):  
        self.scoring = scoring  
        self.estimator = clone(estimator)  
        self.k_features = k_features  
        self.test_size = test_size  
        self.random_state = random_state  
    def fit(self, X, y):  
        X_train, X_test, y_train, y_test = \  
            train_test_split(X, y, test_size=self.test_size,  
                             random_state=self.random_state)  
  
        dim = X_train.shape[1]  
        self.indices_ = tuple(range(dim))  
        self.subsets_ = [self.indices_]  
        score = self._calc_score(X_train, y_train,  
                                 X_test, y_test, self.indices_)  
        self.scores_ = [score]  
        while dim > self.k_features:  
            scores = []  
            subsets = []  
  
            for p in combinations(self.indices_, r=dim - 1):  
                score = self._calc_score(X_train, y_train,  
                                         scores.append(score)  
                                         subsets.append(p)  
  
            best = np.argmax(scores)  
            self.indices_ = subsets[best]  
            self.subsets_.append(self.indices_)  
            dim -= 1  
  
            self.scores_.append(scores[best])  
        self.k_score_ = self.scores_[-1]  
  
    return self  
  
    def transform(self, X):  
        return X[:, self.indices_]  
  
    def _calc_score(self, X_train, y_train, X_test, y_test, indices):  
        self.estimator.fit(X_train[:, indices], y_train)  
        y_pred = self.estimator.predict(X_test[:, indices])  
        score = self.scoring(y_test, y_pred)  
        return score
```

В этом коде мы определили параметр `k_features`, чтобы указать желаемое количество признаков, которые мы хотим вернуть. По умолчанию мы используем `accuracy_score` от `scikit-learn` для оценки производительности модели (оцениватель для классификации) на подмножествах признаков.

Внутри цикла `while` метода `fit` подмножества признаков, созданные функцией `itertools.combinations`, подвергаются оцениванию и сокращению до тех пор, пока подмножество признаков не получит желаемую размерность. На каждой итерации оценка точности лучшего подмножества собирается в список `self.scores_` на основе внутреннего тестового набора данных `x_test`. Мы воспользуемся этими баллами позже, чтобы оценить результаты. Индексы столбцов окончательного подмножества признаков назначаются списку `self.indices_`, который мы можем использовать с помощью метода `transform`, чтобы вернуть новый массив данных с выбранными столбцами признаков. Обратите внимание, что вместо явного вычисления критерия в методе `fit`, мы просто удалили признак, который не содержится в наиболее эффективном подмножестве признаков.

Теперь давайте проверим нашу реализацию SBS в действии, используя классификатор `knn` из `scikit-learn`:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Хотя наша реализация SBS уже разделяет набор данных на тестовый и обучающий наборы внутри функции `fit`, мы по-прежнему передаем алгоритму обучающий набор данных `X_train`. Затем метод `fit` SBS создаст из него новые подмножества для проверки и обучения, поэтому наш тестовый набор также называется *проверочным (validation)* набором данных. Этот подход необходим, чтобы наш *исходный* тестовый набор ни в коем случае не стал частью обучающих данных.

Итак, наш алгоритм SBS собирает оценки лучшего подмножества признаков на каждом этапе. Теперь давайте перейдем к более интересной части нашей реализации и построим график точности классификатора kNN, который был рассчитан на наборе проверочных данных:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.02])
>>> plt.ylabel('Точность')
>>> plt.xlabel('Количество признаков')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

Как можно видеть на рис. 4.9, точность классификатора kNN улучшилась по сравнению с проверочным набором данных, поскольку мы уменьшили количество признаков, что, вероятно, связано с уменьшением проклятия размерности, которое мы обсуждали в контексте алгоритма kNN в главе 3. Кроме того, на этом графике мы видим, что классификатор достиг 100-процентной точности для $k = \{3, 7, 8, 9, 10, 11, 12\}$.

Чтобы удовлетворить собственное любопытство, давайте посмотрим, как выглядит наименьшее подмножество признаков ($k = 3$), которое дало такую хорошую точность на проверочном наборе данных:

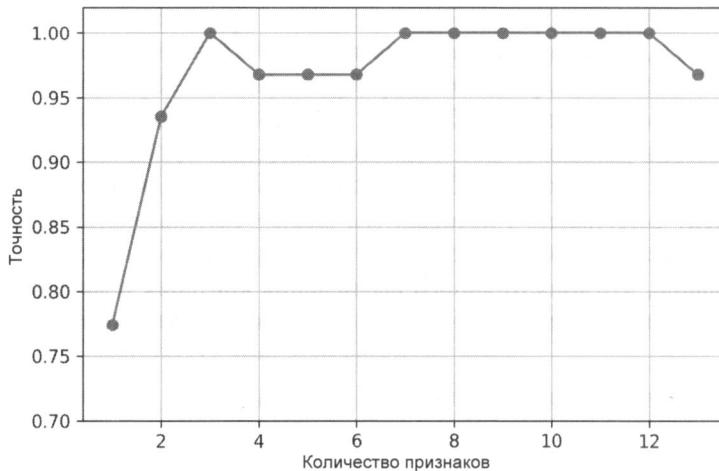


Рис. 4.9. Влияние количества признаков на точность модели

```
>>> k3 = list(sbs.subsets_[10])
>>> print(df_wine.columns[1:][k3])
Index(['Alcohol', 'Malic acid', 'OD280/OD315 of diluted wines'],
      dtype='object')
```

Используя этот код, мы получили индексы столбцов подмножества из трех признаков с 11-й позиции в атрибуте `sbs.subsets_` и вернули соответствующие имена признаков из индекса столбца `DataFrame pandas`.

Далее давайте оценим производительность классификатора kNN на исходном тестовом наборе данных:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Точность при обучении:', knn.score(X_train_std, y_train))
Точность при обучении: 0.967741935484
>>> print('Точность при тестировании:', knn.score(X_test_std, y_test))
Точность при тестировании: 0.962962962963
```

В приведенном примере кода мы использовали полный набор признаков и получили точность примерно 97% на обучающем наборе данных и примерно 96% на тестовом наборе, что свидетельствует о хорошей обобщающей способности нашей модели. Теперь давайте используем выбранное подмножество из трех признаков и посмотрим, насколько хорошо сработает kNN в этом случае:

```
>>> knn.fit(X_train_std[:, k3], y_train)
>>> print('Точность при обучении:',
...       knn.score(X_train_std[:, k3], y_train))
Точность при обучении: 0.951612903226
>>> print('Точность при тестировании:',
...       knn.score(X_test_std[:, k3], y_test))
Точность при тестировании: 0.925925925926
```

Мы видим, что когда осталось менее четверти исходных признаков набора данных Wine, точность прогнозирования на тестовом наборе данных снизилась лишь незначи-

тельно. Это может говорить о том, что оставшиеся три признака содержат почти столь же значимую классифицирующую информацию, что и полный набор данных. Однако мы должны иметь в виду, что набор данных Wine является небольшим набором данных и очень восприимчив к случайности, т. е. к тому, как мы разбиваем набор данных на обучающую и тестовую подгруппы и как мы далее разбиваем набор обучающих данных на подмножества для обучения и проверки.

Хотя мы и не повысили точность модели kNN за счет уменьшения количества признаков, но уменьшили размер набора данных, что может быть полезно в реальных сценариях применения, когда мы сталкиваемся с дорогостоящим этапом сбора данных. Кроме того, существенно сократив количество признаков, мы получим более простые модели, которые легче интерпретировать.



Алгоритмы отбора признаков в scikit-learn

Вы можете найти реализации нескольких различных вариантов последовательного отбора признаков, связанных с простым SBS, который мы реализовали ранее в пакете Python mlxtend по адресу: http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/. Хотя наша реализация mlxtend имеет много дополнительных возможностей, в сотрудничестве с командой scikit-learn мы выпустили упрощенную, удобную для пользователя версию, которая вошла в недавний выпуск v0.24. Использование и поведение этого метода очень похожи на код SBS, который вы видели в этой главе. Если вы хотите узнать больше, обратитесь к документации по адресу: https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html.

Есть много других алгоритмов выбора признаков, доступных через scikit-learn. К ним относятся рекурсивное обратное исключение на основе весов признаков, древовидные методы выбора признаков по важности и одномерные статистические тесты. Всестороннее обсуждение различных методов выбора признаков выходит за рамки этой книги, но хороший обзор с иллюстративными примерами можно найти по адресу: http://scikit-learn.org/stable/modules/feature_selection.html.

4.6. Оценка важности признаков с помощью случайных лесов

В предыдущих разделах этой главы было рассказано, как использовать регуляризацию L1 для обнуления нерелевантных признаков с помощью логистической регрессии, а также как использовать алгоритм SBS для выбора признаков и применять его совместно с алгоритмом kNN. Еще один полезный подход к отбору релевантных признаков из набора данных — обращение к варианту «случайный лес» метода ансамблирования, представленному в главе 3. Используя метод случайного леса, мы можем измерить значимость признаков как усредненное уменьшение примеси, вычисленное по всем деревьям решений в лесу, не делая никаких предположений о том, являются ли наши данные линейно разделимыми или нет. Удобно, что реализация случайного леса в scikit-learn уже собирает для нас значения важности функций, чтобы мы могли получить к ним доступ через атрибут `feature_importances_` после обучения классификатора `RandomForestClassifier`. Выполнив приведенный далее код, мы обучим лес из 500 деревьев

на наборе данных Wine и ранжируем 13 признаков по их соответствующим показателям значимости. Напомним, что модели на основе деревьев не нуждаются в нормализованных или стандартизованных признаках:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=500,
...                                 random_state=1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-s %f" % (f + 1, 30,
...                           feat_labels[indices[f]],
...                           importances[indices[f]]))
>>> plt.title('Значимость признака')
>>> plt.bar(range(X_train.shape[1]),
...           importances[indices],
...           align='center')
>>> plt.xticks(range(X_train.shape[1]),
...             feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()

1) Proline          0.185453
2) Flavanoids      0.174751
3) Color intensity 0.143920
4) OD280/OD315 of diluted wines 0.136162
5) Alcohol          0.118529
6) Hue              0.058739
7) Total phenols   0.050872
8) Magnesium        0.031357
9) Malic acid       0.025648
10) Proanthocyanins 0.025570
11) Alcalinity of ash 0.022366
12) Nonflavanoid phenols 0.013354
13) Ash              0.013279
```

В результате выполнения кода будет построен график, который ранжирует различные признаки в наборе данных Wine по их относительной значимости, — обратите внимание, что значимость признаков нормализована, так что их сумма равна 1.0 (рис. 4.10).

Мы можем заключить, что уровни пролина (Proline) и флавоноидов (Flavanoids), интенсивность цвета (Color intensity), фактор OD280/OD315 разбавленных вин (OD280/OD315 of diluted wines) и концентрация алкоголя (Alcohol) в вине являются наиболее отличительными признаками в наборе данных, основанном на среднем снижении примесей в 500 деревьях решений. Интересно, что два признака с наивысшим рейтингом на графике также входят в подмножество из трех признаков, найденных алгоритмом SBS, который мы реализовали в предыдущем разделе (концентрация алкоголя и фактор OD280/OD315 разбавленных вин).

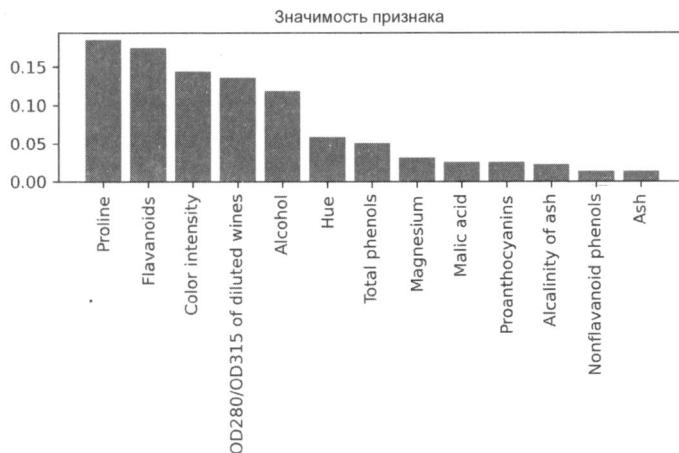


Рис. 4.10. Значимость признаков для модели случайного леса, обученной на наборе данных Wine

Однако в том, что касается интерпретируемости, у метода случайного леса есть важный нюанс, о котором стоит упомянуть. Если два или более признака сильно коррелированы, один признак может иметь очень высокий рейтинг, в то время как информация о других признаках может быть не полностью отражена. С другой стороны, нам не нужно беспокоиться об этой проблеме, если нас интересует просто прогностическая эффективность модели, а не интерпретация значений важности признаков.

В заключение этого раздела о значимости признаков и случайных лесах стоит упомянуть, что scikit-learn также реализует объект `SelectFromModel`, который выбирает признаки на основе заданного пользователем порога после обучения модели, что полезно, если мы хотим использовать `RandomForestClassifier` как селектор признаков и промежуточный шаг в объекте `Pipeline` scikit-learn, который позволяет нам соединять различные этапы предварительной обработки в оценивателе, как вы увидите позже в главе 6. Например, можно установить пороговое значение 0.1, чтобы уменьшить набор данных до пяти наиболее важных признаков, используя следующий код:

```
>>> from sklearn.feature_selection import SelectFromModel
>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>>> X_selected = sfm.transform(X_train)
>>> print('Количество признаков, соответствующих пороговому',
...      'критерию:', X_selected.shape[1])
Количество признаков, соответствующих пороговому критерию: 5
>>> for f in range(X_selected.shape[1]):
...     print("%2d" %-*s %f" % (f + 1, 30,
...                           feat_labels[indices[f]],
...                           importances[indices[f]]))
1) Proline          0.185453
2) Flavanoids       0.174751
3) Color intensity 0.143920
4) OD280/OD315 of diluted wines 0.136162
5) Alcohol          0.118529
```

4.7. Заключение

Мы начали эту главу с рассмотрения полезных приемов, позволяющих убедиться в правильности обработки отсутствующих данных. Прежде чем отправить данные в алгоритм машинного обучения, необходимо также убедиться, что мы правильно кодируем категориальные переменные, и в этой главе вы научились сопоставлять порядковые и номинальные признаки с целочисленными представлениями.

Кроме того, мы кратко обсудили регуляризацию L1, которая помогает избежать переобучения за счет уменьшения сложности модели. В качестве альтернативного подхода к удалению ненужных признаков мы использовали алгоритм последовательного отбора признаков, чтобы выбрать наиболее значимые признаки из набора данных.

В следующей главе вы узнаете о еще одном полезном подходе к уменьшению размерности: извлечении признаков. Этот прием позволяет нам сжимать признаки в подпространство более низкого измерения, а не удалять их полностью, как при отборе признаков.

5

Сжатие данных путем уменьшения размерности

В главе 4 вы узнали о разнообразных подходах к уменьшению размерности набора данных с использованием различных методов выбора признаков. Альтернативным подходом к выбору признаков для уменьшения размерности является *извлечение признаков* (feature extraction). В этой главе говорится о двух фундаментальных методах, которые помогут вам сделать информационное наполнение набора данных более обобщенным, преобразовав его в новое подпространство объектов более низкой размерности, чем исходное. Сжатие данных занимает важное место в машинном обучении — оно помогает нам хранить и анализировать непрерывно растущие объемы данных, производимых и собираемых в эпоху передовых технологий.

В этой главе будут рассмотрены следующие темы:

- ◆ метод главных компонент для сжатия данных без учителя;
- ◆ линейный дискриминантный анализ как метод уменьшения размерности с учителем для достижения максимальной разделимости классов;
- ◆ краткий обзор методов нелинейного уменьшения размерности и t -распределенного стохастического встраивания соседей для визуализации данных.

5.1. Уменьшение размерности без учителя с помощью метода главных компонент

Подобно отбору признаков, мы можем использовать различные методы извлечения признаков, чтобы уменьшить количество признаков в наборе данных. Разница между отбором и извлечением признаков заключается в том, что при использовании алгоритма отбора — такого как *последовательный пошаговый отбор* (sequential backward selection), мы сохраняем исходные признаки, а извлечение признаков применяется для преобразования или проецирования данных в новое пространство признаков.

В контексте уменьшения размерности извлечение признаков можно рассматривать как способ сжатия данных с целью сохранения большей части релевантной информации. На практике извлечение признаков используется не только для увеличения объема свободной памяти или вычислительной эффективности алгоритма обучения, но также может повысить эффективность прогнозирования за счет уменьшения проклятия размерности, особенно при работе с нерегуляризованными моделями.

5.1.1. Основные этапы метода главных компонент

В этом разделе мы обсудим *метод главных компонент* (Principal Component Analysis, PCA) — метод линейного преобразования без обучения с учителем, который широко используется в различных областях и в первую очередь для извлечения признаков и уменьшения размерности. К другим популярным приложениям PCA можно отнести исследовательский анализ данных и шумоподавление сигналов при торговле на фондовом рынке, а также анализ данных генома и уровней экспрессии генов в области биоинформатики.

PCA помогает нам идентифицировать закономерности в данных на основе корреляции между функциями. Попросту говоря, PCA стремится найти направления максимальной дисперсии в многомерных данных и проецирует данные на новое подпространство с равными или меньшими измерениями, чем исходное. Ортогональные оси (главные компоненты) нового подпространства можно интерпретировать как направления максимальной дисперсии с учетом того ограничения, что оси новых признаков ортогональны друг другу, как показано на рис. 5.1.

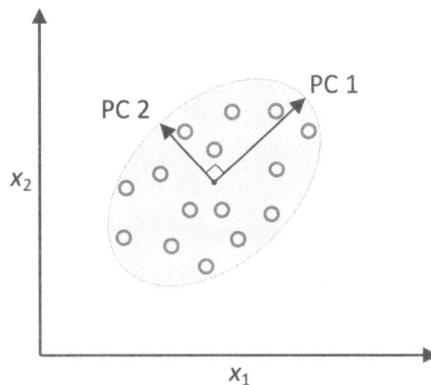


Рис. 5.1. Использование PCA для поиска направлений максимальной дисперсии в наборе данных

Здесь x_1 и x_2 — оси исходных признаков, а $PC\ 1$ и $PC\ 2$ — главные компоненты. Если мы используем PCA для уменьшения размерности, мы строим $d \times k$ -мерную матрицу преобразования W , что позволяет нам сопоставить вектор признаков обучающего экземпляра x с новым k -мерным подпространством признаков, которое имеет меньше измерений, чем исходное d -мерное пространство признаков. Рассмотрим пример преобразования. Предположим, у нас есть вектор признаков x :

$$x = [x_1, x_2, \dots, x_d], \quad x \in \mathbb{R}^d,$$

который затем преобразуется при помощи матрицы преобразования $W \in \mathbb{R}^{d \times k}$:

$$xW = z,$$

и мы получаем следующий выходной вектор:

$$z = [z_1, z_2, \dots, z_k], \quad z \in \mathbb{R}^k.$$

В результате преобразования исходных d -мерных данных в это новое k -мерное подпространство (обычно $k \ll d$) первая главная компонента будет иметь максимально воз-

можную дисперсию. Все последующие главные компоненты будут иметь наибольшую дисперсию при соблюдении ограничения, состоящего в том, что эти компоненты не коррелированы (ортогональны) с другими главными компонентами, — даже если входные признаки коррелированы, результирующие главные компоненты будут взаимно ортогональны (некоррелированы). Обратите внимание, что направления РСА очень чувствительны к масштабированию данных, и нам необходимо стандартизировать признаки до РСА, если они были измерены в разных масштабах, и мы хотим присвоить всем признакам одинаковую важность.

Прежде чем более детально рассмотреть применение алгоритма РСА для уменьшения размерности, перечислим его основные шаги:

1. Стандартизуем d -мерный набор данных.
2. Строим ковариационную матрицу.
3. Раскладываем ковариационную матрицу на собственные векторы и собственные значения.
4. Сортируем собственные значения в порядке убывания, чтобы ранжировать соответствующие собственные векторы.
5. Выбираем k собственных векторов, которые соответствуют k наибольшим собственным значениям, где k — размерность нового подпространства признаков ($k \leq d$).
6. Строим матрицу проекции W из «верхних» k собственных векторов.
7. Преобразовываем d -мерный входной набор данных X , используя матрицу проекции W , чтобы получить новое k -мерное подпространство признаков.

В следующих разделах в качестве учебного упражнения мы шаг за шагом выполним алгоритм РСА, используя собственный код на Python. Затем вы узнаете более удобный способ выполнять РСА с помощью scikit-learn.



Собственная декомпозиция: разложение матрицы на собственные векторы и собственные значения

В основе процедуры РСА, описанной в этом разделе, лежит *собственная декомпозиция* (eigendecomposition) — разложение квадратной матрицы на так называемые *собственные значения* (eigenvalues) и *собственные векторы* (eigenvectors).

Ковариационная (дисперсная) матрица представляет собой частный случай квадратной матрицы — это симметричная матрица, а это означает, что исходная матрица равна своей транспонированной матрице: $A = A^T$.

Когда мы выполняем разложение такой симметричной матрицы, собственные значения являются действительными (а не комплексными) числами, а собственные векторы ортогональны (перпендикулярны) друг другу. Кроме того, собственные значения и собственные векторы идут парами. При разложении ковариационной матрицы на собственные векторы и собственные значения собственные векторы, связанные с наибольшим собственным значением, соответствуют направлению максимальной дисперсии в наборе данных. Здесь это «направление» представляет собой линейное преобразование столбцов признаков набора данных.

Хотя более подробное обсуждение собственных значений и собственных векторов выходит за рамки этой книги, относительно подробное описание с указателями на дополнительные ресурсы можно найти в Википедии по адресу:

https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors.

5.1.2. Пошаговый процесс извлечения основных компонент

Далее мы рассмотрим первые четыре шага алгоритма РСА:

1. Стандартизация данных.
2. Построение ковариационной матрицы.
3. Получение собственных значений и собственных векторов ковариационной матрицы.
4. Сортировка собственных значений по убыванию для ранжирования собственных векторов.

Начнем с загрузки набора данных Wine, с которым мы работали в *главе 4*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv(
...     'https://archive.ics.uci.edu/ml/'
...     'machine-learning-databases/wine/wine.data',
...     header=None
... )
```



Получение набора данных Wine

Вы можете найти копию этого набора данных (и всех других наборов данных, используемых в книге) в ее файловом архиве, когда работаете в автономном режиме, или если сервер UCI по адресу: <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data>, временно недоступен. Например, чтобы загрузить набор данных Wine из локального каталога, нужно заменить строки:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

на следующие:

```
df = pd.read_csv(
    'your/local/path/to/wine.data', header=None
)
```

Затем мы обработаем данные Wine, разделив их на обучающий и тестовый наборы данных, используя 70 и 30 % исходного набора данных соответственно, и стандартизуем их для единичной дисперсии:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                      stratify=y,
...                      random_state=0)
>>> # стандартизация признаков
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
```

```
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Выполнив обязательную предварительную обработку при помощи этого кода, перейдем ко второму шагу — построению ковариационной матрицы. Симметричная ковариационная матрица размером $d \times d$, где d — количество измерений в наборе данных, хранит попарные ковариации между различными признаками. Например, ковариация между двумя признаками x_j и x_k на уровне популяции может быть рассчитана с помощью следующего уравнения:

$$\sigma_{jk} = \frac{1}{n-1} \sum_{i=1}^n (x_j^{(i)} - \mu_j)(x_k^{(i)} - \mu_k).$$

Здесь μ_j и μ_k — выборочные средние признаков j и k соответственно. Напомним, что выборочные средние равны нулю, если мы стандартизировали набор данных. Положительная ковариация между двумя признаками указывает на то, что признаки увеличиваются или уменьшаются вместе, тогда как отрицательная ковариация — на то, что признаки изменяются в противоположных направлениях. Например, ковариационная матрица трех признаков может быть записана следующим образом (учтите, что здесь Σ — это греческая прописная буква «сигма», которую не следует путать с символом математической операции суммирования):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}.$$

Собственные векторы ковариационной матрицы представляют главные компоненты (направления максимальной дисперсии), тогда как соответствующие собственные значения будут определять их величину. В случае набора данных Wine мы получили бы 13 собственных векторов и собственных значений из ковариационной матрицы размером 13×13 .

Далее, на третьем шаге получим собственные пары ковариационной матрицы. Если вы изучали линейную алгебру, то, возможно, помните, что собственный вектор v удовлетворяет следующему условию:

$$\Sigma v = \lambda v.$$

Здесь λ — скалярное собственное значение. Поскольку ручное вычисление собственных векторов и собственных значений является достаточно утомительной и сложной задачей, для получения собственных пар ковариационной матрицы Wine мы воспользуемся функцией `linalg.eig` из NumPy:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n', eigen_vals)
Eigenvalues
[ 4.84274532  2.41602459  1.54845825  0.96120438  0.84166161
 0.6620634   0.51828472  0.34650377  0.3131368   0.10754642
 0.21357215  0.15362835  0.1808613 ]
```

Применив функцию `numpy.cov`, мы вычислили ковариационную матрицу стандартизованного набора обучающих данных. А используя функцию `linalg.eig`, мы выполнили

собственное разложение, в результате чего был получен вектор (`eigen_vals`), состоящий из 13 собственных значений и соответствующих собственных векторов, хранящихся в виде столбцов в матрице 13×13 (`eigen_vecs`).



Собственная декомпозиция в NumPy

Функция `numpy.linalg.eig` была разработана для работы как с симметричными, так и с несимметричными квадратными матрицами. Однако вы можете обнаружить, что в некоторых случаях она возвращает комплексные собственные значения.

Родственная ей функция `numpy.linalg.eigh` была реализована для декомпозиции герметовых матриц, что является численно более устойчивым подходом к работе с симметричными матрицами, такими как ковариационная матрица, — `numpy.linalg.eigh` всегда возвращает реальные собственные значения.

5.1.3. Общая и объясненная дисперсия

Поскольку мы стремимся уменьшить размерность нашего набора данных, сжав его в новое подпространство признаков, то выбираем только такое подмножество собственных векторов (главных компонент), которое содержит большую часть информации (дисперсию). Собственные значения определяют величину собственных векторов, поэтому мы должны отсортировать собственные значения по убыванию величины, — нас интересуют лучшие k собственных векторов на основе величины их соответствующих собственных значений. Но прежде чем мы отберем эти k наиболее информативных собственных векторов, давайте построим для собственных значений графики доли объясненной дисперсии (*variance explained ratio*). Доля объясненной дисперсии собственного значения λ_j представляет собой просто частное собственного значения λ_j и полной суммы собственных значений:

$$\text{Доля объясненной дисперсии} = \frac{\lambda_j}{\sum_{j=1}^d \lambda_j}.$$

Используя функцию `cumsum` библиотеки NumPy, мы можем вычислить накопительную сумму объясненных дисперсий, которую затем построим с помощью функции `step` из Matplotlib:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, align='center',
...           label='Отдельные объясненные дисперсии')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...           label='Совокупная объясненная дисперсия')
>>> plt.ylabel('Доля объясненной дисперсии')
>>> plt.xlabel('Индекс главной компоненты')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Полученный график (рис. 5.2) показывает, что только на первую главную компоненту приходится примерно 40 % дисперсии. Кроме того, мы видим, что первые две основные компоненты в совокупности объясняют почти 60 % дисперсии в наборе данных.

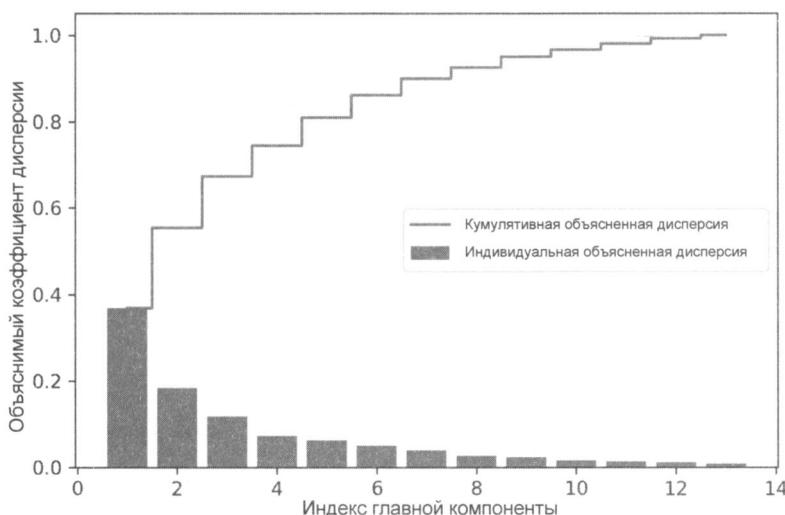


Рис. 5.2. Доля общей дисперсии, определяемой основными компонентами

Хотя график объясненной дисперсии напоминает нам о значениях важности признаков, которые мы вычислили в главе 4 с помощью случайных лесов, нельзя забывать, что PCA — это метод без учителя, а это означает, что информация о метках классов игнорируется. В то время как случайный лес использует информацию о членстве в классе для вычисления примесей узлов, дисперсия просто измеряет разброс значений по оси признаков.

5.1.4. Преобразование признаков

Выполнив разложение ковариационной матрицы на собственные пары, перейдем к последним трем шагам алгоритма, чтобы преобразовать набор данных Wine в новые оси главных компонент. В этом разделе нам осталось выполнить следующие шаги:

1. Выбрать k собственных векторов, которые соответствуют k наибольшим собственным значениям, где k — размерность нового подпространства признаков ($k \leq d$).
2. Построить матрицу проекции W из «верхних» k собственных векторов.
3. Преобразовать d -мерный входной набор данных X , используя матрицу проекции W , чтобы получить новое k -мерное подпространство признаков.

Или, выражаясь менее техническим языком, мы отсортируем собственные пары по убыванию собственных значений, построим матрицу проекции из выбранных собственных векторов и используем ее для преобразования данных в подпространство меньшей размерности.

Начнем с сортировки собственных пар по убыванию собственных значений:

```
>>> # Создаем список пар (eigenvalue, eigenvector)
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...                 for i in range(len(eigen_vals))]
>>> # Сортируем пары (eigenvalue, eigenvector) по убыванию
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Затем соберем два собственных вектора, которые соответствуют двум самым большим собственным значениям, обеспечивающим около 60 % дисперсии в этом наборе данных. Нужно заметить, что для наглядности здесь мы берем только два собственных вектора, поскольку позже в этом разделе мы собираемся изобразить данные с помощью двумерной диаграммы рассеяния. На практике количество основных компонент должно определяться компромиссом между вычислительной эффективностью и производительностью классификатора:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                  eigen_pairs[1][1][:, np.newaxis]))
>>> print('Матрица W:\n', w)
```

Матрица W:

```
[[-0.13724218  0.50303478]
 [ 0.24724326  0.16487119]
 [-0.02545159  0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582  0.28974518]
 [-0.39376952  0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896  0.09048885]
 [-0.30668347  0.00835233]
 [ 0.07554066  0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651  0.38022942]]
```

Выполнив приведенный код, мы создали матрицу проекции W размером 13×2 из двух верхних собственных векторов.



Зеркальные проекции

В зависимости от того, какие версии NumPy и LAPACK вы используете, вы можете получить матрицу W с перевернутыми знаками. Но это не проблема: если v — собственный вектор матрицы Σ , мы имеем

$$\Sigma v = \lambda v.$$

Здесь v — собственный вектор, и $-v$ тоже является собственным вектором, что можно показать следующим образом: используя основные правила алгебры, мы можем умножить обе части равенства на скаляр α :

$$\alpha \Sigma v = \alpha \lambda v.$$

Поскольку матричное умножение является ассоциативным для скалярного умножения, то это равенство мы можем записать так:

$$\Sigma(\alpha v) = \lambda(\alpha v).$$

Теперь мы видим, что αv является собственным вектором с одним и тем же собственным значением λ как для $\alpha = 1$, так и для $\alpha = -1$. Следовательно, и v , и $-v$ являются собственными векторами.

Используя матрицу проекций, преобразуем экземпляр x (представленный в виде 13-мерного вектора-строки) в подпространство PCA (первая и вторая главные компоненты), получив x' — теперь уже двумерный вектор, состоящий из двух новых признаков:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.38299011,  0.45458499])
```

Аналогичным образом мы можем преобразовать весь обучающий набор данных размером 124×13 в две основные компоненты, вычислив скалярное произведение матриц:

$$X' = XW.$$

Наконец, визуализируем преобразованный обучающий набор данных Wine, который теперь хранится в виде двумерной матрицы 124×2 , построив двумерную диаграмму рассеяния:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['o', 's', '^']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=f'Class {l}', marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

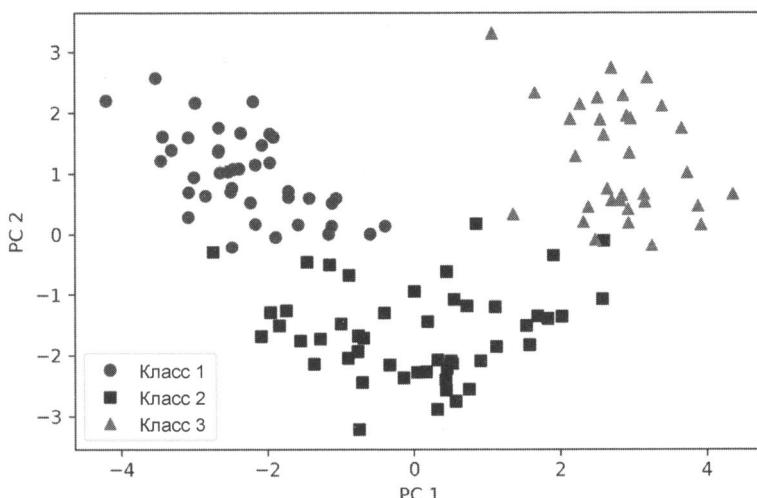


Рис. 5.3. Записи из набора данных Wine, спроектированные на двумерное пространство признаков при помощи PCA

Как можно видеть на рис. 5.3, данные сильнее разбросаны по направлению первой главной компоненты (ось x), чем по направлению второй главной компоненты (ось y), что согласуется с графиком доли объясненной дисперсии, который мы построили в предыдущем разделе. Однако есть шанс надеяться, что линейный классификатор, вероятно, сможет хорошо разделить классы.

Хотя для наглядности на рис. 5.3 мы отобразили информацию о метке класса, нужно помнить, что PCA — это метод без учителя, который не использует никакой информации о метке класса.

5.1.5. Анализ основных компонент в scikit-learn

В предыдущем разделе мы подробно исследовали внутренний механизм алгоритма PCA, а теперь вы узнаете, как использовать класс PCA, реализованный в scikit-learn.

Класс PCA — это еще один класс преобразователя scikit-learn, с помощью которого мы сначала обучаем модель, используя обучающие данные, прежде чем преобразовывать как обучающие данные, так и набор тестовых данных, задействуя одни и те же параметры модели. Теперь применим класс PCA из scikit-learn к обучающему набору данных Wine, классифицируем преобразованные примеры с помощью логистической регрессии и визуализируем области принятия решений с помощью функции plot_decision_regions, которую мы определили в главе 2:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # настройка маркеров и цветов диаграммы
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # рисование разделяющей поверхности
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # рисование экземпляров класса
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Класс {cl}',
                    edgecolor='black')
```

Для вашего удобства вы можете поместить приведенный код `plot_decision_regions` в отдельный файл кода в вашем текущем рабочем каталоге — например, в файл `plot_decision_regions_script.py`, и импортировать его в текущий сеанс Python:

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> # инициализация преобразователя PCA
>>> # и оценивателя логистической регрессии:
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression(multi_class='ovr',
...                         random_state=1,
...                         solver='lbfgs')
>>> # понижение размерности:
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> # обучение модели логистической регрессии на новом наборе:
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

Выполнив этот код, вы получите области принятия решений для обучающих данных, сведенные к осям двух главных компонент (рис. 5.4).

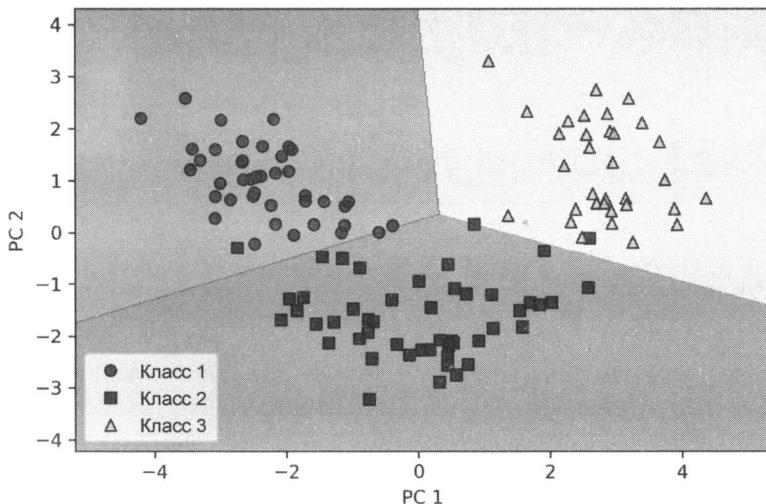


Рис. 5.4. Обучающие данные и области принятия решений логистической регрессии после применения класса `PCA` `scikit-learn` для уменьшения размерности

Если вы сравните прогнозы, сделанные PCA с помощью `scikit-learn` со своей реализацией PCA, то увидите, что полученные графики являются зеркальными по отношению друг к другу. Это не связано с ошибкой в какой-то из реализаций алгоритма — причина

зеркального отображения в том, что в зависимости от решателя собственные векторы могут иметь как отрицательные, так и положительные знаки.

Это не имеет особого значения, но при желании вы можете получить зеркальное отображение, умножив данные на -1 (кстати, собственные векторы обычно масштабируются до единичной длины). Для полноты картины давайте нанесем области принятия решений логистической регрессии на преобразованный набор тестовых данных, чтобы увидеть, может ли модель хорошо разделить классы:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

Выполнив этот код, мы убеждаемся, что логистическая регрессия достаточно хорошо работает в этом небольшом двумерном подпространстве признаков и неправильно классифицирует только несколько примеров из тестового набора данных (рис. 5.5).

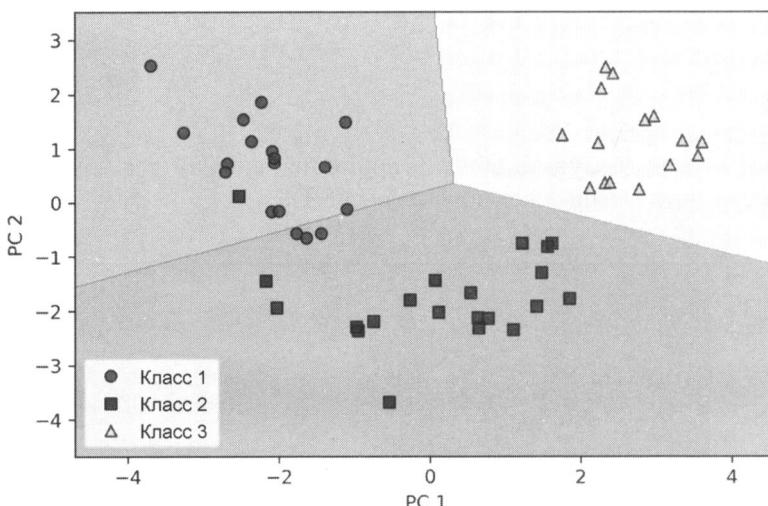


Рис. 5.5. Точки данных тестового набора с областями принятия решений логистической регрессии в пространстве признаков, полученном с помощью PCA

Если нас интересуют доли объясненной дисперсии различных главных компонент, мы можем просто инициализировать класс `PCA` с параметром `n_components = None`, чтобы все главные компоненты были сохранены, а затем получить доступ к доле объясненной дисперсии через атрибут `explained_variance_ratio_`:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.36951469,  0.18434927,  0.11815159,  0.07334252,
       0.06422108,  0.05051724,  0.03954654,  0.02643918,
       0.02389319,  0.01629614,  0.01380021,  0.01172226,
       0.00820609])
```

Обратите внимание, что при инициализации класса PCA мы установили параметр n_components=None, чтобы класс возвращал все главные компоненты в отсортированном порядке — вместо выполнения уменьшения размерности.

5.1.6. Оценка вклада признаков

В этом разделе мы кратко рассмотрим, как можно оценить вклад исходных признаков в главные компоненты. Как вы уже знаете, с помощью PCA мы создаем главные компоненты, которые представляют собой линейные комбинации признаков. Иногда бывает полезно узнать, какой вклад вносит каждый исходный признак в определенную главную компоненту. Этот вклад часто называют *нагрузкой* (loading) признака на компоненту.

Факторные нагрузки могут быть вычислены путем умножения собственных векторов на квадратный корень из собственных значений. Полученные значения затем можно интерпретировать как корреляцию между исходными признаками и главной компонентой. В качестве иллюстрации построим диаграммы нагрузок для первой главной компоненты.

Сначала вычислим матрицу нагрузок 13×13 , умножая собственные векторы на квадратный корень из собственных значений:

```
>>> loadings = eigen_vecs * np.sqrt(eigen_vals)
```

Затем построим на диаграмме нагрузки для первой главной компоненты: `loadings[:, 0]` — которая является первым столбцом в этой матрице:

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), loadings[:, 0], align='center')
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 5.6 мы видим, что, например, алкоголь (Alcohol) имеет отрицательную корреляцию с первой главной компонентой (приблизительно -0.3), тогда как яблочная кислота (Malic acid) имеет положительную корреляцию (приблизительно 0.54). Значение 1 описывает идеальную положительную корреляцию, тогда как значение -1 соответствует идеальной отрицательной корреляции.

В предыдущем примере кода мы вычисляем факторные нагрузки для нашей собственной реализации PCA. Аналогичным образом мы можем получить нагрузки из обученного объекта PCA scikit-learn, где `pca.components_` представляет собственные векторы, а `pca.explained_variance_` представляет собственные значения:

```
>>> sklearn_loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
```

Чтобы сравнить нагрузки PCA scikit-learn с теми, которые мы создали ранее, построим аналогичную диаграмму (рис. 5.7):

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), sklearn_loadings[:, 0], align='center')
```

```
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

Как видите, эта диаграмма (рис. 5.7) идентична той, что изображена на рис. 5.6.

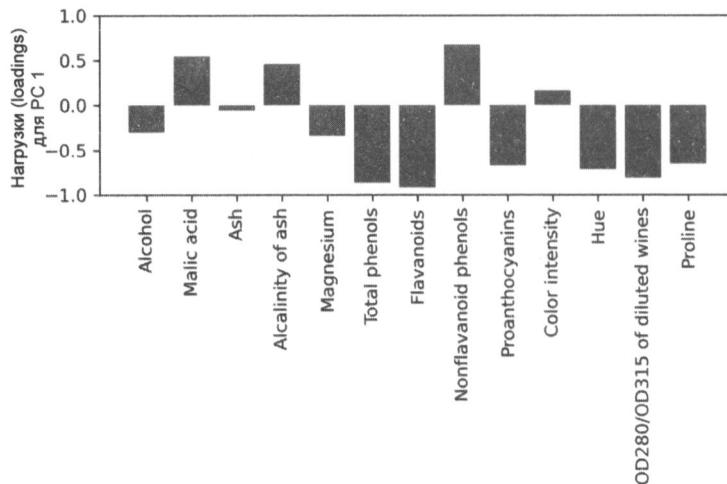


Рис. 5.6. Диаграмма корреляции признаков с первой главной компонентой

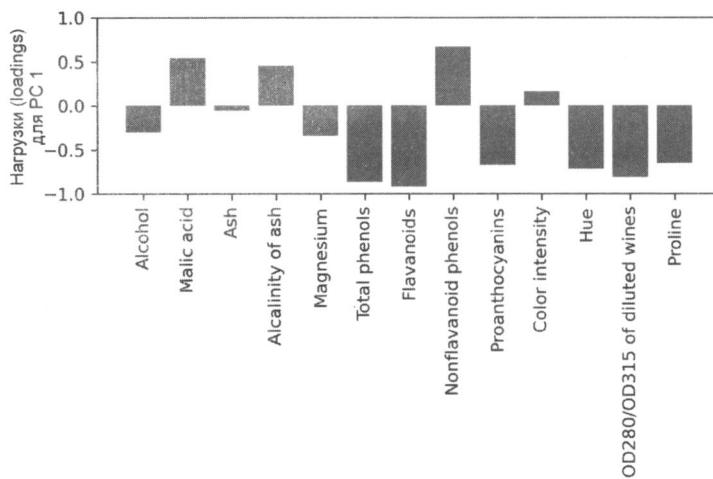


Рис. 5.7. Диаграмма корреляции признаков с первой главной компонентой с использованием scikit-learn

Итак, мы изучили алгоритм РСА как способ извлечения признаков без учителя, а в следующем разделе вы освоите линейный дискриминантный анализ (Linear Discriminant Analysis, LDA), который представляет собой метод линейного преобразования, учитывающий информацию о метках классов.

5.2. Сжатие данных с учителем с помощью линейного дискриминантного анализа

Алгоритм LDA можно использовать для извлечения признаков в качестве метода, который способствует повышению эффективности вычислений и уменьшению риска переобучения из-за проклятия размерности в нерегуляризованных моделях. В целом идея LDA очень похожа на PCA, но в то время как PCA ищет оси ортогональных компонент максимальной дисперсии в наборе данных, цель LDA состоит в том, чтобы найти подпространство признаков, которое оптимизирует разделимость классов. В следующих разделах мы более подробно обсудим сходство между LDA и PCA и шаг за шагом исследуем подход LDA.

5.2.1. Сравнение методов PCA и LDA

И PCA, и LDA представляют собой линейные преобразования, которые применяют для уменьшения количества измерений в наборе данных: первый является алгоритмом без учителя, а второй — с учителем. Мы можем сделать поспешный вывод, что LDA — это заведомо лучший метод извлечения признаков для задач классификации по сравнению с PCA. Однако А. М. Мартинес установил, что в определенных случаях предварительная обработка с помощью PCA способна давать лучшие результаты классификации в задаче распознавания изображений, например если каждый класс состоит только из небольшого числа экземпляров¹.



Линейный дискриминант Фишера

LDA иногда также называют *линейным дискриминантом Фишера*. Рональд Фишер в 1936 г. первоначально предложил использовать линейный дискриминант для задач классификации двух классов². В 1948 году К. Радхакришна Рао обобщил линейный дискриминант Фишера для многоклассовых задач в предположении о ковариациях равных классов и нормально распределенных классах, предложив метод, который мы теперь называем LDA³.

На рис. 5.8 представлен пример применения LDA к задаче классификации с двумя классами (экземпляры из класса 1 показаны кружками, а из класса 2 — крестиками).

Линейный дискриминант, показанный на оси x (*LD 1*), хорошо различает два класса с нормальным распределением. Хотя аналогичный линейный дискриминант, показанный на оси y (*LD 2*), фиксирует большую часть дисперсии в наборе данных, его нельзя признать хорошим линейным дискриминантом, поскольку он не фиксирует какую-либо дискриминационную информацию о классах.

Одно из предположений в LDA состоит в том, что данные распределены нормально. Также предполагается, что классы имеют идентичные ковариационные матрицы и что

¹ См.: «PCA Versus LDA», A. M. Martinez and A. C. Kak, IEEE Transactions on Pattern Analysis and Machine Intelligence, 23(2): 228–233, 2001.

² См.: «The Use of Multiple Measurements in Taxonomic Problems», R. A. Fisher, Annals of Eugenics, 7(2): 179–188, 1936.

³ См.: «The Utilization of Multiple Measurements in Problems of Biological Classification», C. R. Rao, Journal of the Royal Statistical Society. Series B (Methodological), 10(2): 159–203, 1948.

обучающие экземпляры статистически независимы друг от друга. Однако, даже если одно или несколько из этих предположений слегка нарушены, LDA все еще может достаточно хорошо понижать размерность⁴.

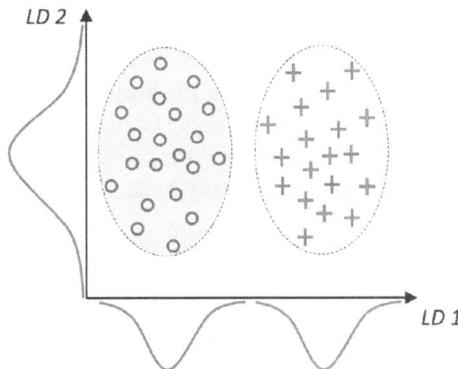


Рис. 5.8. Применение LDA к задаче классификации с двумя классами

5.2.2. Как устроен алгоритм LDA?

Прежде чем мы углубимся в реализацию кода, давайте кратко обобщим основные шаги, необходимые для выполнения LDA:

1. Стандартизируйте d -мерный набор данных (d — количество признаков).
2. Для каждого класса вычислите d -мерный средний вектор.
3. Постройте матрицу межклассового разброса S_B и матрицу внутриклассового разброса $S_W^{-1}S_B$.
4. Вычислите собственные векторы и соответствующие собственные значения матрицы.
5. Отсортируйте собственные значения в порядке убывания, чтобы ранжировать соответствующие собственные векторы.
6. Выберите k собственных векторов, соответствующих k наибольшим собственным значениям, чтобы построить $d \times k$ -мерную матрицу преобразования W , где собственные векторы являются столбцами этой матрицы.
7. Спроецируйте экземпляры набора данных на подпространство новых признаков, используя матрицу преобразования W .

Как видите, LDA очень похож на PCA в том смысле, что мы разлагаем матрицы на собственные значения и собственные векторы, которые образуют новое пространство признаков меньшей размерности. Однако, как было сказано ранее, LDA учитывает информацию о метках классов, которая представлена в виде средних векторов, вычисленных на шаге 2. В следующих разделах мы обсудим эти семь шагов более подробно, иллюстрируя их фрагментами кода.

⁴ См.: «Pattern Classification», 2nd edition, R. O. Duda, P. E. Hart, and D. G. Stork, New York, 2001.

5.2.3. Вычисление матриц разброса

Поскольку мы уже стандартизировали признаки набора данных Wine в разд. 5.1.2, то можем пропустить первый шаг и перейти к вычислению средних векторов, которые будем использовать для построения матриц внутрикласового и межклассового разброса. Каждый средний вектор \mathbf{m}_i хранит среднее значение признака μ_m по отношению к экземплярам класса i :

$$\mathbf{m}_i = \frac{1}{n_i} \sum_{x \in D_i} \mathbf{x}_m.$$

Отсюда мы получаем три средних вектора:

$$\mathbf{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1,2,3\}.$$

Эти средние векторы можно получить с помощью следующего кода, где мы вычисляем один средний вектор для каждой из трех меток:

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(X_train_std[y_train==label], axis=0))
...     print(f'MV {label}: {mean_vecs[label - 1]}\n')
MV 1: [ 0.9066 -0.3497  0.3201 -0.7189  0.5056  0.8807  0.9589 -0.5516  0.5416  0.2338  0.5897
0.6563  1.2075]
MV 2: [-0.8749 -0.2848 -0.3735  0.3157 -0.3848 -0.0433  0.0635 -0.0946  0.0703 -0.8286  0.3144
0.3608 -0.7253]
MV 3: [ 0.1992  0.866  0.1682  0.4148 -0.0451 -1.0286 -1.2876  0.8287 -0.7795  0.9649 -1.209
-1.3622 -0.4013]
```

Используя средние векторы, мы теперь можем вычислить матрицу внутрикласового разброса S_W :

$$S_W = \sum_{i=1}^c S_i.$$

Она вычисляется путем суммирования отдельных матриц разброса S_i каждого отдельного класса i :

$$S_i = \sum_{x \in D_i} (\mathbf{x} - \mathbf{m}_i)(\mathbf{x} - \mathbf{m}_i)^T.$$

```
>>> d = 13 # количество признаков
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row - mv).dot((row - mv).T)
...     S_W += class_scatter
```

```
>>> print('Матрица внутриклассового разброса: '
... f'{S_W.shape[0]}x{S_W.shape[1]}'')
Матрица внутриклассового разброса: 13x13
```

Предположение, которое мы делаем при вычислении матриц разброса, состоит в том, что метки классов в обучающем наборе данных распределены равномерно. Однако если мы выведем на печать количество меток класса, то увидим, что это предположение нарушается:

```
>>> print('Распределение меток класса:',
... np.bincount(y_train)[1:])
Распределение меток класса: [41 50 33]
```

Следовательно, мы должны масштабировать отдельные матрицы разброса S_i , прежде чем суммировать их как матрицу разброса S_W . Разделив матрицы разброса на количество экземпляров классов n_i , мы видим, что вычисление матрицы разброса фактически совпадает с вычислением ковариационной матрицы Σ_i — поскольку ковариационная матрица представляет собой нормализованную версию матрицы разброса:

$$\Sigma_i = \frac{1}{n_i} S_i = \frac{1}{n_i} \sum_{x \in D_i} (x - m_i)(x - m_i)^T.$$

Приведем код для вычисления масштабированной матрицы внутриклассового разброса:

```
>>> d = 13 # количество признаков
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Масшт. матрица внутриклассового разброса: '
... f'{S_W.shape[0]}x{S_W.shape[1]}'')
Масшт. матрица внутриклассового разброса: 13x13
```

После вычисления масштабированной матрицы внутриклассового разброса (или ковариационной матрицы) мы можем перейти к следующему шагу и вычислить матрицу межклассового разброса S_B :

$$S_B = \sum_{i=1}^c n_i(m_i - m)(m_i - m)^T.$$

Здесь m — вычисленное общее среднее значение, включающее экземпляры из всех классов c :

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> mean_overall = mean_overall.reshape(d, 1)

>>> d = 13 # количество признаков
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...     n = X_train_std[y_train == i + 1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1) # вектор столбца
...     S_B += n * (mean_vec - mean_overall).dot(
...     (mean_vec - mean_overall).T)
```

```
>>> print('Матрица межклассового разброса: '
...           f'{S_B.shape[0]}x{S_B.shape[1]})')
Матрица межклассового разброса: 13x13
```

5.2.4. Выбор линейных дискриминантов для нового подпространства признаков

Остальные шаги LDA аналогичны шагам РСА. Однако вместо того, чтобы выполнять собственное разложение ковариационной матрицы, мы решаем обобщенную задачу нахождения собственных значений матрицы $S_W^{-1}S_B$:

```
>>> eigen_vals, eigen_vecs = \
...     np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

После нахождения собственных пар вектор-значение мы можем отсортировать собственные значения в порядке убывания:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                  for i in range(len(eigen_vals))]
>>> eigen_pairs = sorted(eigen_pairs,
...                       key=lambda k: k[0], reverse=True)
>>> print('Собственные значения по убыванию:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
Собственные значения по убыванию:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

В LDA количество линейных дискриминантов не превышает $c - 1$, где c — количество меток классов, поскольку матрица межклассового разброса S_B представляет собой сумму c матриц с рангом один или меньше. Мы действительно видим, что у нас есть только два ненулевых собственных значения (собственные значения с 3 по 13 не совсем равны нулю, но это связано с особенностями арифметики с плавающей запятой в NumPy).



Коллинеарность

В редком случае идеальной коллинеарности (все выровненные точки данных ложатся на прямую линию) ранг ковариационной матрицы будет равен единице, что означает наличие только одного собственного вектора с ненулевым собственным значением.

Чтобы измерить, какая часть информации о разделении классов извлекается линейными дискриминантами (собственными векторами), построим линейные дискриминанты по убыванию собственных значений, подобно диаграмме объясненной дисперсии, которую мы ранее создали для РСА. Для удобства назовем содержание информации о разделении классов *разделимостью*, или — ближе к ее английской транскрипции — *дискриминируемостью* (discriminability):

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real,
...                                         reverse=True)]
...
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, align='center',
...           label='Индивидуальная дискриминируемость')
...
>>> plt.step(range(1, 14), cum_discr, where='mid',
...           label='Накопительная дискриминируемость')
...
>>> plt.ylabel('Коэффициент дискриминируемости')
>>> plt.xlabel('Линейные дискриминанты')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 5.9, уже первые два линейных дискриминанта извлекают 100 % полезной информации о разделении классов в обучающем наборе данных Wine.

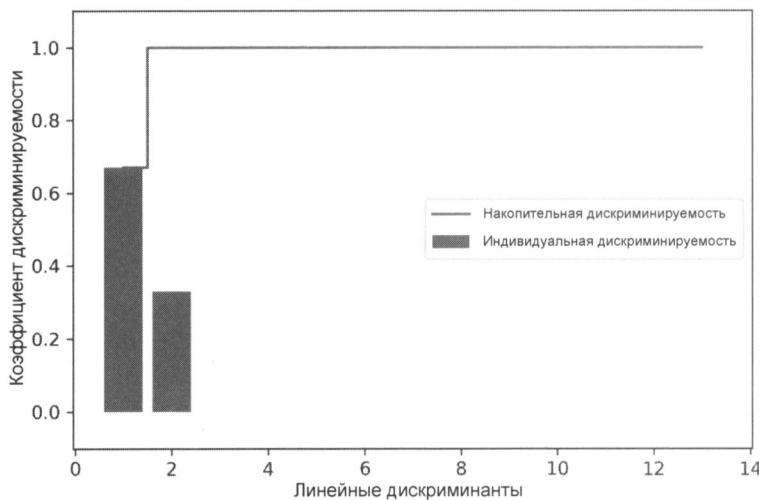


Рис. 5.9. Два лучших дискриминанта извлекают 100 % полезной информации

Теперь объединим два наиболее дискриминируемых столбца собственных векторов, чтобы создать матрицу преобразования:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                  eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Матрица W:\n', w)
```

Matrix W:

```
[[-0.1481 -0.4092]
 [ 0.0908 -0.1577]
 [-0.0168 -0.3537]
 [ 0.1484  0.3223]
 [-0.0163 -0.0817]
 [ 0.1913  0.0842]
 [-0.7338  0.2823]
 [-0.075  -0.0102]
 [ 0.0018  0.0907]
 [ 0.294  -0.2152]
 [-0.0328  0.2747]
 [-0.3547 -0.0124]
 [-0.3915 -0.5958]]
```

5.2.5. Проецирование точек данных на новое функциональное пространство

Используя матрицу преобразования W , которую мы создали в предыдущем подразделе, мы можем преобразовать набор обучающих данных, просто перемножив матрицы:

$$\mathbf{X}' = \mathbf{X}W$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['o', 's', '^']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label=f'Class {l}', marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 5.10, три класса набора данных Wine теперь идеально линейно разделимы в новом подпространстве признаков.

5.2.6. Реализация LDA при помощи scikit-learn

Предыдущая пошаговая реализация послужила хорошим упражнением для изучения механизма работы LDA и понимания различий между LDA и PCA. Теперь импортируем класс LDA, реализованный в scikit-learn:

```
>>> # оператор импорта ниже - это одна строка!
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

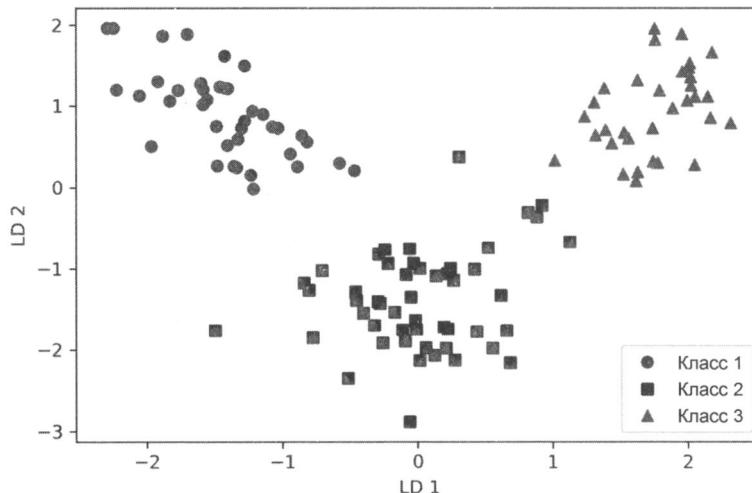


Рис. 5.10. Классы Wine идеально линейно разделимы после проецирования данных на первые два дискриминанта

В качестве примера посмотрим, как классификатор логистической регрессии обрабатывает набор обучающих данных с низкой размерностью после преобразования LDA:

```
>>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                         solver='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 5.11 мы видим, что модель логистической регрессии неправильно классифицирует один экземпляр из класса 2.

Снизив степень регуляризации, мы, вероятно, могли бы сдвинуть границы решений, чтобы модель логистической регрессии правильно классифицировала все примеры в обучающем наборе данных. Однако, что более важно, давайте посмотрим на результаты классификации тестового набора данных:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

Как мы видим на рис. 5.12, классификатор логистической регрессии может с идеальной точностью классифицировать экземпляры в тестовом наборе данных, используя только двумерное подпространство признаков вместо исходных 13 признаков Wine.

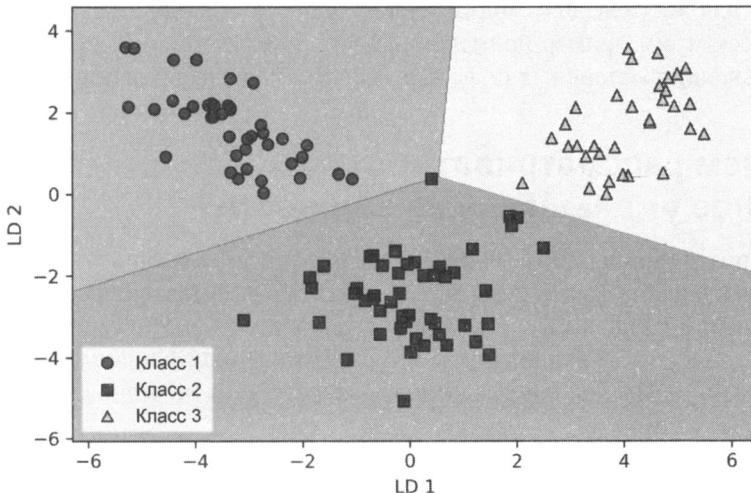


Рис. 5.11. Модель логистической регрессии неправильно определяет класс одного из экземпляров

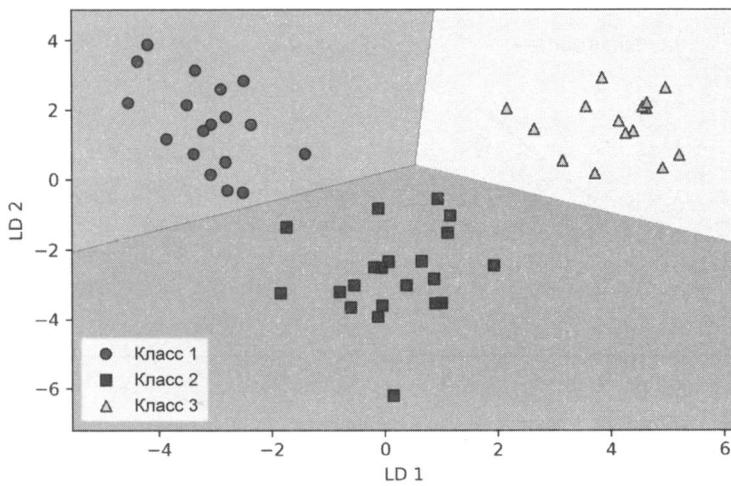


Рис. 5.12. Модель логистической регрессии отлично работает на тестовых данных

5.3. Нелинейное уменьшение размерности и визуализация

В предыдущем разделе мы рассмотрели методы извлечения признаков на основе линейных преобразований PCA и LDA. Далее мы обсудим, почему иногда может оказаться полезным нелинейное уменьшение размерности.

Один из методов нелинейного уменьшения размерности, на который стоит обратить особое внимание, — это *t-распределенное стохастическое встраивание соседей* (t-distributed Stochastic Neighbor Embedding, t-SNE), поскольку он часто используется

в литературе для визуализации многомерных наборов данных в двух или трех измерениях. Мы рассмотрим пример применения t-SNE для построения диаграммы распределения изображений рукописных цифр в двумерном пространстве признаков.

5.3.1. Зачем рассматривать нелинейное уменьшение размерности?

Многие алгоритмы машинного обучения делают предположения о линейной разделимости входных данных. Так, вы уже знаете, что для сходимости персептрону требуются идеально линейно разделимые обучающие данные. Другие алгоритмы, которые мы рассмотрели до сих пор, предполагают, что отсутствие идеальной линейной разделимости связано с шумом: Adaline, логистическая регрессия и стандартный SVM — это лишь некоторые из них.

Однако, если вам доведется иметь дело с нелинейными задачами, которые довольно часто встречаются в реальной жизни, методы линейного преобразования для уменьшения размерности, такие как PCA и LDA, могут оказаться не лучшим выбором (рис. 5.13).

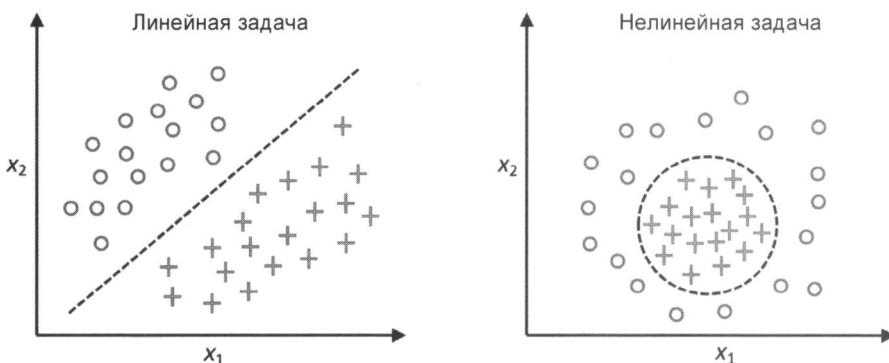


Рис. 5.13. Разница между линейными и нелинейными задачами

Библиотека scikit-learn предоставляет пользователям набор передовых методов нелинейного уменьшения размерности, рассмотрение которых выходит за рамки этой книги. Заинтересованный читатель найдет хороший обзор текущих реализаций алгоритмов этого типа в scikit-learn, дополненный демонстрационными примерами, по адресу: <http://scikit-learn.org/stable/modules/manifold.html>.

Разработку и применение методов нелинейного уменьшения размерности также часто называют *обучением на основе многообразий* (manifold learning), где многообразие означает встраивание топологического пространства более низкой размерности в многомерное пространство. Алгоритмы обучения на основе многообразий должны извлекать информацию о сложной структуре данных, чтобы спроектировать их на пространство с меньшими размерностями, где сохраняется связь между точками данных.

Классическим примером обучения на основе многообразий является фигура трехмерного «швейцарского рулета», показанная на рис. 5.14.

Различные виды трехмерного «швейцарского рулета»

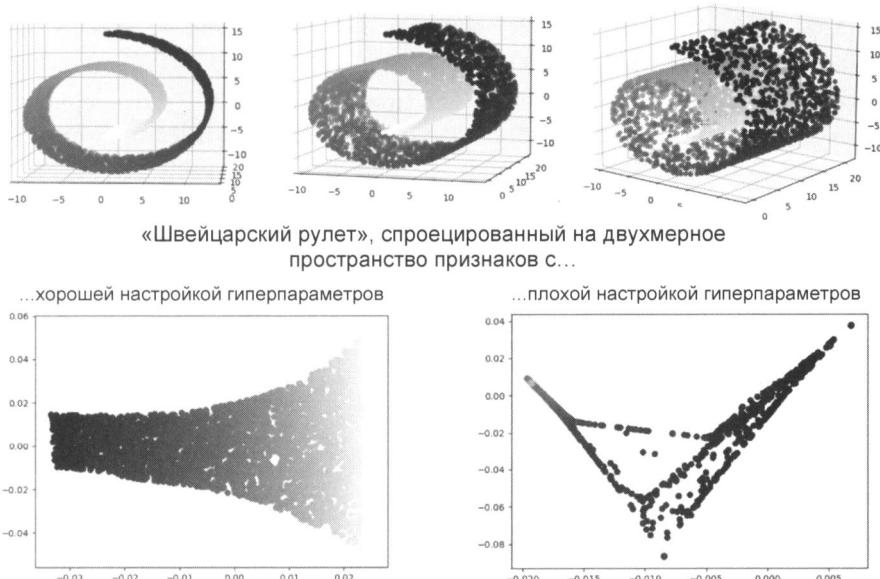


Рис. 5.14. Трехмерный «швейцарский рулет», спроектированный вниз в двумерное пространство

Хотя алгоритмы нелинейного уменьшения размерности и обучения на основе многообразия очень эффективны, необходимо отметить, что эти методы сложны в использовании, а при неудачном выборе гиперпараметров могут причинить больше вреда, чем пользы. Дело в том, что мы часто работаем с многомерными наборами данных, которые не можем легко визуализировать и структура которых не очевидна (в отличие от примера с рулем на рис. 5.14). Более того, если мы не спроектируем набор данных в два или три измерения (чего часто недостаточно для сохранения более сложных взаимосвязей), трудно или даже невозможно оценить качество результатов. По этой причине многие пользователи по-прежнему полагаются на более простые методы уменьшения размерности, такие как PCA и LDA.

5.3.2. Визуализация данных с помощью алгоритма t-SNE

Итак, мы познакомились с нелинейным уменьшением размерности и обсудили некоторые связанные с этим проблемы. Теперь рассмотрим практический пример использования t-SNE, который часто встречается при визуализации сложных наборов данных в двух или трех измерениях.

Если коротко, t-SNE моделирует точки данных на основе их попарных расстояний в многомерном (исходном) пространстве признаков. Затем он находит распределение вероятностей парных расстояний в новом пространстве меньшей размерности, которое близко к распределению вероятностей парных расстояний в исходном пространстве. Или, другими словами, t-SNE учится встраивать точки данных в пространство более низкой размерности, так что попарные расстояния в исходном пространстве сохра-

ются⁵. Однако нельзя забывать, что t-SNE — это метод визуализации, которому для проекции требуется весь набор данных. Поскольку он проецирует точки напрямую (в отличие от PCA, он не использует матрицу проекции), мы не можем применить t-SNE к новым точкам данных.

Следующий код представляет собой пример применения t-SNE к 64-мерному набору данных. Мы начинаем с загрузки набора данных Digits из scikit-learn, который состоит из изображений рукописных цифр с низким разрешением (цифры 0–9):

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
```

Цифры представляют собой изображения 8×8 в градациях серого. Следующий код отображает первые четыре изображения в наборе данных, который в общей сложности состоит из 1797 изображений:

```
>>> fig, ax = plt.subplots(1, 4)
>>> for i in range(4):
>>>     ax[i].imshow(digits.images[i], cmap='Greys')
>>> plt.show()
```

Как можно видеть на рис. 5.15, изображения имеют относительно низкое разрешение — 8×8 пикселов (т. е. 64 пикселя на изображение).

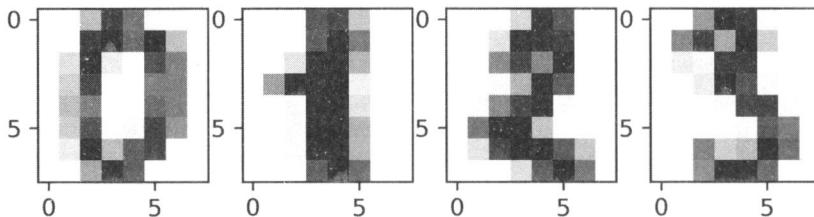


Рис. 5.15. Изображения рукописных цифр с низким разрешением

Атрибут `digits.data` позволяет нам получить доступ к табличной версии этого набора данных, где экземпляры представлены строками, а столбцы соответствуют пикселям:

```
>>> digits.data.shape
(1797, 64)
```

Далее присвоим признаки (пиксели) новой переменной `x_digits`, а метки — другой новой переменной `y_digits`:

```
>>> y_digits = digits.target
>>> X_digits = digits.data
```

Затем импортируем класс t-SNE из scikit-learn и обучим новый объект `tsne`. Используя метод `fit_transform`, мы выполняем обучение t-SNE и преобразование данных за один шаг:

⁵ Занинтересованные читатели найдут более подробную информацию об этом методе в научной статье «Visualizing data using t-SNE» by Maaten and Hinton, Journal of Machine Learning Research, 2018, <https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf>.

```
>>> from sklearn.manifold import TSNE  
>>> tsne = TSNE(n_components=2, init='pca',  
...                 random_state=123)  
>>> X_digits_tsne = tsne.fit_transform(X_digits)
```

Выполнив этот код, мы спроектировали 64-мерный набор данных на двумерное пространство. Мы указали здесь параметр `init='pca'`, который инициализирует встраивание t-SNE с использованием PCA, как это рекомендуется в статье Кобака и Линдермана⁶.

Нужно отметить, что у t-SNE есть дополнительные гиперпараметры — такие как *перплексия* и *скорость обучения* (часто называемая *эпсилон*), которые мы пропустили в примере (точнее, использовали значения scikit-learn по умолчанию). На практике мы рекомендуем вам изучить эти параметры⁷. Наконец, визуализируем 2D-встраивания t-SNE, используя следующий код:

```
>>> import matplotlib.path_effects as PathEffects  
>>> def plot_projection(x, colors):  
  
...     f = plt.figure(figsize=(8, 8))  
...     ax = plt.subplot(aspect='equal')  
...     for i in range(10):  
...         plt.scatter(x[colors == i, 0],  
...                     x[colors == i, 1])  
...     for i in range(10):  
...         xtext, ytext = np.median(x[colors == i, :], axis=0)  
...         txt = ax.text(xtext, ytext, str(i), fontsize=24)  
...         txt.set_path_effects([  
...             PathEffects.Stroke(linewidth=5, foreground="w"),  
...             PathEffects.Normal()])  
  
>>> plot_projection(X_digits_tsne, y_digits)  
>>> plt.show()
```

Как и PCA, метод t-SNE обучается без учителя, и в приведенном коде мы используем метки класса `y_digits` (0–9) только для целей визуализации, передавая функции аргумент цвета. Для наглядности применяется метод `PathEffects` из `Matplotlib`, так что метка класса отображается в центре группы точек данных, принадлежащих каждой соответствующей цифре (через `np.median`). Построенная при помощи этого кода диаграмма расстояния представлена на рис. 5.16.

Мы видим, что t-SNE довольно хорошо, хотя и не идеально, разделяет разные цифры (классы изображений). Можно добиться лучшего разделения, настроив гиперпараметры. Однако определенная степень смешения классов остается неизбежной из-за неразборчивого почерка. Например, изучая отдельные изображения, мы обнаружим, что некоторые экземпляры рукописной цифры 3 действительно похожи на 9, и т. д.

⁶ См. «Initialization is critical for preserving global data structure in both t-SNE and UMAP», Kobak and Linderman, *Nature Biotechnology* Volume 39, p. 156–157, 2021, <https://www.nature.com/articles/s41587-020-00809-z>.

⁷ Более подробную информацию об этих параметрах и их влиянии на результаты можно найти в превосходной статье «How to Use t-SNE Effectively», Wattenberg, Viegas, and Johnson, Distill, 2016, <https://distill.pub/2016/misread-tsne/>.

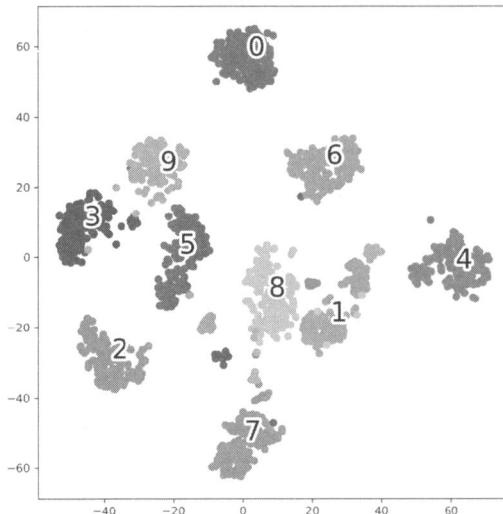


Рис. 5.16. Визуализация того, как t-SNE встраивает рукописные цифры в двухмерное пространство признаков



Равномерная аппроксимация и проекция на основе многообразия

Еще одним популярным методом визуализации является *равномерная аппроксимация и проекция на основе многообразия* (Uniform Manifold Approximation and Projection, UMAP). Метод UMAP может давать столь же хорошие результаты, что и t-SNE (например, см. статью Кобака и Линдермана, упомянутую ранее), но при этом он обычно быстрее работает, и его также можно использовать для проецирования новых данных, что делает его более привлекательным в качестве средства уменьшения размерности, заменяющего PCA. Заинтересованные читатели могут найти дополнительную информацию о UMAP в статье «UMAP: Uniform manifold approximation and projection for dimension reduction», McInnes, Healy, and Melville, 2018 (<https://arxiv.org/abs/1802.03426>). Совместимую со scikit-learn реализацию UMAP можно получить по адресу: <https://umap-learn.readthedocs.io>.

5.4. Заключение

В этой главе вы узнали о двух фундаментальных методах уменьшения размерности для извлечения признаков: PCA и LDA. Используя PCA, мы проецировали данные на подпространство более низкого измерения, чтобы максимизировать дисперсию вдоль ортогональных осей признаков, игнорируя при этом метки классов. LDA, в отличие от PCA, представляет собой метод уменьшения размерности с учителем, т. е. учитывает информацию о классе в наборе обучающих данных, чтобы попытаться максимизировать разделимость классов в линейном пространстве признаков. В заключительной части главы было рассказано о t-SNE — методе нелинейного извлечения признаков, который можно использовать для визуализации данных в двух или трех измерениях.

Вооружившись PCA и LDA в качестве основных методов предварительной обработки данных, вы хорошо подготовились к тому, чтобы узнать в следующей главе о применении различных эффективных методов предварительной обработки и оценки производительности моделей.

6

Современные методы оценки моделей и настройки гиперпараметров

В предыдущих главах было рассказано об основных алгоритмах машинного обучения для задач классификации и о том, как привести входные данные в нужную форму, прежде чем передать их в эти алгоритмы. Теперь пришло время узнать о современных методах создания качественных моделей машинного обучения путем точной настройки алгоритмов и оценки производительности. В этой главе вы узнаете о том, как:

- ◆ оценивать производительность моделей машинного обучения;
- ◆ диагностировать распространенные проблемы алгоритмов машинного обучения;
- ◆ выполнять точную настройку моделей машинного обучения;
- ◆ оценивать прогностические модели с использованием различных показателей производительности.

6.1. Оптимизация рабочих процессов с помощью конвейеров

В предыдущих главах вы изучили различные методы предварительной обработки данных — такие как стандартизация для масштабирования признаков (глава 4) и анализ главных компонент (глава 5). Вы узнали, что параметры сжатия, найденные на обучающем наборе, необходимо применять при обработке новых входных данных. В этом разделе вы научитесь работать с очень удобным инструментом в составе scikit-learn — классом конвейера `Pipeline`. Конвейер позволяет обучить модель, состоящую из произвольного количества шагов преобразования, и применить ее для прогнозирования новых данных.

6.1.1. Загрузка набора данных по раку молочной железы в Висконсине

В этой главе мы будем работать с набором данных по раку молочной железы в штате Висконсин, США, который содержит 569 записей о злокачественных и доброкачественных опухолевых клетках. В первых двух столбцах набора данных хранятся уникальные идентификационные номера образцов и соответствующие диагнозы: M — злокачественный (`malignant`), B — доброкачественный (`benign`) соответственно. Столбцы с 3-го по 32-й содержат 30 признаков в виде действительных чисел, рассчитанных на

основе оцифрованных изображений ядер клеток, которые можно использовать для построения модели предсказания характера опухоли. Набор данных по раку молочной железы, собранный в штате Висконсин, был размещен в репозитории машинного обучения UCI, а более подробную информацию об этом наборе можно найти по адресу: [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)).



Получение набора данных по раку молочной железы в Висконсине

Вы можете найти копию этого набора данных (и всех других наборов данных, используемых в книге) в ее файловом архиве, когда вы работаете в автономном режиме или когда сервер UCI по адресу: <https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data>, временно недоступен. Например, чтобы загрузить набор данных из локального каталога, нужно заменить в коде загрузки набора, приведенном далее, строки:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases'
    '/breast-cancer-wisconsin/wdbc.data',
    header=None
)
```

на следующие:

```
df = pd.read_csv(
    'your/local/path/to/wdbc.data',
    header=None
)
```

Далее мы прочитаем исходный набор данных и разделим его на обучающий и тестовый наборы в три простых шага:

1. Загрузим набор данных непосредственно с веб-сайта UCI с помощью pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                   'machine-learning-databases'
...                   '/breast-cancer-wisconsin/wdbc.data',
...                   header=None)
```

2. Присвоим 30 признаков массиву NumPy x. Используя объект LabelEncoder, преобразуем метки классов из их исходного строкового представления ('м' и 'в') в целые числа:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2: ].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> le.classes_
array(['B', 'M'], dtype=object)
```

3. После кодирования меток классов (диагнозов) в массиве у злокачественные опухоли теперь представлены как класс 1, а доброкачественные опухоли — как класс 0.

Можно проверить это сопоставление, вызвав метод `transform` обученного объекта `LabelEncoder` для двух фиктивных меток класса:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

Прежде чем приступить к построению первого контейнера модели, разделим исходный набор данных на отдельные поднаборы для обучения (80% исходного набора) и тестирования (20% исходного набора):

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                     test_size=0.20,
...                     stratify=y,
...                     random_state=1)
```

6.1.2. Объединение преобразователей и оценивателей в конвейер

Из рассмотренных ранее глав вы узнали, что для достижения оптимальной производительности многим алгоритмам машинного обучения нужны входные признаки, представленные в одном масштабе. Поскольку масштабы признаков в наборе данных о раке молочной железы значительно различаются, необходимо стандартизировать столбцы в наборе данных, прежде чем передавать их линейному классификатору, такому как логистическая регрессия. Кроме того, предположим, что нам нужно сжать данные из начального 30-мерного пространства в более низкое двумерное подпространство с помощью анализа главных компонент (PCA) — метода извлечения признаков для уменьшения размерности, который был представлен в [главе 5](#).

Вместо того, чтобы выполнять этапы обучения модели и преобразования данных для обучающего и тестового наборов данных по отдельности, мы можем объединить объекты `StandardScaler`, `PCA` и `LogisticRegression` в конвейер:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression())
>>> pipe_lr.fit(X_train, y_train)
>>> y_pred = pipe_lr.predict(X_test)
>>> test_acc = pipe_lr.score(X_test, y_test)
>>> print(f'Точность на тестовых данных: {test_acc:.3f}')
Точность на тестовых данных: 0.956
```

Функция `make_pipeline` принимает произвольное количество преобразователей `scikit-learn` (объектов, принимающих методы `fit` и `transform` в качестве входных данных), за которыми следует оцениватель `scikit-learn`, реализующий методы `fit` и `predict`. В приведенном примере кода мы передали два преобразователя `scikit-learn`: `StandardScaler` и `PCA`,

а также оцениватель `LogisticRegression` в качестве входных данных для функции `make_pipeline`, которая создает объект `Pipeline scikit-learn` из этих входных объектов.

Объект `Pipeline` можно рассматривать как метаоцениватель или оболочку для отдельных преобразователей и оценивателей. Если мы вызовем метод `fit` объекта `Pipeline`, то данные будут передаваться по цепочке преобразователей через вызовы методов `fit` и `transform` на этих промежуточных этапах, пока не достигнут оценивателя (последнего элемента в конвейере). Затем оцениватель будет обучен на преобразованных данных.

Когда мы выполняли метод `fit` в конвейере `pipe_lr` в приведенном примере кода, преобразователь `StandardScaler` сначала осуществил вызовы методов `fit` и `transform` для обучающих данных. Затем преобразованные обучающие данные были переданы следующему объекту в конвейере — `PCA`. Как и на предыдущем шаге, `PCA` тоже вызвал методы `fit` и `transform`, но уже для масштабированных входных данных, и передал их в оцениватель, который завершает конвейер.

Наконец, оцениватель `LogisticRegression` был обучен на данных, которые подверглись преобразованиям с помощью `StandardScaler` и `PCA`. Здесь следует подчеркнуть, что количество промежуточных шагов в конвейере не ограничено, однако, если мы хотим использовать конвейер для задач прогнозирования, последний элемент конвейера должен быть оценивателем.

Подобно вызову метода `fit`, конвейеры также реализуют метод `predict`, если последним шагом в конвейере является оцениватель. Когда мы вызываем метод `predict` экземпляра объекта `Pipeline` и передаем в него данные, они будут проходить через промежуточные шаги преобразования при вызовах метода `transform`. На последнем этапе объект оценивателя вернет прогноз на основе преобразованных данных.

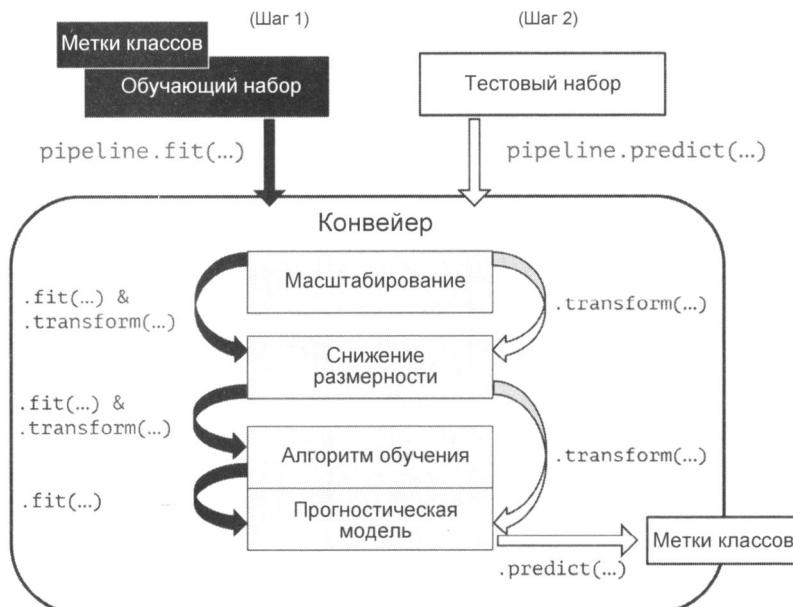


Рис. 6.1. Внутренние процессы объекта `Pipeline` (конвейер)

Конвейеры библиотеки scikit-learn — чрезвычайно полезные инструменты-оболочки, которые мы будем часто использовать в оставшейся части этой книги. Чтобы как следует разобраться в работе объекта Pipeline, внимательно изучите рис. 6.1, на котором схематично представлена последовательность процессов, происходящих в конвейере.

6.2. Использование к-кратной перекрестной проверки для оценки производительности модели

В этом разделе вы узнаете о двух наиболее популярных методах перекрестной проверки: *перекрестной проверке с отложенными данными* (holdout cross-validation) и *k-кратной перекрестной проверке* (k-fold cross-validation), которые позволяют получить надежные оценки обобщающей способности модели, т. е. того, насколько хорошо модель работает на незнакомых объектах.

6.2.1. Перекрестная проверка с отложенными данными

Классическим и популярным подходом к оценке эффективности обобщения моделей машинного обучения является *метод откладывания* (holdout method). В соответствии с этим методом мы разделяем наш первоначальный набор данных на отдельные поднаборы: обучающий и тестовый. Первый используется для обучения модели, а второй — для оценки ее обобщающей способности. Однако в типичных приложениях машинного обучения нас также интересует точная настройка модели и сравнение различных сочетаний параметров для дальнейшего повышения точности при прогнозировании на незнакомых данных. Этот процесс называется *выбором модели* (model selection), но фактически мы решаем задачу выбора *оптимальных* значений параметров настройки (также называемых *гиперпараметрами*). Однако, если во время выбора модели мы снова и снова используем один и тот же набор тестовых данных, постепенно он станет частью наших обучающих данных¹, и, следовательно, модель с большей вероятностью будет страдать от переобучения. Несмотря на этот очевидный недостаток, многие специалисты по-прежнему используют тестовый набор данных для выбора модели, что нельзя назвать хорошей практикой машинного обучения.

Более надежным вариантом метода откладывания при выборе модели является разделение данных на *три* поднабора: обучающий (training dataset), валидационный (validation dataset) и тестовый (test dataset). Сначала на обучающем наборе обучают несколько моделей с различными значениями гиперпараметров, затем производительность моделей оценивают на проверочном поднаборе и по результатам оценки выбирают лучшую модель. Преимущество наличия тестового набора данных, который модель не могла видеть на этапах обучения и выбора, заключается в том, что мы получаем менее предвзятую оценку ее способности обобщать новые данные. Рис. 6.2 иллюстрирует идею перекрестной проверки с отложенными данными, где валидационный набор используют для оценки производительности моделей, обученных с использованием различных значений гиперпараметров. Когда найдено приемлемое сочетание гипер-

¹ Это явление называют *утечкой тестовых данных*. — Прим. пер.

параметров и отобрана соответствующая модель, ее обобщающую способность окончательно проверяют на тестовом наборе данных.

Недостатком метода откладывания является чрезмерная чувствительность оценки производительности к способу, которым выбирают подмножества для обучения и проверки. Иными словами, оценка может существенно различаться на разных выборках из одного и того же исходного набора. Далее мы рассмотрим более надежный метод оценки производительности — k -кратную перекрестную проверку. Фактически это повторение метода откладывания k раз на k подмножествах обучающих данных.



Рис. 6.2. Использование трех разных наборов данных для обучения, проверки и тестиования

6.2.2. k -кратная перекрестная проверка

При k -кратной перекрестной проверке обучающий набор данных случайным образом разбивают на k выборок без замены. Затем $k - 1$ выборок, которые также называют *обучающими подвыборками* (*training fold*), используют для обучения модели, а на одной оставшейся выборке выполняют оценку производительности. Эта процедура повторяется k раз, поэтому мы получаем k моделей и оценок производительности.



Выборка с возвращением и без возвращения

Мы рассмотрели пример, иллюстрирующий выборку с возвращением и без возвращения в главе 3. Если вы не читали эту главу или хотите освежить в памяти информацию, обратитесь к соответствующей врезке в разд. 3.6.3.

Затем мы вычисляем среднюю производительность моделей на основе нескольких независимых тестовых выборок, чтобы получить оценку производительности, которая

менее чувствительна к разбиению обучающих данных по сравнению с методом откладывания данных. Как правило, k -кратную перекрестную проверку применяют для настройки модели, т. е. нахождения оптимальных значений гиперпараметров, которые обеспечивают удовлетворительную обобщающую способность модели, измеренную на тестовых выборках.

Когда найдены удовлетворительные значения гиперпараметров, модель повторно обучаются на полном наборе данных и получают окончательную оценку производительности при помощи независимых тестов. Смысль обучения модели на полном наборе данных после k -кратной перекрестной проверки заключается в том, что, во-первых, нас обычно интересует одна окончательная модель (а не k отдельных моделей), а во-вторых, использование большего объема обучающих данных обычно дает более точную и надежную модель.

Поскольку k -кратная перекрестная проверка представляет собой метод повторной выборки без возвращения, преимущество такого подхода в том, что в каждой итерации каждая запись будет использоваться только один раз, а обучающая и тестовая подвыборки не пересекаются. Кроме того, и тестовые подвыборки также не пересекаются между собой — т. е. между тестовыми подвыборками нет перекрытия. На рис. 6.3 схематически представлена концепция k -кратной перекрестной проверки с $k = 10$. Набор обучающих данных разделен на 10 выборок, и в течение 10 итераций 9 выборок служат для обучения, а одна выборка выступает в роли тестового набора данных для оценки модели.

Кроме того, расчетные характеристики E_i (например, точность классификации или ошибка), полученные для каждой выборки, затем используются для вычисления расчетной средней производительности E модели.

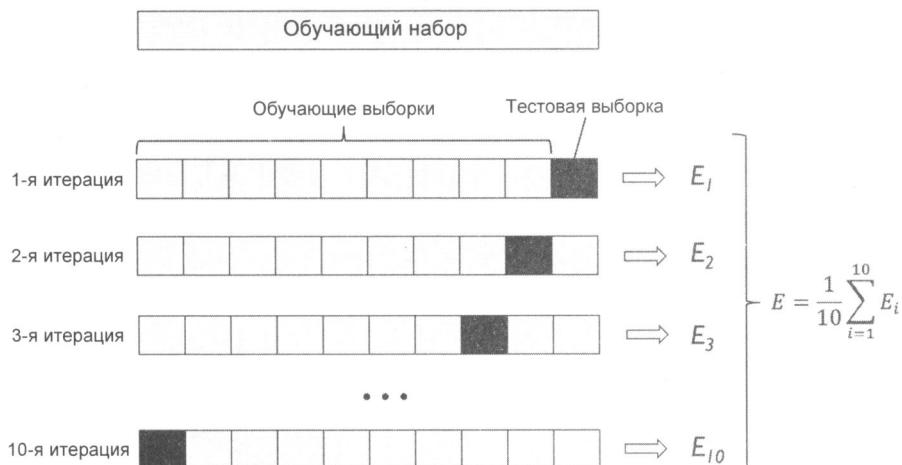


Рис. 6.3. Принцип работы k -кратной перекрестной проверки

Следовательно, k -кратная перекрестная проверка лучше использует набор данных, чем метод откладывания, поскольку при k -кратной перекрестной проверке для оценки используются все точки данных.

Как показывают эмпирические данные, хорошим типовым значением k для k -кратной перекрестной проверки является 10. Например, эксперименты Рона Кохави с различными наборами реальных данных показывают, что 10-кратная перекрестная проверка обеспечивает наилучший компромисс между систематической ошибкой и дисперсией².

Однако, если мы работаем с относительно небольшими обучающими наборами, может быть полезно увеличить количество выборок. Если мы увеличим значение k , на каждой итерации будет доступно больше обучающих данных, что приведет к меньшему пессимистическому смещению при оценке эффективности обобщения за счет усреднения оценок отдельных моделей. Однако большие значения k также увеличат время работы алгоритма перекрестной проверки и дадут оценки с более высокой дисперсией, поскольку обучающие выборки будут больше похожи друг на друга. Если мы работаем с большими наборами данных, то можем смело выбрать меньшее значение для k — например, $k = 5$, и все же получить точную оценку средней производительности модели при снижении вычислительных затрат на повторное обучение и оценку модели на разных выборках.



Перекрестная проверка с исключением по одному

Частным случаем k -кратной перекрестной проверки является метод *перекрестной проверки с исключением по одному* (Leave-One-Out Cross-Validation, LOOCV). В LOOCV мы устанавливаем количество выборок равным количеству обучающих записей ($k = n$) — чтобы на каждой итерации для тестирования использовался только один обучающий экземпляр. Этот метод рекомендуется применять при работе с очень маленькими наборами данных.

Некоторое улучшение по сравнению со стандартным подходом k -кратной перекрестной проверки дает *стратифицированная* k -кратная перекрестная проверка, которая обеспечивает более точные оценки смещения и дисперсии, особенно в случаях неравных долей классов, что также было показано в исследовании Рона Кохави, упомянутом ранее в этом разделе. При стратифицированной перекрестной проверке в каждой подвыборке тщательно сохраняют соотношение классов, чтобы гарантировать, что все подвыборки одинаково представляют пропорции классов в обучающем наборе данных. Следующий код иллюстрирует реализацию стратифицированной проверки с помощью оператора `StratifiedKFold`, входящего в библиотеку scikit-learn:

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print(f'Выборка: {k+1:02d}, '
...           f'Распр. кл.: {np.bincount(y_train[train])}, '
...           f'Точн.: {score:.3f}')
... 
```

² См.: «A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection» by Kohavi, Ron, International Joint Conference on Artificial Intelligence (IJCAI), 14 (12): 1137–1143, 1995, <https://www.ijcai.org/Proceedings/95-2/Papers/016.pdf>.

```

Выборка: 01, Распр. кл.: [256 153], Точн.: 0.935
Выборка: 02, Распр. кл.: [256 153], Точн.: 0.935
Выборка: 03, Распр. кл.: [256 153], Точн.: 0.957
Выборка: 04, Распр. кл.: [256 153], Точн.: 0.957
Выборка: 05, Распр. кл.: [256 153], Точн.: 0.935
Выборка: 06, Распр. кл.: [257 153], Точн.: 0.956
Выборка: 07, Распр. кл.: [257 153], Точн.: 0.978
Выборка: 08, Распр. кл.: [257 153], Точн.: 0.933
Выборка: 09, Распр. кл.: [257 153], Точн.: 0.956
Выборка: 10, Распр. кл.: [257 153], Точн.: 0.956
>>> mean_acc = np.mean(scores)
>>> std_acc = np.std(scores)
>>> print(f'\n Точность по CV: {mean_acc:.3f} +/- {std_acc:.3f}')
Точность по CV: 0.950 +/- 0.014

```

Здесь мы сначала инициализируем итератор `StratifiedKFold` из модуля `sklearn.model_selection` с метками класса `y_train` в обучающем наборе данных и указываем количество выборок через параметр `n_splits`. Мы также задействовали итератор `kfold` для перебора к выборкам и применили возвращенные в `train` индексы для обучения модели логистической регрессии через конвейер, который настроили в начале этой главы. Используя конвейер `pipe_lr`, мы обеспечили правильное масштабирование экземпляров данных (например, стандартизацию) на каждой итерации. Затем мы применили индексы `test` для вычисления оценок точности моделей и помещения их в список `scores` с последующим расчетом средней точности и стандартного отклонения.

Хотя приведенный пример кода хорошо иллюстрирует работу k -кратной перекрестной проверки, `scikit-learn` также содержит метод `cross_val_score`, который позволяет нам менее хлопотно оценивать нашу модель с помощью стратифицированной k -кратной перекрестной проверки:

```

>>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                           X=X_train,
...                           y=y_train,
...                           cv=10,
...                           n_jobs=1)
>>> print(f'Оценки точности по CV: {scores}')
Оценки точности по CV: [ 0.93478261  0.93478261  0.95652174
                           0.95652174  0.93478261  0.95555556
                           0.97777778  0.93333333  0.95555556
                           0.95555556]
>>> print(f'Точность по CV: {np.mean(scores):.3f} '
...       f' +/- {np.std(scores):.3f}')
Точность по CV: 0.950 +/- 0.014

```

Чрезвычайно полезная особенность `cross_val_score` заключается в том, что мы можем распределить оценку по различным выборкам между несколькими ядрами центрального процессора (Central Processing Unit, CPU) своей машины. Если мы установим параметр `n_jobs=1`, для оценки производительности будет задействовано только одно ядро CPU, как и в нашем примере с использованием `StratifiedKFold` ранее. Однако, установив

`n_jobs=2`, мы могли бы распределить 10 раундов перекрестной проверки на два ядра CPU (если они доступны на имеющейся машине), а установив `n_jobs=-1`, мы можем задействовать все доступные ядра CPU на нашей машине для выполнения параллельных вычислений.



Оценка обобщающей способности модели

Подробное обсуждение того, как оценивается дисперсия обобщающей способности при перекрестной проверке, выходит за рамки этой книги, но вы можете обратиться к исчерпывающей статье об оценке модели и перекрестной проверке («Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning» by S. Raschka), которая свободно доступна по адресу: <https://arxiv.org/abs/1811.12808>. В этой статье также обсуждаются альтернативные способы перекрестной проверки — такие как бутстрэп-методы перекрестной проверки .632 и .632+.

Кроме того, вы можете найти подробный анализ вопроса в отличной статье M. Маркатоу и др. («Analysis of Variance of Cross-validation Estimators of the Generalization Error» by M. Markatou, H. Tian, S. Biswas, and G. M. Hripcak, Journal of Machine Learning Research, 6: 1127–1168, 2005), доступной по адресу: <https://www.jmlr.org/papers/v6/markatou05a.html>.

6.3. Алгоритмы отладки с использованием кривых обучения и валидации

В этом разделе мы познакомимся с двумя очень простыми, но мощными диагностическими инструментами, которые улучшают производительность алгоритма обучения: *кривой обучения* (learning curve) и *кривой валидации* (validation curve). Затем мы рассмотрим применение кривых обучения для обнаружения проблем с переобучением (высокая дисперсия) или недообучением (высокое смещение), после чего обсудим, как кривые валидации помогают решить общие проблемы алгоритмов обучения.

6.3.1. Диагностика смещения и дисперсии с помощью кривых обучения

Если модель слишком сложна для имеющегося обучающего набора данных — например, с очень глубоким деревом решений, — она, как правило, переобучается и плохо обобщает новые данные. Тогда — чтобы уменьшить риск переобучения — бывает иногда полезно добавить в набор обучающие записи.

Однако на практике часто оказывается так, что найти дополнительные обучающие данные бывает очень дорого или просто невозможно. Построив график точности модели на обучающем и проверочном наборах как функцию от размера обучающего набора, мы легко определим, страдает ли модель от высокой дисперсии или высокого смещения и может ли сбор дополнительных данных помочь решить эту проблему.

Но прежде чем перейти к построению кривых обучения в scikit-learn, давайте обсудим упомянутые общие проблемы моделей, воспользовавшись обобщенным примером, приведенным на рис. 6.4.

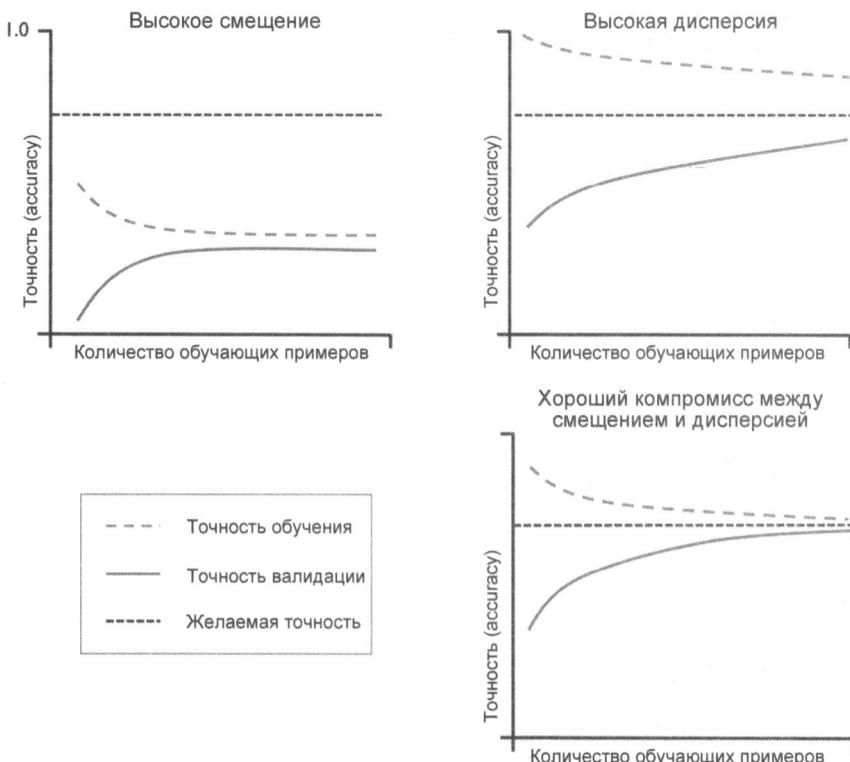


Рис. 6.4. Общие проблемы моделей

На графике в верхнем левом углу рис. 6.4 показана модель с высоким смещением. Эта модель имеет низкую точность при обучении и перекрестной проверке, что указывает на недостаточное обучение. Распространенными способами решения этой проблемы являются увеличение количества параметров модели — например, за счет извлечения или построения дополнительных признаков, или уменьшение степени регуляризации — например, в методе опорных векторов (SVM) или в классификаторах логистической регрессии.

В правом верхнем углу показана модель с высокой дисперсией, на что указывает большой разрыв между точностью обучения и перекрестной проверки (валидации). Это явный признак переобучения модели. Чтобы решить проблему переобучения, мы можем, например, собрать больше обучающих данных, уменьшить сложность модели или увеличить коэффициент регуляризации.

В случае нерегуляризованных моделей также помогает уменьшение количества признаков за счет их отбора (глава 4) или извлечения (глава 5), что уменьшает степень переобучения. Хотя известно, что увеличение объема обучающих данных обычно снижает вероятность переобучения, это не всегда может помочь — например, если обучающие данные очень зашумлены или модель уже очень близка к оптимальной.

Далее вы научитесь решать эти проблемы модели с помощью кривых валидации, но сначала попробуем использовать функцию кривой обучения из scikit-learn для оценки модели:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.model_selection import learning_curve
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         LogisticRegression(penalty='l2',
...                                             max_iter=10000))
>>> train_sizes, train_scores, test_scores = \
...
...             learning_curve(estimator=pipe_lr,
...                             X=X_train,
...                             y=y_train,
...                             train_sizes=np.linspace(
...                                 0.1, 1.0, 10),
...                             cv=10,
...                             n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...            color='blue', marker='o',
...            markersize=5, label='Training accuracy')
>>> plt.fill_between(train_sizes,
...                     train_mean + train_std,
...                     train_mean - train_std,
...                     alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...            color='green', linestyle='--',
...            marker='s', markersize=5,
...            label='Validation accuracy')
>>> plt.fill_between(train_sizes,
...                     test_mean + test_std,
...                     test_mean - test_std,
...                     alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Количество обучающих примеров')
>>> plt.ylabel('Точность')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.03])
>>> plt.show()
```

Обратите внимание, что мы использовали параметр `max_iter=10000` в качестве дополнительного аргумента при создании экземпляра объекта `LogisticRegression` (который по умолчанию использует 1000 итераций), чтобы избежать проблем сходимости при наборе данных меньшего размера или при экстремальных значениях параметров регуляризации (мы обсудим это в следующем разделе). После успешного выполнения приведенного кода мы получим график кривой обучения, показанный на рис. 6.5.

С помощью параметра `train_sizes` в функции `learning_curve` мы можем задавать абсолютное или относительное количество примеров (экземпляров) в обучающем наборе,

которые используются для построения кривых обучения. В нашем случае мы устанавливаем `train_sizes=np.linspace(0.1, 1.0, 10)`, чтобы разбить набор обучающих данных на 10 равномерно распределенных интервалов. По умолчанию функция `learning_curve` использует стратифицированную k -кратную перекрестную проверку для расчета точности классификатора, и мы установили значение $k = 10$ с помощью параметра `cv` для выполнения 10-кратной стратифицированной перекрестной проверки.

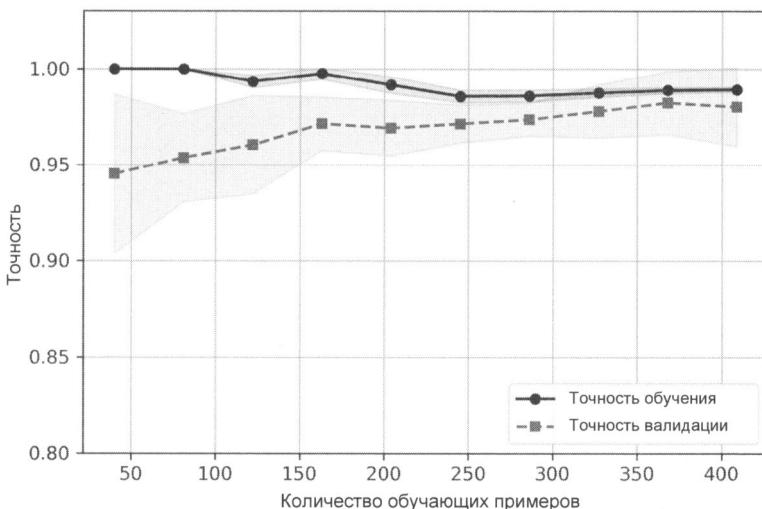


Рис. 6.5. Кривая обучения, показывающая точность на обучающем и проверочном наборах данных в зависимости от количества обучающих примеров

Затем из полученных результатов перекрестной проверки мы вычислили среднюю точность модели для разных значений величины набора обучающих данных и вывели на график с помощью функции `plot` Matplotlib. Кроме того, мы добавили к графику стандартное отклонение средней точности, используя функцию `fill_between`, чтобы показать дисперсию оценки.

По этому графику можно сделать вывод, что наша модель достаточно хорошо работает как с обучающими, так и с проверочными наборами данных, если для обучения применялось более 250 экземпляров. Мы также видим, что при использовании обучающих наборов данных с менее чем 250 экземплярами точность обучения увеличивается, но и разрыв между точностью на обучающем и проверочном наборе тоже растет — это показатель нарастающего переобучения.

6.3.2. Устранение переобучения и недообучения с помощью кривых валидации

Кривые валидации — это полезный инструмент для улучшения качества модели путем решения проблем, вызванных как переобучением, так и недообучением. Кривые валидации связаны с кривыми обучения, но вместо того, чтобы строить кривые в зависимости от размера набора, мы варьируем значения параметров модели — например, обратный параметр регуляризации C в логистической регрессии.

Давайте создадим кривую валидации с помощью scikit-learn:

```
>>> from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                 estimator=pipe_lr,
...                 X=X_train,
...                 y=y_train,
...                 param_name='logisticregression__C',
...                 param_range=param_range,
...                 cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...            color='blue', marker='o',
...            markersize=5, label='Точность обучения')
>>> plt.fill_between(param_range, train_mean + train_std,
...                    train_mean - train_std, alpha=0.15,
...                    color='blue')
>>> plt.plot(param_range, test_mean,
...            color='green', linestyle='--',
...            marker='s', markersize=5,
...            label='Точность валидации')
>>> plt.fill_between(param_range,
...                    test_mean + test_std,
...                    test_mean - test_std,
...                    alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Параметр С')
>>> plt.ylabel('Точность')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Этот код строит кривую валидации относительно параметра С (рис. 6.6).

Подобно функции `learning_curve`, функция `validation_curve` по умолчанию использует стратифицированную k -кратную перекрестную проверку для оценки производительности классификатора. При вызове функции `validation_curve` мы указали параметр, влияние которого хотели оценить. В нашем случае — это С, обратный параметр регуляризации классификатора `LogisticRegression`, который мы обозначили как `'logisticregression__C'` для доступа к объекту `LogisticRegression` внутри конвейера `scikit-learn` в указанном диапазоне значений, который был задан с помощью параметра `param_range`. Как и в примере с кривой обучения в предыдущем разделе, мы нанесли на график среднюю точность обучения и перекрестной проверки (валидации) и соответствующие стандартные отклонения.

Хотя различия в точности для разных значений С незначительны, мы можем видеть, что модель становится немного недообучена, когда мы увеличиваем степень регуляри-

зации (малые значения С). В то же время большие значения С означают снижение степени регуляризации, поэтому у модели возникает склонность к переобучению. В таком случае область компромисса соответствует диапазону значений С от 0.1 до 1.0.

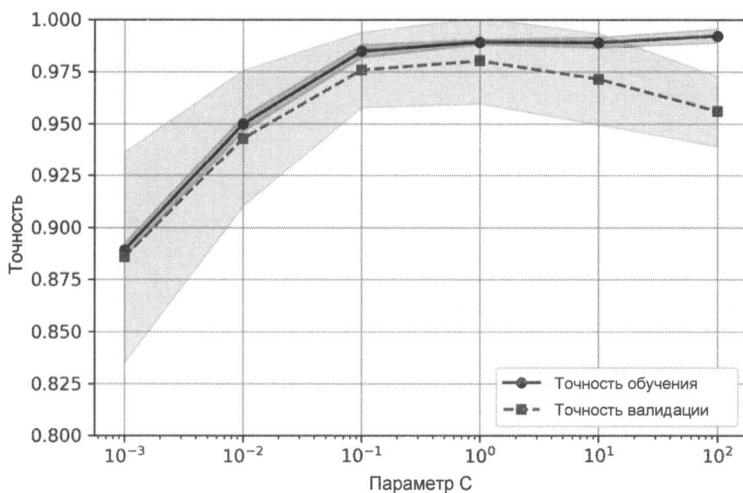


Рис. 6.6. Кривая валидации относительно гиперпараметра С при использовании метода опорных векторов (SVM)

6.4. Точная настройка моделей с помощью поиска по сетке

В машинном обучении у нас есть два типа параметров: извлекаемые из обучающих данных — например, весовые коэффициенты в логистической регрессии, и параметры алгоритма обучения, которые оптимизируются отдельно. Последние являются настроичными параметрами (или гиперпараметрами) модели. К ним относятся, например, параметр регуляризации в логистической регрессии или параметр максимальной глубины дерева решений.

В предыдущем разделе мы использовали кривую валидации для повышения производительности модели за счет настройки одного из ее гиперпараметров. В этом разделе мы рассмотрим популярный метод оптимизации гиперпараметров, называемый *поиском по сетке* (grid search), который способен улучшить производительность модели путем нахождения оптимальной комбинации значений гиперпараметров.

6.4.1. Настройка гиперпараметров с помощью поиска по сетке

Принцип работы поиска по сетке довольно прост и заключается в полном переборе вариантов, когда мы указываем список значений для разных гиперпараметров, а компьютер оценивает производительность модели для каждой комбинации, чтобы найти оптимальную комбинацию значений из этого списка:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = make_pipeline(StandardScaler(),
...                           SVC(random_state=1))
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                  1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{ 'svc__C': param_range,
...                  'svc__kernel': ['linear']},
...                 { 'svc__C': param_range,
...                  'svc__gamma': param_range,
...                  'svc__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring='accuracy',
...                     cv=10,
...                     refit=True,
...                     n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.9846153846153847
>>> print(gs.best_params_)
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

В этом коде мы инициализировали объект `GridSearchCV` из модуля `sklearn.model_selection` для обучения и настройки конвейера SVM. Мы назначаем параметру `param_grid` объекта `GridSearchCV` список словарей, чтобы указать параметры, которые мы хотим настроить. Для линейного SVM мы оценили только обратный параметр регуляризации `C`, а для радиальной базисной функции (RBF) ядра SVM настроили параметры `svc__C` и `svc__gamma`. Учтите, что параметр `svc_gamma` применяется только для ядерных SVM.

Для сравнения моделей, обученных с различными настройками гиперпараметров, `GridSearchCV` использует k -кратную перекрестную проверку. С учетом параметра `cv=10` он будет выполнять 10-кратную перекрестную проверку и вычислять среднюю точность (поскольку задан параметр `scoring='accuracy'`) по этим 10 выборкам, чтобы оценить производительность модели. Мы задали параметр `n_jobs=-1`, чтобы `GridSearchCV` мог использовать все ядра процессора для ускорения поиска в сетке, параллельно обучая модели на разных выборках, но если на вашем компьютере этот параметр вызывает проблемы выполнения, вы можете изменить его значение на `n_jobs=None`, чтобы оценивать модели по порядку на одном ядре.

Завершив поиск по сетке, мы получаем оценку наиболее эффективной модели с помощью атрибута `best_score_` и просматриваем ее параметры с использованием атрибута `best_params_`. В этом конкретном случае SVM-модель с ядром RBF и параметром `svc__C = 100.0` дала наилучшую точность при k -кратной перекрестной проверке: 98.5 %.

Наконец, мы используем независимый тестовый набор данных для оценки производительности выбранной модели, которая доступна через атрибут `best_estimator_` объекта `GridSearchCV`:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
```

```
>>> print(f'Точность при тестировании: {clf.score(X_test, y_test):.3f}')
Точность при тестировании: 0.974
```

Обратите внимание, что после завершения поиска по сетке не требуется вручную запускать обучение модели с лучшими настройками (`gs.best_estimator_`) на обучающем наборе через `clf.fit(X_train, y_train)`. Класс `GridSearchCV` имеет параметр `refit`, который автоматически обучит `gs.best_estimator_` на полном обучающем наборе, если установлен параметр `refit=True` (по умолчанию).

6.4.2. Изучение обширных конфигураций гиперпараметров с помощью рандомизированного поиска

Поскольку поиск по сетке является *исчерпывающим* (т. е. предусматривает перебор *всех* вариантов), оптимальная конфигурация гиперпараметров гарантированно будет найдена, если она содержится в сетке параметров, заданной пользователем. Однако на практике поиск по большим сеткам гиперпараметров обходится слишком дорого. Альтернативным подходом к выборке различных комбинаций параметров является *рандомизированный поиск* (randomized search). При рандомизированном поиске мы случайным образом выбираем лишь некоторые конфигурации гиперпараметров из распределений (или дискретных наборов). Тем не менее этот подход позволяет нам исследовать более широкий диапазон значений гиперпараметров, причем сделать это более эффективным способом с точки зрения вычислительных затрат и времени. Рандомизированный поиск схематически представлен на рис. 6.7, где показана фиксированная сетка из девяти настроек гиперпараметров, а поиск осуществляется с помощью поиска по сетке и рандомизированного поиска.

Главный вывод заключается в том, что, хотя поиск по сетке тщательно исследует все дискретные, указанные пользователем варианты, он может пропустить хорошие конфигурации гиперпараметров, если пространство для поиска слишком мало³.

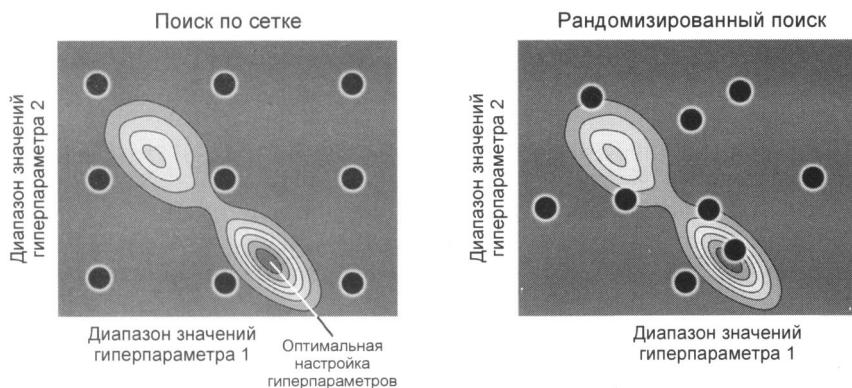


Рис. 6.7. Сравнение поиска по сетке и рандомизированного поиска на примере девяти выборок различных конфигураций гиперпараметров

³ Занинтересованные читатели могут найти дополнительные сведения о рандомизированном поиске, а также данные эмпирических исследований в следующей статье: «Random Search for Hyper-Parameter Optimization» by J. Bergstra, Y. Bengio, Journal of Machine Learning Research, p. 281–305, 2012, <https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf>.

Рассмотрим использование рандомизированного поиска для настройки SVM. Библиотека scikit-learn реализует класс `RandomizedSearchCV`, аналогичный классу `GridSearchCV`, который мы использовали в предыдущем разделе. Основное различие состоит в том, что при рандомизированном поиске мы можем указать распределения вероятностей как часть нашей сетки параметров и задать общее количество оцениваемых конфигураций гиперпараметров. Например, возьмем диапазон значений, который мы использовали для нескольких гиперпараметров при настройке SVM в примере поиска по сетке в предыдущем разделе:

```
>>> import scipy.stats
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                 1.0, 10.0, 100.0, 1000.0]
```

Хотя `RandomizedSearchCV` может принимать аналогичные дискретные списки значений в качестве входных данных для сетки параметров, что полезно при рассмотрении категориальных гиперпараметров, его основное преимущество заключается в том, что список значений можно заменить распределением для выборки. Так, например, мы можем заменить предыдущий список следующим распределением из SciPy:

```
>>> param_range = scipy.stats.loguniform(0.0001, 1000.0)
```

Использование логарифмического распределения вместо обычного равномерного распределения гарантирует, что при достаточно большом количестве испытаний из диапазона $[0.0001, 0.001]$ будет взято такое же количество выборок, как, например, из диапазона $[10.0, 100.0]$. Для проверки возьмем 10 случайных выборок из этого распределения с помощью метода `rvs(10)`:

```
>>> np.random.seed(1)
>>> param_range.rvs(10)
array([8.30145146e-02, 1.10222804e+01, 1.00184520e-04, 1.30715777e-02,
       1.06485687e-03, 4.42965766e-04, 2.01289666e-03, 2.62376594e-02,
       5.98924832e-02, 5.91176467e-01])
```



Указание распределений

`RandomizedSearchCV` поддерживает любые распределения, позволяющие делать выборку из них при помощи вызова метода `rvs()`. Список всех распределений, доступных в настоящее время через `scipy.stats`, можно найти по адресу: <https://docs.scipy.org/doc/scipy/reference/stats.html#probability-distributions>.

Теперь испытаем `RandomizedSearchCV` в действии и настроим SVM, как мы это делали с `GridSearchCV` в предыдущем разделе:

```
>>> from sklearn.model_selection import RandomizedSearchCV
>>> pipe_svc = make_pipeline(StandardScaler(),
...                           SVC(random_state=1))
...
>>> param_grid = [ {'svc_C': param_range,
...                  'svc_kernel': ['linear']},
...                 {'svc_C': param_range,
...                  'svc_gamma': param_range,
...                  'svc_kernel': ['rbf']} ]
```

```
>>> rs = RandomizedSearchCV(estimator=pipe_svc,
...                         param_distributions=param_grid,
...                         scoring='accuracy',
...                         refit=True,
...                         n_iter=20,
...                         cv=10,
...                         random_state=1,
...                         n_jobs=-1)
>>> rs.fit(X_train, y_train)
>>> print(rs.best_score_)
0.9670531400966184

>>> print(rs.best_params_)
{'svc__C': 0.05971247755848464, 'svc__kernel': 'linear'}
```

Глядя на этот пример кода, несложно заметить, что использование `RandomizedSearchCV` очень похоже на `GridSearchCV`, за исключением того, что мы можем задействовать распределения в качестве диапазонов параметров и указать количество итераций (в нашем случае — 20), установив `n_iter=20`.

6.4.3. Поиск гиперпараметров методом последовательного деления пополам

Развивая идею рандомизированного поиска, scikit-learn предлагает реализацию метода последовательного деления пополам — класс `HalvingRandomSearchCV`, который делает поиск подходящих конфигураций гиперпараметров еще более ресурсоэффективным. Алгоритм последовательного деления пополам при наличии большого набора конфигураций-кандидатов последовательно отбрасывает менее перспективные конфигурации гиперпараметров до тех пор, пока не останется только одна лучшая конфигурация. В общем виде эта несложная процедура состоит из таких шагов:

1. Формируем большой набор конфигураций-кандидатов с помощью случайной выборки.
2. Обучаем модели с применением ограниченных ресурсов — например, на небольшом подмножестве обучающих данных (в отличие от использования всего обучающего набора).
3. Отбрасываем нижние 50% моделей, исходя из их прогнозируемой производительности.
4. Возвращаемся к шагу 2 с увеличенным количеством ресурсов.

Эти шаги повторяются до тех пор, пока не останется только одна конфигурация гиперпараметров. Существует также реализация последовательного деления пополам для поиска по сетке под названием `HalvingGridSearchCV`, где на *шаге 1* вместо случайных выборок используются все доступные конфигурации гиперпараметров.

В scikit-learn 1.0 класс `HalvingRandomSearchCV` все еще является экспериментальным и по умолчанию отключен, поэтому мы должны сначала включить его⁴:

```
>>> from sklearn.experimental import enable_halving_search_cv
```

⁴ Этот код может не работать или не поддерживаться в будущих версиях библиотеки.

После включения поддержки экспериментальных классов мы получаем возможность использовать рандомизированный поиск с последовательным делением пополам, как показано в следующем примере кода:

```
>>> from sklearn.model_selection import HalvingRandomSearchCV
>>> hs = HalvingRandomSearchCV(pipe_svc,
...                                param_distributions=param_grid,
...                                n_candidates='exhaust',
...                                resource='n_samples',
...                                factor=1.5,
...                                random_state=1,
...                                n_jobs=-1)
```

Параметр `resource='n_samples'` (по умолчанию) указывает, что мы рассматриваем размер обучающего набора как ресурс, который необходимо варьировать между раундами отбора. С помощью параметра `factor` мы указываем, сколько кандидатов отсеивается в каждом раунде. Например, параметр `factor=2` исключает половину кандидатов, а `factor=1.5` означает, что только $100\% / 1.5 \approx 66\%$ кандидатов проходят в следующий раунд. Вместо выбора фиксированного количества итераций, как в `RandomizedSearchCV`, мы устанавливаем `n_candidates='exhaust'` (по умолчанию). В соответствии с последним параметром метод будет выбирать количество конфигураций гиперпараметров таким образом, чтобы максимальное количество ресурсов (количество записей обучающего набора) использовалось в последнем раунде.

Теперь выполним поиск по аналогии с `RandomizedSearchCV`:

```
>>> hs = hs.fit(X_train, y_train)
>>> print(hs.best_score_)
0.9617647058823529

>>> print(hs.best_params_)
{'svc_C': 4.934834261073341, 'svc_kernel': 'linear'}
>>> clf = hs.best_estimator_
>>> print(f'Точность при тестировании: {hs.score(X_test, y_test):.3f}')
Точность при тестировании: 0.982
```

Если мы сравним результаты методов `GridSearchCV` и `RandomizedSearchCV` из предыдущих двух разделов с методом `HalvingRandomSearchCV`, то увидим, что последний дает модель, которая работает немного лучше на тестовом наборе (точность 98.2 % по сравнению с 97.4 %).



Настройка гиперпараметров с помощью hyperopt

Еще одним популярным инструментом оптимизации гиперпараметров является библиотека `hyperopt` (<https://github.com/hyperopt/hyperopt>), поддерживающая несколько методов оптимизации гиперпараметров, включая рандомизированный поиск и метод [древовидных оценщиков Парзена](#) (Tree-structured Parzen Estimator, TPE). TPE — это байесовский метод оптимизации, основанный на вероятностной модели, которая постоянно обновляется на основе предыдущих оценок гиперпараметров и связанных показателей производительности, вместо того, чтобы рассматривать эти оценки как независимые события. Вы можете узнать больше о TPE в статье «[Algorithms for Hyper-Parameter Optimization](#)», Bergstra J., Bardenet R., Bengio Y., Kegl B. NeurIPS 2011. p. 2546–2554, <https://dl.acm.org/doi/10.5555/2986459.2986743>.

Хотя библиотека hyperopt оснащена интерфейсом общего назначения для оптимизации гиперпараметров, для дополнительного удобства пользователей разработан специальный пакет scikit-learn под названием hyperopt-sklearn, доступный по адресу: <https://github.com/hyperopt/hyperopt-sklearn>.

6.4.4. Выбор алгоритма методом вложенной перекрестной проверки

Использование k -кратной перекрестной проверки в сочетании с поиском по сетке или рандомизированным поиском является хорошим способом точной настройки производительности модели путем изменения значений ее гиперпараметров, как было показано в предыдущих подразделах. Однако, если вы не ограничиваетесь подбором гиперпараметров и хотите выбирать среди различных алгоритмов машинного обучения, рекомендуется использовать *вложенную перекрестную проверку* (nested cross-validation). В своем впечатляющем исследовании систематического смещения при оценке ошибок Судхир Варма и Ричард Саймон пришли к выводу, что при использовании вложенной перекрестной проверки истинная ошибка оценки почти не смешена по сравнению с тестовым набором данных⁵.

При вложенной перекрестной проверке у нас есть внешний цикл k -кратной перекрестной проверки для разделения данных на обучающую и тестовую выборки, а внутренний цикл служит для выбора модели с помощью k -кратной перекрестной проверки на обучающей выборке. Когда модель выбрана, тестовая выборка используется для оценки производительности модели. Рис. 6.8 поясняет идею вложенной перекрестной проверки с пятью внешними и двумя внутренними выборками. Такая схема разбиения удобна при использовании больших наборов данных, где важна производительность вычислений (этот конкретный вариант вложенной перекрестной проверки также называется *перекрестной проверкой 5×2*).

В scikit-learn вложенная перекрестная проверка с поиском по сетке выполняется следующим образом:

```
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                 1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{‘svc_C’: param_range,
...                  ‘svc_kernel’: [‘linear’]},
...                  {‘svc_C’: param_range,
...                  ‘svc_gamma’: param_range,
...                  ‘svc_kernel’: [‘rbf’]}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                      param_grid=param_grid,
...                      scoring=‘accuracy’,
...                      cv=2)
>>> scores = cross_val_score(gs, X_train, y_train,
...                           scoring=‘accuracy’, cv=5)
>>> print(f‘Точность перекр. проверки: {np.mean(scores):.3f} ’
...           f’+/- {np.std(scores):.3f}’)
Точность перекр. проверки: 0.974 +/- 0.015
```

⁵ См.: «Bias in Error Estimation When Using Cross-Validation for Model Selection» by S. Varma and R. Simon, BMC Bioinformatics, 7(1): 91, 2006. <https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-91>.

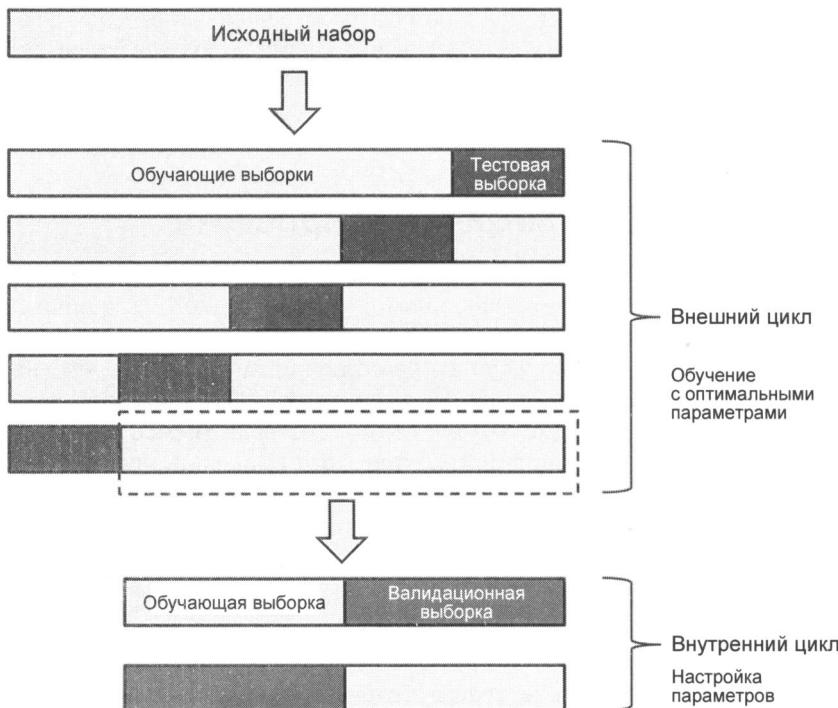


Рис. 6.8. Иллюстрация идеи вложенной перекрестной проверки

Возвращаемая средняя точность перекрестной проверки достаточно достоверно говорит нам о том, чего ожидать, если мы настроим гиперпараметры модели и подадим на ее вход незнакомые данные.

В качестве примера воспользуемся вложенной перекрестной проверкой для сравнения модели SVM с простым классификатором на основе дерева решений (для наглядности мы настроим только его параметр глубины):

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...     estimator=DecisionTreeClassifier(random_state=0),
...     param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...     scoring='accuracy',
...     cv=2
... )
>>> scores = cross_val_score(gs, X_train, y_train,
...                           scoring='accuracy', cv=5)
>>> print(f' Точность перекр. проверки: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
Точность перекр. проверки: 0.934 +/-
 0.016
```

Как видите, в результате вложенной перекрестной проверки выяснилось, что производительность модели SVM (97.4%) заметно лучше, чем у дерева решений (93.4%), поэтому мы ожидаем, что она окажется лучшим выбором для классификации новых данных, которые поступают из той же совокупности, что и этот конкретный набор.

6.5. Обзор различных показателей оценки эффективности

В предыдущих разделах и главах мы оценивали различные модели машинного обучения, используя точность прогнозирования (prediction accuracy), которая, несомненно, является полезной метрикой для количественной оценки производительности модели в целом. Однако есть и другие показатели производительности, которые можно использовать для измерения релевантности модели — такие как точность (precision), полнота, оценка F1 и коэффициент корреляции Мэттьюза (Matthews Correlation Coefficient, MCC).

6.5.1. Чтение матрицы несоответствий

Прежде чем вникать в детали различных метрик оценки, нужно разобраться с понятием *матрицы несоответствий* (confusion matrix), которая определяет производительность алгоритма обучения.

Матрица несоответствий — это просто квадратная матрица, которая сообщает количество *истинно положительных* (True Positive, TP), *истинно отрицательных* (True Negative, TN), *ложноположительных* (False Positive, FP) и *ложноотрицательных* (False Negative, FN) предсказаний классификатора (рис. 6.9).

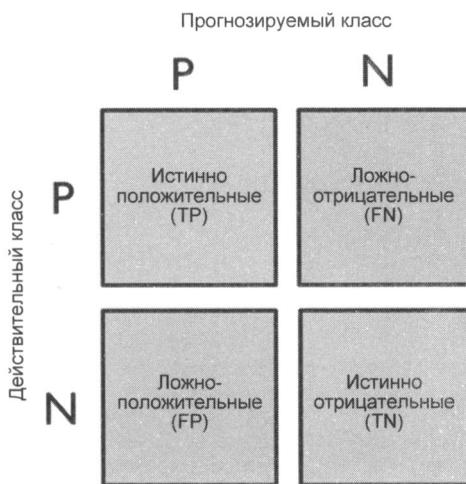


Рис. 6.9. Матрица несоответствий

Хотя эти показатели не составят труда вычислить вручную, сравнивая фактические и предсказанные метки классов, scikit-learn предоставляет удобную функцию `confusion_matrix`, которую можно использовать следующим образом:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
```

```
>>> print(confmat)
[[71 1]
 [ 2 40]]
```

Массив, возвращенный после выполнения кода, предоставляет нам информацию о различных типах ошибок, допущенных классификатором в тестовом наборе данных. Мы можем отобразить эту информацию на схеме матрицы несоответствий, показанной на рис. 6.9, используя функцию `matshow` Matplotlib:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i, s=confmat[i, j],
...                 va='center', ha='center')
>>> ax.xaxis.set_ticks_position('bottom')
>>> plt.xlabel('Предсказанная метка')
>>> plt.ylabel('Истинная метка')
>>> plt.show()
```

Выполнив этот код, вы должны получить матрицу несоответствий, которая облегчает интерпретацию результатов (рис. 6.10). В ячейках матрицы указано соответствующее количество меток.

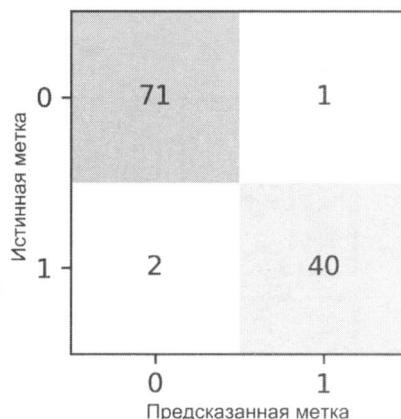


Рис. 6.10. Матрица несоответствий для наших текущих данных

Если считать, что в приведенном примере класс 1 (злокачественная опухоль) является положительным классом, наша модель правильно классифицировала 71 экземпляр, принадлежащий к классу 0 (TN), и 40 экземпляров, принадлежащих к классу 1 (TP) соответственно. Однако наша модель также неправильно классифицировала два примера из класса 1 как класс 0 (FN) и предсказала, что один экземпляр является злокачественной опухолью, хотя на самом деле она доброкачественная (FP). Далее вы узнаете, как использовать эту информацию для расчета различных метрик ошибок.

6.5.2. Оптимизация правильности и полноты модели классификации

Как ошибка прогнозирования (prediction error, ERR), так и *правильность*⁶ (accuracy, ACC) предоставляют общую информацию о том, сколько экземпляров из набора данных было классифицировано неправильно. Под ошибкой понимают сумму всех ложных прогнозов, деленную на общее количество прогнозов, а правильность вычисляют как сумму правильных прогнозов, деленную на общее количество прогнозов соответственно:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}.$$

Кроме того, правильность прогноза можно рассчитать непосредственно из ошибки:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR.$$

Доля истинно положительных случаев (True Positive Rate, TPR) и доля ложноположительных случаев (False Positive Rate, FPR) — это показатели производительности, которые особенно полезны при решении задач с несбалансированными классами:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}.$$

Например, при диагностике опухолей нас весьма заботит обнаружение злокачественных опухолей, чтобы вовремя предоставить пациенту нужное лечение. Однако также важно уменьшить количество обнаруженных доброкачественных опухолей, неправильно классифицированных как злокачественные (FP), чтобы не беспокоить пациентов без необходимости. В отличие от FPR , метрика TPR предоставляет полезную информацию о доле положительных (или релевантных) экземпляров, которые были правильно идентифицированы среди общего множества положительных результатов (P).

Показатели производительности *точность* (precision, PRE) и *полнота* (recall, REC) непосредственно связаны с долями TP и TN , и фактически REC является синонимом TPR :

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}.$$

Другими словами, полнота результата количественно определяет, сколько релевантных записей (положительных) распознано правильно (доля истинно положительных). Точность же количественно определяет, сколько записей, предсказанных как релевантные (сумма истинных и ложных срабатываний), оказались релевантны на самом деле (истинные срабатывания):

$$PRE = \frac{TP}{TP + FP}.$$

⁶ В этом контексте мы будем переводить *accuracy* — как *правильность* (классификации), а *precision* — как *точность*. — Прим. пер.

Вернемся к примеру обнаружения злокачественной опухоли. Оптимизация по критерию полноты помогает свести к минимуму вероятность того, что мы пропустим злокачественную опухоль. Однако это происходит за счет прогнозирования злокачественных опухолей у пациентов, которые здоровы (большое количество ложноположительных выводов). С другой стороны, оптимизируя точность, мы тем самым усиливаем правильность предсказаний злокачественной опухоли. Однако это происходит за счет увеличения доли злокачественных опухолей, которые мы не заметили (большое количество ложноотрицательных выводов).

Чтобы найти компромисс между положительными и отрицательными сторонами оптимизации PRE и REC, используют *среднее гармоническое* PRE и REC, или так называемую оценку *F1*:

$$F1 = 2 \frac{PRE \times REC}{PRE + REC}.$$



Дополнительные источники информации о точности и полноте

Если вы стремитесь более подробно разобраться с различными показателями производительности, такими как точность и полнота, прочитайте технический отчет Дэвида Пауэрса «Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation», который находится в свободном доступе по адресу: <https://arxiv.org/abs/2010.16061>.

Наконец, мерой, обобщающей матрицу несоответствий, является коэффициент корреляции Мэттьюза (Matthews Correlation Coefficient, MCC), который особенно популярен в контексте биологических исследований. MCC рассчитывают следующим образом:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

В отличие от PRE, REC и оценки F1, MCC находится в диапазоне от -1 до 1 и учитывает все элементы матрицы несоответствий (в отличие, например, от оценки F1, которая не включает TN). Хотя значения MCC сложнее интерпретировать, чем показатель F1, коэффициент корреляции Мэттьюза считается лучшим показателем⁷.

Все упомянутые здесь метрики оценки реализованы в scikit-learn и могут быть импортированы из модуля sklearn.metrics, как показано в следующем фрагменте кода:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score
>>> from sklearn.metrics import matthews_corrcoef

>>> pre_val = precision_score(y_true=y_test, y_pred=y_pred)
>>> print(f'Точность: {pre_val:.3f}')
Точность: 0.976
>>> rec_val = recall_score(y_true=y_test, y_pred=y_pred)
>>> print(f'Полнота: {rec_val:.3f}')
Полнота: 0.952
```

⁷ Так отмечено в статье: «The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation», D. Chicco and G. Jurman, BMC Genomics. p. 281–305, 2012, <https://bmcbioinformatics.biomedcentral.com/articles/10.1186/s12864-019-6413-7>.

```
>>> f1_val = f1_score(y_true=y_test, y_pred=y_pred)
>>> print(f'F1: {f1_val:.3f}')
F1: 0.964
>>> mcc_val = matthews_corrcoef(y_true=y_test, y_pred=y_pred)
>>> print(f'MCC: {mcc_val:.3f}')
MCC: 0.943
```

Кроме того, используя параметр `scoring`, мы можем задействовать в `GridSearchCV` вместо `accuracy` и другую метрику. Полный список различных значений, принимаемых параметром `scoring`, можно найти по адресу: http://scikit-learn.org/stable/modules/model_evaluation.html.

Помните, что положительный класс в scikit-learn — это класс, помеченный как 1. Если вам захотелось указать другую *положительную метку*, создайте собственный оцениватель с помощью функции `make_scorer`, которую можно затем напрямую предоставить в качестве аргумента для параметра `scoring` в `GridSearchCV` (в этом примере мы используем в качестве метрики `f1_score`):

```
>>> from sklearn.metrics import make_scorer
>>> c_gamma_range = [0.01, 0.1, 1.0, 10.0]
>>> param_grid = [{ 'svc_C': c_gamma_range,
...                 'svc_kernel': ['linear']},
...                 { 'svc_C': c_gamma_range,
...                 'svc_gamma': c_gamma_range,
...                 'svc_kernel': ['rbf']}]
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                     param_grid=param_grid,
...                     scoring=scorer,
...                     cv=10)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.986202145696
>>> print(gs.best_params_)
{'svc_C': 10.0, 'svc_gamma': 0.01, 'svc_kernel': 'rbf'}
```

6.5.3. Построение рабочей характеристики приемника

Полезным средством выбора классифицирующих моделей на основе их характеристик по отношению к FPR и TPR является график *характеристической кривой обнаружения* (Receiver Operating Characteristic, ROC), вычисляемый путем сдвига порога принятия решения классификатора. Диагональную линию ROC можно интерпретировать как случайное угадывание, а классифицирующие модели, расположенные ниже этой диагонали, считаются худшими, чем случайное угадывание. Идеальный классификатор попал бы в верхний левый угол графика с показателем TPR, равным 1, и FPR, равным 0. На основе кривой ROC мы можем затем вычислить так называемую *площадь ROC под кривой* (ROC Area Under the Curve, ROC AUC), чтобы охарактеризовать производительность классифицирующей модели.

Так же как и кривые ROC, мы можем вычислять *кривые точности-полноты* (precision-recall curve) для различных порогов вероятности классификатора. Функция для построения этих кривых точности-полноты также реализована в scikit-learn⁸.

Выполнив следующий пример кода, мы построим ROC-кривую классификатора, который использует только два признака из набора данных о раке молочной железы в Висконсине, чтобы предсказать, является ли опухоль доброкачественной или злокачественной. Хотя мы собираемся задействовать тот же конвейер логистической регрессии, который определили ранее, на этот раз мы ограничимся лишь двумя признаками. Это сделано, чтобы усложнить задачу для классификатора: скрывая полезную информацию, содержащуюся в других признаках, мы получим более визуально интересную результатирующую кривую ROC. По тем же причинам мы также сокращаем количество выборок в валидаторе StratifiedKFold до трех:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from numpy import interp
>>> pipe_lr = make_pipeline(
...     StandardScaler(),
...     PCA(n_components=2),
...     LogisticRegression(penalty='l2', random_state=1,
...                         solver='lbfgs', C=100.0)
... )
>>> X_train2 = X_train[:, [4, 14]]
>>> cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []
>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(
...         X_train2[train],
...         y_train[train]
...     ).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...               tpr,
...               label=f'Выборка ROC {i+1} (area = {roc_auc:.2f})')
>>> plt.plot([0, 1],
...           [0, 1],
...           linestyle='--',
...           color=(0.6, 0.6, 0.6),
...           label='Случайный выбор (area=0.5)')
```

⁸ Соответствующая документация доступна по адресу:

http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html

```

>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...             label=f'Средн. ROC (area = {mean_auc:.2f})', lw=2)
>>> plt.plot([0, 0, 1],
...           [0, 1, 1],
...           linestyle=':',
...           color='black',
...           label='Идеальная производительность (area=1.0)')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('Доля ложноположительных прогнозов')
>>> plt.ylabel('Доля истинно положительных прогнозов')
>>> plt.legend(loc='lower right')
>>> plt.show()

```

В этом примере кода мы использовали уже знакомый нам класс `StratifiedKFold` из `scikit-learn` и рассчитывали кривую ROC классификатора `LogisticRegression` в нашем конвейере `pipe_lr` с помощью функции `roc_curve` из модуля `sklearn.metrics` отдельно для каждой итерации. Кроме того, мы интерполировали среднюю кривую ROC из трех выборок с помощью функции `interp`, которую мы импортировали из `NumPy`, и рассчитали площадь под кривой с помощью функции `auc`. Полученная кривая ROC указывает на то, что существует определенная степень дисперсии между различными выборками, а среднее значение ROC AUC (0.76) находится между идеальным результатом (1.0) и случайным предположением (0.5).

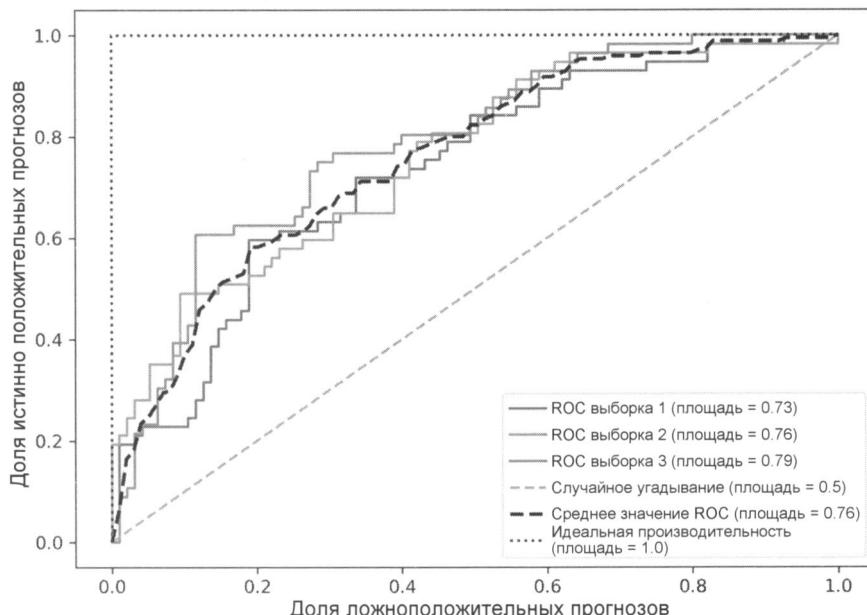


Рис. 6.11. График ROC

Отметим, что если вас интересует только оценка ROC AUC, вы также можете напрямую импортировать из подмодуля `sklearn.metrics` функцию `roc_auc_score`, которую можно использовать аналогично другим функциям оценки (например, `precision_score`), с которыми вы познакомились в предыдущих разделах.

Отчет в форме ROC AUC может дать дополнительное представление о производительности классификатора в отношении несбалансированных выборок. Однако, хотя показатель правильности можно интерпретировать как единую точку отсечки на кривой ROC, А. Брэдли показал, что показатели ROC AUC и правильности в основном соглашаются друг с другом⁹.

6.5.4. Метрики оценки многоклассовой классификации

Метрики, которые мы обсуждали до сих пор, относятся к системам бинарной классификации. Но `scikit-learn` также содержит методы макро- и микроусреднения, позволяющие распространить эти метрики оценки на многоклассовые задачи с помощью классификации типа «один против всех» (One-vs-All, OvA). *Микроусреднение* (micro-average) рассчитывается по отдельным TP, TN, FP и FN системы. Например, микроусредненное значение показателя точности в k -классовой системе можно рассчитать следующим образом:

$$PRE_{micro} = \frac{TP_1 + \dots + TP_k}{TP_1 + \dots + TP_k + FP_1 + \dots + FP_k}.$$

Макроусреднение (macro-average) вычисляют просто как среднее оценки различных систем:

$$PRE_{macro} = \frac{PRE_1 + \dots + PRE_k}{k}.$$

Микроусреднение полезно, если мы хотим одинаково взвесить каждый экземпляр или прогноз, тогда как макроусреднение одинаково взвешивает все классы, чтобы оценить общую производительность классификатора в отношении наиболее часто встречающихся меток классов.

Если мы задействуем бинарные метрики производительности для оценки моделей многоклассовой классификации в `scikit-learn`, то по умолчанию используется нормализованный или взвешенный вариант макроусреднения. Взвешенное макросреднее вычисляется путем взвешивания оценки каждой метки класса по количеству истинных экземпляров при вычислении среднего. Взвешенное макроусреднение полезно, если мы имеем дело с дисбалансом классов, т. е. с разным количеством экземпляров для каждой метки.

Хотя в `scikit-learn` средневзвешенное макроусреднение для многоклассовых задач используется по умолчанию, мы можем явно указать метод усреднения через параметр `average` внутри различных функций оценки, которые импортируем из модуля `sklearn.metrics`. Вот пример для функций `precision_score` ИЛИ `make_scorer`:

⁹ См.: «The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms» by A. P. Bradley, Pattern Recognition, 30(7): 1145–1159, 1997, <https://reader.elsevier.com/reader/sd/pii/S0031320396001422>.

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                             pos_label=1,
...                             greater_is_better=True,
...                             average='micro')
```

6.5.5. Борьба с дисбалансом классов

В этой главе мы несколько раз упоминали дисбаланс классов, но пока не обсуждали, как правильно вести себя в таких случаях. Дисбаланс классов — довольно распространенная проблема при работе с реальными данными. Это ситуация, когда экземпляры из одного или нескольких классов представлены в наборе данных слишком часто. Можно навскидку назвать несколько областей, в которых дисбаланс классов не редкость — например, фильтрация спама, обнаружение мошенничества или проверка на наличие болезней.

Допустим, набор данных по раку молочной железы в Висконсине, с которым мы работали в этой главе, на 90 % состоит из здоровых пациентов. В этом случае можно достичь точности 90 % на тестовом наборе данных, просто постоянно предсказывая преобладающий класс (добропачественная опухоль) для всех экземпляров, даже не прибегая к помощи алгоритма машинного обучения. Следовательно, обучение на наборе данных, обеспечивающем примерно 90-процентную точность безо всяких усилий со стороны модели, будет означать, что она не извлекла ничего полезного из признаков, представленных в анализируемом наборе данных.

В этом разделе мы кратко рассмотрим некоторые методы, помогающие справиться с несбалансированными наборами данных. Но прежде чем приступить к обсуждению различных методов решения этой проблемы, создадим несбалансированный набор данных из нашего набора данных, который изначально состоял из 357 доброкачественных опухолей (класс 0) и 212 злокачественных опухолей (класс 1):

```
>>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

В этом фрагменте кода мы взяли все 357 записей о доброкачественных опухолях и объединили их с первыми 40 записями о злокачественных опухолях, чтобы создать резкий дисбаланс классов. Если мы оценим точность модели, которая всегда предсказывает преобладающий класс (добропачественная опухоль, класс 0), то увидим, что она достигла точности прогнозирования примерно в 90 % случаев:

```
>>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.92443324937027
```

Следовательно, когда мы подбираем классификаторы для «перекошенных» наборов данных, при сравнении моделей имеет смысл сосредоточиться не на правильности, а на других показателях — таких как точность, полнота, кривая ROC, — короче говоря, на том, что нас больше всего волнует в текущем сценарии. Например, нашим приоритетом является выявление большинства пациентов с подозрением на злокачественную опухоль, чтобы рекомендовать дополнительный скрининг, поэтому предпочтительным показателем должна быть полнота результата (*recall*). В приложении электронной почты, где не следует помечать электронные письма как спам, если система не уверена, приоритетной метрикой может быть точность (*precision*).

Помимо оценки готовых моделей, дисбаланс классов влияет непосредственно на процесс машинного обучения. Поскольку алгоритмы машинного обучения обычно оптимизируют функцию вознаграждения или убытка, которая вычисляется как сумма по обучающим экземплярам, которые они видят во время обучения, правило принятия решения, вероятно, будет смещено в сторону доминирующего класса.

Другими словами, алгоритм неявно обучает модель, которая оптимизирует прогнозы на основе наиболее распространенного класса в наборе данных, чтобы минимизировать потери или максимизировать вознаграждение во время обучения.

Один из способов справиться с несбалансированными пропорциями классов во время обучения модели — назначить больший штраф за неправильные прогнозы для миноритарного класса. С помощью scikit-learn настроить такой штраф не составит труда — достаточно установить значение параметра `class_weight='balanced'`, который поддерживает большинство классификаторов.

К другим популярным стратегиям борьбы с дисбалансом классов относятся повышение дискретизации миноритарного класса, понижение дискретизации доминирующего класса и создание синтетических обучающих примеров. К сожалению, не существует универсально хорошего решения или метода, который бы лучше всего работал в различных предметных областях. Поэтому на практике рекомендуется опробовать различные стратегии для решения этой проблемы, оценить результаты и выбрать наиболее подходящий способ.

Библиотека scikit-learn содержит простую функцию `resample`, которая помогает увеличить дискретизацию миноритарного класса путем создания новых выборок из набора данных с возвращением. Следующий код возьмет миноритарный класс из нашего несбалансированного набора данных по раку молочной железы в Висконсине (здесь класс 1) и станет многократно извлекать из него новые экземпляры, пока он не будет содержать то же количество записей, что и класс 0:

```
>>> from sklearn.utils import resample
>>> print('Старое количество экземпляров класса 1:',
...       X_imb[y_imb == 1].shape[0])
Старое количество экземпляров класса 1: 40
>>> X_upsampled, y_upsampled = resample(
...       X_imb[y_imb == 1],
...       y_imb[y_imb == 1],
...       replace=True,
...       n_samples=X_imb[y_imb == 0].shape[0],
...       random_state=123)
>>> print('Новое количество экземпляров класса 1:',
...       X_upsampled.shape[0])
Новое количество экземпляров класса 1: 357
```

Выполнив процедуру повторной выборки, сложим исходные выборки класса 0 с подмножеством класса 1 после повышения дискретизации, чтобы получить сбалансированный набор данных следующим образом:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Как можно видеть, прогнозирование по принципу большинства голосов будет достигать только 50-процентной точности:

```
>>> y_pred = np.zeros(y_bal.shape[0])
>>> np.mean(y_pred == y_bal) * 100
50
```

Точно так же мы могли бы понизить дискретизацию доминирующего набора, удалив обучающие примеры из набора данных. Чтобы понизить дискретизацию с помощью функции `resample`, надо было бы просто поменять местами метку класса 1 с классом 0 в предыдущем примере кода и наоборот.



Генерация новых обучающих данных для устранения дисбаланса классов

Описание другого метода борьбы с дисбалансом классов — создания синтетических обучающих данных — выходит за рамки этой книги. Вероятно, наиболее широко используемым алгоритмом для создания синтетических обучающих данных является метод передискретизации синтетического меньшинства (*Synthetic Minority Over-sampling Technique*, SMOTE). Заинтересованные читатели могут узнать больше об этом методе в исследовательской статье Нитеша Чавла и др. «SMOTE: Synthetic Minority Over-sampling Technique», *Journal of Artificial Intelligence Research*, 16: 321–357, 2002, которая доступна по адресу: <https://www.jair.org/index.php/jair/article/view/10302>. Также настоятельно рекомендуется ознакомиться с библиотекой Python `imbalanced-learn`, которая полностью ориентирована на несбалансированные наборы данных, включая реализацию SMOTE. Соответствующую документацию вы можете скачать по адресу: <https://github.com/scikit-learn-contrib/imbalanced-learn>.

6.6. Заключение

В начале этой главы мы обсудили, как объединить различные методы преобразования и классификаторы в удобные конвейеры, способные помочь нам более эффективно обучать и оценивать модели машинного обучения. Затем мы применили эти конвейеры для выполнения k -кратной перекрестной проверки — одного из основных методов выбора и оценки модели. Используя k -кратную перекрестную проверку, мы построили кривые обучения и проверки для диагностики общих проблем алгоритмов обучения, таких как переобучение и недообучение.

Применяя поиск по сетке, рандомизированный поиск и последовательное деление пополам, мы еще больше усовершенствовали нашу модель. Затем мы использовали матрицы несоответствий и различные показатели производительности для оценки и оптимизации производительности модели при решении конкретных задач.

Завершает главу обсуждение различных методов работы с несбалансированными данными, что является распространенной проблемой во многих реальных ситуациях. Теперь вы должны хорошо владеть необходимыми методами для успешного построения классифицирующих моделей машинного обучения с учителем.

В следующей главе мы рассмотрим ансамблевые методы, позволяющие нам комбинировать несколько моделей и алгоритмов классификации, чтобы еще больше повысить прогностическую точность системы машинного обучения.

7

Объединение различных моделей для ансамблевого обучения

В предыдущей главе мы сосредоточились на передовых методах настройки и оценки различных моделей для задач классификации. В этой главе мы воспользуемся новыми знаниями и исследуем различные способы построения наборов классификаторов, которые часто могут иметь лучшую прогностическую способность, чем любой из его отдельных членов. В этой главе вы научитесь:

- ◆ делать прогнозы на основе большинства голосов;
- ◆ использовать бэггинг, чтобы уменьшить риск переобучения путем построения случайных комбинаций обучающего набора данных с повторением;
- ◆ применять бустинг для создания мощных моделей на основе слабых учеников, которые учатся на своих ошибках.

7.1. Обучение ансамблей моделей

Идея *ансамблевых методов* (*ensemble method*) состоит в том, чтобы объединить различные классификаторы в метаклассификатор, имеющий лучшую обобщающую способность, чем каждый классификатор в отдельности. Например, если предположить, что мы собрали прогнозы от десяти экспертов, ансамблевые методы позволят нам стратегически объединить все десять прогнозов, чтобы получить более точный и надежный прогноз, чем прогнозы каждого отдельного эксперта. Как вы увидите далее в этой главе, существует несколько разных подходов к созданию ансамбля классификаторов. В этом разделе вы получите краткое объяснение принципов работы ансамблей и узнаете, почему их ценят за хорошие результаты обобщения.

В этой главе мы ограничимся изучением самых популярных ансамблевых методов, использующих принцип *мажоритарного голосования* (*majority voting*, голосование абсолютным большинством). Мажоритарное голосование означает, что мы выбираем метку класса, которая была предсказана большинством классификаторов, т. е. получила более 50% голосов. Строго говоря, термин «мажоритарное голосование» относится только к бинарной классификации. Однако принцип мажоритарного голосования легко обобщается на многоклассовую систему, что известно как *голосование относительным большинством* (*plurality voting*)¹.

¹ Понятия «абсолютного» и «относительного» большинства позаимствованы из британской избирательной системы.

Сначала мы должны каким-то образом выбрать метку класса, получившую наибольшее количество голосов, — так называемую *моду*². На рис. 7.1 показаны принципы голосования по абсолютному и относительному большинству голосов для ансамбля из 10 классификаторов, где каждый уникальный символ (треугольник, квадрат и круг) представляет уникальную метку класса.

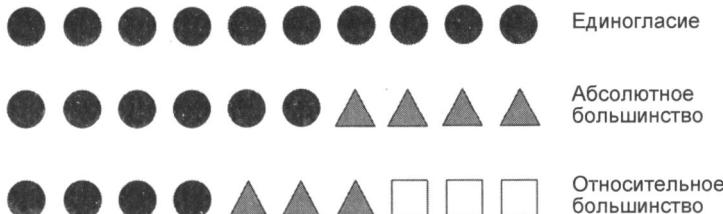


Рис. 7.1. Различные принципы голосования

Используя обучающий набор данных, мы начинаем с обучения m различных классификаторов (C_1, \dots, C_m). В зависимости от применяемого стратегического подхода, ансамбль может быть построен из различных алгоритмов классификации — например, деревьев решений, опорных векторов, логистической регрессии и т. д. В качестве альтернативы можно использовать один и тот же алгоритм базовой классификации, но обучить несколько моделей на различных подмножествах обучающего набора. Одним из ярких примеров такого подхода является алгоритм случайного леса, объединяющий различные классификаторы на основе деревьев решений, которые мы рассмотрели в главе 3. На рис. 7.2 показана схема обобщенного ансамблевого метода с использованием мажоритарного голосования:

Чтобы предсказать метку класса с помощью абсолютного или относительного большинства, мы можем объединить предсказанные метки классов каждого отдельного классификатора C_j и выбрать метку класса \hat{y} , получившую наибольшее количество голосов (моду):

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}.$$

Например, в задаче бинарной классификации, где $\text{class1} = -1$ и $\text{class2} = +1$, мы можем записать мажоритарный прогноз следующим образом:

$$C(\mathbf{x}) = \text{sign} \left[\sum_j^m C_j(\mathbf{x}) \right] = \begin{cases} 1 & \text{if } \sum_j C_j(\mathbf{x}) \geq 0 \\ -1 & \text{в ином случае} \end{cases}.$$

Чтобы понять, почему ансамблевые методы могут работать лучше, чем отдельные классификаторы, нужно воспользоваться на практике некоторыми принципами комбинаторики. В следующем примере мы сделаем предположение, что все n -базовые классификаторы для задачи бинарной классификации имеют одинаковую частоту ошибок ε . Кроме того, мы также предположим, что классификаторы независимы, а коэффициенты ошибок не коррелированы. Исходя из этих предположений, мы можем выразить вероятность ошибки ансамбля базовых классификаторов как функцию массы вероятности биномиального распределения:

² В статистике *мода* — это наиболее частое событие или элемент множества. К примеру: $\text{mode}\{1, 2, 1, 1, 2, 4, 5, 4\} = 1$.

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1 - \varepsilon)^{n-k} = \varepsilon_{\text{ensemble}},$$

где $\binom{n}{k}$ — биномиальный коэффициент для выбора из n по k .

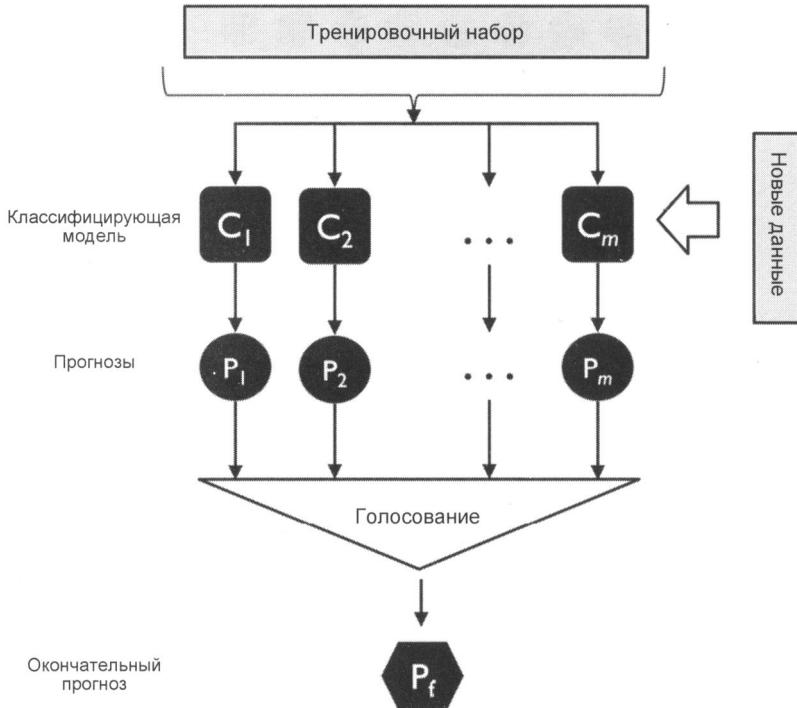


Рис. 7.2. Схема обобщенного ансамблевого подхода

Другими словами, мы вычисляем вероятность того, что предсказание ансамбля неверно. Теперь давайте рассмотрим более конкретный пример из 11 базовых классификаторов ($n = 11$), где каждый классификатор имеет коэффициент ошибок 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \binom{11}{k} 0.25^k (1 - 0.25)^{11-k} = 0.034.$$



Биномиальный коэффициент

В комбинаторике биномиальный коэффициент интерпретируется как количество способов, которыми мы можем выбрать подмножества размером k неупорядоченных элементов из множества размером n , поэтому часто говорят: «число сочетаний из n по k ». Поскольку порядок здесь не имеет значения, биномиальный коэффициент также иногда называют *комбинационным*, или *комбинаторным*, числом, и в несокращенном виде его записывают следующим образом:

$$\frac{n!}{(n - k)! k!}.$$

Здесь символ (!) означает факториал — например: $3! = 3 \times 2 \times 1 = 6$.

Как видите, если выполняются все предположения, то частота ошибок ансамбля (0.034) намного ниже, чем частота ошибок каждого отдельного классификатора (0.25). Заметим, что в этом упрощенном примере разделение 50–50, совершенное четным числом классификаторов n , рассматривается как ошибка, тогда как это верно только в половине случаев. Чтобы сравнить такой идеалистический ансамблевый классификатор с базовым классификатором в диапазоне различных базовых коэффициентов ошибок, реализуем функцию массы вероятности в Python:

```
>>> from scipy.special import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) *
...               error**k *
...               (1-error)**(n_classifier - k)
...             for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

Теперь, имея функцию `ensemble_error`, мы можем вычислить коэффициенты ошибок ансамбля для диапазона различных базовых ошибок от 0.0 до 1.0, чтобы визуализировать взаимосвязь между ансамблевыми и базовыми ошибками на линейном графике (рис. 7.3):

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...                 for error in error_range]
>>> plt.plot(error_range, ens_errors,
...            label='Ансамблевая ошибка',
...            linewidth=2)
>>> plt.plot(error_range, error_range,
...            linestyle='--', label='Базовая ошибка',
...            linewidth=2)
>>> plt.xlabel('Базовая ошибка')
>>> plt.ylabel('Базовая/ансамблевая ошибка')
>>> plt.legend(loc='upper left')
>>> plt.grid(alpha=0.5)
>>> plt.show()
```

Как можно видеть на полученном графике, вероятность ошибки ансамбля всегда ниже, чем ошибка отдельного базового классификатора, если базовые классификаторы работают лучше, чем случайное угадывание ($\varepsilon < 0.5$).

Обратите внимание, что ось Y отображает как базовую ошибку (пунктирная линия), так и ансамблевую (сплошная линия).

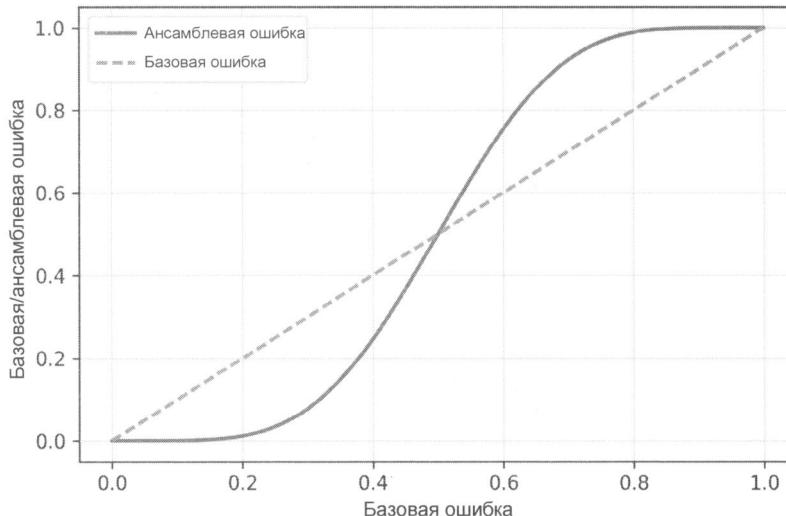


Рис. 7.3. График зависимости между ансамблевой и базовой ошибками

7.2. Объединение классификаторов по методу большинства голосов

Оставив позади краткое введение в ансамблевое обучение, сделаем разминку и реализуем на Python простой ансамблевый классификатор для мажоритарного голосования.



Относительное и мажоритарное большинство

Хотя алгоритм голосования, который мы обсудим в этом разделе, также обобщается на многоклассовые системы посредством относительного большинства, для простоты мы будем использовать термины «мажоритарное большинство» и «мажоритарное голосование», поскольку они часто встречаются в литературе.

7.2.1. Реализация простого мажоритарного классификатора

Алгоритм, который мы собираемся реализовать в этом разделе, позволит нам для достоверности комбинировать различные алгоритмы классификации, снабженные индивидуальными весовыми коэффициентами. Наша цель — создать более сильный метаклассификатор, который уравновешивает слабые стороны отдельных классификаторов применительно к конкретному набору данных. Формально мы можем записать взвешенное большинство голосов следующим образом:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(x) = i).$$

Здесь w_j — вес базового классификатора C_j ; \hat{y} — предсказанная ансамблем метка класса; A — множество уникальных меток класса; χ_A (греч. «хι») — характеристическая,

или индикаторная, функция, которая возвращает 1, если предсказанный класс j -го классификатора соответствует i ($C_j(x) = i$). В случае одинаковых весов мы можем упростить это уравнение и записать его следующим образом:

$$\hat{y} = \text{mode}\{C_1(\mathbf{x}), C_2(\mathbf{x}), \dots, C_m(\mathbf{x})\}.$$

Чтобы лучше понять концепцию *взвешивания*, рассмотрим более конкретный пример. Предположим, что у нас есть ансамбль из трех базовых классификаторов: C_j ($j \in \{1, 2, 3\}$), и мы хотим предсказать метку класса $C_j(x) \in \{0, 1\}$ для определенного экземпляра x . Классификаторы C_1 и C_2 предсказывают метку класса 0, а C_3 предсказывает, что пример принадлежит классу 1. Если прогнозы каждого базового классификатора взвешены одинаково, простое большинство голосов предсказывает, что экземпляр принадлежит классу 0:

$$C_1(\mathbf{x}) \rightarrow 0, \quad C_2(\mathbf{x}) \rightarrow 0, \quad C_3(\mathbf{x}) \rightarrow 1$$

$$\hat{y} = \text{mode}\{0, 0, 1\} = 0$$

Теперь присвоим C_3 вес 0.6, а C_1 и C_2 взвесим с коэффициентом 0.2:

$$\begin{aligned} \hat{y} &= \arg \max_i \sum_{j=1}^m w_j \chi_A(C_j(\mathbf{x}) = i) = \\ &= \arg \max_i [0.2 \times i_0 + 0.2 \times i_0, 0.6 \times i_1] = 1. \end{aligned}$$

Проще говоря, поскольку $3 \times 0.2 = 0.6$, мы можем сказать, что прогноз, сделанный классификатором C_3 , имеет в три раза больший вес, чем прогнозы C_1 или C_2 , которые мы можем записать следующим образом:

$$\hat{y} = \text{mode}\{0, 0, 1, 1, 1\} = 1.$$

Чтобы перевести идею взвешенного большинства голосов в код Python, воспользуемся удобными функциями NumPy `argmax` и `bincount`, где `bincount` подсчитывает количество вхождений каждой метки класса, а функция `argmax` затем возвращает индексную позицию наибольшего счетчика, соответствующую мажоритарной метке класса (предполагается, что метки классов начинаются с 0):

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...                      weights=[0.2, 0.2, 0.6]))
1
```

Как вы помните из описания логистической регрессии, приведенного в главе 3, некоторые классификаторы в scikit-learn также могут возвращать вероятность предсказанной метки класса с помощью метода `predict_proba`. Чтобы использовать для мажоритарного голосования прогнозируемые вероятности классов вместо меток, классификаторы в нашем ансамбле должны быть хорошо откалиброваны. Модифицированная версия мажоритарного голосования для предсказания меток классов на основе вероятностей может быть записана следующим образом:

$$\hat{y} = \arg \max_i \sum_{j=1}^m w_j p_{ij}.$$

Здесь p_{ij} — предсказанная вероятность j -го классификатора для метки класса i .

Чтобы продолжить наш предыдущий пример, давайте предположим, что у нас есть задача бинарной классификации с метками классов $i \in \{0, 1\}$ и ансамблем из трех классификаторов C_j ($j \in \{1, 2, 3\}$). Предположим, что классификаторы C_j возвращают следующие вероятности принадлежности к классу для конкретного экземпляра x :

$$C_1(x) \rightarrow [0.9, 0.1], C_2(x) \rightarrow [0.8, 0.2], C_3(x) \rightarrow [0.4, 0.6].$$

Используя те же веса, что и ранее (0.2, 0.2 и 0.6), мы можем рассчитать вероятности отдельных классов следующим образом:

$$p(i_0|x) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1|x) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y} = \arg \max_i [p(i_0|x), p(i_1|x)] = 0$$

Для реализации взвешенного большинства голосов на основе вероятностей классов мы снова обратимся к библиотеке NumPy, чтобы воспользоваться функциями `np.average` и `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                 [0.8, 0.2],
...                 [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([0.58, 0.42])
>>> np.argmax(p)
0
```

Теперь у нас есть все необходимые блоки кода, и мы можем реализовать класс `MajorityVoteClassifier` на Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator
class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, classifiers, vote='classlabel', weights=None):

        self.classifiers = classifiers
        self.named_classifiers = {
            key: value for key,
            value in _name_estimators(classifiers)
        }
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        if self.vote not in ('probability', 'classlabel'):
```

```

raise ValueError(f"vote должен быть 'probability' "
                  f"или 'classlabel'"
                  f"; got (vote={self.vote})")

if self.weights and
len(self.weights) != len(self.classifiers):
    raise ValueError(f'Количество классификаторов и'
                     f' весов должно совпадать'
                     f'; имеются {len(self.weights)} весов, '
                     f'{len(self.classifiers)} классификаторов')

# Используем LabelEncoder, чтобы проверить, что метки
# начинаются с 0, это важно для вызова np.argmax
# в self.predict
self.lablenc_ = LabelEncoder()
self.lablenc_.fit(y)
self.classes_ = self.lablenc_.classes_
self.classifiers_ = []
for clf in self.classifiers:
    fitted_clf = clone(clf).fit(X,
                                 self.lablenc_.transform(y))
    self.classifiers_.append(fitted_clf)
return self

```

Мы снабдили этот код достаточно подробными комментариями. Но прежде, чем реализовать оставшиеся методы, давайте сделаем небольшой перерыв и поясним блок кода, который на первый взгляд может показаться запутанным: мы подключили здесь родительские классы `BaseEstimator` и `ClassifierMixin`, чтобы использовать некоторые базовые функции, включая методы `get_params` и `set_params`, для получения и назначения параметров классификатора, а также метод `score` для расчета точности предсказания.

Далее мы добавим метод `predict` для прогнозирования метки класса путем мажоритарного голосования на основе меток класса, если мы инициализируем новый объект `MajorityVoteClassifier` с помощью параметра `voice='classlabel'`. В качестве альтернативы мы могли бы инициализировать классификатор ансамбля с параметром `voice='probability'`, чтобы предсказать метку класса на основе вероятностей членства в классе. Кроме того, мы также добавим метод `predict_proba`, возвращающий усредненные вероятности, что полезно при вычислении *площади рабочей характеристики приемника под кривой* (Receiver Operating Characteristic Area Under the Curve, ROC AUC):

```

def predict(self, X):
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X), axis=1)
    else: # 'classlabel' vote
        # Collect results from clf.predict calls
        predictions = np.asarray([
            clf.predict(X) for clf in self.classifiers_
        ]).T

        maj_vote = np.apply_along_axis(
            lambda x: np.argmax(
                np.bincount(x, weights=self.weights)
            ),

```

```

        axis=1, arr=predictions
    )
maj_vote = self.lablenc_.inverse_transform(maj_vote)
return maj_vote

def predict_proba(self, X):
    probas = np.asarray([clf.predict_proba(X)
                         for clf in self.classifiers_])
    avg_proba = np.average(probas, axis=0,
                           weights=self.weights)
    return avg_proba

def get_params(self, deep=True):
    if not deep:
        return super().get_params(deep=False)
    else:
        out = self.named_classifiers.copy()
        for name, step in self.named_classifiers.items():
            for key, value in step.get_params(
                deep=True).items():
                out[f'{name}__{key}'] = value
        return out

```

Обратите также внимание, что в этом коде мы определили нашу собственную модифицированную версию метода `get_params`, чтобы использовать функцию `_name_estimators` для доступа к параметрам отдельных классификаторов в ансамбле. Сначала это может показаться немного сложным, но полное понимание придет, когда мы будем использовать поиск по сетке для настройки гиперпараметров в последующих разделах.



VotingClassifier в scikit-learn

Хотя реализация класса `MajorityVoteClassifier` очень полезна для демонстрационных целей, мы реализовали более сложную версию этого классификатора мажоритарного голосования в `scikit-learn`, взяв за основу код из первого издания этой книги. Ансамблевый классификатор доступен как `sklearn.ensemble.VotingClassifier` в версии `scikit-learn 0.17` и новее. Вы можете узнать больше о `VotingClassifier` по адресу: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html>.

7.2.2. Использование принципа мажоритарного голосования для прогнозирования

Теперь пришло время ввести в действие `MajorityVoteClassifier`, который мы разработали в предыдущем разделе. Но сначала давайте подготовим набор данных, на котором мы можем его протестировать. Поскольку мы уже знакомы с методами загрузки наборов данных из файлов CSV, воспользуемся имеющейся возможностью и загрузим набор данных Iris из модуля `datasets` `scikit-learn`. Кроме того, для наглядности мы выберем только два признака: ширину чашелистика и длину лепестка, чтобы сделать задачу классификации более сложной. Хотя наш классификатор `majorityVoteClassifier` обобща-

ет задачи с несколькими классами, мы будем классифицировать только примеры цветков из классов Iris-versicolor и Iris-virginica, с помощью которых позже вычислим ROC AUC. Код выглядит следующим образом:

```
>>> from sklearn import datasets  
>>> from sklearn.model_selection import train_test_split  
>>> from sklearn.preprocessing import StandardScaler  
>>> from sklearn.preprocessing import LabelEncoder  
>>> iris = datasets.load_iris()  
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]  
>>> le = LabelEncoder()  
>>> y = le.fit_transform(y)
```



Получение вероятности принадлежности к классу из деревьев решений

Для вычисления оценки ROC AUC библиотека scikit-learn использует метод predict_proba (если он применим). В главе 3 было показано, как вычисляются вероятности принадлежности к классу в моделях логистической регрессии. В деревьях решений вероятности рассчитываются на основе вектора частот, который создается для каждого узла во время обучения. В такой вектор собраны значения частоты каждой метки класса, вычисленные из распределения метки класса в этом узле. Затем частоты нормализуют так, чтобы их сумма равнялась 1. Аналогично метки классов k-ближайших соседей объединяют, чтобы вернуть нормализованные частоты меток классов в алгоритме k-ближайших соседей. Хотя нормализованные вероятности, возвращаемые как деревом решений, так и классификатором k-ближайших соседей, могут выглядеть аналогично вероятностям, полученным из модели логистической регрессии, следует знать, что на самом деле они не получены из функций распределения масс.

Далее мы разделим экземпляры набора Iris на 50% обучающих и 50% тестовых данных:

```
>>> X_train, X_test, y_train, y_test =\n...     train_test_split(X, y,\n...                     test_size=0.5,\n...                     random_state=1,\n...                     stratify=y)
```

Используя обучающий набор данных, обучим три разных классификатора:

- ◆ классификатор на основе логистической регрессии;
- ◆ классификатор на основе дерева решений;
- ◆ классификатор по методу k-ближайших соседей.

Затем оценим производительность модели каждого классификатора с помощью 10-кратной перекрестной проверки на обучающем наборе данных, прежде чем объединить их в ансамблевый классификатор:

```
>>> from sklearn.model_selection import cross_val_score  
>>> from sklearn.linear_model import LogisticRegression  
>>> from sklearn.tree import DecisionTreeClassifier  
>>> from sklearn.neighbors import KNeighborsClassifier  
>>> from sklearn.pipeline import Pipeline
```

```

>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                             C=0.001,
...                             solver='lbfgs',
...                             random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                                 criterion='entropy',
...                                 random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                               p=2,
...                               metric='minkowski')
>>> pipe1 = Pipeline([('sc', StandardScaler()),
...                     ('clf', clf1)])
>>> pipe3 = Pipeline([('sc', StandardScaler()),
...                     ('clf', clf3)])
>>> clf_labels = ['Логистическая регрессия', 'Дерево решений', 'KNN']
>>> print('10-кратная перекрестная проверка:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print(f'ROC AUC: {scores.mean():.2f} '
...           f'(+/- {scores.std():.2f}) [{label}]')

```

Результат работы этого кода показывает, что прогностические характеристики отдельных классификаторов почти равны:

10-кратная перекрестная проверка:
ROC AUC: 0.92 (+/- 0.15) [Логистическая регрессия]
ROC AUC: 0.87 (+/- 0.18) [Дерево решений]
ROC AUC: 0.85 (+/- 0.13) [KNN]

У вас мог возникнуть вопрос: почему мы обучали логистическую регрессию и классификатор k-ближайших соседей как часть конвейера? Причина этого в том, что, как было сказано в главе 3, и алгоритмы логистической регрессии, и алгоритмы k-ближайших соседей (использующие метрику евклидова расстояния), в отличие от деревьев решений, не являются масштабно-инвариантными. Хотя все признаки набора Iris измеряются в одном масштабе (сантиметры), хорошим тоном считается работа со стандартизованными признаками.

Теперь давайте перейдем к более интересной части и объединим отдельные классификаторы для голосования по правилу большинства в нашем MajorityVoteClassifier:

```

>>> mv_clf = MajorityVoteClassifier(
...     classifiers=[pipe1, clf2, pipe3]
... )
>>> clf_labels += ['Мажоритарное голосование']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]

```

```
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                               X=X_train,
...                               y=y_train,
...                               cv=10,
...                               scoring='roc_auc')
...     print(f'ROC AUC: {scores.mean():.2f} ±
...           {scores.std():.2f} [{label}]')
ROC AUC: 0.92 (+/- 0.15) [Логистическая регрессия]
ROC AUC: 0.87 (+/- 0.18) [Дерево решений]
ROC AUC: 0.85 (+/- 0.13) [KNN]
ROC AUC: 0.98 (+/- 0.05) [Мажоритарное голосование]
```

Как видите, при 10-кратной перекрестной проверке производительность MajorityVotingClassifier выше, чем у отдельных классификаторов.

7.2.3. Оценка и настройка ансамблевого классификатора

В этом разделе мы вычислим кривые ROC на тестовом наборе данных, чтобы убедиться, что MajorityVoteClassifier хорошо обобщает новые данные. Следует помнить, что тестовый набор данных нельзя использовать для выбора модели — его предназначение в том, чтобы служить источником беспристрастной оценки обобщающей способности классификатора:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyles):
...     # пусть метка положительного класса — это 1
...     y_pred = clf.fit(X_train,
...                       y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...               color=clr,
...               linestyle=ls,
...               label=f'{label} (auc = {roc_auc:.2f})')
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...           linestyle='--',
...           color='gray',
...           linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
```

```
>>> plt.xlabel('Доля ложноположительных прогнозов (FPR)')
>>> plt.ylabel('Доля истинно положительных прогнозов (TPR)')
>>> plt.show()
```

Как можно видеть на рис. 7.4, по линии ROC ансамблевый классификатор хорошо работает с тестовым набором данных ($\text{ROC AUC} = 0.95$). Однако заметно, что и классификатор на основе логистической регрессии столь же хорошо работает с этим набором данных, что, вероятно, связано с высокой дисперсией (в нашем случае — чувствительностью к тому, как мы разделяем набор данных), потому что размер набора относительно небольшой.

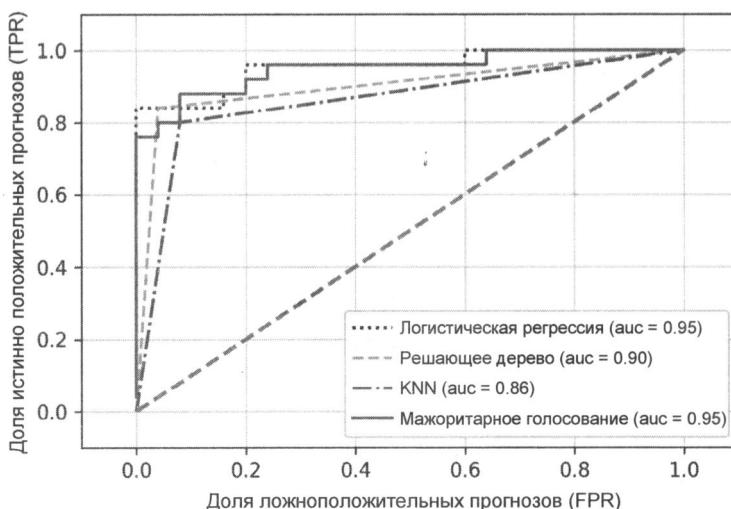


Рис. 7.4. Кривая ROC для различных классификаторов

Поскольку мы выбрали для примеров классификации только два признака, было бы интересно посмотреть, как на самом деле выглядит область принятия решений ансамблевого классификатора.

Хотя необходимости в стандартизации обучающих признаков до подбора модели нет — раз наши конвейеры логистической регрессии и k-ближайших соседей автоматически позаботились об этом, мы выполним стандартизацию набора обучающих данных, чтобы области принятия решений в дереве решений имели тот же масштаб для лучшей наглядности:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
```

```
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                           all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='green',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -5.,
...            s='Ширина чашелистика [стандартизована]',
...            ha='center', va='center', fontsize=12)
>>> plt.text(-12.5, 4.5,
...            s='Длина лепестка [стандартизована]',
...            ha='center', va='center',
...            fontsize=12, rotation=90)
>>> plt.show()
```

Интересно (но вполне ожидаемо), что области принятия решений ансамблевого классификатора представляют собой гибрид областей принятия решений отдельных классификаторов. На первый взгляд, граница решения большинством голосов очень похожа на решение «обрубка» дерева решений, которое ортогонально оси y при ширине чашелистика ≥ 1 . Однако вы также можете заметить нелинейность, внесенную классификатором по методу k-ближайших соседей (рис. 7.5).

Прежде чем приступить к настройке отдельных параметров классификатора для ансамблевой классификации, вызовем метод `get_params`, чтобы иметь общее представление о том, как мы можем получить доступ к отдельным параметрам внутри объекта `GridSearchCV`:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier':
DecisionTreeClassifier(class_weight=None, criterion='entropy',
max_depth=1, max_features=None,
max_leaf_nodes=None,
min_samples_leaf=1,
min_samples_split=2,
min_weight_fraction_leaf=0.0,
random_state=0, splitter='best'),
```

```
'decisiontreeclassifier__class_weight': None,
'decisiontreeclassifier__criterion': 'entropy',
[...]
'decisiontreeclassifier__random_state': 0,
'decisiontreeclassifier__splitter': 'best',
'pipeline-1':
Pipeline(steps=[('sc', StandardScaler(copy=True,
                                         with_mean=True,
                                         with_std=True)),
                ('clf', LogisticRegression(C=0.001,
                                           class_weight=None,
                                           dual=False,
                                           fit_intercept=True,
                                           intercept_scaling=1,
                                           max_iter=100,
                                           multi_class='ovr',
                                           penalty='l2',
                                           random_state=0,
                                           solver='liblinear',
                                           tol=0.0001,
                                           verbose=0))]),
'pipeline-1_clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                    fit_intercept=True, intercept_scaling=1,
                    max_iter=100, multi_class='ovr',
                    penalty='l2', random_state=0,
                    solver='liblinear', tol=0.0001, verbose=0),
'pipeline-1_clf_C': 0.001,
'pipeline-1_clf_class_weight': None,
'pipeline-1_clf_dual': False,
[...]
'pipeline-1_sc_with_std': True,
'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True,
                                         with_mean=True,
                                         with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                             leaf_size=30,
                                             metric='minkowski',
                                             metric_params=None,
                                             n_neighbors=1,
                                             p=2,
                                             weights='uniform'))]),
'pipeline-2_clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                     metric='minkowski', metric_params=None,
                     n_neighbors=1, p=2, weights='uniform'),
'pipeline-2_clf_algorithm': 'auto',
[...]
'pipeline-2_sc_with_std': True)
```

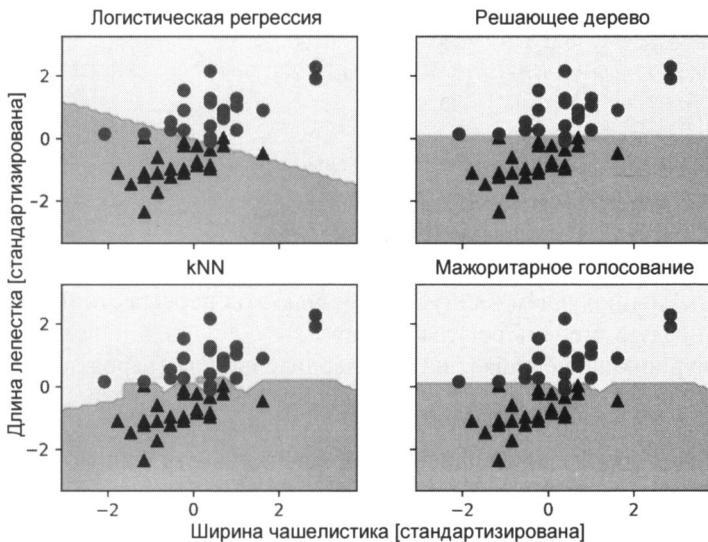


Рис. 7.5. Границы принятия решений для различных классификаторов

Итак, теперь вы знаете, как получить доступ к отдельным атрибутам классификатора при помощи метода `get_params`. Теперь в демонстрационных целях с помощью поиска по сетке настроим параметр обратной регуляризации классификатора С логистической регрессии и глубину дерева решений:

```
>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier_max_depth': [1, 2],
...             'pipeline-1_clf_C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                       param_grid=params,
...                       cv=10,
...                       scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

Завершив поиск по сетке, выведем различные комбинации значений гиперпараметров и средние оценки ROC AUC, рассчитанные с помощью 10-кратной перекрестной проверки:

```
>>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     mean_score = grid.cv_results_['mean_test_score'][r]
...     std_dev = grid.cv_results_['std_test_score'][r]
...     params = grid.cv_results_['params'][r]
...     print(f'{mean_score:.3f} +/- {std_dev:.2f} ({params})')
0.983 +/- 0.05 {'decisiontreeclassifier_max_depth': 1,
                 'pipeline-1_clf_C': 0.001}
0.983 +/- 0.05 {'decisiontreeclassifier_max_depth': 1,
                 'pipeline-1_clf_C': 0.1}
0.967 +/- 0.10 {'decisiontreeclassifier_max_depth': 1,
                 'pipeline-1_clf_C': 100.0}
0.983 +/- 0.05 {'decisiontreeclassifier_max_depth': 2,
                 'pipeline-1_clf_C': 0.001}
```

```
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2,
                 'pipeline-1__clf_C': 0.1}
0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 2,
                 'pipeline-1__clf_C': 100.0}
>>> print(f'Лучшие параметры: {grid.best_params_}')
Лучшие параметры: {'decisiontreeclassifier__max_depth': 1,
                     'pipeline-1__clf_C': 0.001}
>>> print(f'ROC AUC : {grid.best_score_:.2f}')
ROC AUC: 0.98
```

Как можно видеть, мы получаем наилучшие результаты перекрестной проверки, когда выбираем более низкую степень регуляризации ($C = 0.001$), в то время как глубина дерева, по-видимому, вообще не влияет на производительность (вероятно, обрубка дерева решений достаточно для разделения данных). Чтобы напомнить себе, что использовать тестовый набор данных более одного раза для оценки модели — плохая практика, мы не станем оценивать производительность обобщения настроенных гиперпараметров в этом разделе, и поскорее изучим альтернативный подход к ансамблевому обучению: бэггинг.



Построение ансамблей с помощью стекинга

Подход, основанный на большинстве голосов, который мы реализовали в этом разделе, не следует путать со *стекингом* (stacking). Алгоритм стекинга можно рассматривать как двухуровневый ансамбль, в котором первый уровень состоит из отдельных классификаторов, передающих свои прогнозы на второй уровень, где другой классификатор (обычно логистическая регрессия) обучается прогнозам классификатора первого уровня, чтобы сделать окончательные прогнозы. Для получения дополнительной информации о стекинге рекомендуем обратиться к следующим ресурсам:

- алгоритм стекинга более подробно описан Дэвидом Вулпертом в статье «Stacked generalization, Neural Networks», 5(2):241–259, 1992, <https://www.sciencedirect.com/science/article/abs/pii/S0893608005800231>;
- заинтересованные читатели могут найти наш видеоурок о стекинге на YouTube по адресу: <https://www.youtube.com/watch?v=8T2emza6g80>;
- версия классификатора на основе стекирования, совместимая с scikit-learn, доступна на сайте mlxtend по адресу: http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/;
- недавно в scikit-learn был добавлен класс StackingClassifier (доступен в версии 0.22 и новее). Для получения дополнительной информации см. документацию по адресу: <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html>.

7.3. Бэггинг: создание ансамбля классификаторов из исходного набора данных

Бэггинг (bagging, сокращение от bootstrap aggregating) — это метод обучения ансамбля, тесно связанный с классификатором MajorityVoteClassifier, который мы реализовали в предыдущем разделе. Однако вместо того, чтобы использовать один и тот же набор

обучающих данных для обучения отдельных классификаторов в ансамбле, мы берем бутстрэп-выборки (случайные выборки с заменой) из исходного обучающего набора данных, поэтому бэггинг также известен как бутстрэп-агрегация. Механизм бэггинга схематически показан на рис. 7.6.

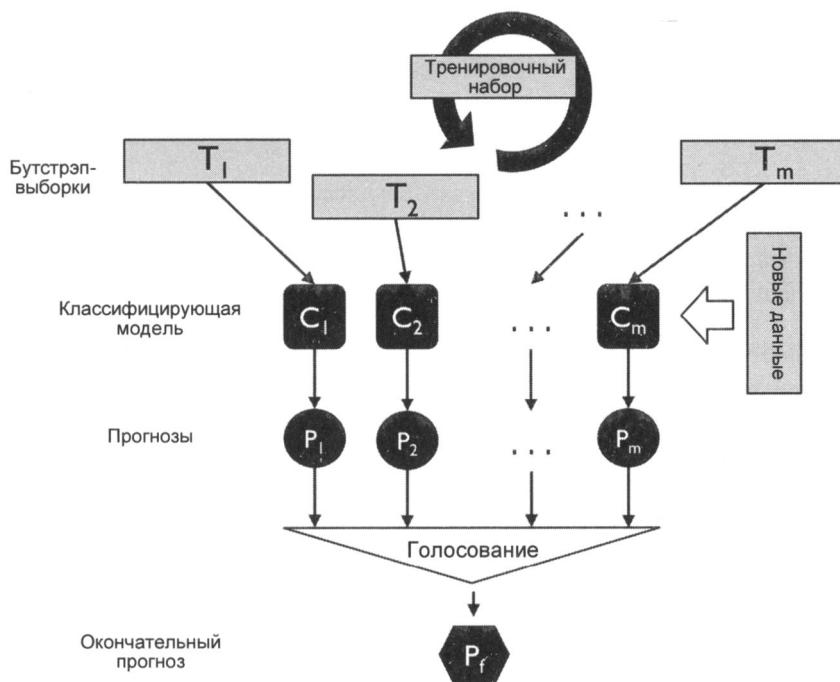


Рис. 7.6. Механизм бэггинга

В следующих разделах мы рассмотрим простой пример реализации бэггинга и используем scikit-learn для классификации образцов вина.

7.3.1. Коротко о бэггинге

Чтобы лучше разобраться, как работает классификатор на основе бэггинга, давайте рассмотрим пример, приведенный на рис. 7.7. Здесь показаны семь разных обучающих экземпляров (обозначенных индексами 1–7), которые выбираются случайным образом с заменой в каждом раунде бэггинга. Каждая бутстрэп-выборка затем служит для обучения классификатора C_j , который чаще всего представляет собой необрезанное дерево решений.

Как можно здесь видеть, каждый классификатор получает случайное подмножество экземпляров из обучающего набора данных. Мы обозначили эти случайные выборки, полученные с помощью бэггинга, как раунд бэггинга 1, раунд бэггинга 2 и т. д.. Каждая подвыборка содержит определенную часть повторов, а некоторые из исходных экземпляров вообще не появляются в наборе данных с повторной выборкой из-за выборки с заменой. После обучения отдельных классификаторов на бутстрэп-выборках их прогнозы агрегируют (объединяют) с помощью мажоритарного голосования.

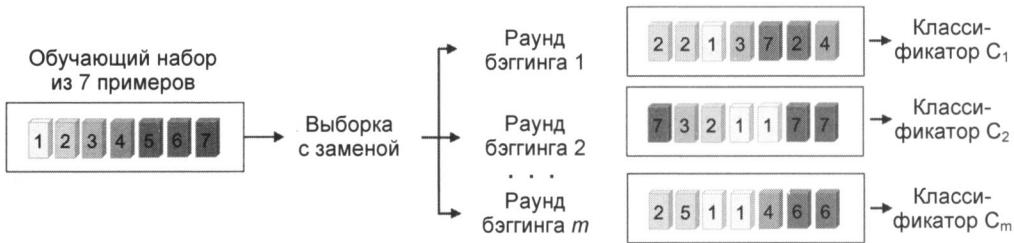


Рис. 7.7. Пример бэггинга

Стоит отметить, что бэггинг также связан с классификатором на основе случайного леса, который мы рассматривали в главе 3. Фактически случайный лес — это особый случай бэггинга, когда мы также используем подмножества случайных признаков при обучении отдельных деревьев решений.



Ансамблирование моделей с использованием бэггинга

Бэггинг впервые был предложен Лео Брейманом в техническом отчете, опубликованном в 1994 году. Он также показал, что бэггинг может повысить точность нестабильных моделей и снизить риск переобучения. Чтобы узнать больше о бэггинге, мы настоятельно рекомендуем прочитать его статью «Bagging predictors» by L. Breiman, Machine Learning, 24(2):123–140, 1996, которая находится в Интернете в свободном доступе.

7.3.2. Применение бэггинга для классификации экземпляров набора данных Wine

Для демонстрации бэггинга в действии понадобится более сложная задача классификации, и мы воспользуемся набором данных Wine, с которым познакомились в главе 4. Здесь мы будем рассматривать только классы вин 2 и 3 и выберем два признака: Alcohol и OD280/OD315 of diluted wines:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                      'machine-learning-databases/'
...                      'wine/wine.data',
...                      header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                     'Malic acid', 'Ash',
...                     'Alcalinity of ash',
...                     'Magnesium', 'Total phenols',
...                     'Flavanoids', 'Nonflavanoid phenols',
...                     'Proanthocyanins',
...                     'Color intensity', 'Hue',
...                     'OD280/OD315 of diluted wines',
...                     'Proline']
...
>>> # отбрасываем класс 1
>>> df_wine = df_wine[df_wine['Class label'] != 1]
```

```
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol',
...                 'OD280/OD315 of diluted wines']].values
```



Получение набора данных Wine

Вы можете найти копию этого набора данных (и всех других наборов данных, используемых в книге) в ее файловом архиве, когда работаете в автономном режиме, или если сервер UCI по адресу <https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data> временно недоступен. Например, чтобы загрузить набор данных Wine из локального каталога, нужно заменить в приведенном коде строки:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

на следующие:

```
df = pd.read_csv(
    'your/local/path/to/wine.data', header=None
)
```

Затем мы закодируем метки классов в двоичный формат и разделим исходный набор данных на обучающий и тестовый наборы в соотношении 80/20 соответственно:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test = \
...         train_test_split(X, y,
...                         test_size=0.2,
...                         random_state=1,
...                         stratify=y)
```

Алгоритм `BaggingClassifier` уже реализован в `scikit-learn`, поэтому его можно просто импортировать из подмодуля `ensemble`. Здесь мы применим в качестве базового классификатора несокращенное дерево решений и создадим ансамбль из 500 деревьев, обученных на разных бутстрэп-выборках из обучающего набора:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 random_state=1,
...                                 max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           max_samples=1.0,
...                           max_features=1.0,
...                           bootstrap=True,
...                           bootstrap_features=False,
...                           n_jobs=1,
...                           random_state=1)
```

Теперь рассчитаем показатель точности прогноза на обучающем и тестовом наборах данных, чтобы сравнить производительность классификатора на основе бэггинга с производительностью одного несокращенного дерева решений:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Точность при обучении и тестировании '
...      f'{tree_train:.3f}/{tree_test:.3f}')
Точность при обучении и тестировании 1.000/0.833
```

Судя по выведенным значениям точности, несокращенное дерево решений правильно предсказывает все метки класса обучающих экземпляров, однако существенно более низкая точность на тестовом наборе указывает на высокую дисперсию (переобучение) модели:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Точность бэггинга при обучении/тестировании '
...      f'{bag_train:.3f}/{bag_test:.3f}')
Точность бэггинга при обучении/тестировании 1.000/0.917
```

Хотя точности дерева решений и бэггинг-классификатора на обучающем наборе данных полностью идентичны (100%), мы видим, что бэггинг-классификатор имеет немноголибо высокую способность к обобщению на тестовом наборе данных. Давайте сравним области принятия решений, построенные деревом решений и бэггинг-классификатором:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                           [tree, bag],
...                           ['Дерево решений', 'Бэггинг']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
```

```

...
    axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...
    axarr[idx].scatter(X_train[y_train==0, 0],
                        X_train[y_train==0, 1],
                        c='blue', marker='^')
...
    axarr[idx].scatter(X_train[y_train==1, 0],
                        X_train[y_train==1, 1],
                        c='green', marker='o')
...
    axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('OD280/OD315 of diluted wines', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...            s='Alcohol',
...            ha='center',
...            va='center',
...            fontsize=12,
...            transform=axarr[1].transAxes)
>>> plt.show()

```

На полученном графике хорошо видно, что кусочно-линейная разделяющая граница трехузлового дерева решений выглядит более гладкой в случае использования бэггинга (рис. 7.8).

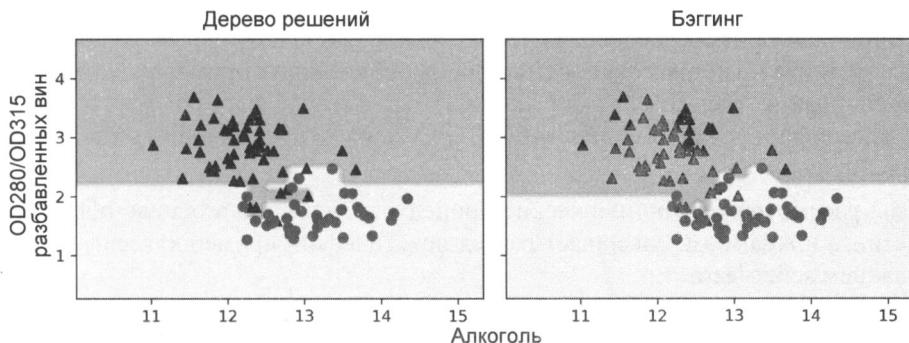


Рис. 7.8. Форма решающей границы обычного дерева решений по сравнению с бэггингом

В этом разделе мы рассмотрели только очень простой пример бэггинга. На практике более сложные задачи классификации и высокая размерность набора данных могут легко привести к переобучению отдельных деревьев решений, и именно в такой ситуации алгоритм бэггинга может сполна проявить свои сильные стороны. Наконец, нужно отметить, что использование бэггинга может послужить эффективным средством для уменьшения дисперсии модели. Однако бэггинг почти не уменьшает систематические ошибки, т. е. он плохо работает с моделями, которые слишком просты, чтобы хорошо улавливать тенденции в данных. Вот почему бэггинг обычно стараются применять к классификаторам с низкой систематической ошибкой — например, к несокращенным деревьям решений.

7.4. Использование «слабых учеников» в механизме адаптивного бустинга

В этом последнем разделе, посвященном ансамблевым методам, мы обсудим *бустинг* (boosting), уделив особое внимание его наиболее распространенной реализации: *адаптивному бустингу* (алгоритм AdaBoost).



История успеха AdaBoost

Основная идея AdaBoost была сформулирована Робертом Шапиром в 1990 г. в статье «The Strength of Weak Learnability», Machine Learning, 5(2): 197–227, 1990, по адресу: <http://rob.schapire.net/papers/strengthofweak.pdf>. После того как Роберт Шапир и Йоав Фройнд представили алгоритм AdaBoost в материалах Тринадцатой международной конференции по машинному обучению (ICML 1996), за последующие годы он стал одним из наиболее широко используемых ансамблевых методов (см. «Experiments with a New Boosting Algorithm» by Y. Freund, R. E. Schapire, and others, ICML, volume 96, 148–156, 1996). В 2003 г. Фройнд и Шапир получили за свою новаторскую работу премию Гёделя — престижную премию, присуждаемую за самые выдающиеся публикации в области компьютерных наук.

При бустинге ансамбль состоит из очень простых базовых классификаторов, также часто называемых *слабыми учениками* (weak learner), которые обычно имеют лишь небольшое преимущество в производительности по сравнению со случным угадыванием, — типичным примером слабого ученика является обрубок дерева решений. Ключевая идея бустинга — концентрация внимания на обучающих примерах, которые трудно классифицировать, т. е. позволить слабым ученикам впоследствии учиться на неправильно классифицированных обучающих примерах, чтобы улучшить производительность ансамбля.

Далее мы рассмотрим алгоритмические процедуры, лежащие в основе общей концепции бустинга и AdaBoost. Завершает раздел практический пример классификации с использованием scikit-learn.

7.4.1. Как работает адаптивный бустинг?

В отличие от бэггинга, первоначальный вариант алгоритма бустинга использует случайные подмножества обучающих примеров, выбранных из набора обучающих данных без замены. Исходную процедуру бустинга можно свести к следующим четырем ключевым шагам:

1. Извлекаем случайное подмножество (выборку) обучающих примеров d_1 без замены из обучающего набора данных D для обучения слабого ученика C_1 .
2. Извлекаем вторую случайную выборку d_2 без замены из набора обучающих данных и добавляем к ней 50% примеров, которые ранее были неправильно классифицированы, чтобы обучить слабого ученика C_2 .
3. Находим в обучающем наборе данных D примеры d_3 , по которым C_1 и C_2 расходятся во мнениях, чтобы обучить третьего слабого ученика — C_3 .
4. Объединяем выводы слабых учащихся C_1 , C_2 и C_3 по принципу мажоритарного голосования.

Как отмечал Лео Брейман³, бустинг может способствовать уменьшению систематической ошибки, а также дисперсии, по сравнению с моделями бэггинга. На практике, однако, алгоритмы бустинга, такие как AdaBoost, также известны своей высокой дисперсией, т. е. склонностью к переобучению⁴.

В отличие от описанной ранее исходной процедуры бустинга, AdaBoost использует для обучения слабых учеников полный набор обучающих данных — при этом обучающие примеры заново взвешиваются на каждой итерации для создания сильного классификатора, который учится на ошибках предыдущих слабых учеников в ансамбле.

Прежде чем углубиться в конкретные детали алгоритма AdaBoost, внимательно рассмотрите рис. 7.9 и постарайтесь лучше понять основную концепцию AdaBoost.

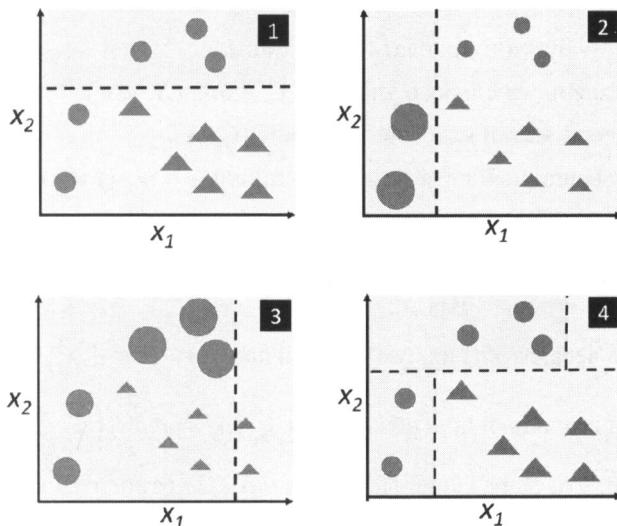


Рис. 7.9. Подход AdaBoost к обучению слабых учеников

- ◆ Давайте начнем изучение этого рисунка с поля под номером 1. На нем представлен набор обучающих данных для бинарной классификации, где всем обучающим примерам присвоены равные веса. Используя этот набор, мы обучаем обрубок решающего дерева (показанного пунктирной линией), который пытается классифицировать примеры двух классов (треугольники и кружки), а также, возможно, минимизировать функцию потерь (или оценку примеси в частном случае ансамблей решающих деревьев).
- ◆ На следующем раунде (поле 2) мы присваиваем больший вес двум ранее неправильно классифицированным примерам (кружки). Кроме того, мы снижаем вес правильно классифицированных примеров. Следующий этап решения теперь будет больше сосредоточен на обучающих примерах с наибольшим весом — обучающих примерах, которые предположительно трудно классифицировать.

³ См. «Bias, variance, and arcing classifiers», 1996 г.

⁴ См. «An improvement of AdaBoost to avoid overfitting» by G. Raetsch, T. Onoda, K. R. Mueller. Proceedings of the International Conference on Neural Information Processing, CiteSeer, 1998.

- ◆ Слабый ученик, показанный на втором поле, неправильно классифицирует три разных примера из класса кружков, которым затем присваивается больший вес, как показано на поле 3.
- ◆ Предполагая, что наш ансамбль AdaBoost состоит только из трех раундов бустинга, мы затем объединяем трех слабых учеников, обученных на разных тренировочных подмножествах, путем взвешенного большинства голосов, как показано на поле 4.

Теперь, когда вы лучше понимаете механизм работы AdaBoost, давайте более подробно рассмотрим алгоритм с использованием псевдокода. Для ясности будем обозначать поэлементное умножение крестиком « \times », а скалярное произведение двух векторов — точкой « \cdot »:

1. Присвоить вектору весов w одинаковые веса, где $\sum_i w_i = 1$.
2. Для j из m раундов бустинга сделать следующее:
 - a. Обучить взвешенного слабого ученика: $C_j = \text{train}(X, y, w)$.
 - b. Спрогнозировать метки классов: $\hat{y} = \text{predict}(C_j, X)$.
 - c. Вычислить взвешенный коэффициент ошибки: $\varepsilon = w \cdot (\hat{y} \neq y)$.
 - d. Вычислить коэффициент: $a_j = 0.5 \log \frac{1 - \varepsilon}{\varepsilon}$.
 - e. Обновить веса: $w := w \times \exp(-a_j \times \hat{y} \times y)$.
 - f. Нормировать веса, чтобы их сумма равнялась 1: $w := w / \sum_i w_i$.
3. Вычислить окончательный прогноз: $\hat{y} = \left(\sum_{j=1}^m (a_j \times \text{predict}(C_j, X)) > 0 \right)$.

Обратите внимание, что выражение $(\hat{y} \neq y)$ на *шаге 2c* относится к двоичному вектору, состоящему из 1 и 0, где 1 присваивается, если прогноз неверен, и 0 в противном случае.

Хотя алгоритм AdaBoost кажется довольно простым, давайте рассмотрим более конкретный пример, используя обучающий набор данных, состоящий из 10 записей, как показано на рис. 7.10.

В первом столбце содержатся индексы обучающих примеров с 1 по 10. Во втором столбце вы можете увидеть значения признаков отдельных примеров (мы предполагаем, что это одномерный набор данных). В третьем столбце показана истинная метка класса y_i для каждой обучающей выборки x_i , где $y_i \in \{1, -1\}$. Начальные веса показаны в четвертом столбце — мы инициализируем веса единообразно (назначая одно и то же постоянное значение) и нормализуем их, чтобы сумма равнялась 1. Поэтому в случае обучающего набора данных с 10 выборками мы присваиваем значение 0.1 каждому весу w_i в векторе весов w . Предсказанные метки классов \hat{y} показаны в пятом столбце, при условии, что критерием разделения служит условие $x \leq 3.0$. Наконец, в последнем столбце таблицы отображаются веса, обновленные в соответствии с правилами, которые мы определили в псевдокоде.

Поскольку вычисление обновлений веса поначалу может показаться сложным, на этот раз мы проследим за ходом вычислений шаг за шагом. Начнем с вычисления взвешенной частоты ошибок ε (*epsilon*), как описано в *шаге 2c*:

Индекс	X	y	Веса	$\hat{y}(x \leq 3.0)$?	Верно?	Новые веса
1	1.0	1	0.1	1	Да	0.072
2	2.0	1	0.1	1	Да	0.072
3	3.0	1	0.1	1	Да	0.072
4	4.0	-1	0.1	-1	Да	0.072
5	5.0	-1	0.1	-1	Да	0.072
6	6.0	-1	0.1	-1	Да	0.072
7	7.0	1	0.1	-1	Нет	0.167
8	8.0	1	0.1	-1	Нет	0.167
9	9.0	1	0.1	-1	Нет	0.167
10	10.0	-1	0.1	-1	Да	0.072

Рис. 7.10. Запуск 10 обучающих проходов с помощью алгоритма AdaBoost

```
>>> y = np.array([1, 1, 1, -1, -1, -1, 1, 1, 1, -1])
>>> yhat = np.array([1, 1, 1, -1, -1, -1, -1, -1, -1, -1])
>>> correct = (y == yhat)
>>> weights = np.full(10, 0.1)
>>> print(weights)
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
>>> epsilon = np.mean(~correct)
>>> print(epsilon)
0.3
```

Здесь `correct` — это логический массив, состоящий из значений `True` и `False`, где первое указывает, что прогноз верен. Оператор `~correct` инвертирует массив, так что `np.mean(~correct)` вычисляет долю неверных прогнозов (`True` считается значением 1, а `False` — 0), т. е. ошибку классификации.

Затем мы вычислим коэффициент α_j , показанный на *шаге 2d*, — он пригодится на *шаге 2e* для обновления весов, а также для весов в прогнозе большинства голосов (*шаг 3*):

```
>>> alpha_j = 0.5 * np.log((1-epsilon) / epsilon)
>>> print(alpha_j)
0.42364893019360184
```

После вычисления коэффициента α_j (`alpha_j`) мы можем обновить вектор весов, используя следующее уравнение:

$$\mathbf{w} := \mathbf{w} \times \exp(-\alpha_j \times \hat{\mathbf{y}} \times \mathbf{y}).$$

Здесь $\hat{\mathbf{y}} \times \mathbf{y}$ — поэлементное умножение между векторами предсказанных и истинных меток классов соответственно. Таким образом, если прогноз \hat{y}_i верен, $\hat{y}_i \times y_i$ будет иметь положительный знак, так что мы уменьшаем i -й вес, поскольку α_j также является положительным числом:

```
>>> update_if_correct = 0.1 * np.exp(-alpha_j * 1 * 1)
>>> print(update_if_correct)
0.06546536707079771
```

Точно так же мы увеличим i -й вес, если \hat{y}_i неправильно предсказал метку, например:

```
>>> update_if_wrong_1 = 0.1 * np.exp(-alpha_j * 1 * -1)
>>> print(update_if_wrong_1)
0.1527525231651947
```

По-другому это можно сделать так:

```
>>> update_if_wrong_2 = 0.1 * np.exp(-alpha_j * -1 * 1)
>>> print(update_if_wrong_2)
0.1527525231651947
```

Воспользуемся полученными значениями для обновления весов следующим образом:

```
>>> weights = np.where(correct == 1,
...                     update_if_correct,
...                     update_if_wrong_1)
>>> print(weights)
array([0.06546537, 0.06546537, 0.06546537, 0.06546537, 0.06546537,
       0.06546537, 0.15275252, 0.15275252, 0.15275252, 0.06546537])
```

Этот код присвоил значение `update_if_correct` всем правильным прогнозам, а значение `update_if_wrong_1` — всем неправильным. Для простоты мы не стали использовать `update_if_wrong_2`, т. к. он в любом случае совпадает с `update_if_wrong_1`.

Выполнив обновление вектора весов, мы нормализуем веса, чтобы их сумма равнялась 1 (*шаг 2*):

$$\mathbf{w} := \frac{\mathbf{w}}{\sum_i w_i}.$$

Реализация в коде может выглядеть так:

```
>>> normalized_weights = weights / np.sum(weights)
>>> print(normalized_weights)
[0.07142857 0.07142857 0.07142857 0.07142857 0.07142857 0.07142857
 0.16666667 0.16666667 0.16666667 0.07142857]
```

В итоге каждый вес, соответствующий правильно классифицированному примеру, будет уменьшен с начального значения 0.1 до 0.0714 для следующего раунда бустинга. Точно так же веса неправильно классифицированных примеров увеличатся с 0.1 до 0.1667.

7.4.2. Применение AdaBoost с помощью scikit-learn

В предыдущем разделе мы вкратце познакомились с принципом работы AdaBoost и готовы перейти к более практической части — обучить классификатор ансамбля AdaBoost с помощью scikit-learn. Воспользуемся тем же частичным набором Wine, что и в предыдущем разделе, для обучения метаклассификатора по методу бэггинга.

В соответствии со значением атрибута `base_estimator` классификатор `AdaBoostClassifier` будет обучен на 500 обрубках дерева решений:

```
>>> from sklearn.ensemble import AdaBoostClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                                 random_state=1,
...                                 max_depth=1)
```

```
>>> ada = AdaBoostClassifier(base_estimator=tree,
...                           n_estimators=500,
...                           learning_rate=0.1,
...                           random_state=1)
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Точность решающего дерева при обучении/проверке '
...       f'{tree_train:.3f}/{tree_test:.3f}')
Точность решающего дерева при обучении/проверке 0.916/0.875
```

Очевидно, обрубок дерева решений не полностью обучился на имеющихся обучающих данных — в отличие от необрязанного дерева решений, которое мы видели в предыдущем разделе:

```
>>> ada = AdaBoost(learning_rate=0.01)
>>> y_train_pred = ada.predict(X_train)
>>> y_test_pred = ada.predict(X_test)
>>> ada_train = accuracy_score(y_train, y_train_pred)
>>> ada_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Точность AdaBoost при обучении/проверке '
...       f'{ada_train:.3f}/{ada_test:.3f}')
Точность AdaBoost при обучении/проверке 1.000/0.917
```

Как видите, модель AdaBoost правильно предсказывает все метки классов обучающего набора данных, а также демонстрирует небольшое увеличение точности на тестовом наборе данных по сравнению с обрубком дерева решений. Однако также заметно, что наша попытка уменьшить систематическую ошибку модели внесла дополнительную дисперсию — увеличился разрыв между точностью на обучающем и тестовом наборе.

Хотя для наглядности мы использовали простой пример, его достаточно, чтобы увидеть, что производительность классификатора AdaBoost немного улучшилась по сравнению с обрубком дерева решений и достигла почти таких же показателей точности, как у классификатора на основе бэггинга, который мы обучали в предыдущем разделе. Однако нужно отметить, что выбирать модель на основе многократного использования одного и того же тестового набора данных — плохая практика. В этом случае оценка обобщающей способности модели может оказаться чрезмерно оптимистичной, о чем мы более подробно говорили в главе 6.

Наконец, давайте взглянем, как выглядят области принятия решений:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                       np.arange(y_min, y_max, 0.1))

>>> f, axarr = plt.subplots(1, 2,
...                        sharex='col',
```

```

...
        sharey='row',
        figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, ada],
...                         ['Дерево решений', 'AdaBoost']):
...     clf.fit(X_train, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                         X_train[y_train==0, 1],
...                         c='blue',
...                         marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                         X_train[y_train==1, 1],
...                         c='green',
...                         marker='o')
...     axarr[idx].set_title(tt)
...     axarr[0].set_ylabel('OD280/OD315 of diluted wines', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...             s='Alcohol',
...             ha='center',
...             va='center',
...             fontsize=12,
...             transform=axarr[1].transAxes)
>>> plt.show()

```

На рис. 7.11 хорошо видно, что разделяющая граница модели AdaBoost существенно сложнее, чем у обрубка дерева решений. Кроме того, заметим, что то, как модель AdaBoost разделяет пространство функций, очень похоже на классификатор на основе бэггинга, который мы обучали в предыдущем разделе.

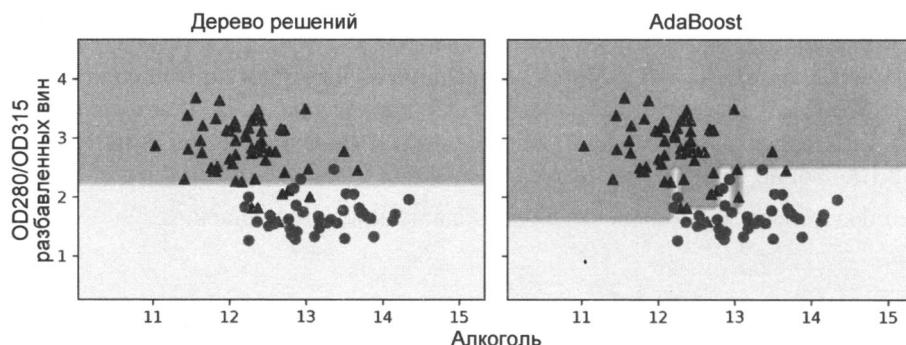


Рис. 7.11. Сравнение решающих границ дерева решений и AdaBoost

Завершая разговор об ансамблевых методах, стоит отметить, что обучение ансамбля увеличивает вычислительную сложность по сравнению с отдельными классификаторами. На практике нам нужно хорошенько подумать о том, хотим ли мы расплачиватьсяся

увеличением вычислительных затрат за зачастую относительно скромное улучшение качества прогнозов.

Широко известным примером такого компромисса является знаменитая премия Netflix в размере 1 миллиона долларов, которая досталась победителям конкурса, применившим ансамблевый алгоритм⁵. Команда-победитель получила главный приз в размере 1 миллиона долларов, однако в Netflix так и не смогли применить модель на практике из-за ее сложности:

«Мы протестировали некоторые новые методы в автономном режиме, но наблюдаемое дополнительное повышение точности, судя по всему, не окупает усилия, необходимые для внедрения их в производственную среду» (<http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html>).

7.5. Градиентный бустинг: обучение ансамбля на основе градиентов потерь

Градиентный бустинг — это еще один вариант представленной в предыдущем разделе концепции бустинга, т. е. последовательного обучения слабых учеников для создания сильного ансамбля. И это чрезвычайно важная тема, поскольку она лежит в основе популярных алгоритмов машинного обучения, таких как XGBoost, который хорошо известен своими победами в соревнованиях Kaggle.

Поначалу алгоритм градиентного бустинга может показаться довольно сложным. Но не волнуйтесь — далее мы рассмотрим его шаг за шагом, начиная с общего подхода. Затем вы узнаете, как градиентный бустинг применяется для классификации, и опробуете его на примере. Наконец, после обзорного знакомства с основами градиентного бустинга мы кратко рассмотрим его популярные реализации, такие как XGBoost, и продемонстрируем применение этого метода на практике.

7.5.1. Сравнение AdaBoost с градиентным бустингом

По сути, градиентный бустинг очень похож на алгоритм AdaBoost, который мы обсуждали ранее в этой главе. AdaBoost обучает обрубки дерева решений на основе ошибок предыдущего обрубка. В частности, ошибки используются для вычисления весов выборки в каждом раунде, а также для вычисления веса классификатора для каждого обрубка дерева решений при объединении отдельных обрубков в ансамбль. Обучение прекращается по достижении максимального количества итераций (обрубков дерева решений). Как и AdaBoost, градиентный бустинг применяет итеративный подход к деревьям решений, используя ошибки прогнозирования. Однако деревья градиентного бустинга обычно глубже, чем обрубки дерева решений, и имеют максимальную глубину от 3 до 6 (или максимальное количество листовых узлов от 8 до 64). Кроме того, в отличие от AdaBoost, градиентный бустинг не использует ошибки прогнозирования для назначения весов выборки, — на их основе формируется целевая переменная для

⁵ Подробности об алгоритме были опубликованы в статье «The BigChaos Solution to the Netflix Grand Prize» by A. Toescher, M. Jahrer, and R. M. Bell, Netflix Prize documentation, 2009, которая доступна по адресу: http://www.stat.osu.edu/~dmsl/GrandPrize2009_BPC_BigChaos.pdf.

подбора следующего дерева. Более того, вместо индивидуального взвешивания для каждого дерева, как в AdaBoost, градиентный бустинг использует глобальную скорость обучения, одинаковую для каждого дерева.

Как видите, AdaBoost и градиентный бустинг во многом схожи, но отличаются в некоторых ключевых аспектах. В следующем разделе мы рассмотрим обобщенный алгоритм градиентного бустинга.

7.5.2. Описание обобщенного алгоритма градиентного бустинга

В этом разделе мы рассмотрим градиентный бустинг для задач классификации. Для простоты возьмем пример бинарной классификации. Заинтересованные читатели могут найти обобщение для многоклассовой классификации с логистическими потерями в разделе 4.6 статьи Дж. Фридмана о градиентном бустинге⁶.



Градиентный бустинг для задач регрессии

Надо сказать, что процедура градиентного бустинга немного сложнее, чем AdaBoost. Для краткости мы опускаем более простой пример регрессии, который был приведен в упомянутой статье Фридмана, но заинтересованным читателям предлагается также посмотреть дополнительный видеоурок по градиентному бустингу для регрессии, который доступен по адресу: <https://www.youtube.com/watch?v=zblsrxc7XpM>.

По сути, градиентный бустинг строит серию деревьев, где каждое дерево обучается на ошибке — разнице между меткой и прогнозируемым значением — предыдущего дерева. С каждым раундом ансамбль деревьев улучшается, поскольку мы подталкиваем каждое дерево в правильном направлении с помощью небольших обновлений. Эти обновления основаны на градиенте потерь, отсюда и происходит название градиентного бустинга.

Теперь перечислим основные шаги градиентного бустинга. Это общий обзор алгоритма, а в следующих разделах мы более подробно исследуем некоторые его части и рассмотрим практический пример.

1. Инициализируем модель, чтобы она возвращала постоянное значение прогноза. Для этого мы используем корневой узел дерева решений, т. е. дерево решений с одним листовым узлом. Обозначим значение, возвращаемое деревом, как \hat{y} , и найдем его путем минимизации дифференцируемой функции потерь L , которую мы определим следующим образом:

$$F_0(x) = \arg \min_{\hat{y}} \sum_{i=1}^n L(y_i, \hat{y}).$$

Здесь n обозначает количество обучающих примеров в нашем наборе данных.

⁶ См. «Greedy function approximation: A gradient boosting machine» by Friedman, 2001,

<https://projecteuclid.org/journals/annals-of-statistics/volume-29/issue-5/Greedyfunction-approximation-A-gradient-boostingmachine/10.1214/aos/1013203451.full>.

2. Для каждого дерева $m = 1, \dots, M$, где M — заданное пользователем общее количество деревьев, мы выполняем следующие вычисления, описанные в шагах 2a–2d:

- Вычисляем разницу между предсказанным значением $F(x_i) = \hat{y}_i$ и меткой класса y_i . Это значение иногда называют *псевдооткликом* (pseudo-response), или *псевдоостатком* (pseudo-residual). Более формально мы можем записать этот псевдоостаток как отрицательный градиент функции потерь по отношению к прогнозируемым значениям:

$$r_{im} = - \left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{m-1}(x)} \quad \text{для } i = 1, \dots, n.$$

В нашем случае $F(x)$ является прогнозом предыдущего дерева: $F_{m-1}(x)$. В первом раунде это будет постоянное значение из дерева (однолистового узла) из шага 1.

- Обучаем дерево по псевдоостатку r_{im} . Мы используем обозначение R_{jm} для обозначения $j = 1 \dots J_m$ листовых узлов результирующего дерева в итерации m .
- Для каждого листового узла R_{jm} вычисляем следующее выходное значение:

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma).$$

В следующем разделе мы углубимся в то, как значение γ_{jm} вычисляется путем минимизации функции потерь. А сейчас вы уже могли заметить, что листовые узлы R_{jm} могут содержать более одного обучающего примера, отсюда и суммирование.

- Обновляем модель, добавив выходные значения γ_m в предыдущее дерево:

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m.$$

Однако вместо добавления полных предсказанных значений текущего дерева γ_m к предыдущему дереву F_{m-1} мы умножаем γ_m на масштабный коэффициент скорости обучения η , который обычно является небольшим числом между 0.01 и 1. Другими словами, мы обновляем модель постепенно, делая небольшие шаги, что помогает избежать переобучения.

Разобравшись с внутренним устройством градиентного бустинга, мы применим этот механизм к задаче классификации.

7.5.3. Применение алгоритма градиентного бустинга для классификации

В этом разделе мы рассмотрим детали реализации алгоритма градиентного бустинга для задач бинарной классификации и воспользуемся логистической функцией потерь, которую представили для логистической регрессии в главе 3. Для одного обучающего примера мы можем записать логистическую потерю следующим образом:

$$L_i = -y_i \log p_i + (1 - y_i) \log(1 - p_i).$$

В главе 3 мы также представили функцию log(odds):

$$\hat{y} = \log(\text{odds}) = \log \left(\frac{p}{1 - p} \right).$$

По причинам, которые станут вам понятны позже, мы применим $\log(\text{odds})$, чтобы переписать логистическую функцию следующим образом (опустив промежуточные шаги):

$$L_i = \log(1 + e^{\hat{y}_i}) - y_i \hat{y}_i.$$

Теперь мы можем определить частную производную функции потерь по этим $\log(\text{odds})$, т. е. по \hat{y} . Производная этой функции потерь по $\log(\text{odds})$:

$$\frac{\partial L_i}{\partial \hat{y}_i} = \frac{e^{\hat{y}_i}}{1 + e^{\hat{y}_i}} - y_i = p_i - y_i.$$

На этом мы закончим с математическими определениями, вернемся к общим шагам градиентного бустинга с 1 по 2d из предыдущего раздела и сформулируем их заново для сценария бинарной классификации.

1. Создадим корневой узел, который минимизирует логистические потери. Оказывается, потери минимальны, если корневой узел возвращает $\log(\text{odds}) \hat{y}$.
2. Для каждого дерева $m = 1, \dots, M$, где M — указанное пользователем количество всех деревьев, мы выполняем следующие вычисления, описанные в шагах 2a–2d:
 - a. Преобразуем $\log(\text{odds})$ в вероятность с помощью знакомой вам логистической функции, которую мы использовали для логистической регрессии (в главе 3):

$$p = \frac{1}{1 + e^{-\hat{y}}}.$$

Затем вычисляем псевдоостаток, который представляет собой отрицательную частную производную потери по отношению к $\log(\text{odds})$ и является разницей между меткой класса и прогнозируемой вероятностью:

$$-\frac{\partial L_i}{\partial \hat{y}_i} = y_i - p_i.$$

- b. Обучаем новое дерево с псевдоостатками.
- c. Для каждого листового узла R_{jm} вычисляем значение γ_{jm} , которое минимизирует функцию логистических потерь. Вычисление включает в себя этап суммирования для обработки конечных узлов, которые содержат несколько обучающих примеров:

$$\begin{aligned} \gamma_{jm} &= \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, F_{m-1}(x_i) + \gamma) = \\ &= \log(1 + e^{\hat{y}_i + \gamma}) - y_i (\hat{y}_i + \gamma) \end{aligned}$$

Пропустив промежуточные математические преобразования, получаем следующее уравнение:

$$\gamma_{jm} = \frac{\sum_i y_i - p_i}{\sum_i p_i (i - p_i)}.$$

Обратите внимание, что суммирование здесь проводится только по примерам в узле, соответствующем листовому узлу R_{jm} , а не по полному обучающему набору.

- d. Обновляем модель, добавив значение гаммы из шага 2c с учетом скорости обучения η :

$$F_m(x) = F_{m-1}(x) + \eta \gamma_m.$$



Почему log(odds), а не вероятности?

Почему деревья возвращают значения log(odds), а не вероятности? Это связано с тем, что мы не можем просто сложить значения вероятности и получить осмысленный результат. (Так что с технической точки зрения градиентный бустинг для классификации использует деревья регрессии.)

В этом разделе мы взяли общий алгоритм повышения градиента и приспособили его для задачи бинарной классификации, заменив, например, общую функцию потерь логистическими потерями, а прогнозируемые значения — значениями log(odds). Однако для многих абстрактных шагов алгоритма может остаться неясной их связь с реальными задачами, и в следующем разделе мы применим эти шаги к конкретному примеру.

7.5.4. Иллюстрация применения градиентного бустинга для классификации

В предыдущих двух разделах мы кратко рассмотрели математические основы алгоритма градиентного бустинга применительно к задаче бинарной классификации. Чтобы сделать эти теоретические соображения более понятными, давайте применим их к небольшому демонстрационному примеру, т. е. к обучающему набору данных из трех записей, показанных на рис. 7.12.

	Признак x_1	Признак x_2	Метка класса y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

Рис. 7.12. Демонстрационный набор данных для иллюстрации применения градиентного бустинга

Мы начнем с *шага 1*, создав корневой узел и вычислив log(odds), и *шага 2a*, преобразовав log(odds) в вероятности принадлежности к классу и вычислив псевдоостатки. Как вы помните, в главе 3 было сказано, что шансы (odds) можно вычислить как количество успехов, деленное на количество неудач. Здесь мы рассматриваем метку 1 как успех, а метку 0 как неудачу, поэтому шансы рассчитываются так: odds = 2/1. Выполняя *шаги 1* и *2a*, получаем результаты, показанные на рис. 7.13.

Затем на *шаге 2b* мы обучаем новое дерево по псевдоостаткам r , а на *шаге 2c* вычисляем выходные значения y для этого дерева, как показано на рис. 7.14.

	Признак x_1	Признак x_2	Метка класса y	Шаг 1: $\hat{y} = \log(\text{odds})$	Шаг 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$	Шаг 2a: $r = y - p$
1	1.12	1.4	1	0.69	0.67	0.33
2	2.45	2.1	0	0.69	0.67	-0.67
3	3.54	1.2	1	0.69	0.67	0.33

Рис. 7.13. Результаты первого раунда применения *шага 1* и *шага 2a*

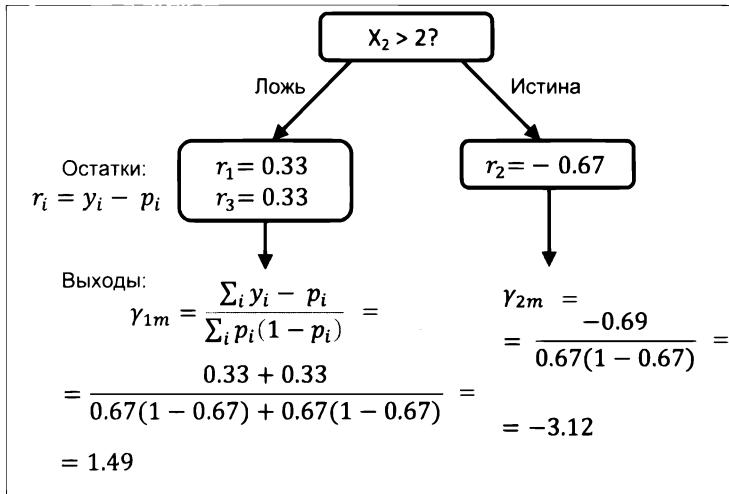


Рис. 7.14. Иллюстрация шагов 2b и 2c, которые подгоняют дерево к остаткам и вычисляют выходные значения для каждого конечного узла

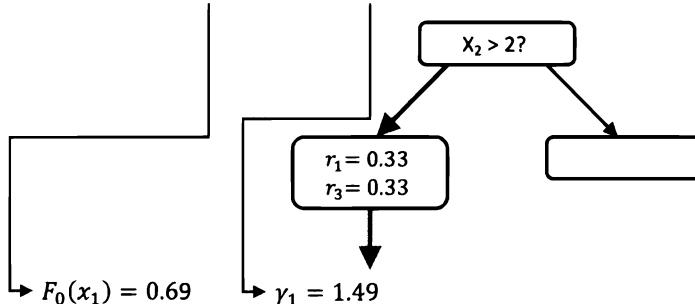
(Обратите внимание, что мы искусственно ограничили дерево только двумя конечными узлами для иллюстрации того, что происходит, если конечный узел содержит более одного примера.)

На последнем *шаге 2d* мы обновляем предыдущую и текущую модели. На рис. 7.15 показан прогноз для первого обучающего примера при скорости обучения $\eta = 0.1$.

→

	Признак x_1	Признак x_2	Метка класса y
1	1.12	1.4	1
2	2.45	2.1	0
3	3.54	1.2	1

Шаг 2d: $F_m(x) = F_{m-1}(x) + \eta \gamma_m$



$$F_1(x_1) = 0.69 + 0.1 \times 1.49 = 0.839$$

Рис. 7.15. Обновление предыдущей модели, показанное в контексте первого обучающего примера

Теперь, когда мы выполнили шаги с $2a$ по $2d$ первого раунда ($m = 1$), мы можем перейти к выполнению шагов с $2a$ по $2d$ для второго раунда ($m = 2$). Во втором раунде мы используем $\log(\text{odds})$, возвращенный обновленной моделью — например, $F_1(x_1) = 0.839$ в качестве входных данных для шага $2a$. Новые значения, полученные во втором раунде, показаны на рис. 7.16.

	x_1	x_2	y	Шаг 1: $F_0(x) = \hat{y} = \log(\text{odds})$	Шаг 2a: $p = \frac{1}{1 + e^{-\hat{y}}}$	Шаг 2a: $r = y - p$	Новый $\log(\text{odds})$ $\hat{y} = F_1(x)$	Шаг 2a: p	Шаг 2a: r
1	1.12	1.4	1	0.69	0.67	0.33	0.839	0.698	0.302
2	2.45	2.1	0	0.69	0.67	-0.67	0.378	0.593	-0.593
3	3.54	1.2	1	0.69	0.67	0.33	0.839	0.698	0.302

Раунд $m = 1$
Раунд $m = 2$

Рис. 7.16. Значения второго раунда рядом со значениями первого раунда

Здесь мы уже можем видеть, что предсказанные вероятности выше для положительного класса и ниже для отрицательного. Следовательно, и остатки становятся меньше. Заметьте, что шаги $2a$ – $2d$ повторяются до тех пор, пока мы не обучим M деревьев, или остаточные значения не станут меньше заданного пользователем порога. Затем, как только алгоритм градиентного бустинга завершится, мы можем использовать его для прогнозирования меток классов, устанавливая пороговые значения вероятности окончательной модели $F_M(x)$ на уровне 0.5 — как для логистической регрессии в главе 3. Однако, в отличие от логистической регрессии, градиентный бустинг состоит из нескольких деревьев и создает нелинейные границы решений. В следующем разделе вы испробуете градиентный бустинг в действии.

7.5.5. Использование XGBoost

Наконец-то мы изучили механизм градиентного бустинга до малейших деталей и можем перейти к практическому применению.

В библиотеке scikit-learn градиентный бустинг реализован как класс `sklearn.ensemble.GradientBoostingClassifier` (подробнее см. <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html>). Важно отметить, что градиентный бустинг — это последовательный процесс, который может быть медленным при обучении. Однако в последние годы появилась более популярная реализация градиентного бустинга, а именно — XGBoost.

XGBoost предлагает несколько приемов и приближений, которые существенно ускоряют процесс обучения. Отсюда и название XGBoost — eXtremal Gradient Boosting (экстремальный градиентный бустинг). Более того, эти приближения и приемы обеспечивают очень хорошие прогностические характеристики. XGBoost не случайно приобрел широкую популярность, поскольку это решение было победителем во многих соревнованиях Kaggle.

Помимо XGBoost, есть и другие популярные реализации градиентного бустинга — например, LightGBM и CatBoost. Основываясь на LightGBM, scikit-learn теперь также

реализует класс `HistGradientBoostingClassifier`, который более эффективен, чем исходный классификатор на основе градиентного бустинга (`GradientBoostingClassifier`).

Вы можете найти более подробную информацию об этих методах на следующих ресурсах:

- ◆ XGBoost: <https://xgboost.readthedocs.io/en/stable/>;
- ◆ LightGBM: <https://lightgbm.readthedocs.io/en/latest/>;
- ◆ CatBoost: <https://catboost.ai>;
- ◆ HistGradientBoostingClassifier:
<https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html>.

Однако, поскольку XGBoost по-прежнему остается одной из самых популярных реализаций градиентного бустинга, мы разберемся, как использовать его на практике. И начнем с его установки — например, через pip:

```
pip install xgboost
```



Установка XGBoost

В этой главе мы использовали XGBoost версии 1.5.0, которую можно установить с помощью команды:

```
pip install XGBoost==1.5.0
```

Более подробное руководство по установке XGBoost доступно по адресу:
<https://xgboost.readthedocs.io/en/stable/install.html>.

К счастью, `XGBClassifier` от XGBoost полностью соответствует API scikit-learn, поэтому использовать его относительно просто:

```
>>> import xgboost as xgb
>>> model = xgb.XGBClassifier(n_estimators=1000, learning_rate=0.01,
...                             max_depth=4, random_state=1,
...                             use_label_encoder=False)
>>> gbm = model.fit(X_train, y_train)
>>> y_train_pred = gbm.predict(X_train)
>>> y_test_pred = gbm.predict(X_test)
>>> gbm_train = accuracy_score(y_train, y_train_pred)
>>> gbm_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Точность XGBoost при обучении/тестировании '
...       f'{gbm_train:.3f}/{gbm_test:.3f}')
Точность XGBoost при обучении/тестировании 0.968/0.917
```

Здесь мы обучаем классификатор на основе градиентного бустинга с 1000 деревьев (рандов) и скоростью обучения 0.01. Обычно рекомендуется скорость обучения от 0.01 до 0.1, однако следует иметь в виду, что скорость обучения используется для масштабирования прогнозов по отдельным раундам. Интуитивно понятно, что чем ниже скорость обучения, тем больше оценок требуется для получения точных прогнозов.

Кроме того, для отдельных деревьев решений у нас есть параметр `max_depth`, для которого мы установили значение 4. Поскольку мы по-прежнему поощряем слабых учеников, значение от 2 до 6 является разумным выбором, но более высокие значения также могут работать хорошо в зависимости от набора данных.

Наконец, параметр `use_label_encoder=False` отключает предупреждающее сообщение, которое информирует пользователей о том, что XGBoost больше не конвертирует метки по умолчанию, и ожидает, что пользователи будут предоставлять метки в целочисленном формате, начиная с метки 0. (Здесь не о чём беспокоиться, поскольку мы придерживаемся этого формата на протяжении всей книги.)

На самом деле доступно гораздо больше настроек, и их детальное обсуждение выходит за рамки этой книги. Однако заинтересованные читатели могут найти более подробную информацию в исходной документации по адресу:

https://xgboost.readthedocs.io/en/latest/python/python_api.html#xgboost.XGBClassifier.

7.6. Заключение

В этой главе мы рассмотрели некоторые из наиболее популярных и широко используемых методов ансамблевого обучения. Ансамблевые методы объединяют различные классифицирующие модели, чтобы компенсировать их отдельные недостатки, что часто позволяет построить стабильные и хорошо работающие совокупные модели, которые очень привлекательны для применения на производстве, а также для соревнований по машинному обучению.

В начале этой главы мы реализовали класс `MajorityVoteClassifier` на чистом языке Python, что позволяет нам составлять разные алгоритмы классификации. Затем мы рассмотрели бэггинг — полезный прием для уменьшения дисперсии модели путем извлечения случайных бутстрэп-выборок из набора обучающих данных и объединения индивидуально обученных классификаторов по принципу большинства голосов (мажоритарного голосования). Наконец, было рассказано о бустинге в виде алгоритма AdaBoost и градиентного бустинга. Это алгоритмы, основанные на обучении слабых учеников, которые впоследствии учатся на ошибках.

В предыдущих главах вы многое узнали о различных алгоритмах обучения, настройке моделей и методах оценки. В следующей главе мы рассмотрим конкретное применение машинного обучения — анализ эмоциональной окраски текста, — которое стало вос требованной темой в эпоху Интернета и социальных сетей.

8

Применение машинного обучения для смыслового анализа текста

В нашу эпоху мнения, отзывы и рекомендации людей, высказанные ими в Интернете и социальных сетях, превратились в ценный ресурс для политтехнологов и предпринимателей. Благодаря современным технологиям мы теперь можем максимально эффективно собирать и анализировать такие данные. В этой главе мы займемся изучением одного из применений обработки естественного языка (Natural Language Processing, NLP) — так называемого анализа эмоциональной окраски текстов, и покажем, как использовать алгоритмы машинного обучения для классификации документов на основе эмоциональной тональности высказываний их авторов. В частности, мы проанализируем набор данных из 50 тыс. отзывов о фильмах из базы данных фильмов в Интернете (IMDb) и построим предиктор, который сможет различать положительные и отрицательные отзывы.

В этой главе будут рассмотрены следующие темы:

- ❖ очистка и подготовка текстовых данных;
- ❖ построение векторов признаков из текстовых документов;
- ❖ обучение модели, различающей положительные и отрицательные отзывы о фильмах;
- ❖ работа с большими наборами текстовых данных с использованием дополнительного обучения;
- ❖ извлечение тем из коллекций документов для последующей категоризации.

8.1. Подготовка набора данных с обзорами фильмов на IMDb

Как уже упоминалось, анализ эмоциональной окраски, иногда также называемый *анализом мнений*, или *анализом настроений*, является популярным прикладным применением более масштабных технологий NLP, — он связан со смысловым анализом документов. Популярной задачей в анализе настроений является классификация документов на основе мнений или эмоций авторов в отношении той или иной темы.

В этой главе мы будем работать с большим набором обзоров фильмов из базы IMDb, собранным Эндрю Маасом и его коллегами¹. Набор данных состоит из 50 тыс. обзоров

¹ См. «Learning Word Vectors for Sentiment Analysis» by A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng и C. Potts, «Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies», p. 142–150, Portland, Oregon, USA, Association for Computational Linguistics, June 2011.

фильмов с четко выраженным мнением авторов, помеченных как положительные или отрицательные, — здесь положительный обзор означает, что фильм получил более шести звезд на IMDb, а отрицательный — что фильм получил менее пяти звезд на IMDb. В следующих разделах мы загрузим этот набор данных, переведем его в формат, пригодный для инструментов машинного обучения, и извлечем значимую информацию из подмножества обзоров фильмов, чтобы построить модель машинного обучения, способную предсказать, понравится или не понравится фильм определенному обозревателю.

8.1.1. Получение набора данных с обзорами фильмов

Сжатый архив набора данных обзора фильмов (84.1 Мбайт) можно загрузить с сайта: <http://ai.stanford.edu/~amaas/data/sentiment/>, в виде TAR-архива, сжатого архиватором gzip:

- ◆ если вы работаете в среде Linux или macOS, откройте новое окно терминала, перейдите в каталог загрузки и выполните команду:

```
tar -zxf aclImdb_v1.tar.gz
```

чтобы распаковать набор данных;

- ◆ если вы работаете с Windows, воспользуйтесь любым бесплатным архиватором — например, 7-Zip (<http://www.7-zip.org>), для извлечения файлов из загруженного архива;
- ◆ кроме того, вы можете распаковать TAR-архив, сжатый gzip, непосредственно в Python следующим образом:

```
>>> import tarfile  
>>> with tarfile.open('aclImdb_v1.tar.gz', 'r:gz') as tar:  
...     tar.extractall()
```

8.1.2. Преобразование набора данных в более удобный формат

Распаковав архив с данными, необходимо собрать отдельные текстовые документы в один файл CSV. Следующий фрагмент кода Python считывает обзоры фильмов в объект DataFrame библиотеки pandas, что может занять до 10 минут на стандартном настольном компьютере.

Для визуализации хода выполнения и примерного подсчета времени до завершения воспользуемся пакетом Python Progress Indicator (PyPrind, <https://pypi.python.org/pypi/PyPrind/>), который был разработан несколько лет назад специально для этих целей. PyPrind можно установить, выполнив команду: pip install pyprind:

```
>>> import pyprind  
>>> import pandas as pd  
>>> import os  
>>> import sys  
>>> # измените параметр 'basepath' на имя каталога, в который распакован набор данных  
>>> basepath = 'aclImdb'
```

```
>>>
>>> labels = {'pos': 1, 'neg': 0}
>>> pbar = pyprind.ProgBar(50000, stream=sys.stdout)
>>> df = pd.DataFrame()
>>> for s in ('test', 'train'):
...     for l in ('pos', 'neg'):
...         path = os.path.join(basepath, s, l)
...         for file in sorted(os.listdir(path)):
...             with open(os.path.join(path, file),
...                       'r', encoding='utf-8') as infile:
...                 txt = infile.read()
...                 df = df.append([[txt, labels[l]]],
...                               ignore_index=True)
...             pbar.update()
>>> df.columns = ['review', 'sentiment']
0% 100%
[#####] | ETA: 00:00:00
Total time elapsed: 00:00:25
```

В этом блоке кода мы сначала инициализировали новый объект индикатора выполнения `pbar` значением 50 тыс. итераций, что соответствует количеству документов, которые мы намерены прочитать. Используя вложенные циклы `for`, мы прошлись по подкаталогам `train` и `test` в главном каталоге `aclImdb` и прочитали отдельные текстовые файлы из подкаталогов `pos` и `neg`, которые в конечном итоге добавили к набору данных `df` (`pandas DataFrame`) вместе с целочисленной меткой класса (1 = положительный и 0 = отрицательный).

Поскольку метки классов в собранном наборе данных упорядочены, необходимо перемешать `DataFrame` с помощью функции `permutation` из подмодуля `np.random` — это будет полезно для разделения исходного набора данных на обучающую и тестовую части в последующих разделах, когда мы будем брать данные со своего локального диска напрямую.

Для нашего же удобства сохраним собранный и перетасованный набор данных обзоров в виде CSV-файла:

```
>>> import numpy as np
>>> np.random.seed(0)
>>> df = df.reindex(np.random.permutation(df.index))
>>> df.to_csv('movie_data.csv', index=False, encoding='utf-8')
```

Перед тем как воспользоваться полученным набором данных, удостоверимся, что мы успешно сохранили данные в нужном формате, прочитав файл CSV и распечатав отрывок из первых трех примеров:

```
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')
>>> # для некоторых компьютеров необходимо переименовать столбец:
>>> df = df.rename(columns={"0": "review", "1": "sentiment"})
>>> df.head(3)
```

Если вы запускаете примеры кода в блокноте Jupyter, то увидите первые три примера набора данных (рис. 8.1).

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Рис. 8.1. Первые три строки набора данных с обзорами фильмов

В качестве следующей проверки удостоверимся, что DataFrame действительно содержит все 50 тыс. строк:

```
>>> df.shape
(50000, 2)
```

8.2. Знакомство с моделью мешка слов

Возможно, вы помните из главы 4, что прежде, чем передать алгоритму машинного обучения категориальные данные, такие как текст или слова, их следует преобразовать в числовую форму. В этом разделе вы познакомитесь с моделью *мешка слов* (bag-of-words), которая позволяет представлять текст в виде векторов числовых признаков. Идея мешка слов довольно проста и вкратце выглядит так:

1. Создаем словарь уникальных токенов — например, слов — из всего набора документов.
2. Для каждого документа строим вектор признаков, который содержит подсчеты того, как часто каждое слово встречается в конкретном документе.

Поскольку уникальные слова в каждом документе представляют собой лишь небольшое подмножество всех слов в словаре, векторы признаков будут в основном состоять из нулей. Такие векторы называются *разреженными*. Не волнуйтесь, если пока не все понятно — в следующих разделах мы шаг за шагом рассмотрим процесс создания простой модели мешка слов.

8.2.1. Преобразование слов в векторы признаков

Чтобы построить модель мешка слов на основе количества слов в соответствующих документах, воспользуемся классом `CountVectorizer`, реализованным в `scikit-learn`. Как вы увидите в следующем фрагменте кода, `CountVectorizer` берет массив текстовых данных, которые могут быть документами или предложениями, и строит для нас модель мешка слов:

```
>>> import numpy as np
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer()
>>> docs = np.array(['The sun is shining',
...                 'The weather is sweet',
...                 'The sun is shining, the weather is sweet,'
...                 'and one and one is two'])
>>> bag = count.fit_transform(docs)
```

Вызвав метод `fit_transform` класса `CountVectorizer`, мы построили словарь модели мешка слов и преобразовали в разреженные векторы признаков следующие три предложения:

- ◆ 'The sun is shining' (Солнце светит);
- ◆ 'The weather is sweet' (Погода прекрасная);
- ◆ 'The sun is shining, the weather is sweet, and one and one is two' (Солнце светит, погода прекрасная, а один плюс один будет два).

Выведем на печать содержание словаря, чтобы проиллюстрировать основные понятия мешка слов:

```
>>> print(count.vocabulary_)
{'and': 0,
'two': 7,
'shining': 3,
'one': 2,
'sun': 4,
'weather': 8,
'the': 6,
'sweet': 5,
'is': 1}
```

Как видно из результата выполнения предыдущей команды, лексический словарь (вocabulary) хранится в словаре Python, который сопоставляет уникальные слова с целочисленными индексами. Теперь выведем только что созданные векторы признаков:

```
>>> print(bag.toarray())
[[0 1 0 1 1 0 1 0 0]
 [0 1 0 0 0 1 1 0 1]
 [2 3 2 1 1 1 2 1 1]]
```

Каждая позиция индекса в векторах признаков, показанных здесь, соответствует целочисленным значениям, которые хранятся как элементы словаря в словаре `CountVectorizer`. Например, первый признак в позиции индекса 0 отражает частотность слова 'and', которое встречается только в последнем предложении, тогда как слово 'is' в позиции индекса 1 (второй признак в векторах признаков) встречается во всех трех предложениях. Эти значения в векторах признаков также называются *необработанными частотами терминов* (raw term frequencies): $tf(t, d)$ — количество раз, когда термин t встречается в документе d . Следует отметить, что в модели мешка слов порядок слов или терминов в предложении или документе не имеет значения. Порядок, в котором частотности терминов появляются в векторе признаков, определяется индексами словаря, которые обычно присваиваются в алфавитном порядке.



N-граммные языковые модели

Последовательность элементов в модели мешка слов, которую мы только что создали, также называется 1-граммной моделью, или *униграммой*, — каждый элемент или токен в словаре представляет одно слово. В более общем смысле непрерывные последовательности элементов NLP — слов, букв или символов — также называются *n*-граммами. Выбор числа n в модели *n*-грамм зависит от конкретного приложения. Например, исследование, проведенное Иоаннисом Канарисом и его коллегами, показало, что *n*-граммы размера 3 и 4 дают хорошие результаты при фильтра-

ции сообщений электронной почты от спама (см. «Words versus character n-grams for anti-spam filtering» by Ioannis Kanaris, Konstantinos Kanaris, Ioannis Houvardas, and Efstathios Stamatatos, International Journal on Artificial Intelligence Tools, World Scientific Publishing Company, 16(06): 1047–1067, 2007).

В соответствии с концепцией n -граммного представления 1-граммное и 2-граммные представления нашего первого предложения «the sun is shining» будут построены следующим образом:

- 1-грамма: «the», «sun», «is», «shining»;
- 2-грамма: «the sun», «sun is», «is shining».

Класс `CountVectorizer` в scikit-learn позволяет нам использовать разные модели n -грамм с помощью параметра `ngram_range`. Хотя по умолчанию $n = 1$, можно переключиться на 2-граммму, инициализировав новый экземпляр `CountVectorizer` с параметром `ngram_range=(2, 2)`.

8.2.2. Оценка релевантности слов с помощью частоты термина и обратной частоты документа

Анализируя текстовые данные, мы часто сталкиваемся со словами, которые встречаются в нескольких документах обоих классов. Эти часто встречающиеся слова обычно не содержат полезной информации, способствующей различению документов. Существует полезный метод под названием *обратная частота документа* (Term Frequency-Inverse Document Frequency, TF-IDF), который можно использовать для понижения веса таких часто встречающихся слов в векторах признаков. Показатель TF-IDF можно определить как произведение частоты термина и обратной частоты документа:

$$tf\text{-}idf(t, d) = tf(t, d) \times idf(t, d).$$

Здесь $tf(t, d)$ — частота термина, о которой мы говорили в предыдущем разделе, а $idf(t, d)$ — обратная частота документа, которую можно рассчитать следующим образом:

$$idf(t, d) = \log \frac{n_d}{1 + df(d, t)}.$$

Здесь n_d — общее количество документов, а $df(d, t)$ — количество документов d , содержащих термин t . Добавление константы 1 к знаменателю не обязательно и служит лишь для присвоения ненулевого значения терминам, которые не встречаются ни в одном из обучающих примеров. Логарифм применен с тем, чтобы низкие частоты документов не получили слишком большой вес.

Библиотека scikit-learn реализует еще один преобразователь — класс `TfidfTransformer`, который берет необработанные частоты терминов из класса `CountVectorizer` в качестве входных данных и преобразует их в TF-IDF:

```
>>> from sklearn.feature_extraction.text import TfidfTransformer
>>> tfidf = TfidfTransformer(use_idf=True,
...                           norm='l2',
...                           smooth_idf=True)
>>> np.set_printoptions(precision=2)
>>> print(tfidf.fit_transform(count.fit_transform(docs)))
...     .toarray()
```

```
[ [ 0.    0.43  0.    0.56  0.56  0.    0.43  0.    0.   ]
[ 0.    0.43  0.    0.    0.    0.56  0.43  0.    0.56]
[ 0.5   0.45  0.5   0.19  0.19  0.19  0.3   0.25  0.19]]
```

Как вы видели в предыдущем разделе, слово 'is' имеет наивысшую частотность в третьем предложении, поскольку там наиболее часто встречается. Однако после преобразования того же вектора признаков в TF-IDF слово 'is' теперь связано в третьем предложении с относительно небольшим весом (0.45), поскольку оно также присутствует и в первом, и во втором предложении и, следовательно, вряд ли может содержать какую-либо информацию, отражающую различия предложений.

Тем не менее если мы вручную вычислим показатель TF-IDF отдельных терминов в наших векторах признаков, то заметим, что `TfidfTransformer` вычисляет его немного иначе, чем стандартные уравнения, которые мы привели в этой книге ранее. Уравнение для обратной частоты документов, реализованное в `scikit-learn`, выглядит следующим образом:

$$idf(t, d) = \log \frac{1 + n_d}{1 + df(d, t)}.$$

Точно так же показатель TF-IDF, вычисленный в `scikit-learn`, немного отличается от уравнения по умолчанию, которое мы определили ранее, и выглядит так:

$$tf-idf(t, d) = tf(t, d) \times (idf(t, d) + 1).$$

Наличие слагаемого «+1» в этом уравнении связано со значением параметра `smooth_idf=True` в предыдущем примере кода и позволяет назначать нулевой вес (т. е. $idf(t, d) = \log(1) = 0$) терминам, которые встречаются во всех документах.

Хотя обычно принято нормализовать частоты необработанных терминов перед вычислением TF-IDF, класс `TfidfTransformer` нормализует TF-IDF напрямую. По умолчанию (параметр `norm='l2'`) `TfidfTransformer` применяет нормализацию L2, которая возвращает вектор длины 1 путем деления ненормализованного вектора признаков v на его норму L2:

$$v_{norm} = \frac{v}{\|v\|_2} = \frac{v}{\sqrt{v_1^2 + v_2^2 + \dots + v_n^2}} = \frac{v}{(\sum_{i=1}^n v_i^2)^{1/2}}.$$

Давайте закрепим понимание принципов работы класса `TfidfTransformer` на примере и вычислим TF-IDF слова 'is' в третьем предложении. Это слово в третьем предложении имеет частоту термина 3 ($tf = 3$), и частота этого термина в целом тоже равна 3, поскольку слово 'is' встречается во всех трех предложениях ($df = 3$). Теперь вычислим обратную частоту документа:

$$idf("is", d_3) = \log \frac{1 + 3}{1 + 3} = 0.$$

Чтобы найти TF-IDF, нам просто нужно добавить 1 к обратной частоте документа и умножить ее на частоту термина:

$$tf-idf("is", d_3) = 3 \times (0 + 1) = 3.$$

Если мы повторим этот расчет для всех терминов в третьем предложении, то получим следующие векторы: [3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]. Однако обратите внимание, что значения в этом векторе признаков отличаются от значений, которые

нам ранее вернул `TfidfTransformer`. Последний шаг, который нам осталось сделать, — это L2-нормализация, которую можно применить следующим образом:

$$\begin{aligned} tf\text{-}idf(d_3)_{norm} &= \frac{[3.39, 3.0, 3.39, 1.29, 1.29, 1.29, 2.0, 1.69, 1.29]}{\sqrt{3.39^2 + 3.0^2 + 3.39^2 + 1.29^2 + 1.29^2 + 1.29^2 + 2.0^2 + 1.69^2 + 1.29^2}} = \\ &= [0.5, 0.45, 0.5, 0.19, 0.19, 0.19, 0.3, 0.25, 0.19] \end{aligned}$$

$$tf\text{-}idf("is", d_3) = 0.45$$

Как видите, теперь результат наших вычислений совпадает с результатом работы класса `TfidfTransformer` от `scikit-learn`. Мы успешно закрепили правила расчета TF-IDF на практике и можем смело перейти к следующему разделу и приступить к обработке данных обзоров фильмов.

8.2.3. Очистка текстовых данных

Итак, вы уже знаете, что такое модель мешка слов, частота терминов и TF-IDF. Однако прежде, чем построить модель мешка слов, нужно сделать один важный шаг — очистить текстовые данные, удалив из них все нежелательные символы.

Чтобы убедиться в необходимости этого шага, отобразим последние 50 символов из первого документа в перетасованном наборе данных обзора фильмов:

```
>>> df.loc[0, 'review'][-50:]
'is seven.<br /><br />Title (Brazil): Not Available'
```

В выведенном фрагменте текста мы видим HTML-разметку, а также знаки препинания и другие небуквенные символы. Хотя HTML-разметка почти не содержит полезной семантики, знаки препинания в определенных случаях могут предоставлять полезную дополнительную информацию. Однако сейчас для простоты мы удалим все знаки препинания, кроме символов смайликов, таких как «:)», поскольку они, безусловно, полезны для анализа эмоциональной окраски.

Справиться с этой задачей нам поможет библиотека регулярных выражений (`regex`) Python с лаконичным названием `re`:

```
>>> import re
>>> def preprocessor(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall('(:-\\.|:-)|(-:\\.)|\\(|\\)D|P|',
...                           text)
...     text = (re.sub('\\W+', ' ', text.lower()) +
...             ' '.join(emoticons).replace('-', ''))
...     return text
```

В этом блоке кода с помощью первого регулярного выражения `<[^>]*>` мы пытаемся удалить из обзоров фильмов всю HTML-разметку. Хотя многие программисты не рекомендуют использовать регулярное выражение для анализа HTML, такого регулярного выражения достаточно для очистки этого конкретного набора данных. Поскольку мы заинтересованы лишь в удалении HTML-разметки и не планируем использовать ее в дальнейшем, простого регулярного выражения будет достаточно. Впрочем, если вы

предпочитаете более сложные инструменты для удаления HTML-разметки из текста, обратите внимание на модуль HTML-парсера Python, описание которого приведено на странице: <https://docs.python.org/3/library/html.parser.html>.

Удалив HTML-разметку, мы использовали более сложное регулярное выражение для поиска смайликов, которые мы решили временно сохранить. Затем с помощью регулярного выражения `(\w)+` мы удалили из текста все символы, не являющиеся словами, и преобразовали текст в символы нижнего регистра.



Работа с заглавными буквами

В этом конкретном примере анализа мы предполагаем, что заглавная буква слова — например, в начале предложения — не содержит семантически релевантной информации. Это допущение не всегда справедливо — например, так мы теряем обозначения имен собственных. Но, опять же, в контексте нашего анализа предположение о том, что регистр букв не содержит информации о тональности текста, является допустимым упрощением.

В заключение мы добавили временно сохраненные смайлики в конец обрабатываемой строки документа. Кроме того, для единства мы удалили из смайликов символ «носа» (например, дефис в смайлике `:-)`), потому что иногда встречаются смайлики без этого символа.



Регулярные выражения

Хотя регулярные выражения представляют собой эффективный и удобный подход к поиску символов в строке, для их использования нужны серьезные знания и навыки. К сожалению, подробное обсуждение регулярных выражений выходит за рамки этой книги. Однако вы можете найти отличное руководство на портале Google Developers по адресу: <https://developers.google.com/edu/python/regular-expressions>, или ознакомиться с официальной документацией модуля `re` из состава Python по адресу: <https://docs.python.org/3.9/library/re.html>.

Добавление символов смайликов в конец очищенных строк документа может показаться не самым элегантным подходом, но здесь не следует забывать, что порядок слов в нашей модели не имеет значения, если словарь состоит только из однословных токенов. Но прежде чем подробнее обсудить разбиение документов на отдельные термины, слова или токены, давайте убедимся в правильной работе функции `preprocessor`:

```
>>> preprocessor(df.loc[0, 'review'][-50:])
'is seven title brazil not available'
>>> preprocessor("</a>This :) is :( a test :-)!")
'this is a test :) :( :)'
```

Наконец, поскольку мы будем часто использовать очищенные текстовые данные в следующих разделах, применим функцию `preprocessor` ко всем обзорам фильмов в `DataFrame`:

```
>>> df['review'] = df['review'].apply(preprocessor)
```

8.2.4. Получение токенов из документов

К этому моменту мы выполнили предварительную обработку набора данных с обзорами фильмов, и теперь нужно подумать о том, как разделить текстовый корпус на отдельные элементы (токены). Один из способов *токенизации* документов — разбить их на отдельные слова, разделив очищенные документы по символам пробела:

```
>>> def tokenizer(text):
...     return text.split()
>>> tokenizer('runners like running and thus they run')
['runners', 'like', 'running', 'and', 'thus', 'they', 'run']
```

В контексте токенизации другой полезный метод — *определение основы слова* (или *стемминг* от англ. *stemming*), т. е. процесс преобразования слова в его корневую морфему. Стемминг позволяет нам сопоставлять родственные слова с одной и той же основой. Первоначальный алгоритм стемминга был разработан в 1979 году Мартином Портером и поэтому известен как *алгоритм Портера*². В наборе инструментов Natural Language Toolkit (NLTK, <http://www.nltk.org>) для Python реализован алгоритм стемминга Портера, которым мы и воспользуемся в следующем примере кода. Чтобы установить NLTK, вы можете просто выполнить команду `conda install nltk` или `pip install nltk`.



Онлайн-документация по NLTK

Хотя NLTK не рассматривается в этой главе, я настоятельно рекомендую вам посетить веб-сайт NLTK, а также прочитать официальную документацию NLTK, которая находится в свободном доступе по адресу: <http://www.nltk.org/book/>, если вас интересуют более продвинутые приложения NLP.

В следующем коде показан пример использования алгоритма Портера:

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter = PorterStemmer()
>>> def tokenizer_porter(text):
...     return [porter.stem(word) for word in text.split()]
>>> tokenizer_porter('runners like running and thus they run')
['runner', 'like', 'run', 'and', 'thu', 'they', 'run']
```

Использовав `PorterStemmer` из пакета `nltk`, мы модифицировали нашу функцию `tokenizer`, чтобы привести слова к их корневой морфеме, что хорошо видно на примере слова *'running'* (бегущий), которое было преобразовано в морфему *'run'* (бег).



Алгоритмы стемминга

Алгоритм стемминга Портера, вероятно, самый старый и самый простой из подобных алгоритмов. Другими популярными алгоритмами стемминга являются более новый стеммер Snowball (Porter2, или английский стеммер) и стеммер Lancaster (стеммер Paice/Husk). Хотя стеммеры Snowball и Lancaster работают быстрее, чем исходный Porter, стеммер Lancaster также более агрессивен, чем Porter, а это означает, что он будет производить более короткие и неясные слова. Эти альтернатив-

² См. «An algorithm for suffix stripping» by Martin F. Porter, Program: Electronic Library and Information Systems, 14(3): 130–137, 1980.

ные алгоритмы стемминга также доступны в пакете NLTK (<http://www.nltk.org/api/nltk.stem.html>).

Стемминг может создавать несуществующие слова (например, 'thu' от слова 'thus'), как показано в предыдущем примере. Альтернативный метод, называемый лемматизацией, направлен на получение канонических (грамматически правильных) форм отдельных слов — так называемых лемм. Однако лемматизация является более сложной и дорогостоящей операцией в вычислительном отношении по сравнению со стеммингом, и на практике было замечено, что переход от стемминга к лемматизации почти не повышает эффективность классификации текста (см. «Influence of Word Normalization on Text Classification», by Michal Toman, Roman Tesar, and Karel Jezek, Proceedings of InSciT, p. 354–358, 2006).

Прежде чем перейти непосредственно к обучению модели с использованием метода мешка слов, кратко коснемся другой полезной темы, называемой *удалением стоп-слов*. Стоп-слова — это всего лишь слова, которые чрезвычайно распространены во всех видах текстов и, вероятно, не несут (или содержат лишь немного) полезной информации, пригодной для различения различных классов документов. Примеры стоп-слов в английском языке: *is, has, and, like*. Удаление стоп-слов особенно полезно, если мы работаем с необработанными или нормализованными частотами терминов, а не с показателями TF-IDF, которые существенно снижают вес часто встречающихся слов.

Для удаления стоп-слов из кинообзоров воспользуемся набором из 127 английских стоп-слов, доступных в библиотеке nltk, которые можно получить, вызвав функцию `nltk.download`:

```
>>> import nltk
>>> nltk.download('stopwords')
```

После загрузки набора стоп-слов мы можем загрузить и применить набор стоп-слов английского языка следующим образом:

```
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> [w for w in tokenizer_porter('a runner likes'
...   ' running and runs a lot')
...   if w not in stop]
['runner', 'like', 'run', 'run', 'lot']
```

8.3. Обучение модели логистической регрессии для классификации документов

Здесь мы обучим модель логистической регрессии, чтобы классифицировать обзоры фильмов на две категории: положительные и отрицательные — на основе модели мешка слов. Сначала разделим DataFrame очищенных текстовых документов на 25 тыс. документов для обучения и 25 тыс. документов для тестирования:

```
>>> X_train = df.loc[:25000, 'review'].values
>>> y_train = df.loc[:25000, 'sentiment'].values
>>> X_test = df.loc[25000:, 'review'].values
>>> y_test = df.loc[25000:, 'sentiment'].values
```

Далее воспользуемся объектом GridSearchCV, чтобы найти оптимальный набор параметров для нашей модели логистической регрессии с использованием 5-кратной стратифицированной перекрестной проверки:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.pipeline import Pipeline
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> tfidf = TfidfVectorizer(strip_accents=None,
...                         lowercase=False,
...                         preprocessor=None)
>>> small_param_grid = [
...     {
...         'vect__ngram_range': [(1, 1)],
...         'vect__stop_words': [None],
...         'vect__tokenizer': [tokenizer, tokenizer_porter],
...         'clf__penalty': ['l2'],
...         'clf__C': [1.0, 10.0]
...     },
...     {
...         'vect__ngram_range': [(1, 1)],
...         'vect__stop_words': [stop, None],
...         'vect__tokenizer': [tokenizer],
...         'vect__use_idf':[False],
...         'vect__norm':[None],
...         'clf__penalty': ['l2'],
...         'clf__C': [1.0, 10.0]
...     },
... ]
>>> lr_tfidf = Pipeline([
...     ('vect', tfidf),
...     ('clf', LogisticRegression(solver='liblinear'))
... ])
>>> gs_lr_tfidf = GridSearchCV(lr_tfidf, small_param_grid,
...                             scoring='accuracy', cv=5,
...                             verbose=2, n_jobs=1)
>>> gs_lr_tfidf.fit(X_train, y_train)
```

Обратите внимание, что для классификатора логистической регрессии мы используем решатель liblinear, поскольку с относительно большими наборами данных он может работать лучше, чем выбор по умолчанию ('lbfgs').



Включение многопроцессорного режима

Мы настоятельно рекомендуем установить значение параметра `n_jobs=-1` (вместо `n_jobs=1`, как в приведенном примере кода), чтобы использовать все доступные ядра процессора на вашем компьютере и ускорить поиск по сетке. Однако некоторые пользователи Windows сообщали о проблемах при запуске приведенного кода с параметром `n_jobs=-1`, связанных с выбором функций `tokenizer` и `tokenizer_porter` при многопроцессорном выполнении в среде Windows. В таком случае обходным решением может быть замена функций `tokenizer` и `tokenizer_porter` на `str.split`. Но учтите, что такая замена на простой `str.split` не поддерживает стемминг.

При инициализации объекта `GridSearchCV` и его сетки параметров с помощью приведенного кода нам пришлось ограничить количество комбинаций параметров, поскольку большое количество векторов признаков, а также большой словарь, могут сделать поиск по сетке довольно затратным в вычислительном отношении. На стандартном настольном компьютере поиск по сетке может занять 5–10 минут.

В приведенном примере кода мы заменили `CountVectorizer` и `TfidfTransformer` на `TfidfVectorizer`, который фактически объединяет в себе эти два класса. Наш `param_grid` состоит из двух словарей параметров. В первом словаре мы использовали для вычисления TF-IDF объект `TfidfVectorizer` с настройками по умолчанию (`use_idf=True`, `smooth_idf=True` и `norm='l2'`), а во втором словаре установили для этих параметров значения `use_idf=False`, `smooth_idf=False` и `norm=None`, чтобы обучить модель на основе необработанных частот терминов. Кроме того, для классификатора логистической регрессии мы обучили модели, задав регуляризацию L2 с помощью параметра `clf_penalty=12`, и сравнили эффекты от различной степени регуляризации, определив диапазон значений для параметра обратной регуляризации С. В качестве дополнительного упражнения попробуйте добавить в параметры поиска по сетке регуляризацию L1, изменив `'clf_penalty': ['l2']` на `'clf_penalty': ['l2', 'l1']`.

После завершения поиска по сетке выведем лучший набор параметров:

```
>>> print(f'Лучший набор параметров: {gs_lr_tfidf.best_params_}')
Лучший набор параметров: {'clf_C': 10.0, 'clf_penalty': 'l2', 'vect_ngram_range':
(1, 1), 'vect_stop_words': None, 'vect_tokenizer': <function tokenizer at
0x169932dc0>}
```

Как видите, мы получили наилучшие результаты поиска по сетке, используя обычный `tokenizer` без стемминга Портера, без библиотеки стоп-слов и TF-IDF в сочетании с классификатором логистической регрессии, который использует регуляризацию L2 со степенью регуляризации `C=10.0`.

Применив лучшую модель в соответствии с результатом поиска по сетке, выведем средние значения оценки точности путем 5-кратной перекрестной проверки на наборе обучающих данных и точности классификации на наборе тестовых данных:

```
>>> print(f'Точность CV: {gs_lr_tfidf.best_score_:.3f}')
Точность CV: 0.897
>>> clf = gs_lr_tfidf.best_estimator_
>>> print(f'Точность на тестовом наборе: {clf.score(X_test, y_test):.3f}')
Точность на тестовом наборе: 0.899
```

Результаты показывают, что наша модель машинного обучения способна предсказать характер отзыва (положительный или отрицательный) с точностью 90%.



Наивный байесовский классификатор

По-прежнему очень распространенной моделью для классификации текста является наивный байесовский классификатор, который приобрел популярность в приложениях для фильтрации спама в электронной почте. Наивные байесовские классификаторы просты в реализации, эффективны в вычислительном отношении и, как правило, особенно хорошо работают с относительно небольшими наборами данных по сравнению с другими алгоритмами. Хотя мы не обсуждаем наивные байесовские классификаторы в этой книге, заинтересованные читатели может прочесть статью

о наивной байесовской классификации текста, которая находится в свободном доступе на arXiv («Naive Bayes and Text Classification I – Introduction and Theory» by S. Raschka, Computing Research Repository (CoRR), abs/1410.5329, 2014, <http://arxiv.org/pdf/1410.5329v3.pdf>). Различные версии наивных байесовских классификаторов, упомянутые в этой статье, реализованы в библиотеке scikit-learn. Вы можете найти обзорную страницу со ссылками на соответствующие классы кода по адресу: https://scikit-learn.org/stable/modules/naive_bayes.html.

8.4. Работа с большими данными: онлайн-алгоритмы и внешнее обучение

Если вы запускали примеры кода из предыдущего раздела, то могли заметить, что построение векторов признаков для 50 тыс. обзоров фильмов во время поиска по сетке требует больших вычислительных затрат. Во многих реальных приложениях нередко приходится работать с еще более объемными наборами данных, которые могут превышать объем памяти компьютера.

Поскольку не у всех есть доступ к суперкомпьютерам, мы научимся применять технологию так называемого *внешнего обучения* (out-of-core learning) — обучение на внешних данных, не умещающихся в память, которая позволяет нам работать с громоздкими наборами данных путем постепенного обучения классификатора на меньших фрагментах набора.



Классификация текста с помощью рекуррентных нейросетей

В главе 15 мы вернемся к этому набору данных и обучим классификатор на основе глубокого обучения (рекуррентную нейронную сеть) для различения эмоциональной окраски обзоров фильмов в наборе IMDb. Этот классификатор на основе нейронной сети следует тому же принципу внешнего обучения с использованием алгоритма оптимизации методом стохастического градиентного спуска, но не требует построения модели мешка слов.

Возможно, вы помните, что еще в главе 2 была введена концепция стохастического градиентного спуска — алгоритма оптимизации, обновляющего веса модели, используя один пример за раз. В этом разделе мы применим функцию `partial_fit` `SGDClassifier` библиотеки scikit-learn для потоковой передачи документов непосредственно с нашего локального диска и обучения модели логистической регрессии с использованием небольших мини-пакетов документов.

Сначала определим функцию `tokenizer`, которая очищает необработанные текстовые данные из файла `movie_data.csv`, созданного в начале этой главы, и разбивает их на однословные токены, попутно удаляя стоп-слова:

```
>>> import numpy as np
>>> import re
>>> from nltk.corpus import stopwords
>>> stop = stopwords.words('english')
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', ' ', text)
```

```

...
    emoticons = re.findall('(?:.;|=)(?:-)?(?:\\)|\\(|D|P)',
                           text)
...
    text = re.sub('[\\W]+', ' ', text.lower()) \
            + ' '.join(emoticons).replace('-', '')
...
    tokenized = [w for w in text.split() if w not in stop]
...
    return tokenized

```

Затем определим функцию генератора `stream_docs`, которая считывает и возвращает один документ за раз:

```

>>> def stream_docs(path):
...     with open(path, 'r', encoding='utf-8') as csv:
...         next(csv) # пропуск заголовка
...         for line in csv:
...             text, label = line[:-3], int(line[-2])
...             yield text, label

```

Чтобы убедиться, что наша функция `stream_docs` работает правильно, давайте прочитаем первый документ из файла `movie_data.csv`, который должен вернуть кортеж, состоящий из текста обзора, а также соответствующей метки класса:

```

>>> next(stream_docs(path='movie_data.csv'))
('In 1974, the teenager Martha Moxley ... ', 1)

```

Теперь определим функцию `get_minibatch`, которая будет получать поток документов из функции `stream_docs` и возвращать нужное количество документов, заданное параметром `size`:

```

>>> def get_minibatch(doc_stream, size):
...     docs, y = [], []
...     try:
...         for _ in range(size):
...             text, label = next(doc_stream)
...             docs.append(text)
...             y.append(label)
...     except StopIteration:
...         return None, None
...     return docs, y

```

К сожалению, мы не можем использовать объект `CountVectorizer` для внешнего обучения, поскольку он должен хранить в памяти весь словарный запас. Кроме того, `TfidfVectorizer` хранит в памяти все векторы признаков обучающего набора данных для вычисления обратных частот документа. Однако в `scikit-learn` реализован еще один удобный векторизатор под названием `HashingVectorizer`. Он не хранит все данные в памяти и использует прием хеширования с помощью 32-битной функции `MurmurHash3` от Остина Эпплби³:

```

>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> from sklearn.linear_model import SGDClassifier

```

³ Дополнительную информацию о `MurmurHash` можно найти на странице: <https://en.wikipedia.org/wiki/MurmurHash>.

```
>>> vect = HashingVectorizer(decode_error='ignore',
...                           n_features=2**21,
...                           preprocessor=None,
...                           tokenizer=tokenizer)
>>> clf = SGDClassifier(loss='log', random_state=1)
>>> doc_stream = stream_docs(path='movie_data.csv')
```

Выполнив приведенный код, мы инициализировали HashingVectorizer с помощью функции tokenizer и установили количество признаков равным 2^{21} . Кроме того, мы повторно инициализировали классификатор логистической регрессии, установив для параметра потерю SGDClassifier значение 'log'. Заметьте, что, выбирая большое количество признаков в HashingVectorizer, мы уменьшаем вероятность возникновения коллизий хешей, но при этом увеличиваем количество коэффициентов в нашей модели логистической регрессии.

Наконец-то начинается действительно интересная часть — настроив все вспомогательные функции, мы можем начать обучение на внешних данных с помощью следующего кода:

```
>>> import pyprind
>>> pbar = pyprind.ProgBar(45)
>>> classes = np.array([0, 1])
>>> for _ in range(45):
...     X_train, y_train = get_minibatch(doc_stream, size=1000)
...     if not X_train:
...         break
...     X_train = vect.transform(X_train)
...     clf.partial_fit(X_train, y_train, classes=classes)
...     pbar.update()
0%                                100%
[########################################] | ETA: 00:00:00
Total time elapsed: 00:00:21
```

Для оценки прогресса обучения мы снова использовали пакет PyPrind. Мы инициализировали объект индикатора выполнения, содержащий 45 делений, а в следующем цикле for перебрали более 45 мини-пакетов документов, где каждый мини-пакет состоит из 1000 документов. Завершив процесс поэтапного обучения, мы задействуем последние 5000 документов для оценки точности нашей модели:

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
>>> X_test = vect.transform(X_test)
>>> print(f'Точность: {clf.score(X_test, y_test):.3f}')
Точность: 0.868
```



Ошибка NoneType

Если вы столкнулись с ошибкой NoneType, возможно, вы дважды выполнили код `X_test, y_test = get_minibatch(...)`. В предыдущем цикле у нас есть 45 итераций, каждая из которых извлекает 1000 документов. Таким образом, на проверку остается ровно 5000 документов, которые мы выделяем в строке

```
>>> X_test, y_test = get_minibatch(doc_stream, size=5000)
```

Если мы выполним этот код дважды, то в генераторе останется недостаточно документов и `X_test` вернет `None` (ничего нет). Следовательно, если вы столкнетесь с ошибкой `NoneType`, вам придется начать заново с предыдущего кода `stream_docs(...)`.

Как видите, точность модели составляет примерно 87%, что немного ниже точности, которой мы достигли в предыдущем разделе, используя поиск по сетке для настройки гиперпараметров. Однако обучение на внешних данных очень эффективно использует память, и его выполнение заняло менее минуты.

Наконец, задействуем оставшиеся 5000 документов для обновления модели:

```
>>> clf = clf.partial_fit(X_test, y_test)
```



Модель word2vec

Более современной альтернативой модели мешка слов является алгоритм word2vec, выпущенный Google в 2013 г. (см. «Efficient Estimation of Word Representations in Vector Space» by T. Mikolov, K. Chen, G. Corrado, and J. Dean, <https://arxiv.org/abs/1301.3781>).

Алгоритм word2vec — это алгоритм обучения без учителя на основе нейронной сети, который пытается автоматически изучить взаимосвязь между словами. Идея word2vec состоит в том, чтобы поместить слова, имеющие сходное значение, в похожие кластеры, после чего, опираясь на это логично выстроенное пространство смыслов, модель может воспроизводить определенные слова, используя простую векторную математику, например: король – мужчина + женщина = королева.

Исходную реализацию на языке C с полезными ссылками на соответствующие документы и альтернативные реализации можно найти по адресу: <https://code.google.com/p/word2vec/>.

8.5. Моделирование тем с использованием скрытого распределения Дирихле

Моделирование тем (topic modeling) охватывает широкий перечень задач назначения тем немаркированным текстовым документам. Например, типичным применением является категоризация документов в большом текстовом корпусе газетных статей. В приложениях моделирования тем мы стараемся присвоить газетным статьям метки категорий — например: спорт, финансы, мировые новости, политика и местные новости. Таким образом, с точки зрения широких категорий машинного обучения, которые мы обсуждали в главе 1, моделирование тем можно рассматривать как задачу кластеризации, т. е. разновидность обучения без учителя.

В этом разделе мы обсудим популярную технику моделирования тем, называемую *скрытым распределением Дирихле* (Latent Dirichlet Allocation, LDA). Однако обратите внимание, что хотя для скрытого распределения Дирихле в источниках часто используют аббревиатуру LDA, его не следует путать с обозначаемым той же аббревиатурой линейным дискриминантным анализом (Linear Discriminant Analysis, LDA) — методом контролируемого уменьшения размерности, который был представлен в главе 5.

8.5.1. Разбор текстовых документов с помощью LDA

Поскольку математические выкладки, лежащие в основе LDA, весьма сложны и требуют знания байесовского вывода, мы подойдем к этой теме с практической точки зрения и постараемся рассказать про этот подход простым языком, понятным непрофессионалу⁴.

LDA — это генеративная вероятностная модель, которая пытается найти группы слов, часто встречающихся вместе в разных документах. Эти часто встречающиеся слова отражают темы документов, если предположить, что каждый документ состоит из смеси разных слов. Входные данные для LDA предоставляет модель мешка слов, которую мы обсуждали ранее в этой главе.

Получив в качестве входных данных матрицу мешка слов, LDA разбивает ее на две новые матрицы:

- ◆ матрица документ-тема;
- ◆ матрица слово-тема.

LDA разлагает матрицу мешка слов таким образом, чтобы перемножением двух матриц мы могли воспроизвести исходную матрицу мешка слов с наименьшей возможной ошибкой. На практике нас интересуют темы, которые LDA нашел в матрице мешка слов. Единственным недостатком LDA можно считать необходимость заранее определить количество тем — это гиперпараметр, который нужно указывать вручную.

8.5.2. Реализация LDA в библиотеке scikit-learn

Здесь для декомпозиции набора обзора фильмов и его классификации по различным темам мы применим реализованный в scikit-learn класс LatentDirichletAllocation. В следующем примере мы ограничиваем анализ 10 различными темами и предлагаем читателям самостоятельно поэкспериментировать с гиперпараметрами алгоритма для дальнейшего изучения тем, которые можно найти в этом наборе данных.

Начнем с загрузки набора обзоров фильмов в DataFrame pandas из локального файла movie_data.csv, который мы создали в начале этой главы:

```
>>> import pandas as pd  
>>> df = pd.read_csv('movie_data.csv', encoding='utf-8')  
>>> # следующая строка нужна на для некоторых компьютеров:  
>>> df = df.rename(columns={"0": "review", "1": "sentiment"})
```

Затем применим уже знакомый вам CountVectorizer для создания матрицы набора слов, которая поступит на вход LDA.

Для удобства мы воспользуемся встроенной библиотекой стоп-слов английского языка scikit-learn в соответствии с параметром stop_words='english':

⁴ Однако заинтересованный читатель может прочитать больше о LDA в следующей статье: «Latent Dirichlet Allocation» by David M. Blei, Andrew Y. Ng, and Michael I. Jordan, Journal of Machine Learning Research 3, p. 993–1022, Jan 2003, <https://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>.

```
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count = CountVectorizer(stop_words='english',
...                         max_df=.1,
...                         max_features=5000)
>>> X = count.fit_transform(df['review'].values)
```

Обратите внимание, что мы установили максимальную частоту встречаемости слов в документе на уровне 10% (`max_df=.1`) — чтобы исключить слова, которые слишком часто встречаются в документах. Причина удаления часто встречающихся слов проста — это могут быть общие слова, содержащиеся во всех документах, которые, следовательно, с меньшей вероятностью связаны с определенной категорией темы того или иного документа. Кроме того, мы уменьшили количество рассматриваемых наиболее часто встречающихся слов до 5000 (`max_features=5000`) — чтобы ограничить размерность этого набора данных и улучшить вывод, выполняемый LDA. Поскольку значения гиперпараметров `max_df=.1` и `max_features=5000` выбраны произвольно, читателям рекомендуется настраивать их при сравнении результатов.

В следующем примере кода показано, как обучить оценщик `LatentDirichletAllocation` на матрице набора слов и вывести 10 различных тем из документов (обучение модели может занять до 5 минут или более на ноутбуке или стандартном настольном компьютере):

```
>>> from sklearn.decomposition import LatentDirichletAllocation
>>> lda = LatentDirichletAllocation(n_components=10,
...                                   random_state=123,
...                                   learning_method='batch')
>>> X_topics = lda.fit_transform(X)
```

Установив параметр `learning_method='batch'`, мы позволяем оценщику `lda` делать оценку на основе всех доступных обучающих данных (матрица мешка слов) за одну итерацию, что медленнее, чем альтернативный метод обучения `'online'`, но может привести к более точным результатам (значение параметра `learning_method='online'` соответствует онлайн-обучению или мини-пакетному обучению, которое мы обсуждали в главе 2 и ранее в этой главе).



Алгоритм ожидания-максимизации

Реализация LDA в библиотеке scikit-learn использует *алгоритм ожидания-максимизации* (Expectation-Maximization, EM) для итеративного обновления оценок параметров. Обсуждение EM-алгоритма выходит за рамки этой книги, но если вам интересно узнать о нем больше, обратитесь к статье в Википедии (<https://ru.wikipedia.org/wiki/EM-алгоритм>) и к подробному руководству по его использованию в LDA в учебнике Колорадо Рида «Latent Dirichlet Allocation: Towards a Deeper Understanding», который свободно доступен по адресу: http://obphio.us/pdfs/lda_tutorial.pdf.

После обучения LDA у нас теперь есть доступ к атрибуту `components_` экземпляра `lda`, в котором хранится матрица, содержащая важность слова (здесь 5000) для каждой из 10 тем в порядке возрастания:

```
>>> lda.components_.shape
(10, 5000)
```

Чтобы проанализировать результаты, давайте выведем пять самых важных слов для каждой из 10 тем. Значения важности слов ранжируются в порядке возрастания. Следовательно, чтобы вывести первые пять слов, нам нужно отсортировать массив тем в обратном порядке:

```
>>> n_top_words = 5
>>> feature_names = count.get_feature_names_out()
>>> for topic_idx, topic in enumerate(lda.components_):
...     print(f'Тема {topic_idx + 1}:')
...     print(' '.join([feature_names[i]
...                     for i in topic.argsort()\
...                     [-n_top_words - 1:-1]]))
Тема 1:
worst minutes awful script stupid
Тема 2:
family mother father children girl
Тема 3:
american war dvd music tv
Тема 4:
human audience cinema art sense
Тема 5:
police guy car dead murder
Тема 6:
horror house sex girl woman
Тема 7:
role performance comedy actor performances
Тема 8:
series episode war episodes tv
Тема 9:
book version original read novel
Тема 10:
action fight guy guys cool
```

Основываясь на пяти самых важных словах для каждой темы, можно догадаться, что LDA определила следующие темы:

1. Просто плохие фильмы (не совсем тематическая категория).
2. Фильмы о семье.
3. Военные фильмы.
4. Художественные фильмы.
5. Детективные фильмы.
6. Фильмы ужасов.
7. Комедии.
8. Фильмы, как-то связанные с сериалами.
9. Фильмы по книгам.
10. Боевики.

Чтобы убедиться, что категории осмысленно связаны с реальными обзорами, давайте выведем текст отзывов на три фильма из категории «фильмы ужасов» (это категория 6 с индексом позиции 5):

```
>>> horror = X_topics[:, 5].argsort() [::-1]
>>> for iter_idx, movie_idx in enumerate(horror[:3]):
...     print(f'\nФильм ужасов #{(iter_idx + 1)}:')
...     print(df['review'][movie_idx][:300], '...')

Фильм ужасов #1:
```

House of Dracula works from the same basic premise as House of Frankenstein from the year before; namely that Universal's three most famous monsters; Dracula, Frankenstein's Monster and The Wolf Man are appearing in the movie together. Naturally, the film is rather messy therefore, but the fact that ...

(Дом Дракулы основан на той же основной идее, что и Дом Франкенштейна годом ранее; а именно, на трех самых известных монстрах Universal; Дракула, Монстр Франкенштейна и Человек-волк появляются в фильме вместе. Естественно, поэтому фильм довольно сумбурный, но то, что...)

Фильм ужасов #2:

Okay, what the hell kind of TRASH have I been watching now? "The Witches' Mountain" has got to be one of the most incoherent and insane Spanish exploitation flicks ever and yet, at the same time, it's also strangely compelling. There's absolutely nothing that makes sense here and I even doubt there ...

(Да ладно, что за чудовищную дрянь я сейчас смотрел? Гора ведьм, должно быть, один из самых бессвязных и безумных испанских фильмов о сверхъестественном, и в то же время он странно убедителен. В нем нет абсолютно никакого смысла, и я даже сомневаюсь ...)

Фильм ужасов #3:

Horror movie time, Japanese style. Uzumaki/Spiral was a total freakfest from start to finish. A fun freakfest at that, but at times it was a tad too reliant on kitsch rather than the horror. The story is difficult to summarize succinctly: a carefree, normal teenage girl starts coming fac ...

(

Типичный фильмов ужасов в японском стиле. Uzumaki/Спираль наполнен чудачеством от начала до конца. Такое чудачество веселит, но временами фильм слишком похож на китч, а не на ужасы. Сюжет сложно пересказать кратко: беззаботная нормальная девочка-подросток начинает встречаться с ...)

Используя приведенный блок кода, мы вывели на печать первые 300 символов для трех наиболее популярных фильмов ужасов. Рецензии — хотя мы не знаем, к какому именно фильму они относятся, — действительно звучат как отзывы на фильмы ужасов (однако можно разразить, что фильм ужасов #2, судя по отзыву, также хорошо подходит для тематической категории 1 «Просто плохие фильмы»).

8.6. Заключение

В этой главе вы научились использовать алгоритмы машинного обучения для смыслового анализа текста и классификации текстовых документов на основе их эмоциональной окраски, что является основной задачей анализа настроений в области NLP. Вы не только узнали, как кодировать документ в виде вектора признаков, используя модель мешка слов, но и научились взвешивать частоту термина по релевантности, применяя показатель TF-IDF.

Работа с текстовыми данными бывает весьма затратной в вычислительном отношении из-за больших векторов признаков, которые создаются во время этого процесса, и в последнем разделе вы узнали, как использовать внешнее или постепенное обучение, чтобы избежать загрузки всего набора данных в память компьютера.

Наконец, вы познакомились с методом моделирования тем на основе скрытого распределения Дирихле и применили его для распределения обзоров фильмов по различным категориям с помощью обучения без учителя.

К этому моменту мы рассмотрели множество концепций машинного обучения, лучших методов и классифицирующих моделей, обучаемых с учителем. В следующей главе мы обратимся к другой подкатегории обучения с учителем — регрессионному анализу, который позволит нам прогнозировать переменные результата на непрерывной шкале, — в отличие от категориальных меток классов в классифицирующих моделях, с которыми мы работали до сих пор.

9

Прогнозирование непрерывных целевых переменных с помощью регрессионного анализа

В предыдущих главах вы усвоили основные принципы обучения с учителем и обучили несколько различных классифицирующих моделей прогнозировать принадлежность к группе, или категориальные переменные. В этой главе мы изучим другую разновидность обучения с учителем: *регрессионный анализ*.

Регрессионные модели применяются для прогнозирования непрерывных целевых переменных, что делает их привлекательными для решения многих научных задач. Они также широко применяются в промышленности — например, для изучения взаимосвязей между переменными, оценки тенденций или составления прогнозов. Одним из примеров является прогнозирование продаж компаний на ближайшие месяцы.

В этой главе мы обсудим основные концепции регрессионных моделей и затронем следующие темы:

- ◆ изучение и визуализация наборов данных;
- ◆ различные подходы к реализации линейных регрессионных моделей;
- ◆ обучение регрессионных моделей, устойчивых к выбросам;
- ◆ оценка регрессионных моделей и диагностика распространенных проблем;
- ◆ подгонка регрессионных моделей к нелинейным данным.

9.1. Знакомство с линейной регрессией

Назначение линейной регрессии — смоделировать взаимосвязь между одним или несколькими признаками и непрерывной целевой переменной. В отличие от классификации (другой разновидности обучения с учителем), регрессионный анализ направлен на прогнозирование результатов на непрерывной шкале, а не на категориальных метках классов.

В ближайших разделах этой главы вы познакомитесь с основным типом линейной регрессии — *простой линейной регрессией* и поймете, как связать ее с более общим многомерным случаем (линейная регрессия с несколькими признаками).

9.1.1. Простая линейная регрессия

Простая (одномерная) линейная регрессия моделирует взаимосвязь между единственным признаком (*независимая переменная*, или *предиктор x*) и целью с непрерывным значением (*переменная отклика y*). Уравнение линейной модели с одним предиктором выглядит очень просто:

$$y = w_1 x + b.$$

Здесь параметр b (смещение, *bias*) представляет точку пересечения оси y , а w_1 — весовой коэффициент предиктора. Наша цель — узнать вес и смещение линейного уравнения для описания взаимосвязи между предиктором и целевой переменной, которые затем можно использовать для прогнозирования откликов на новые значения предиктора, которые не были частью обучающего набора данных.

Основываясь на линейном уравнении, которое мы определили ранее, линейную регрессию можно рассматривать как поиск наиболее подходящей прямой с помощью обучающих примеров (рис. 9.1).

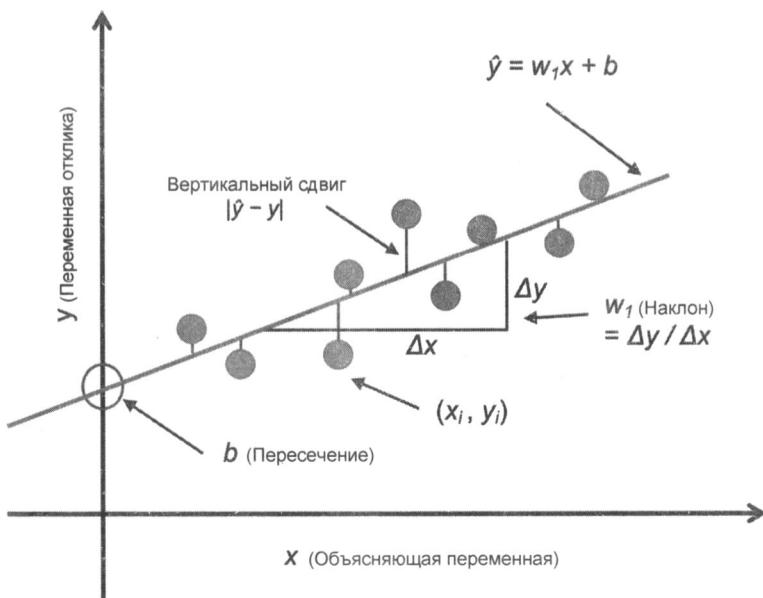


Рис. 9.1. Простой пример линейной регрессии с одним признаком

Эта наиболее подходящая к данным линия также называется *линией регрессии* (*regression line*), а вертикальные отрезки от линии регрессии до обучающих примеров — это так называемые *смещения* (*offset*), или *остатки* (*residual*), — ошибки нашего прогноза.

9.1.2. Множественная линейная регрессия

В предыдущем разделе была представлена простая линейная регрессия — частный случай линейной регрессии с одним предиктором. Разумеется, мы также можем обобщить

модель линейной регрессии на несколько независимых предикторов — этот процесс называется *многофакторной линейной регрессией* (multiple linear regression):

$$y = w_1 x_1 + \cdots + w_m x_m + b = \sum_{i=1}^m w_i x_i + b = \mathbf{w}^T \mathbf{x} + b.$$

На рис. 9.2 показано, как может выглядеть двумерная гиперплоскость модели многофакторной линейной регрессии с двумя признаками.

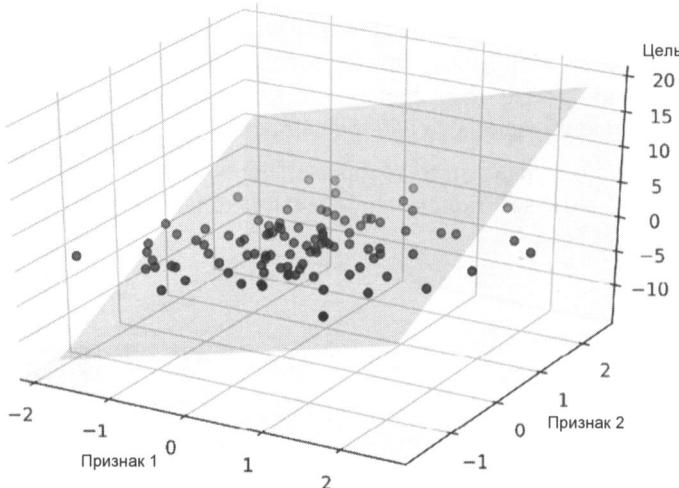


Рис. 9.2. Модель линейной регрессии с двумя признаками

Как видите, визуализацию нескольких гиперплоскостей линейной регрессии на трехмерной диаграмме рассеяния уже сложно интерпретировать при взгляде на статическое изображение. Поскольку у нас нет хороших средств визуализации гиперплоскостей с двумя измерениями на диаграмме рассеяния (многофакторные модели линейной регрессии обучаются на наборах данных с тремя или более признаками), примеры и визуальные представления в этой главе будут в основном сосредоточены на одномерном случае с использованием простой линейной регрессии. Однако простая и многофакторная линейные регрессии основаны на одинаковых принципах и одинаковых методах оценки, поэтому реализации кода, которые мы обсудим в этой главе, также совместимы с обоими типами регрессионной модели.

9.2. Изучение набора данных Ames Housing

Прежде чем приступить к реализации первой модели линейной регрессии, мы исследуем новый набор данных под названием Ames Housing, который содержит информацию о частной жилой недвижимости в городе Эймс, штат Айова, с 2006 по 2010 год. Набор данных был собран Дином Де Коком в 2011 году, дополнительная информация о нем доступна по следующим ссылкам:

- ◆ отчет с описанием набора данных: <http://jse.amstat.org/v19n3/decock.pdf>;
- ◆ подробная документация по признакам набора данных:
<http://jse.amstat.org/v19n3/decock/DataDocumentation.txt>;

- ◆ набор данных в формате, разделенном табуляцией:
<http://jse.amstat.org/v19n3/decock/AmesHousing.txt>.

Любой новый набор данных всегда полезно исследовать с помощью простых средств визуализации, чтобы лучше понять, с чем мы будем работать. Мы займемся этим в следующих разделах.

9.2.1. Загрузка набора данных Ames Housing в DataFrame

Сначала мы загрузим набор данных Ames Housing с помощью функции pandas `read_csv`, которая является быстрой и универсальной и рекомендуется для работы с табличными данными, хранящимися в текстовом формате.

Набор данных Ames Housing состоит из 2930 примеров и 80 признаков. Для простоты мы будем работать только с подмножеством признаков, показанных в следующем списке. Однако, если вам интересно, перейдите по приведенной ранее ссылке на полное описание набора данных и изучите другие переменные в этом наборе данных после прочтения этой главы.

Признаки, с которыми мы будем работать, включая целевую переменную, следующие:

- ◆ Overall Qual — общая оценка материала и отделки дома по шкале от 1 (очень плохо) до 10 (отлично);
- ◆ Overall Cond — общая оценка состояния дома по шкале от 1 (очень плохо) до 10 (отлично);
- ◆ Gr Liv Area — жилая площадь над землей в квадратных футах;
- ◆ Central Air — наличие центрального кондиционера (N=нет, Y=да);
- ◆ Total Bsmt SF — общая площадь подвала в квадратных футах;
- ◆ SalePrice — цена продажи в долларах США (\$).

В оставшейся части этой главы мы будем рассматривать цену продажи (`SalePrice`) как нашу целевую переменную — т. е. мы хотим ее предсказать, используя один или несколько из пяти независимых предикторов. Прежде чем приступить к изучению этого набора данных, загрузим его в DataFrame pandas:

```
import pandas as pd

columns = ['Overall Qual', 'Overall Cond', 'Gr Liv Area',
           'Central Air', 'Total Bsmt SF', 'SalePrice']
df = pd.read_csv('http://jse.amstat.org/v19n3/decock/AmesHousing.txt',
                 sep='\t',
                 usecols=columns)
df.head()
```

Чтобы убедиться, что набор данных был успешно загружен, отобразим первые пять строк набора, как показано на рис. 9.3.

Кроме того, после загрузки набора данных мы также проверим размеры DataFrame, чтобы убедиться, что он содержит ожидаемое количество строк:

	Overall Qual	Overall Cond	Total Bsmt SF	Central Air	Gr Liv Area	SalePrice
0	6	5	1080.0	Y	1656	215000
1	5	6	882.0	Y	896	105000
2	6	6	1329.0	Y	1329	172000
3	7	5	2110.0	Y	2110	244000
4	5	5	928.0	Y	1629	189900

Рис. 9.3. Первые пять строк набора данных о жилье

```
>>> df.shape
(2930, 6)
```

Здесь все, как и ожидалось, — DataFrame содержит 2930 строк.

Еще один аспект, о котором мы должны позаботиться, — это переменная 'Central Air', которая закодирована как тип string (строка), — что можно видеть на рис. 9.3. Как мы узнали из главы 4, для преобразования столбцов DataFrame мы можем использовать метод .map. Следующий код преобразует строку 'Y' в целое число 1, а строку 'N' — в целое число 0:

```
>>> df['Central Air'] = df['Central Air'].map({'N': 0, 'Y': 1})
```

Наконец, проверим, не содержит ли какой-либо из столбцов фрейма данных пропущенные значения:

```
>>> df.isnull().sum()
Overall Qual      0
Overall Cond      0
Total Bsmt SF     1
Central Air        0
Gr Liv Area        0
SalePrice          0
dtype: int64
```

Как можно здесь видеть, переменная признака Total Bsmt SF содержит одно пропущенное значение. Поскольку мы располагаем относительно большим набором данных, самый простой способ справиться с отсутствующим значением признака — удалить соответствующую запись из набора данных (альтернативные методы рассматривались в главе 4):

```
>>> df = df.dropna(axis=0)
>>> df.isnull().sum()
```

```
Overall Qual      0
Overall Cond      0
Total Bsmt SF     0
Central Air        0
Gr Liv Area        0
SalePrice          0
dtype: int64
```

9.2.2. Визуализация важных характеристик набора данных

Исследовательский анализ данных (Exploratory Data Analysis, EDA) — важный и рекомендуемый первый шаг перед обучением модели. Мы воспользуемся некоторыми простыми, но полезными методами из набора графических инструментов EDA, которые помогут нам визуально обнаружить наличие выбросов, а также исследовать распределение данных и отношения между признаками.

Для начала создадим *матрицу диаграмм рассеяния* (scatterplot matrix), которая позволит нам визуально отобразить попарные корреляции между различными признаками в этом наборе данных в одном месте. Для построения диаграммы рассеяния мы воспользуемся функцией `scatterplotmatrix` из библиотеки `mlxtend` (<http://rasbt.github.io/mlxtend/>), содержащей различные удобные функции для машинного обучения и приложений обработки данных в Python.

Вы можете установить пакет `mlxtend` при помощи команды `conda install mlxtend` или `pip install mlxtend`. В этой главе мы использовали `mlxtend` версии 0.19.0.

Завершив установку пакета, вы можете импортировать его и создать матрицу диаграммы рассеяния следующим образом:

```
>>> import matplotlib.pyplot as plt
>>> from mlxtend.plotting import scatterplotmatrix
>>> scatterplotmatrix(df.values, figsize=(12, 10),
...                     names=df.columns, alpha=0.5)
>>> plt.tight_layout()
plt.show()
```

На рис. 9.4 показано, что матрица диаграмм рассеяния предоставляет нам полезную графическую сводку взаимосвязей в наборе данных.

Исследуя эту матрицу, мы теперь можем быстро понять, как распределяются данные, и содержат ли они выбросы. Например, ясно видно (пятый столбец слева в нижней строке), что существует определенная линейная зависимость между размером жилой площади над землей (Gr Liv Area) и ценой продажи (SalePrice).

Кроме того, на нижнем правом графике матрицы мы видим, что переменная SalePrice, кажется, искажена несколькими выбросами.



Предположение о нормальности линейной регрессии

Заметим, что, вопреки распространенному мнению, для обучения модели линейной регрессии не требуется, чтобы предикторы или целевые переменные были нормально распределены. Предположение о нормальности является требованием только для определенных статистических данных и проверки гипотез, которые выходят за рамки этой книги (дополнительную информацию по этой теме см. в книге «Introduction to Linear Regression Analysis» by Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining, Wiley, p. 318–319, 2012).

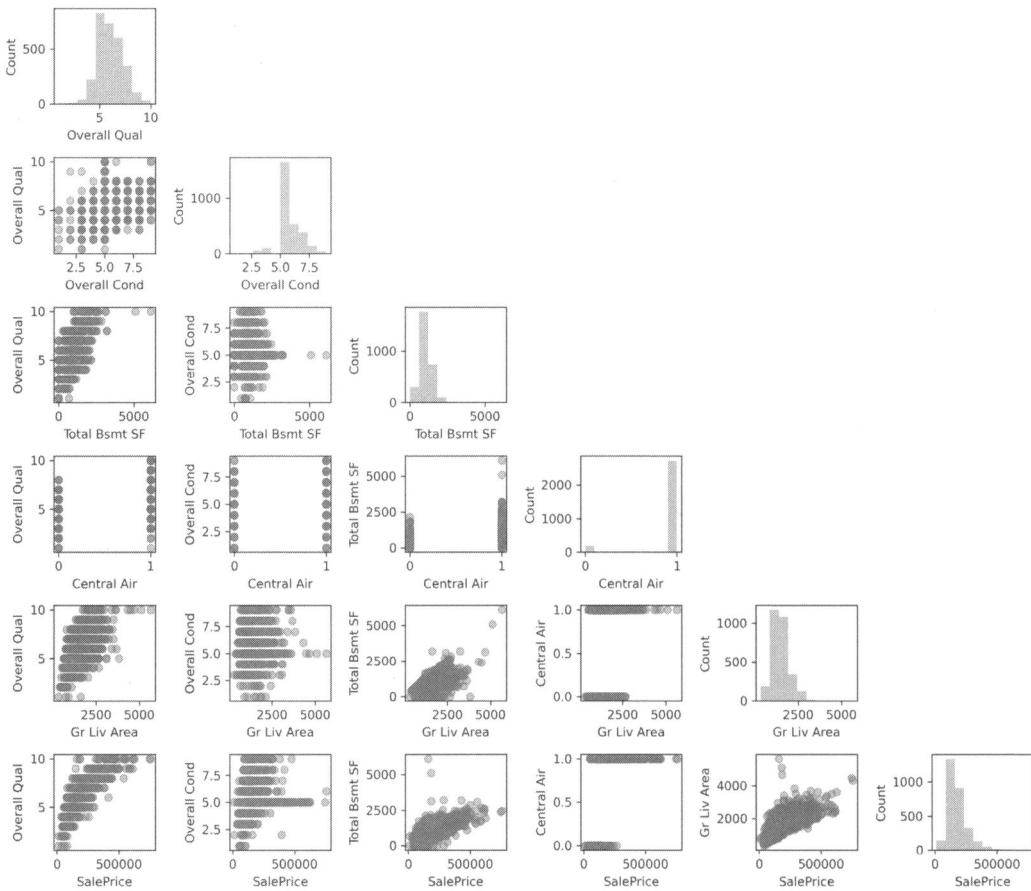


Рис. 9.4. Матрица диаграмм рассеяния наших данных

9.2.3. Просмотр отношений с помощью матрицы корреляции

В предыдущем разделе мы визуализировали распределение переменных из набора данных Ames Housing в виде гистограмм и диаграмм рассеяния. А сейчас мы создадим матрицу корреляции для количественной оценки и суммирования линейных взаимосвязей между переменными. Матрица корреляции тесно связана с матрицей ковариации, которую мы рассмотрели в разд. 5.1. Мы можем интерпретировать матрицу корреляции как масштабированную версию матрицы ковариации. Фактически матрица корреляции идентична матрице ковариации, вычисленной на основе стандартизованных признаков.

Матрица корреляции представляет собой квадратную матрицу, содержащую значения коэффициента корреляции Пирсона (часто обозначаемого символом r), который измеряет линейную зависимость между парами признаков. Значения коэффициента корреляции находятся в диапазоне от -1 до 1 . Два признака имеют полную положительную корреляцию, если $r = 1$, отсутствие корреляции, если $r = 0$, и полную отрицательную

корреляцию, если $r = -1$. Как упоминалось ранее, коэффициент корреляции Пирсона можно просто вычислить как ковариацию между двумя признаками x и y (числитель), деленную на произведение их стандартных отклонений (знаменатель):

$$r = \frac{\sum_{i=1}^n [(x^{(i)} - \mu_x)(y^{(i)} - \mu_y)]}{\sqrt{\sum_{i=1}^n (x^{(i)} - \mu_x)^2} \sqrt{\sum_{i=1}^n (y^{(i)} - \mu_y)^2}} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

Здесь μ обозначает среднее значение соответствующего признака, σ_{xy} — ковариация между признаками x и y , а σ_x и σ_y — стандартные отклонения признаков.



Сравнение ковариации и корреляции для стандартизованных признаков

Можно показать, что ковариация между парой стандартизованных признаков фактически равна их коэффициенту линейной корреляции. Чтобы доказать это, сначала стандартизуем признаки x и y , чтобы получить их z-показатели, которые мы будем обозначать как x' и y' соответственно:

$$x' = \frac{x - \mu_x}{\sigma_x}, \quad y' = \frac{y - \mu_y}{\sigma_y}.$$

Напомним, что ковариация между двумя признаками вычисляется следующим образом:

$$\sigma_{xy} = \frac{1}{n} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y).$$

Поскольку стандартизация центрирует переменную признака с серединой на нуле, теперь мы можем рассчитать ковариацию между масштабированными признаками:

$$\sigma'_{xy} = \frac{1}{n} \sum_i^n (x'^{(i)} - 0)(y'^{(i)} - 0).$$

В результате повторной подстановки мы получаем следующий результат:

$$\begin{aligned}\sigma'_{xy} &= \frac{1}{n} \sum_i^n \left(\frac{x - \mu_x}{\sigma_x} \right) \left(\frac{y - \mu_y}{\sigma_y} \right) \\ \sigma'_{xy} &= \frac{1}{n \cdot \sigma_x \sigma_y} \sum_i^n (x^{(i)} - \mu_x)(y^{(i)} - \mu_y).\end{aligned}$$

Наконец, мы можем упростить это уравнение:

$$\sigma'_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

В следующем примере кода мы задействуем функцию `corrcoef` библиотеки NumPy для пяти столбцов признаков, которые ранее визуализировали в матрице диаграмм рассеяния, а также применим функцию `heatmap` библиотеки `mlxtend` для построения массива матрицы корреляции в виде тепловой карты:

```
>>> import numpy as np
>>> from mlxtend.plotting import heatmap

>>> cm = np.corrcoef(df.values.T)
>>> hm = heatmap(cm, row_names=df.columns, column_names=df.columns)
```

```
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 9.5, матрица корреляции дает еще одно полезное сводное представление, которое поможет нам выбрать признаки на основе их соответствующих линейных корреляций.

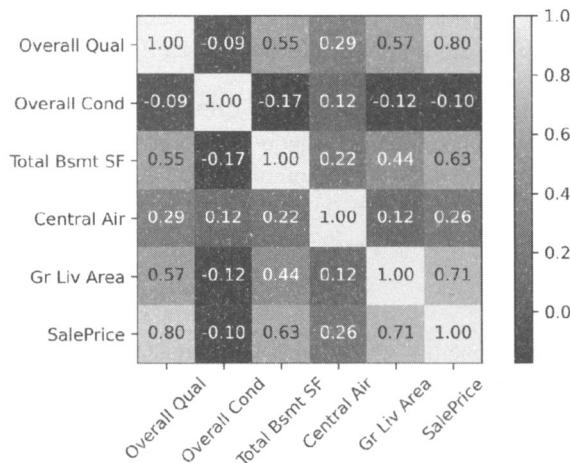


Рис. 9.5. Матрица корреляции выбранных переменных

Чтобы обучить модель линейной регрессии, нам нужны те функции, которые имеют высокую корреляцию с нашей целевой переменной SalePrice. Глядя на матрицу корреляции, мы видим, что SalePrice имеет наибольшую корреляцию с переменной Gr Liv Area (0.71), что можно считать хорошим вариантом исследовательской переменной для знакомства с концепцией модели простой линейной регрессии, представленной в следующем разделе.

9.3. Реализация обычной модели линейной регрессии методом наименьших квадратов

В начале этой главы мы упомянули, что обучение линейной регрессии можно рассматривать как построение прямой линии, наиболее точно соответствующей примерам обучающих данных. Однако мы не давали определения термину «наиболее точное соответствие» и не обсуждали различные методы обучения такой модели. Сейчас мы восполним недостающие части этой головоломки, используя *обычный метод наименьших квадратов* (Ordinary Least Squares, OLS) (иногда также называемый *линейным методом наименьших квадратов*), чтобы оценить параметры линии линейной регрессии, которая минимизирует сумму квадратов вертикальных расстояний (остатки или ошибки) относительно обучающих примеров.

9.3.1. Нахождение регрессии для параметров регрессии с градиентным спуском

Мысленно вернемся к нашей реализации адаптивного линейного нейрона (Adaline) из главы 2. Вы помните, что искусственный нейрон использует линейную функцию активации. Кроме того, мы определили функцию потерь $L(w)$, которую минимизировали, чтобы найти веса с помощью алгоритмов оптимизации, таких как градиентный спуск (GD) и стохастический градиентный спуск (SGD).

Эта функция потерь в Adaline представляет собой *среднеквадратичную ошибку* (Mean Squared Error, MSE), идентичную функции потерь, применяемой для OLS:

$$L(w, b) = \frac{1}{2n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2.$$

Здесь \hat{y} — прогнозируемое значение $\hat{y} = w^T x + b$ (множитель $1/2$ используется только для удобства получения правила обновления GD). По сути, регрессию по методу OLS можно рассматривать как Adaline без пороговой функции, когда мы получаем непрерывные целевые значения вместо меток класса 0 и 1. Чтобы продемонстрировать это, давайте возьмем реализацию Adaline GD из главы 2 и удалим пороговую функцию, таким образом получив нашу первую модель линейной регрессии:

```
class LinearRegressionGD:
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.array([0.])
        self.losses_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)

        return self

    def net_input(self, X):
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        return self.net_input(X)
```



Обновления веса с градиентным спуском

Если вы хотите освежить в памяти, как обновляются веса — путем шага в направлении, противоположном градиенту, — пожалуйста, вернитесь к разд. 2.3.

Чтобы увидеть наш регрессор LinearRegressionGD в действии, воспользуемся признаком Gr Living Area (размер жилой площади над землей в квадратных футах) из набора данных Ames Housing в качестве независимой переменной и обучим модель, которая может предсказывать стоимость продажи жилья. Кроме того, мы стандартизируем переменные для лучшей сходимости алгоритма GD:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> from sklearn.preprocessing import StandardScaler
>>> sc_x = StandardScaler()
>>> sc_y = StandardScaler()
>>> X_std = sc_x.fit_transform(X)
>>> y_std = sc_y.fit_transform(y[:, np.newaxis]).flatten()
>>> lr = LinearRegressionGD(eta=0.1)
>>> lr.fit(X_std, y_std)
```

Обратите внимание на небольшую хитрость при нахождении значения `y_std`, связанную с использованием `np.newaxis` и `flatten`. Большинство классов предварительной обработки данных в scikit-learn предполагают, что данные будут храниться в двумерных массивах. В приведенном примере кода использование метода `np.newaxis` в `y[:, np.newaxis]` добавило в массив новое измерение. Затем, после того как `StandardScaler` вернул масштабированную переменную, мы преобразовали ее для нашего удобства обратно в исходное представление одномерного массива, прибегнув к методу `flatten()`.

В главе 2 мы отметили, что при использовании алгоритмов оптимизации, таких как GD, всегда полезно отображать потери как функцию количества эпох (полных итераций) по набору обучающих данных, чтобы проверить, что алгоритм сошелся к минимуму потерь (здесь *глобальный минимум потерь*):

```
>>> plt.plot(range(1, lr.n_iter+1), lr.losses_)
>>> plt.ylabel('MSE')
>>> plt.xlabel('Epoch')
>>> plt.show()
```

Как вы можете видеть на рис. 9.6, алгоритм GD сошелся примерно после десятой эпохи.

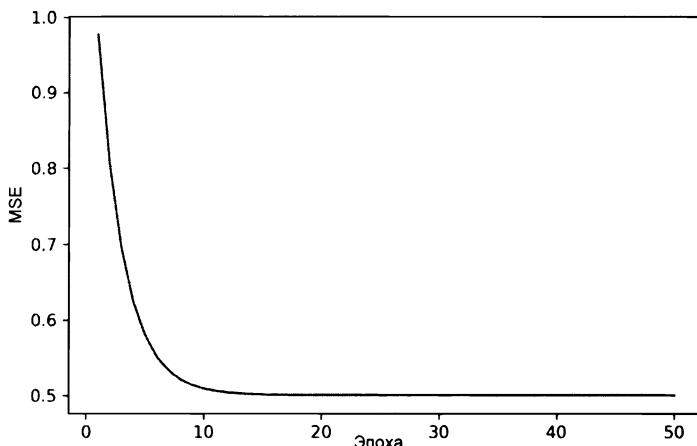


Рис. 9.6. Функция потерь в зависимости от количества эпох

Далее мы выполним визуальную проверку того, насколько хорошо линия линейной регрессии подогнана к обучающим данным. Для этого мы определим простую вспомогательную функцию, которая построит диаграмму рассеивания обучающих примеров и добавит линию регрессии:

```
>>> def lin_regplot(X, y, model):
...     plt.scatter(X, y, c='steelblue', edgecolor='white', s=70)
...     plt.plot(X, model.predict(X), color='black', lw=2)
```

Теперь воспользуемся нашей функцией `lin_regplot` для построения графика жилой площади в зависимости от цены продажи:

```
>>> lin_regplot(X_std, y_std, lr)
>>> plt.xlabel('Жилая площадь над землей (стандартизирована)')
>>> plt.ylabel('Цена продажи (стандартизирована)')
>>> plt.show()
```

Как показано на рис. 9.7, линия линейной регрессии в целом отражает общую тенденцию роста цен на жилье с размером жилой площади.

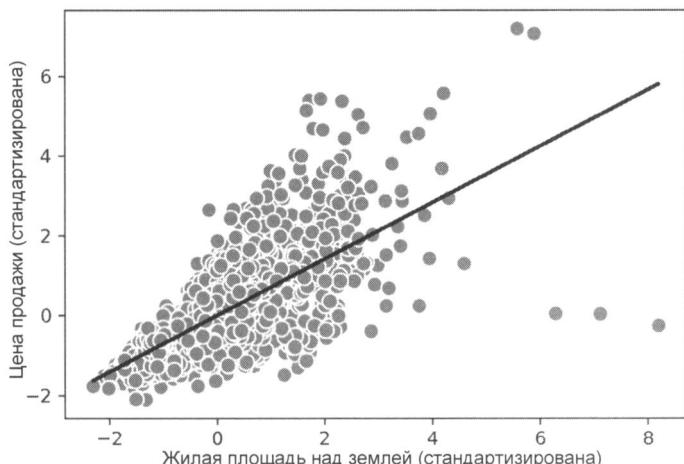


Рис. 9.7. График линейной регрессии цены продажи недвижимости в зависимости от размера жилой площади

Хотя этот вывод имеет смысл, данные также говорят нам о том, что размер жилой площади во многих случаях не очень хорошо объясняет цены на жилье. Позже в этой главе мы обсудим, как количественно оценить эффективность регрессионной модели. Интересно, что мы также можем наблюдать несколько выбросов — например, три точки данных, соответствующие случаям, когда стандартизированная жилая площадь превышает 6. Позже в этой главе мы также обсудим, как справиться с выбросами.

В некоторых случаях также может быть важно сообщать прогнозируемые переменные результата в их исходной шкале. Чтобы масштабировать прогнозируемую цену обратно к исходной цене в долларах США, достаточно применить метод `inverse_transform` класса `StandardScaler`:

```
>>> feature_std = sc_x.transform(np.array([[2500]]))
>>> target_std = lr.predict(feature_std)
```

```
>>> target_reverted = sc_y.inverse_transform(target_std.reshape(-1, 1))
>>> print(f'Sales price: ${target_reverted.flatten()[0]:.2f}')
Sales price: $292507.07
```

В этом примере кода мы использовали ранее обученную модель линейной регрессии для прогнозирования цены дома с надземной жилой площадью 2500 кв. футов. Согласно прогнозу нашей модели такой дом будет стоить 292507.07 доллара.

В качестве примечания также стоит упомянуть, что технически нам не нужно обновлять параметр точки пересечения (например, смещение b), если мы работаем со стандартизованными переменными, поскольку точка пересечения по оси Y всегда равна 0 в этих случаях. Мы можем быстро убедиться в этом, выведя параметры модели:

```
>>> print(f'Slope: {lr.w_[0]:.3f}')
Slope: 0.707
>>> print(f'Intercept: {lr.b_[0]:.3f}')
Intercept: -0.000
```

9.3.2. Оценка коэффициента регрессионной модели с помощью scikit-learn

В предыдущем разделе мы построили работающую модель для регрессионного анализа; однако в реальном приложении нас могут заинтересовать более эффективные реализации. Например, многие оценщики scikit-learn для регрессии используют реализацию метода наименьших квадратов в SciPy (`scipy.linalg.lstsq`), которая, в свою очередь, основана на высокооптимизированном коде из пакета линейной алгебры (Linear Algebra Package, LAPACK). Реализация линейной регрессии в scikit-learn также работает (и даже лучше) с нестандартизованными переменными, поскольку не использует оптимизацию на основе (S)GD, поэтому мы можем пропустить шаг стандартизации:

```
>>> from sklearn.linear_model import LinearRegression
>>> slr = LinearRegression()
>>> slr.fit(X, y)
>>> y_pred = slr.predict(X)
>>> print(f'Наклон: {slr.coef_[0]:.3f}')
Наклон: 111.666
>>> print(f'Пересечение: {slr.intercept_:.3f}')
Пересечение: 13342.979
```

Как видно из результатов выполнения этого кода, модель `LinearRegression` scikit-learn, обученная с нестандартизованными переменными `Gr Liv Area` и `SalePrice`, получила другие коэффициенты, поскольку признаки не были стандартизированы. Однако, если мы сравним эту модель с предыдущей реализацией GD, построив график `SalePrice` в зависимости от `Gr Liv Area`, то увидим, что она одинаково хорошо соответствует данным:

```
>>> lin_regplot(X, y, slr)
>>> plt.xlabel('Жилая площадь над землей, кв. футы')
>>> plt.ylabel('цена продажи, долл. США')
>>> plt.tight_layout()
>>> plt.show()
```

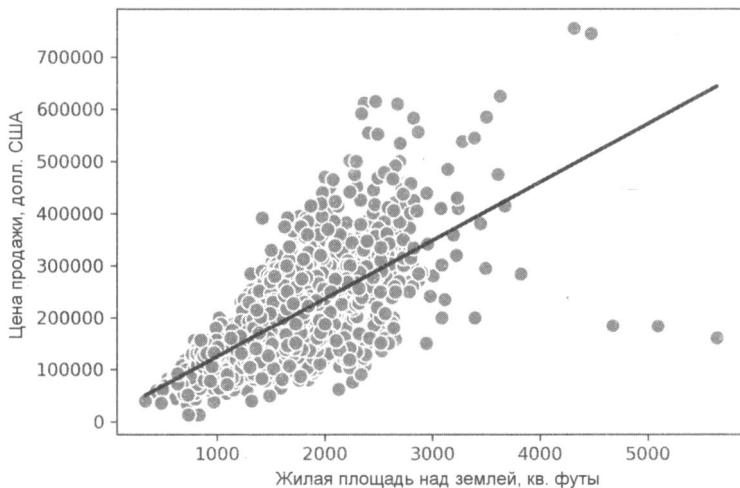


Рис. 9.8. График линейной регрессии с использованием scikit-learn

Например, на рис. 9.8 мы видим, что в целом результат выглядит идентично нашей реализации GD.



Аналитические решения линейной регрессии

Альтернативой использованию библиотек машинного обучения может служить аналитическое решение задачи OLS, включающее систему линейных уравнений, которую можно найти в большинстве вводных учебников по статистике:

$$w = (X^T X)^{-1} X^T y .$$

Реализация на Python выглядит следующим образом:

```
# добавляем вектор-столбец из "единиц"
>>> Xb = np.hstack((np.ones((X.shape[0], 1)), X))
>>> w = np.zeros(X.shape[1])
>>> z = np.linalg.inv(np.dot(Xb.T, Xb))
>>> w = np.dot(z, np.dot(Xb.T, y))
>>> print(f'Наклон: {w[1]:.3f}')
Наклон: 111.666
>>> print(f'Пересечение: {w[0]:.3f}')
Пересечение: 13342.979
```

Преимущество этого метода в том, что он гарантированно находит оптимальное решение аналитическим путем. Однако, если мы работаем с очень большими наборами данных, обращение матрицы в этой формуле (иногда также называемой *нормальным уравнением*) может быть слишком затратным с вычислительной точки зрения, или матрица, содержащая обучающие примеры, может быть сингулярной (необратимой). Вот почему в некоторых случаях предпочитают итерационные методы.

Если вас интересует дополнительная информация о том, как получить нормальные уравнения, прочтите главу из лекций доктора Стивена Поллока «Классическая модель линейной регрессии» в Университете Лестера, которая доступна бесплатно по адресу: <http://www.le.ac.uk/users/dsgp1/COURSES/MESOMET/ECMETXT/06mesmet.pdf>.

Кроме того, если вы хотите сравнить решения линейной регрессии, полученные с помощью GD, SGD, QR-факторизации, сингулярной векторной декомпозиции и аналитически, вам пригодится класс `LinearRegression`, реализованный в `mlxtend` (http://rasbt.github.io/mlxtend/user_guide/regressor/LinearRegression/), позволяющий пользователям переключаться между этими параметрами. Еще одна замечательная библиотека, которую можно порекомендовать для регрессионного моделирования в Python, — это `statsmodels`. Она реализует более продвинутые модели линейной регрессии (<https://www.statsmodels.org/stable/examples/index.html#regression>).

9.4. Обучение устойчивой регрессионной модели с использованием RANSAC

На модели линейной регрессии может сильно повлиять наличие выбросов. В определенных ситуациях очень ограниченное подмножество данных способно оказать большое влияние на коэффициенты модели. Для обнаружения выбросов можно использовать множество статистических тестов, но их описание выходит за рамки книги. Однако удаление выбросов всегда требует нашего собственного суждения как специалистов по данным, а также наших знаний в предметной области.

В качестве альтернативы удалению выбросов мы рассмотрим метод устойчивой регрессии с использованием алгоритма RANdom SAmple Consensus (RANSAC), который обучает модель регрессии на *инлаерах* (*inlier*)¹ — неискаженных выбросами «хороших» примерах.

Итеративный алгоритм RANSAC можно кратко описать следующим образом:

1. Выберите случайное количество примеров, которые будут считаться инлаерами, и обучите на них модель.
2. Протестируйте все остальные точки данных на обученной модели и добавьте к инлаерам те точки, которые попадают в заданный вами допуск.
3. Заново обучите модель, используя все инлаеры.
4. Оцените ошибку обученной модели по сравнению с инлаерами.
5. Завершите алгоритм, если производительность модели соответствует определенному порогу, заданному вами, или если достигнуто предельное количество итераций; в противном случае вернитесь к шагу 1.

В качестве примера воспользуемся линейной моделью в сочетании с алгоритмом RANSAC, реализованным в классе `RANSACRegressor` от `scikit-learn`:

```
>>> from sklearn.linear_model import RANSACRegressor
>>> ransac = RANSACRegressor(
...     LinearRegression(),
...     max_trials=100, # default value
...     min_samples=0.95,
...     residual_threshold=None, # default value
...     random_state=123)
>>> ransac.fit(X, y)
```

¹ В противоположность *аутлаерам* (*outlier*), т. е. выбросам. — Прим. пер.

Мы устанавливаем здесь максимальное количество итераций RANSACRegressor равным 100, а также при помощи параметра `min_samples=0.95` задаем минимальное количество случайно выбранных обучающих примеров в размере 95% от исходного набора данных.

По умолчанию (через параметр `residual_threshold=None`) scikit-learn использует оценку MAD для выбора порогового значения, где MAD означает *медианное абсолютное отклонение* (Median Absolute Deviation) целевых значений y . Однако выбор подходящего значения порога инлаера зависит от конкретной задачи, что является одним из недостатков RANSAC².

Теперь давайте получим инлаеры и аутлаеры с помощью только что обученной модели линейной регрессии RANSAC и построим их вместе с линейной регрессией:

```
>>> inlier_mask = ransac.inlier_mask_
>>> outlier_mask = np.logical_not(inlier_mask)
>>> line_X = np.arange(3, 10, 1)
>>> line_y_ransac = ransac.predict(line_X[:, np.newaxis])
>>> plt.scatter(X[inlier_mask], y[inlier_mask],
...                 c='steelblue', edgecolor='white',
...                 marker='o', label='Инлаеры')
>>> plt.scatter(X[outlier_mask], y[outlier_mask],
...                 c='limegreen', edgecolor='white',
...                 marker='s', label='Аутлаеры')
>>> plt.plot(line_X, line_y_ransac, color='black', lw=2)
>>> plt.xlabel('Жилая площадь над землей, кв. футы')
>>> plt.ylabel('Цена продажи, долл. США')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 9.9, модель линейной регрессии была обучена на подобранным подмножестве инлаеров (обозначены кружками).

Если мы выведем на экран наклон и точку пересечения линии регрессии при помощи следующего кода, эта линия будет немного отличаться от той, которую мы получили в предыдущем разделе без использования RANSAC:

```
>>> print(f'Наклон: {ransac.estimator_.coef_[0]:.3f}')
Наклон: 106.348
>>> print(f'Пересечение: {ransac.estimator_.intercept_:.3f}')
Пересечение: 20190.093
```

Как вы помните, мы установили для параметра `residual_threshold` значение `None`, поэтому для вычисления порога между инлаерами и аутлаерами RANSAC использовал MAD. Для этого набора данных MAD можно рассчитать следующим образом:

² В последние годы было разработано множество различных способов автоматического выбора хорошего порогового значения. Вы можете найти подробное описание этих способов в книге «Automatic Estimation of the Inlier Threshold in Robust Multiple Structures Fitting» by R. Toldo and A. Fusiello, Springer, 2009 (серия «Image Analysis and Processing-ICIAP 2009», p. 123–131).

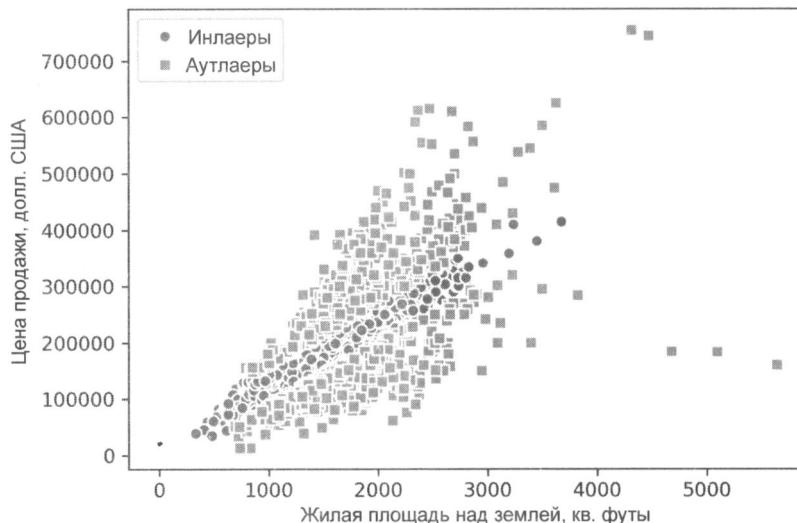


Рис. 9.9. Инлаеры и аутлаеры, выявленные с помощью модели линейной регрессии RANSAC

```
>>> def median_absolute_deviation(data):
...     return np.median(np.abs(data - np.median(data)))
>>> median_absolute_deviation(y)
37000.00
```

Следовательно, если мы хотим пометить как выбросы меньшее количество точек, необходимо выбрать значение `residual_threshold` больше, чем предыдущее MAD. Например, на рис. 9.10 показаны инлаеры и аутлаеры модели линейной регрессии RANSAC с `residual_threshold = 65 000`.

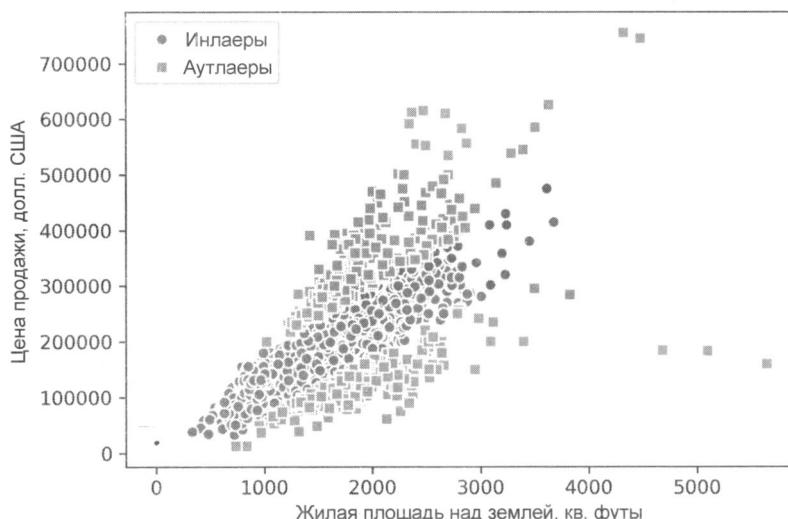


Рис. 9.10. Инлаеры и аутлаеры, определенные с помощью модели линейной регрессии RANSAC при увеличенном значении `residual_threshold`

Используя RANSAC, мы уменьшили потенциальное влияние выбросов, встречающихся в нашем наборе данных, но пока не знаем, улучшит ли этот метод прогностическую эффективность модели на новых данных или нет. Поэтому в следующем разделе мы рассмотрим различные способы получения оценки регрессионной модели, которая является важной частью построения систем для прогнозного моделирования.

9.5. Оценка производительности моделей линейной регрессии

В предыдущем разделе вы научились обучать регрессионную модель. Однако из рассмотренного ранее материала вы знаете, что крайне важно протестировать модель на данных, которые она не видела во время обучения, чтобы получить более объективную оценку ее обобщающей способности.

Возможно, вы помните из главы 6, что для проверки нужно заранее разделить набор данных на обучающий и тестовый наборы, а затем использовать первый для обучения модели, а второй — для оценки ее обобщающей способности на незнакомых данных. Но вместо того, чтобы ограничиться простой регрессионной моделью, теперь мы будем использовать все пять признаков в наборе данных и обучать модель множественной регрессии:

```
>>> from sklearn.model_selection import train_test_split
>>> target = 'SalePrice'
>>> features = df.columns[df.columns != target]
>>> X = df[features].values
>>> y = df[target].values
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=123)
>>> slr = LinearRegression()
>>> slr.fit(X_train, y_train)
>>> y_train_pred = slr.predict(X_train)
>>> y_test_pred = slr.predict(X_test)
```

Поскольку в нашей модели используется несколько независимых переменных, мы не можем визуализировать линию линейной регрессии (или, точнее, гиперплоскость) на двумерном графике, но вполне способны построить остаточные значения (разности, или расстояния по вертикали между фактическими и прогнозируемыми значениями) относительно прогнозируемых значений для диагностики нашей регрессионной модели. *Графики остатков* (residual plot) — широко используемый графический инструмент для диагностики регрессионных моделей. Они помогают обнаружить нелинейность и выбросы, а также проверить, распределены ли ошибки случайным образом.

Выполнив следующий код, мы построим график остатков, для чего просто вычтем истинные целевые переменные из наших прогнозов:

```
>>> x_max = np.max(
...     [np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min(
...     [np.min(y_train_pred), np.min(y_test_pred)])
```

```

>>> fig, (ax1, ax2) = plt.subplots(
...     1, 2, figsize=(7, 3), sharey=True)

>>> ax1.scatter(
...     y_test_pred, y_test_pred - y_test,
...     c='limegreen', marker='s',
...     edgecolor='white',
...     label='Тестовые данные')
>>> ax2.scatter(
...     y_train_pred, y_train_pred - y_train,
...     c='steelblue', marker='o', edgecolor='white',
...     label='Обучающие данные')
>>> ax1.set_ylabel('Остатки')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Предсказанные значения')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
...               color='black', lw=2)
>>> plt.tight_layout()
>>> plt.show()

```

После выполнения кода вы должны увидеть графики остатков для тестовых и обучающих наборов данных с линией, проходящей через начало оси x , как показано на рис. 9.11.

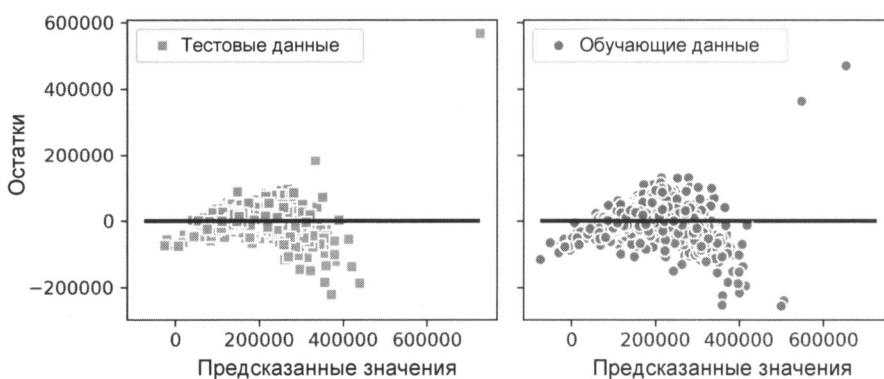


Рис. 9.11. Графики остатков для наших данных

В случае идеального предсказания остатки были бы строго нулевыми, с чем мы, вероятно, никогда не столкнемся в реальных приложениях. Однако от хорошей модели регрессии мы ожидаем, что ошибки будут распределены случайным образом, а остатки — так же случайным образом разбросаны по центральной линии. Если мы видим на графике остатков какие-то закономерности, это означает, что наша модель не смогла извлечь некоторую полезную информацию, которая просочилась в остатки, что в определенной степени можно заметить на приведенном графике. Кроме того, мы также можем использовать остаточные графики для обнаружения выбросов, которые представлены точками с большим отклонением от центральной линии.

Другой полезной количественной мерой производительности модели является *среднеквадратическая ошибка* (Mean Squared Error, MSE), которую мы рассматривали ранее как функцию потерь и старались минимизировать, чтобы обучить модель линейной регрессии. Далее представлена версия MSE без коэффициента масштабирования $^{1/2}$, которая часто используется для упрощения производной потерь при градиентном спуске:

$$MSE = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2.$$

Подобно тому как мы это делали при расчете точности прогноза в задаче классификации, мы можем использовать MSE для перекрестной проверки и выбора модели, как было показано в [главе 6](#).

Как и точность классификации, MSE принято нормализовать в зависимости от размера выборки n . Это позволяет сравнивать выборки разных размеров (например, в контексте кривых обучения).

Давайте вычислим MSE наших прогнозов при обучении и тестировании:

```
>>> from sklearn.metrics import mean_squared_error
>>> mse_train = mean_squared_error(y_train, y_train_pred)
>>> mse_test = mean_squared_error(y_test, y_test_pred)
>>> print(f'MSE при обучении: {mse_train:.2f}')
MSE при обучении: 1497216245.85
>>> print(f'MSE при тестировании: {mse_test:.2f}')
MSE при тестировании: 1516565821.00
```

Здесь видно, что MSE в обучающем наборе данных меньше, чем в тестовом наборе. Это показатель того, что наша модель в рассматриваемом случае немного склонна к переобучению. Заметим, что часто бывает интуитивно более понятно показывать ошибку в исходных единицах измерения (доллар вместо доллара в квадрате), поэтому мы можем выбрать вычисление квадратного корня из MSE, получив *среднеквадратичную ошибку*, или *среднюю абсолютную ошибку* (Mean Absolute Error, MAE), которая немножко меньше подчеркивает неправильный прогноз:

$$MAE = \frac{1}{n} \sum_{i=1}^n |y^{(i)} - \hat{y}^{(i)}|.$$

Вычислить MAE мы можем аналогично вычислению MSE:

```
>>> from sklearn.metrics import mean_absolute_error
>>> mae_train = mean_absolute_error(y_train, y_train_pred)
>>> mae_test = mean_absolute_error(y_test, y_test_pred)
>>> print(f'MAE при обучении: {mae_train:.2f}')
MAE train: 25983.03
>>> print(f'MAE при тестировании: {mae_test:.2f}')
MAE test: 24921.29
```

Основываясь на тестовом наборе MAE, мы можем сказать, что модель ошибается в среднем примерно на 25 тыс. долларов.

Используя для сравнения моделей MAE или MSE, мы должны знать, что они не имеют предельных границ, в отличие, например, от точности классификации. Другими слова-

ми, интерпретации MAE и MSE зависят от набора данных и масштабирования признаков. Например, если цены продажи представлены как кратные 1000 (с суффиксом K — от kilo), та же модель даст более низкий MAE по сравнению с моделью, которая работает с немасштабированными признаками. Чтобы лучше проиллюстрировать этот момент, рассмотрим пример:

$$|\$5000K - 550K| < |\$500\,000 - 550\,000|.$$

Поэтому иногда для лучшей интерпретируемости работы модели бывает полезнее узнать *коэффициент смешанной корреляции* (coefficient of determination, R^2), который можно рассматривать как стандартизированную версию MSE. Или, другими словами, R^2 — это доля дисперсии отклика, которая отражена моделью. Значение R^2 определяется так:

$$R^2 = 1 - \frac{SSE}{SST}.$$

Здесь SSE представляет собой сумму квадратов ошибок, которая аналогична MSE, но не включает нормализацию по размеру выборки n :

$$SSE = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2.$$

А SST — это сумма квадратов:

$$SST = \sum_{i=1}^n (y^{(i)} - \mu_y)^2.$$

Другими словами, SST — это просто дисперсия отклика.

Теперь вкратце покажем, что R^2 действительно представляет собой не что иное, как масштабированную версию MSE:

$$\begin{aligned} R^2 &= 1 - \frac{\frac{1}{n} SSE}{\frac{1}{n} SST} = \\ &= \frac{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2}{\frac{1}{n} \sum_{i=1}^n (y^{(i)} - \mu_y)^2} = \\ &= 1 - \frac{MSE}{Var(y)} \end{aligned}$$

Для обучающего набора данных R^2 находится в пределах от 0 до 1, но может стать отрицательным для тестового набора данных. Отрицательное значение R^2 означает, что регрессионная модель соответствует данным хуже, чем горизонтальная линия, представляющая среднее значение выборки. (На практике это часто происходит в случае экстремального переобучения, или когда мы забываем масштабировать тестовый набор так же, как масштабировали обучающий набор.) Если $R^2 = 1$, модель идеально соответствует данным, и $MSE = 0$.

На обучающих данных R^2 нашей модели составляет 0.77, что не идеально, но и не так уж плохо, учитывая, что мы работаем только с небольшим набором признаков. Однако R^2 на тестовом наборе данных немного меньше (0.75), что указывает на небольшое переобучение:

```
>>> from sklearn.metrics import r2_score
>>> train_r2 = r2_score(y_train, y_train_pred)
>>> test_r2 = r2_score(y_test, y_test_pred)
>>> print(f'R^2 на учебном наборе: {train_r2:.3f}, {test_r2:.3f}')
R^2 на учебном наборе: 0.77, test: 0.75
```

9.6. Использование методов регуляризации для регрессии

Как было сказано в *главе 3*, регуляризация — это один из способов решения проблемы переобучения путем добавления дополнительной информации и, таким образом, уменьшения значений параметров модели, чтобы внести штраф за сложность. Наиболее популярными методами регуляризации линейной регрессии являются так называемая *гребневая регрессия* (ridge regression, ридж-регрессия), *LASSO* и *эластичная сеть*.

Гребневая регрессия — это модель со штрафом L2, в которой мы просто добавляем квадрат суммы весов к функции потерь MSE:

$$L(\mathbf{w})_{\text{Ridge}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|\mathbf{w}\|_2^2.$$

Здесь член L2 определяется следующим образом:

$$\lambda \|\mathbf{w}\|_2^2 = \lambda \sum_{j=1}^m w_j^2.$$

Увеличивая значение гиперпараметра λ , мы увеличиваем силу регуляризации и тем самым уменьшаем веса нашей модели. Обратите внимание, что, как упоминалось в *главе 3*, смещение b не регуляризовано.

Альтернативным подходом, который может привести к разреженным моделям, является метод LASSO. В зависимости от силы регуляризации, некоторые веса могут стать равными нулю, что также делает LASSO полезным в качестве метода управляемого отбора признаков:

$$L(\mathbf{w})_{\text{Lasso}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \|\mathbf{w}\|_1.$$

Штраф L1 для LASSO определяется как сумма абсолютных величин весов модели следующим образом:

$$\lambda \|\mathbf{w}\|_1 = \lambda \sum_{j=1}^m |w_j|.$$

Однако ограничение применимости LASSO заключается в том, что он выбирает не более n признаков, если $m > n$, где n — количество обучающих примеров. Это может быть нежелательно в некоторых приложениях выбора признаков. Однако на практике это свойство LASSO часто является преимуществом, поскольку позволяет избежать насыщения моделей. *Насыщение модели* (saturation of model) происходит, если количество обучающих примеров равно количеству признаков, что является формой чрезмерной

параметризации. Как следствие, насыщенная модель всегда может идеально соответствовать обучающим данным, но представляет собой лишь вариант интерполяции, и от нее нельзя ожидать хорошего обобщения.

Компромиссом между гребневой регрессией и LASSO является эластичная сеть, которая имеет как штраф L1 для создания разреженности, так и штраф L2, поэтому ее можно использовать для отбора более чем n признаков, если $m > n$:

$$L(\mathbf{w})_{\text{Elastic Net}} = \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 + \lambda_2 \|\mathbf{w}\|_2^2 + \lambda_1 \|\mathbf{w}\|_1.$$

Все эти регуляризованные регрессионные модели доступны в библиотеке scikit-learn, и их использование аналогично обычной регрессионной модели, за исключением того, что мы должны указать силу регуляризации с помощью параметра λ , например, оптимизированного с помощью k -кратной перекрестной проверки.

Модель гребневой регрессии можно инициализировать с помощью следующего кода:

```
>>> from sklearn.linear_model import Ridge
>>> ridge = Ridge(alpha=1.0)
```

Обратите внимание, что силу регуляризации определяет параметр `alpha`, который аналогичен параметру λ . Точно так же мы можем инициализировать регрессор LASSO из субмодуля `linear_model`:

```
>>> from sklearn.linear_model import Lasso
>>> lasso = Lasso(alpha=1.0)
```

Наконец, реализация ElasticNet позволяет нам изменять соотношение L1 и L2:

```
>>> from sklearn.linear_model import ElasticNet
>>> elanet = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

Так, если мы установим `l1_ratio` равным 1.0, регрессор `ElasticNet` будет равен регрессии LASSO. Для получения более подробной информации о различных реализациях линейной регрессии обратитесь к документации по адресу: http://scikit-learn.org/stable/modules/linear_model.html.

9.7. Переход от прямой линии к кривой: полиномиальная регрессия

Ранее мы предполагали линейную зависимость между независимыми переменными и переменными отклика. Один из способов объяснить нарушение предположения о линейности — использовать модель полиномиальной регрессии, добавив полиномиальные члены:

$$y = w_1 x + w_2 x^2 + \cdots + w_d x^d + b.$$

Здесь d обозначает степень многочлена. Хотя мы можем использовать полиномиальную регрессию для моделирования нелинейной зависимости, ее по-прежнему относят к моделям множественной линейной регрессии из-за коэффициентов линейной регрессии w . Далее вы узнаете о добавлении таких полиномиальных членов в существующий набор данных и обучении модели полиномиальной регрессии.

9.7.1. Добавление полиномиальных членов с помощью scikit-learn

Давайте воспользуемся классом преобразователя `PolynomialFeatures` из библиотеки `scikit-learn`, чтобы добавить квадратичный член ($d = 2$) к простой задаче регрессии с одной независимой переменной. Затем сравним полиномиальную модель с линейной, выполнив следующие шаги:

- Добавим полиномиальный член второй степени:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = np.array([ 258.0, 270.0, 294.0, 320.0, 342.0,
...                 368.0, 396.0, 446.0, 480.0, 586.0]) \
...             [:, np.newaxis]
>>> y = np.array([ 236.4, 234.4, 252.8, 298.6, 314.2,
...                 342.2, 360.8, 368.0, 391.2, 390.8])
>>> lr = LinearRegression()
>>> pr = LinearRegression()
>>> quadratic = PolynomialFeatures(degree=2)
>>> X_quad = quadratic.fit_transform(X)
```

- Обучим простую модель линейной регрессии для сравнения:

```
>>> lr.fit(X, y)
>>> X_fit = np.arange(250, 600, 10)[:, np.newaxis]
>>> y_lin_fit = lr.predict(X_fit)
```

- Обучим модель множественной регрессии с преобразованными признаками для полиномиальной регрессии:

```
>>> pr.fit(X_quad, y)
>>> y_quad_fit = pr.predict(quadratic.fit_transform(X_fit))
```

- Отобразим результаты на графике:

```
>>> plt.scatter(X, y, label='Обучающие данные')
>>> plt.plot(X_fit, y_lin_fit,
...             label='Линейная', linestyle='--')
>>> plt.plot(X_fit, y_quad_fit,
...             label='Квадратичная')
>>> plt.xlabel('Свободные переменные')
>>> plt.ylabel('Прогнозные или известные целевые переменные')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

На полученном графике (рис. 9.12) отчетливо видно, что полиномиальная аппроксимация отражает связь между откликом и независимыми переменными намного лучше, чем линейная.

Далее мы вычислим оценочные метрики MSE и R^2 :

```
>>> y_lin_pred = lr.predict(X)
>>> y_quad_pred = pr.predict(X_quad)
>>> mse_lin = mean_squared_error(y, y_lin_pred)
```

```
>>> mse_quad = mean_squared_error(y, y_quad_pred)
>>> print(f' Обучение MSE, линейная: {mse_lin:.3f}')
f', квадратичная: {mse_quad:.3f}')
Обучение MSE, линейная: 569.780, квадратичная: 61.330
>>> r2_lin = r2_score(y, y_lin_pred)
>>> r2_quad = r2_score(y, y_quad_pred)
>>> print(f' Обучение R^2, линейная: {r2_lin:.3f}')
f', квадратичная: {r2_quad:.3f}')
Обучение R^2, линейная: 0.832, квадратичная: 0.982
```

Как можно видеть в выводе результатов выполнения кода, MSE уменьшилась с 570 (линейная аппроксимация) до 61 (квадратичная аппроксимация). Кроме того, коэффициент смешанной корреляции отражает более точное соответствие квадратичной модели ($R^2 = 0.982$) по сравнению с линейной моделью ($R^2 = 0.832$) в этой конкретной демонстрационной задаче.

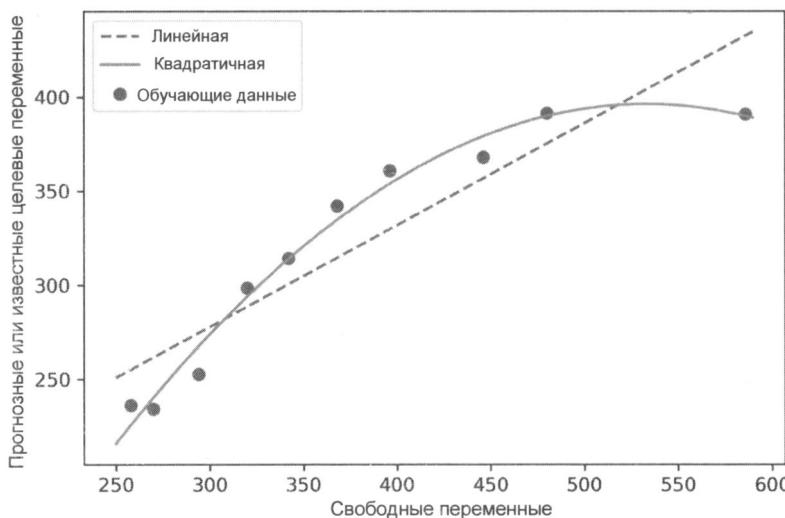


Рис. 9.12. Сравнение линейной и квадратичной моделей

9.7.2. Моделирование нелинейных отношений в наборе данных Ames Housing

В предыдущем разделе вы узнали, как создавать полиномиальные признаки, чтобы обучить модель нелинейным отношениям на демонстрационной задаче. Теперь мы рассмотрим более конкретный пример и применим эти методы к данным в наборе Ames Housing. Поэтому выполняя следующий код, мы смоделируем взаимосвязь между ценой продажи и жилой площадью над землей, используя полиномы второй степени (квадратичные) и третьей степени (кубические), и сравним результаты с линейной моделью.

А начнем мы с удаления трех выбросов с жилой площадью более 4000 квадратных футов, которые мы видели ранее (например, на рис. 9.8), чтобы эти выбросы не исказили наше регрессионное приближение:

```
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> X = X[(df['Gr Liv Area'] < 4000)]
>>> y = y[(df['Gr Liv Area'] < 4000)]

Теперь обучим регрессионные модели:

>>> regr = LinearRegression()

>>> # создаем квадратичные и кубические признаки
>>> quadratic = PolynomialFeatures(degree=2)
>>> cubic = PolynomialFeatures(degree=3)
>>> X_quad = quadratic.fit_transform(X)
>>> X_cubic = cubic.fit_transform(X)

>>> # обучаем регрессии на признаках
>>> X_fit = np.arange(X.min()-1, X.max()+2, 1)[:, np.newaxis]
>>> regr = regr.fit(X, y)
>>> y_lin_fit = regr.predict(X_fit)
>>> linear_r2 = r2_score(y, regr.predict(X))
>>> regr = regr.fit(X_quad, y)
>>> y_quad_fit = regr.predict(quadratic.fit_transform(X_fit))
>>> quadratic_r2 = r2_score(y, regr.predict(X_quad))
>>> regr = regr.fit(X_cubic, y)
>>> y_cubic_fit = regr.predict(cubic.fit_transform(X_fit))
>>> cubic_r2 = r2_score(y, regr.predict(X_cubic))

>>> # выводим результаты на график
>>> plt.scatter(X, y, label='Training points', color='lightgray')
>>> plt.plot(X_fit, y_lin_fit,
...             label=f'Linear (d=1), $R^2${linear_r2:.2f}',
...             color='blue',
...             lw=2,
...             linestyle=':')
>>> plt.plot(X_fit, y_quad_fit,
...             label=f'Quadratic (d=2), $R^2${quadratic_r2:.2f}',
...             color='red',
...             lw=2,
...             linestyle='--')
>>> plt.plot(X_fit, y_cubic_fit,
...             label=f'Cubic (d=3), $R^2${cubic_r2:.2f}',
...             color='green',
...             lw=2,
...             linestyle='--')
>>> plt.xlabel('Жилая площадь над землей, кв. футы ')
>>> plt.ylabel('Цена продажи, долл. США ')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

Получившийся график представлен на рис. 9.13.

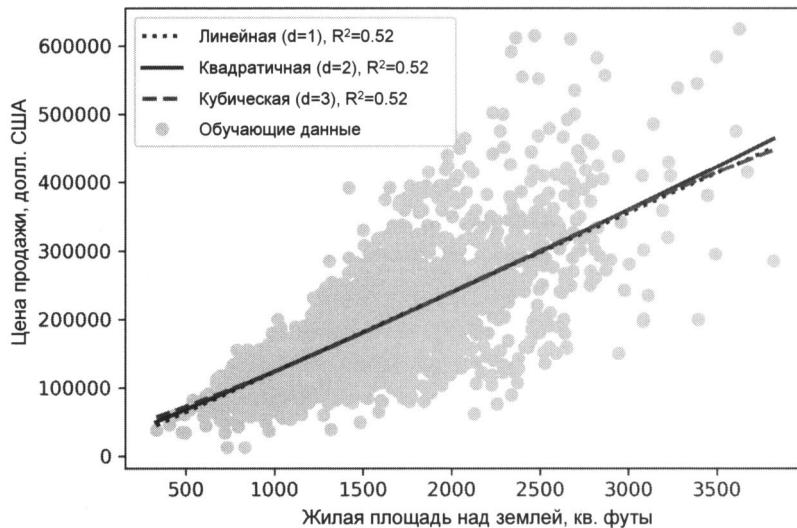


Рис. 9.13. Сравнение различных кривых регрессии, обученных на данных о цене продажи и жилой площади

Очевидно, что использование квадратичных или кубических признаков на самом деле не дает эффекта, поскольку связь между двумя переменными все равно выглядит линейной. Давайте взглянем на другой признак, а именно — на Overall Qual (общее качество). Переменная Overall Qual оценивает общее качество материалов и отделки домов и назначается по шкале от 1 до 10, где 10 — лучший результат:

```
>>> X = df[['Overall Qual']].values
>>> y = df['SalePrice'].values
```

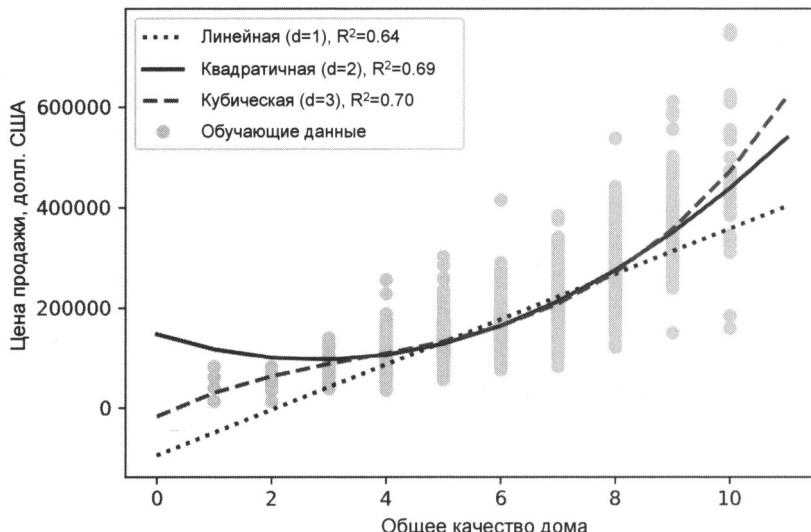


Рис. 9.14. Линейная, квадратичная и кубическая аппроксимация данных о цене продажи и общем качестве дома

Имея это определение переменных x и y , мы можем повторно использовать предыдущий код и получить график, показанный на рис. 9.14.

Как видите, квадратичная и кубическая аппроксимация лучше отражают взаимосвязь между ценой продажи и общим качеством дома, чем линейная аппроксимация. Однако вы должны учитывать, что добавление большого количества полиномиальных признаков усложняет модель и, следовательно, увеличивает вероятность переобучения. Поэтому на практике всегда рекомендуется проверять производительность модели на отдельном тестовом наборе данных, чтобы оценить качество обобщения.

9.8. Моделирование нелинейных отношений с использованием случайных лесов

В этом разделе мы рассмотрим регрессию на основе алгоритма случайного леса, которая концептуально отличается от предыдущих моделей регрессии, описанных в этой главе. Случайный лес, который представляет собой ансамбль нескольких деревьев решений, можно рассматривать как сумму кусочно-линейных функций, в отличие от глобальных моделей линейной и полиномиальной регрессии, которые мы обсуждали ранее. Другими словами, с помощью алгоритма дерева решений мы разделяем входное пространство на более мелкие области, которые становятся лучше управляемыми.

9.8.1. Регрессия на основе алгоритма дерева решений

Преимущество алгоритма дерева решений заключается в том, что он работает с произвольными признаками и не требует их преобразовывать, если мы имеем дело с нелинейными данными, поскольку деревья решений анализируют один признак за раз, а не рассматривают взвешенные комбинации. (Точно так же для деревьев решений не требуется нормализация или стандартизация признаков.) Как отмечалось в главе 3, мы выращиваем дерево решений, итеративно разбивая его узлы до тех пор, пока листья не станут чистыми, или не будет достигнут критерий остановки. Используя деревья решений для задач классификации, мы определили энтропию как меру примесей, чтобы выяснить, какое разделение признаков максимизирует *прирост информации* (Information Gain, IG), который для двоичного ветвления можно записать следующим образом:

$$IG(D_p, x_i) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}).$$

Здесь x_i — функция для выполнения разделения, N_p — количество обучающих примеров в родительском узле, I — функция примесей, D_p — подмножество обучающих примеров в родительском узле, а D_{left} и D_{right} — подмножества обучающих примеров в левом и правом дочерних узлах после ветвления. Помните, что наша цель — найти ветвление признаков, которое максимизирует прирост информации. Другими словами, мы хотим найти ветвление признаков, которое лучше всего уменьшает примеси в дочерних узлах. В главе 3 мы обсуждали примесь Джини и энтропию как меры примеси, которые являются полезными критериями для задач классификации. Однако чтобы использовать дерево решений для регрессии, нам нужна метрика примеси, подходящая для непрерывных переменных, поэтому теперь мы определяем меру примеси узла t как MSE:

$$I(t) = MSE(t) = \frac{1}{N_t} \sum_{i \in D_t} (y^{(i)} - \hat{y}_t)^2.$$

Здесь N_t — количество обучающих примеров в узле t , D_t — обучающее подмножество в узле t , $y^{(i)}$ — истинное целевое значение, а \hat{y}_t — прогнозируемое целевое значение (выборочное среднее):

$$\hat{y}_t = \frac{1}{N_t} \sum_{i \in D_t} y^{(i)}.$$

В контексте регрессии на основе дерева решений MSE часто называют *дисперсией внутри узла*, поэтому критерий разделения также известен как *уменьшение дисперсии*.

Чтобы увидеть, как выглядит линия регрессии, сформированная деревом решений, воспользуемся реализованным в scikit-learn классом DecisionTreeRegressor и смоделируем взаимосвязь между переменными SalePrice и Gr Living Area. Заметим, что SalePrice и Gr Living Area не обязательно демонстрируют нелинейную связь, но эта комбинация функций все же довольно хорошо иллюстрирует общие аспекты дерева регрессии:

```
>>> from sklearn.tree import DecisionTreeRegressor
>>> X = df[['Gr Liv Area']].values
>>> y = df['SalePrice'].values
>>> tree = DecisionTreeRegressor(max_depth=3)
>>> tree.fit(X, y)
>>> sort_idx = X.flatten().argsort()
>>> lin_regplot(X[sort_idx], y[sort_idx], tree)
>>> plt.xlabel('Жилая площадь над землей, кв. футы ')
>>> plt.ylabel('Цена продажи, долл. США')>>> plt.show()
```

На полученном графике (рис. 9.15) мы видим, что дерево решений отражает общую тенденцию в данных. Можно предположить, что дерево регрессии также относительно хорошо выявляет тенденции в нелинейных данных. Однако ограничение этой модели заключается в том, что она не отражает непрерывность и дифференцируемость ожидаемого прогноза. Кроме того, нам нужно соблюдать осторожность при выборе подход-

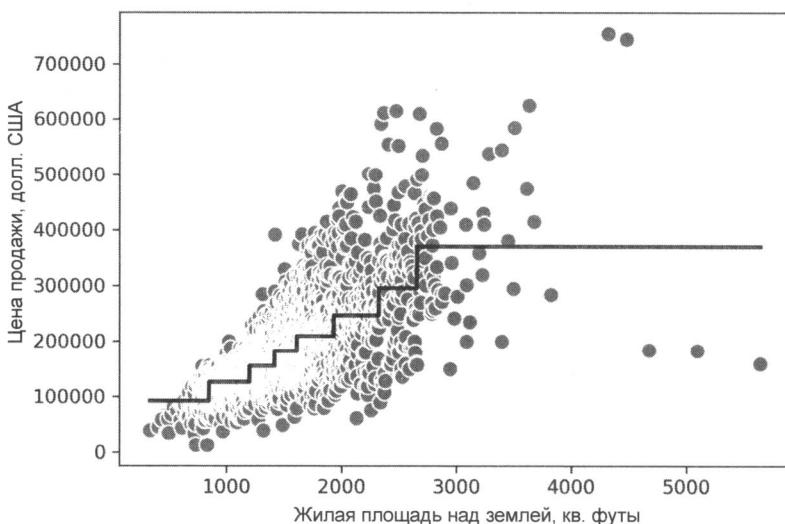


Рис. 9.15. График регрессии на основе дерева решений

дящего значения глубины дерева, чтобы избежать переобучения или недообучения (в нашем случае глубина 3 оказалась хорошим выбором).

Попробуйте провести эксперименты с более глубокими деревьями решений. Обратите внимание, что взаимосвязь между Gr Living Area И SalePrice довольно линейна, поэтому вам также рекомендуется вместо этого применить дерево решений к переменной Overall Qual.

В следующем разделе мы рассмотрим более надежный способ обучения деревьев регрессии — случайные леса.

9.8.2. Регрессия на основе случайного леса

Как вы узнали из главы 3, алгоритм случайного леса представляет собой ансамбль метод, объединяющий несколько деревьев решений. Обобщающая способность случайного леса обычно лучше, чем у отдельного дерева решений, благодаря случайности, которая помогает уменьшить дисперсию модели. Другие преимущества случайных лесов заключаются в том, что они менее чувствительны к выбросам в наборе данных и не нуждаются в кропотливой настройке параметров. Единственный параметр случайного леса, с которым нам обычно приходится экспериментировать, — это количество деревьев в ансамбле. Базовый алгоритм случайного леса для регрессии почти идентичен алгоритму случайного леса для классификации, который мы обсуждали в главе 3. Единственное отличие состоит в том, что мы используем критерий MSE для вычисления отдельных деревьев решений, а прогнозируемая целевая переменная вычисляется как средний прогноз по всем деревьям решений.

Теперь давайте воспользуемся всеми признаками набора данных Ames Housing, чтобы обучить модель регрессии случайного леса на 70 процентах примеров и оценить ее производительность на оставшихся 30 процентах, как мы сделали ранее в разд. 9.5. Код выглядит следующим образом:

```
>>> target = 'SalePrice'  
>>> features = df.columns[df.columns != target]  
>>> X = df[features].values  
>>> y = df[target].values  
>>> X_train, X_test, y_train, y_test = train_test_split(  
...     X, y, test_size=0.3, random_state=123)  
  
>>> from sklearn.ensemble import RandomForestRegressor  
>>> forest = RandomForestRegressor(  
...     n_estimators=1000,  
...     criterion='squared_error',  
...     random_state=1,  
...     n_jobs=-1)  
>>> forest.fit(X_train, y_train)  
>>> y_train_pred = forest.predict(X_train)  
>>> y_test_pred = forest.predict(X_test)  
>>> mae_train = mean_absolute_error(y_train, y_train_pred)  
>>> mae_test = mean_absolute_error(y_test, y_test_pred)  
>>> print(f'MAE при обучении: {mae_train:.2f}')  
MAE при обучении: 8305.18
```

```
>>> print(f'MAE при тестировании: {mae_test:.2f}')
MAE при тестировании: 20821.77
>>> r2_train = r2_score(y_train, y_train_pred)
>>> r2_test = r2_score(y_test, y_test_pred)
>>> print(f'R^2 при обучении: {r2_train:.2f}')
R^2 при обучении: 0.98
>>> print(f'R^2 при тестировании: {r2_test:.2f}')
R^2 при тестировании: 0.85
```

К сожалению, мы видим, что случайный лес склонен к переобучению на обучающих данных. Тем не менее он все еще может относительно хорошо объяснить взаимосвязь между целевыми и независимыми переменными ($R^2 = 0.85$ в тестовом наборе данных). Для сравнения: линейная модель из предыдущего раздела, обученная на том же наборе данных, меньше переобучалась, но хуже работала на тестовом наборе ($R^2 = 0.75$).

Наконец, давайте взглянем на прогнозные остатки:

```
>>> x_max = np.max([np.max(y_train_pred), np.max(y_test_pred)])
>>> x_min = np.min([np.min(y_train_pred), np.min(y_test_pred)])

>>> fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(7, 3), sharey=True)
>>> ax1.scatter(y_test_pred, y_test_pred - y_test,
...               c='limegreen', marker='s', edgecolor='white',
...               label='Тестовые данные')
>>> ax2.scatter(y_train_pred, y_train_pred - y_train,
...               c='steelblue', marker='o', edgecolor='white',
...               label='Обучающие данные')
>>> ax1.set_ylabel('Остатки')

>>> for ax in (ax1, ax2):
...     ax.set_xlabel('Прогнозные значения')
...     ax.legend(loc='upper left')
...     ax.hlines(y=0, xmin=x_min-100, xmax=x_max+100,
...               color='black', lw=2)

>>> plt.tight_layout()
>>> plt.show()
```

В соответствии с коэффициентом R^2 мы видим, что на тестовых данных модель работает хуже, на что указывают выбросы в направлении оси y . Кроме того, распределение остатков не выглядит полностью случайным вокруг нулевой центральной точки, а это означает, что модель не способна извлечь из данных всю исследовательскую информацию. Однако график остатков (рис. 9.16) демонстрирует значительное улучшение по сравнению с аналогичным графиком линейной модели, который мы построили ранее в этой главе.

В идеале ошибка нашей модели должна быть случайной или непредсказуемой. Другими словами, ошибка прогнозов не должна быть связана ни с какой информацией, содержащейся в независимых переменных, — напротив, она должна отражать случайность реальных распределений. Если мы находим закономерности в ошибках предсказания, просматривая, например, график остатков, это означает, что они содержат про-

гностическую информацию. Распространенной причиной этого является «просачивание» полезной информации в остатки.

К сожалению, универсального подхода для борьбы с закономерностями в графиках остатков не существует, и в каждом новом случае нужны эксперименты. В зависимости от доступных нам данных мы можем улучшить модель, преобразовав переменные, настроив гиперпараметры алгоритма обучения, выбрав более простые или более сложные модели, удалив выбросы или добавив дополнительные переменные.

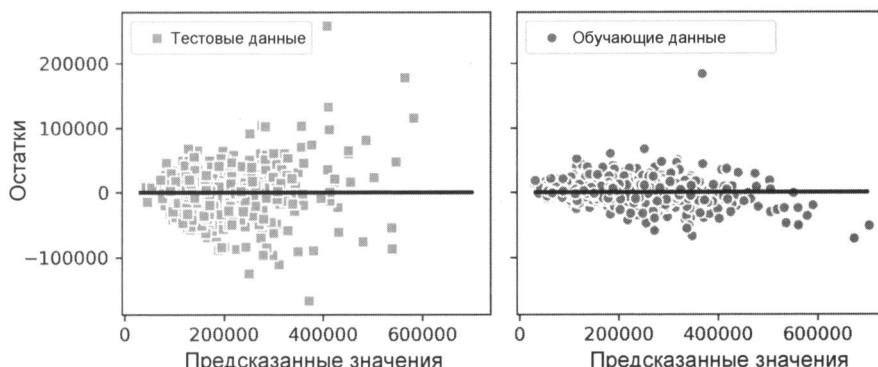


Рис. 9.16. Остатки регрессии, полученной по методу случайного леса

9.9. Заключение

В начале этой главы вы узнали о простом линейном регрессионном анализе для моделирования взаимосвязи между одной независимой переменной и непрерывной переменной отклика. Затем мы обсудили полезную технику объяснительного анализа для выявления закономерностей и аномалий в данных, что является важным первым шагом в задачах прогнозного моделирования.

Мы построили нашу первую модель, реализовав линейную регрессию с оптимизацией на основе градиента. Затем разобрались, как использовать линейные модели scikit-learn для задач регрессии, а также применять метод устойчивой регрессии (RANSAC) в качестве способа противодействия выбросам в данных. Для оценки прогностической эффективности регрессионных моделей мы вычислили среднюю сумму квадратов ошибок и соответствующую метрику R^2 . Кроме того, вы освоили полезный графический инструмент для диагностики проблем регрессионных моделей — график остатков.

Изучив применение регуляризации к регрессионным моделям, чтобы уменьшить сложность модели и избежать переобучения, мы затем рассмотрели несколько подходов к моделированию нелинейных отношений, включая полиномиальное представление признаков и регрессоры случайного леса.

В предыдущих главах мы подробно обсуждали обучение с учителем, классификацию и регрессионный анализ. В следующей главе вы узнаете о еще одной интересной области машинного обучения — обучении без учителя, а также о том, как использовать кластерный анализ для поиска скрытых структур в данных при отсутствии целевых переменных.

10

Работа с неразмеченными данными: кластерный анализ

В предыдущих главах мы использовали методы обучения с учителем для построения моделей машинного обучения, опираясь на готовые ответы, — метки классов уже были доступны в обучающих данных. В этой главе мы переключимся на данные без меток и исследуем кластерный анализ — категорию методов *обучения без учителя*, которые позволяют нам обнаруживать скрытые структуры в данных, где мы заранее не знаем правильный ответ. Цель кластеризации — найти естественное разбиение данных на группы, чтобы элементы в одном кластере были больше похожи друг на друга, чем на элементы из других кластеров.

Кластеризация — захватывающая исследовательская тема. В этой главе вы узнаете о следующих приемах, которые помогают организовать данные в осмысленные структуры:

- ◆ поиск центров сходства с помощью популярного алгоритма *k*-средних;
- ◆ восходящий способ построения иерархических деревьев кластеризации;
- ◆ распознавание объектов произвольной формы с использованием метода кластеризации на основе плотности.

10.1. Группировка объектов по сходству с использованием *k*-средних

В этом разделе будет рассказано об одном из самых популярных алгоритмов кластеризации — методе *k*-средних, который широко используется как в научных кругах, так и в промышленности. Кластеризация (или кластерный анализ) — это метод, который позволяет нам находить группы похожих объектов, которые больше связаны друг с другом, чем с объектами в других группах. К примерам бизнес-приложений кластеризации можно отнести группировку документов, музыки и фильмов по разным темам или поиск клиентов с общими интересами на основе общего покупательского поведения в качестве основы для рекомендательных механизмов.

10.1.1. Кластеризация методом *k*-средних с использованием scikit-learn

Как вы вскоре увидите, алгоритм *k*-средних не только чрезвычайно прост в реализации, но он также очень эффективен в вычислительном отношении по сравнению с другими

алгоритмами кластеризации, что может объяснить его популярность. Алгоритм k-средних относится к категории кластеризации *на основе прототипов*.

Позднее в этой главе мы обсудим две другие категории кластеризации: *иерархическую* и *на основе плотности*.

Кластеризация на основе прототипов означает, что каждый кластер представлен прототипом, который обычно является либо *центроидом* (средним) похожих точек с непрерывными характеристиками, либо *медиоидом* (наиболее репрезентативным элементом или точкой, которая минимизирует расстояние до всех других точек, принадлежащих к конкретному кластеру) в случае категориальных признаков. Хотя метод k-средних очень хорош для идентификации кластеров сферической формы, одним из недостатков этого алгоритма является необходимость заранее указать конкретное количество кластеров k . Неправильный выбор k может привести к снижению качества кластеризации. Позже в этой главе мы обсудим методы «локтя» и силуэтных графиков, которые являются полезными способами оценки качества кластеризации и помогают нам определить оптимальное количество кластеров k .

Хотя кластеризация методом k-средних может применяться к данным в многомерных пространствах, для наглядности мы рассмотрим следующие примеры, используя простой двумерный набор данных:

```
>>> from sklearn.datasets import make_blobs  
>>> X, y = make_blobs(n_samples=150,  
...                     n_features=2,  
...                     centers=3,  
...                     cluster_std=0.5,  
...                     shuffle=True,  
...                     random_state=0)  
>>> import matplotlib.pyplot as plt  
>>> plt.scatter(X[:, 0],  
...               X[:, 1],  
...               c='white',  
...               marker='o',  
...               edgecolor='black',  
...               s=50)  
>>> plt.xlabel('Признак 1')  
>>> plt.ylabel('Признак 2')  
>>> plt.grid()  
>>> plt.tight_layout()  
>>> plt.show()
```

Набор данных, который мы только что создали, состоит из 150 случайно сгенерированных точек, которые примерно сгруппированы в три области с более высокой плотностью, что можно наглядно продемонстрировать с помощью двумерной диаграммы рассеяния (рис. 10.1).

В реальных приложениях кластеризации у нас нет никакой достоверной информации о категории этих точек данных (т. е. информации, предоставленной в качестве эмпирического доказательства, а не вывода), — ведь если бы нам дали метки классов, эта задача попала бы в категорию обучения с учителем. Следовательно, наша цель состоит в том, чтобы сгруппировать примеры, ничего не зная о их принадлежности к клас-

сам, — исключительно на основе сходства их признаков. Этую группировку можно выполнить с помощью следующих четырех шагов алгоритма k -средних:

1. Случайным образом выбрать k центроидов из примеров в качестве начальных центров кластеров.
2. Сопоставить каждый экземпляр с ближайшим центроидом $\mu^{(j)}, j \in \{1, \dots, k\}$.
3. Переместить центроиды в центр группы экземпляров, которые были ему присвоены.
4. Повторять шаги 2 и 3 до тех пор, пока состав кластеров не перестанет изменяться или не будет достигнут заданный пользователем допуск или максимальное количество итераций.

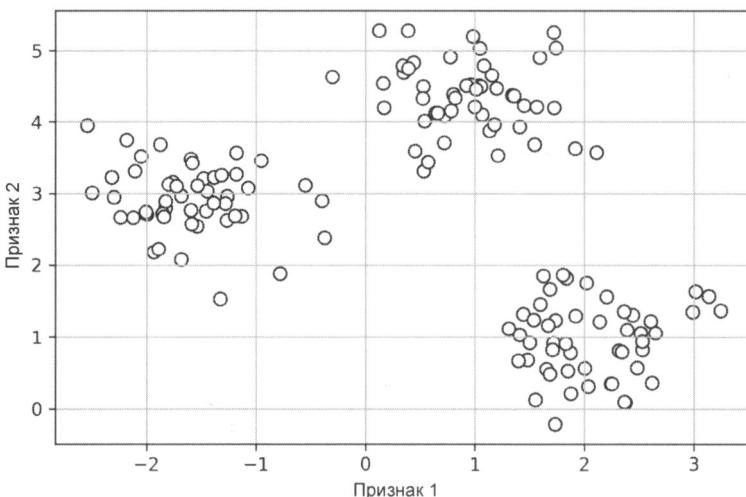


Рис. 10.1. Диаграмма рассеяния нашего неразмеченногонабора данных

Теперь следующий вопрос: как измерить сходство между объектами? Мы можем определить сходство как величину, обратную расстоянию. В качестве расстояния при кластеризации экземпляров с непрерывными признаками обычно используют квадрат евклидова расстояния между двумя точками x и y в m -мерном пространстве:

$$d(x, y)^2 = \sum_{j=1}^m (x_j - y_j)^2 = \|x - y\|_2^2.$$

Учтите, что в этом уравнении индекс j обозначает j -е измерение (столбец признаков) входных данных x и y . В оставшейся части этого раздела мы будем использовать верхние индексы i и j для обозначения индекса примера (записи данных) и индекса кластера соответственно.

Основываясь на этой метрике евклидова расстояния, мы можем описать алгоритм k -средних как простую задачу оптимизации — итеративный подход к минимизации внутрикластерной суммы квадратических ошибок (SSE), которую иногда также называют *инерцией кластера*:

$$SSE = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)} \|x^{(i)} - \mu^{(j)}\|_2^2.$$

Здесь $\mu^{(j)}$ — репрезентативная точка (центроид) кластера j , $w^{(i,j)} = 1$, если пример $x^{(i)}$ находится в кластере j , или 0 в противном случае:

$$w^{(i,j)} = \begin{cases} 1, & \text{если } x^{(i)} \in j \\ 0, & \text{в ином случае} \end{cases}$$

Давайте воспользуемся нашим знанием работы алгоритма k-средних и применим его к нашему демонстрационному набору данных, используя класс KMeans из модуля cluster scikit-learn:

```
>>> from sklearn.cluster import KMeans
>>> km = KMeans(n_clusters=3,
...                 init='random',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
```

Здесь мы установили желаемое количество кластеров равным 3 (необходимость заранее указывать количество кластеров является одним из ограничений метода k-средних). Установили `n_init=10`, чтобы выполнить 10 независимых запусков алгоритма кластеризации k-средних с разными случайными центроидами, а затем выбрать модель с наименьшим значением SSE в качестве окончательного варианта. Параметр `max_iter` задает максимальное количество итераций для каждого отдельного запуска (здесь 300). Учитите, что реализация алгоритма k-средних в scikit-learn останавливается раньше, если она сходится до достижения максимального количества итераций. Однако возможна ситуация, когда алгоритму k-средних не удается сойтись в каком-то запуске. Если было выбрано относительно большое значение `max_iter`, это может привести к большим вычислительным затратам. Один из способов справиться с проблемами сходимости — выбрать более высокие значения для параметра `tol`, который определяет допуск минимальных изменений внутрикластерной SSE для принятия решения о сходимости. В приведенном коде мы выбрали допуск `1e-04` (= 0.0001).

Одна из проблем алгоритма k-средних заключается в том, что один или несколько кластеров могут быть пустыми. Эта проблема не существует для алгоритма k-метроидов, или нечетких C-средних, который мы обсудим позже в этом разделе. Впрочем, об этой ситуации не забыли и в текущей реализации алгоритма k-средних в scikit-learn. Если кластер пуст, алгоритм будет искать точку данных, наиболее удаленную от центроида пустого кластера. Затем в качестве нового центроида будет назначена эта самая дальняя точка.



Масштабирование признаков

Применяя алгоритм k-средних к реальным данным с использованием метрики евклидова расстояния, необходимо убедиться, что признаки измеряются в одном масштабе, и при необходимости применить z-стандартизацию или минимаксное масштабирование.

Итак, мы спрогнозировали метки кластеров `y_km` и обсудили некоторые проблемы алгоритма k-средних. Теперь давайте построим визуальное представление кластеров, кото-

рые алгоритм k-средних обнаружил в наборе данных, вместе с центроидами кластера. Они хранятся в атрибуте `cluster_centers_` обученного объекта KMeans:

```
>>> plt.scatter(X[y_km == 0, 0],
...                 X[y_km == 0, 1],
...                 s=50, c='lightgreen',
...                 marker='s', edgecolor='black',
...                 label='Кластер 1')
>>> plt.scatter(X[y_km == 1, 0],
...                 X[y_km == 1, 1],
...                 s=50, c='orange',
...                 marker='o', edgecolor='black',
...                 label='Кластер 2')
>>> plt.scatter(X[y_km == 2, 0],
...                 X[y_km == 2, 1],
...                 s=50, c='lightblue',
...                 marker='v', edgecolor='black',
...                 label='Кластер 3')
>>> plt.scatter(km.cluster_centers_[:, 0],
...                 km.cluster_centers_[:, 1],
...                 s=250, marker='*',
...                 c='red', edgecolor='black',
...                 label='Центроиды')
>>> plt.xlabel('Признак 1')
>>> plt.ylabel('Признак 2')
>>> plt.legend(scatterpoints=1)
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

На рис. 10.2 вы можете видеть, что алгоритм k-средних поместил три центроида в центр каждой сферы, что для имеющегося набора данных выглядит вполне разумно.

Хотя метод k-средних хорошо справился с этим демонстрационным набором данных, мы не избавились от необходимости заранее указывать количество кластеров k . В реальных приложениях количество кластеров далеко не всегда так очевидно, особенно если мы работаем с многомерным набором данных, который невозможно визуализировать. Другие важные допущения алгоритма k-средних заключаются в том, что кластеры не перекрываются и не являются иерархическими, и мы также предполагаем, что в каждом кластере есть хотя бы один элемент. Позже в этой главе мы увидим другие типы алгоритмов кластеризации: иерархическую кластеризацию и кластеризацию на основе плотности. Ни один из этих алгоритмов не требует, чтобы мы заранее указывали количество кластеров или предполагали сферические структуры в нашем наборе данных.

Сейчас же мы рассмотрим популярный вариант классического алгоритма k-средних под названием k-средних++. Хотя он не устраняет предположения и недостатки алгоритма k-средних, упомянутые в предыдущем абзаце, он может значительно улучшить результаты кластеризации за счет более разумного размещения начальных центров кластеров.

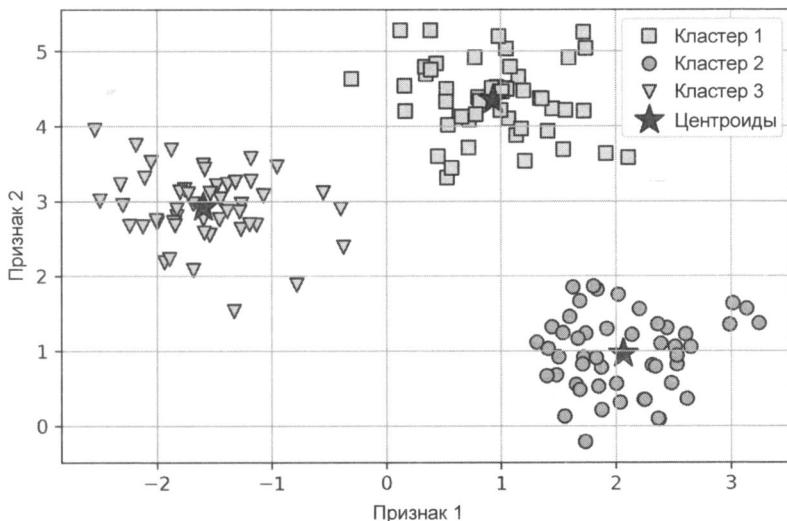


Рис. 10.2. Кластеры, найденные алгоритмом k-средних, и их центроиды

10.1.2. Более разумный способ размещения начальных центроидов: алгоритм k-средних++

До сих пор мы обсуждали классический алгоритм k-средних, который использует случайное начальное число для размещения начальных центроидов, что иногда может приводить к плохой кластеризации или медленной сходимости, если начальные центроиды выбраны неудачно. Один из способов решения этой проблемы — запустить алгоритм k-средних несколько раз на одном и том же наборе данных и выбрать наиболее эффективную модель по критерию SSE.

Другая стратегия заключается в том, чтобы разместить начальные центроиды далеко друг от друга с помощью алгоритма k-средних++, что приводит к лучшим и более обоснованным результатам, чем в случае классического алгоритма k-средних¹.

Инициализация в алгоритме k-средних++ происходит следующим образом:

1. Инициализируем пустое множество \mathbf{M} для хранения k выбранных центроидов.
2. Случайным образом выбираем первый центроид $\mu^{(i)}$ из входных данных и сохраняем его в \mathbf{M} .
3. Для каждой точки данных $x^{(i)}$, не входящей в \mathbf{M} , находим минимальный квадрат расстояния $d(x^{(i)}, \mathbf{M})^2$ до любого из центроидов в \mathbf{M} .
4. Чтобы случайным образом выбрать следующий центроид $\mu^{(p)}$, используем взвешенное распределение вероятностей, равное $\frac{d(\mu^{(p)}, \mathbf{M})^2}{\sum_i d(x^{(i)}, \mathbf{M})^2}$. Например, можно собрать

¹ См. «k-means++: The Advantages of Careful Seeding» by D. Arthur and S. Vassilvitskii in Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms, p. 1027–1035. Society for Industrial and Applied Mathematics, 2007.

все точки в массив и построить взвешенную случайную выборку таким образом, что чем больше квадрат расстояния, тем больше вероятность того, что точка станет следующим центроидом.

5. Повторяем шаги 3 и 4, пока не будут выбраны k центроидов.

6. Запускаем классический алгоритм k-средних.

Чтобы использовать алгоритм k-средних++ с объектом `kMeans` scikit-learn, достаточно установить значение параметра `init = 'k-means++'`. На самом деле '`k-means++`' является для параметра `init` аргументом по умолчанию, который настоятельно рекомендуется на практике. Единственная причина, по которой мы не использовали его в предыдущем примере, заключалась в том, чтобы не вводить сразу слишком много понятий. В оставшейся части этого раздела, посвященного методу k-средних, мы будем использовать алгоритм k-средних++, но вам будет полезно провести больше экспериментов с двумя разными подходами к размещению начальных центроидов кластера (классический вариант через `init='random'` и k-средних++ через `init='k-means++'`).

10.1.3. Жесткая и мягкая кластеризация

Жесткая кластеризация (hard clustering) — это семейство алгоритмов, в которых каждая точка набора данных назначается ровно одному кластеру, как в алгоритмах k-среднего и k-среднего++, которые мы обсуждали ранее в этой главе. Напротив, алгоритмы *мягкой кластеризации* (иногда также называемой *нечеткой кластеризацией*) назначают точку данных одному или нескольким кластерам. Популярным примером мягкой кластеризации является алгоритм *нечетких C-средних* (Fuzzy C-Means, FCM) (также называемый *мягким k-средним*, или *нечетким k-средним*). Первоначальная идея родилась в 1970-х годах, когда Джозеф Данн впервые предложил вариант нечеткой кластеризации для улучшения алгоритма k-средних². Почти десятилетие спустя Джеймс Бедзек опубликовал свою работу по усовершенствованию алгоритма нечеткой кластеризации, который теперь известен как алгоритм FCM³.

Принцип работы FCM очень похож на алгоритм k-средних. Однако мы заменяем жесткое назначение кластеров вероятностями для каждой точки, принадлежащей каждому кластеру. В алгоритме k-средних мы могли бы выразить кластерную принадлежность точки x с помощью разреженного вектора двоичных значений:

$$\begin{cases} x \in \mu^{(1)} & \rightarrow w^{(i,j)} = 0 \\ x \in \mu^{(2)} & \rightarrow w^{(i,j)} = 1 \\ x \in \mu^{(3)} & \rightarrow w^{(i,j)} = 0 \end{cases}.$$

Здесь позиция индекса со значением 1 указывает центр тяжести кластера $\mu^{(j)}$, которому назначена точка (полагая, что $k = 3, j \in \{1, 2, 3\}$). В свою очередь, вектор принадлежности в FCM можно записать следующим образом:

$$\begin{cases} x \in \mu^{(1)} & \rightarrow w^{(i,j)} = 0.1 \\ x \in \mu^{(2)} & \rightarrow w^{(i,j)} = 0.85 \\ x \in \mu^{(3)} & \rightarrow w^{(i,j)} = 0.05 \end{cases}.$$

² См. «A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters», 1973.

³ См. «Pattern Recognition with Fuzzy Objective Function Algorithms», Springer Science+Business Media, 2013.

Здесь каждое значение попадает в диапазон $[0, 1]$ и представляет собой вероятность принадлежности к соответствующему центроиду кластера. Сумма вероятностей членства для точки равна 1. Как и в случае с алгоритмом k-средних, мы можем свести алгоритм FCM к четырем ключевым этапам:

1. Задаем количество центроидов k и случайным образом назначаем каждой точке данное членство в кластере.
2. Вычисляем центроиды кластера $\mu^{(j)}, j \in \{1, 2, 3\}$.
3. Обновляем членство в кластере для каждой точки.
4. Повторяем шаги 2 и 3 до тех пор, пока коэффициенты принадлежности не перестанут изменяться или не будет достигнут заданный пользователем допуск или максимальное количество итераций.

Целевая функция FCM — мы обозначим ее как J_m — очень похожа на внутрикластерную SSE, которую мы минимизируем в алгоритме k-средних:

$$J_m = \sum_{i=1}^n \sum_{j=1}^k w^{(i,j)m} \|x^{(i)} - \mu^{(j)}\|_2^2$$

Однако обратите внимание, что индикатор принадлежности $w^{(i,j)}$ не является двоичным значением, как в k-средних ($w^{(i,j)} \in \{0, 1\}$), а представляет собой действительное значение, обозначающее вероятность принадлежности к кластеру ($w^{(i,j)} \in [0, 1]$). Вы также, возможно, заметили, что мы добавили к $w^{(i,j)}$ дополнительный показатель степени m — это число, большее или равное единице (обычно $m = 2$), является так называемым *коэффициентом нечеткости* (или просто *фаззификатором*), от которого зависит степень нечеткости.

Чем больше значение m , тем меньше становится вес членства в кластере $w^{(i,j)}$, что приводит к формированию более нечетких кластеров. Сама вероятность принадлежности к кластеру рассчитывается так:

$$w^{(i,j)} = \left[\sum_{c=1}^k \left(\frac{\|x^{(i)} - \mu^{(j)}\|_2}{\|x^{(i)} - \mu^{(c)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}.$$

Например, если мы выбрали три центра кластера, как в предыдущем примере k-средних, то могли бы вычислить принадлежность $x^{(i)}$, относящегося к $\mu^{(j)}$, следующим образом:

$$w^{(i,j)} = \left[\left(\frac{\|x^{(i)} - \mu^{(j)}\|_2}{\|x^{(i)} - \mu^{(1)}\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\|x^{(i)} - \mu^{(j)}\|_2}{\|x^{(i)} - \mu^{(2)}\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\|x^{(i)} - \mu^{(j)}\|_2}{\|x^{(i)} - \mu^{(3)}\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}.$$

Центр $\mu^{(j)}$ самого кластера вычисляется как среднее значение всех точек, взвешенных по степени принадлежности каждой точки к этому кластеру ($w^{(i,j)m}$):

$$\mu^{(j)} = \frac{\sum_{i=1}^n w^{(i,j)m} x^{(i)}}{\sum_{i=1}^n w^{(i,j)m}}.$$

Достаточно взглянуть на уравнение для расчета членства в кластере, чтобы сделать вывод, что каждая итерация в FCM обходится дороже, чем итерация в k-средних. С дру-

гой стороны, FCM обычно требует меньшего количества итераций для достижения сходимости. Однако на практике было обнаружено, что и k-средние, и FCM, дают очень похожие результаты кластеризации⁴. К сожалению, в настоящее время алгоритм FCM не реализован в scikit-learn, но заинтересованные читатели могут попробовать реализацию FCM из пакета scikit-fuzzy, который доступен по адресу: <https://github.com/scikit-fuzzy/scikit-fuzzy>.

10.1.4. Использование метода локтя для нахождения оптимального количества кластеров

Одна из основных проблем обучения без учителя заключается в том, что мы не знаем правильного ответа. В нашем наборе данных нет эталонных меток классов, позволяющих нам задействовать методы наподобие тех, что применялись в главе 6 для оценки производительности модели, обучаемой с учителем. Поэтому для количественной оценки качества кластеризации нам необходимо использовать внутренние метрики — такие как внутрикластерная SSE (искажение). Опираясь на эти метрики, мы сможем сравнить производительность различных моделей кластеризации k-средних.

К счастью, нам не нужно явно вычислять внутрикластерную SSE, когда мы используем scikit-learn, т. к. искомое значение уже доступно через атрибут `inertia_` после обучения модели KMeans:

```
>>> print(f'Искажение: {km.inertia_:.2f}')
Искажение: 72.48
```

Зная внутрикластерную SSE, мы можем использовать так называемый *метод локтя* (elbow method), чтобы оценить оптимальное количество кластеров k для той или иной задачи. Известно, что если k увеличить, искажение уменьшится. Это связано с тем, что точки будут ближе к центроидам, которым они назначены. Идея метода локтя состоит в том, чтобы определить значение k , при котором искажение начинает увеличиваться наиболее быстро, что станет гораздо очевиднее, если мы построим график искажения для различных значений k :

```
>>> distortions = []
>>> for i in range(1, 11):
...     km = KMeans(n_clusters=i,
...                  init='k-means++',
...                  n_init=10,
...                  max_iter=300,
...                  random_state=0)
...     km.fit(X)
...     distortions.append(km.inertia_)
>>> plt.plot(range(1,11), distortions, marker='o')
>>> plt.xlabel('Количество кластеров')
>>> plt.ylabel('Искажение')
>>> plt.tight_layout()
>>> plt.show()
```

⁴ См. «Comparative Analysis of k-means and Fuzzy C-Means Algorithms» by S. Ghosh and S. K. Dubey, IJACSA, 4: 35–38, 2013.

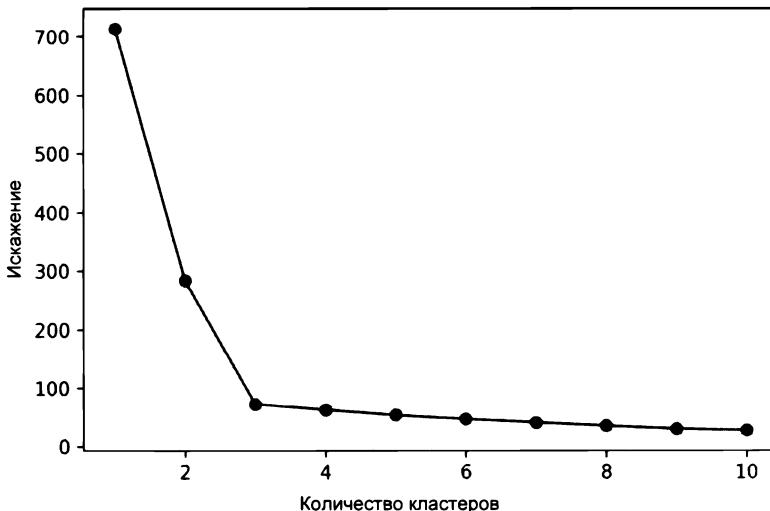


Рис. 10.3. Поиск оптимального количества кластеров с помощью метода локтя

Как можно видеть на рис. 10.3, «локоть» образуется при $k = 3$. Отсюда следует, что $k = 3$ действительно является хорошим выбором для текущего набора данных.

10.1.5. Количественная оценка качества кластеризации с помощью силуэтных графиков

Еще одной внутренней метрикой для оценки качества кластеризации является *силуэтный анализ* (silhouette analysis), который также может применяться к алгоритмам кластеризации, отличным от k-средних, которые мы обсудим позже в этой главе. Силуэтный анализ можно использовать в качестве графического инструмента для измерения того, насколько плотно сгруппированы точки данных в кластерах. Чтобы вычислить *силуэтный коэффициент* (silhouette coefficient) одной точки в нашем наборе данных, нужно выполнить следующие три шага:

1. Вычислить *компактность кластера* (cluster cohesion, иногда говорят *связность кластера*) $a^{(i)}$ как среднее расстояние между точкой $x^{(i)}$ и всеми остальными точками в том же кластере.
2. Вычислить *отделимость кластера* (cluster separation) $b^{(i)}$ от следующего ближайшего кластера как среднее расстояние между точкой $x^{(i)}$ и всеми точками в ближайшем кластере.
3. Вычислить силуэтный коэффициент $s^{(i)}$ как разницу между компактностью и отделимостью кластера, деленную на большее из двух значений:

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\{b^{(i)}, a^{(i)}\}}.$$

Силуэтный коэффициент находится в диапазоне от -1 до 1 . На основании приведенного уравнения мы видим, что силуэтный коэффициент равен 0 , если отделимость и компактность кластера равны ($b^{(i)} = a^{(i)}$). Кроме того, мы приближаемся к идеальному коэффициенту, равному 1 , если $b^{(i)} >> a^{(i)}$, поскольку $b^{(i)}$ количественно определяет, на-

сколько точка отличается от других кластеров, а $a^{(i)}$ говорит нам, насколько она похожа на другие точки в своем собственном кластере.

Силуэтный коэффициент доступен нам как `silhouette_samples` из модуля `metric` `scikit-learn`, и дополнительно для удобства можно импортировать функцию `silhouette_scores`. Функция `silhouette_scores` вычисляет средний силуэтный коэффициент по всем точкам, что эквивалентно операции `numpy.mean(silhouette_samples(...))`. Выполнив следующий код, мы создадим график силуэтных коэффициентов для кластеризации k-средних с количеством кластеров $k = 3$:

```
>>> km = KMeans(n_clusters=3,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)

>>> import numpy as np
>>> from matplotlib import cm
>>> from sklearn.metrics import silhouette_samples
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(
...     X, y_km, metric='euclidean'
... )
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...             c_silhouette_vals,
...             height=1.0,
...             edgecolor='none',
...             color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg,
...               color="red",
...               linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Кластер')
>>> plt.xlabel('Силуэтный коэффициент')
>>> plt.tight_layout()
>>> plt.show()
```

Рассмотрев полученный силуэтный график (рис. 10.4), можно быстро уточнить размеры различных кластеров и определить кластеры, содержащие выбросы.

Как можно здесь видеть, силуэтные коэффициенты не близки к 0 и примерно одинаково далеки от среднего, что в нашем случае является показателем хорошей кластеризации. Кроме того, чтобы обобщить достоинства нашей кластеризации, мы добавили к графику средний силуэтный коэффициент (пунктирная линия).

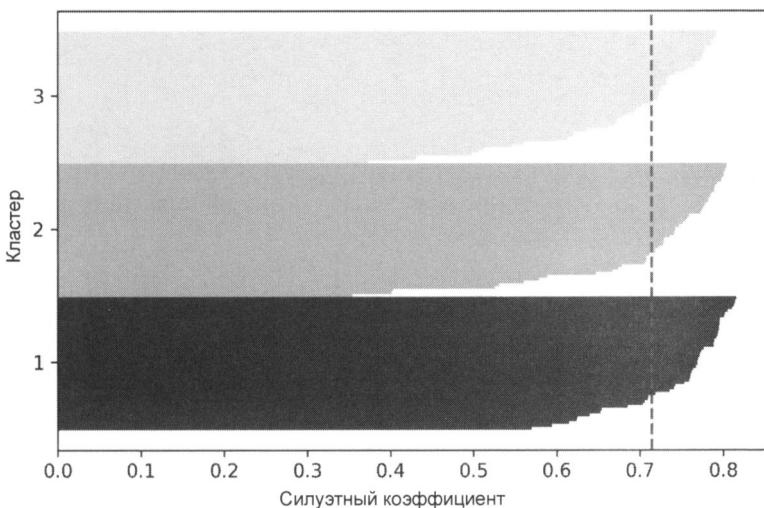


Рис. 10.4. Силуэтный график для примера хорошей кластеризации

Чтобы увидеть, как выглядит силуэтный график для относительно плохой кластеризации, давайте запустим алгоритм k-средних только с двумя центроидами:

```
>>> km = KMeans(n_clusters=2,
...                 init='k-means++',
...                 n_init=10,
...                 max_iter=300,
...                 tol=1e-04,
...                 random_state=0)
>>> y_km = km.fit_predict(X)
>>> plt.scatter(X[y_km == 0, 0],
...               X[y_km == 0, 1],
...               s=50, c='lightgreen',
...               edgecolor='black',
...               marker='s',
...               label='Кластер 1')
>>> plt.scatter(X[y_km == 1, 0],
...               X[y_km == 1, 1],
...               s=50,
...               c='orange',
...               edgecolor='black',
...               marker='o',
...               label='Кластер 2')
```

```
>>> plt.scatter(km.cluster_centers_[:, 0],
...                 km.cluster_centers_[:, 1],
...                 s=250,
...                 marker='*',
...                 c='red',
...                 label='Центроиды')
>>> plt.xlabel('Признак 1')
>>> plt.ylabel('Признак 2')
>>> plt.legend()
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

Как показано на рис. 10.5, один из центроидов оказался между двумя из трех сферических групп входных данных. Хотя нельзя сказать, что кластеризация выглядит ужасно, она неоптимальна.

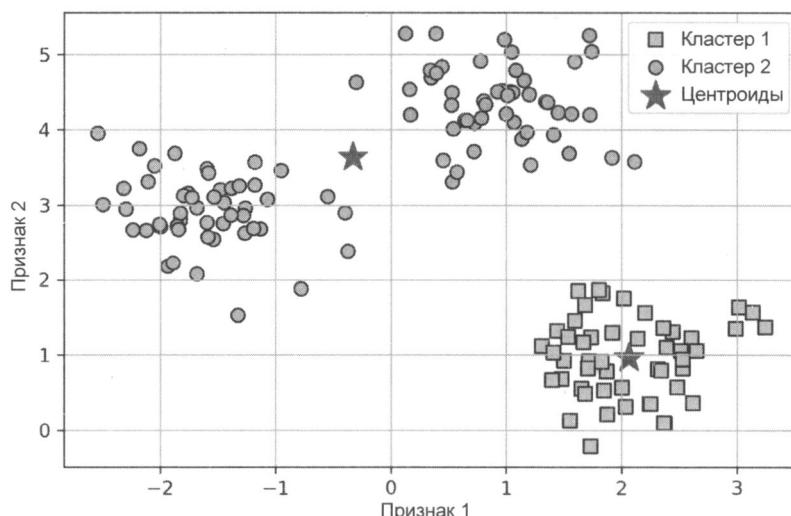


Рис. 10.5. Пример неоптимальной кластеризации

Нужно помнить, что в реальных задачах обычно мы не можем позволить себе роскошь визуализации наборов данных в виде дзумерных диаграмм рассеяния, поскольку нам приходится работать с более многомерными данными. Итак, давайте создадим силуэтный график для оценки полученных результатов:

```
>>> cluster_labels = np.unique(y_km)
>>> n_clusters = cluster_labels.shape[0]
>>> silhouette_vals = silhouette_samples(
...     X, y_km, metric='euclidean'
... )
>>> y_ax_lower, y_ax_upper = 0, 0
>>> yticks = []
>>> for i, c in enumerate(cluster_labels):
...     c_silhouette_vals = silhouette_vals[y_km == c]
```

```
...     c_silhouette_vals.sort()
...     y_ax_upper += len(c_silhouette_vals)
...     color = cm.jet(float(i) / n_clusters)
...     plt.barh(range(y_ax_lower, y_ax_upper),
...             c_silhouette_vals,
...             height=1.0,
...             edgecolor='none',
...             color=color)
...     yticks.append((y_ax_lower + y_ax_upper) / 2.)
...     y_ax_lower += len(c_silhouette_vals)
>>> silhouette_avg = np.mean(silhouette_vals)
>>> plt.axvline(silhouette_avg, color="red", linestyle="--")
>>> plt.yticks(yticks, cluster_labels + 1)
>>> plt.ylabel('Кластер')
>>> plt.xlabel('Силуэтный коэффициент')
>>> plt.tight_layout()
>>> plt.show()
```

Как можно видеть на рис. 10.6, силуэты теперь имеют заметно разную длину и ширину, что свидетельствует об относительно плохой или, по крайней мере, неоптимальной кластеризации.

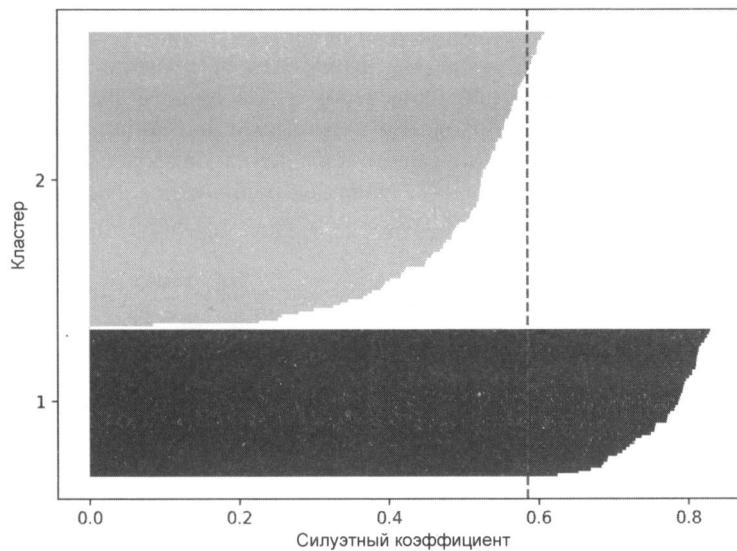


Рис. 10.6. Силуэтный график для примера неоптимальной кластеризации

Итак, вы получили хорошее представление о том, как работает кластеризация, и можно смело приступать к изучению иерархической кластеризации как альтернативы методу k-средних.

10.2. Организация кластеров в виде иерархического дерева

В этом разделе мы рассмотрим альтернативный подход к кластеризации на основе прототипов: *иерархическую кластеризацию* (hierarchical clustering). Одним из преимуществ алгоритма иерархической кластеризации является возможность строить *дендограммы* (визуальные представления бинарной иерархической кластеризации), которые могут помочь в интерпретации результатов путем создания осмысленных таксономий. Еще одно преимущество иерархического подхода заключается в том, что нам не нужно заранее указывать количество кластеров.

Двумя основными подходами к иерархической кластеризации являются *агломеративная* (agglomerative) и *разделительная* (divisive) иерархическая кластеризация. В разделительной иерархической кластеризации мы начинаем с одного кластера, который включает в себя полный набор данных, и итеративно разделяем этот кластер на более мелкие кластеры, пока каждый кластер не будет содержать только одну точку данных. Но здесь мы сосредоточимся на агломеративной кластеризации, которая использует противоположный подход: мы начинаем с каждой точки как отдельного кластера и объединяем ближайшие пары кластеров, пока не останется только один кластер.

10.2.1. Группировка кластеров снизу вверх

Двумя стандартными алгоритмами агломеративной иерархической кластеризации являются подходы *одиночной связи* (single linkage) и *полной связи* (complete linkage). Используя алгоритм одиночной связи, мы вычисляем расстояния между наиболее похожими элементами для каждой пары кластеров и объединяем два кластера, для которых расстояние между наиболее похожими элементами наименьшее. Алгоритм полной связи работает схожим образом, но вместо сравнения наиболее похожих элементов в каждой паре кластеров мы сравниваем наиболее непохожие. Эти два подхода схематически показаны на рис. 10.7.

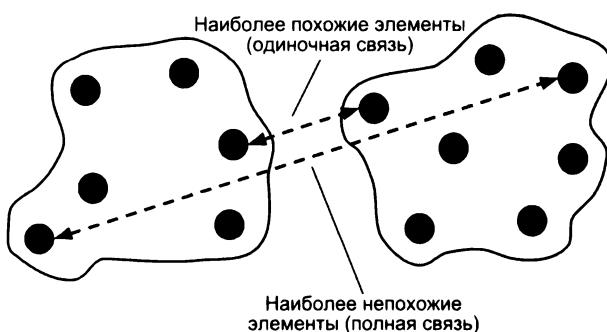


Рис. 10.7. Принцип работы алгоритмов одиночной и полной связи



Альтернативные типы связей

К другим популярным алгоритмам агломеративной иерархической кластеризации относятся *средняя связь* (average linkage) и *связь Уорда* (Ward's linkage). В алгоритме средней связи мы объединяем пары кластеров на основе минимальных средних

расстояний между всеми членами группы в двух кластерах. В алгоритме связи Уорда объединяются два кластера, которые приводят к минимальному увеличению общей внутрикластерной SSE.

В этом разделе мы сосредоточимся на агломеративной кластеризации с использованием алгоритма полной связи. Иерархическая кластеризация с полной связью представляет собой итеративную процедуру, которая состоит из следующих шагов:

1. Вычислить попарную матрицу расстояний всех точек.
2. Представить каждую точку данных в виде одноэлементного кластера.
3. Объединить два ближайших кластера на основе расстояния между самыми неподходящими (отдаленными) элементами.
4. Обновить матрицу связи кластера.
5. Повторять шаги 2–4, пока не останется единственный кластер.

Чуть позднее мы обсудим, как вычислить матрицу расстояний (*шаг 1*). Но сначала сгенерируем случайную выборку данных, с которыми будем работать. Строки представляют разные наблюдения — идентификаторы (ID) 0–4, а столбцы — различные признаки (x, y, z) этих выборок:

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> variables = ['X', 'Y', 'Z']
>>> labels = ['ID_0', 'ID_1', 'ID_2', 'ID_3', 'ID_4']
>>> X = np.random.random([5, 3])*10
>>> df = pd.DataFrame(X, columns=variables, index=labels)
>>> df
```

Выполнив этот код, мы должны увидеть блок данных, содержащий случайно сгенерированные примеры (рис. 10.8).

	X	Y	Z
ID_0	6.964692	2.861393	2.268515
ID_1	5.513148	7.194690	4.231065
ID_2	9.807842	6.848297	4.809319
ID_3	3.921175	3.431780	7.290497
ID_4	4.385722	0.596779	3.980443

Рис. 10.8. Случайно сгенерированная выборка данных

10.2.2. Выполнение иерархической кластеризации с матрицей расстояний

Чтобы вычислить матрицу расстояний, используемую в качестве входных данных для алгоритма иерархической кластеризации, мы применим функцию `pdist` из подмодуля `SciPy spatial.distance`:

```
>>> from scipy.spatial.distance import pdist, squareform
>>> row_dist = pd.DataFrame(squareform(
...                                pdist(df, metric='euclidean')),
...                                columns=labels, index=labels)
>>> row_dist
```

При помощи этого кода мы вычисляем евклидово расстояние между каждой парой входных примеров в нашем наборе данных на основе признаков x, y и z, а затем передаем сжатую матрицу расстояний, возвращенную `pdist`, в качестве входных данных для функции `squareform`, которая создает симметричную матрицу парных расстояний (рис. 10.9).

	ID_0	ID_1	ID_2	ID_3	ID_4
ID_0	0.000000	4.973534	5.516653	5.899885	3.835396
ID_1	4.973534	0.000000	4.347073	5.104311	6.698233
ID_2	5.516653	4.347073	0.000000	7.244262	8.316594
ID_3	5.899885	5.104311	7.244262	0.000000	4.382864
ID_4	3.835396	6.698233	8.316594	4.382864	0.000000

Рис. 10.9. Рассчитанные парные расстояния наших данных

Далее мы применяем к нашим кластерам агломерацию методом полной связи при помощи функции `linkage` из подмодуля `SciPy cluster.hierarchy`, которая возвращает так называемую *матрицу связей* (*linkage matrix*).

Однако прежде, чем вызвать функцию `linkage`, давайте внимательно ознакомимся с ее описанием в документации:

```
>>> from scipy.cluster.hierarchy import linkage
>>> help(linkage)
```

[...]

Параметры:

`y` : ndarray

Сжатая, или избыточная, матрица расстояний. Сжатая матрица расстояний представляет собой плоский массив, содержащий верхний треугольник матрицы расстояний.

Это форма, которую возвращает `pdist`. В качестве альтернативы набор из `m` векторов наблюдения в `n` измерениях может быть передан как массив `m` на `n`.

`method` : str, optional

Применяемый алгоритм связи. Полное описание см. в разделе «Методы связи».

`metric` : str, optional

Применяемая метрика расстояния. Список применимых метрик расстояния см. в описании функции `distance.pdist`.

Возвращает:

`Z` : ndarray

Иерархическая кластеризация, закодированная в виде матрицы связей.

[...]

Из описания функции следует, что в качестве входного атрибута мы можем использовать сжатую матрицу расстояний (верхний треугольник) из функции `pdist`. В качестве альтернативы мы могли бы также предоставить исходный массив данных и использовать метрику 'euclidean' в качестве аргумента функции `linkage`. Однако мы не должны использовать квадратную матрицу расстояний, которую определили ранее, т. к. это приведет к другим значениям расстояний, чем ожидалось. Иными словами, здесь возможны три сценария:

- ◆ **Неправильный подход:** использование квадратной матрицы расстояний, как показано в следующем фрагменте кода, приводит к неправильным результатам:

```
>>> row_clusters = linkage(row_dist,
...                           method='complete',
...                           metric='euclidean')
```

- ◆ **Правильный подход:** использование сжатой матрицы расстояний, как показано в следующем примере кода, дает правильную матрицу связей:

```
>>> row_clusters = linkage(pdist(df, metric='euclidean'),
...                           method='complete')
```

- ◆ **Правильный подход:** использование полной входной матрицы примеров (так называемой *матрицы плана*, *design matrix*), как показано в следующем фрагменте кода, также дает правильную матрицу связей, аналогично предыдущему подходу:

```
>>> row_clusters = linkage(df.values,
...                           method='complete',
...                           metric='euclidean')
```

Чтобы лучше рассмотреть результаты кластеризации, мы можем превратить эти результаты в `DataFrame` `pandas` (лучше всего просматривать в блокноте `Jupyter`) следующим образом:

```
>>> pd.DataFrame(row_clusters,
...                 columns=['row label 1',
...                            'row label 2',
...                            'distance',
...                            'no. of items in clust.'],
...                 index=[f'cluster {(i + 1)}' for i in
...                        range(row_clusters.shape[0])])
```

Как показано на рис. 10.10, матрица связей состоит из нескольких строк, каждая из которых представляет одно слияние. Первый и второй столбцы обозначают наиболее непохожие элементы в каждом кластере, а третий столбец показывает расстояние между этими элементами. Последний столбец содержит количество элементов в каждом кластере.

Теперь, когда у нас есть матрица связей, мы можем представить результаты в виде дендрограммы:

```
>>> from scipy.cluster.hierarchy import dendrogram
>>> # сделать дендрограмму черной (часть 1 из 2)
>>> # from scipy.cluster.hierarchy import set_link_color_palette
```

```

>>> # set_link_color_palette(['black'])
>>> row_dendr = dendrogram(
...     row_clusters,
...     labels=labels,
...     # сделать дендрограмму черной (часть 2 из 2)
...     color_threshold=np.inf
... )
>>> plt.tight_layout()
>>> plt.ylabel('евклидово расстояние')
>>> plt.show()

```

	row label 1	row label 2	distance	no. of items in clust.
cluster 1	0.0	4.0	3.835396	2.0
cluster 2	1.0	2.0	4.347073	2.0
cluster 3	3.0	5.0	5.899885	3.0
cluster 4	6.0	7.0	8.316594	5.0

Рис. 10.10. Матрица связей

Если вы запустили приведенный код или читаете электронную версию этой книги, то видите, что ветви в полученной дендрограмме (рис. 10.11) показаны разными цветами. Цветовая схема взята из списка цветов Matplotlib, которые циклически используются для пороговых значений расстояния в дендрограмме. Чтобы отобразить дендрограмму черным цветом, раскомментируйте соответствующие строки в приведенном коде.

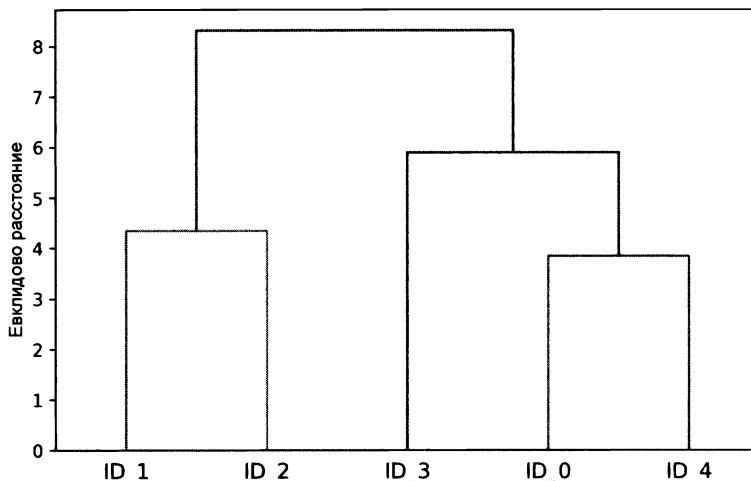


Рис. 10.11. Дендрограмма наших данных

Эта дендрограмма отображает в общем виде различные кластеры, образовавшиеся в ходе агломеративной иерархической кластеризации, — например, вы можете видеть, что примеры ID_0 и ID_4, за которыми следуют ID_1 и ID_2, являются наиболее похожими на основе метрики евклидова расстояния.

10.2.3. Прикрепление дендрограмм к тепловой карте

В практических приложениях иерархические кластерные дендрограммы часто используются в сочетании с тепловой картой, что позволяет нам сопоставлять отдельные значения в массиве данных или в матрице, содержащей наши обучающие примеры, с цветовым кодом. В этом разделе мы обсудим, как прикрепить дендрограмму к тепловой карте и упорядочить ее строки соответствующим образом.

Прикрепление дендрограммы к тепловой карте может вызвать небольшие затруднения, поэтому давайте рассмотрим эту процедуру шаг за шагом:

1. Создаем новый объект `figure` и определяем положение по оси `x`, положение по оси `y`, ширину и высоту дендрограммы с помощью атрибута `add_axes`. Кроме того, поворачиваем дендрограмму на 90 градусов против часовой стрелки. Код выглядит следующим образом:

```
>>> fig = plt.figure(figsize=(8, 8), facecolor='white')
>>> axd = fig.add_axes([0.09, 0.1, 0.2, 0.6])
>>> row_dendr = dendrogram(row_clusters,
...                         orientation='left')
>>> # примечание: для matplotlib < v1.5.1, используйте
>>> # параметр orientation='right'
```

2. Затем переупорядочиваем данные в нашем исходном `DataFrame` в соответствии с метками кластеризации, доступными из объекта `dendrogram`, который по сути является словарем Python, с помощью ключа `leaves`. Эта часть кода выглядит следующим образом:

```
>>> df_rowclust = df.iloc[row_dendr['leaves'][::-1]]
```

3. Далее создаем тепловую карту из переупорядоченного `DataFrame` и размещаем ее рядом с дендрограммой:

```
>>> axm = fig.add_axes([0.23, 0.1, 0.6, 0.6])
>>> cax = axm.matshow(df_rowclust,
...                     interpolation='nearest',
...                     cmap='hot_r')
```

4. Наконец, изменяем внешний вид дендрограммы, удалив оси и скрыв их деления. Кроме того, добавляем цветную линейку и назначаем имена признаков и записей данных меткам осей `x` и `y` соответственно:

```
>>> axd.set_xticks([])
>>> axd.set_yticks([])
>>> for i in axd.spines.values():
...     i.set_visible(False)
>>> fig.colorbar(cax)
>>> axm.set_xticklabels([''] + list(df_rowclust.columns))
>>> axm.set_yticklabels([''] + list(df_rowclust.index))
>>> plt.show()
```

Выполнив перечисленные шаги, мы должны получить тепловую карту с прикрепленной дендрограммой (рис. 10.12).

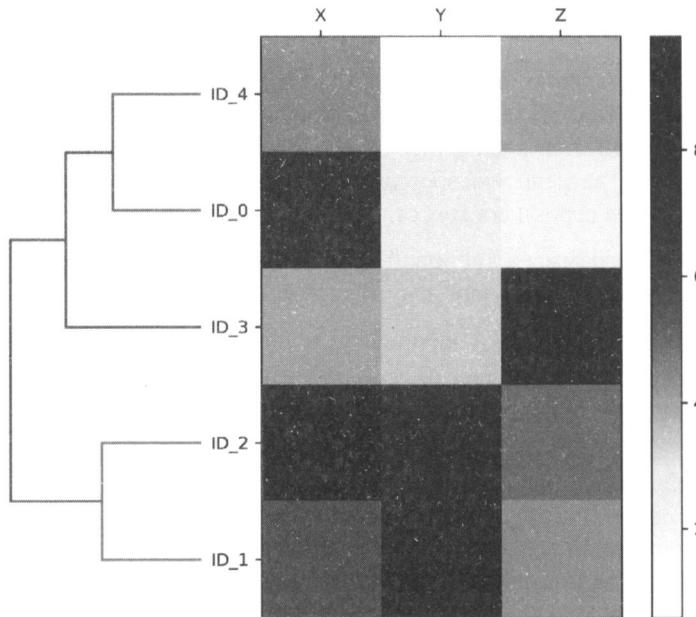


Рис. 10.12. Термометрическая карта и дендрограмма наших данных

Как видите, порядок строк на термометрической карте отражает кластеризацию примеров в дендрограмме. В дополнение к простой дендрограмме, цветные значения каждого примера и функции на термометрической карте дают нам хорошую сводку набора данных.

10.2.4. Агломеративная кластеризация с помощью scikit-learn

В предыдущем разделе вы узнали, как выполнять агломеративную иерархическую кластеризацию с помощью SciPy. Однако в scikit-learn также есть реализация AgglomerativeClustering, которая позволяет нам выбирать количество возвращаемых кластеров. Это полезно, если мы хотим обрезать иерархическое дерево кластера.

Установив для параметра `n_clusters` значение 3, кластеризуем входные примеры в три группы, используя тот же подход полных связей на основе метрики евклидова расстояния, что и ранее:

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> ac = AgglomerativeClustering(n_clusters=3,
...                                 affinity='euclidean',
...                                 linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Метки кластеров: {labels}')
Метки кластеров: [1 0 0 2 1]
```

По спрогнозированным меткам кластеров мы видим, что первый и пятый примеры (`ID_0` и `ID_4`) были назначены одному кластеру (метка 1), а примеры `ID_1` и `ID_2` — другому кластеру (метка 0). Пример `ID_3` помещен в отдельный кластер (метка 2). В целом эти результаты согласуются с результатами, которые мы наблюдали на приведенной ранее

дендrogramme. Следует отметить, что ID_3 больше похож на ID_4 и ID_0, чем на ID_1 и ID_2, как показано на этой дендрограмме, — однако это не ясно из результатов кластеризации scikit-learn. Давайте теперь повторно запустим AgglomerativeClustering, используя параметр n_clusters=2 в следующем блоке кода:

```
>>> ac = AgglomerativeClustering(n_clusters=2,
...                                affinity='euclidean',
...                                linkage='complete')
>>> labels = ac.fit_predict(X)
>>> print(f'Метки кластеров: {labels}')
Метки кластеров: [0 1 1 0 0]
```

Как видите, в этой сокращенной иерархии кластеризации метка ID_3 была назначена кластеру, в котором расположены ID_0 и ID_4. Именно этого мы и ожидали.

10.3. Обнаружение областей высокой плотности с помощью DBSCAN

Хотя мы не можем в одной главе охватить огромное количество различных алгоритмов кластеризации, рассмотрим все же еще один подход к кластеризации — *пространственную кластеризацию зашумленных приложений на основе плотности* (Density-Based Spatial Clustering of Applications with Noise, DBSCAN), который не делает предположений о сферических кластерах, как k-средние, а также не разбивает набор данных на иерархии, которые требуют ручной настройки ограничительного параметра. Как следует из ее названия, кластеризация на основе плотности присваивает метки кластерам, исходя из плотных областей точек. В DBSCAN понятие плотности определяется как количество точек в пределах заданного радиуса ϵ .

В соответствии с алгоритмом DBSCAN, каждому примеру (точке данных) присваивается специальная метка по следующим критериям:

- ◆ точка считается *окруженной* (core point), если хотя бы определенное количество (MinPts) соседних точек попадает в указанный радиус ϵ ;
- ◆ *границная точка* (border point) — это точка, которая имеет меньше соседей, чем MinPts в пределах ϵ , но находится в пределах радиуса ϵ относительно центральной точки;
- ◆ все другие точки, которые не являются ни окруженными, ни границными, считаются *шумовыми точками* (noise point).

После классификации точек как окруженных, границных или шумовых, алгоритм DBSCAN сводится к двум простым шагам:

1. Сформировать отдельный кластер для каждой окруженной точки или связанный группы окруженных точек. (Окруженные точки связаны, если они отстоят друг от друга не дальше, чем на ϵ .)
2. Назначить каждую границную точку кластеру соответствующей окруженной точки.

Чтобы лучше понять, как может выглядеть результат работы алгоритма DBSCAN, обратимся к рис. 10.13, где показан обобщенный пример расположения окруженных, границных и шумовых точек.

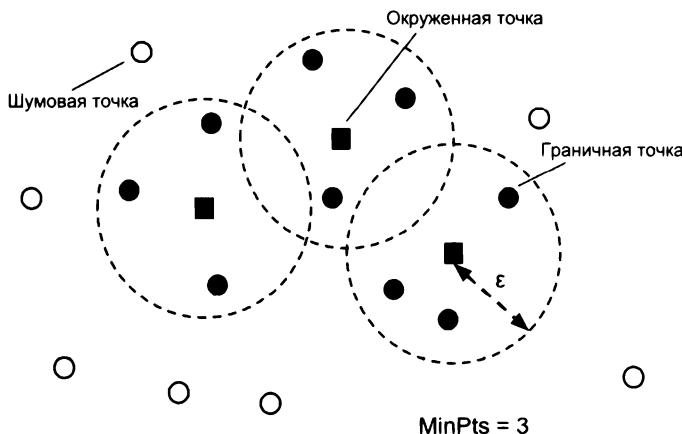


Рис. 10.13. Окруженные, шумовые и граничные точки для DBSCAN

Одним из основных преимуществ использования DBSCAN является то, что он не предполагает сферическую форму кластеров, как метод k-средних. Кроме того, DBSCAN отличается от k-средних и иерархической кластеризации тем, что не обязательно назначает каждую точку кластеру, и благодаря этому может удалять шумовые точки.

В качестве наглядного примера создадим новый набор данных серповидной формы, чтобы сравнить кластеризацию методом k-средних, иерархическую кластеризацию и DBSCAN:

```
>>> from sklearn.datasets import make_moons
>>> X, y = make_moons(n_samples=200,
...                     noise=0.05,
...                     random_state=0)
>>> plt.scatter(X[:, 0], X[:, 1])
>>> plt.xlabel('Признак 1')
>>> plt.ylabel('Признак 2')
>>> plt.tight_layout()
>>> plt.show()
```

На полученном графике (рис. 10.14) отчетливо видны две группы в форме полумесяцев, состоящие из 100 примеров (точек данных) каждая.

Мы начнем с использования алгоритма k-средних и завершим кластеризацией на основе связей, чтобы проверить, может ли один из рассмотренных алгоритмов кластеризации успешно идентифицировать группы серповидной формы как отдельные кластеры. Для этого нужно выполнить следующий код:

```
>>> f, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
>>> km = KMeans(n_clusters=2,
...               random_state=0)
>>> y_km = km.fit_predict(X)
>>> ax1.scatter(X[y_km == 0, 0],
...              X[y_km == 0, 1],
...              c='lightblue',
...              edgecolor='black',
...              s=100)
```

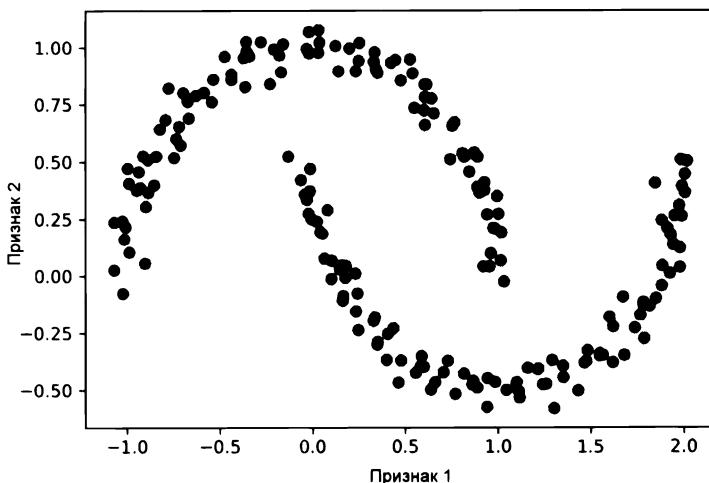


Рис. 10.14. Набор данных с двумя признаками и группировкой точек в форме полумесяцев

```

...
    marker='o',
...
    s=40,
...
    label='cluster 1')
>>> ax1.scatter(X[y_km == 1, 0],
...
    X[y_km == 1, 1],
...
    c='red',
...
    edgecolor='black',
...
    marker='s',
...
    s=40,
...
    label='cluster 2')
>>> ax1.set_title('Кластеризация по k-средним')
>>> ax1.set_xlabel('Признак 1')
>>> ax1.set_ylabel('Признак 2')

>>> ac = AgglomerativeClustering(n_clusters=2,
...
                    affinity='euclidean',
...
                    linkage='complete')
>>> y_ac = ac.fit_predict(X)
>>> ax2.scatter(X[y_ac == 0, 0],
...
    X[y_ac == 0, 1],
...
    c='lightblue',
...
    edgecolor='black',
...
    marker='o',
...
    s=40,
...
    label='Кластер 1')
>>> ax2.scatter(X[y_ac == 1, 0],
...
    X[y_ac == 1, 1],
...
    c='red',
...
    edgecolor='black',
...
    marker='s',
...
    s=40,
...
    label='Кластер 2')

```

```
>>> ax2.set_title('Агломеративная кластеризация')
>>> ax2.set_xlabel('Признак 1')
>>> ax2.set_ylabel('Признак 2')
>>> plt.legend()
>>> plt.tight_layout()
>>> plt.show()
```

Судя по визуальному представлению результатов кластеризации, показанному на рис. 10.15, алгоритм k-средних не смог разделить два кластера, да и алгоритм иерархической кластеризации тоже не совсем справился с этими сложными формами.

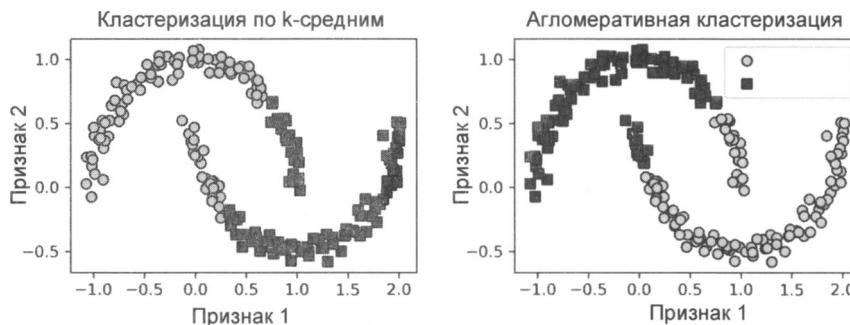


Рис. 10.15. Кластеризация методом k-средних и агломеративная кластеризация на наборе данных в форме полумесяцев

Наконец, попробуем в действии алгоритм DBSCAN и посмотрим, сможет ли он распознать два скопления точек в форме полумесяцев, используя подход, основанный на плотности:

```
>>> from sklearn.cluster import DBSCAN
>>> db = DBSCAN(eps=0.2,
...               min_samples=5,
...               metric='euclidean')
>>> y_db = db.fit_predict(X)
>>> plt.scatter(X[y_db == 0, 0],
...              X[y_db == 0, 1],
...              c='lightblue',
...              edgecolor='black',
...              marker='o',
...              s=40,
...              label='Кластер 1')
>>> plt.scatter(X[y_db == 1, 0],
...              X[y_db == 1, 1],
...              c='red',
...              edgecolor='black',
...              marker='s',
...              s=40,
...              label='Кластер 2')
>>> plt.xlabel('Признак 1')
>>> plt.ylabel('Признак 2')
```

```
>>> plt.legend()  
>>> plt.tight_layout()  
>>> plt.show()
```

Алгоритм DBSCAN успешно распознал группы точек в форме полумесяцев (рис. 10.16), что подчеркивает одну из сильных сторон DBSCAN — кластеризацию данных произвольной формы.

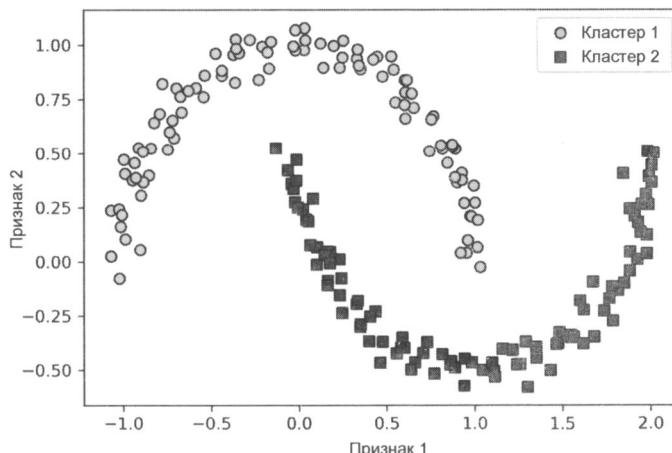


Рис. 10.16. Кластеризация данных в форме полумесяцев с помощью алгоритма DBSCAN

Однако следует отметить и некоторые недостатки DBSCAN. С увеличением количества признаков в нашем наборе данных — при фиксированном количестве обучающих примеров — усиливается негативное влияние проклятия размерности. Это особенно проблематично, если мы используем евклидову метрику расстояния. Впрочем, проблема проклятия размерности присуща не только DBSCAN — она затрагивает и другие алгоритмы кластеризации, использующие евклидову метрику расстояния, — например, алгоритмы k-средних и иерархической кластеризации. Кроме того, в DBSCAN есть два гиперпараметра (`MinPts` и ϵ), которые необходимо оптимизировать, чтобы получить хорошие результаты кластеризации. Поиск хорошей комбинации `MinPts` и ϵ может быть затруднительным, если различия плотности в наборе данных относительно велики.



Графовая кластеризация

До сих пор мы рассматривали три наиболее фундаментальные категории алгоритмов кластеризации: на основе прототипов с k-средними, агломеративно-иерархическую и алгоритм DBSCAN на основе плотности. Однако существует еще и четвертый, более сложный класс, который мы не упоминали в этой главе, — *графовые алгоритмы* кластеризации. Вероятно, наиболее известными членами семейства графовой кластеризации являются алгоритмы спектральной кластеризации.

Хотя существует множество различных реализаций спектральной кластеризации, их все объединяет то, что они используют собственные векторы матрицы сходства или расстояния для получения кластерных отношений. Поскольку изучение спектральной кластеризации выходит за рамки этой книги, рекомендуем прочитать отличный учебник Ульрике фон Люксбург, чтобы узнать больше об этой теме

(«A tutorial on spectral clustering, Statistics and Computing», 17(4): 395–416, 2007). Он находится в свободном доступе на arXiv по адресу: <http://arxiv.org/pdf/0711.0189v1.pdf>.

Отметим, что на практике не всегда очевидно, какой алгоритм кластеризации будет лучше всего работать с конкретным набором данных, особенно если это многомерные данные, что затрудняет или делает невозможной их визуализацию. При этом нужно подчеркнуть, что успех кластеризации зависит не только от алгоритма и его гиперпараметров, — выбор подходящей метрики расстояния и знание предметной области могут оказаться даже важнее.

По этой причине, а также с учетом проклятия размерности обычно принято уменьшать размерность данных до выполнения кластеризации. К методам уменьшения размерности для наборов данных в обучении без учителя относятся анализ основных компонент и t-SNE, которые мы рассмотрели в главе 5. Кроме того, особенно часто наборы данных сжимают до двумерных подпространств, что позволяет визуализировать кластеры и присвоенные метки с помощью двумерных диаграмм рассеяния, которые особенно полезны для оценки результатов.

10.4. Заключение

В этой главе вы узнали о трех различных алгоритмах кластеризации, которые помогают нам обнаружить в данных скрытые структуры или информацию. Мы начали с метода k-средних, который группирует точки данных в сферические формы на основе определенного количества центроидов кластера. Поскольку кластеризация — это метод обучения без учителя, мы не можем воспользоваться эталонными метками для оценки производительности модели. Поэтому мы использовали для оценки качества кластеризации внутренние показатели производительности — такие как метод локтя или силузтный анализ.

Затем мы рассмотрели другой подход к кластеризации — агломеративную иерархическую кластеризацию. Иерархическая кластеризация не требует указывать количество кластеров заранее, а результат можно отобразить в виде дендрограммы, что хорошо помогает интерпретировать результаты. Последний алгоритм кластеризации, который мы изучили в этой главе, — DBSCAN. Он группирует точки на основе локальной плотности и способен устранять шумовые выбросы и идентифицировать произвольные формы.

После этого экскурса в область обучения без учителя пришло время представить некоторые из самых захватывающих алгоритмов машинного обучения для обучения с учителем — многослойные искусственные нейронные сети. Будучи относительно недавно возрождены, нейронные сети снова стали самой горячей темой в исследованиях машинного обучения. Благодаря современным алгоритмам глубокого обучения нейронные сети блестяще справляются с такими сложными задачами, как классификация изображений, обработка естественного языка и распознавание речи. В главе 11 мы построим собственную многослойную нейронную сеть, а в главе 12 начнем работать с библиотекой PyTorch, которая специализируется на очень эффективном обучении многослойных нейросетевых моделей с использованием графических процессоров.

11

Построение многослойной искусственной нейронной сети с нуля

Как вы наверняка знаете, глубокое обучение привлекает большое внимание прессы и, без сомнения, является самой горячей темой в области машинного обучения. *Глубокое обучение* — это область машинного обучения, которая занимается обучением многослойных искусственных нейронных сетей (Neural Network, NN). В этой главе вы познакомитесь с основными понятиями искусственных нейронных сетей, чтобы во всеоружии перейти к следующим главам, в которых будут представлены расширенные библиотеки глубокого обучения на основе Python и архитектуры глубоких нейронных сетей (Deep Neural Network, DNN), особенно хорошо подходящие для работы с текстом и изображениями.

Мы рассмотрим в этой главе следующие темы:

- ◆ принципы устройства и работы многослойных нейронных сетей;
- ◆ реализация фундаментального алгоритма обратного распространения ошибки для обучения нейросети с нуля;
- ◆ обучение базовой многослойной нейросети для классификации изображений.

11.1. Моделирование сложных функций с помощью искусственных нейросетей

В главе 2 мы начали знакомство с алгоритмами машинного обучения на основе искусственных нейронов. Искусственные нейроны и представляют собой строительные блоки многослойных искусственных нейронных сетей, которые мы обсудим в этой главе.

Основные идеи устройства искусственных нейронных сетей были позаимствованы из гипотез о том, как человеческий мозг решает сложные задачи. Хотя искусственные нейронные сети приобрели особенно большую популярность в последнее десятилетие, первые их исследования ведут свою историю еще с 1940-х годов, когда Уоррен МакКаллох и Уолтер Питтс впервые описали, как могут работать нейроны¹.

Тем не менее в течение десятилетий, последовавших за первой реализацией модели нейронов МакКаллоха — Питтса — персептрона Розенблатта в 1950-х годах, — многие

¹ См. «A logical calculus of the ideas immanent in nervous activity», by W. S. McCulloch and W. Pitts, The Bulletin of Mathematical Biophysics, 5(4):115–133, 1943.

исследователи и специалисты по машинному обучению постепенно теряли интерес к нейронным сетям, поскольку им не удавалось найти хороший способ обучения нейронной сети с несколькими слоями. Интерес к нейронным сетям возродился в 1986 г., когда Д. Румельхарт, Г. Хинтон и Р. Уильямс фактически заново открыли алгоритм обратного распространения ошибки, пригодный для эффективного обучения нейронных сетей². Читателям, интересующимся историей искусственного интеллекта, машинного обучения и нейронных сетей, мы также рекомендуем прочитать статью в Википедии о так называемых «зимах ИИ» — периодах времени, когда большая часть исследовательского сообщества теряла интерес к изучению нейронных сетей³.

Однако сегодня нейронные сети невероятно популярны благодаря множеству научных и технических прорывов, сделанных в предыдущее десятилетие. Они привели к созданию алгоритмов и архитектур глубокого обучения — многослойных нейронных сетей. Нейронные сети привлекают не только академических исследователей, но и крупные технологические корпорации, такие как Microsoft, Amazon, Uber, Google, и многие другие, вкладывающие значительные средства в искусственные нейронные сети и исследования глубокого обучения.

На сегодняшний день комплексные нейронные сети, основанные на алгоритмах глубокого обучения, считаются наиболее передовыми инструментами для решения сложных задач — таких как распознавание изображений и голоса. Вот лишь несколько наиболее показательных примеров их применения:

- ◆ прогнозирование потребностей в ресурсах для борьбы с COVID-19 на основе серии рентгеновских снимков (<https://arxiv.org/abs/2101.04909>);
- ◆ моделирование мутаций вируса (<https://science.sciencemag.org/content/371/6526/284>);
- ◆ использование данных из социальных сетей для отслеживания экстремальных погодных явлений (<https://onlinelibrary.wiley.com/doi/abs/10.1111/1468-5973.12311>);
- ◆ улучшенное текстовое описание фотографий для слепых или слабовидящих людей (<https://tech.fb.com/how-facebook-is-using-ai-to-improve-photo-descriptions-for-people-who-are-blind-or-visually-impaired/>).

11.1.1. Вкратце об однослойной нейронной сети

В этой главе мы ведем речь о многослойных нейросетях — о том, как они работают, и как их можно научить решать сложные задачи. Но прежде чем мы углубимся в изучение архитектуры многослойной нейросети, давайте вспомним важный аспект работы однослойных сетей, о которых шла речь в главе 2, — а именно алгоритм Adaline (adaptive linear neuron), схематически показанный на рис. 11.1.

В главе 2 мы разработали реализацию алгоритма Adaline для выполнения бинарной классификации и использовали алгоритм оптимизации градиентного спуска для обучения весовых коэффициентов модели. В каждую эпоху (проход через обучающий набор

² См. «Learning representations by backpropagating errors», by D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Nature*, 323 (6088): 533–536, 1986.

³ См. https://ru.wikipedia.org/wiki/Зима_искусственного_интеллекта.

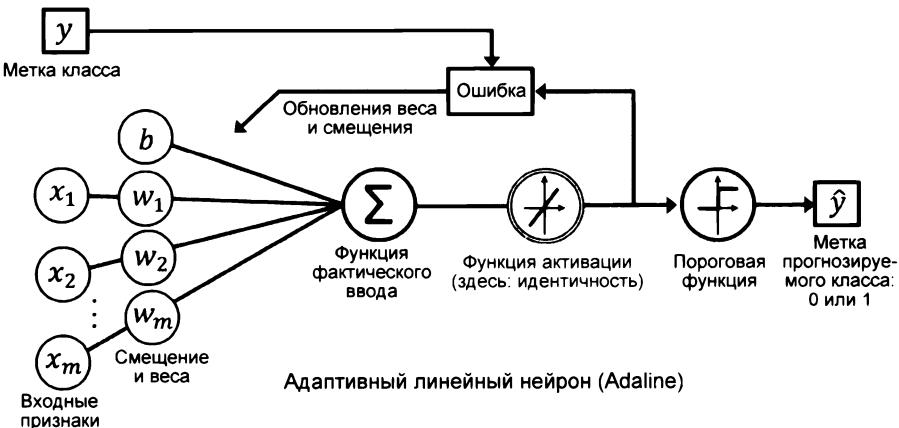


Рис. 11.1. Алгоритм Adaline

данных) мы обновляли вектор весов w и смещение b , используя следующее правило обновления:

$$w := w + \Delta w, b := b + \Delta b,$$

где $\Delta w_j = -\eta \frac{\partial L}{\partial w_j}$ и $\Delta b = -\eta \frac{\partial L}{\partial b}$ представляют собой каждый вес w_i в векторе w и смещение соответственно.

Другими словами, мы вычислили градиент на основе всего набора обучающих данных и обновили веса модели, сделав шаг в направлении, противоположном градиенту потерь $\nabla L(w)$ (Для простоты мы сосредоточимся на весах и опустим смещение в следующих рассуждениях, однако, как вы помните из главы 2, к смещению применяются те же подходы.) Чтобы найти оптимальные веса модели, мы оптимизировали целевую функцию, которую определили по метрике среднего квадрата ошибок (Mean of Squared Errors, MSE) как функцию потерь $L(w)$. Кроме того, мы умножили градиент на коэффициент скорости обучения η , который пришлось тщательно выбирать, чтобы найти компромисс между скоростью обучения и риском проскочить глобальный минимум функции потерь.

При оптимизации методом градиентного спуска мы обновляли все веса одновременно после каждой эпохи и определяли частную производную для каждого веса w_j в векторе весов w следующим образом:

$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i (y^{(i)} - a^{(i)})^2 = -\frac{2}{n} \sum_i (y^{(i)} - a^{(i)}) x_j^{(i)}.$$

Здесь $y^{(i)}$ — метка целевого класса конкретной выборки $x^{(i)}$, а $a^{(i)}$ — активация нейрона, которая в частном случае Adaline является линейной функцией.

Кроме того, мы определили функцию активации $\sigma(\cdot)$ следующим образом:

$$\sigma(\cdot) = z = a.$$

Здесь действующий вход z представляет собой линейную комбинацию весов, соединяющих входной слой с выходным слоем:

$$z = \sum_j w_j x_j + b = w^T x + b.$$

Одновременно с использованием активации $\sigma(\cdot)$ для вычисления обновления градиента мы применяли пороговую функцию для сжатия выходных данных с непрерывным значением в прогнозируемые метки двоичных классов:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0; \\ 0 & \text{в ином случае} \end{cases}$$



Соглашение о наименовании

Обратите внимание, что хотя Adaline состоит из двух слоев: одного входного слоя и одного выходного слоя — она называется *однослоиной сетью* из-за единственной связи между входным и выходным слоями.

В главе 2 вы также узнали об одном трюке для ускорения обучения модели — так называемой оптимизации методом *стохастического градиентного спуска* (Stochastic Gradient Descent, SGD). Алгоритм SGD аппроксимирует потери одной обучающей выборки (онлайн-обучение) или небольшого подмножества обучающих примеров (минипакетное обучение). Мы вернемся к этой концепции позже, когда будем программировать и обучать многослойный персептрон (Multilayer Perceptron, MLP). Помимо ускорения обучения — за счет более частых обновлений веса по сравнению с градиентным спуском, — зашумленная природа SGD также считается полезной при обучении многослойных нейронных сетей с нелинейными функциями активации, которые не имеют выпуклой функции потерь. Здесь внесенный шум помогает избежать локальных минимумов потерь, но мы подробнее обсудим эту тему чуть позже.

11.1.2. Знакомство с архитектурой многослойной нейронной сети

В этом разделе вы узнаете, как соединить несколько одиночных нейронов в многослойную нейронную сеть с прямым распространением сигнала. Такой особый тип *полносвязной сети* называется *многослойным персептроном* (Multilayer Perceptron, MLP). На рис. 11.2 показано устройство MLP, состоящего из двух слоев.

Помимо слоя ввода данных, MLP, изображенный на рис. 11.2, имеет один скрытый слой и один выходной слой. Элементы (узлы, нейроны) скрытого слоя полностью связаны с узлами входного слоя, а выходной слой полностью связан со скрытым слоем. Если сеть имеет более одного скрытого слоя, ее называют *глубокой нейронной сетью*. (Заметим, что в некоторых контекстах слой ввод данных также рассматривается как слой. Однако в этом случае модель Adaline, которая представляет собой однослоиную нейронную сеть, формально будет считаться двухслойной нейронной сетью, что несколько противоречит здравому смыслу.)



Добавление дополнительных скрытых слоев

Для создания более глубокой сетевой архитектуры мы можем добавить в MLP любое количество скрытых слоев. Фактически количество слоев и элементов слоя можно рассматривать как дополнительные гиперпараметры, нуждающиеся в оптимизации для конкретной задачи с использованием метода перекрестной проверки, который мы обсуждали в главе 6.

Однако по мере добавления новых слоев в сеть градиенты потерь, которые мы рассчитаем позже с помощью алгоритма обратного распространения, будут становиться

ся все меньше. Эта проблема исчезающего градиента (*vanishing gradient*) усложняет обучение модели. Для борьбы с явлением исчезающего градиента были разработаны специальные алгоритмы глубокого обучения, которые мы обсудим более подробно в следующих главах.

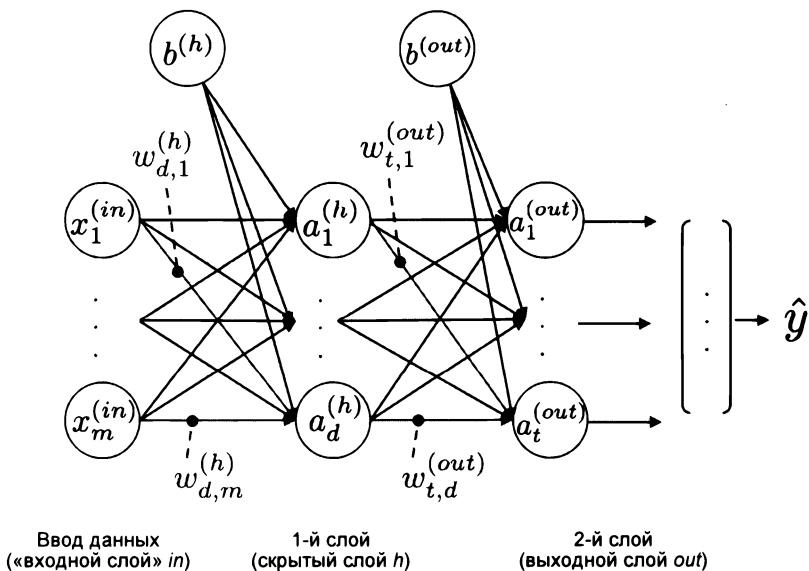


Рис. 11.2. Двухслойный MLP

Как показано на рис. 11.2, мы обозначаем i -й элемент активации на l -м уровне как $a_i^{(l)}$. Чтобы сделать математическую запись проще и понятнее, мы не станем использовать числовые индексы для ссылки на слои, но задействуем верхний индекс *in* — для входных признаков, верхний индекс *h* — для скрытого слоя и верхний индекс *out* — для выходного слоя. Например, $x_i^{(in)}$ будет обозначать i -е значение входного признака, $a_i^{(h)}$ — i -й узел в скрытом слое, а $a_i^{(out)}$ — i -й узел в выходном слое. Кроме того, буква b на рис. 11.2 обозначает смещение. На самом деле $b^{(h)}$ и $b^{(out)}$ — это векторы, количество элементов которых равно количеству узлов в слое, которому они соответствуют. Например, $b^{(h)}$ хранит d элементов смещения, где d — количество узлов в скрытом слое. Если это пока звучит непонятно, не волнуйтесь. Изучение кода, в котором мы инициализируем матрицы весов и единичные векторы смещения, поможет внести ясность в эти понятия.

Каждый узел в слое l связан со всеми узлами в слое $l + 1$ через весовой коэффициент. Например, связь между k -м узлом в слое l и j -м узлом в слое $l + 1$ будет записана так: $w_{j,k}^{(l+1)}$. Возвращаясь к рис. 11.2, мы обозначим матрицу весов, соединяющую входной слой со скрытым, как $W^{(h)}$, а матрицу, соединяющую скрытый слой с выходным, — как $W^{(out)}$.

Хотя для задачи бинарной классификации достаточно единственного узла в выходном слое, на рис. 11.2 показана более общая форма нейронной сети, которая позволяет нам выполнять многоклассовую классификацию посредством обобщения метода один про-

тие всех (One-versus-All, OvA). Чтобы лучше понять, как он работает, вспомните унитарное представление категориальных переменных, которое мы рассмотрели в главе 4.

Например, мы можем закодировать три метки класса в знакомом наборе данных Iris ($0 = \text{Setosa}$, $1 = \text{Versicolor}$, $2 = \text{Virginica}$) следующим образом:

$$0 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, 1 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, 2 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

Это унитарное векторное представление позволяет нам решать задачи классификации с произвольным количеством уникальных меток классов, присутствующих в наборе обучающих данных.

Если вы впервые столкнулись с обозначениями компонентов нейронной сети, то применение верхних и нижних индексов может сначала показаться немного запутанным. Но не беспокойтесь, обозначения станут более понятными в следующих разделах, когда мы векторизуем представление нейросети. Как уже было сказано ранее, мы суммируем веса, которые соединяют входной и скрытый слои, с помощью матрицы $W^{(h)}$ размерностью $d \times m$, где d — количество скрытых узлов, а m — количество входных узлов.

11.1.3. Активация нейронной сети прямого распространения

В этом разделе мы опишем процесс *прямого распространения* (forward propagation) для расчета выходных данных модели MLP. Чтобы показать, как этот процесс связан с обучением модели MLP, представим процедуру обучения в виде трех простых шагов:

1. Начиная со входного слоя, передаем паттерны обучающих данных по сети для создания выходных данных.
2. На основе выходных данных сети вычисляем потерю, которую мы стремимся минимизировать. Для этого нам понадобится функция потерь (мы опишем ее позже).
3. Распространяем потерю обратно, находим ее производную по каждому элементу веса и смещения в сети и обновляем модель.

Мы повторяем эти три шага в течение нескольких эпох и находим параметры веса и смещения MLP, а затем используем прямое распространение для расчета выходных данных сети и применяем к ним пороговую функцию для получения прогнозируемых меток классов в унитарном представлении, которое мы описали в предыдущем разделе.

Теперь давайте пройдемся по отдельным шагам прямого распространения, чтобы понять, как из паттернов в обучающих данных получаются выходные данные. Поскольку каждый узел в скрытом слое связан со всеми узлами во входных слоях, мы сначала вычисляем активацию скрытого слоя $a_1^{(h)}$:

$$\begin{aligned} z_1^{(h)} &= x_1^{(in)} w_{1,1}^{(h)} + x_2^{(in)} w_{1,2}^{(h)} + \dots + x_m^{(in)} w_{1,m}^{(h)} \\ a_1^{(h)} &= \sigma(z_1^{(h)}) \end{aligned}$$

Здесь $a_1^{(h)}$ — действующий вход, а $\sigma(\cdot)$ — функция активации, которая должна быть дифференцируемой, чтобы обучить веса нейронных связей на основе метода градиента.

Чтобы иметь возможность решать сложные задачи, такие как классификация изображений, нашей модели MLP нужны нелинейные функции активации — например, сигмоидная (логистическая) функция активации, с которой вы познакомились в посвященном логистической регрессии разделе главы 3:

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Как вы помните, сигмоидная функция представляет собой S-образную кривую, которая отображает действующий вход z на логистическое распределение в диапазоне от 0 до 1, пересекающее ось y в точке $z = 0$ (рис. 11.3).

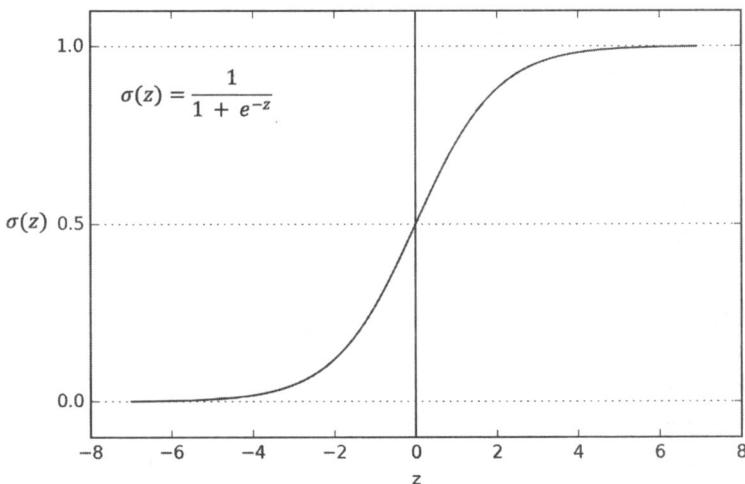


Рис. 11.3. Сигмоидная функция активации

MLP является типичным примером *искусственной нейросети прямого распространения* (feedforward artificial neural network). Термин «прямое распространение» относится к тому факту, что выход каждого слоя служит входом для следующего слоя, и эта структура не содержит циклов, в отличие от рекуррентных сетей — архитектуры, которой мы кратко коснемся в этой главе и обсудим более подробно в главе 15. Термин «многослойный персептрон» может показаться немного неуместным, поскольку искусственные нейроны в этой сетевой архитектуре обычно представляют собой сигмоидные блоки, а не персептроны. Но мы можем рассматривать нейроны в MLP как блоки логистической регрессии, которые возвращают значения в непрерывном диапазоне от 0 до 1.

Для повышения эффективности и удобочитаемости кода запишем активацию в более компактной форме, используя понятия линейной алгебры, что позволит нам создать векторизованную версию кода NumPy, а не писать несколько вложенных и дорогостоящих в вычислительном отношении циклов for Python:

$$z^{(h)} = x^{(in)} W^{(h)T} + b^{(h)}$$

$$a^{(h)} = \sigma(z^{(h)})$$

Здесь $x^{(in)}$ — наш вектор признаков размером $1 \times m$, а $W^{(h)}$ — матрица весов размера $d \times m$, где d — количество узлов в скрытом слое. Следовательно, транспонированная матрица $W^{(h)T}$ имеет размерность $m \times d$. Вектор смещения $b^{(h)}$ состоит из d элементов (по одному элементу на каждый скрытый узел).

После умножения матрицы на вектор мы получаем входной вектор $z^{(h)}$ размера $1 \times d$ для вычисления активации $a^{(h)}$ (где $a^{(h)} \in \mathbb{R}^{1 \times d}$).

Кроме того, мы можем обобщить это вычисление на все n примеров в обучающем наборе данных:

$$Z^{(h)} = X^{(in)} W^{(h)T} + b^{(h)}.$$

Здесь $X^{(in)}$ теперь представляет собой матрицу размера $n \times m$, а перемножение матриц даст нам входную матрицу $Z^{(h)}$ размера $n \times d$. Наконец, мы применяем функцию активации $\sigma(\cdot)$ к каждому значению во входной матрице, чтобы получить матрицу активации $n \times d$ в следующем слое (здесь это выходной слой):

$$A^{(h)} = \sigma(Z^{(h)}).$$

Аналогично можно записать активацию выходного слоя в векторизованной форме для нескольких примеров:

$$Z^{(out)} = A^{(h)} W^{(in)T} + b^{(out)}.$$

Здесь мы умножаем транспонированную матрицу $W^{(out)}$ $t \times d$ (t — количество выходных элементов) на матрицу $A^{(h)}$ размера $n \times d$ и добавляем вектор смещения $b^{(out)}$ размера t к получившейся матрице $Z^{(out)}$ размера $n \times t$. (Строки в этой матрице представляют собой выходные данные для каждого примера.)

Наконец, мы применяем сигмоидную функцию активации, чтобы получить выход нашей сети с непрерывным значением:

$$A^{(out)} = \sigma(Z^{(out)}).$$

Подобно $Z^{(out)}$, $A^{(out)}$ представляет собой матрицу размера $n \times t$.

11.2. Классификация рукописных цифр

В предыдущем разделе мы рассмотрели теоретические основы многослойных нейронных сетей, которые могут показаться немного сложными, если вы новичок в этой теме. Прежде чем продолжить обсуждение алгоритма обратного распространения ошибки, который применяется для обучения весов модели MLP, давайте немного отвлечемся от теории и посмотрим на нейросеть в действии.



Дополнительные ресурсы по алгоритму обратного распространения ошибки

Теоретические основы нейронных сетей могут быть сложны для неподготовленного читателя, поэтому мы хотим порекомендовать вам дополнительные ресурсы, в которых более подробно или с другой точки зрения раскрыты некоторые из тем, которые мы обсуждаем в этой главе:

- глава 6 книги «Deep Feedforward Networks, Deep Learning», by I. Goodfellow, Y. Bengio, and A. Courville, MIT Press, 2016 (свободно доступна на <http://www.deeplearningbook.org>)⁴;
- «Pattern Recognition and Machine Learning», by C. M. Bishop, Springer New York, 2006;
- слайды видеолекций из курса глубокого обучения Себастьяна Рашки:
<https://sebastianraschka.com/blog/2021/dl-course.html#l08-multinomial-logistic-regression--softmax-regression>;
<https://sebastianraschka.com/blog/2021/dl-course.html#l09-multilayer-perceptrons-and-backpropagation>.

В этом разделе мы реализуем и обучим нашу первую многослойную нейронную сеть для классификации рукописных цифр из популярного набора данных Mixed National Institute of Standards and Technology (MNIST), который был создан Яном Лекуном и его коллегами и служит популярным эталонным набором данных для алгоритмов машинного обучения⁵.

11.2.1. Получение и подготовка набора данных MNIST

Набор данных MNIST свободно доступен по адресу <http://yann.lecun.com/exdb/mnist/> и состоит из следующих четырех частей:

1. Образы набора обучающих данных — `train-images-idx3-ubyte.gz` (9.9 Мбайт, 47 Мбайт в разархивированном виде и 60 тыс. примеров).
2. Метки набора обучающих данных — `train-labels-idx1-ubyte.gz` (29 Кбайт, 60 Кбайт в разархивированном виде и 60 тыс. меток).
3. Образы набора тестовых данных — `t10k-images-idx3-ubyte.gz` (1.6 Мбайт, 7.8 Мбайт в разархивированном виде и 10 тыс. примеров).
4. Метки набора тестовых данных — `t10k-labels-idx1-ubyte.gz` (5 Кбайт, 10 Кбайт в разархивированном виде и 10 тыс. меток).

Набор данных MNIST был создан на основе двух наборов данных Национального института стандартов и технологий США (NIST). Обучающий набор состоит из цифр, написанных руками 250 разных людей, среди которых 50% — учащиеся старших классов и 50% — сотрудники Бюро переписи населения. Важно отметить, что тестовый набор данных содержит рукописные цифры от разных людей в том же соотношении.

Вместо того, чтобы загружать упомянутые файлы набора данных и предварительно конвертировать их в массивы NumPy, мы применим новую функцию `fetch_openml` библиотеки scikit-learn, которая позволяет нам более удобно загружать набор данных MNIST:

⁴ См. перевод этой книги: «Глубокое обучение», Курвиль Аарон, Гудфеллоу Ян, Бенджио Иошуа, М.: ДМК Пресс, 2018.

⁵ См. «Gradient-Based Learning Applied to Document Recognition», by Y. LeCun, L. Bottou, Y. Bengio and P. Haffner, Proceedings of the IEEE, 86(11): 2278–2324, 1998.

```
>>> from sklearn.datasets import fetch_openml
>>> X, y = fetch_openml('mnist_784', version=1,
...                      return_X_y=True)
>>> X = X.values
>>> y = y.astype(int).values
```

Функция `fetch_openml` загружает набор данных MNIST из OpenML (<https://www.openml.org/d/554>) в виде объектов `DataFrame` и `Series` `pandas`, поэтому мы задействуем атрибут `.values` для получения массивов NumPy. (Если вы работаете с версией `scikit-learn` старше 1.0, `fetch_openml` загружает массивы NumPy напрямую, поэтому вы можете не использовать атрибут `.values`.) Массив размера $n \times m$ хранит соответствующие 70 тыс. меток классов. Мы можем убедиться в этом, проверив размеры массивов:

```
>>> print(X.shape)
(70000, 784)
>>> print(y.shape)
(70000,)
```

Изображения в наборе данных MNIST имеют размер 28×28 пикселов, и каждый пиксель представлен значением интенсивности в оттенках серого. В нашем случае функция `fetch_openml` уже развернула пиксельную матрицу 28×28 в одномерные векторы-строки, которые представляют строки в нашем массиве `x` (784 на строку или изображение) в приведенном коде. Второй массив — `y`, возвращаемый функцией `fetch_openml`, содержит соответствующую целевую переменную — метки классов рукописных цифр (целые числа 0–9).

Далее мы приводим значения пикселов в MNIST к диапазону от -1 до 1 (первоначально было от 0 до 255) с помощью следующей строки кода:

```
>>> X = ((X / 255.) - .5) * 2
```

Это преобразование необходимо, потому что оптимизация на основе градиента в этом числовом диапазоне гораздо более стабильна (как обсуждалось в главе 2). Обратите внимание, что мы масштабировали изображения попиксельно, что отличается от подхода масштабирования признаков, к которому мы прибегали в предыдущих главах. Тогда мы вычисляли параметры масштабирования исходя из набора обучающих данных и использовали их для масштабирования каждого столбца в наборе обучающих данных и наборе тестовых данных. Однако при работе с пикселями изображения также широко распространено центрирование их значений по нулю и масштабирование до диапазона [-1, 1], и этот прием обычно хорошо работает на практике.

Чтобы получить представление о том, как выглядят изображения в MNIST, визуализируем примеры цифр от 0 до 9 после изменения формы 784-пиксельных векторов из нашей матрицы признаков в исходное изображение 28×28 , которое мы можем построить с помощью функции `imshow` `matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> fig, ax = plt.subplots(nrows=2, ncols=5,
...                        sharex=True, sharey=True)
>>> ax = ax.flatten()
>>> for i in range(10):
...     img = X[y == i][0].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
```

```
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

Выполнив этот код, вы должны увидеть изображения, выстроенные в сетку 2×5 и представляющие собой пример написания каждой уникальной цифры (рис. 11.4).

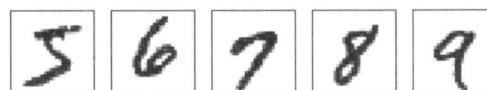
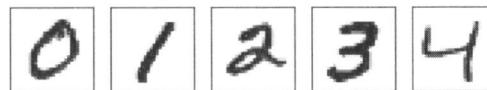


Рис. 11.4. Примеры написания одной случайно выбранной рукописной цифры из каждого класса

Кроме того, будет полезно отобразить несколько примеров написания одной и той же цифры, чтобы увидеть, насколько на самом деле различаются почерки разных людей:

```
>>> fig, ax = plt.subplots(nrows=5,
...                         ncols=5,
...                         sharex=True,
...                         sharey=True)
>>> ax = ax.flatten()
>>> for i in range(25):
...     img = X[y == 7][i].reshape(28, 28)
...     ax[i].imshow(img, cmap='Greys')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

После выполнения этого кода вы должны увидеть первые 25 вариантов цифры 7 (рис. 11.5).



Рис. 11.5. Различные варианты написания рукописной цифры 7

Наконец, разделим набор данных на обучающий, валидационный и тестовый сегменты. Следующий код разделит набор данных так, чтобы 55 тыс. изображений использовались для обучения, 5 тыс. изображений — для валидации и 10 тыс. — для тестирования:

```
>>> from sklearn.model_selection import train_test_split
>>> X_temp, X_test, y_temp, y_test = train_test_split(
...     X, y, test_size=10000, random_state=123, stratify=y
... )
>>> X_train, X_valid, y_train, y_valid = train_test_split(
...     X_temp, y_temp, test_size=5000,
...     random_state=123, stratify=y_temp
... )
```

11.2.2. Реализация многослойного персептрона

Сейчас мы разработаем с нуля многослойный персептрон (MLP), предназначенный для классификаций изображений в наборе данных MNIST. Для простоты мы реализуем MLP только с одним скрытым слоем. Поскольку поначалу этот подход может показаться немного сложным, рекомендуем загрузить пример кода для этой главы с GitHub (<https://github.com/rasbt/machine-learning-book>), чтобы иметь возможность изучать реализацию MLP с комментариями и подсветкой синтаксиса для лучшей читаемости кода.

Если вы не запускаете код из содержащегося в сопровождающем книгу файловом архиве файла Jupyter Notebook (ch11.ipynb) или у вас нет доступа к Интернету, скопируйте код NeuralNetMLP из этой главы в файл сценария Python в вашем текущем рабочем каталоге (например, в файл `neuronet.py`), который затем можно импортировать в текущий сеанс Python с помощью следующей команды:

```
from neuralnet import NeuralNetMLP
```

Код содержит компоненты, о которых мы еще не говорили, — например, алгоритм обратного распространения ошибки. Не беспокойтесь, если в коде есть непонятные вам места, — мы подробно разберемся с ними в этой главе позже. Однако даже простой просмотр кода на этом этапе облегчит понимание теории.

Итак, рассмотрим реализацию MLP, начиная с двух вспомогательных функций для вычисления логистической сигмоидной активации и преобразования массивов меток целочисленного класса в метки с унитарным кодированием:

```
import numpy as np

def sigmoid(z):
    return 1. / (1. + np.exp(-z))

def int_to_onehot(y, num_labels):

    ary = np.zeros((y.shape[0], num_labels))
    for i, val in enumerate(y):
        ary[i, val] = 1

    return ary
```

Далее приведена реализация главного класса для нашего MLP под названием NeuralNetMLP. В ней доступны три метода класса: `__init__()`, `forward()` и `backward()`, которые мы обсудим по порядку, начиная с конструктора `__init__()`:

```
class NeuralNetMLP:

    def __init__(self, num_features, num_hidden,
                 num_classes, random_seed=123):
        super().__init__()

        self.num_classes = num_classes

        # скрытый слой
        rng = np.random.RandomState(random_seed)

        self.weight_h = rng.normal(
            loc=0.0, scale=0.1, size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # выход
        self.weight_out = rng.normal(
            loc=0.0, scale=0.1, size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)
```

Конструктор `__init__` создает экземпляры матриц весов и векторов смещения для скрытого и выходного слоев. Давайте посмотрим, как они используются для прогнозирования в методе `forward`:

```
def forward(self, x):
    # Скрытый слой

    # размерность входа: [n_examples, n_features]
    # dot [n_hidden, n_features].T
    # размерность выхода: [n_examples, n_hidden]
    z_h = np.dot(x, self.weight_h.T) + self.bias_h
    a_h = sigmoid(z_h)

    # Выходной слой
    # размерность входа: [n_examples, n_hidden]
    # dot [n_classes, n_hidden].T
    # размерность выхода: [n_examples, n_classes]
    z_out = np.dot(a_h, self.weight_out.T) + self.bias_out
    a_out = sigmoid(z_out)

    return a_h, a_out
```

Метод `forward` принимает один или несколько обучающих примеров и возвращает прогнозы. На самом деле он возвращает значения активации как из скрытого, так и выходного слоя (`a_h` и `a_out` соответственно). Хотя `a_out` представляет вероятности принадлежности к классу, которые мы можем преобразовать в интересующие нас метки классов, нам также нужны значения активации из скрытого слоя `a_h` для оптимизации параметров модели, т. е. элементов веса и смещения скрытого и выходного слоев.

Наконец, рассмотрим метод `backward`, который обновляет параметры веса и смещения нейронной сети:

```
def backward(self, x, a_h, a_out, y):  
  
    #####  
    ### Веса выходного слоя  
    #####  
  
    # унитарное кодирование  
    y_onehot = int_to_onehot(y, self.num_classes)  
  
    # Часть 1: dLoss/dOutWeights  
    ## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight  
    ## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet  
    ## для удобства повторного использования  
  
    # размер входа/выхода: [n_examples, n_classes]  
    d_loss_d_a_out = 2.*(a_out - y_onehot) / y.shape[0]  
  
    # размер входа/выхода: [n_examples, n_classes]  
    d_a_out_d_z_out = a_out * (1. - a_out) # сигмоидная производная  
  
    # размер выхода: [n_examples, n_classes]  
    delta_out = d_loss_d_a_out * d_a_out_d_z_out  
  
    # градиент для выходных весов  
  
    # [n_examples, n_hidden]  
    d_z_out_dw_out = a_h  
  
    # размер входа: [n_classes, n_examples] dot [n_examples, n_hidden]  
    # размер выхода: [n_classes, n_hidden]  
    d_loss_dw_out = np.dot(delta_out.T, d_z_out_dw_out)  
    d_loss_db_out = np.sum(delta_out, axis=0)  
  
    #####  
    # Часть 2: dLoss/dHiddenWeights  
    ## = DeltaOut * dOutNet/dHiddenAct * dHiddenAct/dHiddenNet * dHiddenNet/dWeight  
  
    # [n_classes, n_hidden]  
    d_z_out_a_h = self.weight_out  
  
    # размер выхода: [n_examples, n_hidden]  
    d_loss_a_h = np.dot(delta_out, d_z_out_a_h)  
  
    # [n_examples, n_hidden]  
    d_a_h_d_z_h = a_h * (1. - a_h) # сигмоидная производная
```

```
# [n_examples, n_features]
d_z_h_d_w_h = x

# размер выхода: [n_hidden, n_features]
d_loss_d_w_h = np.dot((d_loss_a_h * d_a_h_d_z_h).T, d_z_h_d_w_h)
d_loss_d_b_h = np.sum((d_loss_a_h * d_a_h_d_z_h), axis=0)

return (d_loss_dw_out, d_loss_db_out,
        d_loss_d_w_h, d_loss_d_b_h)
```

Метод `backward` реализует так называемый алгоритм *обратного распространения*, который вычисляет градиенты потерь по отношению к параметрам веса и смещения. Подобно тому, как это делалось в Adaline, найденные градиенты затем используются для обновления этих параметров посредством градиентного спуска. Многослойные нейронные сети более сложны, чем их однослойные сородичи, поэтому мы разберемся с математическими основами вычисления градиентов в отдельном разделе после обсуждения кода, а пока будем просто рассматривать метод `backward` как способ вычисления градиентов, которые применяются для обновлений градиентного спуска. Для простоты сочтем, что функция потерь, на которой основан этот вывод, — уже знакомые потери MSE, которые мы использовали в Adaline. В последующих главах мы рассмотрим альтернативные функции потерь — такие как многоклассовые кросс-энтропийные потери, которые являются обобщением потерь бинарной логистической регрессии на нескольких классах.

Глядя на приведенную реализацию класса `NeuralNetMLP`, вы, возможно, заметили, что эта объектно-ориентированная реализация отличается от знакомого API scikit-learn, который базируется на методах `.fit()` и `.predict()`. В нашем случае основными методами класса `NeuralNetMLP` являются `.forward()` и `.backward()`. Одна из причин этого заключается в том, что сложную нейронную сеть становится немного легче понять с точки зрения того, как через нее проходит информация.

Другая причина заключается в том, что эта реализация отчасти похожа на то, как работают более продвинутые библиотеки глубокого обучения, такие как PyTorch, которыми мы воспользуемся в следующих главах для реализации более сложных нейронных сетей.

Подготовив класс `NeuralNetMLP`, для создания экземпляра нового объекта `NeuralNetMLP` мы используем следующий код:

```
>>> model = NeuralNetMLP(num_features=28*28,
...                         num_hidden=50,
...                         num_classes=10)
```

Экземпляр `model` принимает изображения MNIST, преобразованные в 784-мерные векторы (в формате `X_train`, `X_valid` или `X_test`, который мы определили ранее) для 10 целочисленных классов (цифры 0–9). Скрытый слой состоит из 50 узлов. Кроме того, как можно заметить, взглянув на ранее определенный метод `.forward()`, мы используем для простоты сигмоидную функцию активации после первого скрытого слоя и выходного слоя. В последующих главах вы узнаете об альтернативных функциях активации как для скрытого, так и для выходного слоев.

На рис. 11.6 показана архитектура нейронной сети, которую мы создали при помощи приведенного в этом разделе кода.

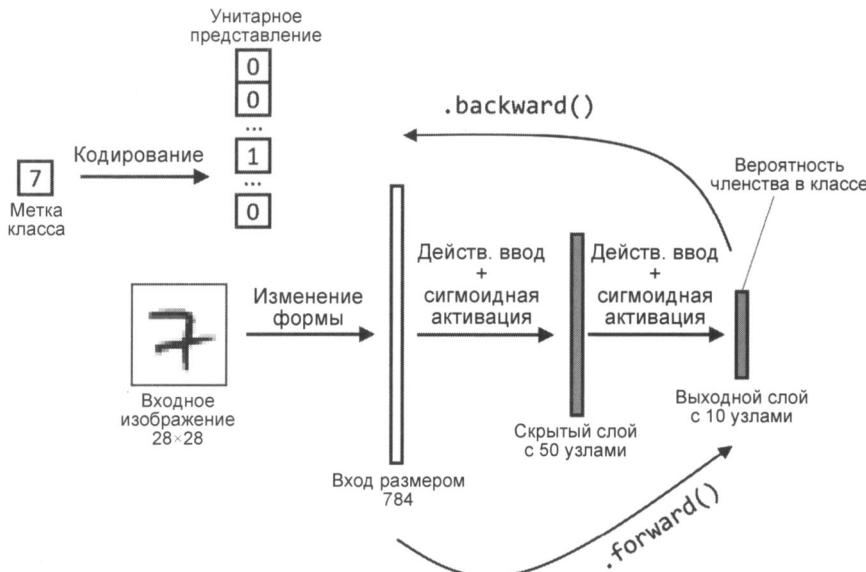


Рис. 11.6. Архитектура нейронной сети для распознавания рукописных цифр

Далее мы реализуем обучающую функцию, которую будем использовать для обучения сети на мини-пакетах данных с помощью метода обратного распространения ошибки.

11.2.3. Обучающий цикл нейронной сети

В предыдущем разделе мы реализовали класс NeuralNetMLP и инициировали модель. Теперь ее необходимо обучить. Мы решим эту задачу за несколько шагов: сначала определим некоторые вспомогательные функции для загрузки данных, а затем встроим эти функции в цикл обучения, перебирающий набор данных в течение нескольких эпох.

Первая функция, которую надо определить, — это генератор мини-пакетов, принимающий наш набор данных и делящий его на мини-пакеты требуемого размера для обучения модели методом стохастического градиентного спуска:

```
>>> import numpy as np
>>> num_epochs = 50
>>> minibatch_size = 100

>>> def minibatch_generator(X, y, minibatch_size):
...     indices = np.arange(X.shape[0])
...     np.random.shuffle(indices)
...     for start_idx in range(0, indices.shape[0] - minibatch_size + 1, minibatch_size):
...         batch_idx = indices[start_idx:start_idx + minibatch_size]
...         yield X[batch_idx], y[batch_idx]
```

Прежде чем перейти к следующим функциям, надо убедиться, что генератор мини-пакетов работает должным образом и создает мини-пакеты нужного размера. Следующий код пытается выполнить итерацию по набору данных, а затем выводит размер

мини-пакетов. Обратите внимание, что в следующих примерах кода мы удалим операторы `break`:

```
>>> # итерация по эпохам обучения
>>> for i in range(num_epochs):
...     # итерация по мини-пакетам
...     minibatch_gen = minibatch_generator(
...         X_train, y_train, minibatch_size)
...     for X_train_mini, y_train_mini in minibatch_gen:
...         break
...     break
>>> print(X_train_mini.shape)
(100, 784)
>>> print(y_train_mini.shape)
(100,)
```

Как видите, код возвращает мини-пакеты размером 100, как и предполагалось.

Затем надо определить функцию потерь и метрику производительности, которыми мы воспользуемся для мониторинга процесса обучения и оценки модели. Функция потерь и точности MSE может быть реализована так:

```
>>> def mse_loss(targets, probas, num_labels=10):
...     onehot_targets = int_to_onehot(
...         targets, num_labels=num_labels
...     )
...     return np.mean((onehot_targets - probas)**2)

>>> def accuracy(targets, predicted_labels):
...     return np.mean(predicted_labels == targets)
```

Для проверки этой функции вычислим начальные значения MSE и точности на валидационном наборе для модели, которую мы создали в предыдущем разделе:

```
>>> _, probas = model.forward(X_valid)
>>> mse = mse_loss(y_valid, probas)
>>> print(f'Начальная MSE при валидации: {mse:.1f}')
Начальная MSE при валидации: 0.3

>>> predicted_labels = np.argmax(probas, axis=1)
>>> acc = accuracy(y_valid, predicted_labels)
>>> print(f'Начальная точность при валидации: {acc*100:.1f}%')
Начальная точность при валидации: 9.4%
```

В этом примере кода `model.forward()` возвращает активацию скрытого и выходного слоев. Как вы помните, у нас есть 10 выходных узлов (по одному на каждую уникальную метку класса). Поэтому при вычислении MSE мы сначала преобразовали обычные метки классов в метки с унитарным кодированием при помощи функции `mse_loss()`. На практике не имеет значения, усредняем ли мы сначала строку или столбцы матрицы квадратов разностей, поэтому мы просто вызываем `np.mean()` без указания оси, чтобы она возвращала скаляр.

Поскольку мы задействовали логистическую сигмоидную функцию, активации выходного слоя представляют собой значения в диапазоне [0, 1]. В общей сложности выход-

ной слой выдает 10 значений в диапазоне [0, 1], поэтому мы воспользовались функцией `np.argmax()` для выяснения индекса наибольшего значения. Этот индекс фактически обозначает прогнозируемую метку класса. Затем мы сравнили истинные метки с предсказанными метками классов, чтобы вычислить точность с помощью определенной нами функции `precision()`. Судя по значению, которое код вывел на экран, точность не очень высока. Однако, учитывая, что у нас есть сбалансированный набор данных с 10 классами, точность предсказания на уровне около 10 процентов — это именно то, чего следует ожидать от необученной модели, производящей случайные предсказания.

Используя приведенный код, мы могли бы вычислить производительность для всего обучающего набора, если бы предоставили `y_train` в качестве входных данных, а метки — с помощью `x_train`. Однако на практике объем памяти компьютера обычно ограничивает размер данных, которые модель может принять за один прямой проход (из-за умножения больших матриц). Поэтому мы вычисляем MSE и точность с использованием генератора мини-пакетов. Следующая функция будет постепенно вычислять MSE и точность, перебирая набор данных по одному мини-пакету за раз, чтобы повысить эффективность использования памяти:

```
>>> def compute_mse_and_acc(nnet, X, y, num_labels=10,
...                         minibatch_size=100):
...     mse, correct_pred, num_examples = 0., 0, 0
...     minibatch_gen = minibatch_generator(X, y, minibatch_size)
...     for i, (features, targets) in enumerate(minibatch_gen):
...         _, probas = nnet.forward(features)
...         predicted_labels = np.argmax(probas, axis=1)
...         onehot_targets = int_to_onehot(
...             targets, num_labels=num_labels
...         )
...         loss = np.mean((onehot_targets - probas)**2)
...         correct_pred += (predicted_labels == targets).sum()
...         num_examples += targets.shape[0]
...         mse += loss
...     mse = mse/i
...     acc = correct_pred/num_examples
...     return mse, acc
```

Прежде чем перейти к реализации цикла обучения, протестируем функцию на мини-пакетах и вычислим MSE начального обучающего набора и точность модели, которую мы создали в предыдущем разделе, и убедимся, что она работает так, как предполагалось:

```
>>> mse, acc = compute_mse_and_acc(model, X_valid, y_valid)
>>> print(f'Начальная MSE при валидации: {mse:.1f}')
Начальная MSE при валидации: 0.3
>>> print(f'Начальная точность при валидации: {(acc*100:.1f)%}')
Начальная точность при валидации: 9.4%
```

Как следует из приведенного вывода, подход с генерацией мини-пакетов дает те же результаты, что и ранее определенные функции MSE и точности, за исключением небольшой ошибки округления в MSE (0.27 по сравнению с 0.28), которой в нашем случае можно пренебречь.

Итак, теперь мы можем перейти к основной части и разработать код для обучения модели:

```
>>> def train(model, X_train, y_train, X_valid, y_valid, num_epochs,
...           learning_rate=0.1):
...     epoch_loss = []
...     epoch_train_acc = []
...     epoch_valid_acc = []
...
...     for e in range(num_epochs):
...         # Итерация по мини-пакетам
...         minibatch_gen = minibatch_generator(
...             X_train, y_train, minibatch_size)
...         for X_train_mini, y_train_mini in minibatch_gen:
...             ##### Вычисление выходов #####
...             a_h, a_out = model.forward(X_train_mini)
...
...             ##### Вычисление градиентов #####
...             d_loss_d_w_out, d_loss_d_b_out, \
...             d_loss_d_w_h, d_loss_d_b_h = \
...             model.backward(X_train_mini, a_h, a_out,
...                           y_train_mini)
...
...             ##### Обновление весов #####
...             model.weight_h -= learning_rate * d_loss_d_w_h
...             model.bias_h -= learning_rate * d_loss_d_b_h
...             model.weight_out -= learning_rate * d_loss_d_w_out
...             model.bias_out -= learning_rate * d_loss_d_b_out
...
...             ##### Ведение журнала эпох #####
...             train_mse, train_acc = compute_mse_and_acc(
...                 model, X_train, y_train
...             )
...             valid_mse, valid_acc = compute_mse_and_acc(
...                 model, X_valid, y_valid
...             )
...             train_acc, valid_acc = train_acc*100, valid_acc*100
...             epoch_train_acc.append(train_acc)
...             epoch_valid_acc.append(valid_acc)
...             epoch_loss.append(train_mse)
...             print(f'Эпоха: {e+1:03d}/{num_epochs:03d} '
...                   f'| Train MSE: {train_mse:.2f} '
...                   f'| Train Acc: {train_acc:.2f}% '
...                   f'| Valid Acc: {valid_acc:.2f}%')
...
...     return epoch_loss, epoch_train_acc, epoch_valid_acc
```

На верхнем уровне функция `train()` перебирает несколько эпох и в каждой эпохе использует ранее определенную функцию `minibatch_generator()` для перебора всего обучающего набора в мини-пакетах для обучения методом стохастического градиентного

спуска. Внутри цикла `for` генератора мини-пакетов мы получаем выходные данные модели `a_h` и `a_out` с помощью ее метода `.forward()`. Затем мы вычисляем градиенты потерь с помощью метода модели `.backward()` — теоретические основы мы объясним вам в следующем разделе. Используя градиенты потерь, мы обновляем веса, добавляя отрицательный градиент, умноженный на скорость обучения. Аналогичный подход мы применяли ранее для Adaline. Например, чтобы обновить веса модели скрытого слоя, мы выполняем следующую строку:

```
model.weight_h -= learning_rate * d_loss_d_w_h
```

Для одиночного веса w_j это соответствует следующему обновлению на основе частной производной:

$$w_j := w_j - \eta \frac{\partial L}{\partial w_j}.$$

Наконец, последняя часть приведенного кода вычисляет потери и точность прогнозирования на обучающем и тестовом наборах, чтобы отслеживать ход обучения.

Теперь запустите следующий код, чтобы обучить нашу модель в течение 50 эпох (это может занять несколько минут):

```
>>> np.random.seed(123) # перемешивание обучающего набора
>>> epoch_loss, epoch_train_acc, epoch_valid_acc = train(
...     model, X_train, y_train, X_valid, y_valid,
...     num_epochs=50, learning_rate=0.1)
```

Во время обучения вы должны увидеть следующий вывод:

```
Epoch: 001/050 | Train MSE: 0.05 | Train Acc: 76.17% | Valid Acc: 76.02%
Epoch: 002/050 | Train MSE: 0.03 | Train Acc: 85.46% | Valid Acc: 84.94%
Epoch: 003/050 | Train MSE: 0.02 | Train Acc: 87.89% | Valid Acc: 87.64%
Epoch: 004/050 | Train MSE: 0.02 | Train Acc: 89.36% | Valid Acc: 89.38%
Epoch: 005/050 | Train MSE: 0.02 | Train Acc: 90.21% | Valid Acc: 90.16%
...
Epoch: 048/050 | Train MSE: 0.01 | Train Acc: 95.57% | Valid Acc: 94.58%
Epoch: 049/050 | Train MSE: 0.01 | Train Acc: 95.55% | Valid Acc: 94.54%
Epoch: 050/050 | Train MSE: 0.01 | Train Acc: 95.59% | Valid Acc: 94.74%
```

Причина, по которой мы выводим на экран все эти числа, заключается в том, что при обучении нейросети действительно полезно следить за изменением точности модели на обучающих и тренировочных данных. Это помогает нам судить о том, хорошо ли работает сетевая модель с текущей архитектурой и гиперпараметрами. Например, если мы наблюдаем низкую точность на обучающих и проверочных данных, то, вероятно, существует проблема с набором обучающих данных или настройки гиперпараметров слишком далеки от идеальных.

В общем случае обучение глубоких нейронных сетей обходится относительно дорого по сравнению с другими моделями, которые мы обсуждали до сих пор. Поэтому при определенных обстоятельствах было бы хорошо остановить процесс обучения как можно раньше и начать его заново с новыми настройками гиперпараметров. Кроме того, если мы обнаружим, что модель склонна к переобучению (что заметно по увеличивающемуся разрыву между производительностью на обучающем и тестовом наборах), также имеет смысл остановить обучение досрочно.

В следующем разделе мы более подробно обсудим производительность нашей нейросетевой модели.

11.2.4. Оценка производительности нейронной сети

В следующем разделе мы займемся подробным изучением механизма обратного распространения ошибки, а сейчас проанализируем производительность модели, которую только что обучили.

В функции `train()` мы собрали потери при обучении, а также точность модели на обучающем и валидационном наборах для каждой эпохи, чтобы визуализировать результаты с помощью `Matplotlib`. Давайте сначала посмотрим на потери по метрике MSE при обучении:

```
>>> plt.plot(range(len(epoch_loss)), epoch_loss)
>>> plt.ylabel('Среднеквадратичная ошибка (MSE)')
>>> plt.xlabel('Эпохи')
>>> plt.show()
```

Этот код отображает потери за 50 эпох (рис. 11.7).

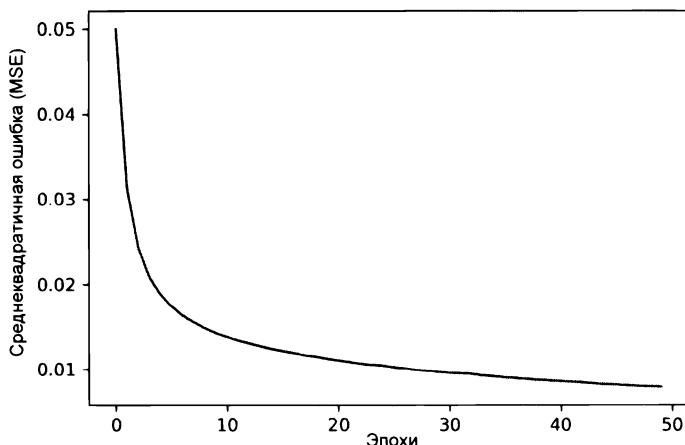


Рис. 11.7. График MSE в зависимости от количества эпох обучения

Как можно здесь видеть, потери существенно уменьшились в течение первых 10 эпох и, очевидно, модель медленно сходится в последние 10 эпох. Однако небольшой наклон между эпохами 40 и 50 указывает на то, что потери могут уменьшаться при обучении в течение дополнительных эпох.

Теперь отобразим на графике точность модели на обучающем и валидационном наборах:

```
>>> plt.plot(range(len(epoch_train_acc)), epoch_train_acc,
...           label='Обучение')
>>> plt.plot(range(len(epoch_valid_acc)), epoch_valid_acc,
...           label='Валидация')
>>> plt.ylabel('Точность')
```

```
>>> plt.xlabel('Эпохи')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

Этот код отображает значения точности за 50 эпох обучения (рис. 11.8).

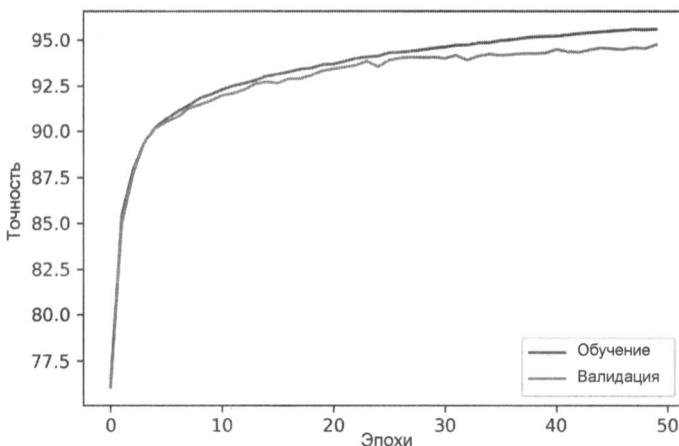


Рис. 11.8. Точность классификации в зависимости от количества эпох обучения

Как можно здесь видеть, разрыв между точностью на обучающих и валидационных данных увеличивается по мере увеличения количества эпох. Примерно на 25-й эпохе значения точности обучения и валидации почти равны, а затем сеть начинает демонстрировать первые признаки переобучения.



Борьба с переобучением

Один из способов борьбы с переобучением — увеличение степени регуляризации L2, о чём говорилось в главе 3. Другим полезным способом снижения эффекта переобучения является *дропаут* (исключение данных), о котором мы расскажем в главе 14.

Наконец, оценим способность модели к обобщению, рассчитав точность прогноза на тестовом наборе данных:

```
>>> test_mse, test_acc = compute_mse_and_acc(model, X_test, y_test)
>>> print(f'Точность при тестировании: {test_acc*100:.2f}%')
```

Точность при тестировании: 94.51%

Мы видим, что точность при тестировании очень близка к точности при валидации на последней эпохе (94.74%), которую мы измерили во время обучения в предыдущем разделе. Более того, соответствующая точность на обучающих данных лишь немногого выше — 95.59%. Это подтверждает предположение о том, что наша модель переобучилась, но совсем незначительно.

Для дальнейшей точной настройки модели мы могли бы изменить количество скрытых узлов, скорость обучения или использовать различные другие приемы, которые разрабатывались годами, но выходят за рамки этой книги. В главе 14 вы узнаете о другой

архитектуре нейронных сетей, известной своей хорошей производительностью при работе с наборами данных изображений.

Кроме того, там будут представлены дополнительные приемы повышения производительности — такие как адаптивная скорость обучения, более сложные алгоритмы оптимизации на основе SGD, пакетная нормализация и дропаут.

К другим распространенным приемам, которые выходят за рамки этой книги, относятся:

- ◆ добавление сквозных соединений, которые являются главной особенностью архитектуры остаточных нейронных сетей⁶;
- ◆ использование планировщиков скорости, изменяющих скорость обучения непосредственно во время обучающего процесса⁷;
- ◆ присоединение функций потерь к более ранним уровням в сетях, как это делается в популярной архитектуре Inception v3⁸.

Наконец, давайте взглянем на примеры изображений, с которыми не смог справиться наш MLP. Для этого извлечем и отобразим первые 25 неправильно классифицированных изображений из тестового набора:

```
>>> X_test_subset = X_test[:1000, :]
>>> y_test_subset = y_test[:1000]
>>> _, probas = model.forward(X_test_subset)
>>> test_pred = np.argmax(probas, axis=1)
>>> misclassified_images = \
... X_test_subset[y_test_subset != test_pred][:25]
>>> misclassified_labels = test_pred[y_test_subset != test_pred][:25]
>>> correct_labels = y_test_subset[y_test_subset != test_pred][:25]
>>> fig, ax = plt.subplots(nrows=5, ncols=5,
... sharex=True, sharey=True,
... figsize=(8, 8))
>>> ax = ax.flatten()
>>> for i in range(25):
... img = misclassified_images[i].reshape(28, 28)
... ax[i].imshow(img, cmap='Greys', interpolation='nearest')
... ax[i].set_title(f'{i+1})')
... f' True: {correct_labels[i]}\n'
... f' Predicted: {misclassified_labels[i]}')
>>> ax[0].set_xticks([])
>>> ax[0].set_yticks([])
>>> plt.tight_layout()
>>> plt.show()
```

⁶ См. «Deep residual learning for image recognition» by K. He, X. Zhang, S. Ren, and J. Sun, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, p. 770–778, 2016.

⁷ См. «Cyclical learning rates for training neural networks» by L. N. Smith, 2017 IEEE Winter Conference on Applications of Computer Vision (WACV), p. 464–472, 2017.

⁸ См. «Rethinking the Inception architecture for computer vision» by C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, p. 2818–2826, 2016.

Выполнив этот код, вы должны увидеть матрицу изображений 5×5 (рис. 11.9), где первое число в подзаголовках указывает номер примера (индекс), второе число представляет собой метку истинного класса (**True**), а третье число обозначает метку прогнозируемого класса (**Predicted**).



Рис. 11.9. Рукописные цифры, которые модель не смогла правильно классифицировать

Как можно здесь видеть, сеть, среди прочего, не способна распознавать семерки, когда они содержат горизонтальную линию, как в примерах 19 и 20. Вспоминая рис. 11.7, где мы отобразили различные обучающие примеры цифры 7, мы можем предположить, что написанная от руки цифра 7 с горизонтальной линией недостаточно часто встречается в нашем наборе данных и модель просто не научилась ее уверенно распознавать.

11.3. Обучение искусственной нейронной сети

Теперь, когда вы увидели нейронную сеть в действии и, просматривая код, получили общее представление о том, как она работает, настало время изучить базовые понятия в области машинного обучения — такие как вычисление потерь и алгоритм обратного распространения, которые мы уже использовали для обучения модели.

11.3.1. Вычисление функции потерь

Как было сказано ранее, мы использовали для обучения многослойной нейронной сети метрику потери MSE (как в Adaline), поскольку это немного упрощает получение гра-

диентов. В последующих главах мы обсудим другие функции потерь — такие как, например, функция потерь перекрестной энтропии для нескольких категорий (обобщение потери бинарной логистической регрессии), которая является более распространенным вариантом при обучении классифицирующих нейросетей.

В предыдущем разделе мы реализовали MLP для многоклассовой классификации, возвращающей выходной вектор из t элементов, которые нам нужно сравнить с целевым вектором размерности $t \times 1$ в представлении с унитарным кодированием. Если мы, используя этот MLP, предскажем метку класса входного изображения 2, активация третьего слоя и цели может выглядеть следующим образом:

$$\mathbf{a}^{(out)} = \begin{bmatrix} 0.1 \\ 0.9 \\ \vdots \\ 0.3 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}.$$

Следовательно, нашу потерю по метрике MSE нужно либо суммировать, либо усреднять по t элементам активации в нашей сети в дополнение к усреднению по n примерам в наборе данных или мини-пакете:

$$L(\mathbf{W}, \mathbf{b}) = \frac{1}{n} \sum_1^n \frac{1}{t} \sum_{j=1}^t (y_j^{[i]} - a_j^{(out)[i]})^2.$$

Здесь снова верхний индекс $[i]$ — это индекс конкретного примера в наборе обучающих данных.

Напомним, что наша цель — минимизировать функцию потерь $L(\mathbf{W})$. Значит, нам нужно вычислить частную производную параметров \mathbf{W} по каждому весу для каждого слоя в сети:

$$\frac{\partial}{\partial w_{j,l}} = L(\mathbf{W}, \mathbf{b}).$$

В следующем разделе мы поговорим об алгоритме обратного распространения, который позволяет нам вычислять эти частные производные для минимизации функции потерь.

Обратите внимание, что \mathbf{W} состоит из нескольких матриц. В MLP с одним скрытым слоем у нас есть матрица весов $\mathbf{W}^{(h)}$, которая соединяет вход со скрытым слоем, и матрица $\mathbf{W}^{(out)}$, которая соединяет скрытый слой с выходным слоем. Визуальное представление трехмерного тензора \mathbf{W} показано на рис. 11.10.

Глядя на этот упрощенный рисунок, можно подумать, что и $\mathbf{W}^{(h)}$, и $\mathbf{W}^{(out)}$ имеют одинаковое количество строк и столбцов, что обычно не так, если только мы не инициализируем MLP с одинаковым количеством скрытых узлов, узлов вывода и входных признаков.

Если пока это звучит непонятно, постарайтесь следить за ходом рассуждений в следующем разделе, где мы более подробно обсудим размерность $\mathbf{W}^{(h)}$ и $\mathbf{W}^{(out)}$ в контексте алгоритма обратного распространения. Кроме того, рекомендуем еще раз изучить код NeuralNetMLP, который снабжен полезными комментариями о размерности различных матриц и векторных преобразований.

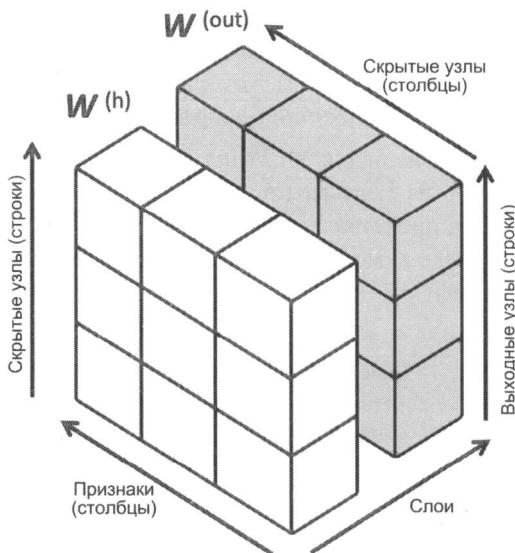


Рис. 11.10. Визуальное представление трехмерного тензора

11.3.2. Подробнее о механизме обратного распространения

Хотя идея обратного распространения была представлена сообществу исследователей нейронных сетей более 30 лет назад⁹, она остается наиболее широко используемым и очень эффективным алгоритмом обучения искусственных нейронных сетей. Если вас интересует история алгоритма обратного распространения ошибки, прочтите прекрасную обзорную статью Юргена Шмидхубера «Кто изобрел обратное распространение?», которую вы можете найти в Интернете по адресу: <http://people.idsia.ch/~juergen/who-invented-backpropagation.html>.

В этом разделе мы постараемся представить краткое и ясное описание, а также более полную картину того, как работает этот увлекательный алгоритм, и лишь затем углубимся в математические детали. По сути, мы можем рассматривать алгоритм обратного распространения ошибки как очень эффективный в вычислительном отношении подход к вычислению частных производных сложной невыпуклой функции потерь в многослойных нейронных сетях. Наша цель — использовать эти производные для обучения весовых коэффициентов (параметров) такой многослойной искусственной нейросети. Проблема параметризации нейросетей заключается в том, что мы обычно имеем дело с очень большим количеством параметров модели в многомерном пространстве признаков. В отличие от функций потерь однослойных сетей, таких как Adaline или логистическая регрессия, с которыми вы познакомились в предыдущих главах, поверхность ошибок функции потерь нейронной сети не является выпуклой или гладкой по отношению к параметрам. На этой многомерной поверхности потерь (локальных минимумов)

⁹ См. «Learning representations by backpropagating errors» by D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Nature, 323: 6088, p. 533–536, 1986.

есть много выпуклостей, которые мы должны преодолеть, чтобы найти *глобальный минимум* функции потерь.

Вспомните *цепное правило* вычисления производных из вводных занятий по математическому анализу. Цепное правило — это способ вычисления производной сложной вложенной функции, такой как $f(g(x))$, следующим образом:

$$\frac{d}{dx} [f(g(x))] = \frac{df}{dg} \cdot \frac{dg}{dx}.$$

Точно так же мы можем использовать цепное правило для произвольно длинной композиции функций. Например, предположим, что у нас есть пять различных функций: $f(x)$, $g(x)$, $h(x)$, $u(x)$ и $v(x)$, и пусть F — композиция функций: $F(x) = f(g(h(u(v(x)))))$. Применяя цепное правило, мы можем вычислить производную этой функции:

$$\frac{dF}{dx} = \frac{d}{dx} F(x) = \frac{d}{dx} f(g(h(u(v(x))))) = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{du} \cdot \frac{du}{dv} \cdot \frac{dv}{dx}.$$

В контексте компьютерной алгебры для очень эффективного решения таких задач был разработан набор методов, известных как *автоматическое дифференцирование*¹⁰.

Автоматическое дифференцирование имеет два режима: прямой и обратный, так что обратное распространение — это просто частный случай автоматического дифференцирования в обратном режиме. Ключевым моментом здесь является то, что применение цепного правила в прямом режиме может быть довольно затратным, поскольку нам придется перемножать большие матрицы для каждого слоя (якобианы), которые мы в конечном итоге умножим на вектор для получения выходных данных.

Хитрость обратного режима заключается в том, что мы проходим по цепному правилу справа налево. Мы умножаем матрицу на вектор, что дает другой вектор, который умножается на следующую матрицу, и т. д. Умножение матрицы на вектор в вычислительном отношении намного дешевле, чем умножение матрицы на матрицу, поэтому обратное распространение является одним из самых популярных алгоритмов, используемых при обучении нейронных сетей.



Вспомним основы линейной алгебры

Чтобы хорошо усвоить принцип обратного распространения, нам нужно воспользоваться некоторыми понятиями из дифференциального исчисления, что выходит за рамки этой книги. Однако вы можете обратиться к обзору наиболее фундаментальных понятий, которые будут полезными в этом контексте. В нем говорится про производные функций, частные производные, градиенты и якобиан. Соответствующий учебник свободно доступен по адресу: https://sebastianraschka.com/pdf/books/dlb/appendix_d_calculus. Если вы не знакомы с основами линейной алгебры или вам нужно кратко освежить знания, то имеет смысл прочитать этот учебник (или какой-либо другой) в качестве дополнительного вспомогательного ресурса, прежде чем переходить к следующему разделу.

¹⁰ Если вы хотите узнать больше об автоматическом дифференцировании в приложениях машинного обучения, прочитайте статью А. Г. Байдина и Б. А. Перлмуттера (A. G. Baydin and B. A. Pearlmutter, «Automatic Differentiation of Algorithms for Machine Learning», arXiv preprint arXiv:1404.7456, 2014), которая находится в свободном доступе на arXiv по адресу: <http://arxiv.org/pdf/1404.7456.pdf>.

11.3.3. Обучение нейронных сетей с помощью обратного распространения

В этом разделе мы рассмотрим математические основы механизма обратного распространения — чтобы разобраться, как происходит обучение весовых коэффициентов нейронной сети. Если у вас недостаточно опыта занятий прикладной математикой, следующие уравнения могут сначала показаться вам относительно сложными.

В предыдущем разделе было показано вычисление потери как разницы между активацией последнего слоя и меткой целевого класса. Теперь мы рассмотрим, как работает алгоритм обратного распространения для обновления весов в нашей модели MLP с математической точки зрения. Мы реализовали этот алгоритм в методе `.backward()` класса `NeuralNetMLP()`. Как вы помните из начала этой главы, чтобы получить активацию выходного слоя, нам сначала нужно применить прямое распространение, которое можно записать следующим образом:

- ◆ $\mathbf{Z}^{(h)} = \mathbf{X}^{(in)} \mathbf{W}^{(h)T} + \mathbf{b}^{(h)}$ — действующий вход скрытого слоя;
- ◆ $\mathbf{A}^{(h)} = \sigma(\mathbf{Z}^{(h)})$ — активация скрытого слоя;
- ◆ $\mathbf{Z}^{(out)} = \mathbf{A}^{(h)} \mathbf{W}^{(out)T} + \mathbf{b}^{(out)}$ — действующий вход выходного слоя;
- ◆ $\mathbf{A}^{(out)} = \sigma(\mathbf{Z}^{(out)})$ — активация выходного слоя.

Коротко говоря, мы просто распространяем входные признаки через сетевые связи, как показано стрелками на рис. 11.11, для сети с двумя входными признаками, тремя скрытыми узлами и двумя выходными узлами.

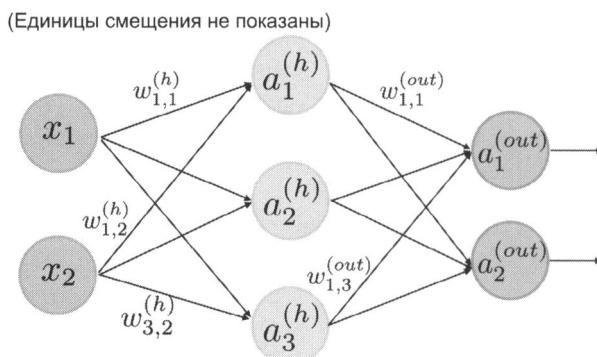


Рис. 11.11. Прямое распространение входных признаков в нейросети

При обратном распространении мы распространяем ошибку в обратном направлении — справа налево. Обратное распространение можно рассматривать как применение цепного правила к вычислению прямого распространения для нахождения градиента потерь по отношению к весам модели (и единицам смещения). Для простоты мы проиллюстрируем этот процесс на примере частной производной, используемой для обновления первого веса в матрице весов выходного слоя. Пути обратного распространения на рис. 11.12 выделены жирными стрелками.

Если мы запишем действующие входы z в явном виде, вычисление частной производной, показанное на рис. 11.12, примет следующий вид:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}}.$$

Чтобы вычислить эту частную производную, которая используется для обновления $w_{1,1}^{(out)}$, достаточно вычислить три отдельных члена частной производной и перемножить результаты. Для простоты мы опустим усреднение по отдельным примерам в мини-пакете, поэтому исключим член усреднения $\frac{1}{n} \sum_{i=1}^n$ из следующих уравнений.

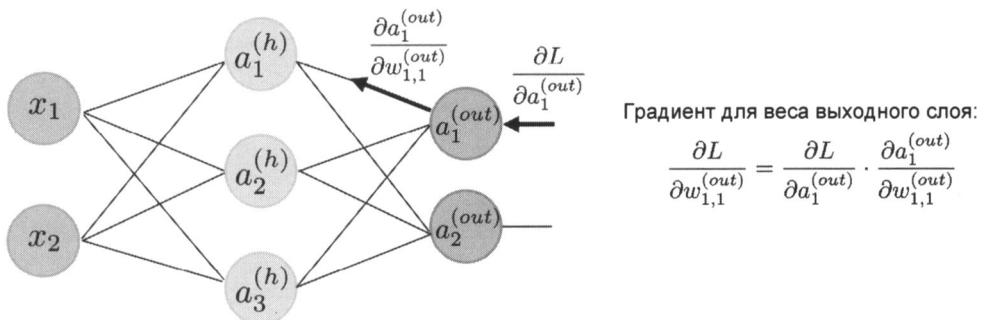


Рис. 11.12. Обратное распространение ошибки нейросети

Начнем с члена $\frac{\partial L}{\partial a_1^{(out)}}$, который является частной производной потери MSE (она упрощается до квадрата ошибки, если мы опускаем размер мини-пакета) по отношению к прогнозируемой выходной оценке первого выходного узла:

$$\frac{\partial L}{\partial a_1^{(out)}} = \frac{\partial}{\partial a_1^{(out)}} (y_1 - a_1^{(out)})^2 = 2(a_1^{(out)} - y)$$

Следующий член — это производная логистической сигмоидной функции активации, которую мы использовали в выходном слое:

$$\begin{aligned} \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} &= \frac{\partial}{\partial z_1^{(out)}} \frac{1}{1 + e^{z_1^{(out)}}} = \dots = \left(\frac{1}{1 + e^{z_1^{(out)}}} \right) \left(1 - \frac{1}{1 + e^{z_1^{(out)}}} \right) = \\ &= a_1^{(out)}(1 - a_1^{(out)}). \end{aligned}$$

Наконец, вычислим производную действующего входа по весу:

$$\frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = \frac{\partial}{\partial w_{1,1}^{(out)}} a_1^{(h)} w_{1,1}^{(out)} + b_1^{(out)} = a_1^{(h)}.$$

Собрав все компоненты вместе, получаем следующее уравнение:

$$\frac{\partial L}{\partial w_{1,1}^{(out)}} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial w_{1,1}^{(out)}} = 2(a_1^{(out)} - y) \cdot a_1^{(out)}(1 - a_1^{(out)}) \cdot a_1^{(h)}.$$

После чего используем найденное значение для обновления веса с помощью уже знакомого вам обновления стохастического градиентного спуска со скоростью обучения η :

$$w_{1,1}^{(out)} := w_{1,1}^{(out)} - \eta \frac{\partial L}{\partial w_{1,1}^{(out)}}.$$

В нашей реализации `NeuralNetMLP()` мы вычисляем $\frac{\partial L}{\partial w_{1,1}^{(out)}}$ в векторизованной форме

в методе `.backward()` следующим образом:

```
# Часть 1: dLoss/dOutWeights
## = dLoss/dOutAct * dOutAct/dOutNet * dOutNet/dOutWeight
## where DeltaOut = dLoss/dOutAct * dOutAct/dOutNet для удобства повторного использования

# Размер входа/выхода: [n_examples, n_classes]
d_loss_d_a_out = 2.*(a_out - y_onehot) / y.shape[0]

# Размер входа/выхода: [n_examples, n_classes]
d_a_out_d_z_out = a_out * (1. - a_out) # Производная сигмоиды

# Размер выхода: [n_examples, n_classes]
delta_out = d_loss_d_a_out * d_a_out_d_z_out # Заполнитель delta

# Градиент для выходных весов

# [n_examples, n_hidden]

d_z_out_dw_out = a_h

# Размер входа: [n_classes, n_examples] dot [n_examples, n_hidden]
# Размер выхода: [n_classes, n_hidden]
d_loss_dw_out = np.dot(delta_out.T, d_z_out_dw_out)
d_loss_db_out = np.sum(delta_out, axis=0)
```

Как показано в приведенном фрагменте кода, мы создали следующую переменную-заполнитель `delta`:

$$\delta_1^{(out)} = \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}}.$$

Это связано с тем, что члены $\delta^{(out)}$ также участвуют в вычислении частных производных (или градиентов) весов скрытого слоя, — следовательно, мы можем повторно использовать $\delta^{(out)}$.

Что же касается весов скрытого слоя, то на рис. 11.13 показано, как вычислить частную производную потерь по первому весу скрытого слоя.

Важно подчеркнуть, что, поскольку вес $w_{1,1}^{(h)}$ связан с обоими выходными узлами, мы должны использовать цепное правило с несколькими переменными для суммирования двух путей, выделенных жирными стрелками. Как и раньше, мы можем переписать уравнение, включив действующие входные данные z , а затем найти отдельные члены:

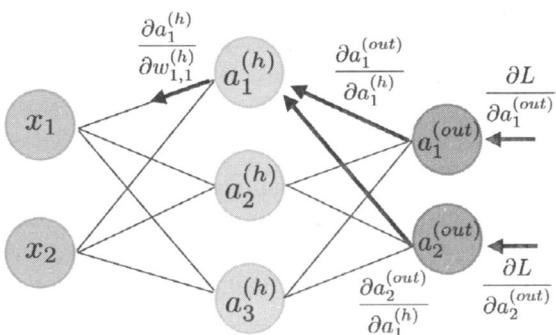
$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(out)}} &= \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial z_1^{(out)}} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \\ &+ \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial z_2^{(out)}} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}.\end{aligned}$$

Если мы повторно используем член $\delta^{(out)}$, вычисленный ранее, это уравнение можно упростить до вида:

$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(h)}} &= \delta_1^{(out)} \cdot \frac{\partial z_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}} + \\ &+ \delta_2^{(out)} \cdot \frac{\partial z_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial z_1^{(h)}} \cdot \frac{\partial z_1^{(h)}}{\partial w_{1,1}^{(h)}}.\end{aligned}$$

Предыдущие члены могут быть относительно легко найдены по отдельности, как мы делали ранее, потому что здесь не используются новые производные. Например, $\frac{\partial a_1^{(h)}}{\partial z_1^{(h)}}$

является производной сигмоидной активации, т. е. $a_1^{(h)}(1 - a_1^{(h)})$, и т. д. Мы оставим вычисление остальных членов уравнения в качестве дополнительного упражнения для читателей.



Градиент для веса выходного слоя:

$$\begin{aligned}\frac{\partial L}{\partial w_{1,1}^{(h)}} &= \frac{\partial L}{\partial a_1^{(out)}} \cdot \frac{\partial a_1^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}} + \\ &+ \frac{\partial L}{\partial a_2^{(out)}} \cdot \frac{\partial a_2^{(out)}}{\partial a_1^{(h)}} \cdot \frac{\partial a_1^{(h)}}{\partial w_{1,1}^{(h)}}\end{aligned}$$

Рис. 11.13. Вычисление частных производных потерь по первому весу скрытого слоя

11.4. О сходимости в нейронных сетях

Вы можете спросить, почему мы не воспользовались обычным градиентным спуском, а вместо этого выбрали для нашего нейросетевого классификатора рукописных цифр мини-пакетное обучение. Вспомните наше обсуждение стохастического градиентного спуска (SGD), который мы использовали для реализации онлайн-обучения. В онлайн-обучении мы вычисляем градиент на основе одного обучающего примера ($k = 1$) за раз, чтобы выполнить обновление веса. Хотя это стохастический подход, он часто приводит к очень точным решениям с гораздо более быстрой сходимостью, чем обычный гради-

ентный спуск. Мини-пакетное обучение — это особая форма SGD, в которой мы вычисляем градиент на основе подмножества k из n обучающих примеров, где $1 < k < n$. Мини-пакетное обучение имеет преимущество перед онлайн-обучением, поскольку мы можем использовать векторизованные представления для повышения эффективности вычислений и при этом обновлять веса намного быстрее, чем при обычном градиентном спуске. Для наглядности, мини-пакетное обучение можно рассматривать как прогнозирование явки избирателей на президентских выборах на основе опроса репрезентативной выборки населения, а не всего населения (что было бы равносильно проведению реальных выборов).

Многослойные нейронные сети гораздо сложнее обучить, чем такие простые алгоритмы, как Adaline, логистическая регрессия или метод опорных векторов. В многослойных сетях у нас обычно есть сотни, тысячи или даже миллиарды весов, которые нам нужно оптимизировать. К сожалению, выходная функция имеет шероховатую поверхность, и алгоритм оптимизации может легко попасть в ловушку локальных минимумов, как показано на рис. 11.14.



Рис. 11.14. Алгоритмы оптимизации могут попасть в ловушку локальных минимумов

Это представление чрезвычайно упрощено, поскольку нейронная сеть обычно имеет много измерений, что делает невозможным визуальное представление многомерной поверхности потерь в трехмерном пространстве. Так, на рис. 11.14 показана поверхность потерь для единственного веса по оси x . Однако основная идея заключается в том, чтобы избежать ловушки локальных минимумов. Увеличение скорости обучения повышает шанс пройти мимо таких локальных минимумов. С другой стороны, мы также увеличиваем вероятность проскочить глобальный оптимум, если скорость обучения окажется слишком велика. Поскольку мы инициализируем веса модели случайным образом, обычно мы начинаем с глубоко ошибочного решения задачи оптимизации.

11.5. Несколько заключительных слов о реализации нейронной сети

Может возникнуть вопрос: зачем мы вообще изучали всю эту теорию, если нам нужно было всего лишь реализовать простую многослойную нейронную сеть, способную классифицировать рукописные цифры, и для этого было бы достаточно воспользоваться библиотекой машинного обучения Python с открытым исходным кодом? И действи-

тельно, в следующих главах мы рассмотрим более сложные нейросетевые модели, которые будем обучать с использованием библиотеки PyTorch с открытым исходным кодом (<https://pytorch.org>).

Хотя реализация модели с нуля в этой главе поначалу кажется немного утомительной, она послужила вам хорошим упражнением для понимания основ обратного распространения ошибки и обучения нейронных сетей. Понимание математической основы алгоритмов имеет решающее значение для правильного и успешного применения методов машинного обучения.

Теперь, когда вы узнали, как работают нейронные сети прямого распространения, мы готовы исследовать более сложные глубокие нейросети с помощью PyTorch. Эта библиотека позволяет более эффективно создавать нейронные сети, как будет показано в [главе 12](#).

PyTorch, первоначально выпущенная в сентябре 2016 г., приобрела большую популярность среди исследователей машинного обучения, которые используют ее для построения глубоких нейросетей благодаря встроенной в нее возможности оптимизировать математические выражения для вычислений в многомерных массивах с использованием графических процессоров (GPU).

Наконец, мы должны отметить, что scikit-learn тоже имеет базовую реализацию MLP под названием `MLPClassifier`, которую вы можете найти по адресу: https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html. Хотя эта реализация великолепна и очень удобна при обучении базовых MLP, для разработки и обучения многослойных нейронных сетей мы настоятельно рекомендуем применять специализированные библиотеки глубокого обучения — такие как PyTorch.

11.6. Заключение

В этой главе вы познакомились с основными понятиями из области многослойных искусственных нейронных сетей, которые в настоящее время являются самой горячей темой в исследованиях машинного обучения. В [главе 2](#) мы начали с простых однослойных структур нейронной сети, а теперь задействовали множество нейронов в мощной архитектуре нейронной сети для решения сложных задач — таких как распознавание рукописных цифр. Вы воочию увидели механизм работы популярного алгоритма обратного распространения ошибки, который стал одним из основных функциональных компонентов многих нейросетевых моделей, используемых в глубоком обучении. Теперь вы готовы перейти к изучению более сложных архитектур DNN. В оставшихся главах мы рассмотрим более продвинутые концепции глубокого обучения и библиотеку PyTorch с открытым исходным кодом, которая позволяет более эффективно создавать и обучать многослойные нейронные сети.

12

Глубокое обучение нейронных сетей на основе PyTorch

В этой главе мы перейдем от изучения математических основ машинного обучения и глубокого обучения к знакомству с PyTorch. PyTorch — это одна из самых популярных библиотек глубокого обучения, доступных в настоящее время, и она позволяет реализовывать нейронные сети гораздо эффективнее, чем любая из наших предыдущих реализаций на основе NumPy. В этой главе мы начнем работу с PyTorch, и вы убедитесь, что эта библиотека значительно увеличивает производительность обучения.

Освоение материала этой главы откроет следующий этап нашего путешествия в области машинного и глубокого обучения. Мы рассмотрим здесь такие темы:

- ◆ роль PyTorch в повышении производительности обучения;
- ◆ применение классов PyTorch `Dataset` и `DataLoader` для создания конвейеров ввода и обеспечения эффективного обучения модели;
- ◆ применение PyTorch для написания оптимизированного кода машинного обучения;
- ◆ применение модуля `torch.nn` для удобной реализации распространенных архитектур глубокого обучения;
- ◆ выбор функций активации для искусственных нейронных сетей.

12.1. PyTorch и производительность обучения

PyTorch может значительно ускорить выполнение задач машинного обучения. Чтобы понять, как это происходит, нужно начать с обсуждения некоторых проблем с производительностью и быстродействием, которые обычно возникают, когда мы выполняем на нашем оборудовании громоздкие вычисления. Затем мы в общих чертах рассмотрим, что такое PyTorch, и каков будет наш подход к обучению в этой главе.

12.1.1. Проблемы с быстродействием

Как известно, в последние годы быстродействие компьютерных процессоров постоянно возрастает. Это позволяет нам использовать более мощные и сложные системы обучения, а значит, мы можем улучшить прогностическую эффективность моделей машинного обучения. Даже самые дешевые настольные компьютеры сегодня поставляются с многоядерными центральными процессорами (CPU).

В предыдущих главах вы узнали, что многие функции в scikit-learn дают нам возможность распределять трудоемкие вычисления по нескольким процессорам. Однако по умолчанию Python ограничен выполнением кода на одном ядре из-за *глобальной блокировки интерпретатора* (Global Interpreter Lock, GIL). И даже когда мы используем многопроцессорную библиотеку Python для распределения наших вычислений по нескольким ядрам, приходится учитывать, что самые современные настольные компьютеры редко имеют составе своего центрального процессора более 8 или 16 таких ядер.

Как вы помните, в главе 11 мы построили очень простой многослойный персепtron (MLP) только с одним скрытым слоем, состоящим из 100 элементов. Нам пришлось оптимизировать примерно 80 тыс. весовых параметров ($[784 \times 100 + 100] + [100 \times 10] + 10 = 79\,510$) для очень простой задачи классификации изображений. Изображения в MNIST довольно маленькие (28×28 пикселов), но мы можем только представить себе взрывной рост количества параметров, если захотим добавить дополнительные скрытые слои или работать с изображениями с более высокой плотностью пикселов. Такая задача быстро станет для одного процессора невыполнимой. Возникает вопрос: как более эффективно решать подобные задачи?

Очевидное решение этой проблемы — задействовать *графические процессоры* (Graphics Processing Unit, GPU), которые стали настоящими рабочими лошадками машинного обучения. Видеокарту с таким процессором можно рассматривать как небольшой процессорный кластер внутри вашего компьютера. Еще одним преимуществом такого подхода являются намного более выдающиеся характеристики графических процессоров по сравнению с обычными современными центральными процессорами (рис. 12.1).

Технические характеристики	Intel® Core™ i9-11900KB Processor	NVIDIA GeForce® RTX™ 3080 Ti
Базовая тактовая частота	3,3 ГГц	1,37 ГГц
Количество ядер	16 (32 потока)	10240
Пропускная способность памяти	45,8 Гбит/с	912,1 Гбит/с
Вычисления с плавающей запятой	742 GFLOPS	34,10 TFLOPS
Цена	~\$540,00	~\$1200,00

Рис. 12.1. Сравнение характеристик обычного центрального и графического процессоров

Источниками информации, приведенной на рис. 12.1, являются следующие веб-сайты (дата обращения: июль 2021 г.):

- ◆ <https://ark.intel.com/content/www/us/en/ark/products/215570/intel-core-i9-11900kb-processor-24m-cache-up-to-4-90-ghz.html>;
- ◆ <https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080-3080ti/>.

По цене в 2,2 раза выше, чем у обычного современного центрального процессора, мы можем получить GPU, который имеет в 640 раз больше ядер и выполняет примерно в 46 раз больше вычислений с плавающей запятой в секунду. Что удерживает нас от

повсеместного использования графических процессоров для задач машинного обучения? Проблема в том, что написать специальный код для графических процессоров намного труднее, чем выполнить код Python в интерпретаторе. Существуют специальные пакеты — такие как CUDA и OpenCL, предназначенные для выполнения на GPU. Однако написание кода на CUDA или OpenCL тоже требует специальных навыков, и, вероятно, это не самый удобный способ реализации и запуска алгоритмов машинного обучения. Хорошая новость заключается в том, что именно для этого была разработана библиотека PyTorch!

12.1.2. Что такое PyTorch?

PyTorch — это масштабируемый многоплатформенный программный интерфейс для реализации и запуска алгоритмов машинного обучения, включая удобные оболочки для глубокого обучения. Открытый и бесплатный код PyTorch впервые был опубликован в сентябре 2016 года под модифицированной лицензией BSD. Многие исследователи машинного обучения и специалисты-практики из научных кругов и промышленности адаптировали PyTorch для разработки прикладных решений глубокого обучения — таких как Tesla Autopilot, Uber Pyro и Hugging Face’s Transformers (<https://pytorch.org/ecosystem/>).

В целях ускорения процесса обучения моделей PyTorch позволяет выполнять его на обычных центральных процессорах, графических процессорах и устройствах XLA — таких как тензорные процессоры (TPU). Однако наибольшего быстродействия удается достичь при использовании графических процессоров и устройств XLA. PyTorch официально поддерживает графические процессоры, работающие с CUDA и ROCm. За основу PyTorch взяли библиотеку Torch (www.torch.ch). Как следует из названия PyTorch, интерфейс Python является основным направлением в разработке PyTorch.

Архитектура PyTorch основана на графе вычислений, состоящем из набора узлов. Каждый узел представляет собой операцию, которая может иметь ноль или более входов или выходов. PyTorch предоставляет императивную среду программирования, которая оценивает операции, выполняет вычисления и немедленно возвращает конкретные значения. Следовательно, граф вычислений в PyTorch определяется неявно, а не строится заранее с последующим выполнением.

С математической точки зрения *тензоры* можно рассматривать как обобщение скаляров, векторов, матриц и т. п. Точнее, скаляр может быть определен как тензор ранга 0, вектор — как тензор ранга 1, матрица — как тензор ранга 2, а матрицы, сложенные в стек в третьем измерении, могут быть определены как тензоры ранга 3. Тензоры в PyTorch аналогичны массивам NumPy, за исключением того, что тензоры оптимизированы для автоматического дифференцирования и могут работать на графических процессорах.

Чтобы прояснить понятие тензора, рассмотрим рис. 12.2, на котором представлены тензоры рангов 0 и 1 в первой строке и тензоры рангов 2 и 3 во второй.

Теперь, когда вы в общих чертах познакомились с библиотекой PyTorch, мы перейдем к примерам ее использования.

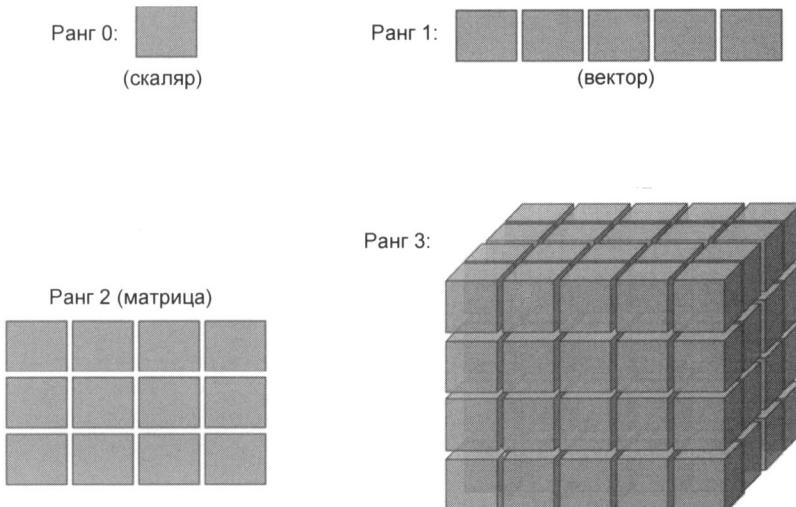


Рис. 12.2. Различные типы тензоров в PyTorch

12.1.3. Как мы будем изучать PyTorch?

Сначала мы рассмотрим модель программирования PyTorch и, в частности, создание тензоров и управление ими. Затем вы узнаете, как загружать данные и использовать модуль `torch.utils.data`, который позволяет эффективно перебирать набор данных. Кроме того, мы обсудим существующие, готовые к использованию наборы данных в подмодуле `torch.utils.data.Dataset` и покажем, как их применять.

Изучив эти основы, мы перейдем к знакомству с модулем нейронной сети PyTorch `torch.nn`, а затем приступим к созданию моделей машинного обучения, научимся составлять и обучать модели, а также сохранять обученные модели на диске для будущей оценки.

12.2. Первые шаги с PyTorch

В этом разделе вы сделаете первые шаги в использовании низкоуровневого API PyTorch. Установив PyTorch, мы узнаем, как создавать тензоры в PyTorch, и рассмотрим различные способы управления ими — такие как изменение их формы, типа данных и т. д.

12.2.1. Установка PyTorch

Перед установкой PyTorch рекомендуется ознакомиться с самыми свежими инструкциями по проведению этого процесса на официальном сайте <https://pytorch.org>. Далее мы опишем основные шаги, которые будут работать на большинстве систем.

В зависимости от настроек вашей системы, обычно достаточно просто использовать установщик `pip` Python и установить PyTorch из PyPI, выполнив в терминале следующую команду:

```
pip install torch torchvision
```

Эта команда установит последнюю стабильную версию (1.9.0 на момент подготовки книги). Чтобы установить именно версию 1.9.0, которая гарантированно совместима с дальнейшими примерами кода, измените предыдущую команду следующим образом:

```
pip install torch==1.9.0 torchvision==0.10.0
```

Если вы хотите использовать графические процессоры (что настоятельно рекомендуется), вам потребуется компьютер, оснащенный видеокартой NVIDIA с поддержкой CUDA и cuDNN. Если ваш компьютер удовлетворяет этому требованию, вы можете установить PyTorch с поддержкой графического процессора следующим образом:

◆ для CUDA 11.1:

```
pip install torch==1.9.0+cu111 torchvision==0.10.0+cu111 -f  
https://download.pytorch.org/whl/torch_stable.html
```

◆ или для CUDA 10.2 (на момент подготовки книги):

```
pip install torch==1.9.0 torchvision==0.10.0\ -f  
https://download.pytorch.org/whl/torch_stable.html
```

Поскольку двоичные исполняемые файлы macOS не поддерживают CUDA, вы можете установить их из исходного кода: <https://pytorch.org/get-started/locally/#mac-from-source>.

Для получения дополнительной информации о процессе установки и настройки PyTorch изучите официальные рекомендации по адресу: <https://pytorch.org/get-started/locally/>.

Не забывайте, что PyTorch находится в активной разработке, поэтому каждые пару месяцев выходят новые ее версии с существенными изменениями. Чтобы узнать свою версию PyTorch, введите в окне терминала следующую команду:

```
python -c 'import torch; print(torch.__version__)'
```



Устранение неполадок при установке PyTorch

Если у вас возникли проблемы с процедурой установки, ознакомьтесь с рекомендациями для конкретных систем и платформ, которые приведены на странице: <https://pytorch.org/get-started/locally/>. Обратите также внимание, что весь код, приведенный в этой главе, может быть запущен на обычном CPU. При этом использование графического процессора совершенно необязательно, но рекомендуется, если вы хотите в полной мере насладиться преимуществами PyTorch. Например, в то время как обучение некоторых нейросетевых моделей на CPU может занять неделю, те же самые модели можно обучить на современном графическом процессоре всего за несколько часов. Если у вас есть соответствующая видеокарта, ознакомьтесь с фирменным руководством, чтобы настроить ее должным образом. Вам также может пригодиться руководство, в котором даны пояснения по установке драйвера видеокарты NVIDIA, CUDA и cuDNN в Ubuntu (не обязательные, но рекомендуемые требования для запуска PyTorch на графическом процессоре): https://sebastianraschka.com/pdf/books/dlb/appendix_h_cloud-computing.pdf. Кроме того, как будет показано в главе 17, вы также можете бесплатно обучать свои модели с помощью графического процессора через Google Colab.

12.2.2. Создание тензоров в PyTorch

В этом разделе мы изучим несколько различных способов создания тензоров, а затем рассмотрим некоторые их свойства и узнаем, как с ними обращаться. Мы можем создать тензор из простого списка или массива NumPy, используя функцию `torch.tensor` или `torch.from_numpy`:

```
>>> import torch
>>> import numpy as np
>>> np.set_printoptions(precision=3)
>>> a = [1, 2, 3]
>>> b = np.array([4, 5, 6], dtype=np.int32)
>>> t_a = torch.tensor(a)
>>> t_b = torch.from_numpy(b)
>>> print(t_a)
>>> print(t_b)
tensor([1, 2, 3])
tensor([4, 5, 6], dtype=torch.int32)
```

После запуска этого кода будут созданы тензоры `t_a` и `t_b` с их свойствами `shape=(3,)` и `dtype=int32` в соответствии с источником данных тензора. Как и в случае массивов NumPy, мы можем посмотреть эти свойства:

```
>>> t_ones = torch.ones(2, 3)
>>> t_ones.shape
torch.Size([2, 3])
>>> print(t_ones)
tensor([[1., 1., 1.],
       [1., 1., 1.]])
```

Наконец, при помощи следующего кода можно создать тензор случайных значений:

```
>>> rand_tensor = torch.rand(2,3)
>>> print(rand_tensor)
tensor([[0.1409, 0.2848, 0.8914],
       [0.9223, 0.2924, 0.7889]])
```

12.2.3. Управление типом данных и формой тензора

Изучение способов управления тензорами необходимо, чтобы сделать их подходящими для ввода в модель или в операцию. В этом разделе вы узнаете, как управлять типами данных и формами тензоров с помощью нескольких функций PyTorch, которые выполняют операции преобразования, транспонирования и сжатия (удаления измерений).

Функцию `torch.to()` можно использовать для изменения типа данных тензора на желаемый тип:

```
>>> t_a_new = t_a.to(torch.int64)
>>> print(t_a_new.dtype)
torch.int64
```

По адресу: https://pytorch.org/docs/stable/tensor_attributes.html доступны инструкции для всех других типов данных.

Как вы увидите в следующих главах, некоторые операции требуют, чтобы входные тензоры имели определенное количество измерений (т. е. ранг), связанных с определенным количеством элементов (форма). Поэтому часто возникает необходимость изменить форму тензора, добавить новое измерение или удалить лишнее. Для этих целей PyTorch предоставляет полезные функции (или операции) — такие как `torch.transpose()`, `torch.reshape()` и `torch.squeeze()`. Давайте рассмотрим несколько примеров:

◆ транспонирование тензора:

```
>>> t = torch.rand(3, 5)
>>> t_tr = torch.transpose(t, 0, 1)
>>> print(t.shape, '-->', t_tr.shape)
torch.Size([3, 5]) --> torch.Size([5, 3])
```

◆ изменение формы тензора (например, преобразование из одномерного вектора в двумерный массив):

```
>>> t = torch.zeros(30)
>>> t_reshape = t.reshape(5, 6)
>>> print(t_reshape.shape)
torch.Size([5, 6])
```

◆ удаление ненужных измерений (измерения с размером 1, которые не нужны):

```
>>> t = torch.zeros(1, 2, 1, 4, 1)
>>> t_sqz = torch.squeeze(t, 2)
>>> print(t.shape, '-->', t_sqz.shape)
torch.Size([1, 2, 1, 4, 1]) --> torch.Size([1, 2, 4, 1])
```

12.2.4. Применение к тензорам математических операций

При построении моделей машинного обучения не обойтись без математических процедур и, в частности, операций линейной алгебры. В этом разделе мы рассмотрим некоторые широко используемые операции линейной алгебры — такие как поэлементное произведение, умножение матриц и вычисление нормы тензора.

Мы начнем с создания двух случайных тензоров: одного с равномерным распределением в диапазоне $[-1, 1]$, а другого — со стандартным нормальным распределением:

```
>>> torch.manual_seed(1)
>>> t1 = 2 * torch.rand(5, 2) - 1
>>> t2 = torch.normal(mean=0, std=1, size=(5, 2))
```

Метод `torch.rand` возвращает тензор, заполненный случайными числами из равномерного распределения в диапазоне $[0, 1]$.

Обратите внимание, что `t1` и `t2` имеют одинаковую форму. Далее, чтобы вычислить поэлементное произведение `t1` и `t2`, выполним следующий код:

```
>>> t3 = torch.multiply(t1, t2)
>>> print(t3)
tensor([[ 0.4426, -0.3114],
       [ 0.0660, -0.5970],
```

```
[ 1.1249,  0.0150],
[ 0.1569,  0.7107],
[-0.0451, -0.0352]])
```

Для вычисления среднего значения, суммы и стандартного отклонения по определенной оси (или осям) предназначены функции: `torch.mean()`, `torch.sum()` и `torch.std()`. Например, среднее значение каждого столбца в `t1` можно вычислить следующим образом:

```
>>> t4 = torch.mean(t1, axis=0)
>>> print(t4)
tensor([-0.1373,  0.2028])
```

Матричное произведение между `t1` и `t2` (т. е. $t_1 \times t_2^T$, где верхний индекс T означает транспонирование) можно вычислить с помощью функции `torch.matmul()`:

```
>>> t5 = torch.matmul(t1, torch.transpose(t2, 0, 1))
>>> print(t5)
tensor([[ 0.1312,  0.3860, -0.6267, -1.0096, -0.2943],
        [ 0.1647, -0.5310,  0.2434,  0.8035,  0.1980],
        [-0.3855, -0.4422,  1.1399,  1.5558,  0.4781],
        [ 0.1822, -0.5771,  0.2585,  0.8676,  0.2132],
        [ 0.0330,  0.1084, -0.1692, -0.2771, -0.0804]])
```

В свою очередь, вычисление произведения $t_1^T \times t_2$ выполняется путем транспонирования `t1`, в результате чего получается массив размером 2×2 :

```
>>> t6 = torch.matmul(torch.transpose(t1, 0, 1), t2)
>>> print(t6)
tensor([[ 1.7453,  0.3392],
        [-1.6038, -0.2180]])
```

Наконец, функция `torch.linalg.norm()` служит для вычисления L^p -нормы тензора. Например, мы можем вычислить L^2 -норму `t1` следующим образом:

```
>>> norm_t1 = torch.linalg.norm(t1, ord=2, dim=1)
>>> print(norm_t1)
tensor([0.6785, 0.5078, 1.1162, 0.5488, 0.1853])
```

Чтобы убедиться, что этот фрагмент кода правильно вычисляет норму L^2 для тензора `t1`, сравните результат со следующей функцией NumPy:

```
np.sqrt(np.sum(np.square(t1.numpy())), axis=1))
```

12.2.5. Разделение, стекирование и конкатенация тензоров

В этом подразделе мы рассмотрим операции PyTorch по разделению тензора на несколько тензоров и, наоборот, стекирование и объединение нескольких тензоров в один.

Предположим, что у нас есть один тензор и нужно разделить его на два или более тензоров. Для этого в PyTorch имеется удобная функция `torch.chunk()`, которая делит входной тензор на список тензоров одинакового размера. Целое число, заданное аргументом `chunks`, определяет количество фрагментов тензора по нужному измерению, заданному

аргументом `dim`. В этом случае общий размер входного тензора по указанному измерению должен быть кратен желаемому количеству фрагментов. В качестве альтернативы можно указать желаемые размеры фрагментов в списке, используя функцию `torch.split()`. Далее представлены примеры кода для обоих этих вариантов:

- ◆ заданное количество фрагментов:

```
>>> torch.manual_seed(1)
>>> t = torch.rand(6)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293, 0.7999])
>>> t_splits = torch.chunk(t, 3)
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279], dtype=float32),
 array([0.403, 0.735], dtype=float32),
 array([0.029, 0.8 ], dtype=float32)]
```

В этом примере тензор размером 6 был разделен на список из трех тензоров, каждый из которых имеет размер 2. Если размер тензора не кратен количеству разбиений, последний фрагмент будет меньше;

- ◆ список размеров различных фрагментов (в качестве альтернативы вместо определения количества фрагментов можно напрямую указать размеры выходных тензоров. Здесь мы разбиваем тензор размером 5 на тензоры размером 3 и 2):

```
>>> torch.manual_seed(1)
>>> t = torch.rand(5)
>>> print(t)
tensor([0.7576, 0.2793, 0.4031, 0.7347, 0.0293])
>>> t_splits = torch.split(t, split_size_or_sections=[3, 2])
>>> [item.numpy() for item in t_splits]
[array([0.758, 0.279, 0.403], dtype=float32),
 array([0.735, 0.029], dtype=float32)]
```

Иногда мы работаем с несколькими тензорами, и нам нужно объединить или сложить их в стек, чтобы создать один тензор. В этом случае пригодятся такие функции PyTorch, как `torch.stack()` и `torch.cat()`. Для примера создадим одномерный тензор `A` размером 3, содержащий единицы, и одномерный тензор в размером 2, содержащий нули, и выполним их конкатенацию (объединение) в одномерный тензор с размером 5:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(2)
>>> C = torch.cat([A, B], axis=0)
>>> print(C)
tensor([1., 1., 1., 0., 0.])
```

Если взять одномерные тензоры `A` и `B`, оба размером 3, то можно выполнить *стекирование* (своего рода укладку их «штабелем»), чтобы сформировать двумерный тензор `S`:

```
>>> A = torch.ones(3)
>>> B = torch.zeros(3)
>>> S = torch.stack([A, B], axis=1)
>>> print(S)
```

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

PyTorch API включает множество операций, которые можно использовать для построения модели, обработки данных и многое другое. Однако рассмотрение каждой функции выходит за рамки этой книги, и мы сосредоточимся лишь на наиболее важных из них. Полный список операций и функций можно найти на странице документации PyTorch по адресу: <https://pytorch.org/docs/stable/index.html>.

12.3. Построение конвейеров ввода в PyTorch

Занимаясь обучением глубокой нейросетевой модели, мы обычно делаем это постепенно, используя итеративный алгоритм оптимизации — такой как стохастический градиентный спуск, что было показано в предыдущих главах.

Как упоминалось в начале этой главы, `torch.nn` — это модуль для построения нейросетевых моделей. В тех случаях, когда обучающий набор данных достаточно мал и умещается в память в виде тензора, мы можем напрямую использовать этот тензор для обучения. Однако в большинстве типовых сценариев набор данных слишком велик, чтобы поместиться в память компьютера, и нам потребуется загружать данные с внешнего запоминающего устройства (например, с жесткого диска или твердотельного накопителя) фрагментами, т. е. пакет за пакетом. (Обратите внимание, что в этой главе мы используем термин «пакет» вместо «мини-пакет», чтобы соответствовать терминологии PyTorch.) Кроме того, иногда необходимо построить конвейер для предварительной обработки входных данных и выполнения определенных преобразований — таких как центрирование среднего значения, масштабирование или добавление шума, чтобы расширить процедуру обучения и предотвратить переобучение.

Было бы довольно неудобно регулярно организовывать цепочку функций предварительной обработки вручную. К счастью, PyTorch предоставляет специальный класс для создания эффективных и удобных конвейеров предварительной обработки. В этом разделе мы представим обзор различных методов создания объектов PyTorch `Dataset` и `DataLoader`, а также реализации загрузки, перетасовки и пакетной обработки данных.

12.3.1. Создание объекта `DataLoader` из существующих тензоров

Если данные уже существуют в виде тензорного объекта, списка Python или массива NumPy, мы можем легко создать загрузчик набора данных с помощью класса `torch.utils.data.DataLoader()`. Он возвращает объект класса `DataLoader`, который можно использовать для перебора отдельных элементов во входном наборе данных. В качестве простого примера рассмотрим следующий код, который создает набор данных из списка значений от 0 до 5:

```
>>> from torch.utils.data import DataLoader
>>> t = torch.arange(6, dtype=torch.float32)
>>> data_loader = DataLoader(t)
```

Мы можем легко перебирать записи набора данных следующим образом:

```
>>> for item in data_loader:
...     print(item)
tensor([0.])
tensor([1.])
tensor([2.])
tensor([3.])
tensor([4.])
tensor([5.])
```

Допустим, нам нужно создать из этого набора данных пакеты размером 3. Воспользуемся аргументом `batch_size` в следующем коде:

```
>>> data_loader = DataLoader(t, batch_size=3, drop_last=False)
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', batch)
batch 1: tensor([0., 1., 2.])
batch 2: tensor([3., 4., 5.])
```

Этот код создает из текущего набора данных два пакета, где первые три элемента перейдут в пакет #1, а остальные элементы — в пакет #2. Необязательный аргумент `drop_last` полезен в случаях, когда количество элементов в тензоре не кратно желаемому размеру пакета. Мы можем удалить последний неполный пакет, установив для `drop_last` значение `True` (по умолчанию для `drop_last` установлено значение `False`).

Мы всегда можем выполнить итеративную обработку набора данных напрямую, но, как вы только что видели, `DataLoader` обеспечивает автоматическую и настраиваемую пакетную обработку набора данных.

12.3.2. Объединение двух тензоров в совместный набор данных

Часто бывает так, что данные хранятся в двух (или более) тензорах. Например, у нас может быть тензор для признаков и тензор для меток. В таких случаях нужно построить набор данных, объединяющий эти тензоры, что позволит извлекать элементы тензоров в кортежах.

Предположим, что у нас есть два тензора: `t_x` и `t_y`. Тензор `t_x` содержит значения признаков, каждое размером 3, а тензор `t_y` хранит метки классов. Для примера мы сначала создадим эти два тензора:

```
>>> torch.manual_seed(1)
>>> t_x = torch.rand([4, 3], dtype=torch.float32)
>>> t_y = torch.arange(4)
```

Теперь мы должны сформировать совместный набор данных из этих двух тензоров. Сперва мы создадим класс `Dataset`:

```
>>> from torch.utils.data import Dataset
>>> class JointDataset(Dataset):
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...     
```

```
...     def __len__(self):
...         return len(self.x)
...
...
...     def __getitem__(self, idx):
...         return self.x[idx], self.y[idx]
```

Пользовательский класс `Dataset` должен содержать методы, которые впоследствии будет использовать загрузчик данных `DataLoader`:

- ◆ `__init__()`: — здесь выполняется вся предварительная работа, такая как чтение существующих массивов, загрузка файла, фильтрация данных и т. д.;
- ◆ `__getitem__()`: — возвращает соответствующий образец по заданному индексу.

Затем мы создаем совместный набор данных `t_x` и `t_y` с помощью пользовательского класса `Dataset`:

```
>>> from torch.utils.data import TensorDataset
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Наконец, мы можем вывести каждую запись объединенного набора данных:

```
>>> for example in joint_dataset:
...     print(' x: ', example[0], ' y: ', example[1])
 x: tensor([0.7576, 0.2793, 0.4031]) y: tensor(0)
 x: tensor([0.7347, 0.0293, 0.7999]) y: tensor(1)
 x: tensor([0.3971, 0.7544, 0.5695]) y: tensor(2)
 x: tensor([0.4388, 0.6387, 0.5247]) y: tensor(3)
```

Если второй набор данных представляет собой помеченный набор данных в виде тензоров, то можно использовать класс `torch.utils.data.TensorDataset`. Иными словами, вместо применения пользовательского класса `JointDataset` мы создаем объединенный набор данных следующим образом:

```
>>> joint_dataset = TensorDataset(t_x, t_y)
```

Имейте в виду, что частым источником ошибок является потеря поэлементного соответствия между исходными объектами (`x`) и метками (`y`). Это случается, например, если два набора данных перемешиваются по отдельности. Однако, как только они объединены в один набор данных, можно безопасно применять все остальные операции.

Если у нас есть набор данных, созданный из списка имен файлов изображений, имеющихся на диске, можно определить функцию для загрузки изображений по именам файлов. Позже в этой главе вы увидите пример применения нескольких преобразований к набору данных.

12.3.3. Перемешивание, группировка и повторение

Как отмечалось в главе 2, при обучении нейросетевой модели с использованием оптимизации методом стохастического градиентного спуска важно подавать обучающие данные в виде случайно перемешанных пакетов. Вы уже знаете, как указать размер пакета с помощью аргумента `batch_size` объекта загрузчика данных. Далее вы узнаете, как перетасовывать и повторять наборы данных (мы продолжим работу с полученным ранее объединенным набором данных).

Для начала создадим перемешивающую версию загрузчика данных из набора `joint_dataset`:

```
>>> torch.manual_seed(1)
>>> data_loader = DataLoader(dataset=joint_dataset, batch_size=2, shuffle=True)
```

Здесь каждый пакет содержит две записи данных (x) и соответствующие метки (y). Затем мы перебираем записи загрузчика данных одну за другой:

```
>>> for i, batch in enumerate(data_loader, 1):
...     print(f'batch {i}:', 'x:', batch[0],
...           '\n          y:', batch[1])
batch 1: x: tensor([[0.4388, 0.6387, 0.5247],
[0.3971, 0.7544, 0.5695]])
y: tensor([3, 2])
batch 2: x: tensor([[0.7576, 0.2793, 0.4031],
[0.7347, 0.0293, 0.7999]])
y: tensor([0, 1])
```

Строки перемешиваются без потери однозначного соответствия между элементами x и y .

Кроме того, при обучении модели в течение нескольких эпох нам нужно перетасовать и перебрать набор данных для желаемого количества эпох. Итак, давайте дважды выполним перебор пакетного набора данных:

```
>>> for epoch in range(2):
>>>     print(f'epoch {epoch+1}')
>>>     for i, batch in enumerate(data_loader, 1):
...         print(f'batch {i}:', 'x:', batch[0],
...               '\n          y:', batch[1])
epoch 1
batch 1: x: tensor([[0.7347, 0.0293, 0.7999],
[0.3971, 0.7544, 0.5695]])
y: tensor([1, 2])
batch 2: x: tensor([[0.4388, 0.6387, 0.5247],
[0.7576, 0.2793, 0.4031]])
y: tensor([3, 0])
epoch 2
batch 1: x: tensor([[0.3971, 0.7544, 0.5695],
[0.7576, 0.2793, 0.4031]])
y: tensor([2, 0])
batch 2: x: tensor([[0.7347, 0.0293, 0.7999],
[0.4388, 0.6387, 0.5247]])
y: tensor([1, 3])
```

Выполнение этого кода дает нам два различных набора пакетов. В первой эпохе первый пакет содержит пару значений [$y=1$, $y=2$], а второй пакет — пару значений [$y=3$, $y=0$]. Во второй эпохе два пакета содержат пары значений [$y=2$, $y=0$] и [$y=1$, $y=3$] соответственно. Для каждой итерации элементы в пакете также перемешиваются.

12.3.4. Создание набора данных из файлов на локальном диске

В этом разделе мы создадим набор данных из файлов изображений, хранящихся на диске. В файловом архиве, сопровождающем книгу, имеется папка `ch12\cat_dog_images\`, в которой содержатся шесть изображений кошек и собак в формате JPEG, используемые в примерах кода этой главы.

Этот небольшой набор данных предназначен для демонстрации того, как обычно проходит создание набора данных из сохраненных файлов. Мы воспользуемся двумя дополнительными модулями: `PIL.Image` — для чтения содержимого файла изображения и `torchvision.transforms` — для декодирования необработанного содержимого и изменения размера изображений.



Модули `PIL.Image` и `torchvision.transforms`

Модули `PIL.Image` и `torchvision.transforms` предоставляют много полезных функций, описание которых выходит за рамки книги. Вы можете самостоятельно ознакомиться с официальной документацией, чтобы узнать больше об этих функциях:

- <https://pillow.readthedocs.io/en/stable/reference/Image.html> — для `PIL.Image`;
- <https://pytorch.org/vision/stable/transforms.html> — для `torchvision.transforms`.

Прежде чем двигаться дальше, давайте посмотрим на содержимое этих файлов, воспользовавшись библиотекой `pathlib` для создания списка файлов изображений:

```
>>> import pathlib
>>> imgdir_path = pathlib.Path('cat_dog_images')
>>> file_list = sorted([str(path) for path in
...     imgdir_path.glob('*.*jpg')])
>>> print(file_list)
['cat_dog_images/dog-03.jpg', 'cat_dog_images/cat-01.jpg', 'cat_dog_images/cat-02.jpg', 'cat_dog_images/cat-03.jpg', 'cat_dog_images/dog-01.jpg', 'cat_dog_images/dog-02.jpg']
```

Затем отобразим на экране эти примеры изображений с помощью `Matplotlib` (рис. 12.3):

```
>>> import matplotlib.pyplot as plt
>>> import os
>>> from PIL import Image
>>> fig = plt.figure(figsize=(10, 5))
>>> for i, file in enumerate(file_list):
...     img = Image.open(file)
...     print('Размеры изображения:', np.array(img).shape)
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(img)
...     ax.set_title(os.path.basename(file), size=15)
>>> plt.tight_layout()
>>> plt.show()
```

Размеры изображения: (900, 1200, 3)

Размеры изображения: (900, 1200, 3)

Размеры изображения: (900, 1200, 3)

Размеры изображения: (900, 742, 3)

Размеры изображения: (800, 1200, 3)

Размеры изображения: (800, 1200, 3)

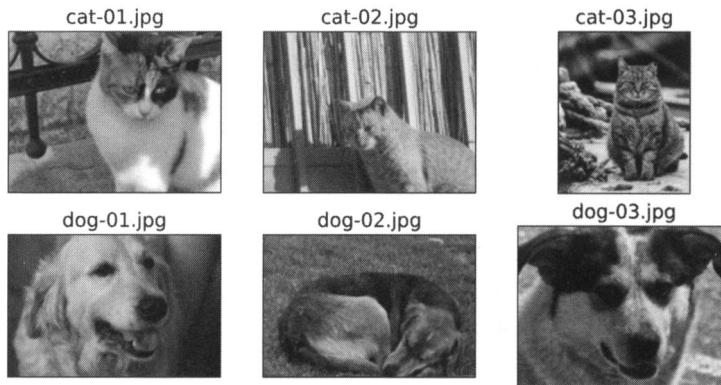


Рис. 12.3. Изображения кошек и собак

Легко заметить, что изображения имеют разные пропорции. Если вы выведете соотношения сторон (или формы массива данных) этих изображений, то увидите, что некоторые изображения имеют 900 пикселов в высоту и 1200 пикселов в ширину (900×1200), некоторые — 800×1200 , а одно — 900×742 . Позже мы выполним предварительную обработку и приведем эти изображения к одинаковому размеру. Еще один момент, который следует учитывать, заключается в том, что метками для этих изображений являются имена их файлов. Поэтому мы извлекаем эти метки из списка имен файлов, присваивая метку 1 собакам и метку 0 кошкам:

```
>>> labels = [1 if 'dog' in
...             os.path.basename(file) else 0
...             for file in file_list]
>>> print(labels)
[0, 0, 0, 1, 1, 1]
```

Теперь у нас есть два списка: список имен файлов (или путей к каждому изображению) и список их меток. В предыдущем разделе вы узнали, как создать совместный набор данных из двух массивов. Здесь мы поступим аналогичным образом:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels):
...         self.file_list = file_list
...         self.labels = labels
...
...     def __getitem__(self, index):
...         file = self.file_list[index]
...         label = self.labels[index]
...         return file, label
...
...     def __len__(self):
...         return len(self.labels)
```

```
>>> image_dataset = ImageDataset(file_list, labels)
>>> for file, label in image_dataset:
...     print(file, label)

cat_dog_images/cat-01.jpg 0
cat_dog_images/cat-02.jpg 0
cat_dog_images/cat-03.jpg 0
cat_dog_images/dog-01.jpg 1
cat_dog_images/dog-02.jpg 1
cat_dog_images/dog-03.jpg 1
```

Объединенный набор данных содержит имена файлов и метки.

Затем к этому набору данных нужно применить следующие операции: загрузить содержимое изображения по его пути к файлу, декодировать необработанное содержимое и изменить его размер до желаемого значения — например, 80×120 . Воспользуемся модулем `torchvision.transforms` для изменения размера изображений и преобразования загруженных пикселов в тензоры:

```
>>> import torchvision.transforms as transforms
>>> img_height, img_width = 80, 120
>>> transform = transforms.Compose([
...     transforms.ToTensor(),
...     transforms.Resize((img_height, img_width)),
... ])
```

Теперь обновим класс `ImageDataset` с помощью только что определенного метода `transform`:

```
>>> class ImageDataset(Dataset):
...     def __init__(self, file_list, labels, transform=None):
...         self.file_list = file_list
...         self.labels = labels
...         self.transform = transform
...
...     def __getitem__(self, index):
...         img = Image.open(self.file_list[index])
...         if self.transform is not None:
...             img = self.transform(img)
...         label = self.labels[index]
...         return img, label
...
...     def __len__(self):
...         return len(self.labels)
>>>
>>> image_dataset = ImageDataset(file_list, labels, transform)
```

Наконец, отобразим преобразованные изображения с помощью `Matplotlib`:

```
>>> fig = plt.figure(figsize=(10, 6))
>>> for i, example in enumerate(image_dataset):
...     ax = fig.add_subplot(2, 3, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
```

```

...
    ax.imshow(example[0].numpy().transpose((1, 2, 0)))
...
    ax.set_title(f'{example[1]}', size=15)
...
>>> plt.tight_layout()
>>> plt.show()

```

После выполнения этого кода на экран будут выведены изображения вместе с их метками (рис. 12.4).

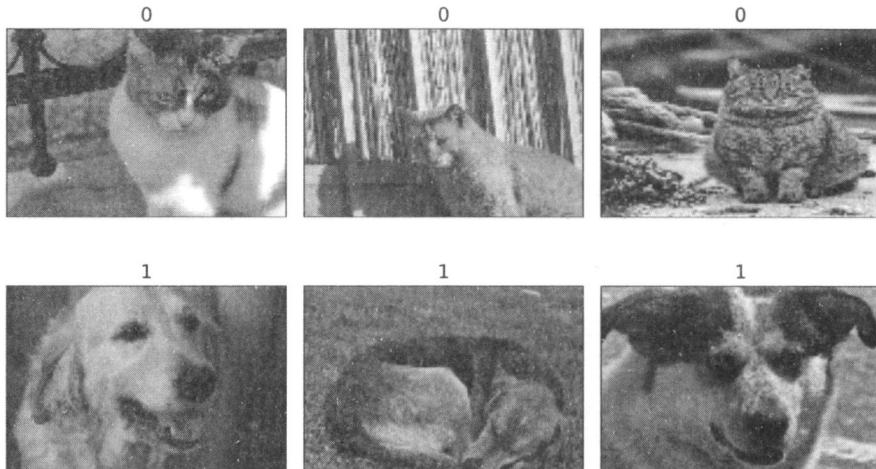


Рис. 12.4. Изображения с метками

Метод `getitem` в классе `ImageDataset` объединяет все четыре шага в одну функцию, включая загрузку необработанного содержимого (изображений и меток), декодирование изображений в тензоры и изменение размера изображений. Затем функция возвращает набор данных, к которому можно применять с помощью загрузчика данных другие операции, известные вам из предыдущих разделов, такие как перемешивание и пакетная обработка.

12.3.5. Получение доступных наборов данных из библиотеки `torchvision.datasets`

Библиотека (модуль) `torchvision.datasets` содержит хорошую коллекцию свободно доступных наборов данных изображений для обучения или оценки моделей глубокого обучения. Аналогичным образом библиотека `torchtext.datasets` предоставляет наборы данных для моделей естественного языка. В этой главе в качестве примера мы используем модуль `torchvision.datasets`.

Наборы данных `torchvision` (<https://pytorch.org/vision/stable/datasets.html>) хорошо отформатированы и снабжены информативными описаниями, включающими формат признаков и меток, их тип и размерность, а также ссылку на оригинальный источник набора данных. Еще одним преимуществом является то, что все эти наборы данных являются подклассами `torch.utils.data.Dataset`, поэтому все функции, рассмотренные в предыдущих разделах, можно применять напрямую. Итак, давайте попробуем использовать эти наборы данных на практике.

Если вы еще не установили библиотеку изображений `torchvision` вместе с PyTorch ранее, вам необходимо установить ее через `pip` из командной строки:

```
pip install torchvision
```

Со списком доступных наборов данных вы можете ознакомиться по адресу: <https://pytorch.org/vision/stable/datasets.html>.

Далее мы рассмотрим получение двух разных наборов данных: `CelebA` (`celeb_a`) и набора цифровых данных `MNIST`.

Давайте сначала поработаем с набором данных `CelebA`¹, представленным в виде объекта `torchvision.datasets.CelebA`². Его описание содержит полезную информацию, которая поможет нам понять структуру этого набора данных:

- ◆ база данных имеет три подмножества: `'train'`, `'valid'` и `'test'`. Мы можем выбрать конкретное подмножество или загрузить их все с параметром `split`;
- ◆ изображения сохраняются в формате `PIL.Image`. Мы можем получить преобразованную версию, используя пользовательскую функцию `transform`, такую как `transforms.ToTensor` и `transforms.Resize`;
- ◆ существуют различные типы целей, которые мы можем использовать, включая `'attributes'`, `'identity'` и `'landmarks'`. Тип `'attributes'` — это 40 атрибутов лица человека на изображении (выражение лица, макияж, свойства волос и т. п.); `'identity'` — это идентификатор человека для изображения, а `'landmarks'` относится к словарю извлеченных точек лица (положение глаз, носа и т. п.).

Затем мы вызываем класс `torchvision.datasets.CelebA` для загрузки данных, сохранения их на диске в указанной папке и загрузки в объект `torch.utils.data.Dataset`:

```
>>> import torchvision
>>> image_path = './'
>>> celeba_dataset = torchvision.datasets.CelebA(
...     image_path, split='train', target_type='attr', download=True
... )
1443490838/? [01:28<00:00, 6730259.81it/s]
26721026/? [00:03<00:00, 8225581.57it/s]
3424458/? [00:00<00:00, 14141274.46it/s]
6082035/? [00:00<00:00, 21695906.49it/s]
12156055/? [00:00<00:00, 12002767.35it/s]
2836386/? [00:00<00:00, 3858079.93it/s]
```

Вы можете столкнуться с ошибкой `BadZipFile: File is not a zip file` (файл не является ZIP-архивом) или `RuntimeError: The daily quota of the file img_align_celeba.zip is exceeded and it can't be downloaded` (дневная квота файла `img_align_celeba.zip` превышена, и он не может быть загружен). Это ограничение Google Drive, и его можно преодолеть, только повторив попытку позже. Дело в том, что у Google Drive есть максимальная дневная квота на скачивание, которую быстро превышают файлы `CelebA`. Чтобы обойти это ограничение, вы можете вручную загрузить файлы из источника: <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>.

¹ См. <http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>.

² См. <https://pytorch.org/vision/stable/datasets.html#celeba>.

mmlab.ie.cuhk.edu.hk/projects/CelebA.html. В загруженной папке celeba\ разархивируйте файл img_align_celeba.zip. Путь image_path в приведенном коде — это корневой каталог загруженной папки, т. е. celeba\ . Если к этому моменту у вас уже есть загруженные файлы, вы можете просто установить: download=False. Мы настоятельно рекомендуем ознакомиться в блокноте Jupyter с сопроводительной документацией кода по адресу: https://github.com/rasbt/machine-learning-book/blob/main/ch12/ch12_part1.ipynb.

Создав экземпляры наборов данных, нужно удостовериться, относится ли объект к классу `torch.utils.data.Dataset`:

```
>>> assert isinstance(celeba_dataset, torch.utils.data.Dataset)
```

Как мы упоминали ранее, набор данных уже разделен на сегменты данных для обучения, тестирования и валидации, и мы загружаем только обучающий набор. Кроме того, мы используем только цель 'attributes'. Чтобы увидеть, как выглядят примеры данных, выполните следующий код:

Обратите внимание, что обучающий пример в этом наборе данных представляет собой кортеж вида (`PIL.Image`, `attributes`). Если мы хотим передать этот набор данных в модель глубокого обучения во время обучения с учителем, нам нужно переформатировать его как кортеж (`features tensor`, `label`). Для метки мы будем использовать в качестве примера категорию 'Smiling' (улыбающиеся), которая является 31-м элементом.

Наконец, возьмем из набора первые 18 примеров и отобразим их на экране вместе со значениями соответствующих меток 'Smiling' (рис. 12.5):

```
>>> from itertools import islice
>>> fig = plt.figure(figsize=(12, 8))
>>> for i, (image, attributes) in islice(enumerate(celeba_dataset), 18):
...     ax = fig.add_subplot(3, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image)
...     ax.set_title(f'{attributes[31]}', size=15)
>>> plt.show()
```

Это все, что нам нужно было сделать, чтобы получить и использовать набор данных изображений CelebA.

Теперь перейдем ко второму набору данных из `torchvision.datasets.MNIST` (<https://pytorch.org/vision/stable/datasets.html#mnist>). Давайте посмотрим, как его можно использовать для получения набора данных MNIST:

- ◆ база данных состоит из двух разделов: 'train' и 'test'. Нужно выбрать конкретное подмножество для загрузки;
 - ◆ изображения сохраняются в формате `PIL.Image`. Можно получить преобразованную версию, используя пользовательскую функцию `transform`, такую как `transforms.ToTensor` и `transforms.Resize`;
 - ◆ существуют 10 классов изображений — от 0 до 9.



Рис. 12.5. Обучающие примеры из набора изображений улыбающихся знаменитостей

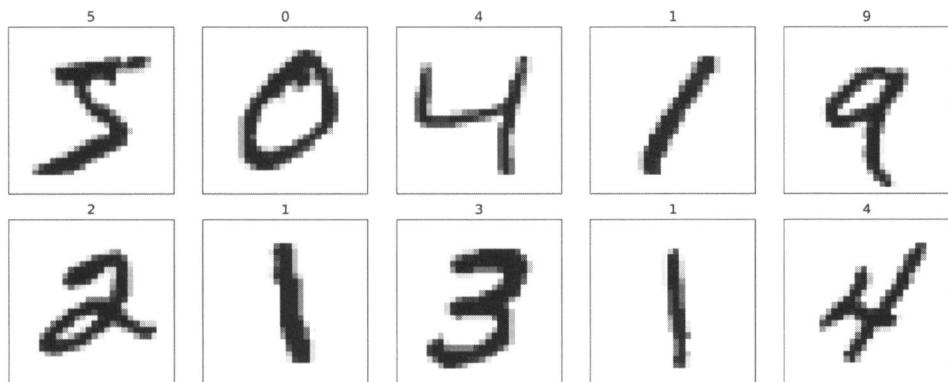


Рис. 12.6. Примеры правильно распознанных рукописных цифр

При помощи следующего кода мы загрузим раздел 'train', преобразуем элементы в кортежи и отобразим 10 извлеченных примеров рукописных цифр из этого набора данных (рис. 12.6):

```
>>> mnist_dataset = torchvision.datasets.MNIST(image_path, 'train',
download=True)
>>> assert isinstance(mnist_dataset, torch.utils.data.Dataset)
>>> example = next(iter(mnist_dataset))
>>> print(example)
(<PIL.Image.Image image mode=L size=28x28 at 0x126895B00>, 5)
>>> fig = plt.figure(figsize=(15, 6))
>>> for i, (image, label) in islice(enumerate(mnist_dataset), 10):
...     ax = fig.add_subplot(2, 5, i+1)
```

```
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(image, cmap='gray_r')
...     ax.set_title(f'{label}', size=15)
>>> plt.show()
```

На этом мы завершаем изучение способов создания наборов данных и управления ими, а также извлечения наборов данных из библиотеки `torchvision.datasets` и переходим к созданию нейросетевых моделей в PyTorch.

12.4. Построение нейросетевой модели в PyTorch

До сих пор в этой главе мы рассказывали об основных служебных компонентах PyTorch, предназначенных для оперирования тензорами и организации данных в форматы, которые применяются во время обучения. В этом разделе мы наконец реализуем нашу первую прогностическую модель в PyTorch. Поскольку фреймворк PyTorch хоть и более гибкий, но и более сложный, чем библиотеки машинного обучения типа scikit-learn, мы начнем с простой модели линейной регрессии.

12.4.1. Модуль нейронной сети PyTorch (`torch.nn`)

Обладающий элегантным дизайном модуль `torch.nn` призван помочь разработчикам создавать и обучать нейронные сети. Он позволяет легко генерировать прототипы и формировать сложные модели всего за несколько строк кода.

Чтобы в полной мере использовать возможности модуля и настроить его для решения вашей задачи, необходимо четко понимать, что он делает. Мы будем двигаться небольшими шагами и сначала обучим базовую модель линейной регрессии на демонстрационном наборе данных без использования каких-либо функций из модуля `torch.nn`, не задействуя ничего, кроме основных тензорных операций PyTorch.

Затем мы начнем постепенно добавлять функции из модулей `torch.nn` и `torch.optim`. Как вы увидите в следующих разделах, эти модули чрезвычайно упрощают построение нейросетевой модели. Мы также воспользуемся преимуществами поддерживаемых в PyTorch функций конвейера наборов данных — таких как `Dataset` и `DataLoader`, о которых шла речь в предыдущем разделе.

Наиболее часто используемый способ построения нейронной сети в PyTorch — через класс `nn.Module`, который позволяет для формирования сети накладывать слои один на другой. Благодаря этому, мы получаем более гибкий контроль над реализацией метода прямого распространения. Примеры построения модели на основе класса `nn.Module` также будут здесь приведены.

Наконец, как вы скоро увидите, обученную модель можно сохранить и повторно загрузить для использования в будущем.

12.4.2. Построение модели линейной регрессии

Сейчас мы построим простую модель для решения задачи линейной регрессии. Сначала создадим в NumPy демонстрационный набор данных и отобразим его на диаграмме распределения обучающих примеров (рис. 12.7):

```
>>> X_train = np.arange(10, dtype='float32').reshape((10, 1))
>>> y_train = np.array([1.0, 1.3, 3.1, 2.0, 5.0,
...                     6.3, 6.6, 7.4, 8.0,
...                     9.0], dtype='float32')
>>> plt.plot(X_train, y_train, 'o', markersize=10)
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

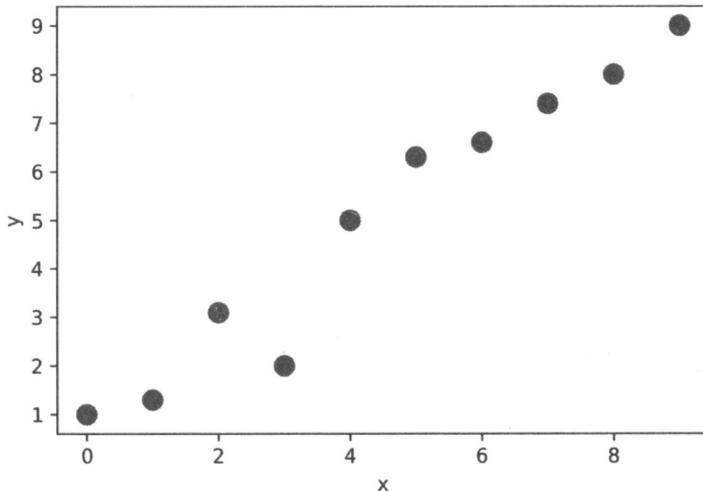


Рис. 12.7. Диаграмма распределения обучающих примеров

Затем мы стандартизируем признаки (центрируем их по среднему и делим на стандартное отклонение) и создаем для обучающего набора объект `Dataset` и соответствующий загрузчик PyTorch с размером пакета 1:

```
>>> from torch.utils.data import TensorDataset
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm)
>>> y_train = torch.from_numpy(y_train).float()
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> batch_size = 1
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Теперь мы можем определить нашу модель линейной регрессии как $z = wx + b$. Модуль `torch.nn`, с которым мы собираемся работать, предоставляет нам предопределенные слои для построения сложных нейросетевых моделей, но сначала все же нужно разобраться, как определить модель с нуля. Поэтому использовать предопределенные слои вы научитесь в этой главе чуть позже.

Итак, сейчас мы построим модель линейной регрессии с нуля: определим параметры веса и смещения нашей модели `weight` и `bias` и функцию `model()`, от которой зависит, как модель использует входные данные для генерации прогноза на выходе:

```
>>> torch.manual_seed(1)
>>> weight = torch.randn(1)
```

```
>>> weight.requires_grad_()
>>> bias = torch.zeros(1, requires_grad=True)
>>> def model(xb):
...     return xb @ weight + bias
```

Далее нам нужно определить функцию потерь, которую следует минимизировать, чтобы найти оптимальные веса модели. В нашем случае мы выберем в качестве нашей функции потерь среднеквадратичную ошибку (MSE):

```
>>> def loss_fn(input, target):
...     return (input-target).pow(2).mean()
```

Кроме того, чтобы найти весовые коэффициенты модели, мы применим стохастический градиентный спуск. В этом разделе мы выполним обучение с помощью процедуры стохастического градиентного спуска самостоятельно, но в дальнейшем будем использовать для этого метод SGD из пакета оптимизации `torch.optim`.

Чтобы реализовать алгоритм стохастического градиентного спуска, нам нужно вычислить градиенты. Мы не станем вычислять градиенты вручную, а воспользуемся для этого функцией PyTorch `torch.autograd.backward`. Сам пакет `torch.autograd` и его различные классы и функции для реализации автоматического дифференцирования мы детально рассмотрим в главе 13.

Теперь мы можем задать скорость обучения и обучить модель на протяжении 200 эпох. Код для обучения модели на пакетной версии набора данных выглядит следующим образом:

```
>>> learning_rate = 0.001
>>> num_epochs = 200
>>> log_epochs = 10
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch.long())
...         loss.backward()
...         with torch.no_grad():
...             weight -= weight.grad * learning_rate
...             bias -= bias.grad * learning_rate
...             weight.grad.zero_()
...             bias.grad.zero_()
...         if epoch % log_epochs==0:
...             print(f'Эпоха {epoch} Потеря {loss.item():.4f}')
```

Эпоха 0 Потеря 5.1701

Эпоха 10 Потеря 30.3370

Эпоха 20 Потеря 26.9436

Эпоха 30 Потеря 0.9315

Эпоха 40 Потеря 3.5942

Эпоха 50 Потеря 5.8960

Эпоха 60 Потеря 3.7567

Эпоха 70 Потеря 1.5877

Эпоха 80 Потеря 0.6213

Эпоха 90 Потеря 1.5596

Эпоха 100 Потеря 0.2583

```
Эпоха 110 Потеря 0.6957
Эпоха 120 Потеря 0.2659
Эпоха 130 Потеря 0.1615
Эпоха 140 Потеря 0.6025
Эпоха 150 Потеря 0.0639
Эпоха 160 Потеря 0.1177
Эпоха 170 Потеря 0.3501
Эпоха 180 Потеря 0.3281
Эпоха 190 Потеря 0.0970
```

Чтобы увидеть обученную регрессионную модель, нам нужно построить ее график. В качестве тестовых данных мы создадим массив значений NumPy, равномерно распределенных между 0 и 9. Поскольку мы обучали нашу модель на стандартизованных признаках, необходимо применить точно такую же стандартизацию к тестовым данным:

```
>>> print('Окончательные параметры:', weight.item(), bias.item())
Окончательные параметры: 2.669806480407715 4.879569053649902
>>> X_test = np.linspace(0, 9, num=100, dtype='float32').reshape(-1, 1)
>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm)
>>> y_pred = model(X_test_norm).detach().numpy()
>>> fig = plt.figure(figsize=(13, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(X_train_norm, y_train, 'o', markersize=10)
>>> plt.plot(X_test_norm, y_pred, '--', lw=3)
>>> plt.legend(['Обучающие примеры', 'Линейная регрессия'], fontsize=15)
>>> ax.set_xlabel('x', size=15)
>>> ax.set_ylabel('y', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()
```

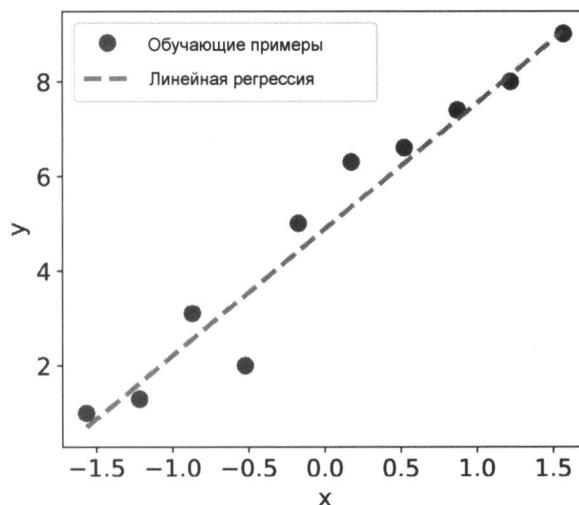


Рис. 12.8. Модель линейной регрессии хорошо подогнана к данным

Диаграмма распределения обучающих примеров и график обученной модели линейной регрессии показаны на рис. 12.8.

12.4.3. Обучение модели с помощью модулей *torch.nn* и *torch.optim*

В предыдущем примере мы рассмотрели обучение модели с помощью самостоятельно разработанной функции потерь *loss_fn()* и оптимизации методом стохастического градиентного спуска. Однако написание функции потерь и обновлений градиента представляет собой рутинную работу, повторяемую от проекта к проекту. Модуль *torch.nn* предоставляет набор готовых функций потерь, а модуль *torch.optim* поддерживает наиболее часто используемые алгоритмы оптимизации, которые можно вызывать для обновления параметров на основе вычисленных градиентов. Чтобы познакомиться с ними на практике, создадим новую функцию потерь *MSE* и оптимизатор стохастического градиентного спуска:

```
>>> import torch.nn as nn
>>> loss_fn = nn.MSELoss(reduction='mean')
>>> input_size = 1
>>> output_size = 1
>>> model = nn.Linear(input_size, output_size)
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Обратите внимание, что в этом коде для создания линейного слоя мы используем класс *torch.nn.Linear*, а не определяем его вручную.

Теперь мы можем просто вызвать метод *step()* класса *optimizer* для обучения модели и передать ему пакетный набор данных (например, *train_dl*, который был создан в предыдущем примере):

```
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         # 1. Генерируем прогнозы
...         pred = model(x_batch)[:, 0]
...         # 2. Вычисляем потери
...         loss = loss_fn(pred, y_batch)
...         # 3. Вычисляем градиенты
...         loss.backward()
...         # 4. Обновляем параметры, используя градиенты
...         optimizer.step()
...         # 5. Обнуляем градиенты
...         optimizer.zero_grad()
...     if epoch % log_epochs==0:
...         print(f'Эпохи {epoch} Потери {loss.item():.4f}')
```

После обучения модели визуализируйте результаты и убедитесь, что они аналогичны результатам предыдущего примера. Чтобы получить параметры веса и смещения, мы можем сделать следующее:

```
>>> print('Окончательные параметры:', model.weight.item(), model.bias.item())
Окончательные параметры: 2.646660089492798 4.883835315704346
```

12.4.4. Построение многослойного персептрана для классификации цветков в наборе данных Iris

Поясняя построение модели с нуля, мы обучили ее, используя стохастическую оптимизацию методом градиентного спуска. И хотя мы начали наше путешествие в мир нейронных сетей с такого простейшего примера, даже его достаточно, чтобы понять, что в определении модели с нуля на практике нет ничего хорошего. Поэтому PyTorch предоставляет нам в модуле `torch.nn` заранее определенные слои, которые можно легко использовать в качестве строительных блоков нейросетевой модели. В этом разделе вы познакомитесь с применением предопределенных слоев для решения задачи классификации цветков ириса (выбор класса между тремя видами ирисов) и построением двухслойного персептрана с помощью модуля `torch.nn`. Сначала получим данные из `sklearn.datasets`:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> iris = load_iris()
>>> X = iris['data']
>>> y = iris['target']
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=1./3, random_state=1)
```

Здесь мы случайным образом выбираем 100 примеров ($\frac{2}{3}$) для обучения и оставляем 50 примеров ($\frac{1}{3}$) для тестирования.

Далее стандартизируем признаки (центрированием по среднему и делением на стандартное отклонение) и создаем для обучающего набора объект `Dataset` и соответствующий `DataLoader` PyTorch, устанавливая для `DataLoader` размер пакета 2:

```
>>> X_train_norm = (X_train - np.mean(X_train)) / np.std(X_train)
>>> X_train_norm = torch.from_numpy(X_train_norm).float()
>>> y_train = torch.from_numpy(y_train)
>>> train_ds = TensorDataset(X_train_norm, y_train)
>>> torch.manual_seed(1)
>>> batch_size = 2
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Теперь мы готовы задействовать модуль `torch.nn` для эффективного построения модели. В частности, используя класс `nn.Module`, мы можем составить стек из нескольких слоев и построить нейронную сеть. Список всех доступных слоев можно получить по адресу: <https://pytorch.org/docs/stable/nn.html>. Для этой задачи мы применим слой `Linear`, который также известен как *полносвязный*, или *плотный*, слой и может быть представлен в виде функции

$$\mathcal{f}(w \times x + b),$$

где x — тензор, содержащий входные признаки; w и b — матрица весов и вектор смещения, \mathcal{f} — функция активации.

Каждый слой нейронной сети получает входные данные от предыдущего слоя, следовательно, его размерность (ранг и форма) постоянна и строго определена. Как правило, нам нужно позаботиться о размерности вывода только при разработке архитектуры нейронной сети. В нашем случае мы определим модель с двумя скрытыми слоями.

Первый получает на вход четыре признака и проецирует их на 16 нейронов. Второй слой получает выходные данные предыдущего слоя (который имеет размер 16) и проецирует их на три выходных нейрона, т. к. у нас есть три метки класса. Это можно сделать при помощи следующего кода:

```
>>> class Model(nn.Module):
...     def __init__(self, input_size, hidden_size, output_size):
...         super().__init__()
...         self.layer1 = nn.Linear(input_size, hidden_size)
...         self.layer2 = nn.Linear(hidden_size, output_size)
...     def forward(self, x):
...         x = self.layer1(x)
...         x = nn.Sigmoid()(x)
...         x = self.layer2(x)
...         return x
>>> input_size = X_train_norm.shape[1]
>>> hidden_size = 16
>>> output_size = 3
>>> model = Model(input_size, hidden_size, output_size)
```

Здесь мы использовали сигмоидную функцию активации для первого слоя и активацию softmax для последнего (выходного) слоя. Активация softmax в последнем слое служит для многоклассовой классификации, поскольку в этой задаче мы имеем три метки класса (именно поэтому у нас три нейрона в выходном слое). Мы обсудим другие функции активации и их применение позже в этой главе.

Далее, для нашей задачи мы выбираем функцию потерь на основе перекрестной энтропии и оптимизатор Adam:

```
>>> learning_rate = 0.001
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```



Оптимизатор Adam — это надежный метод оптимизации на основе градиента, о котором мы подробно поговорим в главе 14.

Теперь можно обучить модель. Установим количество эпох равным 100. Обучение модели классификации цветков реализует следующий код:

```
>>> num_epochs = 100
>>> loss_hist = [0] * num_epochs
>>> accuracy_hist = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist[epoch] += loss.item() * y_batch.size(0)
...         is_correct = (torch.argmax(pred, dim=1) == y_batch).float()
...         accuracy_hist[epoch] += is_correct.sum()
```

```

...     loss_hist[epoch] /= len(train_dl.dataset)
...     accuracy_hist[epoch] /= len(train_dl.dataset)

```

Списки `loss_hist` и `precision_hist` сохраняют потери при обучении и точность модели на обучающих данных после каждой эпохи. Мы можем использовать их для визуализации кривых обучения:

```

>>> fig = plt.figure(figsize=(12, 5))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(loss_hist, lw=3)
>>> ax.set_title('Training loss', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(accuracy_hist, lw=3)
>>> ax.set_title('Training accuracy', size=15)
>>> ax.set_xlabel('Epoch', size=15)
>>> ax.tick_params(axis='both', which='major', labelsize=15)
>>> plt.show()

```

Полученные кривые обучения (потери при обучении и точность при обучении) показаны на рис. 12.9.

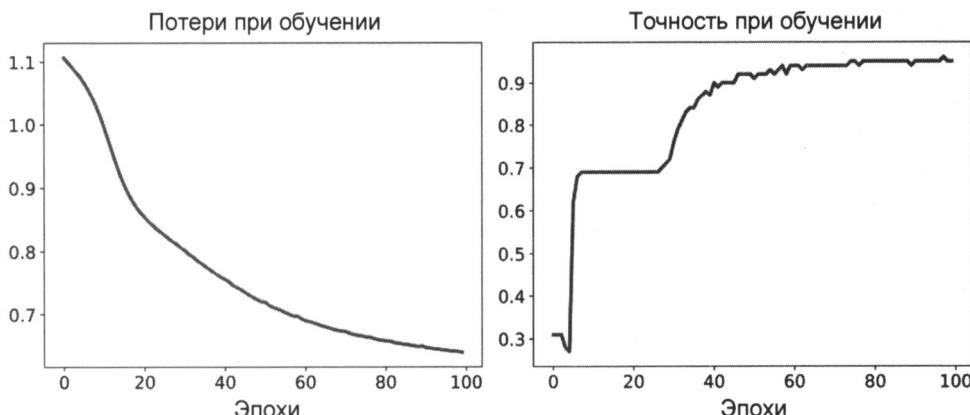


Рис. 12.9. Кривые потерь и точности обучения

12.4.5. Оценка обученной модели на тестовом наборе данных

Оценим точность классификации обученной модели на тестовом наборе данных:

```

>>> X_test_norm = (X_test - np.mean(X_train)) / np.std(X_train)
>>> X_test_norm = torch.from_numpy(X_test_norm).float()
>>> y_test = torch.from_numpy(y_test)
>>> pred_test = model(X_test_norm)
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()
>>> accuracy = correct.mean()

```

```
>>> print(f'Точность на тестовых данных: {accuracy:.4f}')  
Точность на тестовых данных: 0.9800
```

Поскольку мы обучили нашу модель на стандартизованных признаках, необходимо применить точно такую же стандартизацию и к тестовым данным. Точность классификации составляет 0.98 (т. е. 98%).

12.4.6. Сохранение и повторная загрузка обученной модели

Обученные модели можно сохранить для использования в будущем при помощи следующих строк кода:

```
>>> path = 'iris_classifier.pt'  
>>> torch.save(model, path)
```

Вызов функции `save(model)` сохранит как архитектуру модели, так и все обученные параметры. Как правило, модели сохраняют в файлах с расширениями `pt` или `pth`.

Теперь повторно загрузим сохраненную модель. Поскольку мы сохранили и архитектуру, и веса модели, для загрузки достаточно буквально одной строки:

```
>>> model_new = torch.load(path)
```

Проверим архитектуру модели, вызвав `model_new.eval()`:

```
>>> model_new.eval()  
Model(  
    (layer1): Linear(in_features=4, out_features=16, bias=True)  
    (layer2): Linear(in_features=16, out_features=3, bias=True)  
)
```

Наконец, оценим загруженную модель на тестовом наборе данных, чтобы убедиться, что результаты такие же, как и раньше:

```
>>> pred_test = model_new(X_test_norm)  
>>> correct = (torch.argmax(pred_test, dim=1) == y_test).float()  
>>> accuracy = correct.mean()  
>>> print(f'Точность на тестовых данных: {accuracy:.4f}')  
Точность на тестовых данных: 0.9800
```

Если вы хотите сохранить только обученные параметры, используйте `save(model.state_dict())` следующим образом:

```
>>> path = 'iris_classifier_state.pt'  
>>> torch.save(model.state_dict(), path)
```

Чтобы загрузить сохраненные параметры, нам сначала нужно построить модель, как мы это делали раньше, и только потом загрузить параметры из файла в модель:

```
>>> model_new = Model(input_size, hidden_size, output_size)  
>>> model_new.load_state_dict(torch.load(path))
```

12.5. Выбор функций активации для многослойных нейронных сетей

Для простоты мы до сих пор обсуждали сигмоидную функцию активации только в контексте многослойных нейронных сетей прямого распространения — мы использовали ее на скрытом уровне, а также на выходном уровне многослойного персептрона в главе 11.

В этой книге *сигмоидальная логистическая функция* $\sigma(z) = \frac{1}{1 + e^{-z}}$ для краткости называется просто *сигмоидной функцией*, как часто делается в литературе по машинному обучению. В следующих разделах этой главы будет рассказано об альтернативных нелинейных функциях, полезных для реализации многослойных нейронных сетей.

Технически мы можем использовать любую функцию в качестве функции активации в многослойных нейронных сетях, если она дифференцируема. Мы даже можем применять линейные функции активации, такие как в Adaline (см. главу 2). Однако на практике нет смысла действовать линейные функции активации как для скрытых, так и для выходных слоев, поскольку в типичной искусственной нейронной сети для решения сложных задач обязательно нужна нелинейность. Дело в том, что, любая сумма линейных функций дает только линейную функцию.

Логистическая (сигмоидная) функция активации, которую мы использовали в главе 11, вероятно, наиболее точно имитирует концепцию нейрона в мозгу — мы можем рассматривать ее как вероятность срабатывания нейрона. Однако сигмоидную функцию сложно применять, если на вход поступает большое отрицательное значение, поскольку в этом случае выход сигмоидной функции будет близок к нулю. Если сигмоидная функция возвращает результат, близкий к нулю, нейронная сеть будет обучаться очень медленно и с большей вероятностью застрянет в локальном минимуме ландшафта потерь во время обучения. Вот почему в качестве функции активации в скрытых слоях часто используют *гиперболический тангенс* (hyperbolic tangent).

Но прежде чем перейти к знакомству с гиперболическим тангенсом, кратко повторим теоретические основы логистической функции и рассмотрим обобщение, которое делает ее более полезной для задач многоклассовой классификации.

12.5.1. Несколько слов о логистической функции

Как упоминалось во введении к этому разделу, логистическая функция фактически является частным случаем сигмоидной функции. Возможно вы помните из главы 3, что мы можем использовать логистическую функцию для моделирования вероятности принадлежности примера x к положительному классу (классу 1) в задаче бинарной классификации.

Пусть у нас действующий вход z , представленный следующим уравнением:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \sum_{i=0}^m w_i x_i = w^T x.$$

Тогда логистическая (сигмоидная) функция вычисляется так:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}.$$

Заметьте, что w_0 — это смещение (точка пересечения оси y , что означает $x_0 = 1$). В качестве более конкретного примера возьмем модель для двумерной точки данных x , и пусть модель имеет следующие весовые коэффициенты, назначенные вектору w :

```
>>> import numpy as np
>>> X = np.array([1, 1.4, 2.5]) ## первое значение должно быть 1
>>> w = np.array([0.4, 0.3, 0.5])
>>> def net_input(X, w):
...     return np.dot(X, w)
>>> def logistic(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> def logistic_activation(X, w):
...     z = net_input(X, w)
...     return logistic(z)
>>> print(f'P(y=1|x) = {logistic_activation(X, w):.3f}')
P(y=1|x) = 0.888
```

Если мы вычислим действующий вход (z) и используем его для активации логистического нейрона с этими конкретными значениями признаков и весовыми коэффициентами, мы получим значение 0.888, которое можно интерпретировать как принадлежность нашего примера x к положительному классу с вероятностью 88.8%.

В главе 11 мы использовали метод унитарного кодирования для представления эталонных многоклассовых меток и разработали выходной слой, состоящий из нескольких узлов логистической активации. Однако, как показано в следующем примере кода, выходной уровень, состоящий из нескольких узлов логистической активации, не дает осмысленно интерпретируемых значений вероятности:

```
>>> # W : массив с формой = (n_output_units, n_hidden_units+1)
>>> #     первый столбец - это смещение
>>> W = np.array([[1.1, 1.2, 0.8, 0.4],
...                 [0.2, 0.4, 1.0, 0.2],
...                 [0.6, 1.5, 1.2, 0.7]])
>>> # A : массив данных с формой = (n_hidden_units + 1, n_samples)
>>> #     первый столбец массива должен быть 1
>>> A = np.array([[1, 0.1, 0.4, 0.6]])
>>> Z = np.dot(W, A[0])
>>> y_probas = logistic(Z)
>>> print('Вход сети: \n', Z)
Вход сети:
[1.78 0.76 1.65]
>>> print('Значения выходных узлов:\n', y_probas)
Значения выходных узлов:
[ 0.85569687  0.68135373  0.83889105]
```

Как следует из приведенного вывода, полученные значения нельзя интерпретировать как вероятности для задачи с тремя классами. Причина этого в том, что их сумма не равна 1. Однако на самом деле это не представляет большой проблемы, если мы используем нашу модель для прогнозирования только меток классов, а не вероятностей членства в классе. Один из способов предсказать метку класса на основании полученных ранее выходов — использовать максимальное значение:

```
>>> y_class = np.argmax(Z, axis=0)
>>> print('Прогнозируемая метка класса:', y_class)
Прогнозируемая метка класса: 0
```

Тем не менее иногда бывает полезно вычислить интерпретируемые вероятности классов для многоклассовых прогнозов. В следующем разделе мы рассмотрим обобщение логистической функции — функцию `softmax`, которая поможет нам в решении этой задачи.

12.5.2. Оценка вероятностей классов в мультиклассовой классификации с помощью функции `softmax`

В предыдущем разделе вы видели, как можно получить метку класса, используя функцию `argmax`. Функция `softmax` — это мягкая форма функции `argmax`, — вместо того, чтобы выдавать индекс одного класса, она возвращает вероятность каждого класса. Следовательно, это позволяет нам вычислять интерпретируемые вероятности классов в многоклассовых задачах (полиномиальная логистическая регрессия).

В `softmax` вероятность конкретного примера с действующим входом z , принадлежащего i -му классу, может быть вычислена с членом нормализации в знаменателе, т. е. с суммой экспоненциально взвешенных линейных функций:

$$p(z) = \sigma(z) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}}.$$

Чтобы увидеть функцию `softmax` в действии, выполним простой код на Python:

```
>>> def softmax(z):
...     return np.exp(z) / np.sum(np.exp(z))
>>> y_probas = softmax(Z)
>>> print('Вероятности:\n', y_probas)
Вероятности:
[ 0.44668973  0.16107406  0.39223621]
>>> np.sum(y_probas)
1.0
```

Как видите, предсказанные вероятности классов теперь в сумме равны 1, чего и следовало ожидать. Примечательно и то, что прогнозируемая метка класса такая же, как после применения функции `argmax` к логистическому выходу.

Это позволяет нам рассматривать результат работы функции `softmax` как нормализованный вывод, который полезен для получения значимых прогнозов принадлежности к классу в задачах многоклассовой классификации. Поэтому при построении соответствующей модели в PyTorch можно использовать функцию `torch.softmax()` для оценки вероятности принадлежности к классам каждого экземпляра данных из входного пакета. Чтобы продемонстрировать пример функции активации `torch.softmax()` в PyTorch, мы преобразуем в следующем коде z в тензор с дополнительным измерением, зарезервированным для размера пакета:

```
>>> torch.softmax(torch.from_numpy(Z), dim=0)
tensor([0.4467, 0.1611, 0.3922], dtype=torch.float64)
```

12.5.3. Расширение выходного спектра с использованием гиперболического тангенса

Другой сигмоидной функцией, которая часто применяется в скрытых слоях искусственных нейронных сетей, является гиперболический тангенс (часто обозначается \tanh). Его можно рассматривать как повторно нормированную версию логистической функции:

$$\sigma_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma_{\tanh}(z) = 2 \times \sigma_{\text{logistic}}(2z) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

Преимущество гиперболического тангенса перед логистической функцией состоит в том, что он имеет более широкий выходной спектр в открытом интервале $(-1, 1)$, что позволяет улучшить сходимость алгоритма обратного распространения ошибки³.

В отличие от гиперболического тангенса, логистическая функция возвращает выходной сигнал в пределах открытого интервала $(0, 1)$. Для простого сравнения логистической функции и гиперболического тангенса построим график двух этих сигмоидных функций:

```
>>> import matplotlib.pyplot as plt
>>> def tanh(z):
...     e_p = np.exp(z)
...     e_m = np.exp(-z)
...     return (e_p - e_m) / (e_p + e_m)
>>> z = np.arange(-5, 5, 0.005)
>>> log_act = logistic(z)
>>> tanh_act = tanh(z)
>>> plt.ylim([-1.5, 1.5])
>>> plt.xlabel('Действ. вход $z$')
>>> plt.ylabel('Активация $\phi(z)$')
>>> plt.axhline(1, color='black', linestyle=':')
>>> plt.axhline(0.5, color='black', linestyle=':')
>>> plt.axhline(0, color='black', linestyle=':')
>>> plt.axhline(-0.5, color='black', linestyle=':')
>>> plt.axhline(-1, color='black', linestyle=':')
>>> plt.plot(z, tanh_act,
...            linewidth=3, linestyle='--',
...            label='tanh')
>>> plt.plot(z, log_act,
...            linewidth=3,
...            label='logistic')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

Как можно видеть на рис. 12.10, формы двух сигмоидных кривых очень похожи, однако функция гиперболического тангенса имеет вдвое большее выходное пространство, чем логистическая функция.

³ См. «Neural Networks for Pattern Recognition», C. M. Bishop, Oxford University Press, p. 500–501, 1995.

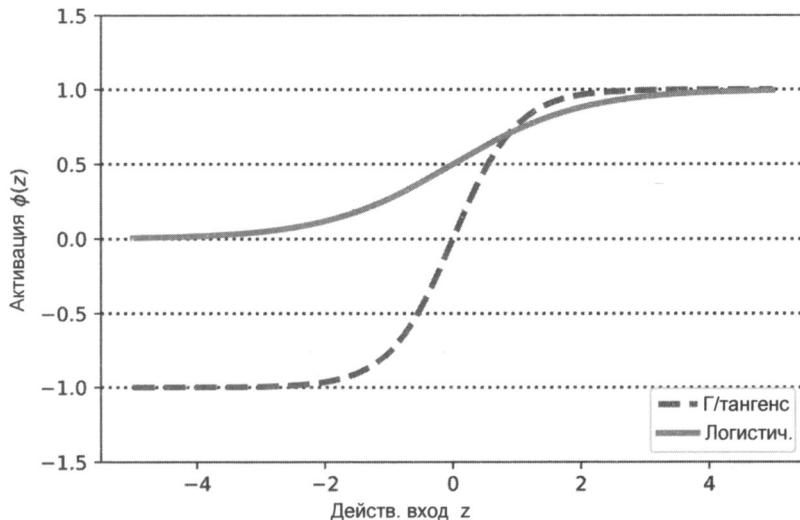


Рис. 12.10. Сравнение тангенциальной и логистической функций

Заметим, что мы подробно реализовали функции `logistic` и `tanh` исключительно для иллюстрации. На практике можно использовать функцию `tanh` библиотеки NumPy.

В качестве альтернативы при построении нейросетевой модели мы можем использовать функцию `torch.tanh(x)` в PyTorch, что дает нам аналогичные результаты:

```
>>> np.tanh(z)
array([-0.9999092 , -0.99990829, -0.99990737, ..., 0.99990644, 0.99990737, 0.99990829])
>>> torch.tanh(torch.from_numpy(z))
tensor([-0.9999, -0.9999, -0.9999, ..., 0.9999, 0.9999, 0.9999],
      dtype=torch.float64)
```

Кроме того, логистическая функция доступна в модуле `special SciPy`:

```
>>> from scipy.special import expit
>>> expit(z)
array([0.00669285, 0.00672617, 0.00675966, ..., 0.99320669, 0.99324034,
       0.99327383])
```

Для выполнения тех же вычислений можно использовать функцию `torch.sigmoid()` в PyTorch:

```
>>> torch.sigmoid(torch.from_numpy(z))
tensor([0.0067, 0.0067, 0.0068, ..., 0.9932, 0.9932, 0.9933],
      dtype=torch.float64)
```



Заметим, что использование `torch.sigmoid(x)` дает результаты, эквивалентные примененному ранее классу `torch.nn.Sigmoid(x)`. То есть `torch.nn.Sigmoid` — это класс, которому вы можете передать параметры для создания объекта, чтобы управлять его поведением. Напротив, `torch.sigmoid` — это функция.

12.5.4. Спрямленная линейная активация

Спрямленная линейная активация (Rectified Linear Unit, ReLU) — еще одна функция активации, часто используемая в глубоких нейронных сетях. Но прежде, чем поближе познакомиться с ReLU, нужно сделать шаг назад и рассмотреть проблему исчезающего градиента тангенциальной и логистических активаций.

Чтобы понять, в чем заключается эта проблема, давайте предположим, что у нас изначально есть действующий вход $z_1 = 20$, который меняется на $z_2 = 25$. Вычисляя активацию \tanh , мы получаем $\sigma(z_1) = 1.0$ и $\sigma(z_2) = 1.0$, что свидетельствует об отсутствии изменений на выходе (из-за асимптотического поведения функции \tanh и погрешности вычислений).

Это означает, что производная активаций по действующему входу уменьшается, когда z становится большим. В результате обучение весов выполняется очень медленно,

Функция активации	Уравнение	Пример применения	График
Линейная	$\sigma(z) = z$	Adaline, линейная регрессия	
Ступенчатая (функция Хевисайда)	$\sigma(z) = \begin{cases} 0 & z < 0 \\ 0.5 & z = 0 \\ 1 & z > 0 \end{cases}$	Вариант персептрона	
Знаковая (signum)	$\sigma(z) = \begin{cases} -1 & z < 0 \\ 0 & z = 0 \\ 1 & z > 0 \end{cases}$	Вариант персептрона	
Кусочно-линейная	$(z) = \begin{cases} 0 & z \leq -\frac{1}{2} \\ z + \frac{1}{2} & -\frac{1}{2} \leq z \leq \frac{1}{2} \\ 1 & z \geq \frac{1}{2} \end{cases}$	Метод опорных векторов	
Логистическая (сигмоида)	$\sigma(z) = \frac{1}{1 + e^{-z}}$	Логистическая регрессия, многослойная нейросеть	
Гиперболический тангенс (tanh)	$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	Многослойные и рекурсивные нейросети	
ReLU	$\sigma(z) = \begin{cases} 0 & z < 0 \\ z & z > 0 \end{cases}$	Многослойные и сверточные нейросети	

Рис. 12.11. Функции активации, описанные в этой книге

поскольку члены градиента близки к нулю. Функция ReLU устраняет эту проблему. Математически ReLU определяется так:

$$\sigma(z) = \max(0, z).$$

ReLU по-прежнему является нелинейной функцией, которая хороша для моделирования сложных функций с помощью нейронных сетей. Кроме того, производная ReLU по входу всегда равна 1 для положительных входных значений. Следовательно, эта функция активации решает проблему исчезающих градиентов, что делает ее хорошим выбором для построения глубоких нейронных сетей. В PyTorch мы можем так применить активацию ReLU `torch.relu()`:

```
>>> torch.relu(torch.from_numpy(z))
tensor([0.0000, 0.0000, 0.0000, ..., 4.9850, 4.9900, 4.9950],
      dtype=torch.float64)
```

В следующей главе мы будем использовать ReLU в качестве функции активации для многослойных сверточных сетей.

Мы прошли большой путь и изучили различные функции активации, которые обычно задействуются в искусственных нейронных сетях. Давайте завершим главу обзором различных функций активации, с которыми мы уже встречались в этой книге (рис. 12.11). А список всех функций активации, доступных в модуле `torch.nn`, можно найти по адресу: <https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions>.

12.6. Заключение

В этой главе вы узнали, как использовать PyTorch — библиотеку с открытым исходным кодом для числовых вычислений, ориентированную на глубокое обучение. Хотя PyTorch труднее использовать, чем NumPy, из-за встроенной поддержки графических процессоров она позволяет очень эффективно строить и обучать большие многослойные нейронные сети.

Мы исследовали создание модели в PyTorch, построив модель с нуля с помощью базовых тензорных функций. Разработка моделей становится утомительной, когда приходится программировать на уровне умножения матрицы на вектор и определять каждую деталь каждой операции. Преимущество PyTorch заключается в том, что она позволяет разработчикам легко выполнять такие базовые операции и строить более сложные модели.

Затем мы изучили модуль `torch.nn`, который значительно упрощает создание нейросетевых моделей по сравнению с реализацией с нуля и обеспечивает создание сложных моделей машинного обучения и нейронных сетей и их эффективное выполнение.

Наконец, вы узнали о различных функциях активации и изучили их поведение и применение. В частности, в этой главе мы рассмотрели `tanh`, `softmax` и `ReLU`.

В следующей главе мы продолжим наше путешествие в мир нейронных сетей и еще глубже погрузимся в PyTorch. Нам предстоит работать с графиками вычислений PyTorch и пакетом автоматического дифференцирования. Попутно вы усвоите много новых концепций — таких как вычисление градиента.

13

Углубленное знакомство с PyTorch

В главе 12 вы узнали, как определять тензоры и оперировать ими, а также научились использовать модуль `torch.utils.data` для создания конвейеров обработки входных данных. Затем мы построили и обучили многослойный персептрон для классификации набора данных Iris с использованием модуля нейронной сети PyTorch `torch.nn`.

Теперь, когда вы приобрели практический опыт обучения нейронной сети с помощью PyTorch, пришло время поближе познакомиться с этой библиотекой и изучить богатый набор функций, которыми мы будем пользоваться для реализации более продвинутых моделей глубокого обучения в последующих главах.

В этой главе для реализации нейронной сети мы будем использовать различные возможности API PyTorch. В частности, мы снова обратимся к возможностям модуля `torch.nn`, предоставляющего несколько уровней абстракции, что делает реализацию стандартных архитектур очень простой и удобной. Этот модуль также позволяет нам создавать пользовательские слои нейросети, что очень полезно в исследовательских проектах, требующих дополнительной настройки. Позже в этой главе мы организуем такой пользовательский слой.

Чтобы проиллюстрировать различные способы построения модели с помощью модуля `torch.nn`, мы рассмотрим классическую задачу *исключающего ИЛИ* (XOR). Сначала мы будем строить многослойные персептроны, используя класс `Sequential`. Затем перейдем и к другим методам — таким как создание подкласса `nn.Module` для определения пользовательских слоев. Наконец, мы поработаем над двумя реальными проектами, которые охватывают этапы машинного обучения от ввода данных до прогнозирования.

Таким образом, в этой главе мы рассмотрим следующие темы:

- ◆ работу с графиками вычислений в PyTorch;
- ◆ работу с тензорными объектами PyTorch;
- ◆ решение классической задачи XOR и понятие емкости модели;
- ◆ построение сложных нейросетевых моделей с использованием классов PyTorch `Sequential` и `nn.Module`;
- ◆ вычисление градиентов с использованием автоматического дифференцирования и `torch.autograd`.

13.1. Основные возможности PyTorch

Как было сказано в предыдущей главе, PyTorch предоставляет нам масштабируемый многоплатформенный программный интерфейс для реализации и запуска алгоритмов машинного обучения. После первого выпуска в 2016 г. и выхода версии 1.0 в 2018 г. PyTorch стал одним из двух самых популярных фреймворков для глубокого обучения. В библиотеке PyTorch применяются динамические графы вычислений, преимущество которых заключается в том, что они более гибкие по сравнению с их статическими аналогами. Динамические графы вычислений удобны для отладки — PyTorch позволяет чередовать этапы объявления графа и его оценки. Вы можете выполнять код построчно, имея при этом полный доступ ко всем переменным. Это очень важная особенность, которая делает разработку и обучение нейронной сети очень удобными.

PyTorch — полностью бесплатная библиотека с открытым исходным кодом. В ее разработке принимает участие большая команда инженеров-программистов, которые постоянно расширяют и улучшают эту библиотеку. Благодаря открытому исходному коду она также пользуется мощной поддержкой со стороны сообщества заинтересованных сторонних разработчиков, которые активно вносят в нее свой посильный вклад. Это сделало библиотеку PyTorch полезной как для академических исследователей, так и для разработчиков производственных решений. Еще одним следствием указанных обстоятельств стало наличие обширной документации и учебных пособий, которые помогают новым пользователям PyTorch.

Важной ключевой особенностью PyTorch, которая также была отмечена в предыдущей главе, является ее способность работать с одним или несколькими графическими процессорами (GPU). Это позволяет пользователям очень эффективно обучать модели глубокого обучения на больших наборах данных в крупномасштабных системах.

И последнее, но не менее важное — PyTorch поддерживает мобильное развертывание, что также делает ее очень подходящим инструментом для производства.

В следующем разделе мы рассмотрим, как тензоры и функции в PyTorch связаны между собой посредством графа вычислений.

13.2. Граф вычислений PyTorch

PyTorch выполняет свои вычисления с использованием *направленного ациклического графа* (Directed Acyclic Graph, DAG). В этом разделе вы узнаете, как определяют графы для простых арифметических вычислений. Затем мы рассмотрим парадигму динамического графа, а также создание графов на лету в PyTorch.

13.2.1. Вкратце о графике вычислений

Вся работа PyTorch основана на построении *графа вычислений*, который служит для получения взаимосвязей между тензорами от входа до вывода. Допустим, у нас есть тензоры ранга 0 (скаляры) a , b и c , и мы хотим найти $z = 2 \times (a - b) + c$.

Это математическое выражение можно представить в виде графа вычислений (рис. 13.1).

Реализация уравнения $z = 2 \times (a - b) + c$
в виде графа вычислений

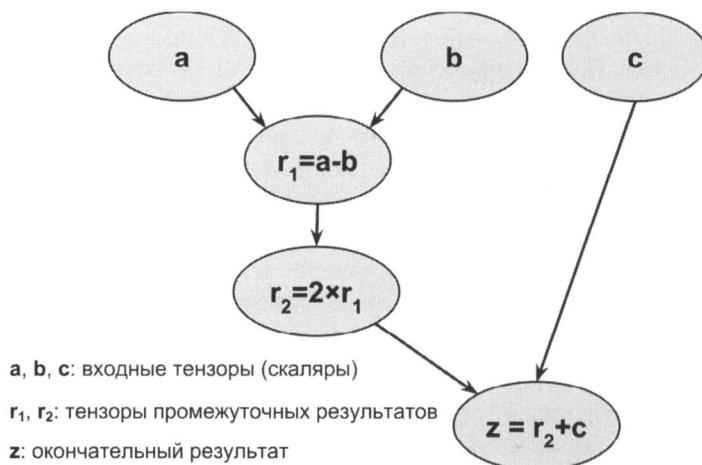


Рис. 13.1. Работа графа вычислений

Как видите, график вычислений — это просто сеть узлов. Каждый узел похож на операцию, которая применяет функцию к своему входному тензору или тензорам и возвращает ноль или более тензоров в качестве выходных данных. PyTorch строит этот график вычислений и использует его для соответствующего вычисления градиентов. В следующем разделе вы увидите несколько примеров создания графа вычислений с использованием PyTorch.

13.2.2. Создание графа в PyTorch

Давайте рассмотрим простой пример, иллюстрирующий создание графа в PyTorch для вычисления $z = 2 \times (a - b) + c$, как показано на рис. 13.1. Переменные a , b и c — это скаляры (одиночные числа), и мы определяем их как тензоры PyTorch. Для создания графа достаточно определить обычную функцию Python с a , b и c в качестве входных аргументов — например:

```

>>> import torch
>>> def compute_z(a, b, c):
...     r1 = torch.sub(a, b)
...     r2 = torch.mul(r1, 2)
...     z = torch.add(r2, c)
...     return z
  
```

Чтобы выполнить вычисление, нужно вызывать эту функцию с тензорными объектами в качестве аргументов. Заметим, что такие функции PyTorch, как `add` (сложение), `sub` (или `subtract`, вычитание) и `mul` (или `multiply`, умножение), также позволяют нам предоставлять входные данные более высокого ранга в форме тензорного объекта PyTorch. В следующем примере кода мы предоставляем скалярные входные данные (ранг 0), а также входные данные ранга 1 и ранга 2 в виде списков:

```
>>> print('Скалярный вход:', compute_z(torch.tensor(1),
...      torch.tensor(2), torch.tensor(3)))
Скалярный вход: tensor(1)
>>> print('Вход ранга 1:', compute_z(torch.tensor([1]),
...      torch.tensor([2]), torch.tensor([3])))
Вход ранга 1: tensor([1])
>>> print('Вход ранга 2:', compute_z(torch.tensor([[1]]),
...      torch.tensor([[2]]), torch.tensor([[3]])))
Вход ранга 2: tensor([[1]])
```

Как можно видеть, создать граф вычислений в PyTorch достаточно легко. Далее мы рассмотрим тензоры PyTorch, которые можно использовать для хранения и обновления параметров модели.

13.3. Тензорные объекты PyTorch для хранения и обновления параметров модели

Вы уже встречали тензорные объекты в главе 12. В PyTorch специальный тензорный объект, для которого нужно вычислять градиенты, позволяет нам хранить и обновлять параметры наших моделей во время обучения. Такой тензор можно создать, просто присвоив параметру `requires_grad` значение `True` для заданных пользователем начальных значений. Заметим, что на момент подготовки книги (середина 2021 г.) только тензоры с плавающей запятой и комплексный тип `dtype` могли требовать вычисления градиентов. Следующий код генерирует тензорные объекты типа `float32`:

```
>>> a = torch.tensor(3.14, requires_grad=True)
>>> print(a)
tensor(3.1400, requires_grad=True)
>>> b = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
>>> print(b)
tensor([1., 2., 3.], requires_grad=True)
```

Обратите внимание, что по умолчанию для параметра `requires_grad` задано значение `False`. Это значение можно установить в `True` при помощи метода `requires_grad_()`.



`method_()` — это прямой метод в PyTorch, который используется для операций без копирования входных данных.

Теперь рассмотрим следующий пример:

```
>>> w = torch.tensor([1.0, 2.0, 3.0])
>>> print(w.requires_grad)
False
>>> w.requires_grad_()
>>> print(w.requires_grad)
True
```

Напомним, что нейросетевым моделям инициализация параметров случайными значениями весовых коэффициентов необходима для нарушения симметрии во время обратного распространения: в противном случае многослойная нейросеть была бы не полез-

нее однослойной — такой как логистическая регрессия. При создании тензора PyTorch тоже можно использовать схему случайной инициализации. PyTorch генерирует случайные числа на основе различных распределений вероятностей¹. В следующем примере мы рассмотрим некоторые стандартные методы инициализации, которые также доступны в модуле `torch.nn.init`².

Рассмотрим создание тензора с инициализацией Глорота — классической схемой случайной инициализации, предложенной Ксавьером Глоротом и Йошуа Бенжио. Для этого сначала создадим пустой тензор и оператор с именем `init` как объект класса `GlorotNormal`. Затем заполним этот тензор значениями в соответствии с инициализацией Глорота, вызывая метод `xavier_normal_()`. В следующем примере мы инициализируем тензор формы 2×3 :

```
>>> import torch.nn as nn
>>> torch.manual_seed(1)
>>> w = torch.empty(2, 3)
>>> nn.init.xavier_normal_(w)
>>> print(w)
tensor([[ 0.4183,  0.1688,  0.0390],
       [ 0.3930, -0.2858, -0.1051]])
```



Инициализация Глорота

На раннем этапе развития глубокого обучения было замечено, что инициализация начальных весов равномерным или нормальным случайным распределением часто приводит к плохой работе модели во время обучения.

В 2010 г. Глорот и Бенжио исследовали эффект инициализации и предложили новую, более надежную схему инициализации для облегчения обучения глубоких сетей. Общая идея инициализации Глорота состоит в том, чтобы приблизительно сбалансировать дисперсию градиентов на разных слоях. В противном случае некоторые слои могут получить слишком много внимания во время обучения, в то время как другие слои станут отставать.

Согласно исследовательской работе Глорота и Бенжио, если мы хотим инициализировать веса с равномерным распределением, то должны выбрать интервал этого равномерного распределения следующим образом:

$$W \sim \text{Uniform}\left(-\frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}, \frac{\sqrt{6}}{\sqrt{n_{in} + n_{out}}}\right).$$

Здесь n_{in} — количество входных нейронов, умноженное на веса, а n_{out} — количество выходных нейронов, которые передаются в следующий слой. Для инициализации весов гауссовским (нормальным) распределением мы рекомендуем выбрать стандартное отклонение этого гауссовского распределения:

$$\sigma = \frac{\sqrt{2}}{\sqrt{n_{in} + n_{out}}}.$$

PyTorch поддерживает инициализацию Глорота как при равномерном, так и при нормальном распределении весов.

¹ См. <https://pytorch.org/docs/stable/torch.html#random-sampling>

² См. <https://pytorch.org/docs/stable/nn.init.html>.

Для получения дополнительной информации о схеме инициализации Глорота и Бенджио, включая ее математическое обоснование, мы рекомендуем статью «Understanding the difficulty of deep feedforward neural networks», Xavier Glorot and Yoshua Bengio, 2010, которая находится в свободном доступе по адресу: <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf>.

Теперь в качестве более близкого к практике варианта применения определим два объекта `Tensor` внутри базового класса `nn.Module`:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.w1 = torch.empty(2, 3, requires_grad=True)
...         nn.init.xavier_normal_(self.w1)
...         self.w2 = torch.empty(1, 2, requires_grad=True)
...         nn.init.xavier_normal_(self.w2)
```

Затем эти два тензора можно использовать в качестве весов, градиенты которых будут вычисляться с помощью автоматического дифференцирования.

13.4. Вычисление градиентов с помощью автоматического дифференцирования

Как вы уже знаете, оптимизация нейронной сети требует вычисления градиентов потерь по отношению к ее весам. Это необходимо для алгоритмов оптимизации — таких как стохастический градиентный спуск (SGD). Кроме того, у градиентов есть и другие применения — такие как диагностика сети, помогающие выяснить, почему модель дает определенный прогноз для тестового примера. Поэтому имеет смысл более детально рассмотреть вычисление градиента.

13.4.1. Вычисление градиентов потерь по обучаемым переменным

PyTorch поддерживает *автоматическое дифференцирование*, которое можно рассматривать как реализацию цепного правила для вычисления градиентов вложенных функций. Далее для простоты мы будем использовать термин «градиент» для обозначения как частных производных, так и градиентов.



Частные производные и градиенты

Частную производную $\frac{\partial f}{\partial x_1}$ можно рассматривать как скорость изменения много-

мерной функции — т. е. функции с несколькими входами $f(x_1, x_2, \dots)$, по отношению к одному из ее входов (например, x_1). Градиент ∇f функции представляет собой вектор, состоящий из всех частных производных входных данных:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots \right).$$

Когда мы определяем последовательность операций, которые приводят нас к выходным или даже промежуточным тензорам, PyTorch предоставляет контекст для вычисления градиентов этих тензоров по отношению к их зависимым узлам в графе вычислений. Чтобы вычислить такие градиенты, мы можем вызвать метод `backward` из модуля `torch.autograd`. Он рассчитывает сумму градиентов того или иного тензора относительно листовых узлов (конечных узлов) в графе.

Давайте возьмем простой пример, в котором вычислим $z = wx + b$ и определим потерю как квадрат потери между целью y и прогнозом z : $Loss = (y - z)^2$. В более общем случае, когда у нас есть несколько прогнозов и целей, мы вычисляем потери как сумму квадратов ошибок: $Loss = \sum_i (y_i - z_i)^2$. Чтобы реализовать это вычисление в PyTorch, определим параметры модели w и b как переменные (тензоры с атрибутом `requires_grad=True`, установленным в `True`), а входные данные x и y — как тензоры по умолчанию. Мы вычислим тензор потерь и используем его для расчета градиентов параметров модели w и b следующим образом:

```
>>> w = torch.tensor(1.0, requires_grad=True)
>>> b = torch.tensor(0.5, requires_grad=True)
>>> x = torch.tensor([1.4])
>>> y = torch.tensor([2.1])
>>> z = torch.add(torch.mul(w, x), b)
>>> loss = (y-z).pow(2).sum()
>>> loss.backward()
>>> print('dL/dw : ', w.grad)
>>> print('dL/db : ', b.grad)
dL/dw :  tensor(-0.5600)
dL/db :  tensor(-0.4000)
```

Вычисление значения z является прямым проходом в нейронной сети. Мы применили метод `backward` к тензору `loss` для вычисления $\frac{\partial Loss}{\partial w}$ и $\frac{\partial Loss}{\partial b}$. Так как это очень про-

стой пример, мы можем получить $\frac{\partial Loss}{\partial w} = 2x(wx + b - y)$ непосредственно из формулы, чтобы убедиться, что вычисленные градиенты соответствуют результатам, полученным в предыдущем примере кода:

```
>>> # проверка вычисленного градиента
>>> print(2 * x * ((w * x + b) - y))
tensor([-0.5600], grad_fn=<MulBackward0>)
```

А проверку b мы оставляем в качестве упражнения для читателя.

13.4.2. Как работает автоматическое дифференцирование?

Автоматическое дифференцирование представляет собой набор вычислительных методов для нахождения градиентов произвольных арифметических операций. Во время этого процесса градиенты (выраженные в виде последовательности операций) получаются путем накопления градиентов за счет повторных применений цепного правила. Чтобы лучше понять принцип автоматического дифференцирования, рассмотрим по-

следовательность вложенных вычислений $y = f(g(h(x)))$ с входом x и выходом y . Эту последовательность можно разбить на несколько шагов:

1. $u_0 = x$.
2. $u_1 = h(x)$.
3. $u_2 = g(u_1)$.
4. $u_3 = f(u_2) = y$.

Производная $\frac{dy}{dx}$ может быть вычислена двумя разными способами: прямым накоплением, которое начинается с $\frac{du_3}{dx} = \frac{du_3}{du_2} \frac{du_2}{du_0}$, и обратным накоплением, которое начинается с $\frac{dy}{du_0} = \frac{dy}{du_1} \frac{du_1}{du_0}$. PyTorch использует обратное накопление, которое лучше подходит для реализации обратного распространения ошибки.

13.4.3. Состязательные примеры

Вычисление градиентов потерь по отношению ко входному примеру используется для генерации *состязательных примеров* (или *состязательных атак*). В компьютерном зрении состязательные примеры — это примеры, которые генерируются путем добавления небольшого незаметного шума (или возмущений) ко входному примеру, что приводит к их неправильной классификации глубокой нейросетью. Рассмотрение состязательных примеров выходит за рамки книги, но если вас интересует эта тема, вы можете прочитать о ней в статье Кристиана Сегеди³.

13.5. Упрощение реализации популярных архитектур с помощью модуля `torch.nn`

Вы уже видели несколько примеров построения модели нейронной сети прямого распространения (например, многослойного персептрона) и определения последовательности слоев с помощью класса `nn.Module`. Прежде чем приступить к подробному изучению `nn.Module`, давайте кратко рассмотрим другой подход к созданию этих слоев — с помощью `nn.Sequential`.

13.5.1. Реализация моделей на основе `nn.Sequential`

С помощью класса `nn.Sequential`⁴ слои, находящиеся внутри модели, соединяются каскадным образом. В следующем примере мы построим модель с двумя полностью связанными слоями:

³ См. Christian Szegedy et al., «Intriguing properties of neural networks» по адресу: <https://arxiv.org/pdf/1312.6199.pdf>.

⁴ См. <https://pytorch.org/docs/master/generated/torch.nn.Sequential.html#sequential>.

```
>>> model = nn.Sequential(  
... nn.Linear(4, 16),  
... nn.ReLU(),  
... nn.Linear(16, 32),  
... nn.ReLU()  
... )  
>>> model  
Sequential(  
(0): Linear(in_features=4, out_features=16, bias=True)  
(1): ReLU()  
(2): Linear(in_features=16, out_features=32, bias=True)  
(3): ReLU()  
)
```

Мы указали слои и создали экземпляр `model` после передачи слоев классу `nn.Sequential`. Выход первого полносвязного слоя используется как вход для первого слоя `ReLU`. Выход первого слоя `ReLU` становится входом для второго полносвязного слоя. Наконец, выход второго полносвязного слоя служит в качестве входа для второго слоя `ReLU`.

Мы можем дополнительно настроить эти слои, например применяя к параметрам различные функции активации, инициализаторы или методы регуляризации. Исчерпывающий и полный список доступных опций для большинства из этих категорий можно найти в официальной документации:

- ◆ выбор функций активации — <https://pytorch.org/docs/stable/nn.html#non-linear-activations-weighted-sum-nonlinearity>;
- ◆ инициализация параметров слоя через `nn.init` — <https://pytorch.org/docs/stable/nn.init.html>;
- ◆ применение регуляризации L2 к параметрам слоя (для предотвращения переобучения) с помощью параметра `weight_decay` некоторых оптимизаторов в `torch.optim` — <https://pytorch.org/docs/stable/optim.html>;
- ◆ применение регуляризации L1 к параметрам слоя (для предотвращения переобучения) путем добавления штрафного члена L1 к тензору потерь, что мы реализуем далее.

В следующем примере кода мы настроим первый полносвязный слой, указав начальное распределение значений веса. Затем настроим второй полносвязный слой, вычислив штрафной член L1 для матрицы весов:

```
>>> nn.init.xavier_uniform_(model[0].weight)  
>>> l1_weight = 0.01  
>>> l1_penalty = l1_weight * model[2].weight.abs().sum()
```

Здесь мы инициализировали вес первого линейного слоя с помощью инициализации Глорота и вычислили норму L1 веса второго линейного слоя.

Кроме того, мы также можем указать тип оптимизатора и функцию потерь для обучения. Опять же, полный список всех доступных опций можно найти в официальной документации:

- ◆ оптимизация при помощи `torch.optim` — <https://pytorch.org/docs/stable/optim.html#algorithms>;
- ◆ функции потерь — <https://pytorch.org/docs/stable/nn.html#loss-functions>.

13.5.2. Выбор функции потерь

В том, что касается выбора алгоритмов оптимизации, наиболее широко используемыми методами являются SGD и Adam. Выбор функции потерь зависит от задачи — например, для задачи регрессии можно использовать среднеквадратичную ошибку.

Семейство функций потерь по критерию перекрестной энтропии предназначено для задач классификации, которые подробно обсуждаются в главе 14.

Кроме того, вы можете использовать приемы, изученные в предыдущих главах (например, приемы оценки модели из главы 6) в сочетании с соответствующими метриками. Например, достоверность и полнота, точность, площадь под кривой (AUC), а также ложноотрицательные и ложноположительные оценки — вполне подходящие метрики для оценки моделей классификации.

В следующем примере мы воспользуемся оптимизатором SGD и потерей перекрестной энтропии для задачи бинарной классификации:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Далее мы обратимся к более практическому примеру — решению классической задачи классификации XOR, где для построения модели применим класс `nn.Sequential()`. Попутно вы также узнаете о способности модели обрабатывать нелинейные разделяющие границы. Затем мы рассмотрим создание модели с помощью `nn.Module`, что даст нам больше гибкости и контроля над слоями сети.

13.5.3. Решение задачи классификации XOR

Задача классификации XOR (исключающего ИЛИ) — это классическая задача для анализа способности модели построить нелинейную разделяющую границу между двумя классами. Сначала мы создадим демонстрационный набор данных из 200 обучающих примеров с двумя признаками (x_0, x_1) , взятыми из равномерного распределения в интервале $[-1, 1]$. Затем присвоим метку истинности каждому обучающему примеру i в соответствии со следующим правилом:

$$y^{(i)} = \begin{cases} 0 & \text{if } x_0^{(i)} \times x_1^{(i)} < 0 \\ 1 & \text{в ином случае} \end{cases}$$

Мы воспользуемся половиной данных (100 обучающих примеров) для обучения, а оставшейся половиной — для валидации. Следующий код осуществляет генерацию данных и разделение их на обучающие и проверочные примеры:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> torch.manual_seed(1)
>>> np.random.seed(1)
>>> x = np.random.uniform(low=-1, high=1, size=(200, 2))
>>> y = np.ones(len(x))
>>> y[x[:, 0] * x[:, 1] < 0] = 0
>>> n_train = 100
>>> x_train = torch.tensor(x[:n_train, :], dtype=torch.float32)
>>> y_train = torch.tensor(y[:n_train], dtype=torch.float32)
```

```
>>> x_valid = torch.tensor(x[n_train:, :], dtype=torch.float32)
>>> y_valid = torch.tensor(y[n_train:], dtype=torch.float32)
>>> fig = plt.figure(figsize=(6, 6))
>>> plt.plot(x[y==0, 0], x[y==0, 1], 'o', alpha=0.75, markersize=10)
>>> plt.plot(x[y==1, 0], x[y==1, 1], '<', alpha=0.75, markersize=10)
>>> plt.xlabel(r'$x_1$', size=15)
>>> plt.ylabel(r'$x_2$', size=15)
>>> plt.show()
```

В результате выполнения этого кода будет построена диаграмма рассеяния обучающих и проверочных примеров, показанных разными маркерами в зависимости от их меток класса (рис. 13.2).

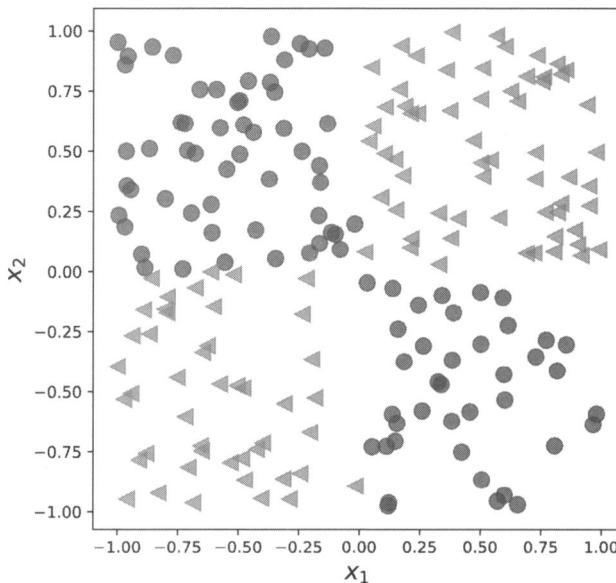


Рис. 13.2. Диаграмма рассеяния обучающих и проверочных примеров

В предыдущем разделе мы упоминали основные инструменты, необходимые для реализации классификатора в PyTorch. Теперь нам нужно решить, какую архитектуру выбрать для этой задачи и набора данных. Как правило, чем больше у нас слоев и чем больше нейронов в каждом слое, тем больше будет *емкость модели* (model capacity). В нашем случае емкость модели можно рассматривать как меру того, насколько легко модель может аппроксимировать сложные функции. Хотя наличие большего количества параметров означает, что сеть может обучиться более сложным функциям, более крупные модели обычно труднее обучаются (и склонны к переобучению). На практике всегда полезно начинать с простой модели в качестве основы — например, с однослойной нейросети, такой как логистическая регрессия:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 1),
...     nn.Sigmoid()
... )
>>> model
```

```
Sequential(  
    (0): Linear(in_features=2, out_features=1, bias=True)  
    (1): Sigmoid()  
)
```

Определив модель, мы инициализируем функцию потерь перекрестной энтропии для бинарной классификации и оптимизатор SGD:

```
>>> loss_fn = nn.BCELoss()  
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Затем создаем загрузчик данных, который использует размер пакета 2 для обучающих данных:

```
>>> from torch.utils.data import DataLoader, TensorDataset  
>>> train_ds = TensorDataset(x_train, y_train)  
>>> batch_size = 2  
>>> torch.manual_seed(1)  
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

Далее обучаем модель на протяжении 200 эпох и записываем историю эпох обучения:

```
>>> torch.manual_seed(1)  
>>> num_epochs = 200  
>>> def train(model, num_epochs, train_dl, x_valid, y_valid):  
...     loss_hist_train = [0] * num_epochs  
...     accuracy_hist_train = [0] * num_epochs  
...     loss_hist_valid = [0] * num_epochs  
...     accuracy_hist_valid = [0] * num_epochs  
...     for epoch in range(num_epochs):  
...         for x_batch, y_batch in train_dl:  
...             pred = model(x_batch)[:, 0]  
...             loss = loss_fn(pred, y_batch)  
...             loss.backward()  
...             optimizer.step()  
...             optimizer.zero_grad()  
...             loss_hist_train[epoch] += loss.item()  
...             is_correct = ((pred>=0.5).float() == y_batch).float()  
...             accuracy_hist_train[epoch] += is_correct.mean()  
...             loss_hist_train[epoch] /= n_train/batch_size  
...             accuracy_hist_train[epoch] /= n_train/batch_size  
...             pred = model(x_valid)[:, 0]  
...             loss = loss_fn(pred, y_valid)  
...             loss_hist_valid[epoch] = loss.item()  
...             is_correct = ((pred>=0.5).float() == y_valid).float()  
...             accuracy_hist_valid[epoch] += is_correct.mean()  
...             return loss_hist_train, loss_hist_valid,  
...                    accuracy_hist_train, accuracy_hist_valid  
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Заметим, что история эпох обучения включает в себя потери при обучении и потери при проверке (валидации), а также точность при обучении и при проверке, что полезно для визуального контроля поведения модели после обучения. Наконец, мы построим

графики эффективности обучения, включающие потери при обучении и валидации, а также соответствующую точность:

```
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Потери при обучении', 'Потери при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Точность при обучении', 'Точность при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
```

После выполнения кода на экран будут выведены две панели — для графиков потерь и точности (рис. 13.3).

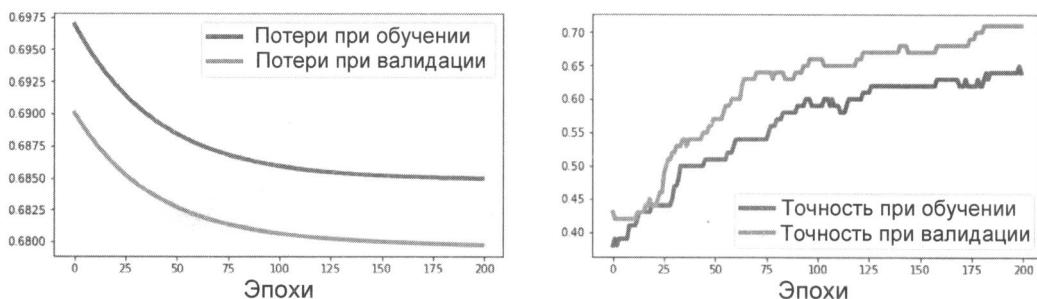


Рис. 13.3. Результаты потерь и точности

Очевидно, что простая модель без скрытого слоя может вывести только линейную границу решения, которая не может решить задачу XOR. Поэтому мы видим, что потери как для обучающих, так и для проверочных наборов данных очень высоки, а точность классификации весьма низкая.

Чтобы получить нелинейную разделяющую границу, нужно добавить один или несколько скрытых слоев, связанных нелинейными функциями активации. Теорема универсальной аппроксимации утверждает, что нейронная сеть прямого распространения с одним скрытым слоем и достаточно большим количеством скрытых элементов может относительно хорошо аппроксимировать произвольные непрерывные функции. Таким образом, один из подходов к более удовлетворительному решению задачи XOR состоит в том, чтобы добавить скрытый слой и сравнивать различное количество скрытых элементов, пока не будут достигнуты удовлетворительные результаты на проверочном наборе. Добавление дополнительных скрытых элементов соответствует увеличению ширины слоя.

В качестве альтернативы можно также добавить больше скрытых слоев, что сделает модель более глубокой. Преимущество создания более глубокой, а не широкой сети заключается в том, что для достижения сравнимой емкости модели требуется меньше параметров.

Однако недостатком глубоких (по сравнению с широкими) моделей является то, что глубокие модели склонны к исчезновению и взрыву градиентов, а это затрудняет их обучение.

В качестве упражнения попробуйте добавить один, два, три и четыре скрытых слоя, каждый из которых содержит четыре скрытых элемента. В следующем примере мы рассмотрим нейросеть прямого распространения с двумя скрытыми слоями:

```
>>> model = nn.Sequential(
...     nn.Linear(2, 4),
...     nn.ReLU(),
...     nn.Linear(4, 4),
...     nn.ReLU(),
...     nn.Linear(4, 1),
...     nn.Sigmoid()
... )
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> model
Sequential(
  (0): Linear(in_features=2, out_features=4, bias=True)
  (1): ReLU()
  (2): Linear(in_features=4, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
  (5): Sigmoid()
)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Воспользуемся готовым кодом для построения графиков потерь и точности из предыдущего примера и получим новые графики (рис. 13.4).

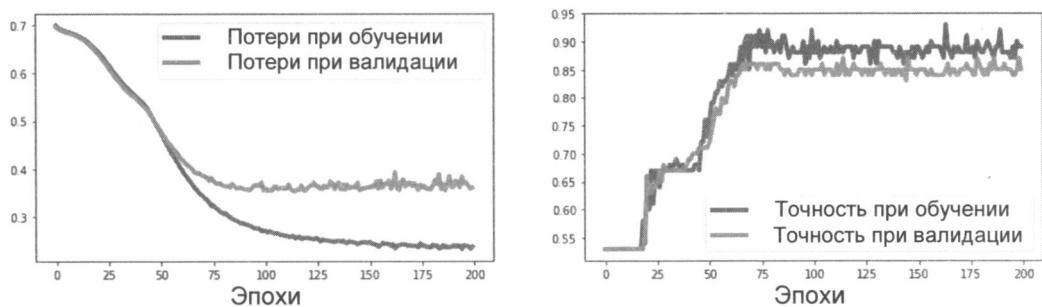


Рис. 13.4. Графики потерь и точности после добавления двух скрытых слоев

Теперь мы видим, что модель может построить нелинейную разделяющую границу для имеющихся данных, и она достигает 100-процентной точности на обучающем наборе. Точность на проверочном наборе данных составляет 95%, что указывает на первые признаки переобучения.

13.5.4. Более гибкое построение моделей с помощью `nn.Module`

В предыдущем примере мы использовали класс PyTorch `Sequential` для создания полно-связной нейросети с несколькими уровнями. Это очень распространенный и удобный способ построения моделей. Однако, к сожалению, он не позволяет нам создавать более сложные модели с несколькими входными, выходными или промежуточными ветвями. Вот где может пригодиться `nn.Module`.

Альтернативный способ построения сложных моделей — создание подкласса `nn.Module`. При этом мы создаем новый класс, производный от `nn.Module`, и определяем метод `__init__()` в качестве конструктора, а метод `forward()` используется для определения прямого прохода. В функции-конструкторе `__init__()` мы определяем слои как атрибуты класса, чтобы к ним можно было получить доступ через атрибут ссылки `self`, а затем в методе `forward()` указываем, как эти слои должны использоваться при прямом проходе нейросети. Далее показан пример кода для определения нового класса, реализующего предыдущую модель:

```
>>> class MyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         l1 = nn.Linear(2, 4)
...         a1 = nn.ReLU()
...         l2 = nn.Linear(4, 4)
...         a2 = nn.ReLU()
...         l3 = nn.Linear(4, 1)
...         a3 = nn.Sigmoid()
...         l = [l1, a1, l2, a2, l3, a3]
...         self.module_list = nn.ModuleList(l)
...
...     def forward(self, x):
...         for f in self.module_list:
...             x = f(x)
...         return x
```

Обратите внимание, что мы поместили все слои в объект `nn.ModuleList`, который представляет собой просто объект `list`, состоящий из элементов `nn.Module`. Это делает код более читаемым и понятным.

Определив экземпляр нового класса, мы можем обучить его, как делали это ранее:

```
>>> model = MyModule()
>>> model
MyModule(
  (module_list): ModuleList(
    (0): Linear(in_features=2, out_features=4, bias=True)
    (1): ReLU()
    (2): Linear(in_features=4, out_features=4, bias=True)
    (3): ReLU()
    (4): Linear(in_features=4, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> history = train(model, num_epochs, train_dl, x_valid, y_valid)
```

Далее, помимо истории обучения, мы воспользуемся библиотекой mlxtend для визуализации проверочных данных и разделяющей границы.

Библиотеку mlxtend можно установить через conda или pip следующим образом:

```
conda install mlxtend -c conda-forge
pip install mlxtend
```

Чтобы вычислить разделяющую границу модели, нам нужно добавить метод predict() в класс MyModule:

```
>>>     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
```

Он возвращает предсказанный класс примера (0 или 1).

Следующий код отображает качество обучения вместе со смещением области принятия решения:

```
>>> from mlxtend.plotting import plot_decision_regions
>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(history[0], lw=4)
>>> plt.plot(history[1], lw=4)
>>> plt.legend(['Потери при обучении', 'Потери при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(history[2], lw=4)
>>> plt.plot(history[3], lw=4)
>>> plt.legend(['Точность при обучении', 'Точность при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(X=x_valid.numpy(),
...                         y=y_valid.numpy().astype(np.integer),
...                         clf=model)
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()
```

После выполнения этого кода будет построен график (рис. 13.5) с тремя отдельными панелями: для потерь, для точности и для диаграммы рассеяния проверочных примеров с разделяющей границей.

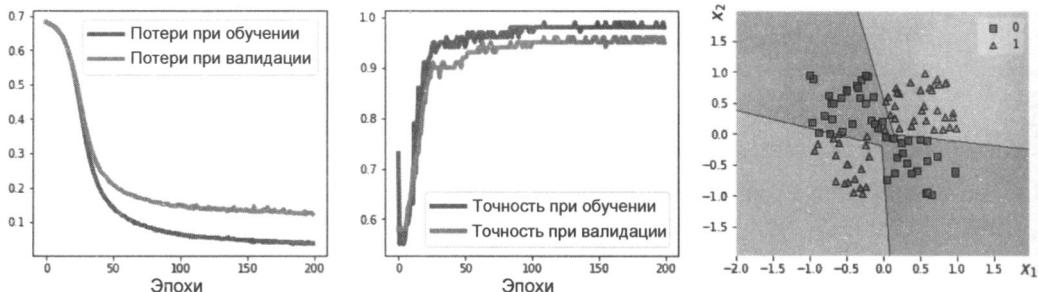


Рис. 13.5. Результаты обучения модели и диаграмма рассеяния

13.5.5. Создание в PyTorch пользовательских слоев

В тех случаях, когда нужно включить в модель новый слой, который еще не поддерживает PyTorch, мы можем определить новый класс, производный от класса `nn.Module`. Это особенно полезно при разработке нового слоя или настройке существующего.

Рассмотрим простой пример создания пользовательского слоя. Допустим, нам нужно определить новый линейный слой, который вычисляет $w(x + \epsilon) + b$, где ϵ представляет собой случайную величину (шумовую переменную). Чтобы реализовать это вычисление, определим новый класс как подкласс `nn.Module`. Для этого нового класса мы должны определить как метод конструктора `__init__()`, так и метод `forward()`. В конструкторе мы определяем переменные и другие необходимые тензоры для нашего пользовательского слоя. Мы можем создавать переменные и инициализировать их в конструкторе, если конструктору передается `input_size`. В качестве альтернативы мы можем отложить инициализацию переменной (например, если мы заранее не знаем точную форму ввода) и делегировать ее другому методу для последующего создания переменной.

Для более конкретного примера определим новый слой с именем `NoisyLinear`, который реализует вычисление приведенного ранее выражения $w(x + \epsilon) + b$:

```
>>> class NoisyLinear(nn.Module):
...     def __init__(self, input_size, output_size,
...                  noise_stddev=0.1):
...         super().__init__()
...         w = torch.Tensor(input_size, output_size)
...         self.w = nn.Parameter(w) # nn.Parameter это Tensor
...                                     # это параметр модуля.
...         nn.init.xavier_uniform_(self.w)
...         b = torch.Tensor(output_size).fill_(0)
...         self.b = nn.Parameter(b)
...         self.noise_stddev = noise_stddev
...
...     def forward(self, x, training=False):
...         if training:
...             noise = torch.normal(0.0, self.noise_stddev, x.shape)
...             x_new = torch.add(x, noise)
```

```

...
    else:
...
        x_new = x
...
    return torch.add(torch.mm(x_new, self.w), self.b)

```

В конструкторе мы добавили аргумент `noise_stddev`, чтобы задать стандартное отклонение для распределения ϵ , которое извлекается из распределения Гаусса. Кроме того, обратите внимание, что в методе `forward()` был задействован дополнительный аргумент `training=False`. Мы сделали это, чтобы указать, используется ли слой во время обучения или только для прогнозирования (иногда это также называют логическим выводом). Также существуют определенные методы, которые по-разному ведут себя в режимах обучения и прогнозирования. В следующих главах вы встретите пример такого метода — `Dropout`. В приведенном фрагменте кода мы также указали, что случайный вектор ϵ должен генерироваться и добавляться ко входным данным только во время обучения и не использоваться для вывода или оценки.

Прежде чем мы сделаем следующий шаг и включим наш пользовательский слой `NoisyLinear` в состав модели, давайте проверим его на простом примере.

1. В следующем коде мы определим новый экземпляр этого слоя и применим его ко входному тензору. Затем мы трижды вызовем слой для одного и того же входного тензора:

```

>>> torch.manual_seed(1)
>>> noisy_layer = NoisyLinear(4, 2)
>>> x = torch.zeros((1, 4))
>>> print(noisy_layer(x, training=True))
tensor([[ 0.1154, -0.0598]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=True))
tensor([[ 0.0432, -0.0375]], grad_fn=<AddBackward0>)
>>> print(noisy_layer(x, training=False))
tensor([[0., 0.]], grad_fn=<AddBackward0>)

```



Обратите внимание, что выходные данные для первых двух вызовов различаются, потому что слой `NoisyLinear` добавил ко входному тензору случайный шум. Третий вызов выводит `[0, 0]`, т. к. мы не добавляли шум, указав `training=False`.

2. Теперь создадим новую модель, аналогичную предыдущей, для решения задачи классификации XOR. Как и прежде, мы будем использовать класс `nn.Module` для построения модели, но на этот раз включим в модель слой `NoisyLinear` в качестве первого скрытого слоя многослойного персептрона:

```

>>> class MyNoisyModule(nn.Module):
...     def __init__(self):
...         super().__init__()
...         self.l1 = NoisyLinear(2, 4, 0.07)
...         self.a1 = nn.ReLU()
...         self.l2 = nn.Linear(4, 4)
...         self.a2 = nn.ReLU()
...         self.l3 = nn.Linear(4, 1)
...         self.a3 = nn.Sigmoid()
...

```

```

...     def forward(self, x, training=False):
...         x = self.l1(x, training)
...         x = self.a1(x)
...         x = self.l2(x)
...         x = self.a2(x)
...         x = self.l3(x)
...         x = self.a3(x)
...         return x
...
...     def predict(self, x):
...         x = torch.tensor(x, dtype=torch.float32)
...         pred = self.forward(x)[:, 0]
...         return (pred>=0.5).float()
...
>>> torch.manual_seed(1)
>>> model = MyNoisyModule()
>>> model
MyNoisyModule(
    (l1): NoisyLinear()
    (a1): RELU()
    (l2): Linear(in_features=4, out_features=4, bias=True)
    (a2): RELU()
    (l3): Linear(in_features=4, out_features=1, bias=True)
    (a3): Sigmoid()
)

```

3. Обучим модель, как и раньше. В рассматриваемом случае, чтобы вычислить прогноз для обучающего пакета, мы используем `pred = model(x_batch, True)[:, 0]` вместо `pred = model(x_batch)[:, 0]`:

```

>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.015)
>>> torch.manual_seed(1)
>>> loss_hist_train = [0] * num_epochs
>>> accuracy_hist_train = [0] * num_epochs
>>> loss_hist_valid = [0] * num_epochs
>>> accuracy_hist_valid = [0] * num_epochs
>>> for epoch in range(num_epochs):
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch, True)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train[epoch] += loss.item()
...         is_correct =
...             (pred>=0.5).float() == y_batch
...             ).float()
...             accuracy_hist_train[epoch] += is_correct.mean()
...         loss_hist_train[epoch] /= n_train/batch_size
...         accuracy_hist_train[epoch] /= n_train/batch_size

```

```

...     pred = model(x_valid)[:, 0]
...     loss = loss_fn(pred, y_valid)
...     loss_hist_valid[epoch] = loss.item()
...     is_correct = ((pred>=0.5).float() == y_valid).float()
...     accuracy_hist_valid[epoch] += is_correct.mean()

```

4. Обучив модель, снова построим графики потери и точности и разделяющую границу:

```

>>> fig = plt.figure(figsize=(16, 4))
>>> ax = fig.add_subplot(1, 3, 1)
>>> plt.plot(loss_hist_train, lw=4)
>>> plt.plot(loss_hist_valid, lw=4)
>>> plt.legend(['Потери при обучении', 'Потери при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 2)
>>> plt.plot(accuracy_hist_train, lw=4)
>>> plt.plot(accuracy_hist_valid, lw=4)
>>> plt.legend(['Точность при обучении', 'Точность при валидации'], fontsize=15)
>>> ax.set_xlabel('Эпохи', size=15)
>>> ax = fig.add_subplot(1, 3, 3)
>>> plot_decision_regions(
...     X=x_valid.numpy(),
...     y=y_valid.numpy().astype(np.integer),
...     clf=model
... )
>>> ax.set_xlabel(r'$x_1$', size=15)
>>> ax.xaxis.set_label_coords(1, -0.025)
>>> ax.set_ylabel(r'$x_2$', size=15)
>>> ax.yaxis.set_label_coords(-0.025, 1)
>>> plt.show()

```

После выполнения этого кода будут построены графики, показанные на рис. 13.6.

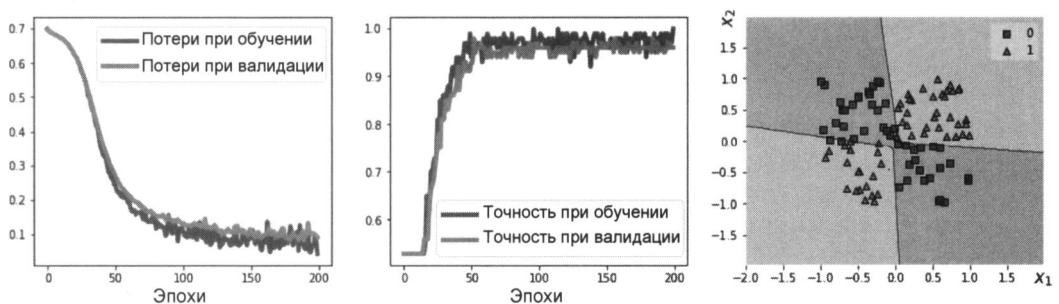


Рис. 13.6. Результаты обучения модели с использованием NoisyLinear в качестве первого скрытого слоя

Этот пример помог нам научиться определять новый пользовательский слой, являющийся подклассом nn.Module, и использовать его так же, как любой другой стандартный слой torch.nn. Хотя в этом конкретном примере NoisyLinear не помог улучшить произво-

дительность модели, наша цель в основном заключалась в том, чтобы научиться создавать пользовательский слой с нуля. В общем случае создание нового слоя может быть полезно, например, если вы разрабатываете новый алгоритм, зависящий от нового слоя, помимо существующих.

13.6. Проект № 1: прогнозирование расхода топлива автомобиля

До сих пор в этой главе мы в основном уделяли внимание модулю `torch.nn`, а для простоты при построении моделей использовали `nn.Sequential`. Затем мы сделали построение моделей более гибким с помощью `nn.Module` и реализовали нейронные сети прямого распространения, к которым добавили настраиваемые слои. Теперь мы приступаем к реальному проекту прогнозирования потребления топлива автомобилем в милях на галлон (Miles Per Gallon, MPG). Мы пройдем при этом через базовые этапы задачи машинного обучения — такие как предварительная обработка данных, конструирование признаков, обучение, прогнозирование (вывод) и оценка.

13.6.1. Работа со столбцами признаков

В приложениях машинного и глубокого обучения мы можем столкнуться с различными типами признаков: непрерывными, неупорядоченными категориальными (номинальными) и упорядоченными категориальными (порядковыми). Как вы помните, в [главе 4](#) мы рассмотрели различные типы признаков, и вы узнали, как обращаться с каждым из них. Хотя числовые данные могут быть как непрерывными, так и дискретными, в контексте машинного обучения с PyTorch термин «числовые данные» относится к непрерывным значениям с плавающей запятой.

Иногда наборы признаков состоят из смеси различных типов. Рассмотрим, например, набор из семи различных признаков (рис. 13.7).

Признаки, показанные здесь (год выпуска, количество цилиндров, литраж (рабочий объем), мощность в лошадиных силах, вес, полоса разгона (ускорение) и страна происхождения), были получены из набора данных Auto MPG — общепринятого эталонного набора данных машинного обучения для прогнозирования потребления топлива в MPG⁵.

Мы будем расценивать пять признаков из набора данных Auto MPG: количество цилиндров (`Cylinders`), рабочий объем (`Displacement`), мощность в ЛС (`Horsepower`), вес (`Weight`) и ускорение (`Acceleration`) как «числовые» (в нашем случае — непрерывные) характеристики. Модельный год (`Model Year`) можно рассматривать как упорядоченный категориальный (порядковый) признак. А страну происхождения (`Origin`) — как неупорядоченный категориальный (номинальный) признак с тремя возможными дискретными значениями: 1, 2 и 3, которые соответствуют США, Европе и Японии соответственно.

Мы начнем с загрузки данных и применения шагов предварительной обработки, включая удаление неполных строк, разделение набора данных на обучающий и тестовый сегменты, а также стандартизацию непрерывных признаков:

⁵ Полный набор данных и его описание доступны в репозитории машинного обучения UCI по адресу: <https://archive.ics.uci.edu/ml/datasets/auto+mpg>.

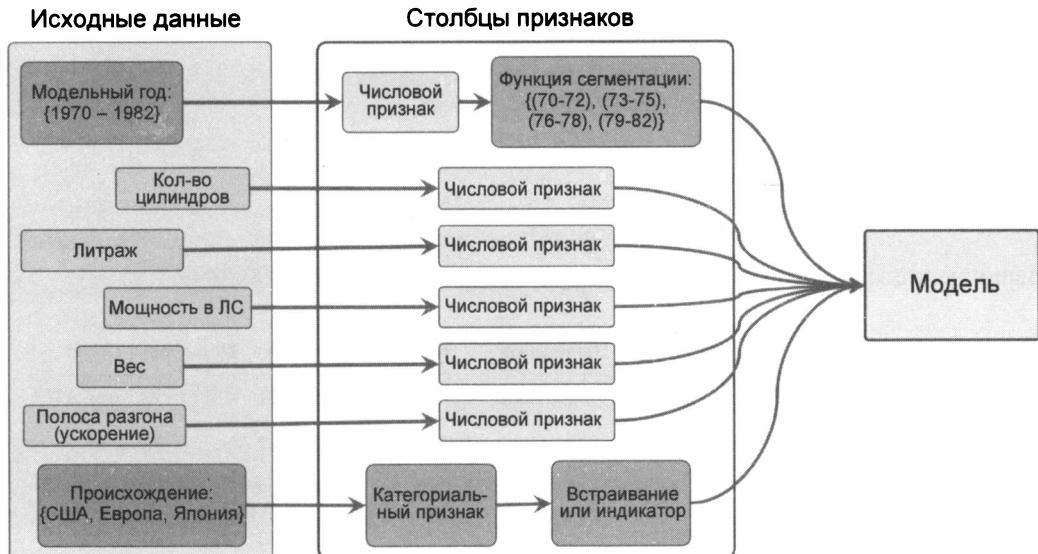


Рис. 13.7. Структура данных Auto MPG

```

>>> import pandas as pd
>>> url = 'http://archive.ics.uci.edu/ml/' \
...      'machine-learning-databases/auto-mpg/auto-mpg.data'
>>> column_names = ['MPG', 'Cylinders', 'Displacement', 'Horsepower',
...                  'Weight', 'Acceleration', 'Model Year', 'Origin']
>>> df = pd.read_csv(url, names=column_names,
...                   na_values = "?", comment='\t',
...                   sep=" ", skipinitialspace=True)
>>>
>>> ## отбрасываем строки со значениями NA
>>> df = df.dropna()
>>> df = df.reset_index(drop=True)
>>>
>>> ## разделение на обучающий и тестовый сегменты:
>>> import sklearn
>>> import sklearn.model_selection
>>> df_train, df_test = sklearn.model_selection.train_test_split(
...     df, train_size=0.8, random_state=1
... )
>>> train_stats = df_train.describe().transpose()
>>>
>>> numeric_column_names = [
...     'Cylinders', 'Displacement',
...     'Horsepower', 'Weight',
...     'Acceleration'
... ]
>>> df_train_norm, df_test_norm = df_train.copy(), df_test.copy()

```

```
>>> for col_name in numeric_column_names:
...     mean = train_stats.loc[col_name, 'mean']
...     std = train_stats.loc[col_name, 'std']
...     df_train_norm.loc[:, col_name] = \
...         (df_train_norm.loc[:, col_name] - mean)/std
...     df_test_norm.loc[:, col_name] = \
...         (df_test_norm.loc[:, col_name] - mean)/std
>>> df_train_norm.tail()
```

Выполнение этого кода дает нам таблицу, показанную на рис. 13.8.

	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	ModelYear	Origin
203	28.0	-0.824303	-0.901020	-0.736562	-0.950031	0.255202	76	3
255	19.4	0.351127	0.413800	-0.340982	0.293190	0.548737	78	1
72	13.0	1.526556	1.144256	0.713897	1.339617	-0.625403	72	1
235	30.5	-0.824303	-0.891280	-1.053025	-1.072585	0.475353	77	1
37	14.0	1.526556	1.563051	1.636916	1.470420	-1.359240	71	1

Рис. 13.8. Предварительно обработанные данные Auto MG

Объект pandas DataFrame, который мы создали с помощью приведенного фрагмента кода, содержит пять столбцов со значениями типа float. Эти столбцы будут представлять собой непрерывные признаки.

Сгруппируем теперь довольно детализированную информацию о модельном году (ModelYear) в сегменты, чтобы упростить задачу обучения модели, которую мы создадим позже. Для этого мы поместим каждый автомобиль в одну из четырех групп (интервалы для групп были выбраны произвольно, чтобы проиллюстрировать подход группировки):

$$\text{bucket} = \begin{cases} 0 & \text{if } \text{year} < 73 \\ 1 & \text{if } 73 \leq \text{year} < 76 \\ 2 & \text{if } 76 \leq \text{year} < 79 \\ 3 & \text{if } \text{year} \geq 79 \end{cases}$$

Чтобы распределить автомобили по группам, сначала определим три предельных значения: [73, 76, 79] для признака модельного года. Эти пороговые значения служат для указания полузамкнутых интервалов — например, $(-\infty, 73]$, $[73, 76)$, $[76, 79)$ и $[76, \infty)$. Затем исходные числовые признаки передаются функции torch.bucketize⁶ для создания индексов сегментов. Группировку выполняет следующий код:

```
>>> boundaries = torch.tensor([73, 76, 79])
>>> v = torch.tensor(df_train_norm['Model Year'].values)
>>> df_train_norm['Model Year Bucketed'] = torch.bucketize(
...     v, boundaries, right=True
... )
```

⁶ См. <https://pytorch.org/docs/stable/generated/torch.bucketize.html>.

```
>>> v = torch.tensor(df_test_norm['Model Year'].values)
>>> df_test_norm['Model Year Bucketed'] = torch.bucketize(
...     v, boundaries, right=True
... )
>>> numeric_column_names.append('Model Year Bucketed')
```

Мы добавили этот столбец с сегментированными признаками в список Python `numeric_column_names`.

Теперь определим список для неупорядоченного категориального признака `Origin`. В PyTorch существуют два способа работы с категориальными признаками: использование слоя встраивания через `nn.Embedding`⁷ или использование вектора с унитарным кодированием (также называемого *индикатором*). Например, при кодировании индекс 0 будет закодирован как [1, 0, 0], индекс 1 — как [0, 1, 0] и т. д. В свою очередь, слой встраивания сопоставляет каждый индекс вектору случайных чисел типа `float`, который можно обучить. (Вы можете рассматривать слой встраивания как более эффективную реализацию унитарного кода, умноженного на обучаемую матрицу весов.)

Когда количество категорий велико, использование слоя встраивания с меньшим количеством размеров, чем количество категорий, может повысить производительность.

В следующем фрагменте кода мы применим метод унитарного кодирования для категориального признака, чтобы преобразовать его в плотный формат:

```
>>> from torch.nn.functional import one_hot
>>> total_origin = len(set(df_train_norm['Origin']))
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_train_norm['Origin'].values) % total_origin)
>>> x_train_numeric = torch.tensor(
...     df_train_norm[numeric_column_names].values)
>>> x_train = torch.cat([x_train_numeric, origin_encoded], 1).float()
>>> origin_encoded = one_hot(torch.from_numpy(
...     df_test_norm['Origin'].values) % total_origin)
>>> x_test_numeric = torch.tensor(
...     df_test_norm[numeric_column_names].values)
>>> x_test = torch.cat([x_test_numeric, origin_encoded], 1).float()
```

После кодирования категориального признака в трехмерный плотный признак мы объединили его с числовыми признаками, обработанными на предыдущем шаге. Наконец, создадим тензоры меток из эталонных значений MPG следующим образом:

```
>>> y_train = torch.tensor(df_train_norm['MPG'].values).float()
>>> y_test = torch.tensor(df_test_norm['MPG'].values).float()
```

Таким образом, в этом разделе мы рассмотрели наиболее распространенные подходы к предварительной обработке данных и созданию признаков в PyTorch.

13.6.2. Обучение регрессионной модели DNN

Закончив создание обязательных признаков и меток, перейдем к созданию загрузчика данных `DataLoader`, который использует размер пакета 8 для обучающих данных:

⁷ См. <https://pytorch.org/docs/stable/generated/torch.nn.Embedding.html>.

```
>>> train_ds = TensorDataset(x_train, y_train)
>>> batch_size = 8
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

и построим модель с двумя полносвязанными слоями, в одном из которых будет 8 скрытых элементов, а в другом — 4:

```
>>> hidden_units = [8, 4]
>>> input_size = x_train.shape[1]
>>> all_layers = []
>>> for hidden_unit in hidden_units:
...     layer = nn.Linear(input_size, hidden_unit)
...     all_layers.append(layer)
...     all_layers.append(nn.ReLU())
...     input_size = hidden_unit
>>> all_layers.append(nn.Linear(hidden_units[-1], 1))
>>> model = nn.Sequential(*all_layers)
>>> model
Sequential(
  (0): Linear(in_features=9, out_features=8, bias=True)
  (1): ReLU()
  (2): Linear(in_features=8, out_features=4, bias=True)
  (3): ReLU()
  (4): Linear(in_features=4, out_features=1, bias=True)
)
```

Построив модель, определим функцию потерь MSE для регрессии и применим стохастический градиентный спуск для оптимизации:

```
>>> loss_fn = nn.MSELoss()
>>> optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Затем запустим обучение модели в течение 200 эпох и будем выводить на экран потери при обучении для каждого 20 эпох:

```
>>> torch.manual_seed(1)
>>> num_epochs = 200
>>> log_epochs = 20
>>> for epoch in range(num_epochs):
...     loss_hist_train = 0
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)[:, 0]
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         loss_hist_train += loss.item()
...     if epoch % log_epochs==0:
...         print(f'Эпоха {epoch} Потеря '
...               f'{loss_hist_train/len(train_dl):.4f}')
```

```
Эпоха 0 Потеря 536.1047
Эпоха 20 Потеря 8.4361
Эпоха 40 Потеря 7.8695
Эпоха 60 Потеря 7.1891
Эпоха 80 Потеря 6.7062
Эпоха 100 Потеря 6.7599
Эпоха 120 Потеря 6.3124
Эпоха 140 Потеря 6.6864
Эпоха 160 Потеря 6.7648
Эпоха 180 Потеря 6.2156
```

После 200 эпох потеря при обучении близка к 5. Теперь мы можем оценить производительность регрессии обученной модели на тестовом наборе данных. Чтобы предсказать целевые значения новых точек данных, передадим их признаки в модель:

```
>>> with torch.no_grad():
...     pred = model(x_test.float())[:, 0]
...     loss = loss_fn(pred, y_test)
...     print(f'MSE при тестировании: {loss.item():.4f}')
...     print(f'MAE при тестировании: {nn.L1Loss()(pred, y_test).item():.4f}')
MSE при тестировании: 9.6130
MAE при тестировании: 2.1211
```

MSE на тестовом наборе составляет 9.6, а средняя абсолютная ошибка (MAE) — 2.1. На этом мы завершаем работу с проектом регрессии и переходим к проекту классификации, приведенному в следующем разделе.

13.7. Проект № 2: классификация рукописных цифр из набора MNIST

В этом проекте мы выполним классификацию рукописных цифр набора MNIST. В предыдущем разделе мы подробно рассмотрели четыре основных шага машинного обучения в PyTorch, которые нужно будет повторить и сейчас.

В главе 12 вы узнали, как загружать доступные наборы данных из модуля `torchvision`. Начнем с загрузки набора данных MNIST с помощью модуля `torchvision`.

1. Шаг настройки включает загрузку набора данных и указание гиперпараметров (размер обучающего и тестового наборов, а также размер мини-пакетов):

```
>>> import torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor(),
... ])
>>> mnist_train_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=False
... )
```

```
>>> mnist_test_dataset = torchvision.datasets.MNIST(  
...      root=image_path, train=False,  
...      transform=transform, download=False  
... )  
>>> batch_size = 64  
>>> torch.manual_seed(1)  
>>> train_dl = DataLoader(mnist_train_dataset,  
...                         batch_size, shuffle=True)
```

Здесь мы построили загрузчик данных с пакетами из 64 примеров. Теперь выполним предварительную обработку загруженных наборов данных.

2. Признаки в этом проекте — это пиксельы изображений, которые мы получили на *шаге 1*. Пользовательское преобразование мы определили с помощью `torchvision.transforms.Compose`. В этом простом случае наше преобразование состоит только из одного метода — `ToTensor()`. Метод `ToTensor()` преобразует пиксельные признаки в тензор с плавающей запятой, а также нормализует пиксельы, переводя их значения из диапазона [0, 255] в диапазон [0, 1] (в *главе 14* вы познакомитесь с некоторыми дополнительными методами преобразования данных при работе с более сложными наборами изображений). Метки представляют собой целые числа от 0 до 9, соответствующие десяти цифрам. Следовательно, нам не нужно применять к ним какое-либо масштабирование или дальнейшее преобразование. Мы можем получить доступ к необработанным пикселям с помощью атрибута `data`. Не забудьте масштабировать их до диапазона [0, 1].

Построение модели происходит на следующем шаге — после предварительной обработки данных.

3. Строим нейросетевую модель:

```
>>> hidden_units = [32, 16]  
>>> image_size = mnist_train_dataset[0][0].shape  
>>> input_size = image_size[0] * image_size[1] * image_size[2]  
>>> all_layers = [nn.Flatten()]  
>>> for hidden_unit in hidden_units:  
...     layer = nn.Linear(input_size, hidden_unit)  
...     all_layers.append(layer)  
...     all_layers.append(nn.ReLU())  
...     input_size = hidden_unit  
>>> all_layers.append(nn.Linear(hidden_units[-1], 10))  
>>> model = nn.Sequential(*all_layers)  
>>> model  
Sequential(  
  (0): Flatten(start_dim=1, end_dim=-1)  
  (1): Linear(in_features=784, out_features=32, bias=True)  
  (2): ReLU()  
  (3): Linear(in_features=32, out_features=16, bias=True)  
  (4): ReLU()  
  (5): Linear(in_features=16, out_features=10, bias=True)  
)
```



Обратите внимание, что модель начинается со слоя сглаживания, который преобразует входное изображение в одномерный тензор. Это связано с тем, что входные изображения имеют форму [1, 28, 28]. Модель имеет два скрытых слоя с 32 и 16 элементами соответственно и заканчивается выходным слоем из десяти элементов, представляющих десять классов, активируемых функцией softmax. На следующем этапе мы обучим модель и оценим ее на тестовом наборе.

4. Применяем модель для обучения, оценки и прогнозирования:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> for epoch in range(num_epochs):
...     accuracy_hist_train = 0
...     for x_batch, y_batch in train_dl:
...         pred = model(x_batch)
...         loss = loss_fn(pred, y_batch)
...         loss.backward()
...         optimizer.step()
...         optimizer.zero_grad()
...         is_correct = (
...             torch.argmax(pred, dim=1) == y_batch
...         ).float()
...         accuracy_hist_train += is_correct.sum()
...     accuracy_hist_train /= len(train_dl.dataset)
...     print(f'Эпоха {epoch} Точность '
...           f'{accuracy_hist_train:.4f}')
Эпоха 0 Точность 0.8531
...
Эпоха 9 Точность 0.9691
...
Эпоха 19 Точность 0.9813
```

В этой модели мы использовали функцию потерь перекрестной энтропии для многоклассовой классификации и оптимизатор Adam для градиентного спуска (подробнее об оптимизаторе Adam говорится в главе 14). Мы обучали модель на 20 эпохах и отображали точность обучения для каждой эпохи. Обученная модель достигла точности 96.3% на тренировочном наборе, а теперь мы оценим ее на тестовом наборе:

```
>>> pred = model(mnist_test_dataset.data / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) ==
...     mnist_test_dataset.targets
... ).float()
>>> print(f'Точность при тестировании: {is_correct.mean():.4f}')
Точность при тестировании: 0.9645
```

Точность при тестировании составляет 95.6%.

Вы только что научились создавать классифицирующую модель с помощью PyTorch.

13.8. API PyTorch более высокого уровня: краткое введение в PyTorch-Lightning

В последние годы сообщество PyTorch разработало несколько различных библиотек и API-интерфейсов, работающих поверх PyTorch. К наиболее известным примерам относятся fastai⁸, Catalyst⁹, PyTorch Lightning¹⁰ и PyTorch-Ignite¹¹.

В этом разделе мы рассмотрим PyTorch Lightning (сокращенно Lightning) — широко используемую библиотеку PyTorch, которая упрощает обучение глубоких нейронных сетей за счет удаления большей части стандартного кода. Однако, несмотря на то, что главное преимущество Lightning заключается в ее простоте и гибкости, она позволяет нам использовать множество дополнительных функций — таких как поддержка нескольких графических процессоров и быстрое обучение с низкой точностью, о чем вы можете узнать в официальной документации по адресу: <https://pytorch-lightning.rtfd.io/en/latest/>.



Существует также введение в PyTorch-Ignite, расположенное по адресу: https://github.com/rasbt/machine-learning-book/blob/main/ch13/ch13_part4_ignite.ipynb.

Ранее мы реализовали многослойный персепtron для классификации рукописных цифр в наборе данных MNIST. В следующих разделах мы повторно реализуем этот классификатор с помощью Lightning.



Установка PyTorch Lightning

Lightning можно установить через pip или conda, в зависимости от ваших предпочтений. Например, команда для установки Lightning через pip выглядит следующим образом:

```
pip install pytorch-lightning
```

А вот команда для установки Lightning через conda:

```
conda install pytorch-lightning -c conda-forge
```

Примеры кода в следующих подразделах основаны на PyTorch Lightning версии 1.5, которую можно установить, заменив pytorch-lightning на pytorch-lightning==1.5 в приведенных здесь командах.

13.8.1. Настройка модели PyTorch Lightning

Начнем мы с построения модели, которую будем обучать в следующих разделах. Определить модель для Lightning относительно просто, поскольку она основана на обычном коде Python и PyTorch. Все, что требуется для реализации модели Lightning, — это использовать LightningModule вместо обычного модуля PyTorch. Чтобы воспользоваться

⁸ См. <https://docs.fast.ai/>.

⁹ См. <https://github.com/catalyst-team/catalyst>.

¹⁰ См. <https://www.pytorchlightning.ai>, <https://lightning-flash.readthedocs.io/en/latest/quickstart.html>.

¹¹ См. <https://github.com/pytorch/ignite>.

удобными функциями PyTorch, такими как API-интерфейс обучателя и автоматическое ведение журнала, мы просто определим несколько методов со специальными именами:

```
import pytorch_lightning as pl
import torch
import torch.nn as nn

from torchmetrics import Accuracy

class MultiLayerPerceptron(pl.LightningModule):
    def __init__(self, image_shape=(1, 28, 28), hidden_units=(32, 16)):
        super().__init__()
        # Новые атрибуты PyTorch Lighting:
        self.train_acc = Accuracy()
        self.valid_acc = Accuracy()
        self.test_acc = Accuracy()

        # Модель как в предыдущей секции:
        input_size = image_shape[0] * image_shape[1] * image_shape[2]
        all_layers = [nn.Flatten()]
        for hidden_unit in hidden_units:
            layer = nn.Linear(input_size, hidden_unit)
            all_layers.append(layer)
            all_layers.append(nn.ReLU())
            input_size = hidden_unit

        all_layers.append(nn.Linear(hidden_units[-1], 10))
        self.model = nn.Sequential(*all_layers)

    def forward(self, x):
        x = self.model(x)
        return x

    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.train_acc.update(preds, y)
        self.log("train_loss", loss, prog_bar=True)
        return loss

    def training_epoch_end(self, outs):
        self.log("train_acc", self.train_acc.compute())
    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = nn.functional.cross_entropy(self(x), y)
        preds = torch.argmax(logits, dim=1)
        self.valid_acc.update(preds, y)
        self.log("valid_loss", loss, prog_bar=True)
```

```

    self.log("valid_acc", self.valid_acc.compute(), prog_bar=True)
    return loss

def test_step(self, batch, batch_idx):
    x, y = batch
    logits = self(x)
    loss = nn.functional.cross_entropy(self(x), y)
    preds = torch.argmax(logits, dim=1)
    self.test_acc.update(preds, y)
    self.log("test_loss", loss, prog_bar=True)
    self.log("test_acc", self.test_acc.compute(), prog_bar=True)
    return loss

def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.parameters(), lr=0.001)
    return optimizer

```

Рассмотрим все эти методы по порядку. Как можно видеть, конструктор `_init_` содержит тот же код модели, который мы использовали в предыдущем разделе. Что в нем нового, так это атрибуты точности — такие как `self.train_acc = Accuracy()`. Это позволяет нам отслеживать точность во время обучения. Метод `Accuracy` импортирован из модуля `torchmetrics`, который должен автоматически устанавливаться вместе с `Lightning`. Если у вас не получилось импортировать `torchmetrics`, попробуйте установить его с помощью команды¹²:

```
pip install torchmetrics
```

Метод `forward` реализует простой прямой проход, который возвращает логиты (выходные данные последнего полностью подключенного уровня нашей сети перед уровнем `softmax`), когда мы вызываем нашу модель, передавая ей входные данные. Логиты, вычисляемые с помощью метода `forward` путем вызова `self(x)`, используются для этапов обучения, валидации и тестирования, которые будут описаны далее.

Методы `training_step`, `training_epoch_end`, `validation_step`, `test_step` и `configure_optimizers` — это методы, специально предназначенные для `Lightning`. Например, `training_step` определяет один прямой проход во время обучения, где мы также отслеживаем точность и потери, чтобы проанализировать их позже. Обратите внимание, что мы вычисляем точность с помощью `self.train_acc.update(preds, y)`, но пока не регистрируем ее в журнале. Метод `training_step` выполняется для каждого отдельного пакета во время обучения, а с помощью метода `training_epoch_end`, который выполняется в конце каждой эпохи обучения, мы вычисляем точность на тренировочном наборе на основе значений точности, которые мы накопили в процессе обучения.

Методы `validation_step` и `test_step` определяют — аналогично методу `training_step`, — как должны быть вычислены процессы валидации и оценки. Как и в случае `training_step`, каждый вызов `validation_step` и `test_step` получает один пакет, поэтому мы регистрируем точность, полученную из атрибута `Accuracy` метода `torchmetric`. Однако обратите внимание, что `validation_step` вызывается только через определенные проме-

¹² Дополнительную информацию по установке этого модуля можно найти по адресу: <https://torchmetrics.readthedocs.io/en/latest/pages/quickstart.html>.

жутки времени — например, после каждой эпохи обучения. Вот почему мы регистрируем точность при валидации внутри шага валидации, тогда как точность при обучении мы регистрируем после каждой эпохи обучения, — иначе график точности, который мы проверяем позже, будет выглядеть слишком зашумленным.

Наконец, с помощью метода `configure_optimizers` мы указываем оптимизатор, используемый для обучения. В следующих двух разделах мы обсудим подготовку набора данных и обучение модели.

13.8.2. Настройка загрузчиков данных для Lightning

Есть три основных способа подготовки набора данных для Lightning. Мы можем:

- ◆ сделать набор данных частью модели;
- ◆ настроить загрузчики данных как обычно и передать их методу обучения `Lightning Trainer` — он представлен в следующем разделе;
- ◆ создать модуль `LightningDataModule`.

Здесь мы воспользуемся модулем `LightningDataModule`, который предлагает наиболее организованный подход.

`LightningDataModule` состоит из пяти основных методов, как показано в следующем примере кода:

```
from torch.utils.data import DataLoader
from torch.utils.data import random_split
from torchvision.datasets import MNIST
from torchvision import transforms

class MnistDataModule(pl.LightningDataModule):
    def __init__(self, data_path='./'):
        super().__init__()
        self.data_path = data_path
        self.transform = transforms.Compose([transforms.ToTensor()])

    def prepare_data(self):
        MNIST(root=self.data_path, download=True)

    def setup(self, stage=None):
        # stage может принимать значения
        # 'fit', 'validate', 'test', or 'predict'
        # здесь укажите нужное
        mnist_all = MNIST(
            root=self.data_path,
            train=True,
            transform=self.transform,
            download=False
        )

        self.train, self.val = random_split(
            mnist_all, [55000, 5000], generator=torch.Generator().manual_seed(1)
        )
```

```

        self.test = MNIST(
            root=self.data_path,
            train=False,
            transform=self.transform,
            download=False
        )

    def train_dataloader(self):
        return DataLoader(self.train, batch_size=64, num_workers=4)

    def val_dataloader(self):
        return DataLoader(self.val, batch_size=64, num_workers=4)

    def test_dataloader(self):
        return DataLoader(self.test, batch_size=64, num_workers=4)

```

В методе `prepare_data` мы определяем общие шаги — такие как загрузка набора данных. В методе `setup` определяем наборы данных, используемые для обучения, валидации и тестирования. Поскольку в MNIST не выделены данные для валидации, мы применяем функцию `random_split`, чтобы разделить обучающий набор из 60 тыс. примеров на 55 тыс. примеров для обучения и 5 тыс. — для валидации.

Методы загрузки данных не нуждаются в пояснениях и определяют, как загружаются соответствующие наборы данных. Теперь мы можем инициализировать модуль данных и использовать его для обучения, валидации и тестирования в следующих разделах:

```

torch.manual_seed(1)
mnist_dm = MnistDataModule()

```

13.8.3. Обучение модели с помощью класса PyTorch Lightning Trainer

Пришло время пожинать плоды своих трудов по настройке модели с помощью именованных методов, а также модуля данных Lightning. Lightning реализует класс `Trainer`, который делает обучение модели очень удобным, беря на себя все промежуточные шаги — такие как вызовы `zero_grad()`, `reverse()` и `optimizer.step()`. Кроме того, в качестве бонуса он позволяет нам легко указать один или несколько графических процессоров (если они доступны):

```

mnistclassifier = MultiLayerPerceptron()

if torch.cuda.is_available(): # если есть GPU
    trainer = pl.Trainer(max_epochs=10, gpus=1)
else:
    trainer = pl.Trainer(max_epochs=10)

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)

```

С помощью этого кода мы обучаем наш многослойный персептрон в течение 10 эпох. Во время обучения мы видим удобный указатель процесса, который отслеживает эпохи и основные показатели, такие как потери при обучении и валидации:

```
Epoch 9: 100% 939/939 [00:07<00:00, 130.42it/s, loss=0.1, v_num=0, train_
loss=0.260, valid_loss=0.166, valid_acc=0.949]
```

После завершения обучения можно более подробно изучить записанные в журнал показатели, как показано в следующем подразделе.

13.8.4. Оценка модели с помощью TensorBoard

В предыдущем разделе мы убедились в удобстве использования класса Trainer. Еще одна приятная особенность Lightning — возможность ведения журнала. Напомним, что ранее мы указали несколько шагов `self.log` в нашей модели Lightning. После (и даже в процессе) обучения мы можем визуализировать их в TensorBoard. (Заметим, что Lightning также поддерживает другие механизмы журнала. Для получения дополнительной информации обратитесь к официальной документации, доступной по адресу: <https://pytorch-lightning.readthedocs.io/en/latest/common/loggers.html>.)



Установка TensorBoard

TensorBoard можно установить через pip или conda, в зависимости от ваших предпочтений. Например, команда для установки TensorBoard через pip выглядит следующим образом:

```
pip install tensorboard
```

А вот команда для установки Lightning через conda:

```
conda install tensorboard -c conda-forge
```

Код в следующем подразделе основан на TensorBoard версии 2.4, которую вы можете установить, заменив в приведенных командах `tensorboard` на `tensorboard==2.4`.

По умолчанию Lightning отслеживает обучение в подкаталоге с именем `lightning_logs`. Чтобы визуализировать обучающие прогоны, выполните следующий код в терминале командной строки, который откроет TensorBoard в вашем браузере:

```
tensorboard --logdir lightning_logs/
```

В качестве альтернативы, если вы запускаете код в блокноте Jupyter, можно добавить следующий код в ячейку блокнота Jupyter, чтобы отобразить панель инструментов TensorBoard непосредственно в блокноте:

```
%load_ext tensorboard
%tensorboard --logdir lightning_logs/
```

На рис. 13.9 показана информационная панель TensorBoard, отображающая точность обучения и проверки. Обратите внимание, что в нижнем левом углу показан переключатель `version_0`. Если вы запускаете обучающий код несколько раз, Lightning будет сохранять результаты как отдельные вложенные папки: `version_0`, `version_1`, `version_2` и т. д.

Глядя на графики точности при обучении и валидации, приведенные на рис. 13.9, мы можем предположить, что обучение модели в течение нескольких дополнительных эпох может улучшить ее производительность.

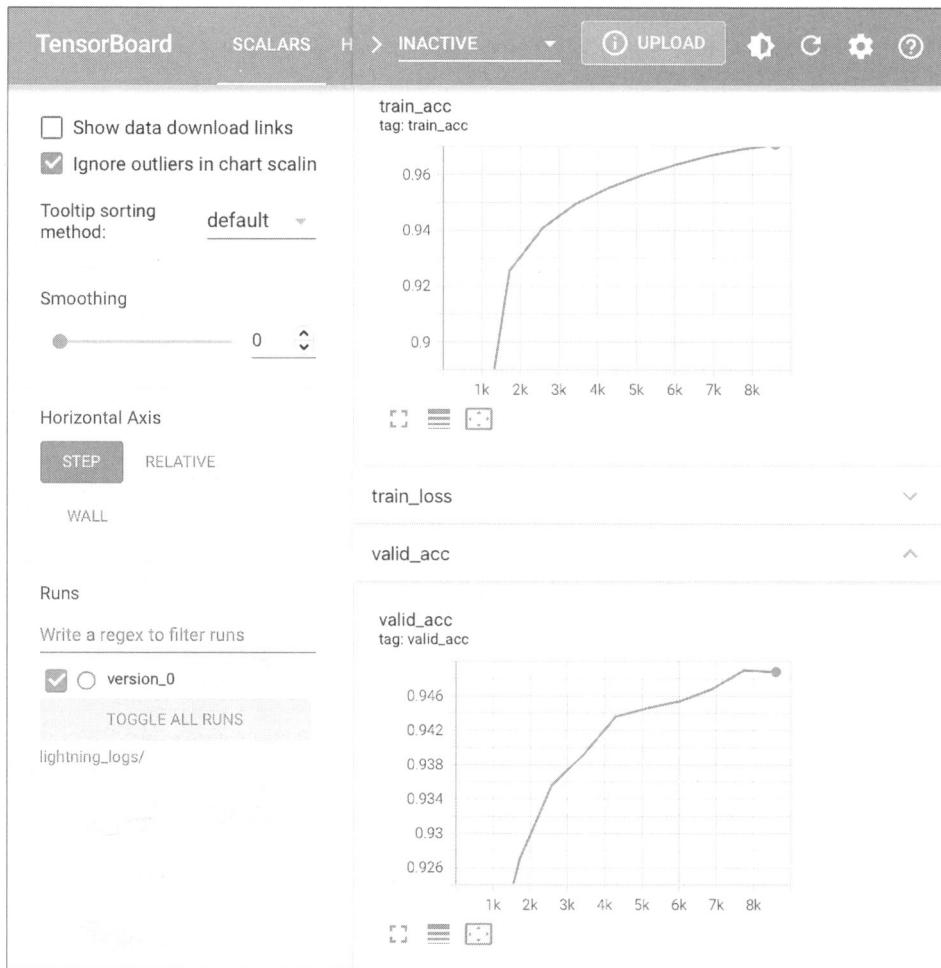


Рис. 13.9. Панель инструментов TensorBoard

Lightning позволяет нам загружать обученную модель и удобно проводить дополнительное обучение на протяжении нескольких эпох. Как упоминалось ранее, Lightning сохраняет результаты обучающих прогонов в отдельных подкаталогах. На рис. 13.10 показано содержимое подкаталога `version_0`, в котором находятся файлы журнала и контрольная точка модели для повторной загрузки.

Например, мы можем использовать следующий код, чтобы загрузить последнюю контрольную точку модели из этой папки и обучить модель с помощью метода `fit`:

```
if torch.cuda.is_available(): # if you have GPUs
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
                                logs/version_0/checkpoints/epoch=8-step=7739.ckpt', gpus=1)
else:
    trainer = pl.Trainer(max_epochs=15, resume_from_checkpoint='./lightning_
                                logs/version_0/checkpoints/epoch=8-step=7739.ckpt')

trainer.fit(model=mnistclassifier, datamodule=mnist_dm)
```

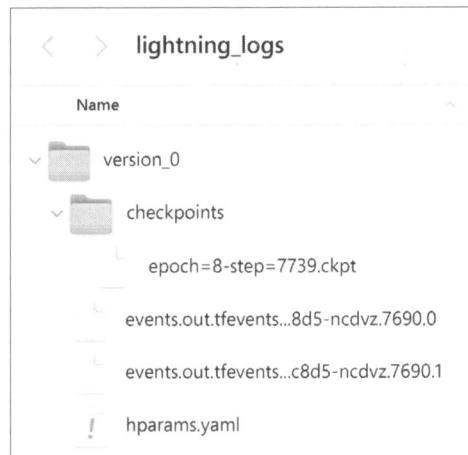


Рис. 13.10. Файлы журнала PyTorch Lightning

Здесь мы установили для `max_epochs` значение 15, что позволило обучить модель на 5 дополнительных эпохах (ранее мы обучали ее на 10 эпохах).

Теперь снова обратимся к панели управления TensorBoard (рис. 13.11) и посмотрим, стоило ли обучать модель в течение нескольких дополнительных эпох.

Как можно здесь видеть, TensorBoard позволяет нам показывать результаты дополнительных эпох обучения (`version_1`) рядом с предыдущими (`version_0`), что очень удобно. Действительно, мы видим, что обучение еще на пяти эпохах улучшило точность при валидации. На этом этапе мы можем принять решение обучить модель на еще большем количестве эпох, что оставляем вам в качестве упражнения.

Когда обучение завершено, оценим модель на тестовом наборе, используя следующий код:

```
trainer.test(model=mnistclassifier, datamodule=mnist_dm)
```

Результирующая производительность модели на тестовом наборе после обучения в течение 15 эпох составляет примерно 95%:

```
[{'test_loss': 0.14912301301956177, 'test_acc': 0.9499600529670715}]
```

Имейте в виду, что PyTorch Lightning также автоматически сохраняет для нас модель. Если вы захотите повторно использовать модель позже, ее можно без труда загрузить с помощью следующего кода:

```
model = MultiLayerPerceptron.load_from_checkpoint("path/to/checkpoint.ckpt")
```



Где узнать больше о PyTorch Lightning?

Чтобы узнать больше о Lightning, посетите официальный веб-сайт с учебными пособиями и примерами по адресу: <https://pytorch-lightning.readthedocs.io>.

У Lightning также есть активное сообщество в Slack, которое тепло встречает новых пользователей и участников. Чтобы узнать больше, посетите официальный веб-сайт Lightning по адресу: <https://www.pytorchlightning.ai>.

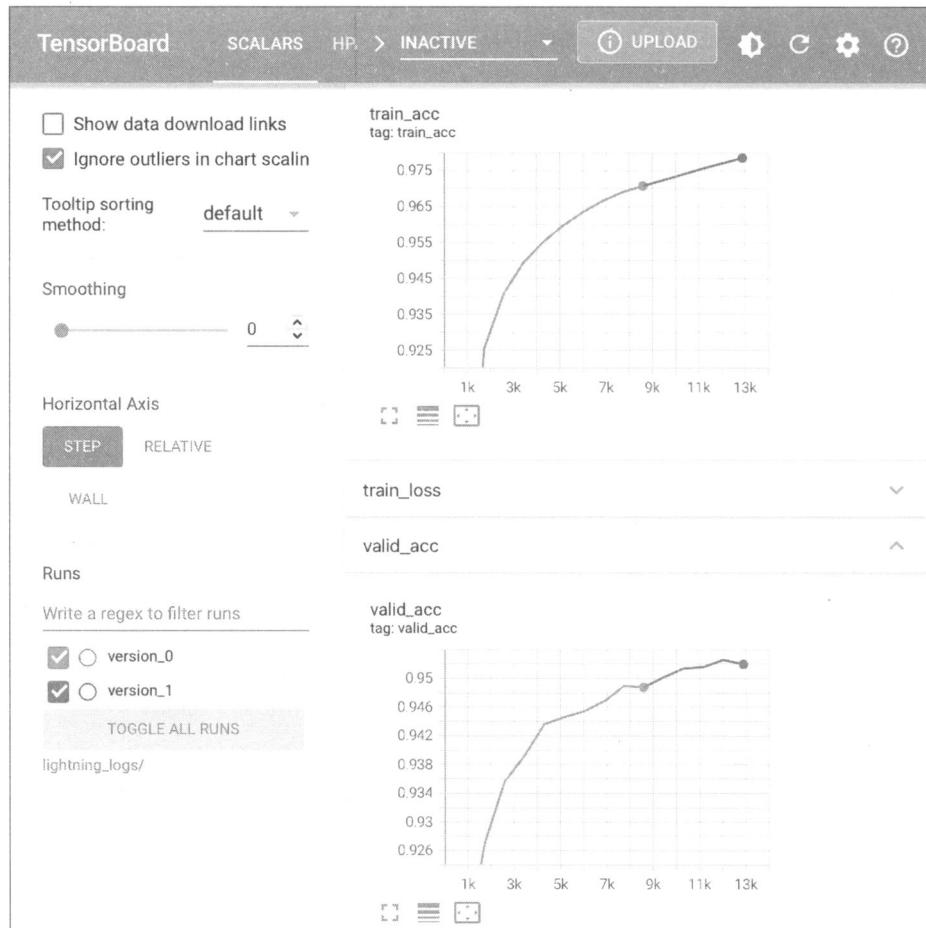


Рис. 13.11. Панель инструментов TensorBoard после обучения модели в течение пяти дополнительных эпох

13.9. Заключение

В этой главе мы рассмотрели самые важные и полезные функции PyTorch. Начали мы с обсуждения динамического графа вычислений PyTorch, который делает реализацию вычислений очень удобной. Мы также рассмотрели семантику определения тензорных объектов PyTorch в качестве параметров модели.

После знакомства с концепцией вычисления частных производных и градиентов произвольных функций мы более подробно познакомились с модулем `torch.nn`, предостав员ющим нам удобный интерфейс для построения более сложных глубоких нейросетевых моделей. Наконец, мы завершили эту главу, решив задачу регрессии и классификации с помощью новых навыков.

Итак, вы изучили основные механизмы PyTorch, а в следующей главе будет представлена концепция архитектуры *сверточных нейронных сетей* (convolutional neural network, CNN) для глубокого обучения — мощных моделей, показывающих отличные результаты в области компьютерного зрения.

14

Классификация изображений с помощью глубоких сверточных нейронных сетей

В предыдущей главе мы подробно рассмотрели различные аспекты создания нейронной сети с помощью PyTorch и модулей автоматического дифференцирования, вы также познакомились с тензорами и специальными функциями и научились работать с `torch.nn`. Здесь же будет рассказано о *сверточных нейронных сетях* (Convolutional Neural Network, CNN), удобных для выполнения классификации изображений. Мы начнем с обсуждения основных функциональных блоков CNN, а затем углубимся в архитектуру CNN и покажем, как реализовать CNN в PyTorch.

В этой главе мы рассмотрим следующие темы:

- ◆ операции свертки в одном и двух измерениях;
- ◆ функциональные блоки архитектуры CNN;
- ◆ реализация глубоких CNN в PyTorch;
- ◆ способы дополнения данных для повышения производительности обобщения;
- ◆ реализация классификатора лиц на основе CNN для распознавания улыбки.

14.1. Функциональные блоки CNN

CNN — это семейство моделей, основанное на имитации принципов работы зрительной области коры человеческого головного мозга при распознавании им различных объектов. История CNN восходит к 1990-м годам, когда Ян Лекун и его коллеги предложили новую архитектуру нейронной сети для классификации рукописных цифр по изображениям¹.



Зрительная область коры головного мозга

Первые представления о том, как функционирует зрительная область коры нашего мозга, получены Дэвидом Хьюбелем и Торстеном Визелем в 1959 году, когда они ввели микроэлектрод в первичную зрительную область коры головного мозга кошки, находящейся в состоянии анестезии. Они заметили, что нейроны по-разному реагировали на различные изображения, которые ей показывали. В конечном итоге это привело к открытию различных слоев зрительной области коры. В то время как

¹ См. «Handwritten Digit Recognition with a Back-Propagation Network» by Y. LeCun, and colleagues, 1989, опубликовано в материалах конференции Neural Information Processing Systems (NeurIPS).

первичный слой в основном обнаруживает края и прямые линии, слои более высокого порядка больше фокусируются на извлечении сложных форм и узоров.

Благодаря выдающимся успехам в задачах классификации изображений этот тип нейросети прямого распространения привлек большое внимание исследователей и привел к огромным достижениям в области компьютерного зрения. Несколько лет спустя, в 2019 году, Ян Лекун вместе с двумя другими исследователями, Йошуа Бенджио и Джейфри Хинтоном, имена которых вы встречали в предыдущих главах, получили премию Тьюринга (самую престижную награду в области компьютерных наук) за вклад в области искусственного интеллекта.

В следующих разделах мы обсудим более общие концепции CNN и разберемся, почему сверточные архитектуры часто называют «слоями извлечения признаков». Затем мы углубимся в теоретическое определение операции свертки, которая обычно используется в CNN, и рассмотрим примеры вычисления сверток в одном и двух измерениях.

14.1.1. Устройство CNN и понятие иерархии признаков

Успешное извлечение характерных (релевантных) признаков является ключом к высокой производительности любого алгоритма машинного обучения, но традиционные модели машинного обучения полагаются на входные признаки, которые могут быть предоставлены экспертом в предметной области или получены вычислительными методами извлечения признаков.

Определенные типы нейронных сетей, такие как CNN, могут автоматически извлекать из необработанных данных признаки, которые наиболее полезны для конкретной задачи. По этой причине принято рассматривать слои CNN как экстракторы признаков: первые слои (сразу после входного слоя) извлекают *низкоуровневые признаки* из необработанных данных, а последующие слои (часто это полносвязные слои, как в многослойном персептроне) используют эти признаки для прогнозирования непрерывного целевого значения или метки класса.

Некоторые типы многослойных нейросетей, и в частности глубокие CNN, строят так называемую *иерархию признаков*, послойно объединяя низкоуровневые признаки для формирования признаков высокого уровня. Например, если мы имеем дело с изображениями, то низкоуровневые признаки — такие как края и пятна, извлекаются из более ранних слоев, которые объединяются для формирования высокоуровневых признаков. Эти признаки высокого уровня могут образовывать более сложные формы — например, общие контуры таких объектов, как здания, кошки или собаки.

Как показано на рис. 14.1, CNN вычисляет *карты признаков* (feature map) из входного изображения, где каждый элемент приходит из локального участка пикселов во входном изображении.

Этот локальный участок пикселов называется *локальным рецептивным полем* (local receptive field). CNN обычно очень хорошо работают с изображениями, и это во многом связано с двумя ключевыми аспектами:

- ◆ *разреженная связность* — один элемент на карте объектов связан только с небольшим участком пикселов. (Это сильно отличается от связи со всем входным изображением, как в случае MLP. Возможно, вам будет полезно оглянуться назад и вспом-

- нить, как в главе 11 реализована полносвязная сеть, подключенная ко всему изображению.);
- ◆ совместное использование параметров — одинаковые веса используются для разных фрагментов входного изображения.

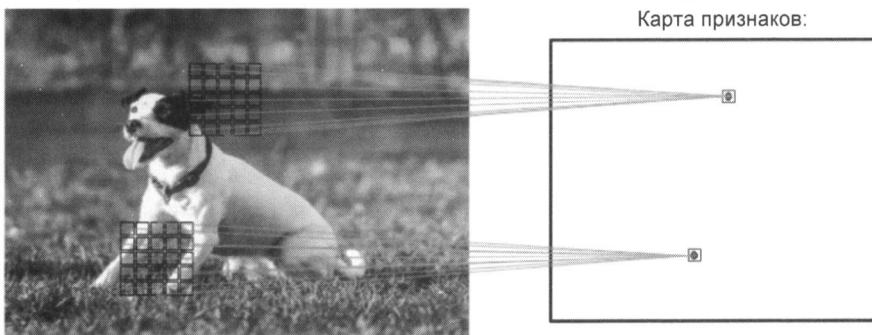


Рис. 14.1. Создание карты признаков из изображения (фото Александра Даммера на Unsplash)

Как прямое следствие этих двух подходов, замена обычного полносвязного MLP слоем свертки существенно уменьшает количество весов (параметров) в сети, и мы видим улучшение способности выделять *значимые* признаки. В контексте данных изображения имеет смысл предположить, что соседние пиксели, как правило, более релевантны друг другу, чем пиксели, находящиеся далеко друг от друга.

Как правило, CNN состоят из нескольких слоев свертки и подвыборки, за которыми в конце концов следуют один или несколько полносвязных слоев. Полносвязные слои, по сути, представляют собой MLP, где каждый входной узел i связан с каждым выходным узлом j с весом w_{ij} (более подробно мы рассмотрели этот вопрос в главе 11).

Важно отметить, что слои подвыборки, обычно известные как *объединяющие слои* (pooling layer, слой пулинга), не имеют обучаемых параметров, — например, в объединяющих слоях нет весов или смещений. Однако как сверточный, так и полносвязный слой имеет веса и смещения, которые оптимизируются во время обучения.

В следующих разделах мы более подробно изучим сверточные и объединяющие слои и посмотрим, как они работают. Чтобы разобраться, как работают операции свертки, мы начнем со свертки в одном измерении, которая иногда используется для работы с определенными типами последовательных данных, такими как текст. После знакомства с одномерными свертками мы рассмотрим типичные двумерные свертки, которые обычно применяются к двумерным изображениям.

14.1.2. Выполнение дискретных сверток

Дискретная свертка (или просто *свертка*) — это фундаментальная операция в CNN. Поэтому важно понимать, как работает эта операция. Здесь мы рассмотрим математическое определение и обсудим некоторые простые алгоритмы для вычисления сверток одномерных тензоров (векторов) и двумерных тензоров (матриц).

Учтите, что формулы и описания в этом разделе предназначены исключительно для того, чтобы пояснить вам, как работают операции свертки в CNN. На самом деле в та-

ких пакетах, как PyTorch, применяются гораздо более эффективные реализации сверточных операций, как вы увидите далее в этой главе.



Математические обозначения

В этой главе мы будем использовать:

- нижний индекс — для обозначения размера многомерного массива (тензора). Например, $A_{n_1 \times n_2}$ — это двумерный массив размером $n_1 \times n_2$;
- квадратные скобки [] — для обозначения индексов многомерного массива. Например, $A[i, j]$ обозначает элемент с индексом i, j матрицы A ;
- специальный символ * — для обозначения операции свертки между двумя векторами или матрицами, которую не следует путать с оператором умножения * в Python.

Дискретные свертки в одном измерении

Давайте начнем с некоторых основных определений и обозначений. Дискретная свертка для двух векторов x и w обозначается так: $y = x * w$, где вектор x является нашим входом (иногда называется *сигналом*), а w — *фильтром*, или *ядром*. Дискретная свертка математически определяется следующим образом:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i - k] w[k].$$

Как упоминалось ранее, скобки [] служат для обозначения индексов элементов вектора. Индекс i проходит через каждый элемент выходного вектора y . В приведенной формуле есть два странных момента, которые нам нужно прояснить: индексы от $-\infty$ до $+\infty$ и отрицательные индексы для x .

Тот факт, что сумма проходит через индексы от $-\infty$ до $+\infty$, кажется странным, главным образом потому, что в приложениях машинного обучения мы всегда имеем дело с конечными векторами признаков. Например, если x имеет 10 признаков с индексами 0, 1, 2, ..., 8, 9, то индексы $-\infty$: -1 и 10 : $+\infty$ выходят за границы для x . Поэтому для правильного вычисления суммы в приведенной формуле предполагается, что x и w заполнены нулями. Это даст нам выходной вектор y , который также имеет бесконечный размер и также содержит множество нулей. Поскольку на практике это бесполезно, x дополняется только конечным числом нулей.

Этот процесс называется *заполнением нулями* (zero-padding), или просто *заполнением* (padding). Количество нулей, дополненных с каждой стороны, мы будем обозначать как p . Пример заполнения одномерного вектора x показан на рис. 14.2.

Предположим, что исходный входной вектор x и фильтр w содержат n и m элементов соответственно, где $m \leq n$. Следовательно, дополненный вектор x^p имеет размер $n + 2p$. Формула для практического вычисления дискретной свертки примет тогда следующий вид:

$$y = x * w \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i + m - k] w[k].$$



Рис. 14.2. Пример заполнения одномерного вектора

Теперь, когда мы решили проблему с бесконечным индексом, второй проблемой является вычисление индексов x в выражении $i + m - k$. Здесь важно отметить, что x и w в этой сумме индексируются в разных направлениях. Вычисление суммы с одним индексом, идущим в обратном направлении, эквивалентно вычислению суммы с обоими индексами в прямом направлении после «переворачивания» одного из векторов: x или w — после их заполнения. Перевернув один из векторов, мы можем просто вычислить их скалярное произведение. Предположим, мы переворачиваем фильтр w , чтобы получить перевернутый фильтр w^r . Затем вычисляем скалярное произведение $x[i:i+m].w^r$ для получения одного элемента $y[i]$, где $x[i:i+m]$ — фрагмент x размером m . Эта операция повторяется, как в подходе со скользящим окном, чтобы получить все элементы вывода.

На рис. 14.3 показан пример с $x = [3, 2, 1, 7, 1, 2, 5, 4]$ и $w = \begin{bmatrix} 1 & 3/4 & 1 & 1/4 \\ 2 & 3 & 4 \end{bmatrix}$, где вычисляются первые три выходных элемента.

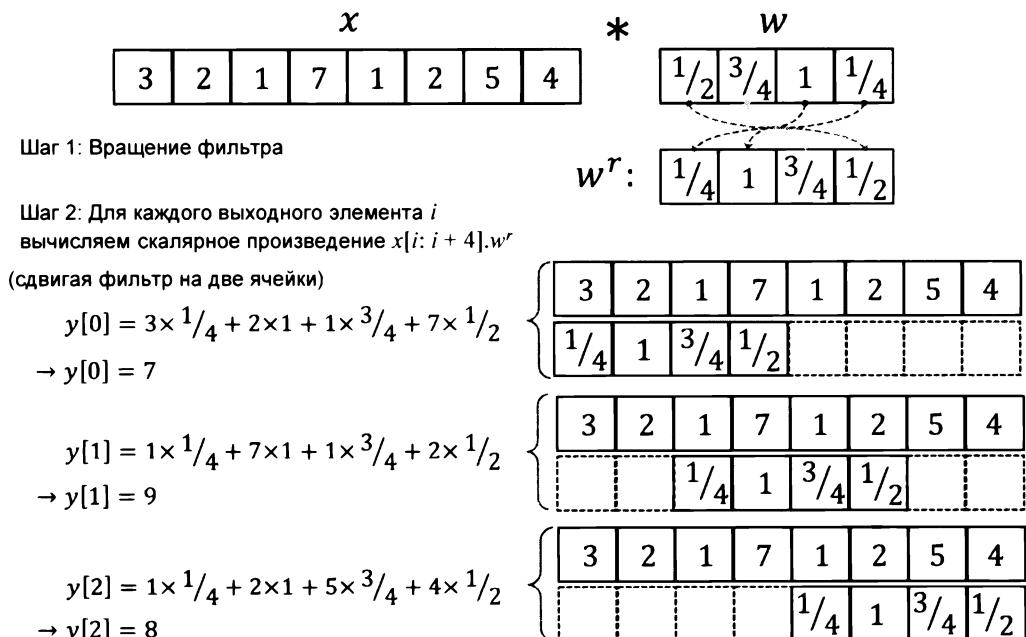


Рис. 14.3. Шаги вычисления дискретной свертки

В этом примере мы видим, что размер заполнения равен нулю ($p = 0$). Также обратите внимание, что перевернутый фильтр w' смещается на две ячейки при каждом сдвиге. Это еще один гиперпараметр свертки — *шаг (stride) s*. В приведенном примере шаг равен двум, т. е. $s = 2$. Шаг должен быть положительным числом, меньшим, чем размер входного вектора. Подробнее о заполнении и шаге мы поговорим в следующем разделе.



Взаимная корреляция

Взаимная корреляция (или просто корреляция) между входным вектором и фильтром обозначается $y = x * w$ и очень похожа на родственника свертки с небольшим отличием — при взаимной корреляции умножение выполняется в одном и том же направлении. Следовательно, нет необходимости вращать матрицу фильтра w в каждом измерении. Математически перекрестная корреляция определяется следующим образом:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k] w[k].$$

К корреляции можно применить приведенные ранее правила заполнения и шага. Заметим, что большинство сред глубокого обучения (включая PyTorch) реализуют взаимную корреляцию, но называют ее *сверткой*, что является распространенным соглашением в области глубокого обучения.

Заполнение входных данных для управления размером выходных карт признаков

До сих пор мы использовали заполнение нулями только в свертках для вычисления выходных векторов конечного размера. Технически можно применять заполнение с любым $p \geq 0$. В зависимости от выбора p граничные ячейки можно обрабатывать иначе, чем ячейки, расположенные в середине x .

Теперь рассмотрим пример, где $n = 5$ и $m = 3$. Тогда при $p = 0$ ячейка $x[0]$ используется для вычисления только одного выходного элемента (например, $y[0]$), а $x[1]$ — при вычислении двух выходных элементов (например, $y[0]$ и $y[1]$). Очевидно, что такое разное отношение к элементам x может искусственно придавать больше значения среднему элементу $x[2]$, т. к. он фигурирует в большинстве вычислений. Мы можем избежать этой проблемы, если выберем $p = 2$, и в этом случае каждый элемент x будет участвовать в вычислении трех элементов y .

Кроме того, размер вывода y также зависит от выбора стратегии заполнения, которой мы следуем. На практике обычно используются три режима заполнения: *полный (full)*, *равный (same)* и *правильный (valid)*:

- ◆ в режиме полного заполнения $p = m - 1$. Такое заполнение увеличивает размеры вывода, поэтому оно редко находит место в архитектурах CNN;
- ◆ режим равенства обычно используют для того, чтобы выходной вектор имел тот же размер, что и входной вектор x . В этом случае параметр заполнения p вычисляется в соответствии с размером фильтра, а также требованием, чтобы размеры входа и выхода были одинаковыми;
- ◆ наконец, вычисление свертки в правильном режиме относится к случаю, когда $p = 0$ (нет заполнения).

На рис. 14.4 показаны три разных режима заполнения для простого ввода 5×5 пикселов с размером ядра 3×3 и шагом 1.

Наиболее часто используемым режимом заполнения в CNN является равное заполнение. Одно из его преимуществ по сравнению с другими режимами — сохранение размера вектора (или высоты и ширины входных изображений), когда мы работаем над задачами, связанными с изображениями в компьютерном зрении, — что делает проектирование сетевой архитектуры более удобным.

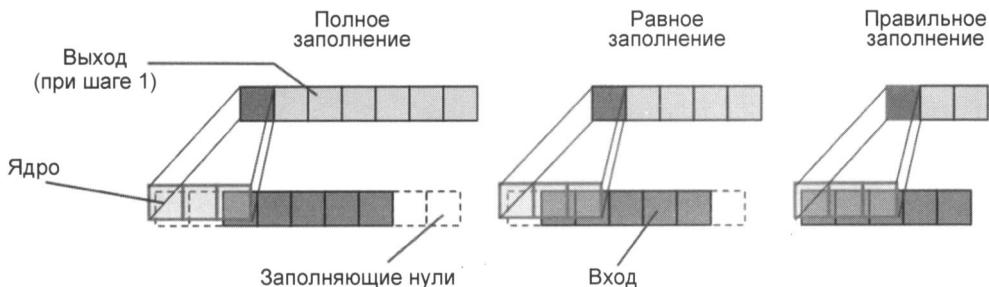


Рис. 14.4. Три режима заполнения

Одним из серьезных недостатков правильного заполнения по сравнению с полным и равным является существенное уменьшение объема тензоров в многослойной нейросети, что может отрицательно сказаться на ее производительности. На практике следует стараться сохранить пространственный размер, используя одинаковый режим для сверточных слоев, а для уменьшения размера применять объединяющие слои или сверточные слои с шагом 2, как описано в соответствующем учебном курсе².

В случае же полного заполнения размер вывода получается больше, чем размер ввода. Полное заполнение обычно применяется в приложениях обработки сигналов, где важно минимизировать граничные эффекты. Однако в контексте глубокого обучения граничные эффекты обычно не представляют проблемы, поэтому на практике мы редко встречаем полное заполнение.

Определение размера вывода свертки

Выходной размер свертки определяется общим числом сдвигов фильтра w вдоль входного вектора. Предположим, что входной вектор имеет размер n , а размер фильтра — m . Следовательно, размер вывода, полученного в результате вычисления $y = x * w$ с заполнением p и шагом s , определяется так:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1.$$

Здесь $\lfloor \cdot \rfloor$ обозначает операцию *floor*.

² См. «Striving for Simplicity: The All Convolutional Net ICLR» by Jost Tobias Springenberg, Alexey Dosovitskiy, and others, 2015, <https://arxiv.org/abs/1412.6806>.



Операция floor

Операция `floor` возвращает наибольшее целое число, равное или меньшее, чем входное значение, например:

$$\text{floor}(1.77) = \lfloor 1.77 \rfloor = 1.$$

Рассмотрим следующие два случая:

- ◆ необходимо вычислить размер вывода для входного вектора размером 10 с ядром свертки размером 5, заполнением 2 и шагом 1:

$$n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left\lceil \frac{10 + 2 \times 2 - 5}{1} \right\rceil + 1 = 10.$$

(Заметим, что здесь размер вывода равен размеру ввода, поэтому мы можем сделать вывод, что это режим одинакового заполнения.)

- ◆ как изменится размер вывода для того же входного вектора, если мы применяем ядро размером 3 и шаг 2?

$$n = 10, m = 3, p = 2, s = 2 \rightarrow o = \left\lceil \frac{10 + 2 \times 2 - 3}{2} \right\rceil + 1 = 6.$$

Если вам интересно узнать больше о размере вывода свертки, мы рекомендуем статью Винсента Дюмулена и Франческо Визина³.

Наконец, в качестве примера вычисления одномерной свертки в следующем блоке кода показана простейшая реализация, а результаты выполнения этого кода сравниваются с результатом вызова готовой функции `numpy.convolve`:

```
>>> import numpy as np
>>> def conv1d(x, w, p=0, s=1):
...     w_rot = np.array(w[::-1])
...     x_padded = np.array(x)
...     if p > 0:
...         zero_pad = np.zeros(shape=p)
...         x_padded = np.concatenate([
...             zero_pad, x_padded, zero_pad
...         ])
...     res = []
...     for i in range(0, int((len(x_padded) - len(w_rot))) + 1, s):
...         res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
...     return np.array(res)
>>> ## Testing:
>>> x = [1, 3, 2, 4, 5, 6, 1, 3]
>>> w = [1, 0, 3, 1, 2]
>>> print('Реализация Conv1d:',
...       conv1d(x, w, p=2, s=1))
Реализация Conv1d: [ 5. 14. 16. 26. 24. 34. 19. 22.]
>>> print('Результаты NumPy:',
...       np.convolve(x, w, mode='same'))
Результаты NumPy: [ 5 14 16 26 24 34 19 22]
```

³ См. «A guide to convolution arithmetic for deep learning» by Vincent Dumoulin and Francesco Visin, <https://arxiv.org/abs/1603.07285>.

До сих пор мы в основном рассматривали свертки для векторов (одномерные свертки) и начали с одномерного варианта, чтобы облегчить понимание концепции. В следующем разделе мы более подробно рассмотрим двумерные свертки, которые являются базовыми компонентами CNN для задач, связанных с изображениями.

Вычисление дискретной двумерной свертки

Понятия, которые вы изучили в предыдущих разделах, легко применимы к двумерным данным. Когда мы имеем дело с двумерными входными данными в виде матрицы $X_{m_1 \times n_2}$ и матрицы фильтра $W_{m_1 \times m_2}$, где $m_1 \leq n_1$ и $m_2 \leq n_2$, то матрица $Y = X * W$ будет результатом двумерной свертки между X и W . Математически эту операцию можно записать следующим образом:

$$Y = X * W \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2].$$

Нужно заметить, что если убрать одно из измерений, оставшаяся формула будет точно такой же, как та, которую мы использовали ранее для вычисления одномерной свертки. Фактически все ранее упомянутые методы, такие как заполнение нулями, вращение матрицы фильтра и использование шагов, также применимы к двумерным сверткам при условии, что они распространяются на оба измерения независимо. На рис. 14.5 показана двумерная свертка входной матрицы размером 8×8 с использованием ядра размером 3×3 . Входная матрица дополняется нулями с $p = 1$. Результат двумерной свертки тоже будет иметь размер 8×8 .

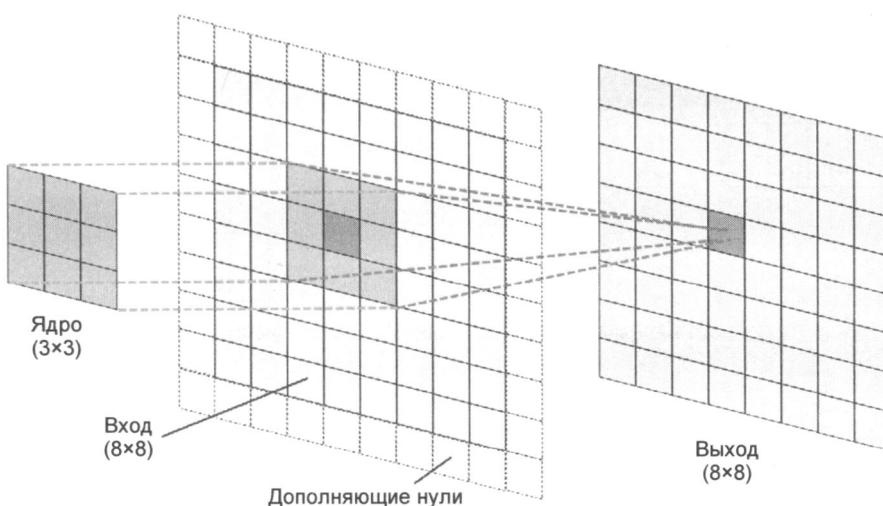


Рис. 14.5. Результат двумерной свертки

В следующем примере показано вычисление двумерной свертки между входной матрицей $X_{3 \times 3}$ и матрицей ядра $W_{3 \times 3}$ с использованием заполнения $p = (1, 1)$ и шага $s = (2, 2)$. В соответствии с указанным заполнением к каждой стороне входной матрицы добавляется один слой нулей, в результате чего получается заполненная (padded) матрица $X_{5 \times 5}^{\text{padded}}$ (рис. 14.6).

$$\begin{array}{c|ccccc}
 & & X & & \\
 \hline
 & 0 & 0 & 0 & 0 & 0 \\
 & 0 & 2 & 1 & 2 & 0 \\
 & 0 & 5 & 0 & 1 & 0 \\
 & 0 & 1 & 7 & 3 & 0 \\
 & 0 & 0 & 0 & 0 & 0
 \end{array} * \begin{array}{c|ccc}
 & W \\
 \hline
 0.5 & 0.7 & 0.4 \\
 0.3 & 0.4 & 0.1 \\
 0.5 & 1 & 0.5
 \end{array}$$

Рис. 14.6. Вычисление двумерной свертки между входной матрицей и матрицей ядра

Относительно исходного фильтра повернутый фильтр имеет вид:

$$W^r = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Учтите, что это вращение не является транспонированием матрицы. Чтобы получить повернутый фильтр в NumPy, мы можем написать `W_rot=W[::-1,::-1]`. Затем можем сдвинуть повернутую матрицу фильтра вдоль дополненной входной матрицы X^{padded} как скользящее окно и вычислить сумму поэлементного произведения, которая обозначена оператором \odot на рис. 14.7.

В результате мы получим матрицу Y размером 2×2 .

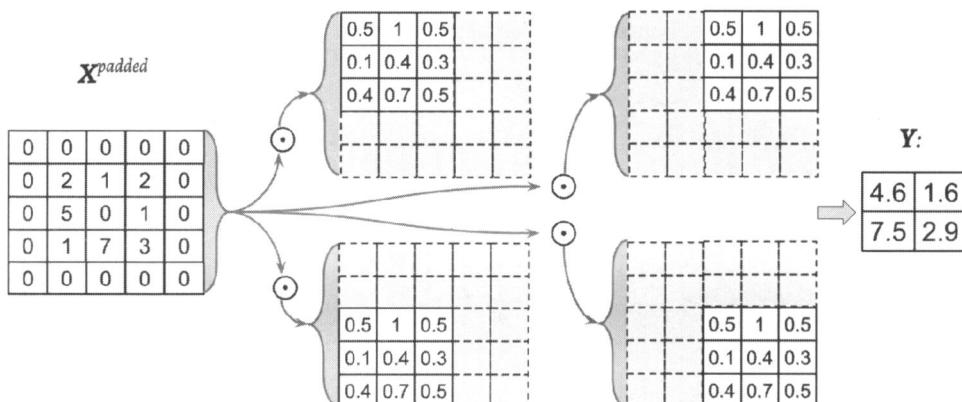


Рис. 14.7. Вычисление суммы поэлементного произведения

Давайте также реализуем двумерную свертку с помощью описанного ранее простого алгоритма. Пакет `scipy.signal` предоставляет способ вычисления двумерной свертки с помощью функции `scipy.signal.convolve2d`:

```

>>> import numpy as np
>>> import scipy.signal
>>> def conv2d(X, W, p=(0, 0), s=(1, 1)):
...     W_rot = np.array(W)[:, ::-1, ::-1]
...     X_orig = np.array(X)
...     n1 = X_orig.shape[0] + 2*p[0]
...     n2 = X_orig.shape[1] + 2*p[1]
...     X_padded = np.zeros(shape=(n1, n2))
    
```

```

...
    X_padded[p[0]:p[0]+X_orig.shape[0],
    ...
        p[1]:p[1]+X_orig.shape[1]] = X_orig
...
...
    res = []
    for i in range(0,
        int((X_padded.shape[0] - \
    ...     W_rot.shape[0])/s[0])+1, s[0]):
        res.append([])
        for j in range(0,
            int((X_padded.shape[1] - \
    ...     W_rot.shape[1])/s[1])+1, s[1]):
            X_sub = X_padded[i:i+W_rot.shape[0],
    ...
                    j:j+W_rot.shape[1]]
            res[-1].append(np.sum(X_sub * W_rot))
    ...
    return(np.array(res))
>>> X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
>>> W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]
>>> print('Реализация Conv2d:\n',
...       conv2d(X, W, p=(1, 1), s=(1, 1)))
Реализация Conv2d:
[[ 11. 25. 32. 13.]
 [ 19. 25. 24. 13.]
 [ 13. 28. 25. 17.]
 [ 11. 17. 14.  9.]]
>>> print('Результат SciPy:\n',
...       scipy.signal.convolve2d(X, W, mode='same'))
Результат SciPy:
[[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]

```



Эффективные алгоритмы вычисления свертки

Здесь мы предоставили буквальную реализацию вычисления двумерной свертки для лучшего понимания теоретической основы происходящего. Однако эта реализация очень неэффективна с точки зрения требований к памяти и вычислительной сложности. Поэтому ее не следует использовать в реальных приложениях с использованием нейросетей.

В большинстве инструментов, таких как PyTorch, матрицу фильтра на самом деле не врашают. Более того, в последние годы были разработаны гораздо более эффективные алгоритмы, использующие для вычисления сверток преобразование Фурье. Также важно отметить, что на практике в нейросетевых приложениях размер ядра свертки обычно намного меньше размера входного изображения.

Например, современные CNN обычно используют ядра размером 1×1 , 3×3 или 5×5 , для которых были разработаны эффективные алгоритмы, позволяющие выполнять сверточные операции намного эффективнее, — например, алгоритм минимальной фильтрации Винограда. Изучение этих алгоритмов выходит за рамки нашей книги, но если вы хотите узнать больше, рекомендуем прочитать книгу Эндрю Лавина и Скотта Грея «Fast Algorithms for Convolutional Neural Networks», которая находится в свободном доступе по адресу: <https://arxiv.org/abs/1509.09308>.

В следующем разделе мы обсудим подвыборку, или объединение, что является еще одной важной операцией, часто используемой в CNN.

14.1.3. Слои подвыборки

Подвыборка (subsampling) обычно применяется в двух формах операций объединения в CNN: по максимуму (max-pooling) и по среднему (mean-pooling, или average-pooling). Слой объединения обычно обозначают так: $P_{n_1 \times n_2}$. Здесь нижний индекс определяет размер окрестности (количество соседних пикселов в каждом измерении), где выполняется операция объединения. Мы называем такую окрестность *размером объединения* (pooling size).

Операция объединения показана на рис. 14.8. Здесь объединение по максимуму берет максимальное значение из окрестности пикселов, а объединение по среднему вычисляет их среднее значение.



Рис. 14.8. Пример объединения по максимальному и среднему значению

Операция объединения дает нам двойную выгоду:

- ◆ объединение (по максимуму) вводит локальную инвариантность. Это означает, что небольшие изменения в локальном окружении не меняют результат объединения. Следовательно, это помогает создавать признаки, которые более устойчивы к шуму во входных данных. Рассмотрим следующий пример, который показывает, что объединение по максимуму двух разных входных матриц X_1 и X_2 приводит к одному и тому же результату:

$$\begin{aligned}
 X_1 = & \left[\begin{array}{cccccc} 10 & 255 & 125 & 0 & 170 & 100 \\ 70 & 255 & 105 & 25 & 25 & 70 \\ 255 & 0 & 150 & 0 & 10 & 10 \\ 0 & 255 & 10 & 10 & 150 & 20 \\ 70 & 15 & 200 & 100 & 95 & 0 \\ 35 & 25 & 100 & 20 & 0 & 60 \end{array} \right] \quad \left. \right\} \text{max pooling } P_{2 \times 2} \rightarrow \left[\begin{array}{ccc} 255 & 125 & 170 \\ 255 & 150 & 150 \\ 70 & 200 & 95 \end{array} \right]; \\
 X_2 = & \left[\begin{array}{cccccc} 100 & 100 & 100 & 50 & 100 & 50 \\ 95 & 255 & 100 & 125 & 125 & 170 \\ 80 & 40 & 10 & 10 & 125 & 150 \\ 255 & 30 & 150 & 20 & 120 & 125 \\ 30 & 30 & 150 & 100 & 70 & 70 \\ 70 & 30 & 100 & 200 & 70 & 95 \end{array} \right]
 \end{aligned}$$

- ◆ объединение уменьшает размер объектов, что приводит к повышению эффективности вычислений. Кроме того, уменьшение количества признаков также может снизить риск переобучения.



Объединение с перекрытием и без перекрытия

Традиционно предполагается, что объединение происходит без перекрытия — т. е. для непересекающихся окрестностей, что можно сделать, установив параметр шага равным размеру объединения. Например, для объединяющего слоя без перекрытия $P_{n_1 \times n_2}$ требуется параметр шага $s = (n_1, n_2)$. Объединение с перекрытием происходит, если шаг меньше, чем размер объединения. Пример использования объединения с перекрытием в сверточной сети описан в статье «ImageNet Classification with Deep Convolutional Neural Networks» by A. Krizhevsky, I. Sutskever, and G. Hinton, 2012, которая находится в свободном доступе по адресу: <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>.

Хотя объединение по-прежнему является важной частью многих архитектур CNN, разработано несколько архитектур CNN без использования объединяющих слоев. Вместо того, чтобы использовать для уменьшения размера объекта объединяющие слои, исследователи используют сверточные слои с шагом 2.

В некотором смысле сверточный слой с шагом 2 можно рассматривать как объединяющий слой с обучаемыми весами. Если вас интересует эмпирическое сравнение различных архитектур CNN с объединяющими слоями и без них, мы рекомендуем прочитать следующую исследовательскую статью⁴.

14.2. Практическая реализация CNN

Итак, вы познакомились с основными функциональными блоками CNN. Принципы, на которых они основаны, на самом деле не сложнее, чем у традиционных многослойных нейросетей. Можно сказать, что самой важной операцией в традиционной нейросети является умножение матриц. Например, мы используем умножение матриц для вычисления предварительных активаций (или действующих входов), как $z = Wx + b$. Здесь x — это вектор-столбец (матрица $\mathbb{R}^{n \times 1}$), представляющий пиксели, а W — матрица весов, соединяющая входы пикселов с каждым скрытым узлом.

В CNN эту операцию заменяет операция свертки $Z = W * X + b$, где X — матрица, представляющая пиксели в области *высота × ширина*. В обоих случаях предварительные активации передаются функции активации для получения активации скрытого узла $A = \sigma(Z)$, где σ — функция активации. Кроме того, как вы помните, подвыборка — это еще один функциональный блок CNN, который может быть представлен в форме объединения, как было показано в предыдущем разделе.

⁴ См. «Striving for Simplicity: The All Convolutional Net» by Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Эта статья находится в свободном доступе по адресу: <https://arxiv.org/abs/1412.6806>.

14.2.1. Работа с несколькими входными или цветовыми каналами

Входные данные сверточного слоя могут содержать один или несколько двумерных массивов или матриц размером $N_1 \times N_2$ (например, высота и ширина изображения в пикселях). Эти матрицы $N_1 \times N_2$ называются *каналами*. Обычные реализации сверточных слоев предполагают представление входных данных в виде тензора ранга 3 — например, трехмерного массива $X_{N_1 \times N_2 \times C_m}$, где C_m — количество входных каналов. Допустим, на вход первого слоя CNN поступает изображение. Если это изображение цветное и представлено в цветовом режиме RGB, то $C_m = 3$ (красный, зеленый и синий цветовые RGB-каналы). Однако если изображение в градациях серого, то мы имеем $C_m = 1$, потому что на вход поступает только один канал со значениями интенсивности пикселов.



Чтение файла изображения

При работе с изображениями их можно считывать в массивы NumPy, используя тип данных `uint8` (8-битное целое число без знака), чтобы уменьшить загрузку памяти по сравнению, например, с 16-, 32- или 64-битными целочисленными типами.

8-битные целые числа без знака принимают значения в диапазоне [0, 255], что достаточно для хранения информации о пикселях в изображениях RGB, которые принимают значения в том же диапазоне.

В главе 12 вы видели, что PyTorch предоставляет модуль для загрузки/хранения и управления изображениями через `torchvision`. Напомним, как читать изображение (этот пример RGB-изображения находится в папке с примерами кода главы сопровождающего книгу файлового архива):

```
>>> import torch
>>> from torchvision.io import read_image
>>> img = read_image('example-image.png')
>>> print('Размер изображения:', img.shape)
Размер изображения: torch.Size([3, 252, 221])
>>> print('Количество каналов:', img.shape[0])
Количество каналов: 3
>>> print('Тип данных изображения:', img.dtype)
Тип данных изображения: torch.uint8
>>> print(img[:, 100:102, 100:102])
tensor([[ [179, 182],
          [180, 182]],

         [[134, 136],
          [135, 137]],

         [[110, 112],
          [111, 113]]], dtype=torch.uint8)
```

Учтите, что при использовании `torchvision` тензоры входного и выходного изображения имеют формат `Tensor[channels, image_height, image_width]`.

Теперь, когда вы знакомы со структурой входных данных, возникает следующий вопрос: как включить несколько входных каналов в операцию свертки, которую мы

обсуждали в предыдущих разделах? Ответ очень прост: мы выполняем операцию свертки для каждого канала отдельно, а затем складываем результаты вместе, используя матричное суммирование. Свертка, связанная с каждым каналом c , имеет собственную матрицу ядра вида $W[:, :, c]$.

Общий результат предварительной активации рассчитывается по следующей формуле:

$$\text{При входной выборке } X_{n_1 \times n_2 \times C_{in}}, \text{ матрице ядра } W_{m_1 \times m_2 \times C_{in}}, \text{ и величине смещения } b \Rightarrow \begin{cases} Z^{Conv} = \sum_{c=1}^{C_{in}} W[:, :, c] * X[:, :, c] \\ \text{Предактивация: } Z = Z^{Conv} + b_c \\ \text{Карта признаков: } A = \sigma(Z) \end{cases}$$

Конечным результатом A является карта признаков. Обычно сверточный слой CNN имеет более одной карты признаков. Если мы используем несколько карт признаков, тензор ядра становится четырехмерным: $\text{ширина} \times \text{высота} \times C_{in} \times C_{out}$. Здесь $\text{ширина} \times \text{высота}$ — это размер ядра, C_{in} — количество входных каналов, а C_{out} — количество выходных карт признаков. Включим количество выходных карт объектов в предыдущую формулу и обновим ее следующим образом:

$$\text{При входной выборке } X_{n_1 \times n_2 \times C_{in}}, \text{ матрице ядра } W_{m_1 \times m_2 \times C_{in} \times C_{out}}, \text{ и векторе смещения } b_{C_{out}} \Rightarrow \begin{cases} Z^{Conv}[:, :, k] = \sum_{c=1}^{C_{in}} W[:, :, c, k] * X[:, :, c] \\ Z[:, :, k] = Z^{Conv}[:, :, k] + b[k] \\ A[:, :, k] = \sigma(Z[:, :, k]) \end{cases}$$

Завершая обсуждение вычисления сверток в контексте нейросетей, рассмотрим пример на рис. 14.9, где показан сверточный слой, за которым следует слой объединения. В этом примере есть три входных канала и четырехмерный тензор ядра. Каждая матрица ядра обозначается как $m_1 \times m_2$, и их три — по одной на каждый входной канал. Всего имеется пять таких ядер, на которые приходится пять выходных карт признаков. Наконец, есть объединяющий слой для подвыборки карт объектов.

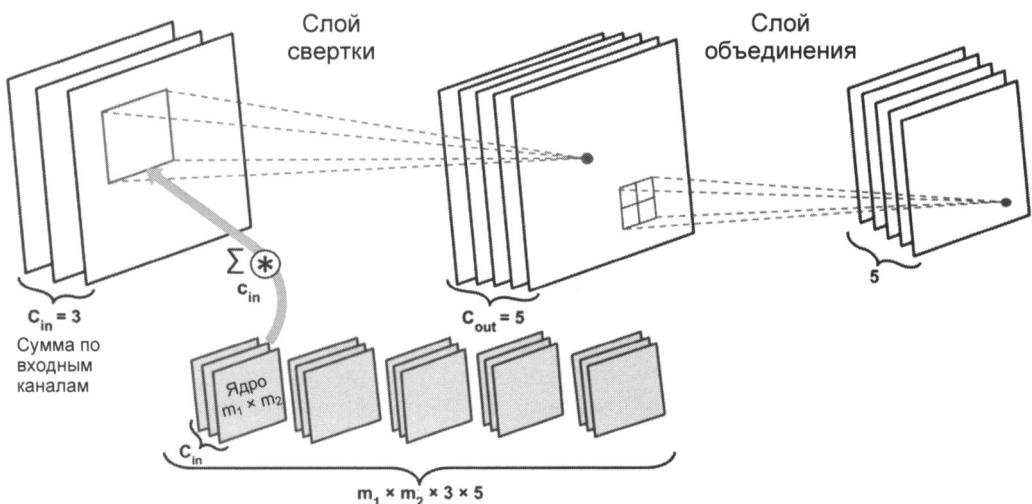


Рис. 14.9. Реализация CNN



Сколько обучаемых параметров нейросети в приведенном примере?

Чтобы проиллюстрировать преимущества свертки, совместного использования параметров и разреженной связи, рассмотрим этот пример подробнее. Сверточный слой в сети, показанной на рис. 14.9, — четырехмерный тензор. То есть с ядром связано $m_1 \times m_2 \times 3 \times 5$ параметров. Кроме того, для каждой выходной карты объектов сверточного слоя существует вектор смещения. Размер вектора смещения равен 5. Слои объединения не имеют обучаемых параметров — следовательно, мы можем записать такое выражение:

$$m_1 \times m_2 \times 3 \times 5 + 5.$$

Если входной тензор имеет размер $n_1 \times n_2 \times 3$, и мы предположим, что свертка выполняется в режиме равного заполнения, то выходные карты признаков будут иметь размер $n_1 \times n_2 \times 5$.

Если использовать полносвязный слой вместо сверточного, это число будет намного больше. В случае полносвязного слоя количество параметров матрицы весов для того же количества выходных узлов было бы следующим:

$$(n_1 \times n_2 \times 3) \times (n_1 \times n_2 \times 5) = (n_1 \times n_2)^2 \times 3 \times 5.$$

Кроме того, размер вектора смещения составляет $n_1 \times n_2 \times 5$ (один элемент смещения для каждого узла вывода). Учитывая, что $m_1 < n_1$ и $m_2 < n_2$, мы видим, что разница в количестве обучаемых параметров значительна.

Наконец, как уже упоминалось, операции свертки обычно выполняются путем обработки входного изображения с несколькими цветовыми каналами как наборами матриц — т. е. мы выполняем свертку для каждой матрицы отдельно, а затем складываем результаты, как показано на рис. 14.9. Однако операцию свертки также можно распространить и на трехмерное пространство, если вы работаете с трехмерными наборами данных⁵.

В следующем разделе мы поговорим о регуляризации нейронных сетей.

14.2.2. Регуляризация L2 и прореживание

Выбор размера сети, независимо от того, имеем ли мы дело с традиционной полносвязной нейросетью или CNN, всегда был сложной проблемой, поскольку для достижения достаточно хорошей производительности размер весовой матрицы и количество слоев должны быть правильно подобраны.

В главе 13 мы отмечали, что простая сеть без скрытого слоя может построить только линейную разделяющую границу, чего недостаточно для решения задачи XOR или другой нелинейной задачи. Емкость сети соотносится с уровнем сложности функции, которую она может научиться аппроксимировать. Небольшие сети или сети с относительно небольшим количеством параметров имеют низкую емкость и, как следствие, не всегда могут выучить базовую структуру сложных наборов данных. Однако слишком большие сети склонны к переобучению, когда сеть просто запоминает обучающие данные и очень хорошо работает с обучающим набором, но плохо справляется с незнакомыми данными.

⁵ Например, как показано в статье «VoxNet: A 3D Convolutional Neural Network for Real-Time Object Recognition» by Daniel Maturana and Sebastian Scherer, 2015, которая доступна по адресу: https://www.ri.cmu.edu/pub_files/2015/9/voxnet_maturana_scherer_iros15.pdf.

мым тестовым набором. Начиная работать над решением реальной задачи машинного обучения, мы априори не знаем, насколько большой должна быть сеть.

Один из способов решения этой проблемы — построить сеть с относительно большой емкостью (на практике нам нужно выбрать емкость, немного большую, чем необходимо), чтобы она хорошоправлялась с обучающим набором данных. Затем, чтобы предотвратить переобучение, мы можем применить одну или несколько стратегий регуляризации для достижения хорошей обобщающей способности модели на незнакомых данных.

В главах 3 и 4 мы познакомились с регуляризацией L1 и L2. Оба метода могут предотвратить или уменьшить эффект переобучения, добавляя к потере специальный штраф, что приводит к уменьшению весовых параметров во время обучения. Хотя методы L1 и L2 также можно использовать для регуляризации нейросетей со скрытыми слоями (причем L2 является более распространенным вариантом), существуют и другие методы регуляризации нейросетей, такие как *прореживание* (dropout)⁶, которые мы обсудим в этом разделе. Но прежде, чем перейти к обсуждению прореживания, отметим, что для использования регуляризации L2 в сверточной или полно связной сети (напомним, полно связные слои реализуются через `torch.nn.Linear` в PyTorch) вы можете просто добавить штраф L2 определенного слоя к функции потерь в PyTorch следующим образом:

```
>>> import torch.nn as nn
>>> loss_func = nn.BCELoss()
>>> loss = loss_func(torch.tensor([0.9]), torch.tensor([1.0]))
>>> l2_lambda = 0.001
>>> conv_layer = nn.Conv2d(in_channels=3,
...                         out_channels=5,
...                         kernel_size=5)
>>> l2_penalty = l2_lambda * sum(
...     [(p**2).sum() for p in conv_layer.parameters()])
...
>>> loss_with_penalty = loss + l2_penalty
>>> linear_layer = nn.Linear(10, 16)
>>> l2_penalty = l2_lambda * sum(
...     [(p**2).sum() for p in linear_layer.parameters()])
...
>>> loss_with_penalty = loss + l2_penalty
```



Затухание веса по сравнению с регуляризацией L2

Альтернативным способом применения регуляризации L2 является установка для параметра `weight_decay` в оптимизаторе PyTorch положительного значения — например:

```
optimizer = torch.optim.SGD(
    model.parameters(),
    weight_decay=l2_lambda,
    ...
)
```

⁶ Прореживание также называют *отсевом*, *отбрасыванием* или просто *дропаутом*. — Прим. пер.

Хотя регуляризация L2 и `weight_decay` не являются строго идентичными, можно показать, что они эквивалентны при использовании оптимизаторов стохастического градиентного спуска (SGD). Заинтересованные читатели могут найти дополнительную информацию в статье «Decoupled Weight Decay Regularization» by Ilya Loshchilov and Frank Hutter, 2019, которая находится в свободном доступе по адресу: <https://arxiv.org/abs/1711.05101>.

В последние годы прореживание стало популярным методом регуляризации глубоких нейронных сетей, позволяющим избежать переобучения, и тем самым улучшить обобщающую способность⁷. Прореживание обычно применяется к скрытым узлам более высоких слоев и работает следующим образом: на этапе обучения нейросети часть скрытых узлов случайным образом отбрасывают на каждой итерации с вероятностью p_{drop} (или сохраняют с вероятностью $p_{keep} = 1 - p_{drop}$). Эта вероятность отбрасывания определяется пользователем, и чаще всего выбирают $p = 0.5$, как обсуждалось в упомянутой только что статье Нитиша Шриваставы и его коллег. При отбрасывании определенной доли входных нейронов веса, связанные с оставшимися нейронами, пересчитываются для компенсации выпавших нейронов.

Эффект этого случайного исключения заключается в том, что сеть вынуждена изучать избыточное представление данных. Сеть не может полагаться на активацию какого-либо конкретного набора скрытых узлов, поскольку они могут быть отключены в любой момент во время обучения, и вынуждена извлекать из данных более общие и надежные закономерности.

Как показывает опыт, случайное прореживание эффективно предотвращает переобучение. На рис. 14.10 показан пример применения прореживания с вероятностью $p = 0.5$ на этапе обучения, при котором половина нейронов становится неактивной случайным образом (отбрасываемые узлы выбираются случайным образом при каждом прямом

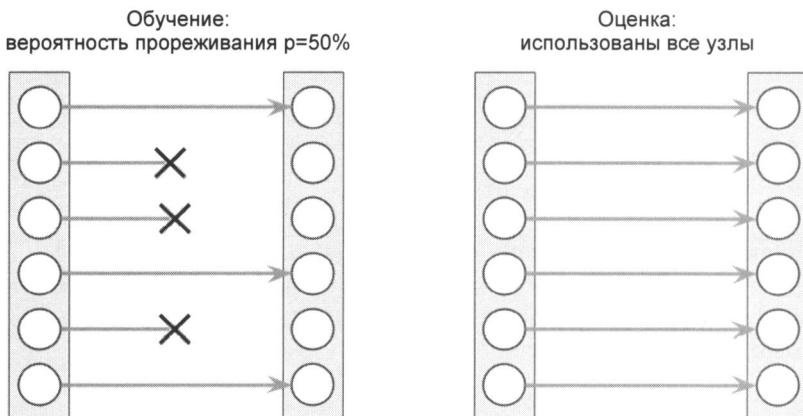


Рис. 14.10. Применение прореживания на этапе обучения

⁷ См. «Dropout: A Simple Way to Prevent Neural Networks from Overfitting» by N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, Journal of Machine Learning Research 15.1, p. 1929–1958, 2014.
<http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>.

проходе обучения). Однако во время прогнозирования все нейроны будут участвовать в вычислении предварительных активаций следующего слоя.

Следует помнить один важный момент: узлы могут выпадать случайным образом только во время обучения, тогда как на этапе оценки (вывода) должны быть задействованы все скрытые узлы (например, $p_{drop} = 0$ или $p_{keep} = 1$). Чтобы гарантировать, что все активации имеют один и тот же масштаб как во время обучения, так и во время прогнозирования, активации задействованных нейронов должны быть соответствующим образом масштабированы (например, путем уменьшения активации вдвое, если вероятность отсева была установлена равной $p = 0.5$).

Однако, поскольку постоянно масштабировать активации при прогнозировании неудобно, PyTorch и другие фреймворки масштабируют активации во время обучения (например, удваивая их, если вероятность отсева была установлена равной $p = 0.5$). Этот подход обычно называют *обратным прореживанием* (*inverse dropout*).

Хотя взаимосвязь не очевидна сразу, прореживание можно интерпретировать как консенсус (усреднение) ансамбля моделей. Как было показано в главе 7, при ансамблевом обучении мы обучаем несколько моделей независимо друг от друга. Затем во время прогнозирования используем консенсус всех обученных моделей. Мы уже знаем, что ансамбли моделей работают лучше, чем отдельные модели. Однако в глубоком обучении как обучение нескольких моделей, так и сбор и усреднение выходных данных нескольких моделей требуют больших вычислительных ресурсов. Прореживание предлагает обходной путь с эффективным способом одновременного обучения нескольких моделей и вычисления их средних прогнозов во время тестирования или прогнозирования.

Как мы уже сказали, взаимосвязь между ансамблями моделей и прореживанием не очевидна. Да и надо еще учитывать, что при прореживании у нас получается другая модель для каждой мини-партии (из-за случайной установки весов на ноль во время каждого прямого прохода).

Затем, перебирая мини-пакеты, мы, по существу, выбираем модели $M = 2^h$, где h — количество скрытых узлов.

Однако важный аспект, который отличает прореживание от обычного ансамбля моделей, заключается в том, что мы совместно используем веса этих условных «разных моделей», что можно рассматривать как форму регуляризации. Затем во время «вывода» (например, прогнозирования меток в тестовом наборе данных) мы можем усреднить все эти разные модели, которые выбрали во время обучения. Впрочем, это очень дорого.

Таким образом, усреднение моделей, т. е. вычисление среднего геометрического вероятности принадлежности к классу, возвращаемой моделью i , можно выполнить следующим образом:

$$p_{\text{Ensemble}} = \left[\prod_{j=1}^M p^{(i)} \right]^{\frac{1}{M}}.$$

Хитрость прореживания заключается в том, что это среднее геометрическое ансамблей моделей (здесь — моделей M) можно аппроксимировать путем масштабирования прогнозов последней (или окончательной) модели, выбранной во время обучения, с коэффициентом $1/(1-p)$, что обходится намного дешевле, чем явное вычисление среднего

геометрического с использованием приведенного уравнения. (На самом деле приближение точно эквивалентно истинному среднему геометрическому значению, если мы рассматриваем линейные модели.)

14.2.3. Функции потерь для задач классификации

В главе 12 мы рассматривали различные функции активации — такие как ReLU, сигмоидная и тангенциальная. Некоторые из них (в частности, ReLU) в основном используются в промежуточных (скрытых) слоях нейросети, чтобы внести в модель нелинейность. Но другие — такие как `sigmoid` (для бинарных классификаторов) и `softmax` (для многоклассовых классификаторов) — добавляют на последнем (выходном) уровне, чтобы в качестве выходных данных модели получить вероятности принадлежности к классу. Если в выходной слой не включена сигмоидная или softmax-активация, то модель будет вычислять логиты вместо вероятностей принадлежности к классу.

Исходя из формы классификации (бинарная или многоклассовая) и типа вывода (логиты или вероятности), необходимо выбрать для обучения модели соответствующую функцию потерь. Бинарная перекрестная энтропия — это функция потерь для бинарной классификации (с одним выходным узлом), а категориальная перекрестная энтропия — это функция потерь для многоклассовой классификации. В модуле `torch.nn` функция потерь категориальной перекрестной энтропии принимает эталонные метки как целевые числа (например, $y = 2$, из трех классов: 0, 1 и 2).

На рис. 14.11 показаны две функции потерь, доступные в `torch.nn` для работы с обоими вариантами классификации: бинарной и многоклассовой с целочисленными метками. Обе функции потерь также имеют возможность получать прогнозы в виде логитов или вероятностей принадлежности к классу.

Функции потерь	Варианты классификации	Пример Использование вероятностей	Пример Использование логитов
<code>BCELoss</code> или <code>BCEWithLogitsLoss</code>	Бинарная	<code>BCELoss</code> <code>y_true:</code> 1 <code>y_pred:</code> 0.69	<code>BCEWithLogitsLoss</code> <code>y_true:</code> 1 <code>y_pred:</code> 0.8
<code>NLLLoss</code> или <code>CrossEntropyLoss</code>	Много-классовая	<code>NLLLoss</code> <code>y_true:</code> 2 <code>y_pred:</code> 0.30 0.15 0.55	<code>CrossEntropyLoss</code> <code>y_true:</code> 2 <code>y_pred:</code> 1.5 0.8 2.1

Рис. 14.11. Два примера функций потерь в PyTorch

Нужно сказать, что расчет потери перекрестной энтропии путем использования логитов, а не вероятностей принадлежности к классу обычно предпочтительнее из соображений численной стабильности. В случае бинарной классификации мы можем либо предоставить логиты в качестве входных данных для функции потерь `nn.BCEWithLogitsLoss()`, либо вычислить вероятности на основе логитов и передать их

функции потерь nn.BCELoss(). В случае многоклассовой классификации мы можем либо предоставить логиты в качестве входных данных для функции потерь nn.CrossEntropyLoss(), либо вычислить логарифмические вероятности на основе логитов и передать их функции потерь отрицательного логарифмического правдоподобия nn.NLLLoss().

В следующем коде показано, как использовать эти функции потерь с двумя разными форматами, где в качестве входных данных задают либо логиты, либо вероятности принадлежности к классу:

```
>>> ##### Бинарная перекрестная энтропия
>>> logits = torch.tensor([0.8])
>>> probas = torch.sigmoid(logits)
>>> target = torch.tensor([1.0])
>>> bce_loss_fn = nn.BCELoss()
>>> bce_logits_loss_fn = nn.BCEWithLogitsLoss()
>>> print(f'BCE (w Probas): {bce_loss_fn(probas, target):.4f}')
BCE (w Probas): 0.3711
>>> print(f'BCE (w Logits): '
...     f'{bce_logits_loss_fn(logits, target):.4f}')
BCE (w Logits): 0.3711
>>> ##### Категориальная перекрестная энтропия
>>> logits = torch.tensor([[1.5, 0.8, 2.1]])
>>> probas = torch.softmax(logits, dim=1)
>>> target = torch.tensor([2])
>>> cce_loss_fn = nn.NLLLoss()
>>> cce_logits_loss_fn = nn.CrossEntropyLoss()
>>> print(f'CCE (w Logits): '
...     f'{cce_logits_loss_fn(logits, target):.4f}')
CCE (w Probas): 0.5996
>>> print(f'CCE (w Probas): '
...     f'{cce_loss_fn(torch.log(probas), target):.4f}')
CCE (w Logits): 0.5996
```

Нужно отметить, что иногда вы можете столкнуться с ситуацией, когда для бинарной классификации используется потеря категориальной перекрестной энтропии. Обычно, когда у нас есть задача бинарной классификации, модель возвращает одно выходное значение для каждого примера. Мы интерпретируем этот единственный результат модели как вероятность положительного класса (например, класса 1), $P(\text{class} = 1|x)$. В задаче бинарной классификации подразумевается, что $P(\text{class} = 0|x) = 1 - P(\text{class} = 1|x)$, следовательно, нам не нужен второй выход, чтобы получить вероятность отрицательного класса. Однако иногда на практике специалисты предпочитают возвращать два результата для каждого обучающего примера и интерпретировать их как вероятности каждого класса по отдельности: $P(\text{class} = 0|x)$ против $P(\text{class} = 1|x)$. В таком случае рекомендуется использовать функцию softmax (вместо логистической сигмоиды) для нормализации выходных данных (чтобы их сумма равнялась 1), а подходящей функцией потерь тогда будет категориальная перекрестная энтропия.

14.3. Реализация глубокой CNN с использованием PyTorch

Как вы помните, в главе 13 мы решали задачу распознавания рукописных цифр с помощью модуля `torch.nn`. Напомним также, что мы достигли точности около 95.6%, используя нейросеть с двумя линейными скрытыми слоями.

Теперь мы реализуем CNN и посмотрим, сможет ли она достичь более высокой точности прогнозирования по сравнению с предыдущей моделью для классификации рукописных цифр. К слову, полносвязные слои, с которыми мы работали в главе 13, хорошо справились с этой задачей. Однако в некоторых приложениях — таких как чтение рукописных номеров банковских счетов — даже небольшие ошибки могут дорого обойтись. Поэтому крайне важно свести эту погрешность к минимуму.

14.3.1. Архитектура многослойной CNN

Архитектура сети, которую мы собираемся реализовать, показана на рис. 14.12. Входные данные представляют собой изображения в градациях серого 28×28 . С учетом количества каналов (которое равно 1 для изображений в градациях серого) и пакета входных изображений размеры входного тензора будут равны (*размер пакета* $\times 28 \times 28 \times 1$).

Входные данные проходят через два сверточных слоя с размером ядра 5×5 . Первая свертка имеет 32 карты выходных признаков, а вторая — 64 карты. За каждым слоем свертки следует слой подвыборки в форме операции объединения по максимуму $P_{2 \times 2}$. Затем полносвязный слой передает выходные данные второму полносвязному слою, который действует как последний выходной слой `softmax`.



Рис. 14.12. Глубокая CNN

Размерности тензоров в каждом слое следующие:

- ◆ Ввод: [*размер пакета* $\times 28 \times 28 \times 1$];
- ◆ Свертка_1: [*размер пакета* $\times 28 \times 28 \times 32$];
- ◆ Объединение_1: [*размер пакета* $\times 14 \times 14 \times 32$];
- ◆ Свертка_2: [*размер пакета* $\times 14 \times 14 \times 64$];
- ◆ Объединение_2: [*размер пакета* $\times 7 \times 7 \times 64$];

- ◆ Полносвязный_1: [размер пакета×1024];
- ◆ Полносвязный_2 и слой softmax: [размер пакета×10].

Для сверточных ядер мы используем `stride=1`, чтобы входные размеры сохранялись в результирующих картах признаков. В объединяющих слоях задаем `kernel_size=2` — для подвыборки изображения и уменьшения размера выходных карт признаков. И реализуем эту сеть с помощью модуля `torch.nn`.

14.3.2. Загрузка и предварительная обработка данных

Начнем с загрузки набора данных MNIST с помощью модуля `torchvision` и создадим наборы для обучения и тестирования, как мы это делали в главе 13:

```
>>> import torchvision
>>> from torchvision import transforms
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor()
... ])
>>> mnist_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=True
... )
>>> from torch.utils.data import Subset
>>> mnist_valid_dataset = Subset(mnist_dataset,
...                                 torch.arange(10000))
>>> mnist_train_dataset = Subset(mnist_dataset,
...                                 torch.arange(
...                                     10000, len(mnist_dataset)
...                               ))
>>> mnist_test_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=False,
...     transform=transform, download=False
... )
```

Набор данных MNIST поставляется с предварительно заданной схемой разделения его для обучения и тестирования, но нам еще нужно выделить из тестовой части поднабор для валидации. Поэтому мы возьмем для валидации первые 10 тыс. обучающих примеров. Обратите внимание, что изображения не сортируются по метке класса, поэтому нам не нужно беспокоиться о том, не относятся ли изображения в наборе для валидации к одному и тому же классу.

Далее мы создадим загрузчик данных в виде пакетов из 64 изображений для обучающего и валидационного набора соответственно:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(mnist_train_dataset,
...                         batch_size,
...                         shuffle=True)
```

```
>>> valid_dl = DataLoader(mnist_valid_dataset,
...                         batch_size,
...                         shuffle=False)
```

Признаки, которые мы читаем, имеют значения в диапазоне [0, 1]. Кроме того, мы уже конвертировали изображения в тензоры. Метки являются целыми числами от 0 до 9, представляющими десять цифр. Следовательно, нам не нужно делать какое-либо масштабирование или дальнейшее преобразование.

Завершив подготовку набора данных, мы готовы создать только что описанную CNN.

14.3.3. Реализация CNN с использованием модуля `torch.nn`

Для реализации CNN в PyTorch воспользуемся классом `Sequential` модуля `torch.nn` для объединения различных слоев — таких как свертка, объединение и прореживание, а также полносвязных слоев. Модуль `torch.nn` предоставляет классы для каждого из них: `nn.Conv2d` — для двумерного слоя свертки, `nn.MaxPool2d` и `nn.AvgPool2d` — для подвыборки (объединение по максимуму и по среднему) и `nn.Dropout` — для регуляризации с помощью прореживания. Мы рассмотрим каждый из этих классов более подробно.

Настройка слоев CNN в PyTorch

Для построения слоя с помощью класса `Conv2d` необходимо указать количество выходных каналов (что эквивалентно количеству выходных карт признаков или количеству выходных фильтров) и размеры ядра.

Кроме того, есть необязательные параметры, которые можно использовать для настройки слоя свертки. Наиболее часто применяются `stride` (со значением по умолчанию 1 для обоих измерений: x , y) и `padding`, определяющий ширину неявного заполнения в обоих измерениях. Дополнительные параметры конфигурации приведены в официальной документации, доступной по адресу: <https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html>.

Стоит отметить, что обычно, когда мы читаем изображение, размерность по умолчанию для каналов — это первое измерение тензорного массива (или второе — учитывая наличие размера пакета). В целом это называется форматом NCHW, где N обозначает количество изображений в пакете, C — каналы, а H и W — высоту и ширину соответственно.

Это важный момент, потому что класс `Conv2D` предполагает, что входные данные по умолчанию имеют формат NCHW. (Другие инструменты, такие как TensorFlow, используют формат NHWC.) Так что, если вы столкнетесь с данными, каналы которых размещены в последнем измерении, вам потребуется поменять местами оси в своих данных, чтобы переместить каналы в первое измерение (или второе — с учетом наличия размера пакета). После создания слоя его можно вызвать, передав четырехмерный тензор, причем первое измерение зарезервировано для пакета примеров, второе — соответствует каналу; а два других измерения являются пространственными.

В соответствии с архитектурой модели CNN, которую мы хотим построить, за каждым слоем свертки следует слой объединения для операции подвыборки (уменьшения раз-

мера карт признаков). Классы `MaxPool2d` и `AvgPool2d` создают слои объединения по максимуму и по среднему соответственно. Аргумент `kernel_size` определяет размер окна (или окрестности), которое будет использоваться для вычисления максимального или среднего значения. Кроме того, параметр `stride` можно использовать для настройки объединяющего слоя, как было сказано ранее.

Наконец, класс `Dropout` создает слой прореживания для регуляризации с учетом аргумента `p`, который служит для определения вероятности p_{drop} отбрасывания входных узлов во время обучения, что мы тоже обсуждали ранее. При вызове этого слоя его поведением можно управлять с помощью функций `model.train()` и `model.eval()`, позволяющих указать, что вызов выполняется для обучения или для вывода соответственно. При использовании прореживания переключение этих двух режимов имеет решающее значение для обеспечения правильного поведения модели — узлы отбрасываются случайным образом только во время обучения, а не во время оценки или вывода.

Создание CNN в PyTorch

Теперь, когда вы познакомились с необходимыми классами, можно приступить к созданию модели CNN, показанной на рис. 14.12. В следующем коде мы воспользуемся классом `Sequential` и добавим с его помощью слои свертки и объединения:

```
>>> model = nn.Sequential()
>>> model.add_module(
...     'conv1',
...     nn.Conv2d(
...         in_channels=1, out_channels=32,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu1', nn.ReLU())
>>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>>> model.add_module(
...     'conv2',
...     nn.Conv2d(
...         in_channels=32, out_channels=64,
...         kernel_size=5, padding=2
...     )
... )
>>> model.add_module('relu2', nn.ReLU())
>>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))
```

Здесь мы добавили в модель два слоя свертки, использовав для каждого сверточного слоя ядро размером 5×5 и `padding=2`. Как вы знаете, применение равного режима заполнения сохраняет пространственные (вертикальные и горизонтальные) размеры карт признаков, так что входные и выходные данные имеют одинаковую высоту и ширину (а количество каналов может различаться только с точки зрения количества использованных фильтров). Как упоминалось ранее, пространственное измерение выходной карты признаков рассчитывается следующим образом:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1,$$

где n — пространственное измерение входной карты признаков, а p , m и s обозначают заполнение, размер ядра и шаг соответственно. Следовательно, чтобы добиться $o = i$, необходимо использовать $p = 2$.

Слои объединения по максимуму с размером ядра 2×2 и шагом 2 уменьшают пространственные размеры вдвое. (Обратите внимание, что если параметр `stride` не указан в `MaxPool2D`, по умолчанию он устанавливается равным размеру ядра.)

Хотя на этом этапе мы можем рассчитать размер карт признаков вручную, PyTorch предоставляет нам удобный метод для расчета:

```
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 64, 7, 7])
```

Передав размеры входа в виде кортежа $(4, 1, 28, 28)$ (4 изображения в пакете, 1 канал и размер изображения 28×28), определенного для этого примера, мы рассчитали, что выход имеет форму $(4, 64, 7, 7)$, что указывает на карты признаков с 64 каналами и пространственным размером 7×7 . Первое измерение соответствует размеру пакета, для которого мы использовали произвольное значение 4.

Следующий слой, который нужно добавить, — это полносвязный слой для реализации классификатора поверх наших сверточных слоев и слоев объединения. Входные данные для этого слоя должны иметь ранг 2, т. е. форму [размер пакета \times входные узлы]. Следовательно, нам нужно снизить размерность выхода предыдущих слоев, чтобы выполнить это требование для полносвязного слоя:

```
>>> model.add_module('flatten', nn.Flatten())
>>> x = torch.ones((4, 1, 28, 28))
>>> model(x).shape
torch.Size([4, 3136])
```

Как следует из просмотра размерности в последней строке, входные размеры для полносвязного слоя настроены правильно. Далее мы добавим два полносвязных слоя со слоем прореживания между ними:

```
>>> model.add_module('fc1', nn.Linear(3136, 1024))
>>> model.add_module('relu3', nn.ReLU())
>>> model.add_module('dropout', nn.Dropout(p=0.5))
>>> model.add_module('fc2', nn.Linear(1024, 10))
```

Последний полносвязный слой с именем 'fc2' имеет 10 выходных узлов для 10 меток классов в наборе данных MNIST. На практике мы обычно используем активацию `softmax` для получения вероятностей принадлежности к классам каждого входного примера, предполагая, что классы являются взаимоисключающими, поэтому суммы вероятностей для каждого примера равны 1. Однако функция `softmax` уже используется внутри реализации `CrossEntropyLoss` в PyTorch, поэтому нет необходимости добавлять ее в явном виде после выходного слоя. Следующий код создаст функцию потерь и оптимизатор для модели:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```



Оптимизатор Adam

В этом примере реализации для обучения модели CNN мы использовали класс `torch.optim.Adam`. Оптимизатор Adam — это надежный метод оптимизации на основе градиента, подходящий для задач невыпуклой оптимизации и машинного обучения. Он создан на основе двух популярных методов оптимизации: RMSProp и AdaGrad.

Ключевое преимущество оптимизатора Adam заключается в выборе размера шага обновления, полученного методом скользящего среднего из значений моментов градиента. Вы можете узнать больше об оптимизаторе Adam из статьи «*Adam: A Method for Stochastic Optimization*» by Diederik P. Kingma and Jimmy Lei Ba, 2014. Статья находится в свободном доступе по адресу: <https://arxiv.org/abs/1412.6980>.

Теперь мы можем обучить модель, определив функцию `train`:

```
>>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)
...             loss = loss_fn(pred, y_batch)
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item()*y_batch.size(0)
...             is_correct = (
...                 torch.argmax(pred, dim=1) == y_batch
...                 ).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...             loss_hist_train[epoch] /= len(train_dl.dataset)
...             accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...             model.eval()
...             with torch.no_grad():
...                 for x_batch, y_batch in valid_dl:
...                     pred = model(x_batch)
...                     loss = loss_fn(pred, y_batch)
...                     loss_hist_valid[epoch] += \
...                         loss.item()*y_batch.size(0)
...                     is_correct = (
...                         torch.argmax(pred, dim=1) == y_batch
...                         ).float()
...                     accuracy_hist_valid[epoch] += is_correct.sum()
...                     loss_hist_valid[epoch] /= len(valid_dl.dataset)
...                     accuracy_hist_valid[epoch] /= len(valid_dl.dataset)
...
...             print(f'Epoch {epoch+1}/{num_epochs} | Loss: {loss_hist_train[epoch]} | Accuracy: {accuracy_hist_train[epoch]}')
...     print(f'Training completed! Final Loss: {loss_hist_train[-1]} | Final Accuracy: {accuracy_hist_train[-1]}')
... 
```

```

...
    print(f'Точность эпохи {epoch+1}: '
...
        f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
        f'{accuracy_hist_valid[epoch]:.4f}')
...
    return loss_hist_train, loss_hist_valid, \
...
        accuracy_hist_train, accuracy_hist_valid

```

Использование для обучения указанных настроек: `model.train()` и оценки `model.eval()` — автоматически установит режим для слоя прореживания и соответствующим образом перемасштабирует скрытые веса, так что нам вообще не нужно об этом беспокоиться. Далее мы обучим эту модель CNN с использованием валидационного набора, который мы создали для отслеживания процесса обучения:

```

>>> torch.manual_seed(1)
>>> num_epochs = 20
>>> hist = train(model, num_epochs, train_dl, valid_dl)
Точность эпохи 1: 0.9503 val_accuracy: 0.9802
...
Точность эпохи 9: 0.9968 val_accuracy: 0.9892
...
Точность эпохи 20: 0.9979 val_accuracy: 0.9907

```

По истечении 20 эпох мы можем визуализировать кривые обучения (рис. 14.13):

```

>>> import matplotlib.pyplot as plt
>>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist[0], '-o', label='Потери при обучении')
>>> ax.plot(x_arr, hist[1], '--<', label='Потери при валидации')
>>> ax.legend(fontsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist[2], '-o', label='Точность при обучении')
>>> ax.plot(x_arr, hist[3], '--<', label='Точность при валидации')
...
    label='Точность при валидации')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Точность', size=15)
>>> plt.show()

```

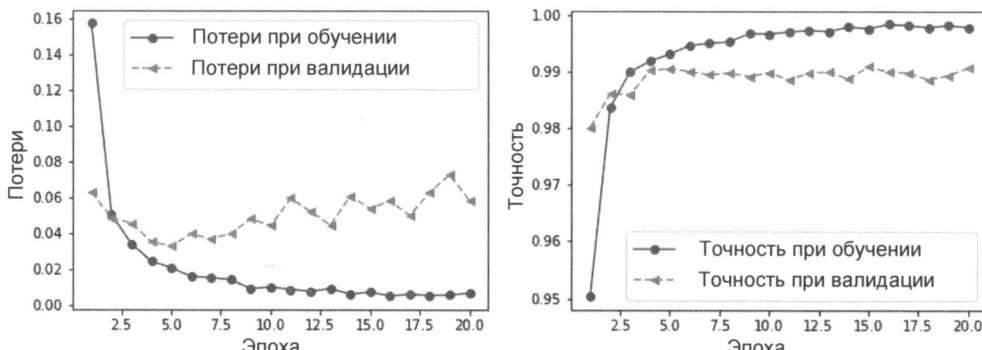


Рис. 14.13. Графики потерь и точности при обучении и валидации

Теперь оценим обученную модель на тестовом наборе данных:

```
>>> pred = model(mnist_test_dataset.data.unsqueeze(1) / 255.)
>>> is_correct = (
...     torch.argmax(pred, dim=1) == mnist_test_dataset.targets
... ).float()
>>> print(f'Точность при тестировании: {is_correct.mean():.4f}')
Точность при тестировании: 0.9914
```

Модель CNN достигает точности 99.07%. Помните, что в *главе 13* мы получили примерно 95-процентную точность, используя только полносвязные слои, а не сверточные.

Наконец, мы можем получить результаты прогнозирования в виде вероятностей принадлежности к классу и преобразовать их в прогнозируемые метки, используя для поиска элемента с максимальной вероятностью функцию `torch.argmax`. Мы сделаем это для партии из 12 примеров и отобразим ввод и предсказанные метки:

```
>>> fig = plt.figure(figsize=(12, 4))
>>> for i in range(12):
...     ax = fig.add_subplot(2, 6, i+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     img = mnist_test_dataset[i][0][0, :, :]
...     pred = model(img.unsqueeze(0).unsqueeze(1))
...     y_pred = torch.argmax(pred)
...     ax.imshow(img, cmap='gray_r')
...     ax.text(0.9, 0.1, y_pred.item(),
...             size=15, color='blue',
...             horizontalalignment='center',
...             verticalalignment='center',
...             transform=ax.transAxes)
>>> plt.show()
```

На рис. 14.14 показаны рукописные данные и их предполагаемые метки. Как можно видеть, в этом наборе графических примеров все предсказанные метки верны.

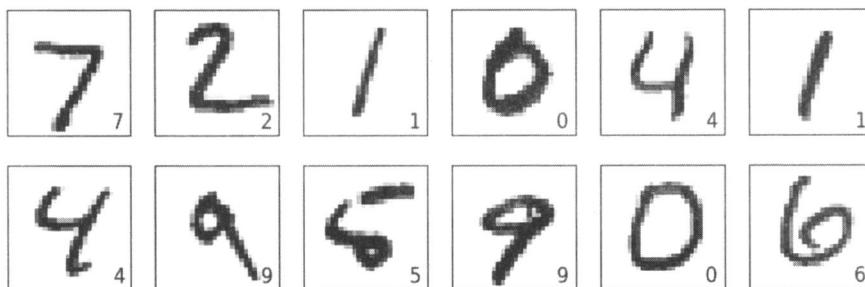


Рис. 14.14. Прогнозируемые метки для рукописных цифр

В *главе 11* мы также отображали несколько примеров неправильно идентифицированных цифр. Здесь мы оставляем отображение таких примеров в качестве упражнения для читателя.

14.4. Классификация улыбающихся лиц с помощью CNN

В этом разделе мы построим CNN для классификации улыбающихся лиц с использованием набора данных CelebA. Как было сказано в главе 12, набор данных CelebA содержит 202 599 изображений лиц знаменитостей. Кроме того, для каждого изображения доступно 40 бинарных атрибутов лица, отражающих в том числе, улыбается знаменитость или нет, и ее возраст (молодая или пожилая).

Цель этого раздела — основываясь на полученных вами знаниях, построить и обучить модель CNN для прогнозирования наличия улыбки на изображениях лиц. Здесь для простоты — чтобы ускорить процесс обучения — мы задействуем только небольшую часть обучающих данных (16 тыс. обучающих примеров). Однако для улучшения качества обобщения и уменьшения времени переобучения на таком небольшом наборе данных мы применим метод, называемый *дополнением данных* (data augmentation).

14.4.1. Загрузка набора данных CelebA

Начнем мы с загрузки данных — аналогично тому, как мы это делали в предыдущем разделе для набора данных MNIST. Данные CelebA поставляются с разбивкой на три набора: для обучения, для валидации и для тестирования. С помощью следующего кода мы подсчитаем количество примеров в каждом разделе:

```
>>> image_path = './'
>>> celeba_train_dataset = torchvision.datasets.CelebA(
...     image_path, split='train',
...     target_type='attr', download=True
... )
>>> celeba_valid_dataset = torchvision.datasets.CelebA(
...     image_path, split='valid',
...     target_type='attr', download=True
... )
>>> celeba_test_dataset = torchvision.datasets.CelebA(
...     image_path, split='test',
...     target_type='attr', download=True
... )
>>>
>>> print('Обучение:', len(celeba_train_dataset))
Обучение: 162770
>>> print('Валидация:', len(celeba_valid_dataset))
Валидация: 19867
>>> print('Test set:', len(celeba_test_dataset))
Тестирование: 19962
```



Альтернативные способы загрузки набора данных CelebA

Набор данных CelebA относительно велик (примерно 1.5 Гбайт), а ссылка для скачивания через torchvision, как известно, нестабильна. Если у вас возникли проблемы с выполнением приведенного кода, вы можете загрузить файлы с официального сайта CelebA вручную (<https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>) или

воспользоваться нашей ссылкой для скачивания: <https://drive.google.com/file/d/1m8-EBPgi5MRubrm6iQjafK2QMHBMSfJ/view?usp=sharing>. В этом случае будет загружен файл *celeba.zip*, который вам необходимо распаковать в текущий каталог, откуда вы запускаете код. Кроме того, после загрузки и распаковки папки *celeba* вам необходимо повторно запустить приведенный код с параметром `download=False` вместо `download=True`. Если у вас возникнут проблемы с этим способом, пожалуйста, откройте обсуждение на <https://github.com/rasbt/machine-learning-book>, чтобы мы смогли помочь вам разобраться.

Далее мы обсудим дополнение данных как метод повышения производительности глубоких нейронных сетей.

14.4.2. Преобразование изображений и дополнение данных

Дополнение данных охватывает широкий набор методов для работы со случаями, когда данные для обучения ограничены. Например, некоторые методы дополнения данных позволяют нам изменять или даже искусственно синтезировать дополнительные данные и тем самым увеличивать производительность модели глубокого обучения за счет снижения переобучения. Хотя дополнение данных применяют не только к изображениям, существует набор преобразований, предназначенных специально для изображений, — таких как обрезка частей изображения, зеркальное отражение и изменение контраста, яркости и насыщенности. Давайте рассмотрим некоторые из этих преобразований, доступных через модуль `torchvision.transforms`. В следующем блоке кода мы извлечем пять примеров из набора данных *celeba_train_dataset* и применим пять различных типов преобразования: 1) обрезка изображения до ограничительной рамки; 2) отражение изображения по горизонтали; 3) изменение контрастности; 4) изменение яркости и 5) обрезка изображения по центру и изменение размера полученного изображения до исходного размера (218, 178). В приведенном далее коде мы визуализируем результаты этих преобразований, показывая каждое в отдельном столбце для сравнения:

```
>>> fig = plt.figure(figsize=(16, 8.5))
>>> ## Столбец 1: обрезка до ограничительной рамки
>>> ax = fig.add_subplot(2, 5, 1)
>>> img, attr = celeba_train_dataset[0]
>>> ax.set_title('Обрезка до \nограничительной \nрамки', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 6)
>>> img_cropped = transforms.functional.crop(img, 50, 20, 128, 128)
>>> ax.imshow(img_cropped)
>>>
>>> ## Столбец 2: отражение (по горизонтали)
>>> ax = fig.add_subplot(2, 5, 2)
>>> img, attr = celeba_train_dataset[1]
>>> ax.set_title('Отражение (по горизонтали)', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 7)
```

```
>>> img_flipped = transforms.functional.hflip(img)
>>> ax.imshow(img_flipped)
>>>
>>> ## Столбец 3: изменение контраста
>>> ax = fig.add_subplot(2, 5, 3)
>>> img, attr = celeba_train_dataset[2]
>>> ax.set_title('Изменение контраста', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 8)
>>> img_adj_contrast = transforms.functional.adjust_contrast(
...     img, contrast_factor=2
... )
>>> ax.imshow(img_adj_contrast)
>>>
>>> ## Столбец 4: изменение яркости
>>> ax = fig.add_subplot(2, 5, 4)
>>> img, attr = celeba_train_dataset[3]
>>> ax.set_title('Изменение яркости', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 9)
>>> img_adj_brightness = transforms.functional.adjust_brightness(
...     img, brightness_factor=1.3
... )
>>> ax.imshow(img_adj_brightness)
>>>
>>> ## Столбец 5: обрезка относительно центра
>>> ax = fig.add_subplot(2, 5, 5)
>>> img, attr = celeba_train_dataset[4]
>>> ax.set_title('Обрезка по \nцентру и подгонка \nразмера', size=15)
>>> ax.imshow(img)
>>> ax = fig.add_subplot(2, 5, 10)
>>> img_center_crop = transforms.functional.center_crop(
...     img, [0.7*218, 0.7*178]
... )
>>> img_resized = transforms.functional.resize(
...     img_center_crop, size=(218, 178)
... )
>>> ax.imshow(img_resized)
>>> plt.show()
```

На рис. 14.15 показаны результаты произведенных преобразований. Здесь исходные изображения приведены в первой строке, а их преобразованные версии — во второй. Обратите внимание, что для первого преобразования (крайний левый столбец) ограничивающая рамка определяется четырьмя числами: координатой верхнего левого угла ограничивающей рамки (здесь $x = 20$, $y = 50$), а также шириной и высотой рамки (ширина = 128, высота = 128). Точки начала координат (0, 0) для изображений, загружаемых PyTorch (а также другими пакетами, такими как `imageio`), являются верхний левый угол изображения.

Преобразования в приведенном блоке кода являются детерминированными. Однако все подобные преобразования также можно рандомизировать, что рекомендуется делать

при дополнении обучающих данных. Например, можно выполнить обрезку по случайной ограничивающей рамке (когда координаты верхнего левого угла рамки генерируются случайным образом), изображение может быть случайным образом перевернуто либо по горизонтальной, либо по вертикальной оси с вероятностью 0.5, или коэффициент изменения контрастности можно выбирать из заданного диапазона случайным образом, но с равномерным распределением. Кроме того, можно создать последовательность преобразований.



Рис. 14.15. Различные преобразования изображений

Например, мы можем сначала произвольно обрезать изображение, затем случайным образом отразить его и, наконец, изменить его размер до нужных значений. Поскольку у нас используются случайные элементы, мы задаем случайное начальное число для воспроизводимости результатов работы кода:

```
>>> torch.manual_seed(1)
>>> fig = plt.figure(figsize=(14, 12))
>>> for i, (img, attr) in enumerate(celeba_train_dataset):
...     ax = fig.add_subplot(3, 4, i*4+1)
...     ax.imshow(img)
...     if i == 0:
...         ax.set_title('Оригинал', size=15)
...
...     ax = fig.add_subplot(3, 4, i*4+2)
...     img_transform = transforms.Compose([
...         transforms.RandomCrop([178, 178])
...     ])
...     img_cropped = img_transform(img)
...     ax.imshow(img_cropped)
```

```

...
    if i == 0:
        ax.set_title('Шаг 1: Случайная обрезка', size=15)
...
...
    ax = fig.add_subplot(3, 4, i*4+3)
    img_transform = transforms.Compose([
        transforms.RandomHorizontalFlip()
    ])
    img_flip = img_transform(img_cropped)
    ax.imshow(img_flip)
...
    if i == 0:
        ax.set_title('Шаг 2: Случайное отражение', size=15)
...
...
    ax = fig.add_subplot(3, 4, i*4+4)
    img_resized = transforms.functional.resize(
        img_flip, size=(128, 128)
    )
    ax.imshow(img_resized)
...
    if i == 0:
        ax.set_title('Шаг 3: Изменение размера', size=15)
...
    if i == 2:
...
        break
>>> plt.show()

```

На рис. 14.16 показаны случайные преобразования трех примеров изображений.

При каждой новой обработке этих трех примеров, мы будем получать немного разные изображения из-за случайного компонента преобразований.

Для удобства мы можем определить функции преобразования, чтобы использовать этот конвейер для дополнения данных во время загрузки набора. В следующем коде мы определим функцию `get_smile`, которая будет извлекать метку улыбки из списка `'attributes'`:

```
>>> get_smile = lambda attr: attr[31]
```

Мы также определим функцию `transform_train`, которая создаст преобразованное изображение (где мы сначала произвольно обрежем изображение, затем случайным образом отразим его и, наконец, изменим его размер до желаемого размера 64×64):

```

>>> transform_train = transforms.Compose([
...     transforms.RandomCrop([178, 178]),
...     transforms.RandomHorizontalFlip(),
...     transforms.Resize([64, 64]),
...     transforms.ToTensor(),
... ])

```

Однако мы будем применять дополнение данных только к обучающим примерам, а не к проверочным или тестовым изображениям. Код для обработки валидационного или тестового набора выглядит следующим образом (здесь мы сначала просто обрежем изображение, а затем изменим его до нужного размера 64×64):

```

>>> transform = transforms.Compose([
...     transforms.CenterCrop([178, 178]),

```

```
...     transforms.Resize([64, 64]),
...     transforms.ToTensor(),
... ])
```

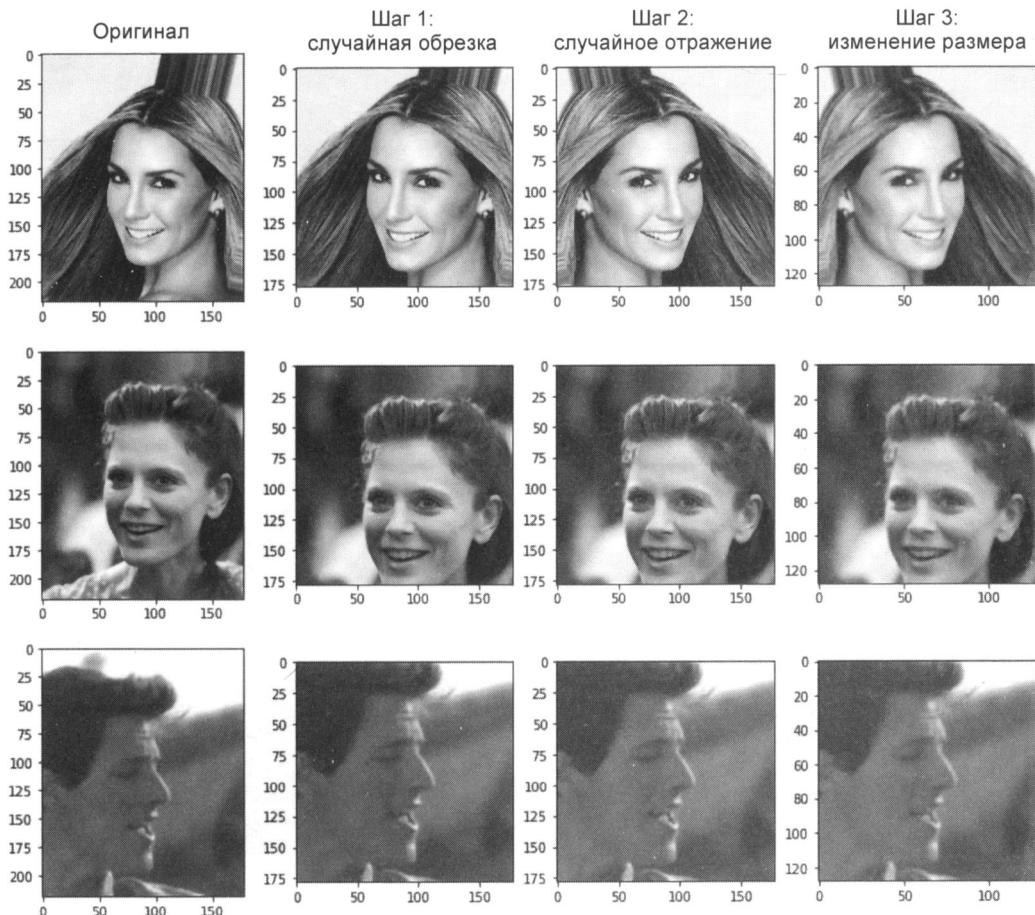


Рис. 14.16. Случайные преобразования изображений

Теперь, чтобы увидеть дополнение данных в действиях, применим функцию `transform_train` к нашему набору обучающих данных и пройдем по набору данных пять раз:

```
>>> from torch.utils.data import DataLoader
>>> celeba_train_dataset = torchvision.datasets.CelebA(
...     image_path, split='train',
...     target_type='attr', download=False,
...     transform=transform_train, target_transform=get_smile
... )
>>> torch.manual_seed(1)
>>> data_loader = DataLoader(celeba_train_dataset, batch_size=2)
>>> fig = plt.figure(figsize=(15, 6))
>>> num_epochs = 5
```

```
>>> for j in range(num_epochs):
...     img_batch, label_batch = next(iter(data_loader))
...     img = img_batch[0]
...     ax = fig.add_subplot(2, 5, j + 1)
...     ax.set_xticks([])
...     ax.set_yticks([])
...     ax.set_title(f'Эпоха {j}:', size=15)
...     ax.imshow(img.permute(1, 2, 0))
...
...     img = img_batch[1]
...     ax = fig.add_subplot(2, 5, j + 6)
...     ax.set_xticks([])
...     ax.set_yticks([])
...     ax.imshow(img.permute(1, 2, 0))
>>> plt.show()
```

На рис. 14.17 показаны пять результатов преобразований для дополнения данных, примененных к двум примерам изображений.



Рис. 14.17. Результаты пяти преобразований изображения

Далее мы применим функцию `transform` к нашим наборам данных для валидации и тестирования:

```
>>> celeba_valid_dataset = torchvision.datasets.CelebA(
...     image_path, split='valid',
...     target_type='attr', download=False,
...     transform=transform, target_transform=get_smile
... )
>>> celeba_test_dataset = torchvision.datasets.CelebA(
...     image_path, split='test',
...     target_type='attr', download=False,
...     transform=transform, target_transform=get_smile
... )
```

Вместо того чтобы использовать все доступные данные для обучения и валидации, мы возьмем подмножество из 16 тыс. обучающих примеров и 1000 примеров для валида-

ции, поскольку наша цель здесь — намеренно обучить нашу модель с небольшим набором данных:

```
>>> from torch.utils.data import Subset
>>> celeba_train_dataset = Subset(celeba_train_dataset,
...                                 torch.arange(16000))
>>> celeba_valid_dataset = Subset(celeba_valid_dataset,
...                                 torch.arange(1000))
...
>>> print('Набор для обучения:', len(celeba_train_dataset))
Набор для обучения: 16000
>>> print(':', len(celeba_valid_dataset))
Набор для валидации: 1000
```

Теперь мы можем создать загрузчики данных для трех наборов данных:

```
>>> batch_size = 32
>>> torch.manual_seed(1)
>>> train_dl = DataLoader(celeba_train_dataset,
...                        batch_size, shuffle=True)
...
>>> valid_dl = DataLoader(celeba_valid_dataset,
...                        batch_size, shuffle=False)
...
>>> test_dl = DataLoader(celeba_test_dataset,
...                       batch_size, shuffle=False)
```

Итак, загрузчики данных готовы, и в следующем разделе мы разработаем модель CNN, а также обучим ее и оценим точность классификации.

14.4.3. Обучение классификатора улыбки на основе CNN

К настоящему времени создание модели с помощью модуля `torch.nn` и ее обучение не должно вызывать у вас затруднений.

Наша CNN устроена так: модель получает на вход изображения размером $3 \times 64 \times 64$ (три цветовых канала). Входные данные проходят через четыре сверточных слоя для создания 32, 64, 128 и 256 карт объектов с использованием фильтров с размером ядра 3×3 и заполнением 1 в режиме равного заполнения. За первыми тремя сверточными слоями следует объединение по максимуму $P_{2 \times 2}$. Для регуляризации также включены два пропреживающих слоя:

```
>>> model = nn.Sequential()
>>> model.add_module(
...     'conv1',
...     nn.Conv2d(
...         in_channels=3, out_channels=32,
...         kernel_size=3, padding=1
...     )
... )
...
>>> model.add_module('relu1', nn.ReLU())
>>> model.add_module('pool1', nn.MaxPool2d(kernel_size=2))
>>> model.add_module('dropout1', nn.Dropout(p=0.5))
>>>
```

```
>>> model.add_module(  
...     'conv2',  
...     nn.Conv2d(  
...         in_channels=32, out_channels=64,  
...         kernel_size=3, padding=1  
...     )  
... )  
>>> model.add_module('relu2', nn.ReLU())  
>>> model.add_module('pool2', nn.MaxPool2d(kernel_size=2))  
>>> model.add_module('dropout2', nn.Dropout(p=0.5))  
>>>  
>>> model.add_module(  
...     'conv3',  
...     nn.Conv2d(  
...         in_channels=64, out_channels=128,  
...         kernel_size=3, padding=1  
...     )  
... )  
>>> model.add_module('relu3', nn.ReLU())  
>>> model.add_module('pool3', nn.MaxPool2d(kernel_size=2))  
>>>  
>>> model.add_module(  
...     'conv4',  
...     nn.Conv2d(  
...         in_channels=128, out_channels=256,  
...         kernel_size=3, padding=1  
...     )  
... )  
>>> model.add_module('relu4', nn.ReLU())
```

Давайте посмотрим на форму выходных карт объектов после применения этих слоев с использованием демонстрационного пакетного ввода (четыре произвольных изображения):

```
>>> x = torch.ones((4, 3, 64, 64))  
>>> model(x).shape  
torch.Size([4, 256, 8, 8])
```

Итак, у нас имеются 256 карт признаков (или каналов) размером 8×8 . Теперь мы можем добавить полносвязный слой, чтобы добраться до выходного слоя с единственным узлом. Если мы изменим размерность карт объектов, количество входных узлов для этого полносвязного слоя составит $8 \times 8 \times 256 = 16\,384$. В качестве альтернативы рассмотрим новый слой, называемый *глобальным объединением по среднему* (global average-pooling), который вычисляет среднее значение каждой карты объектов отдельно, тем самым уменьшая количество скрытых узлов до 256. После него мы можем добавить полносвязный слой. Хотя мы не обсуждали глобальное объединение по среднему в явном виде, концептуально оно очень похоже на другие способы объединения. Его можно рассматривать, по сути, как частный случай объединения по среднему, когда размер области объединения равен размеру входных карт признаков.

Чтобы лучше все это понять, взгляните на рис. 14.18, где показан пример карты входных признаков [размер пакета $\times 8 \times 64 \times 64$]. Каналы пронумерованы индексом $k = 0, 1, \dots, 7$.

Операция глобального объединения по среднему вычисляет среднее значение каждого канала, так что выходные данные будут иметь форму [размер пакета×8]. Затем мы сожмем выходные данные слоя глобального усреднения.

Без сжатия вывода форма будет [размер пакета×8×1×1], т. к. глобальное усреднение уменьшит пространственное измерение 64×64 до 1×1, как и показано на рис. 14.18.

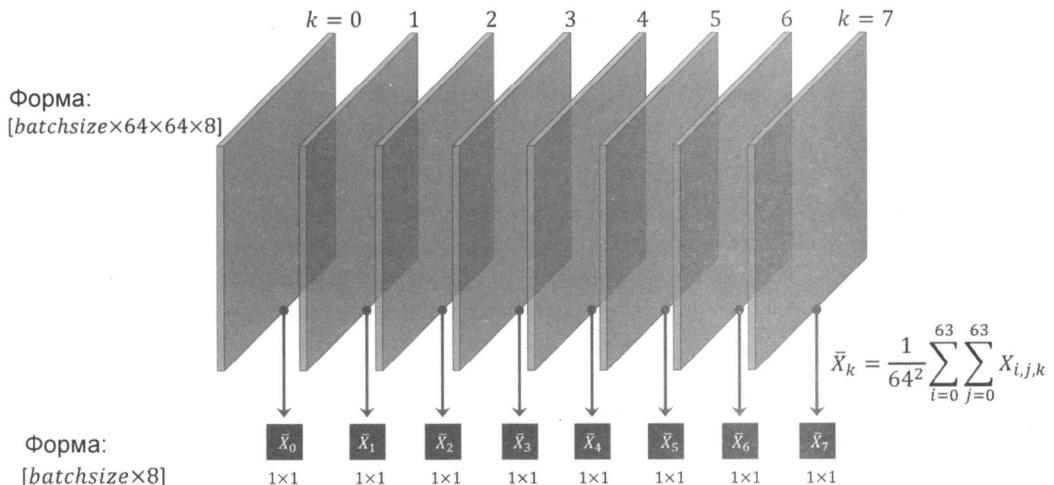


Рис. 14.18. Карты входных признаков

Таким образом, учитывая, что в нашем случае форма карт признаков до этого слоя равна [размер пакета×256×8×8], мы ожидаем получить на выходе 256 узлов, т. е. форма на выходе будет [размер пакета×256]. Давайте добавим этот слой и заново вычислим форму выхода, чтобы убедиться, что это так:

```
>>> model.add_module('pool4', nn.AvgPool2d(kernel_size=8))
>>> model.add_module('flatten', nn.Flatten())
>>> x = torch.ones((4, 3, 64, 64))
>>> model(x).shape
torch.Size([4, 256])
```

Наконец, можно добавить полносвязный слой, чтобы получить единственный выходной узел. В этом случае мы можем выбрать функцию активации 'sigmoid':

```
>>> model.add_module('fc', nn.Linear(256, 1))
>>> model.add_module('sigmoid', nn.Sigmoid())
>>> x = torch.ones((4, 3, 64, 64))
>>> model(x).shape
torch.Size([4, 1])
>>> model
Sequential(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (relu1): ReLU()
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout1): Dropout(p=0.5, inplace=False)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): AvgPool2d(kernel_size=8, stride=8, padding=0)
  (flatten): Flatten()
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

```
(relu2): ReLU()
(pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(dropout2): Dropout(p=0.5, inplace=False)
(conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(relu3): ReLU()
(pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(relu4): ReLU()
(pool4): AvgPool2d(kernel_size=8, stride=8, padding=0)
(flatten): Flatten(start_dim=1, end_dim=-1)
(fc): Linear(in_features=256, out_features=1, bias=True)
(sigmoid): Sigmoid()
)
```

На следующем шаге мы создадим функции потерь и оптимизатора (снова оптимизатор Adam). Для бинарной классификации с одним вероятностным выводом мы используем BCELoss для функции потерь:

```
>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Теперь мы можем обучить модель, определив функцию train:

```
>>> def train(model, num_epochs, train_dl, valid_dl):
...     loss_hist_train = [0] * num_epochs
...     accuracy_hist_train = [0] * num_epochs
...     loss_hist_valid = [0] * num_epochs
...     accuracy_hist_valid = [0] * num_epochs
...     for epoch in range(num_epochs):
...         model.train()
...         for x_batch, y_batch in train_dl:
...             pred = model(x_batch)[:, 0]
...             loss = loss_fn(pred, y_batch.float())
...             loss.backward()
...             optimizer.step()
...             optimizer.zero_grad()
...             loss_hist_train[epoch] += loss.item() * y_batch.size(0)
...             is_correct = ((pred >= 0.5).float() == y_batch).float()
...             accuracy_hist_train[epoch] += is_correct.sum()
...             loss_hist_train[epoch] /= len(train_dl.dataset)
...             accuracy_hist_train[epoch] /= len(train_dl.dataset)
...
...         model.eval()
...         with torch.no_grad():
...             for x_batch, y_batch in valid_dl:
...                 pred = model(x_batch)[:, 0]
...                 loss = loss_fn(pred, y_batch.float())
...                 loss_hist_valid[epoch] += \
...                     loss.item() * y_batch.size(0)
...                 is_correct = \
...                     ((pred >= 0.5).float() == y_batch).float()
...                 accuracy_hist_valid[epoch] += is_correct.sum()
```

```

...
    loss_hist_valid[epoch] /= len(valid_dl.dataset)
    accuracy_hist_valid[epoch] /= len(valid_dl.dataset)

...
    print(f'Точность после эпохи {epoch+1}: '
          f'{accuracy_hist_train[epoch]:.4f} val_accuracy: '
          f'{accuracy_hist_valid[epoch]:.4f}')
...
    return loss_hist_train, loss_hist_valid, \
           accuracy_hist_train, accuracy_hist_valid

```

Затем обучим эту модель CNN в течение 30 эпох и воспользуемся набором данных валидации, который мы создали для отслеживания процесса обучения:

```

>>> torch.manual_seed(1)
>>> num_epochs = 30
>>> hist = train(model, num_epochs, train_dl, valid_dl)
Точность после эпохи 1: 0.6286 val_accuracy: 0.6540
...
Точность после эпохи 15: 0.8544 val_accuracy: 0.8700
...
Точность после эпохи 30: 0.8739 val_accuracy: 0.8710

```

Теперь построим кривую обучения и сравним потери и точность при обучении и валидации после каждой эпохи (рис. 14.19):

```

>>> x_arr = np.arange(len(hist[0])) + 1
>>> fig = plt.figure(figsize=(12, 4))
>>> ax = fig.add_subplot(1, 2, 1)
>>> ax.plot(x_arr, hist[0], '-o', label='Потери при обучении')
>>> ax.plot(x_arr, hist[1], '--<', label='Протери при валидации')
>>> ax.legend(fontsize=15)
>>> ax = fig.add_subplot(1, 2, 2)
>>> ax.plot(x_arr, hist[2], '-o', label='Точность при обучении')
>>> ax.plot(x_arr, hist[3], '--<', label='Точность при валидации')
>>> ax.legend(fontsize=15)
>>> ax.set_xlabel('Эпоха', size=15)
>>> ax.set_ylabel('Точность', size=15)
>>> plt.show()

```

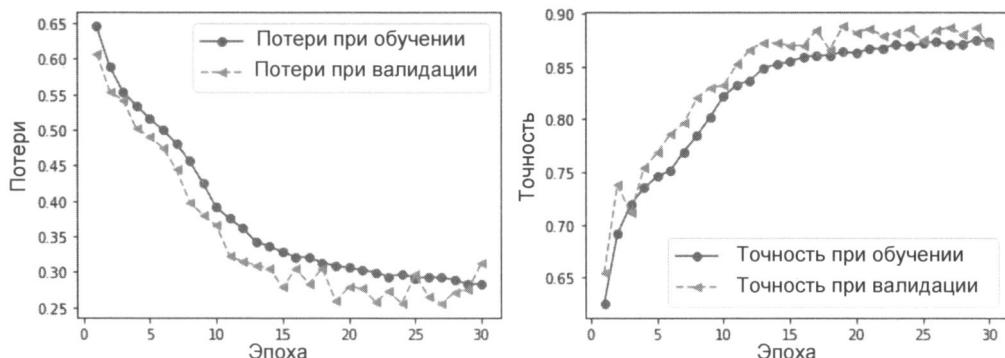


Рис. 14.19. Сравнение результатов при обучении и валидации

Если, судя по кривым, достигнут достаточный уровень обучения и точности, можно перейти к оценке модели на наборе тестовых данных:

```
>>> accuracy_test = 0
>>> model.eval()
>>> with torch.no_grad():
...     for x_batch, y_batch in test_dl:
...         pred = model(x_batch)[:, 0]
...         is_correct = ((pred>=0.5).float() == y_batch).float()
...         accuracy_test += is_correct.sum()
>>> accuracy_test /= len(test_dl.dataset)
>>> print(f'Точность при тестировании: {accuracy_test:.4f}')
Точность при тестировании: 0.8446
```

Наконец, вы уже знаете, как получить результаты прогнозирования для нескольких тестовых примеров. В следующем коде мы возьмем небольшое подмножество в 10 примеров из последнего пакета нашего предварительно обработанного набора тестовых данных (`test_dl`). Затем вычислим вероятности того, что каждый пример относится к классу 1 (что соответствует наличию улыбки на основе меток, предоставленных в CelebA), и отобразим примеры вместе с их метками истинности (Ground Truth labels, GT) и предсказанными вероятностями (Probabilities, Pr):

```
>>> pred = model(x_batch)[:, 0] * 100
>>> fig = plt.figure(figsize=(15, 7))
>>> for j in range(10, 20):
...     ax = fig.add_subplot(2, 5, j-10+1)
...     ax.set_xticks([]); ax.set_yticks([])
...     ax.imshow(x_batch[j].permute(1, 2, 0))
...     if y_batch[j] == 1:
...         label='Улыбка'
...     else:
...         label = 'Нет улыбки'
...     ax.text(
...         0.5, -0.15,
...         f'GT: {label:s}\nPr(Smile)={pred[j]:.0f}%',
...         size=16,
...         horizontalalignment='center',
...         verticalalignment='center',
...         transform=ax.transAxes
...     )
>>> plt.show()
```

На рис. 14.20 показаны 10 примеров изображений вместе с их метками истинности и вероятностью того, что они принадлежат к классу 1 (улыбка).

Вероятности принадлежности к классу 1 (т. е. наличие улыбки по CelebA) приведены под каждым изображением. Как видите, наша обученная модель полностью точна на этом наборе из 10 тестовых примеров.

В качестве дополнительного упражнения предлагаем читателям попробовать задействовать весь набор обучающих данных вместо созданного нами небольшого подмножества. Кроме того, вы можете модифицировать архитектуру CNN. Например, изменить

вероятность прореживания и количество фильтров в разных сверточных слоях. Кроме того, можно заменить глобальное объединение по среднему полностью подключенным слоем. Если вы используете весь обучающий набор данных с архитектурой CNN, которую мы обучали в этой главе, то сможете достичь точности выше 90%.

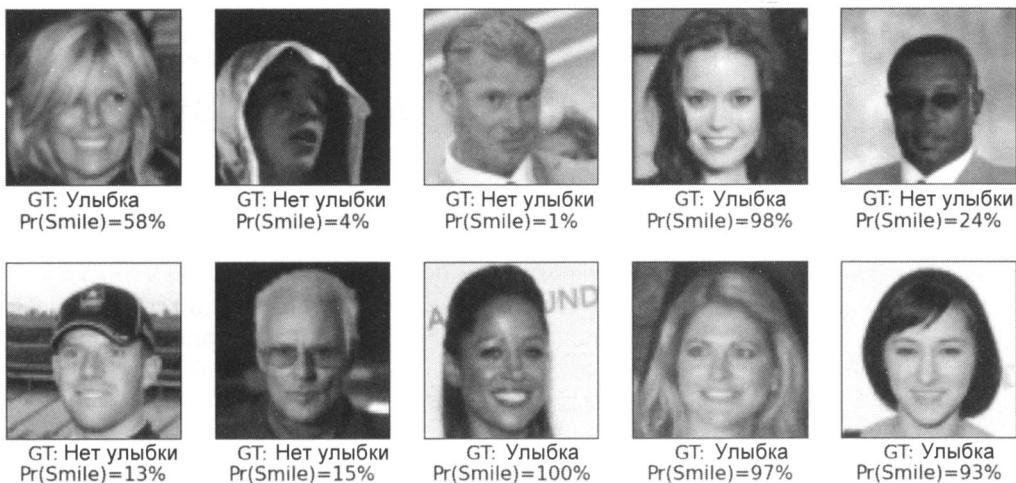


Рис. 14.20. Метки изображений и вероятности того, что они принадлежат классу 1 (улыбка)

14.5. Заключение

В этой главе вы узнали о CNN и их основных компонентах. Мы начали с операции свертки и рассмотрели одномерные и двумерные реализации. Затем мы познакомились с другой разновидностью слоя, которая встречается в нескольких распространенных архитектурах CNN: подвыборкой, или объединением. В первую очередь мы сосредоточились на двух наиболее распространенных формах объединения: по максимальному и по среднему значению.

Затем, объединив все эти компоненты, мы создали глубокие CNN с помощью модуля `torch.nn`. Первую из созданных сетей мы применили к уже знакомой задаче распознавания рукописных цифр из набора MNIST, а вторую — к более сложному набору данных, состоящему из изображений лиц, и обучили CNN определять наличие улыбки. Попутно вы также узнали о дополнении данных и различных преобразованиях, которые мы можем применять к изображениям лиц с помощью модуля `torchvision.transforms`.

В следующей главе мы перейдем к рекуррентным нейронным сетям (Recurrent Neural Network, RNN). Они предназначены для изучения структуры последовательных данных, и у них есть несколько интересных применений, включая языковой перевод и генерацию подписей к изображениям.

15

Моделирование последовательных данных с помощью рекуррентных нейронных сетей

Предыдущая глава была посвящена сверточным нейронным сетям (CNN). Мы рассмотрели функциональные блоки этой архитектуры и способы реализации глубоких CNN в PyTorch. Вы также узнали, как использовать CNN для классификации изображений. Здесь же мы познакомимся с *рекуррентными нейронными сетями* (RNN) и их применением в моделировании последовательных данных.

В этой главе будут рассмотрены следующие темы:

- ◆ понятие последовательных данных;
- ◆ RNN для моделирования последовательностей;
- ◆ долгая краткосрочная память;
- ◆ усеченное обратное распространение во времени;
- ◆ реализация многослойной RNN для моделирования последовательности в PyTorch;
- ◆ первый проект: анализ эмоциональной окраски в наборе данных обзоров фильмов IMDb;
- ◆ второй проект: моделирование языка на уровне символов с ячейками LSTM на текстовых данных из книги Жюля Верна «Таинственный остров»;
- ◆ использование обрезки градиента, чтобы избежать взрыва градиентов.

15.1. Знакомство с понятием последовательных данных

Давайте начнем обсуждение RNN с рассмотрения природы *последовательных данных* (*sequential data*), которые часто называют просто *последовательностями* (*sequence*). Мы рассмотрим уникальные свойства последовательностей, которые отличают их от других видов данных. Затем вы познакомитесь со способами представления последовательных данных и с различными категориями моделей для последовательных данных, которые основаны на входах и выходах модели. Это поможет нам исследовать связь между RNN и последовательностями.

15.1.1. Моделирование последовательных данных: порядок имеет значение

То, что элементы в последовательности появляются в определенном порядке и не являются независимыми друг от друга, делает последовательности уникальными по сравнению с другими типами данных. Типичные алгоритмы машинного обучения для обучения с учителем предполагают, что входные данные являются *независимыми и одинаково распределенными*, поэтому обучающие примеры тоже являются *взаимно независимыми* и имеют одинаковое базовое распределение. Соответственно, исходя из предположения об их взаимной независимости, порядок, в котором обучающие примеры даются модели, не имеет значения. Так, если у нас есть выборка, состоящая из n обучающих примеров, $x^{(1)}, x^{(2)}, \dots, x^{(n)}$, порядок, в котором мы используем данные для обучения нашей модели, значения не имеет. Примером этого сценария может служить набор данных Iris, с которым мы работали ранее. В наборе данных Iris каждый цветок был измерен независимо от других, и размеры одного цветка не влияют на размеры другого.

Однако, когда мы имеем дело с последовательностями, предположение о взаимной независимости данных неверно, — по определению порядок имеет значение. Примером такого сценария может быть прогнозирование рыночной стоимости конкретной акции. Например, пусть у нас есть выборка из n обучающих примеров, где каждый элемент представляет собой рыночную стоимость определенной акции в определенный день. Если наша задача состоит в том, чтобы спрогнозировать стоимость акций на следующие три дня, имеет смысл рассматривать предыдущие цены акций в порядке поступления по времени для выявления тенденции, а не использовать эти обучающие примеры в рандомизированном порядке.

15.1.2. Данные временных рядов — особый тип последовательных данных

Данные временных рядов — это особый тип последовательных данных, где каждый пример связан с измерением времени. В данных временных рядов выборки берутся с последовательными временными метками, и поэтому измерение времени определяет порядок точек данных. Например, данными временных рядов являются курсы акций и голосовые записи.

С другой стороны, не все последовательные данные имеют измерение времени. Например, в текстовых данных или последовательностях ДНК экземпляры упорядочены, но текст или ДНК не имеют составляющую времени. В этой главе мы сосредоточимся на примерах обработки естественного языка (Natural Language Processing, NLP) и моделирования текста, которые не являются данными временных рядов. Надо, впрочем, заметить, что RNN также можно использовать для данных временных рядов, но их рассмотрение выходит за рамки этой книги.

15.1.3. Способы представления последовательностей

Мы установили, что порядок между точками данных важен для последовательных данных, поэтому теперь нам нужно найти способ использовать эту информацию о порядке в модели машинного обучения. В этой главе мы будем представлять последовательно-

сти так: $\langle x^{(1)}, x^{(2)}, \dots, x^{(T)} \rangle$. Надстрочные индексы указывают порядок экземпляров, а длина последовательности равна T . В качестве характерного примера последовательностей рассмотрим данные временного ряда, где каждая точка данных $x^{(t)}$ принадлежит определенному времени t . На рис. 15.1 показан пример данных временного ряда, где как входные признаки x , так и целевые метки y естественным образом следуют порядку в соответствии с их временной осью, — следовательно, и x , и y являются последовательностями.

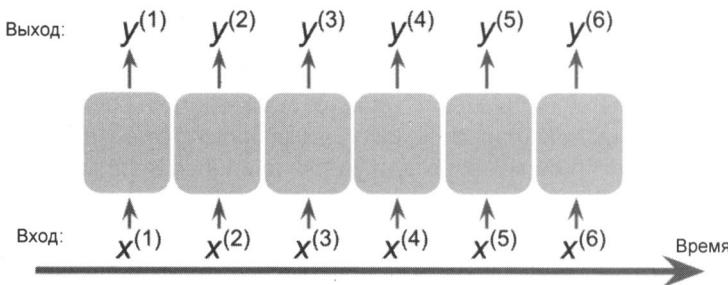


Рис. 15.1. Пример данных временного ряда

Как мы уже упоминали, стандартные модели нейросетей, которые мы рассматривали до сих пор, — такие как многослойные персептроны (MLP) и CNN для данных изображений, предполагают, что обучающие примеры независимы друг от друга и, следовательно, не включают информацию о порядке следования. Можно сказать, что такие модели не «запоминают» ранее увиденные обучающие примеры. Например, экземпляры данных проходят этапы прямого и обратного распространения, а веса обновляются независимо от порядка, в котором обрабатываются обучающие примеры.

RNN, напротив, предназначены для моделирования последовательностей и способны запоминать предыдущие данные и соответствующим образом обрабатывать новые события, что является явным преимуществом при работе с последовательными данными.

15.1.4. Различные категории моделирования последовательности

Моделирование последовательности имеет множество интересных применений — таких как языковой перевод (например, перевод текста с английского на немецкий), создание подписей к изображениям и генерация текста. Однако, чтобы выбрать подходящую архитектуру и подход, необходимо понимать и уметь различать разные задачи моделирования последовательности. На рис. 15.2, основанном на пояснениях из отличной статьи Андрея Карпати¹, представлены основные типы задач моделирования последовательности, которые зависят от категорий отношений входных и выходных данных.

Давайте обсудим более подробно различные категории отношений между входными и выходными данными, приведенными на этом рисунке. Если ни входные, ни выходные

¹ См. «The Unreasonable Effectiveness of Recurrent Neural Networks» by Andrej Karpathy, 2015, <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>.

данные не представляют собой последовательности, то мы имеем дело со стандартными данными и можем просто использовать для моделирования таких данных многослойный персепtron (или другую классифицирующую модель, ранее рассмотренную в этой книге). Однако, если вход или выход представляет собой последовательность, задача моделирования, скорее всего, попадает в одну из следующих категорий:

- ◆ **многие к одному:** входные данные представляют собой последовательность, а выходные данные — вектор фиксированного размера или скаляр, а не последовательность. Например, при анализе тональности высказываний входными данными являются текстовые данные (например, рецензия на фильм), а выходными данными — метка класса (например, метка, указывающая, понравился ли фильм рецензенту);
- ◆ **один ко многим:** входные данные представлены в обычном формате, а не в виде последовательности, но выходные данные представляют собой последовательность. Примером этой категории являются подписи к изображениям: входные данные — это изображение, а выходные данные — фраза на естественном языке, описывающая содержание этого изображения;
- ◆ **многие ко многим:** и входной, и выходной массив представляет собой последовательность. Эту категорию можно дополнительно разделить в зависимости от того, синхронизированы ли вход и выход. Примером синхронизированной задачи моделирования «многие ко многим» является классификация видео, где каждый кадр в видео снабжен меткой. Примером *отложенной* задачи моделирования «многие ко многим» может быть перевод текста с одного языка на другой, — исходное предложение должно быть полностью прочитано и обработано машиной, прежде чем будет выполнен его перевод на другой язык.

Выделив три основных категории моделирования последовательностей, мы можем перейти к обсуждению структуры RNN.

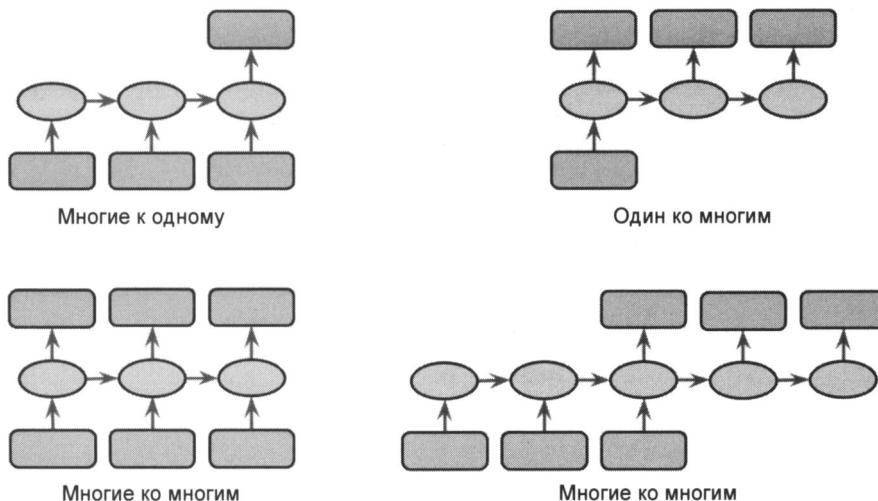


Рис. 15.2. Наиболее распространенные типы задач с последовательными данными

15.2. RNN для моделирования последовательностей

В этом разделе, прежде чем приступить к реализации RNN в PyTorch, мы обсудим основные принципы работы рекуррентной нейросети. Начнем мы с рассмотрения типичной структуры RNN, которая содержит рекурсивный компонент для моделирования последовательных данных. После чего узнаем, как в типичной RNN вычисляется активация нейронов. Эти знания послужат основой для понимания общих проблем при обучении RNN, а затем мы обсудим решения этих проблем, такие как LSTM (Long Short-Term Memory, сети долгой кратковременной памяти) и управляемые рекуррентные блоки (Gated Recurrent Units, GRU).

15.2.1. Определение потока данных в RNN

Итак, архитектура RNN. На рис. 15.3 для сравнения показаны потоки данных в стандартной нейросети прямого распространения и в RNN.

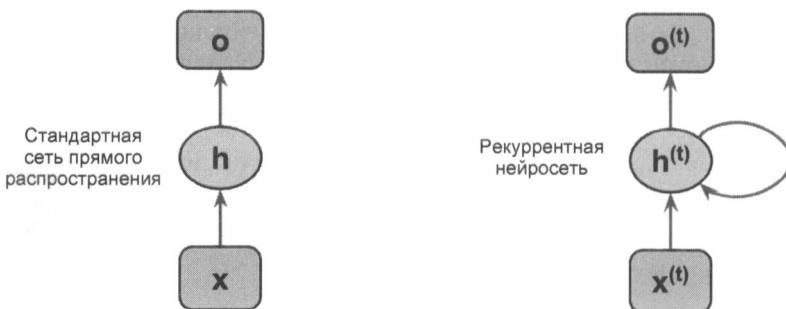


Рис. 15.3. Поток данных в стандартной нейросети прямого распространения и в RNN

Обе эти сети имеют только один скрытый слой. В приведенном представлении узлы не отображаются, но предполагается, что входной слой x , скрытый слой h и выходной слой o являются векторами, содержащими множество элементов.



Определение типа вывода из RNN

Эта общая архитектура RNN включает в себя две категории моделирования последовательности, где входом является последовательность. Как правило, рекуррентный слой может возвращать на выходе или последовательность $\langle o^{(0)}, o^{(1)}, \dots, o^{(T)} \rangle$, или просто последний вывод (при $t = T$) — т. е. $o^{(T)}$. Таким образом, это может быть либо вариант «многие ко многим», либо «многие к одному», если, например, мы используем только последний элемент $o^{(T)}$ в качестве окончательного результата.

Позже вы увидите, как эти варианты реализованы в модуле PyTorch `torch.nn`, когда мы будем подробно рассматривать поведение рекуррентного слоя в отношении возвращаемой последовательности в качестве вывода.

В стандартной сети прямого распространения поток информации движется от входа к скрытому слою, а затем от скрытого слоя к выходному слою. В свою очередь, в RNN

скрытый слой получает входные данные как от входного слоя текущего временного шага, так и от скрытого слоя предыдущего временного шага.

Поток информации в смежных временных шагах в скрытом слое позволяет сети помнить о прошлых событиях. Этот поток информации обычно отображается в виде петли, также известной — в терминах графовых структур — как *рекуррентное ребро* (recurrent edge), откуда и происходит название архитектуры RNN.

Подобно многослойным персепtronам, RNN могут состоять из нескольких скрытых слоев. Имейте в виду, что RNN с одним скрытым слоем принято называть *однослоиной RNN*, которую не следует путать с однослойными нейросетями без скрытого слоя, такими как Adaline или логистическая регрессия. На рис. 15.4 показаны RNN с одним скрытым слоем (вверху) и RNN с двумя скрытыми слоями (внизу).

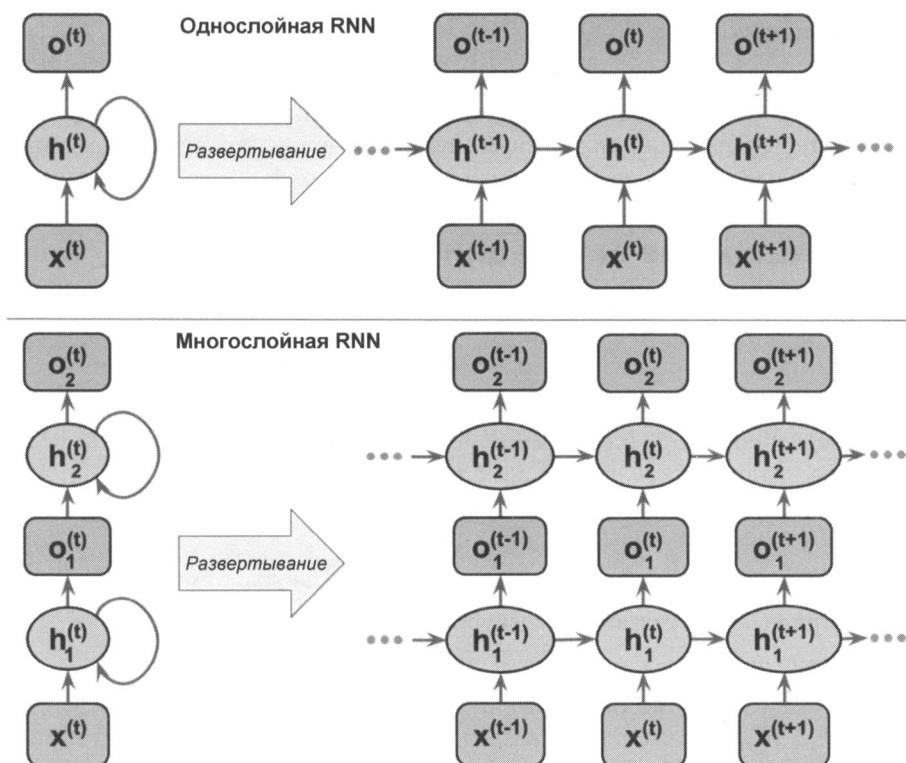


Рис. 15.4. Примеры RNN с одним (вверху) и двумя (внизу) скрытыми слоями

Чтобы изучить архитектуру RNN и поток информации, можно *развернуть во времени* компактное представление с рекуррентным ребром, как показано на рис. 15.4, справа.

Как мы знаем, каждый скрытый узел в стандартной нейросети получает только один вход — предварительную активацию, связанную со входным слоем. Напротив, каждый скрытый узел в RNN получает два *разных набора* входных данных — предварительную активацию от входного слоя и активацию того же скрытого слоя из предыдущего временного шага $t - 1$.

На первом временнóм шаге $t = 0$ скрытые узлы инициализируются нулями или небольшими случайными значениями. Затем на временнóм шаге $t > 0$ скрытые узлы получают свои входные данные из точки данных в текущий момент времени $x^{(t)}$ и предыдущие значения скрытых узлов в момент $t - 1$, обозначенные как $h^{(t-1)}$.

Точно так же в случае многослойной RNN мы можем обобщить информационный поток следующим образом:

- ◆ $layer = 1$: здесь скрытый слой представлен как $h_1^{(t)}$, и он получает входные данные от точки данных $x^{(t)}$ и скрытых значений в том же слое, но на предыдущем временнóм шаге $h_1^{(t-1)}$;
- ◆ $layer = 2$: второй скрытый слой $h_2^{(t)}$ получает входные данные от выходов нижнего слоя на текущем временнóм шаге ($o_1^{(t)}$) и собственные скрытые значения из предыдущего временнóм шага $h_2^{(t-1)}$.

Поскольку в этом случае каждый рекуррентный слой должен получать на вход последовательность, все рекуррентные слои, кроме последнего, должны возвращать на выходе последовательность (т. е. позже нам нужно будет установить параметр `return_sequences=True`). Поведение последнего рекуррентного слоя зависит от типа задачи.

15.2.2. Вычисление активаций в RNN

Теперь, когда мы рассмотрели структуру и общий поток информации в RNN, можно перейти к вычислению фактической активации скрытых слоев, а также выходного слоя. Для простоты мы будем рассматривать только один скрытый слой, однако тот же подход применим и к многослойным RNN.

Каждое направленное ребро (соединение между блоками) в представлении RNN, которое мы только что рассмотрели, связано с матрицей весов. Эти веса не зависят от времени t — следовательно, они являются общими по оси времени. В однослоиной RNN фигурируют следующие весовые матрицы (рис. 15.5):

- ◆ W_{xh} — матрица весов между входными данными $x^{(t)}$ и скрытым слоем h ;
- ◆ W_{hh} — матрица весов, связанная с рекуррентным ребром;
- ◆ W_{ho} — матрица весов между скрытым слоем и выходным слоем.

В некоторых реализациях весовые матрицы W_{xh} и W_{hh} объединяются в составную матрицу $W_h = [W_{xh}; W_{hh}]$. Позже в этом разделе мы также будем использовать это обозначение.

Вычисление активаций очень похоже на стандартные многослойные персептроны и другие типы нейронных сетей с прямой связью. Для скрытого слоя действующий ввод z_h (предварительная активация) вычисляется с помощью линейной комбинации, т. е. мы вычисляем сумму произведений весовых матриц на соответствующие векторы и добавляем компоненту смещения:

$$z_h^{(t)} = W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h.$$

Активации скрытых узлов на временнóм шаге t рассчитываются следующим образом:

$$h^{(t)} = \sigma_h(z_h^{(t)}) = \sigma_h(W_{xh}x^{(t)} + W_{hh}h^{(t-1)} + b_h).$$

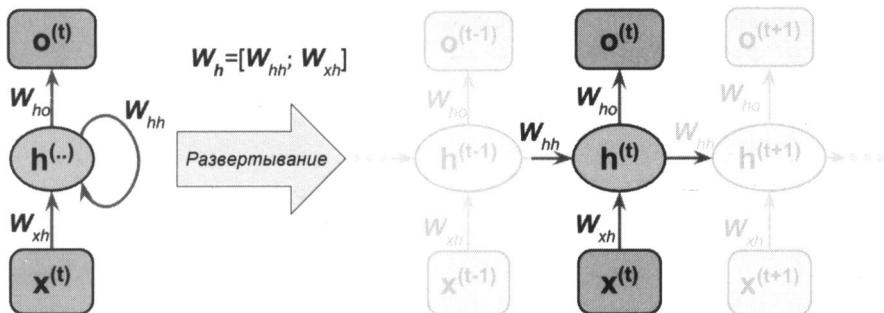


Рис. 15.5. Применение весов к однослойной RNN

Здесь b_h — вектор смещения для скрытых узлов, а $\sigma(\cdot)$ — функция активации скрытого слоя.

Если вы хотите использовать составную матрицу весов $W_h = [W_{xh}; W_{hh}]$, формула для вычисления скрытых единиц примет такой вид:

$$h^{(t)} = \sigma_h \left([W_{xh}; W_{hh}] \begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h \right).$$

После вычисления активаций скрытых узлов на текущем временнóм шаге вычисляются активации выходных узлов:

$$o^{(t)} = \sigma_o (W_{ho} h^{(t)} + b_o).$$

Для наглядности на рис. 15.6 показан процесс вычисления этих активаций в обоих вариантах.

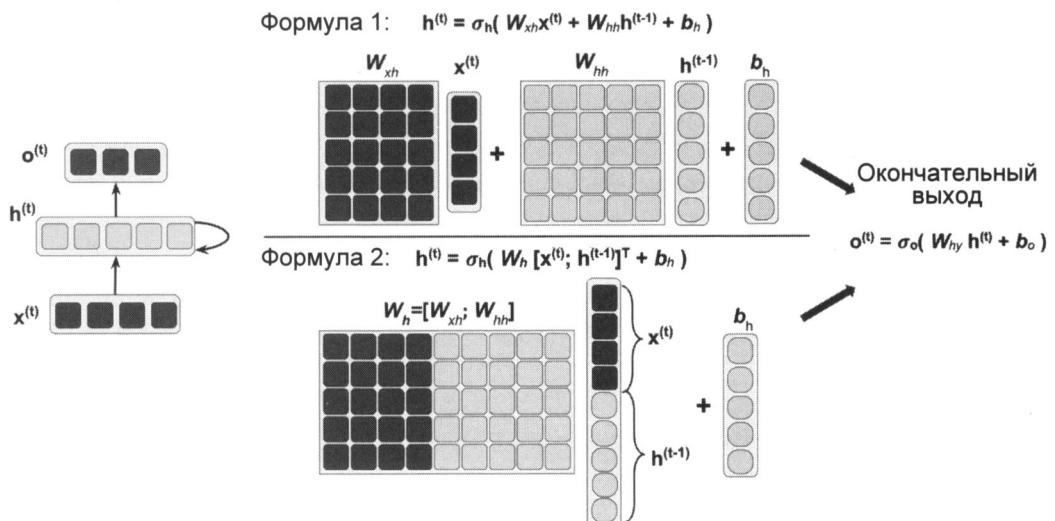


Рис. 15.6. Вычисление активаций



Обучение RNN с использованием обратного распространения во времени (Backpropagation Through Time, BPTT)

Алгоритм обучения для RNN был представлен в 1990 году в статье «Backpropagation Through Time: What It Does and How to Do It», Paul Werbos, Proceedings of IEEE, 78(10): 1550–1560, 1990.

Вывод градиентов несколько сложен, но основная идея заключается в том, что общие потери L представляют собой сумму всех функций потерь в моменты времени от $t = 1$ до $t = T$:

$$L = \sum_{t=1}^T L^{(t)}.$$

Поскольку потери в момент t зависят от скрытых узлов на всех предыдущих временных шагах $1 : t$, градиент будет вычисляться следующим образом:

$$\frac{\partial L^{(t)}}{\partial W_{hh}} = \frac{\partial L^{(t)}}{\partial o^{(t)}} \times \frac{\partial o^{(t)}}{\partial h^{(t)}} \times \left(\sum_{k=1}^t \frac{\partial h^{(t)}}{\partial h^{(k)}} \times \frac{\partial h^{(k)}}{\partial W_{hh}} \right).$$

Здесь $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ вычисляется как произведение соседних временных шагов:

$$\frac{\partial h^{(t)}}{\partial h^{(k)}} = \prod_{i=k+1}^t \frac{\partial h^{(i)}}{\partial h^{(i-1)}}.$$

15.2.3. Скрытая и выходная рекуррентность

До сих пор вы видели рекуррентные сети, в которых рекуррентная петля охватывает скрытый слой. На рис. 15.7 это показано как рекуррентная связь «скрытый к скрытому» (hidden-to-hidden recurrence). Однако надо сказать, что существует альтернативная модель, в которой рекуррентное соединение исходит из выходного слоя. В этом случае активация сети из выходного слоя на предыдущем временном шаге o'^{-1} может быть добавлена одним из двух способов:

- ◆ к скрытому слою на текущем временном шаге h' — показана на рис. 15.7 как рекуррентная связь «выход к скрытому» (output-to-hidden recurrence);
- ◆ к выходному слою на текущем временном шаге o' — показана на рис. 15.7 как рекуррентная связь «выход к выходу» (output-to-output recurrence).

Как можно здесь видеть, различия между этими архитектурами проявляются в структуре рекуррентных связей. Следуя принятой системе обозначений, веса, относящиеся к рекуррентной связи от скрытого слоя к скрытому слою, обозначим через W_{hh} , от выхода к скрытому слою — через W_{oh} и от выхода к выходу — через W_{oo} . В некоторых статьях веса, относящиеся к рекуррентным связям, также обозначаются как W_{rec} .

Чтобы продемонстрировать, как рекуррентность работает на практике, вручную вычислим прямой проход для одного из вариантов рекуррентной модели. С модулем `torch.nn` рекуррентный слой может быть определен через объект `RNN`, соответствующий рекуррентной связи от скрытого слоя к скрытому. В следующем коде мы создадим рекуррентный слой на основе `RNN` и выполним прямой проход входной последовательности

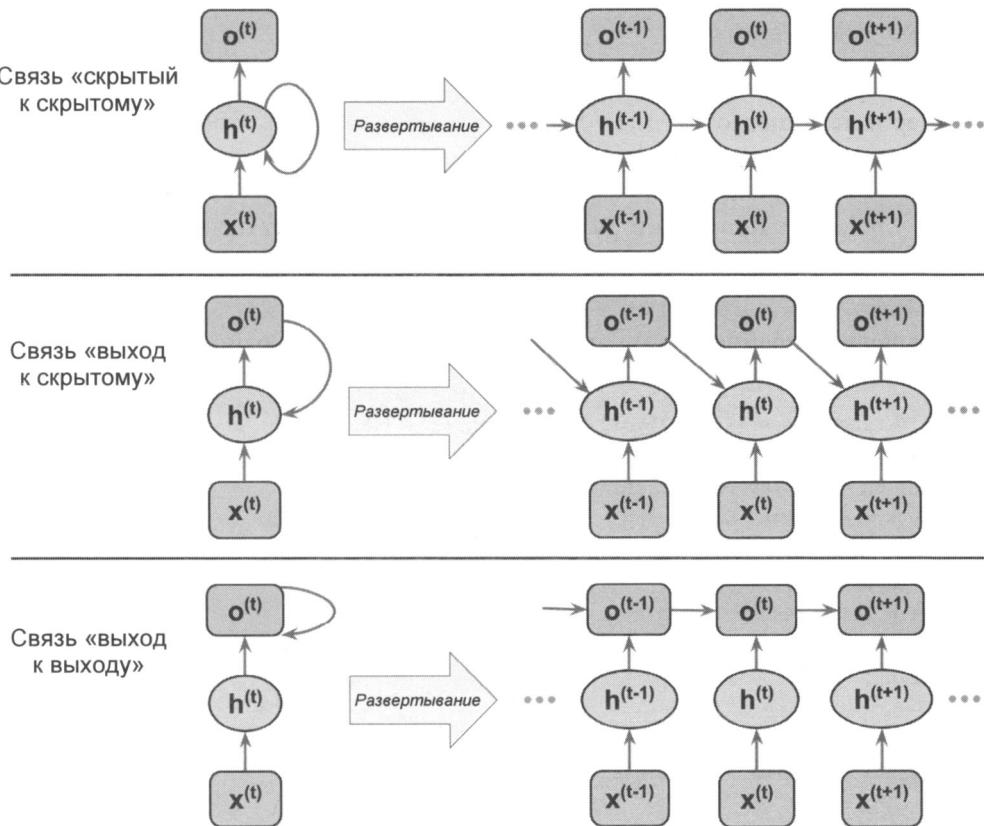


Рис. 15.7. Различные варианты рекуррентных соединений

длины 3 для вычисления выходных данных. Мы также вручную вычислим прямой проход и сравним результаты с результатами работы RNN.

Сначала создадим слой и назначим веса и смещения для наших ручных вычислений:

```
>>> import torch
>>> import torch.nn as nn
>>> torch.manual_seed(1)
>>> rnn_layer = nn.RNN(input_size=5, hidden_size=2,
...                      num_layers=1, batch_first=True)
>>> w_xh = rnn_layer.weight_ih_0
>>> w_hh = rnn_layer.weight_hh_0
>>> b_xh = rnn_layer.bias_ih_0
>>> b_hh = rnn_layer.bias_hh_0
>>> print('Форма W_xh:', w_xh.shape)
>>> print('Форма W_hh:', w_hh.shape)
>>> print('Форма b_xh:', b_xh.shape)
>>> print('Форма b_hh:', b_hh.shape)
Форма W_xh: torch.Size([2, 5])
Форма W_hh: torch.Size([2, 2])
```

Форма `b_xh: torch.Size([2])`

Форма `b_hh: torch.Size([2])`

Вход этого слоя имеет форму `(batch_size, sequence_length, 5)`, где первое измерение — это размер пакета (поскольку мы установили `batch_first=True`), второе измерение соответствует последовательности, а последнее измерение соответствует признакам. Для входной последовательности длины 3 мы должны получить выходную последовательность $\langle o^{(0)}, o^{(1)}, o^{(2)} \rangle$. Кроме того, RNN по умолчанию использует один слой, и вы можете задать значение `num_layers`, чтобы получить несколько слоев и сформировать стековую RNN.

Теперь выполним прямой проход для `rnn_layer`, вычислим выходные данные на каждом временному шаге и сравним их:

```
>>> x_seq = torch.tensor([[1.0]*5, [2.0]*5, [3.0]*5]).float()
>>> ## выход простой RNN:
>>> output, hn = rnn_layer(torch.reshape(x_seq, (1, 3, 5)))
>>> ## вручную вычисляем выход:
>>> out_man = []
>>> for t in range(3):
...     xt = torch.reshape(x_seq[t], (1, 5))
...     print(f'Шаг времени {t} =>')
...     print(' Вход :', xt.numpy())
...
...     ht = torch.matmul(xt, torch.transpose(w_xh, 0, 1)) + b_xh
...     print(' Скрытый :', ht.detach().numpy())
...
...     if t > 0:
...         prev_h = out_man[t-1]
...     else:
...         prev_h = torch.zeros((ht.shape))
...     ot = ht + torch.matmul(prev_h, torch.transpose(w_hh, 0, 1)) \
...         + b_hh
...     ot = torch.tanh(ot)
...     out_man.append(ot)
...     print(' Выход (вручную) :', ot.detach().numpy())
...     print(' Выход RNN :', output[:, t].detach().numpy())
...     print()

Шаг времени 0 =>
    Вход          : [[1. 1. 1. 1. 1.]]
    Скрытый      : [[-0.4701929 0.5863904]]
    Выход (ручной): [[-0.3519801 0.52525216]]
    Выход RNN    : [[-0.3519801 0.52525216]]

Шаг времени 1 =>
    Вход          : [[2. 2. 2. 2. 2.]]
    Скрытый      : [[-0.88883156 1.2364397 ]]
    Выход (ручной): [[-0.68424344 0.76074266]]
    Выход RNN    : [[-0.68424344 0.76074266]]
```

Шаг времени 2 =>

Вход	: [[3. 3. 3. 3. 3.]]
Скрытый	: [[-1.3074701 1.886489]]
Выход (ручной)	: [[-0.8649416 0.90466356]]
Выход RNN	: [[-0.8649416 0.90466356]]

В нашем ручном прямом вычислении мы задействовали функцию активации гиперболического тангенса (\tanh), поскольку она также используется в RNN (активация по умолчанию). Как видно из приведенного вывода, результаты ручных прямых вычислений точно совпадают с выходными данными уровня RNN на каждом временном шаге. Надеюсь, это практическое задание открыло вам секреты работы рекуррентных сетей.

15.2.4. Проблемы изучения дальних взаимодействий

Обратное распространение во времени (BPTT), о котором кратко упоминалось ранее, создает некоторые новые проблемы. Из-за мультипликативного фактора $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ при вычислении градиентов функции потерь возникают явления так называемого *исчезающего и взрывающегося градиента*.

Эти явления демонстрируют примеры на рис. 15.8, где для простоты показана RNN только с одним скрытым элементом.

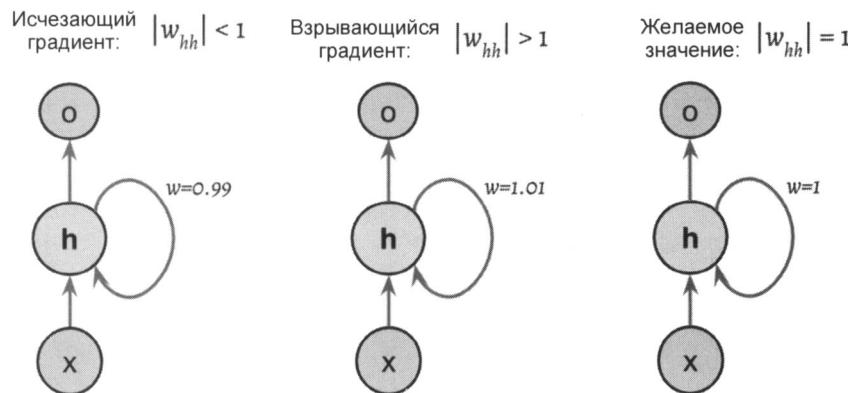


Рис. 15.8. Проблемы при вычислении градиентов функции потерь

По сути, $\frac{\partial h^{(t)}}{\partial h^{(k)}}$ включает в себя $t - k$ умножений, — следовательно, умножение веса w на самого себя $t - k$ раз приводит к результату w^{t-k} . В результате, если $|w| < 1$, результат умножений становится очень малым, когда $t - k$ велико. С другой стороны, если вес рекуррентного ребра $|w| > 1$, то w^{t-k} становится очень большим, когда $t - k$ велико. Большое значение $t - k$ означает так называемое *далнее взаимодействие*. Простое буквальное решение, позволяющее избежать исчезновения или взрыва градиентов, достигается при $|w| = 1$ ².

² Если вы хотите изучить этот вопрос более подробно, прочтите статью «On the difficulty of training recurrent neural networks» by R. Pascanu, T. Mikolov, and Y. Bengio, 2012, <https://arxiv.org/pdf/1211.5063.pdf>.

На практике есть как минимум три решения этой проблемы:

- ◆ отсечение градиента;
- ◆ усеченное обратное распространение во времени (Truncated Backpropagation Through Time, TBPTT);
- ◆ сети долгой кратковременной памяти (Long Short-Term Memory, LSTM).

Используя отсечение градиента, мы указываем *отсечку*, т. е. пороговое значение для градиентов, и присваиваем значение отсечки градиентам, которые превышают это значение. Напротив, TBPTT просто ограничивает количество временных шагов, на которые сигнал может распространяться в обратном направлении после каждого прямого прохода. Например, даже если последовательность состоит из 100 элементов или шагов, мы можем применить обратное распространение только для самых последних 20 временных шагов.

Хотя и отсечение градиента, и TBPTT могут решить проблему взрывающегося градиента, усеченное распространение ограничивает количество шагов, на которые градиент может эффективно возвращаться и должным образом обновлять веса. Подход LSTM, разработанный в 1997 году Зеппом Хохрайтером и Юргеном Шмидхубером, более успешно решает проблемы исчезающего и взрывающегося градиента при моделировании долгосрочных зависимостей за счет использования ячеек памяти. Далее мы обсудим LSTM более подробно.

15.2.5. Ячейки долгой краткосрочной памяти

Как сказано ранее, сети LSTM были разработаны для решения проблемы исчезающего градиента³. Функциональным узлом LSTM является *ячейка памяти*, которая по существу представляет или заменяет скрытый слой стандартных RNN.

В каждой ячейке памяти есть рекуррентное ребро с весом $w = 1$ для преодоления проблем исчезающего и взрывающегося градиента. Значения, связанные с этим рекуррентным ребром, вместе называются *состоянием ячейки* (cell state). Развёрнутая структура современной ячейки LSTM показана на рис. 15.9.

Обратите внимание, что для получения состояния ячейки на текущем временнóм шаге $C^{(t)}$ из состояния на предыдущем временнóм шаге $C^{(t-1)}$ не применяется умножение на какой-либо весовой коэффициент. Потоком информации в этой ячейке памяти управляет несколько вычислительных блоков, которые будут описаны далее. На рис. 15.9 символ \odot означает *поэлементное произведение* (поэлементное умножение), а символ \oplus — *поэлементное суммирование* (поэлементное сложение). Кроме того, $x^{(t)}$ обозначает входные данные в момент времени t , а $h^{(t-1)}$ указывает на скрытые узлы в момент времени $t - 1$. Четыре прямоугольника содержат обозначения функций активации: либо сигмоидной функции (σ), либо \tanh , и набор весов — эти блоки применяют линейную комбинацию, выполняя умножение матрицы на вектор на своих входах (которые равны $h^{(t-1)}$ и $x^{(t)}$). Эти вычислительные блоки с сигмоидными функциями активации, выходы которых проходят через операцию \odot , называются *вентилями*, или *гейтами*.

³ См. «Long Short-Term Memory» by S. Hochreiter and J. Schmidhuber, Neural Computation, 9(8): 1735–1780, 1997.

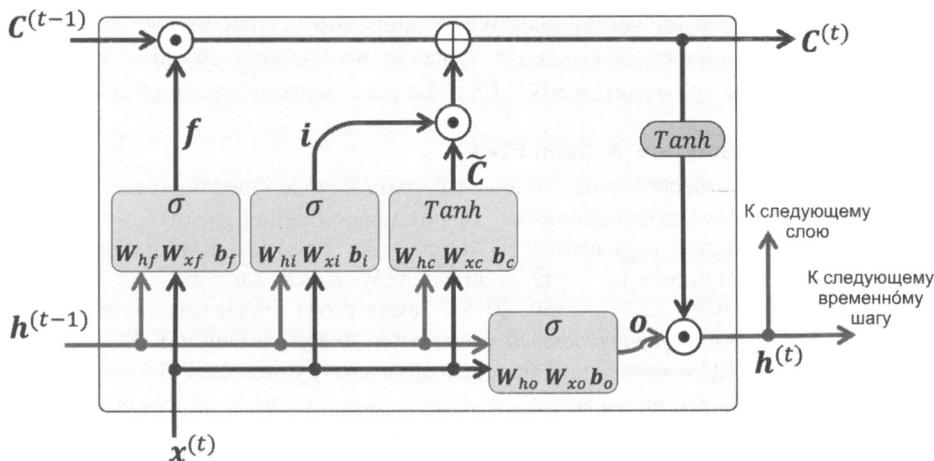


Рис. 15.9. Развернутая структура ячейки LSTM

В ячейке LSTM есть три разных типа гейтов, которые известны как *гейт утраты*, или забывания (forget gate), *гейт входа*, или входной (input gate), и *гейт выхода* (output gate).

Гейт утраты f_t позволяет ячейке памяти сбрасывать состояние, не увеличивая его до бесконечности. Фактически гейт утраты определяет, какую информацию можно пропустить, а какую скрыть. Гейт f_t вычисляется следующим образом:

$$f_t = \sigma(W_{xf}x^{(t)} + W_{hf}h^{(t-1)} + b_f).$$

Заметим, что гейт утраты не был частью исходного варианта ячейки LSTM — он был добавлен несколько лет спустя для улучшения исходной модели⁴.

Входной гейт i_t и значение-кандидат \tilde{C}_t отвечают за обновление состояния ячейки. Они вычисляются следующим образом:

$$\begin{aligned} i_t &= \sigma(W_{xi}x^{(t)} + W_{hi}h^{(t-1)} + b_i) \\ \tilde{C}_t &= \tanh(W_{xc}x^{(t)} + W_{hc}h^{(t-1)} + b_c). \end{aligned}$$

Состояние ячейки в момент времени t вычисляется так:

$$C^{(t)} = (C^{(t-1)} \odot f_t) \oplus (i_t \odot \tilde{C}_t).$$

Выходной гейт o_t решает, как обновлять значения скрытых узлов:

$$o_t = \sigma(W_{xo}x^{(t)} + W_{ho}h^{(t-1)} + b_o).$$

С учетом приведенных выражений скрытые узлы на текущем временном шаге вычисляются следующим образом:

$$h^{(t)} = o_t \odot \tanh(C^{(t)}).$$

Структура ячейки LSTM и лежащие в ее основе вычисления могут показаться очень запутанными и сложными для реализации. Однако хорошая новость заключается в том,

⁴ См. «Learning to Forget: Continual Prediction with LSTM» by F. Gers, J. Schmidhuber, and F. Cummins, Neural Computation 12, 2451–2471, 2000.

что PyTorch уже предлагает все необходимые операции в оптимизированных функциях-оболочках, что позволяет нам легко и эффективно определять наши ячейки LSTM. Позже в этой главе мы применим RNN и LSTM к реальным наборам данных.



Другие продвинутые модели RNN

Модели LSTM обеспечивают базовый подход к моделированию долгосрочных зависимостей в последовательностях. Тем не менее важно отметить, что в литературе описано множество вариантов LSTM (см. «An Empirical Exploration of Recurrent Network Architectures» by Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever, Proceedings of ICML, 2342–2350, 2015). Также стоит упомянуть более свежий подход — управляемый рекуррентный блок (Gated Recurrent Unit, GRU), предложенный в 2014 году. GRU имеют более простую архитектуру, чем LSTM, — следовательно, они более эффективны в вычислительном отношении, а их производительность в некоторых задачах, таких как моделирование полифонической музыки, сравнима с LSTM. Если вам интересно узнать больше об этих современных архитектурах RNN, обратитесь к статье «Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling» by Junyoung Chung and others, 2014, <https://arxiv.org/pdf/1412.3555v1.pdf>.

15.3. Реализация RNN для моделирования последовательностей в PyTorch

Ознакомившись с теоретическими основами RNN, мы готовы перейти к более практической части этой главы — реализации RNN в PyTorch. В оставшейся части главы мы будем применять RNN к двум распространенным задачам:

1. Анализ эмоциональной окраски (sentiment analysis).
2. Языковое моделирование (language modeling).

Эти два проекта, которые мы рассмотрим далее, очень увлекательны, но в то же время весьма сложны. Поэтому вместо того, чтобы разбирать весь код целиком, мы разобьем реализацию на несколько шагов и подробно обсудим каждый шаг. Если вам нравится начинать с общего понимания и вы хотите увидеть весь код, прежде чем углубляться в обсуждение, можете сразу взглянуть на реализацию кода.

15.3.1. Проект № 1: предсказание эмоциональной окраски отзывов на фильмы IMDb

В главе 8 мы говорили о том, что анализ эмоциональной окраски (тональности) фактически означает анализ мнения, выраженного в предложении или текстовом документе. В этом разделе мы реализуем многослойную RNN для анализа тональности текста с использованием архитектуры «многие к одному», а в следующем — реализуем RNN «многие ко многим» для приложения языкового моделирования. Хотя выбранные примеры намеренно просты и предназначены для лишь ознакомления с основными концепциями RNN, языковое моделирование имеет широкий спектр интересных применений — таких как создание чат-ботов, дающих компьютерам возможность напрямую взаимодействовать с людьми, общаясь с ними на естественном языке.

Подготовка данных отзывов на фильмы

В главе 8 мы предварительно обработали и очистили набор данных отзывов. Сейчас мы сделаем то же самое. Но сначала установим `torchtext` (в апреле 2023 года была доступна версия 0.15.1) с помощью команды:

```
pip install torchtext
```

после чего импортируем необходимые модули и считаем данные следующим образом:

```
>>> from torchtext.datasets import IMDB  
>>> train_dataset = IMDB(split='train')  
>>> test_dataset = IMDB(split='test')
```

Каждый набор состоит из 25 тыс. примеров, а каждый пример — из двух элементов: метки тональности, которую в итоге должна научиться предсказывать модель (`neg` означает негативную тональность, а `pos` — позитивную), и текста отзыва на фильм (входные признаки). Текстовый компонент этих отзывов представляет собой последовательности слов, и модель RNN классифицирует каждую последовательность как положительный (1) или отрицательный (0) отзыв.

Однако, прежде чем мы сможем передать данные в модель RNN, нам нужно применить несколько шагов предварительной обработки:

1. Разделить набор обучающих данных на отдельные разделы: для обучения и для валидации.
2. Определить уникальные слова в наборе обучающих данных.
3. Сопоставить каждое уникальное слово с уникальным целым числом и закодировать текст отзыва в виде целых чисел (индексов каждого уникального слова).
4. Разделить набор данных на мини-пакеты в качестве входных данных для модели.

Приступим к первому шагу: создадим поднаборы для обучения и валидации из набора `train_dataset`, который мы считали ранее:

```
>>> ## Шаг 1: создание наборов данных  
>>> from torch.utils.data.dataset import random_split  
>>> torch.manual_seed(1)  
>>> train_dataset, valid_dataset = random_split(  
...     list(train_dataset), [20000, 5000])
```

Как уже отмечалось ранее, исходный набор обучающих данных содержит 25 тыс. примеров. Мы случайным образом выбираем из него 20 тыс. примеров для обучения и 5 тыс. — для валидации.

Чтобы подготовить данные для ввода в неройсеть, нам нужно закодировать их в числовые значения, как было сказано в *шагах 2 и 3*. Для этого мы сначала найдем уникальные слова (токены) в обучающем наборе данных. Хотя поиск уникальных токенов — это процесс, для которого мы можем использовать наборы данных Python, более эффективным может стать использование класса `Counter` из пакета `collections`, который является частью стандартной библиотеки Python.

В следующем коде мы создадим новый объект `Counter` (`token_counts`), который будет собирать уникальные частоты слов. Заметьте, что в этом конкретном приложении (и в отличие от модели мешка слов) нас интересует только набор уникальных слов,

и нам не потребуется подсчет слов, который создается как побочный продукт. Чтобы разделить текст на слова (или токены), мы повторно воспользуемся функцией `tokenizer`, разработанной нами в главе 8, которая заодно удаляет HTML-разметку, а также знаки препинания и другие небуквенные символы.

Код для сбора уникальных токенов выглядит следующим образом:

```
>>> ## Шаг 2: поиск уникальных токенов (слов)
>>> import re
>>> from collections import Counter, OrderedDict
>>>
>>> def tokenizer(text):
...     text = re.sub('<[^>]*>', '', text)
...     emoticons = re.findall(
...         '(?:[:|;|=](?:-)?(?:\:|\\(|D|P)', text.lower())
...     )
...     text = re.sub('[\W]+', ' ', text.lower()) +\
...         ''.join(emoticons).replace('-', '')
...     tokenized = text.split()
...     return tokenized
>>>
>>> token_counts = Counter()
>>> for label, line in train_dataset:
...     tokens = tokenizer(line)
...     token_counts.update(tokens)
>>> print('Размер словаря:', len(token_counts))
Размер словаря: 69023
```

Если вы хотите узнать больше о классе `Counter`, обратитесь к его документации по адресу: <https://docs.python.org/3/library/collections.html#collections.Counter>.

Далее мы сопоставим каждое уникальное слово с уникальным целым числом. Это можно сделать вручную с помощью словаря Python, где ключи представляют собой уникальные токены (слова), а значение, связанное с каждым ключом, является уникальным целым числом. Однако пакет `torchtext` уже предоставляет класс `Vocab`, который можно использовать для создания такого сопоставления и кодирования всего набора данных. Так что сначала мы создадим объект `vocab`, передав ему токены сопоставления упорядоченного словаря с соответствующими частотами их появления (упорядоченный словарь — это отсортированный `token_counts`). Затем добавим в словарь два специальных токена: `<pad>` (от слова `padding`, заполнение) и `<unk>` (от слова `unknown`, неизвестный):

```
>>> ## Шаг 3: кодируем каждый уникальный токен целым числом
>>> from torchtext.vocab import vocab
>>> sorted_by_freq_tuples = sorted(
...     token_counts.items(), key=lambda x: x[1], reverse=True
... )
>>> ordered_dict = OrderedDict(sorted_by_freq_tuples)
>>> vocab = vocab(ordered_dict)
>>> vocab.insert_token("<pad>", 0)
>>> vocab.insert_token("<unk>", 1)
>>> vocab.set_default_index(1)
```

Чтобы продемонстрировать использование объекта vocab, преобразуем пример текста в список целочисленных значений:

```
>>> print([vocab[token] for token in ['this', 'is',
...     'an', 'example']])
[11, 7, 35, 457]
```

Нужно помнить, что в данных для валидации или тестирования могут встретиться токены, которых нет в обучающих данных и, следовательно, не включенных в сопоставление. Если у нас есть q токенов (т. е. размер token_counts, переданный в Vocab, который в нашем случае равен 69 023), то всем токенам, которые не были замечены ранее и, следовательно, не включены в token_counts, будет присвоено целое число 1 (заполнитель для неизвестного токена). Иными словами, индекс 1 зарезервирован для неизвестных слов. Другим зарезервированным значением является целое число 0, которое служит так называемым *токеном заполнения* для корректировки длины последовательности. Позже, при построении модели RNN в PyTorch, мы рассмотрим этот нулевой заполнитель более подробно.

Теперь определим функции: text_pipeline — для соответствующего преобразования каждого текста в наборе данных и label_pipeline — для преобразования каждой метки в 1 или 0:

```
>>> ## Шаг 3-А: определение функций для преобразования
>>> text_pipeline = \
...     lambda x: [vocab[token] for token in tokenizer(x)]
>>> label_pipeline = lambda x: 1. if x == 'pos' else 0.
```

Мы будем генерировать пакеты примеров с помощью DataLoader и передавать конвойеры обработки данных, объявленные ранее, в аргумент collate_fn. Функции кодирования текста и преобразования метки мы обернем в функцию collate_batch:

```
>>> ## Шаг 3-В: обертка функций кодирования и преобразования
... def collate_batch(batch):
...     label_list, text_list, lengths = [], [], []
...     for _label, _text in batch:
...         label_list.append(label_pipeline(_label))
...         processed_text = torch.tensor(text_pipeline(_text),
...                                         dtype=torch.int64)
...         text_list.append(processed_text)
...         lengths.append(processed_text.size(0))
...     label_list = torch.tensor(label_list)
...     lengths = torch.tensor(lengths)
...     padded_text_list = nn.utils.rnn.pad_sequence(
...         text_list, batch_first=True)
...     return padded_text_list, label_list, lengths
>>>
>>> ## Получение небольшого пакета
>>> from torch.utils.data import DataLoader
>>> dataloader = DataLoader(train_dataset, batch_size=4,
...                           shuffle=False, collate_fn=collate_batch)
```

До сих пор мы преобразовывали последовательности слов в последовательности целых чисел, а метки положительного или отрицательного отзыва — соответственно в 1

или 0. Однако есть одна проблема, которую нам нужно решить: последовательности в настоящее время имеют разную длину (как показано в результате выполнения следующего кода для четырех примеров). Хотя, как правило, RNN могут обрабатывать последовательности разной длины, нам все равно нужно сделать так, чтобы все последовательности в мини-пакете имели одинаковую длину — для эффективного хранения их в тензоре.

PyTorch предоставляет удобный метод `pad_sequence()`, автоматически дополняющий последовательные элементы, которые должны быть объединены в пакет, значениями-заполнителями (0), чтобы все последовательности в пакете имели одинаковый размер. В предыдущем коде мы уже создали загрузчик небольших пакетов из обучающего набора данных и применили функцию `collate_batch`, которая сама включает вызов `pad_sequence()`.

Чтобы проиллюстрировать, как работает заполнение, мы возьмем первый пакет и выведем размеры отдельных элементов до объединения их в мини-пакеты, а также размеры полученных мини-пакетов:

```
>>> text_batch, label_batch, length_batch = next(iter(dataloader))
>>> print(text_batch)
tensor([[ 35, 1742,      7,    449,    723,      6,   302,      4,
...
0,      0,      0,      0,      0,      0,      0]],

>>> print(label_batch)
tensor([1., 1., 1., 0.])
>>> print(length_batch)
tensor([165, 86, 218, 145])
>>> print(text_batch.shape)
torch.Size([4, 218])
```

Выведенные размеры тензоров свидетельствуют о том, что количество столбцов в первом пакете равно 218 — это получается в результате объединения первых четырех примеров в один пакет и использования максимального размера этих примеров. Это также означает, что остальные три примера (длина которых составляет 165, 86 и 145 соответственно) в полученном пакете дополнены настолько, насколько это необходимо, чтобы соответствовать указанному размеру.

Наконец, разделим все три набора данных на загрузчики данных с размером пакета 32:

```
>>> batch_size = 32
>>> train_dl = DataLoader(train_dataset, batch_size=batch_size,
...                         shuffle=True, collate_fn=collate_batch)
>>> valid_dl = DataLoader(valid_dataset, batch_size=batch_size,
...                         shuffle=False, collate_fn=collate_batch)
>>> test_dl = DataLoader(test_dataset, batch_size=batch_size,
...                        shuffle=False, collate_fn=collate_batch)
```

Теперь данные представлены в подходящем формате для модели RNN, которая будет реализована далее. Однако сначала мы обсудим встраивание признаков, которое является необязательным, но настоятельно рекомендуемым этапом предварительной обработки, служащим для уменьшения размерности векторов слов.

Слой встраивания для кодирования предложений

Во время подготовки данных на предыдущем этапе мы сгенерировали последовательности одинаковой длины. Элементами этих последовательностей были целые числа, соответствующие индексам уникальных слов. Эти индексы слов могут быть преобразованы во входные признаки несколькими различными способами. Один из простых прямолинейных способов — применить унитарное кодирование для преобразования индексов в векторы из нулей и единиц. А затем каждое слово сопоставить с вектором, размер которого равен количеству уникальных слов во всем наборе данных. Учитывая, что количество уникальных слов (размер словаря) может быть порядка $10^4\text{--}10^5$, что также будет равно количеству входных признаков, модель, обученная на таких признаках, может страдать от проклятия размерности. Кроме того, эти признаки очень разреженные, поскольку нулю равны все элементы вектора, кроме одного.

Более элегантный подход состоит в том, чтобы сопоставить каждое слово с вектором фиксированного размера с вещественными элементами (не обязательно целыми числами). В отличие от векторов с унитарным кодированием, мы можем использовать векторы конечного размера для представления бесконечного количества действительных чисел. (Теоретически мы можем извлечь бесконечные действительные числа из заданного интервала — например, $[-1, 1]$.)

В этом заключается идея встраивания — метода изучения признаков, который мы можем использовать здесь для автоматического изучения существенных признаков для представления слов в нашем наборе данных. Зная количество уникальных слов n_{words} , мы можем выбрать размер векторов встраивания (также известный как размерность встраивания), намного меньший, чем количество уникальных слов ($embedding_dim << n_{words}$), чтобы представить весь словарь как входные признаки.

Преимущества встраивания по сравнению с унитарным кодированием заключаются в следующем:

- ◆ уменьшение размерности пространства признаков для устранения эффекта проклятия размерности;
- ◆ извлечение существенных признаков, поскольку слой встраивания в нейросети можно оптимизировать (или обучить).

На рис. 15.10 показано, как работает встраивание путем сопоставления индексов токенов с обучаемой матрицей встраивания.

Если у нас набор токенов размером $n + 2$ (n — размер набора токенов, плюс индекс 0 зарезервирован для заполнителя, а 1 — для слов, отсутствующих в наборе токенов), будет создана матрица встраивания размером $(n + 2) \times embedding_dim$, где каждая ее строка представляет числовые признаки, связанные с токеном. Следовательно, когда в качестве входных данных для встраивания задан целочисленный индекс i , алгоритм будет искать соответствующую строку матрицы с индексом i и возвращать числовые признаки. Матрица встраивания служит входным слоем для наших моделей. На практике создать слой встраивания можно просто с помощью `nn.Embedding`. Давайте рассмотрим пример, в котором мы создадим слой встраивания и применим его к партии из двух образцов:

```
>>> embedding = nn.Embedding(  
...     num_embeddings=10,
```

```

...
    embedding_dim=3,
...
    padding_idx=0)
>>> # пакет из 2 примеров по 4 индекса каждый
>>> text_encoded_input = torch.LongTensor([[1,2,4,5],[4,3,2,0]])
>>> print(embedding(text_encoded_input))
tensor([[-0.7027,  0.3684, -0.5512],
       [-0.4147,  1.7891, -1.0674],
       [ 1.1400,  0.1595, -1.0167],
       [ 0.0573, -1.7568,  1.9067]],

      [[ 1.1400,  0.1595, -1.0167],
       [-0.8165, -0.0946, -0.1881],
       [-0.4147,  1.7891, -1.0674],
       [ 0.0000,  0.0000,  0.0000]]], grad_fn=<EmbeddingBackward>)

```

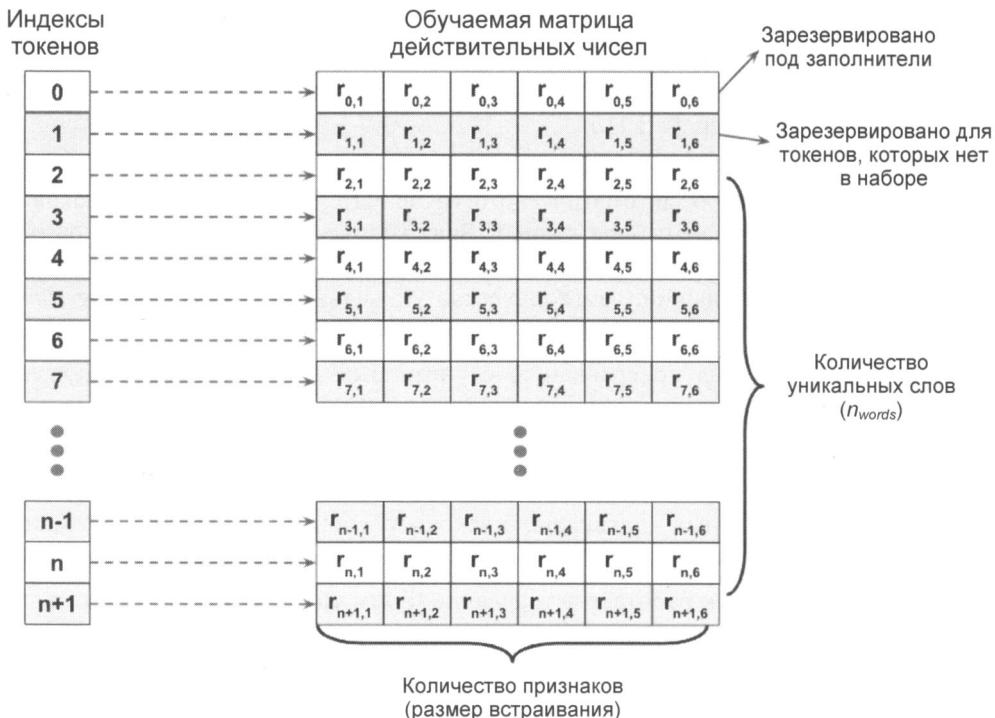


Рис. 15.10. Схематическое представление того, как работает встраивание

Входные данные для этой модели (слой встраивания) должны иметь ранг 2 с размером $batchsize \times input_length$, где $batchsize$ — размер пакета, а $input_length$ — длина последовательностей (здесь 4). Например, входная последовательность в мини-пакете может быть $<1, 5, 9, 2>$, где каждый ее элемент является индексом уникальных слов. Выходные данные будут иметь размерность $batchsize \times input_length \times embedding_dim$, где $embedding_dim$ — размер встраиваемых признаков (здесь установлено значение 3). Другой аргумент, предоставляемый слою встраивания, — $num_embeddings$, представляет собой уникальные целочисленные значения, которые модель получит в качестве вход-

ных данных (например, $n + 2$, — здесь установлено значение 10). Следовательно, матрица встраивания в этом случае имеет размер 10×3 .

Аргумент `padding_idx` указывает индекс маркера для заполнения (здесь 0), который, если он указан, не будет вносить вклад в градиент во время обучения. В нашем примере длина исходной последовательности второго примера равна 3, и мы дополнили ее еще одним элементом 0. Таким образом, выход встраивания для дополненного элемента равен $[0, 0, 0]$.

Построение модели RNN

Теперь мы готовы построить модель RNN. Используя класс `nn.Module`, мы можем объединить слой встраивания, рекуррентные слои RNN и полносвязные нерекуррентные слои. Для рекуррентных слоев нам можно взять любую из следующих реализаций:

- ◆ RNN — обычный слой RNN, т. е. полносвязный рекуррентный слой;
- ◆ LSTM — RNN с долгой краткосрочной памятью, которая полезна для захвата долгосрочных зависимостей;
- ◆ GRU — рекуррентный слой с управляемым рекуррентным блоком в качестве альтернативы LSTM⁵.

Чтобы продемонстрировать построение многослойной модели RNN с использованием одного из этих рекуррентных слоев, в следующем примере мы создадим модель с двумя рекуррентными слоями типа `RNN`. В конце мы добавим нерекуррентный полносвязный слой в качестве выходного слоя, который будет возвращать одно выходное значение в качестве прогноза:

```
>>> class RNN(nn.Module):
...     def __init__(self, input_size, hidden_size):
...         super().__init__()
...         self.rnn = nn.RNN(input_size, hidden_size, num_layers=2,
...                           batch_first=True)
...         # self.rnn = nn.GRU(input_size, hidden_size, num_layers,
...         #                   batch_first=True)
...         # self.rnn = nn.LSTM(input_size, hidden_size, num_layers,
...         #                   batch_first=True)
...         self.fc = nn.Linear(hidden_size, 1)
...
...     def forward(self, x):
...         _, hidden = self.rnn(x)
...         out = hidden[-1, :, :] # мы используем конечное скрытое состояние
...                               # из последнего скрытого слоя как
...                               # входные данные полносвязного слоя
...         out = self.fc(out)
...         return out
>>>
```

⁵ Как предложено в книге «Learning Phrase Representations Using RNN Encoder–Decoder for Statistical Machine Translation» by K. Cho et al., 2014, <https://arxiv.org/abs/1406.1078v3>.

```
>>> model = RNN(64, 32)
>>> print(model)
>>> model(torch.randn(5, 3, 64))
RNN(
  (rnn): RNN(64, 32, num_layers=2, batch_first=True)
  (fc): Linear(in_features=32, out_features=1, bias=True)
)
tensor([[ 0.0010],
       [ 0.2478],
       [ 0.0573],
       [ 0.1637],
       [-0.0073]], grad_fn=<AddmmBackward>)
```

Как видите, построить модель RNN с использованием рекуррентных слоев довольно просто. В следующем разделе мы вернемся к нашей задаче анализа тональности текста и построим модель RNN для ее решения.

Построение модели RNN для задачи анализа тональности

Поскольку у нас очень длинные последовательности, чтобы учесть протяженные эффекты, мы применим слой LSTM. Создание модели RNN для анализа тональности (эмоциональной окраски) текста мы начнем со слоя встраивания, производящего встраивание слов с размером признака 20 (`embed_dim=20`). Затем добавим рекуррентный слой типа LSTM, после чего добавим полносвязный слой в качестве скрытого слоя и еще один полносвязный слой в качестве выходного слоя, который в виде прогноза будет возвращать одно значение вероятности принадлежности к классу через активацию логистической сигмоиды:

```
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim, rnn_hidden_size,
...                  fc_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(vocab_size,
...                                       embed_dim,
...                                       padding_idx=0)
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                           batch_first=True)
...         self.fc1 = nn.Linear(rnn_hidden_size, fc_hidden_size)
...         self.relu = nn.ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths):
...         out = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False, batch_first=True
...         )
...         out, (hidden, cell) = self.rnn(out)
...         out = hidden[-1, :, :]
...         out = self.fc1(out)
```

```

...
    out = self.relu(out)
...
    out = self.fc2(out)
...
    out = self.sigmoid(out)
...
    return out
>>>
>>> vocab_size = len(vocab)
>>> embed_dim = 20
>>> rnn_hidden_size = 64
>>> fc_hidden_size = 64
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
                  rnn_hidden_size, fc_hidden_size)
>>> model
RNN(
  (embedding): Embedding(69025, 20, padding_idx=0)
  (rnn): LSTM(20, 64, batch_first=True)
  (fc1): Linear(in_features=64, out_features=64, bias=True)
  (relu): ReLU()
  (fc2): Linear(in_features=64, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)

```

Далее мы разработаем функцию `train`, которая обучает модель на заданном наборе данных в течение одной эпохи и возвращает точность классификации и потери:

```

>>> def train(dataloader):
...
    model.train()
...
    total_acc, total_loss = 0, 0
...
    for text_batch, label_batch, lengths in dataloader:
        optimizer.zero_grad()
        pred = model(text_batch, lengths)[:, 0]
        loss = loss_fn(pred, label_batch)
        loss.backward()
        optimizer.step()
...
        total_acc += (
            (pred >= 0.5).float() == label_batch
            ).float().sum().item()
...
        total_loss += loss.item()*label_batch.size(0)
...
    return total_acc/len(dataloader.dataset), \
           total_loss/len(dataloader.dataset)

```

Аналогично разработаем функцию оценки для измерения производительности модели на текущем наборе данных:

```

>>> def evaluate(dataloader):
...
    model.eval()
...
    total_acc, total_loss = 0, 0
...
    with torch.no_grad():
...
        for text_batch, label_batch, lengths in dataloader:
            pred = model(text_batch, lengths)[:, 0]
            loss = loss_fn(pred, label_batch)

```

```

...
    total_acc += (
        (pred>=0.5).float() == label_batch
    ).float().sum().item()
...
    total_loss += loss.item()*label_batch.size(0)
...
    return total_acc/len(dataloader.dataset), \
           total_loss/len(dataloader.dataset)
...

```

На следующем шаге мы создадим функцию потерь и оптимизатора (оптимизатор Adam). Для бинарной классификации с одним выводом вероятности принадлежности к классу мы используем в качестве функции потерь бинарную потерю перекрестной энтропии (BCELoss):

```

>>> loss_fn = nn.BCELoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

```

Теперь обучим модель на протяжении 10 эпох и отобразим результаты обучения и валидации:

```

>>> num_epochs = 10
>>> torch.manual_seed(1)
>>> for epoch in range(num_epochs):
...     acc_train, loss_train = train(train_dl)
...     acc_valid, loss_valid = evaluate(valid_dl)
...     print(f'Эпоха {epoch} точность: {acc_train:.4f}')
...             f' val_accuracy: {acc_valid:.4f}')
Эпоха 0 точность: 0.5843 val_accuracy: 0.6240
Эпоха 1 точность: 0.6364 val_accuracy: 0.6870
Эпоха 2 точность: 0.8020 val_accuracy: 0.8194
Эпоха 3 точность: 0.8730 val_accuracy: 0.8454
Эпоха 4 точность: 0.9092 val_accuracy: 0.8598
Эпоха 5 точность: 0.9347 val_accuracy: 0.8630
Эпоха 6 точность: 0.9507 val_accuracy: 0.8636
Эпоха 7 точность: 0.9655 val_accuracy: 0.8654
Эпоха 8 точность: 0.9765 val_accuracy: 0.8528
Эпоха 9 точность: 0.9839 val_accuracy: 0.8596

```

Обучив эту модель в течение 10 эпох, оценим ее на тестовых данных:

```

>>> acc_test, _ = evaluate(test_dl)
>>> print(f'test_accuracy: {acc_test:.4f}')
test_accuracy: 0.8512

```

Тест показал точность 85%. Заметим, что этот результат не самый лучший по сравнению с современными методами, применяемыми к набору данных IMDb. Но наша цель здесь состояла в том, чтобы просто продемонстрировать создание RNN в PyTorch.

Подробнее о двунаправленной RNN

Давайте теперь установим значение параметра `bidirectional` слоя LSTM в `True`, что заставит рекуррентный слой проходить через входные последовательности с обоих направлений: от начала до конца, а также в обратном направлении:

```
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim, rnn_hidden_size, fc_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(
...             vocab_size, embed_dim, padding_idx=0
...         )
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                            batch_first=True, bidirectional=True)
...         self.fc1 = nn.Linear(rnn_hidden_size*2, fc_hidden_size)
...         self.relu = nn.ReLU()
...         self.fc2 = nn.Linear(fc_hidden_size, 1)
...         self.sigmoid = nn.Sigmoid()
...
...     def forward(self, text, lengths):
...         out = self.embedding(text)
...         out = nn.utils.rnn.pack_padded_sequence(
...             out, lengths.cpu().numpy(), enforce_sorted=False, batch_first=True
...         )
...         out, (hidden, cell) = self.rnn(out)
...         out = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)
...         out = self.fc1(out)
...         out = self.relu(out)
...         out = self.fc2(out)
...         out = self.sigmoid(out)
...         return out
>>>
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim,
...               rnn_hidden_size, fc_hidden_size)
>>> model
RNN(
    (embedding): Embedding(69025, 20, padding_idx=0)
    (rnn): LSTM(20, 64, batch_first=True, bidirectional=True)
    (fc1): Linear(in_features=128, out_features=64, bias=True)
    (relu): ReLU()
    (fc2): Linear(in_features=64, out_features=1, bias=True)
    (sigmoid): Sigmoid()
)
```

Двунаправленный слой RNN выполняет два прохода по каждой входной последовательности: прямой и обратный (будьте внимательны, и не путайте их с прямым и обратным проходами в контексте обратного распространения ошибки). Результатирующие скрытые состояния этих прямых и обратных проходов обычно объединяются в одно скрытое состояние путем конкатенации. Возможны и другие способы слияния: суммирование, умножение (умножение результатов двух проходов) и усреднение (взятие среднего из двух результатов).

Можно также попробовать иные типы рекуррентных слоев — например, обычный RNN. Однако, как оказалось, модель, построенная с использованием обычных рекуррентных

слоев, не сможет достичь хороших результатов прогнозирования (даже на обучающих данных). Например, если вы попытаетесь заменить двунаправленный слой LSTM в приведенном коде односторонним слоем nn.RNN (вместо nn.LSTM) и обучите модель на полноразмерных последовательностях, вы можете заметить, что потери на протяжении обучения даже не уменьшаются. Причина в том, что последовательности в этом наборе данных слишком длинные, поэтому модель со слоем RNN не в состоянии изучить долгосрочные зависимости и может страдать от проблем с исчезновением или взрывом градиента.

15.3.2. Проект № 2: моделирование языка на уровне символов в PyTorch

Языковое моделирование — увлекательное применение для нейронных сетей, позволяющее машинам выполнять задачи, связанные с человеческим языком, — например, генерировать предложения на естественном языке⁶.

В модели, которую мы сейчас построим, входными данными является текстовый документ, и наша цель — разработать модель, которая может генерировать новый текст, похожий по стилю на входной документ. Примерами ввода для такой модели могут служить книга или компьютерная программа на каком-либо языке программирования.

В языковом моделировании на уровне символов ввод разбивается на последовательность символов, которые вводятся в нашу сеть по одному символу за раз. Сеть будет обрабатывать каждый новый символ в сочетании с памятью о ранее увиденных символах, чтобы предсказать следующий.

На рис. 15.11 показан пример моделирования языка на уровне символов (**EOS** здесь означает *end of sequence*, т. е. «конец последовательности»).

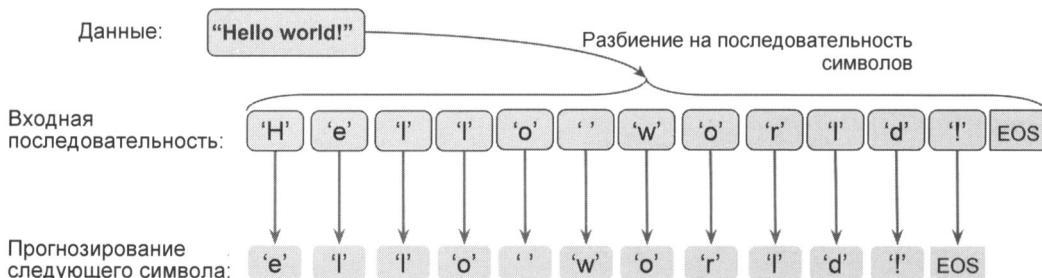


Рис. 15.11. Моделирование языка на уровне символов

Мы можем разбить решение этой задачи на три отдельных этапа: подготовка данных, построение модели RNN и выполнение предсказания следующего символа, а также выборки для создания нового текста.

⁶ Одно из интересных исследований в этой области представлено в статье «Generating Text with Recurrent Neural Networks» by Ilya Sutskever, James Martens, and Geoffrey E. Hinton, Proceedings of the 28th International Conference on Machine Learning (ICML-11), 2011, <https://pdfs.semanticscholar.org/93c2/0e38c85b69fc2d2eb314b3c1217913f7db11.pdf>.

Предварительная обработка набора данных

Приступим к подготовке данных для моделирования языка на уровне символов.

Чтобы получить исходные данные, посетите веб-сайт Project Gutenberg по адресу: <https://www.gutenberg.org/>, где размещены тысячи бесплатных электронных книг. Для нашего примера вы можете скачать в текстовом формате опубликованную в 1874 году книгу Жюля Верна «Таинственный остров» (<https://www.gutenberg.org/files/1268/1268-0.txt>).

Обратите внимание, что указанная ссылка приведет вас прямо на страницу загрузки. Если вы используете macOS или операционную систему Linux, то можете загрузить этот файл с помощью следующей команды в терминале:

```
curl -O https://www.gutenberg.org/files/1268/1268-0.txt
```

Если этот ресурс по каким-либо причинам окажется недоступным, имейте в виду, что копия искомого текста включена в каталог кода этой главы в репозитории кода книги по адресу: <https://github.com/rasbt/machine-learning-book>, а также в сопровождающий книгу файловый архив (см. файл ch15\1268-0.txt)

После успешной загрузки набора данных мы можем прочитать его в сеансе Python как обычный текст. Так что давайте, используя следующий код, прочитаем текст прямо из загруженного файла и удалим части с начала и с конца (они содержат специальные описания проекта Gutenberg). Затем создадим переменную Python `char_set`, которая представляет набор уникальных символов, присутствующих в этом тексте:

```
>>> import numpy as np
>>> ## Чтение и предварительная обработка текста
>>> with open('1268-0.txt', 'r', encoding="utf8") as fp:
...     text=fp.read()
>>> start_indx = text.find('THE MYSTERIOUS ISLAND')
>>> end_indx = text.find('End of the Project Gutenberg')
>>> text = text[start_indx:end_indx]
>>> char_set = set(text)
>>> print('Общая длина:', len(text))
Общая длина: 1112350
>>> print('Уникальных символов:', len(char_set))
Уникальных символов: 80
```

После загрузки и предварительной обработки текста мы получим последовательность общей длиной 1 112 350 символов, состоящую из 80 уникальных символов. Однако большинство нейросетевых библиотек и реализаций RNN не могут работать со входными данными в строковом формате, поэтому нам приходится преобразовывать текст в числовой формат. Для этого мы создадим простой словарь Python `char2int`, который сопоставляет каждый символ с целым числом. Нам также понадобится обратное сопоставление, чтобы преобразовать вывод нашей модели обратно в текст. Хотя обратное сопоставление можно сделать с помощью словаря, который связывает целочисленные ключи со значениями символов, использование массива NumPy и индексация массива для сопоставления индексов с этими уникальными символами более эффективны. На рис. 15.12 приведен пример преобразования символов в целые числа и наоборот для слов «Hello» и «world»:

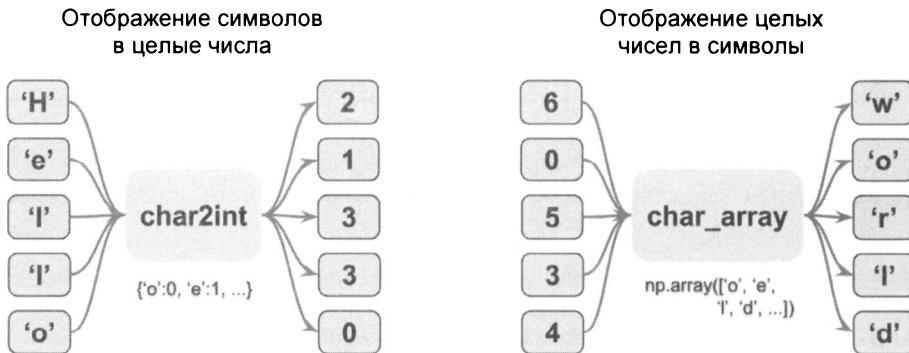


Рис. 15.12. Сопоставление символов и целых чисел

Построение словаря для сопоставления символов с целыми числами и обратное сопоставление посредством индексации массива NumPy, показанное на рис. 15.12, выглядит следующим образом:

```
>>> chars_sorted = sorted(char_set)
>>> char2int = {ch:i for i,ch in enumerate(chars_sorted)}
>>> char_array = np.array(chars_sorted)
>>> text_encoded = np.array(
...     [char2int[ch] for ch in text],
...     dtype=np.int32
... )
>>> print('Размер закодированного текста:', text_encoded.shape)
Размер закодированного текста: (1112350,)
>>> print(text[:15], '== Кодирование ==>', text_encoded[:15])
>>> print(text_encoded[15:21], '== Декодирование ==>',
...         ''.join(char_array[text_encoded[15:21]]))
THE MYSTERIOUS == Кодирование ==> [44 32 29 1 37 48 43 44 29 42 33 39 45 43 1]
[33 43 36 25 38 28] == Декодирование ==> ISLAND
```

Массив NumPy `text_encoded` содержит закодированные значения для всех символов в тексте. Выведем сопоставления первых пяти символов из этого массива:

```
>>> for ex in text_encoded[:5]:
...     print('{} -> {}'.format(ex, char_array[ex]))
44 -> T
32 -> H
29 -> E
1 ->
37 -> M
```

Теперь давайте отступим на шаг назад и посмотрим на общую картину того, что мы пытаемся сделать. Что касается задачи генерации текста, то ее можно сформулировать как задачу классификации.

Предположим, у нас есть набор неполных последовательностей текстовых символов, как показано на рис. 15.13.



Рис. 15.13. Прогнозирование следующего символа текстовой последовательности

Последовательности, показанные в левом поле, можем рассматривать как входные данные. Чтобы сгенерировать новый текст, необходимо разработать модель, которая может предсказать следующий символ заданной входной последовательности, где входная последовательность представляет неполный текст. Например, увидев последовательность «Deep Learn», модель должна предсказать «i» в качестве следующего символа. Поскольку у нас есть 80 уникальных символов, фактически это задача многоклассовой классификации.

Применяя подход многоклассовой классификации, мы можем итеративно генерировать новый текст, начиная с последовательности длиной 1 (т. е. одной буквы), как показано на рис. 15.14.

Чтобы реализовать задачу генерации текста в PyTorch, сначала урежем длину последовательности до 40. Это означает, что входной тензор x состоит из 40 токенов. На практике длина последовательности влияет на качество генерируемого текста. Более длинные последовательности обычно позволяют получить более осмысленные предложения. С более короткими последовательностями модель может сосредоточиться на правильном захвате отдельных слов, по большей части игнорируя контекст. Хотя более длинные последовательности обычно приводят к более осмысленным предложениям, модель RNN столкнется с проблемой отслеживания долгосрочных зависимостей. Таким образом, на практике поиск оптимального значения длины последовательности является задачей оптимизации гиперпараметров, которую приходится решать эмпирическим путем. В нашем случае мы собираемся выбрать длину 40, поскольку это выглядит как хороший компромисс.

Как вы можете видеть на рис. 15.14, входные данные x и цели y смешены на один символ. Следовательно, мы разделяем текст на фрагменты размером 41: первые 40 символов будут формировать входную последовательность x , а последние 40 — целевую последовательность y .

Мы уже сохранили весь закодированный текст в исходном порядке в `text_encoded`. А сейчас сначала создадим текстовые фрагменты, состоящие из 41 символа каждый. Затем избавимся от последнего фрагмента, если он короче 41 символа. В результате новый фрагментированный набор данных с именем `text_chunks` всегда будет содержать

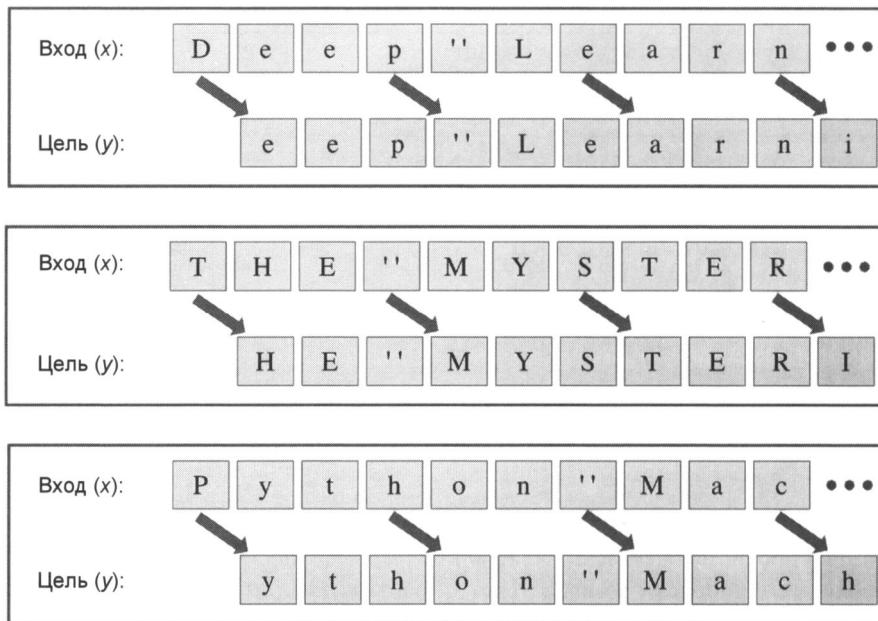


Рис. 15.14. Генерация продолжения текста с использованием подхода многоклассовой классификации

последовательности размером 41 символ. Фрагменты из 41 символа затем станут использоваться для построения последовательности x (т. е. входных данных), а также последовательности y (т. е. цели), которые будут иметь по 40 элементов каждая. Так, последовательность x будет состоять из элементов с номерами $[0, 1, \dots, 39]$. Кроме того, поскольку последовательность y окажется сдвинута на одну позицию относительно x , ее соответствующие индексы будут $[1, 2, \dots, 40]$. В завершение мы преобразуем результат в объект Dataset, применив самоопределенный класс Dataset:

```
>>> import torch
>>> from torch.utils.data import Dataset
>>> seq_length = 40
>>> chunk_size = seq_length + 1
>>> text_chunks = [text_encoded[i:i+chunk_size]
...                 for i in range(len(text_encoded)-chunk_size+1)]
>>> from torch.utils.data import Dataset
>>> class TextDataset(Dataset):
...     def __init__(self, text_chunks):
...         self.text_chunks = text_chunks
...
...     def __len__(self):
...         return len(self.text_chunks)
...
...     def __getitem__(self, idx):
...         text_chunk = self.text_chunks[idx]
...         return text_chunk[:-1].long(), text_chunk[1:].long()
>>>
>>> seq_dataset = TextDataset(torch.tensor(text_chunks))
```

Давайте взглянем на некоторые примеры последовательностей из этого преобразованного набора данных:

```
>>> for i, (seq, target) in enumerate(seq_dataset):
...     print(' Вход (x): ',
...           repr(''.join(char_array[seq])))
...     print('Цель (y): ',
...           repr(''.join(char_array[target])))
...     print()
...     if i == 1:
...         break
...
Input (x): 'THE MYSTERIOUS ISLAND ***\n\n\n\nProduced b'
Target (y): 'HE MYSTERIOUS ISLAND ***\n\n\n\nProduced by'

Input (x): 'HE MYSTERIOUS ISLAND ***\n\n\n\nProduced by'
Target (y): 'E MYSTERIOUS ISLAND ***\n\n\n\nProduced by '
```

Наконец, последний шаг в подготовке набора данных — преобразовать этот набор данных в мини-пакеты:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> seq_dl = DataLoader(seq_dataset, batch_size=batch_size,
...                      shuffle=True, drop_last=True)
```

Построение модели RNN символьного уровня

Теперь, когда набор данных готов, построить модель будет относительно просто:

```
>>> import torch.nn as nn
>>> class RNN(nn.Module):
...     def __init__(self, vocab_size, embed_dim, rnn_hidden_size):
...         super().__init__()
...         self.embedding = nn.Embedding(vocab_size, embed_dim)
...         self.rnn_hidden_size = rnn_hidden_size
...         self.rnn = nn.LSTM(embed_dim, rnn_hidden_size,
...                           batch_first=True)
...         self.fc = nn.Linear(rnn_hidden_size, vocab_size)
...
...     def forward(self, x, hidden, cell):
...         out = self.embedding(x).unsqueeze(1)
...         out, (hidden, cell) = self.rnn(out, (hidden, cell))
...         out = self.fc(out).reshape(out.size(0), -1)
...         return out, hidden, cell
...
...     def init_hidden(self, batch_size):
...         hidden = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         cell = torch.zeros(1, batch_size, self.rnn_hidden_size)
...         return hidden, cell
```

Заметим, что нам понадобятся логиты в качестве выходных данных модели, чтобы мы могли выбирать наиболее подходящие прогнозы модели для создания нового текста. Мы вернемся к этому моменту позже.

А сейчас мы можем указать параметры модели и создать модель RNN:

```
>>> vocab_size = len(char_array)
>>> embed_dim = 256
>>> rnn_hidden_size = 512
>>> torch.manual_seed(1)
>>> model = RNN(vocab_size, embed_dim, rnn_hidden_size)
>>> model
RNN(
    (embedding): Embedding(80, 256)
    (rnn): LSTM(256, 512, batch_first=True)
    (fc): Linear(in_features=512, out_features=80, bias=True)
)
```

Следующим шагом станет создание функции потерь и оптимизатора (оптимизатор Adam). Для многоклассовой классификации (количество классов `vocab_size=80`) с единственным логит-выводом для каждого целевого символа мы воспользуемся в качестве функции потерь `CrossEntropyLoss`:

```
>>> loss_fn = nn.CrossEntropyLoss()
>>> optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

Теперь обучим модель на 10 тыс. эпох. В каждую эпоху мы используем только один пакет, случайно выбранный из загрузчика данных `seq_dl`. Мы также будем отображать потери при обучении для каждого 500 эпох:

```
>>> num_epochs = 10000
>>> torch.manual_seed(1)
>>> for epoch in range(num_epochs):
...     hidden, cell = model.init_hidden(batch_size)
...     seq_batch, target_batch = next(iter(seq_dl))
...     optimizer.zero_grad()
...     loss = 0
...     for c in range(seq_length):
...         pred, hidden, cell = model(seq_batch[:, c], hidden, cell)
...         loss += loss_fn(pred, target_batch[:, c])
...     loss.backward()
...     optimizer.step()
...     loss = loss.item()/seq_length
...     if epoch % 500 == 0:
...         print(f'Эпоха {epoch} потери: {loss:.4f}')
```

Эпоха 0 потери: 1.9689

Эпоха 500 потери: 1.4064

Эпоха 1000 потери: 1.3155

Эпоха 1500 потери: 1.2414

Эпоха 2000 потери: 1.1697

Эпоха 2500 потери: 1.1840

Эпоха 3000 потери: 1.1469

Эпоха 3500 потери: 1.1633

```
Эпоха 4000 потери: 1.1788
Эпоха 4500 потери: 1.0828
Эпоха 5000 потери: 1.1164
Эпоха 5500 потери: 1.0821
Эпоха 6000 потери: 1.0764
Эпоха 6500 потери: 1.0561
Эпоха 7000 потери: 1.0631
Эпоха 7500 потери: 0.9904
Эпоха 8000 потери: 1.0053
Эпоха 8500 потери: 1.0290
Эпоха 9000 потери: 1.0133
Эпоха 9500 потери: 1.0047
```

Осталось оценить модель на задаче создания нового текста, предложив ей заданную короткую строку. В следующем разделе мы определим функцию для оценки обученной модели.

Этап оценки: создание новых отрывков текста

Модель RNN, которую мы обучили в предыдущем разделе, возвращает логиты размёром 80 для каждого уникального символа. Эти логиты могут быть легко преобразованы с помощью функции softmax в вероятности того, что конкретный символ будет встречен в качестве следующего символа. Чтобы предсказать следующий символ в последовательности, мы можем просто выбрать элемент с максимальным значением логита, что эквивалентно выбору символа с наибольшей вероятностью. Однако вместо того, чтобы всегда выбирать символ с наибольшей вероятностью, мы внесем в выбор случайную составляющую, — в противном случае модель всегда будет создавать один и тот же текст. PyTorch уже предоставляет класс torch.distributions.categorical.Categorical, который мы можем использовать для получения случайных выборок из категориального распределения. Чтобы увидеть, как он работает, сгенерируем несколько случайных выборок из трех категорий [0, 1, 2] с входными логитами [1, 1, 1]:

```
>>> from torch.distributions.categorical import Categorical
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 1.0]])
>>> print('Вероятности:', ...
... nn.functional.softmax(logits, dim=1).numpy()[0])
Вероятности: [0.33333334 0.33333334 0.33333334]
>>> m = Categorical(logits=logits)
>>> samples = m.sample((10,))
>>> print(samples.numpy())
[[0]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]
 [2]
 [1]
 [1]]
```

Как видите, при данных логитах категории имеют одинаковые вероятности (т. е. эти категории равновероятные). Поэтому, если мы используем большой размер выборки ($\text{num_samples} \rightarrow \infty$), то ожидаем, что количество вхождений каждой категории достигнет $\approx \frac{1}{3}$ размера выборки. Если же изменить логиты на [1, 1, 3], то можно ожидать получения большего количества вхождений для категории 2 (когда из этого распределения взято очень большое количество примеров):

```
>>> torch.manual_seed(1)
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>>> print('Вероятности:', nn.functional.softmax(logits, dim=1).numpy()[0])
Вероятности: [0.10650698 0.10650698 0.78698605]
>>> m = Categorical(logits=logits)
>>> samples = m.sample((10,))
>>> print(samples.numpy())
[[0]
 [2]
 [2]
 [1]
 [2]
 [1]
 [2]
 [2]
 [2]
 [2]]
```

Таким образом, используя класс `Categorical`, мы можем генерировать примеры на основе логитов, вычисленных нашей моделью.

Определим функцию `sample()`, которая получает короткую начальную строку `starting_str` и генерирует новую строку `generated_str`, изначально совпадающую со входной строкой. Стока `starting_str` кодируется последовательностью целых чисел `encoded_input`. Затем `encoded_input` передается модели RNN по одному символу за раз для обновления скрытых состояний. Последний символ `encoded_input` передается модели для создания нового символа. Напомним, что выходные данные модели RNN представляют собой логиты (здесь вектор размером 80, что является общим количеством возможных символов), по которым модель выбирает следующий символ после наблюдения за входной последовательностью.

Здесь мы используем только вывод `logits` (т. е. $\mathbf{o}^{(T)}$), который передается классу `Categorical` для создания новой выборки. Эта новая выборка преобразуется в символ, который затем добавляется в конец сгенерированной строки `generated_text`, увеличивая ее длину на 1. Затем процесс повторяется до тех пор, пока длина сгенерированной строки не достигнет желаемого значения. Процесс использования сгенерированной последовательности в качестве входных данных для создания новых элементов называется *авторегрессией*.

Код функции `sample()` выглядит следующим образом:

```
>>> def sample(model, starting_str,
...             len_generated_text=500,
...             scale_factor=1.0):
```

```

...
    encoded_input = torch.tensor(
        [char2int[s] for s in starting_str]
    )
    encoded_input = torch.reshape(
        encoded_input, (1, -1)
    )
    generated_str = starting_str
...
    model.eval()
    hidden, cell = model.init_hidden(1)
    for c in range(len(starting_str)-1):
        _, hidden, cell = model(
            encoded_input[:, c].view(1), hidden, cell
        )
...
    last_char = encoded_input[:, -1]
    for i in range(len_generated_text):
        logits, hidden, cell = model(
            last_char.view(1), hidden, cell
        )
        logits = torch.squeeze(logits, 0)
        scaled_logits = logits * scale_factor
        m = Categorical(logits=scaled_logits)
        last_char = m.sample()
        generated_str += str(char_array[last_char])
...
    return generated_str

```

Давайте попробуем сгенерировать новый текст⁷:

```

>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island'))
The island had been made
and oyloire with think, captain?" asked Neb; "we do."

```

It was found, they full to time to remove. About this neur prowers, perhaps
ended? It is might be
rather rose?"

"Forward!" exclaimed Pencroft, "they were it? It seems to me?"

"The dog Top--"

"What can have been struggling sventy."

Pencroft calling, themselves in time to try them what proves that the sailor
and Neb bounded this tenarvan's feelings, and then
still hid head a grand furiously watched to the dorner nor his only

⁷ Переводить на русский язык этот текст смысла не имеет, потому что он: а) генерируется моделью по английскому обучающему набору; б) весьма корявый и не совсем осмысленный; в) показан просто в качестве примера работы функции генератора. — Прим. пер.

Как видите, модель выдает в основном правильные слова, но в ряде случаев предложения содержат смысл лишь частично. Вы можете дополнительно настроить параметры обучения, такие как длина входных последовательностей для обучения, а также архитектуру модели.

Кроме того, чтобы контролировать предсказуемость сгенерированных выборок (т. е. генерировать текст в соответствии с изученными шаблонами из обучающего текста вместо добавления большей случайности), логиты, вычисленные моделью RNN, можно масштабировать перед передачей в `Categorical` для выборки. Коэффициент масштабирования α можно интерпретировать как аналог температуры в физике. Более высокие температуры приводят к большей энтропии или случайности по сравнению с более предсказуемым поведением при более низких температурах. При масштабировании логитов с $\alpha < 1$, вероятности, вычисленные функцией `softmax`, становятся более равномерными, как показано в следующем коде:

```
>>> logits = torch.tensor([[1.0, 1.0, 3.0]])
>>> print('Вероятности до масштабирования: ',
...       nn.functional.softmax(logits, dim=1).numpy()[0])
>>> print('Вероятности после масштабирования 0.5:',
...       nn.functional.softmax(0.5*logits, dim=1).numpy()[0])
>>> print('Вероятности после масштабирования 0.1:',
...       nn.functional.softmax(0.1*logits, dim=1).numpy()[0])
Вероятности до масштабирования: [0.10650698 0.10650698 0.78698604]
Вероятности после масштабирования 0.5: [0.21194156 0.21194156 0.57611688]
Вероятности после масштабирования 0.1: [0.31042377 0.31042377 0.37915245]
```

Как можно видеть, масштабирование логитов при $\alpha = 0.1$ приводит к почти однородным вероятностям [0.31, 0.31, 0.38]. Теперь мы можем сравнить сгенерированный текст с $\alpha = 2.0$ и $\alpha = 0.5$, как показано далее:

◆ $\alpha = 2.0 \rightarrow$ более предсказуемо:

```
>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=2.0))
The island is one of the colony?" asked the sailor, "there is not to be
able to come to the shores of the Pacific."
"Yes," replied the engineer, "and if it is not the position of the forest, and the marshy
way have been said, the dog was not first on the shore, and
found themselves to the corral.
The settlers had the sailor was still from the surface of the sea, they were not received
for the sea. The shore was to be able to inspect the
windows of Granite House.
The sailor turned the sailor was the hor
```

◆ $\alpha = 0.5 \rightarrow$ более случайно:

```
>>> torch.manual_seed(1)
>>> print(sample(model, starting_str='The island',
...              scale_factor=0.5))
The island
deep incomele.
```

Manyl's', House, won's calcon-sglanderlessly," everful ineriorouins.,
pyra" into
truth. Sometinivabes, iskumar gave-zen."

Blesched but what cotch quadrap which little cedass
fell oprely
by-andonem. Peditivall--"i dove Gurgeon. What resolt-eartnated to him
ran trail.

Withinhe)tiny turns returned, after owner plan bushelsion lairs; they
were
know? Whalerin branch I
pires, Dougg!-iteun," returnwe aid masses atong thoughts! Dak,
Hem-arches yone, Veay wantzer? Woblding,
Herbert, омер

Судя по результатам, масштабирование логитов с коэффициентом $\alpha = 0.5$ (увеличение температуры) приводит к генерации более случайного и хаотичного текста. Поэтому приходится искать компромисс между новизной сгенерированного текста и его правильностью.

В этом разделе мы работали с генерацией текста на уровне символов, которая представляет собой задачу моделирования последовательностей (seq2seq). Хотя сам по себе рассмотренный здесь пример может быть не очень полезен, легко придумать несколько полезных приложений для этих типов моделей — например, аналогичная модель RNN может быть обучена как чат-бот, чтобы помогать пользователям получать ответы на простые вопросы.

15.4. Заключение

В этой главе вы сначала узнали о характерных свойствах последовательностей, отличающих их от других типов данных, таких как структурированные данные или изображения. Затем мы рассмотрели основы RNN для моделирования последовательностей. Вы также познакомились с тем, как работает базовая модель RNN, и обсудили ее ограничения в отношении отслеживания долгосрочных зависимостей в последовательных данных. Далее мы изучили ячейки LSTM, которые используют механизм управляемых рекуррентных блоков для уменьшения эффектов градиентного взрыва и исчезающего градиента, присущих базовым моделям RNN.

После знакомства с основными понятиями, лежащими в основе RNN, мы реализовали несколько моделей RNN с различными рекуррентными слоями с помощью PyTorch. В частности, мы разработали модель RNN для анализа эмоциональной тональности текста, а также модель RNN для генерации текста.

В следующей главе вы увидите, как можно дополнить RNN механизмом внимания, который помогает ей моделировать долгосрочные зависимости в задачах перевода. Затем вы познакомитесь с новой архитектурой глубокого обучения под названием Transformer, благодаря которой в последнее время обработка естественного языка поднялась на принципиально новый уровень.

16

Трансформеры: улучшение обработки естественного языка с помощью механизмов внимания

В предыдущей главе вы узнали о рекуррентных нейронных сетях (RNN) и их применении для обработки естественного языка (Natural Language Processing, NLP) в рамках проекта анализа эмоциональной окраски текста. Однако недавно появилась новая архитектура, которая, как было показано, ощутимо превосходит основанные на RNN модели преобразования одних последовательностей в другие (seq2seq) в некоторых задачах NLP. Это так называемая архитектура Transformer. Для краткости дальше мы будем называть различные модели на основе этой архитектуры просто *трансформерами*.

Трансформеры произвели революцию в обработке естественного языка и лидируют во многих областях, начиная с автоматического перевода с одного естественного языка на другой¹ и моделирования фундаментальных свойств белковых последовательностей² до создания ИИ, помогающего людям писать код программ³.

В этой главе мы рассмотрим основные механизмы *внимания* (attention) и *самовнимания* (self-attention) и то, как они задействуются в оригинальной архитектуре Transformer. Затем вы познакомитесь с некоторыми из наиболее значимых моделей NLP, основанных на этой архитектуре, и узнаете, как использовать крупномасштабную языковую модель, такую как BERT, с библиотекой PyTorch.

Таким образом, в этой главе будут раскрыты следующие темы:

- ◆ улучшение RNN с помощью механизма внимания;
- ◆ знакомство с автономным механизмом самовнимания;
- ◆ принцип работы оригинальной архитектуры Transformer;
- ◆ сравнение больших языковых моделей-трансформеров;
- ◆ тонкая настройка BERT для классификации эмоциональной окраски текста.

¹ См. <https://ai.googleblog.com/2020/06/recent-advances-in-google-translate.html>.

² См. <https://www.pnas.org/content/118/15/e2016239118.short>.

³ См. <https://github.blog/2021-06-29-introducing-github-copilotai-pair-programmer>.

16.1. Добавляем к RNN механизм внимания

В этом разделе мы обсудим причины, вызвавшие разработку механизма внимания, помогающего прогностическим моделям фокусироваться на одних частях входной последовательности больше, чем на других, и его первоначальном применении в составе RNN. В первую очередь обратите внимание на историческую перспективу, объясняющую, почему был разработан механизм внимания. Если отдельные математические выкладки покажутся вам сложными, можете смело пропустить их, т. к. они не столь важны для понимания материала следующего раздела, объясняющего механизм самовнимания трансформеров, которому в основном и посвящена эта глава.

16.1.1. Как внимание помогает RNN извлекать информацию?

Чтобы понять назначение механизма внимания, рассмотрим традиционную модель RNN для задачи перевода естественного языка, которая анализирует *всю* входную последовательность (например, одно или несколько предложений) перед выполнением перевода (рис. 16.1).

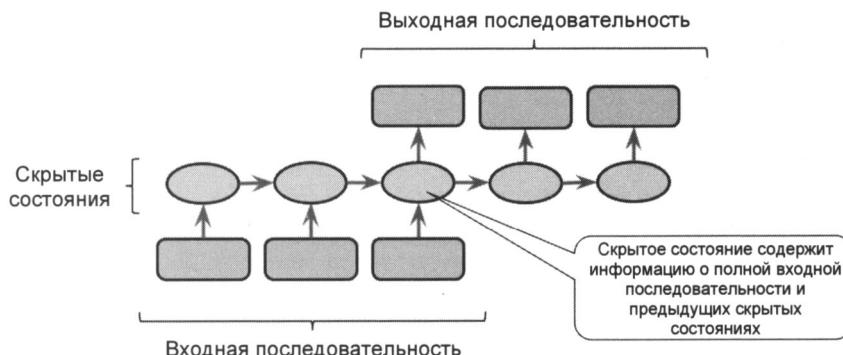


Рис. 16.1. Традиционная архитектура кодера-декодера RNN для задачи моделирования «последовательность в последовательность» (seq2seq)

Почему RNN анализирует входное предложение целиком, прежде чем выдать первый вывод? Дело в том, что перевод предложения слово за словом может привести к грамматическим ошибкам, как показано на рис. 16.2.

Входное предложение на немецком: Kannst du mir helfen diesen Satz zu uebersetzen ?
 Пословный перевод (неправильный): Can you me help this sentence to translate ?

Входное предложение на немецком: Kannst du mir helfen diesen Satz zu uebersetzen ?
 Правильный перевод на английский: Can you help me to translate this sentence ?

Рис. 16.2. Пословный перевод предложения может привести к грамматическим ошибкам

Однако одним из ограничений подхода seq2seq является то, что RNN пытается запомнить всю входную последовательность с помощью единственного скрытого блока перед ее преобразованием. Однако сжатие всей информации в один скрытый блок может привести к потере информации, особенно в случае длинных последовательностей. Было бы полезно иметь доступ ко всей входной последовательности на каждом временном шаге, как это делают люди во время перевода предложений.

В отличие от обычной RNN, механизм внимания позволяет оснащенной им RNN получать доступ ко всем входным элементам на каждом временном шаге. Однако иметь доступ ко всем элементам входной последовательности на каждом временном шаге может быть сложно. Чтобы помочь RNN сосредоточиться на наиболее важных элементах входной последовательности, механизм внимания присваивает каждому входному элементу разные веса внимания. Эти веса определяют, насколько важным или релевантным является определенный элемент входной последовательности на том или ином временном шаге. Например, слова «*mir*», «*helfen*» и *zu*» в варианте перевода, показанном на рис. 16.2, являются более подходящими для получения выходного слова «*help*», чем слова «*kannst*», «*du*» и «*Satz*».

В следующем разделе мы рассмотрим архитектуру RNN, оснащенную механизмом внимания, который помогает обрабатывать длинные последовательности в задачах языкового перевода.

16.1.2. Оригинальный механизм внимания для RNN

Предметом нашего интереса здесь является обобщенный механизм внимания, изначально разработанный для языкового перевода⁴.

Если на вход поступает последовательность $x = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$, механизм внимания присваивает вес каждому элементу $x^{(i)}$ (или, точнее, его скрытому представлению) и помогает модели определить, на какой части ввода она должна сосредоточиться. Предположим, например, что наш ввод — это предложение, и слово с большим весом в большей степени способствует нашему пониманию всего предложения. RNN с механизмом внимания, показанная на рис. 16.3, иллюстрирует общую идею генерации второго выходного слова. Она состоит из двух моделей RNN, назначение которых мы объясним далее.

16.1.3. Обработка входных данных с использованием двунаправленной RNN

Показанная на рис. 16.3 первая RNN (RNN #1) — это двунаправленная RNN, которая генерирует векторы контекста c_i . Вектор контекста можно считать расширенной версией входного вектора $x^{(i)}$. Другими словами, входной вектор c , также включает в себя информацию от всех других входных элементов через механизм внимания. Как показано на рис. 16.3, RNN #2 затем использует этот подготовленный RNN #1 вектор контекста для генерации выходных данных. В оставшейся части этого подраздела мы рассмотрим, как работает RNN #1, а к RNN #2 вернемся в следующем подразделе.

⁴ Впервые представлен в статье «Neural Machine Translation by Jointly Learning to Align and Translate» by Bahdanau D., Cho K., and Bengio Y., 2014, <https://arxiv.org/abs/1409.0473>.

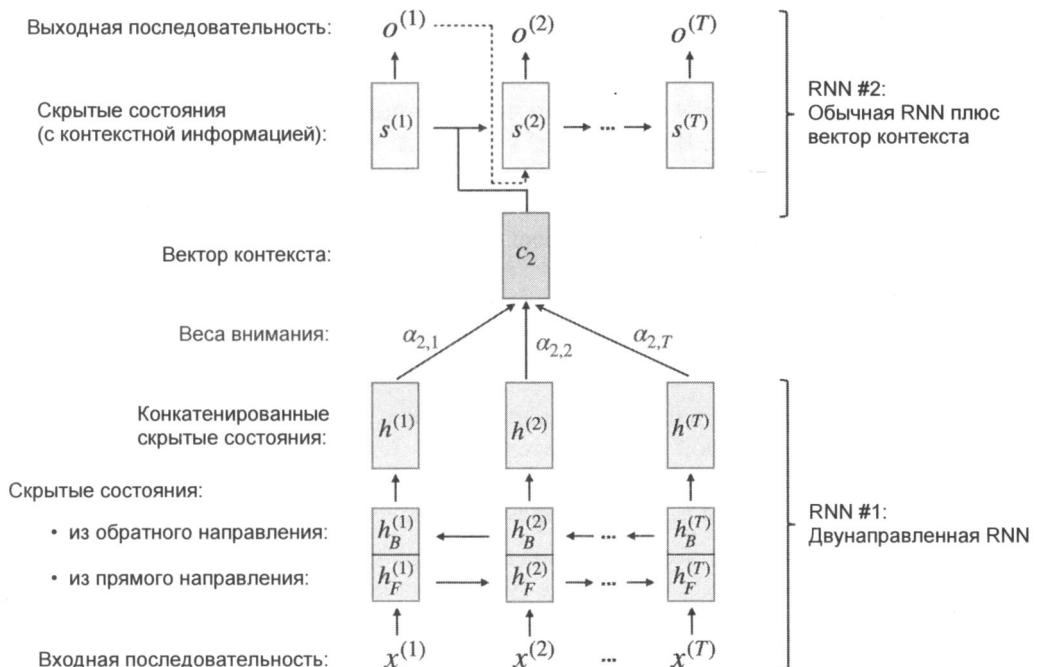


Рис. 16.3. RNN с механизмом внимания

Двунаправленная RNN #1 обрабатывает входную последовательность x как в обычном прямом направлении ($1 \dots T$), так и в обратном ($T \dots 1$). Разбор последовательности в обратном направлении имеет тот же эффект, что и обращение исходной входной последовательности, — по аналогии с чтением предложения в обратном порядке. Это действие объясняется необходимостью получения дополнительной информации, поскольку текущие входные данные могут зависеть от элементов последовательности, которые шли в предложении либо до, либо после, либо и от тех и от других.

Следовательно, при двукратном чтении входной последовательности (т. е. вперед и назад) мы получаем два скрытых состояния для каждого элемента входной последовательности. Например, для второго элемента входной последовательности $x^{(2)}$ мы получаем скрытое состояние $h_F^{(2)}$ из прямого прохода и скрытое состояние $h_B^{(2)}$ из обратного. Затем эти два скрытых состояния объединяются, чтобы сформировать конкатенированное скрытое состояние. Так, если $h_F^{(2)}$, и $h_B^{(2)}$ являются 128-мерными векторами, конкатенированное скрытое состояние $h^{(2)}$ будет состоять из 256 элементов. Мы можем рассматривать это связанное скрытое состояние как «аннотацию» исходного слова, поскольку оно содержит информацию j -го слова в обоих направлениях.

В следующем подразделе вы увидите, как эти конкатенированные скрытые состояния обрабатываются и используются второй RNN для генерации выходных данных.

16.1.4. Генерация выходных данных из векторов контекста

Показанную на рис. 16.3 RNN #2 мы можем рассматривать как основную RNN, генерирующую выходные данные. В дополнение к скрытым состояниям она получает на вход так называемые *векторы контекста*. Вектор контекста c_i — это взвешенная версия конкатенированных скрытых состояний $h^{(1)} \dots h^{(T)}$, которую мы получили из RNN #1, рассмотренной в предыдущем подразделе. Мы можем вычислить вектор контекста i -го входа как взвешенную сумму:

$$c_i = \sum_{j=1}^T \alpha_{ij} h^{(j)}.$$

Здесь α_{ij} представляет веса внимания для входной последовательности $j = 1 \dots T$ в контексте i -го элемента входной последовательности. Заметьте, что каждый i -й элемент входной последовательности имеет уникальный набор весов внимания. Мы обсудим вычисление весов внимания α_{ij} в следующем подразделе.

А в оставшейся части этого подраздела мы рассмотрим, как векторы контекста задействуются во второй RNN (RNN #2). Так же как и обычная RNN, она использует скрытые состояния. Учитывая скрытый слой между упомянутой в предыдущем подразделе «аннотацией» и окончательным выводом, обозначим скрытое состояние в момент времени i как $s^{(i)}$. Теперь RNN #2 получает наш вектор контекста c_i на каждом временном шаге i в качестве входных данных.

На рис. 16.3 мы видим, что скрытое состояние $s^{(i)}$ зависит от предыдущего скрытого состояния $s^{(i-1)}$, предыдущего целевого слова $y^{(i-1)}$ и вектора контекста c_i , которые служат для генерации предсказанного вывода $o^{(i)}$ для целевого слова $y^{(i)}$ в момент i . Обратите внимание, что вектор последовательности u соответствует вектору последовательности, представляющему правильный перевод входной последовательности x , доступный во время обучения. Во время обучения истинная метка (слово) $y^{(i)}$ подается в следующее состояние $s^{(i+1)}$. Но поскольку эта истинная информация о метке недоступна для прогнозирования (вывода), вместо этого мы передаем прогнозируемый вывод $o^{(i)}$, как и показано на рис. 16.3.

Подведем краткий итог нашему обсуждению. RNN, основанная на внимании, состоит из двух частей: RNN #1 подготавливает векторы контекста из элементов входной последовательности, а RNN #2 получает векторы контекста в качестве входных данных. Векторы контекста вычисляются с помощью взвешенной суммы входных данных, где весовые коэффициенты — это веса внимания α_{ij} . В следующем подразделе обсуждается вычисление этих весов внимания.

16.1.5. Вычисление весов внимания

В завершение давайте рассмотрим последнюю недостающую часть нашей головоломки — веса внимания. Поскольку эти веса попарно соединяют входы (аннотации) и выходы (контексты), каждый вес внимания α_{ij} имеет два индекса: j — обозначает позицию индекса ввода, а i — соответствует позиции индекса вывода. Вес внимания α_{ij} — это нормализованная версия оценки выравнивания e_{ij} , которая показывает, насколько хорошо входные данные вокруг позиции j совпадают с выходными данными в позиции i .

Если конкретизировать сказанное, то вес внимания вычисляется путем нормализации оценок выравнивания:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}.$$

Обратите внимание, что это уравнение похоже на функцию `softmax`, которую мы обсуждали в разд. 12.5.2. Следовательно, веса внимания $\alpha_{i1}, \dots, \alpha_{iT}$ в сумме дают 1.

Теперь, подводя итог, мы можем разделить модель RNN, основанную на внимании, на три части: первая часть вычисляет двунаправленные аннотации ввода, вторая — состоит из рекуррентного блока, который очень похож на базовую модель RNN, за исключением того, что он использует векторы контекста вместо исходного ввода, а третья — отвечает за вычисление весов внимания и векторов контекста, которые описывают взаимосвязь между каждой парой входных и выходных элементов.

В архитектуре Transformer также используется механизм внимания, но, в отличие от RNN, основанной на внимании, она полагается исключительно на механизм *самовнимания* (self-attention) и не использует рекуррентный процесс, присутствующий в RNN. Другими словами, модель-трансформер обрабатывает всю входную последовательность одновременно, а не по одному элементу за раз. В следующем разделе представлена базовая форма механизма самовнимания, а после нее мы подробно обсудим архитектуру Transformer.

16.2. Знакомство с механизмом самовнимания

В предыдущем разделе было показано, что механизмы внимания могут помочь RNN запоминать контекст при работе с длинными последовательностями. Как вы увидите далее, можно построить архитектуру, полностью основанную на внимании, без рекуррентных частей RNN. Такая основанная на внимании архитектура известна под названием Transformer, и мы обсудим ее более подробно позже.

Сначала принцип работы трансформеров может показаться немного сложным. Поэтому, прежде чем перейти к обсуждению трансформеров, представленному в следующем разделе, мы пока что более подробно рассмотрим здесь механизм самовнимания. На самом деле, как вы сейчас увидите, самовнимание — всего лишь разновидность механизма внимания, который мы обсуждали в предыдущем разделе. Механизм внимания можно рассматривать как операцию, которая соединяет два разных модуля — т. е. кодировщик и декодер RNN. Как будет показано далее, самовнимание сосредоточивается лишь на входных данных и улавливает только зависимости между входными элементами без связывания двух модулей.

В следующем подразделе мы рассмотрим базовую форму самовнимания без каких-либо обучаемых параметров, очень похожую на этап предварительной обработки входных данных. Затем, в следующем далее подразделе мы представим распространенную версию самовнимания, которая используется в архитектуре Transformer и содержит обучаемые параметры.

16.2.1. Начнем с базовой формы самовнимания

Чтобы подойти к понятию самовнимания, давайте предположим, что у нас есть входная последовательность длины T : $x^{(1)}, \dots, x^{(T)}$, а также выходная последовательность $z^{(1)}, z^{(2)}, \dots, z^{(T)}$. Чтобы избежать путаницы, мы будем помечать символами o окончательный вывод трансформера, а символами z — выводы слоя самовнимания, потому что это промежуточные шаги в модели.

Каждый i -й элемент в последовательностях $x^{(i)}$ и $z^{(i)}$ является вектором размера d (т. е. $x^{(i)} \in R^d$), представляющим информацию о признаках для ввода в позиции i , что аналогично RNN. Причем для задачи seq2seq целью самовнимания является моделирование зависимостей текущего элемента ввода от всех других элементов ввода. Для этого механизмы самовнимания состоят из трех этапов. Сначала мы получаем веса важности на основе сходства между текущим элементом и всеми другими элементами в последовательности. Затем нормализуем веса, что обычно предполагает использование уже знакомой нам функции softmax. И в завершение используем эти веса в сочетании с соответствующими элементами последовательности для вычисления значения внимания.

С формальной точки зрения выход самовнимания $z^{(i)}$ представляет собой взвешенную сумму всех T входных последовательностей $x^{(j)}$ (где $j = 1 \dots T$). Например, для i -го входного элемента соответствующее выходное значение вычисляется так:

$$z^{(i)} = \sum_{j=1}^T \alpha_{ij} x^{(j)}.$$

Следовательно, мы можем рассматривать $z^{(i)}$ как контекстно-зависимый вектор встраивания во входном векторе $x^{(i)}$, включающем в себя все другие элементы входной последовательности, взвешенные по их соответствующим весам внимания. Здесь веса внимания α_{ij} вычисляются на основе подобия между текущим входным элементом $x^{(i)}$ и всеми другими элементами входной последовательности $x^{(1)} \dots x^{(T)}$. То есть это сходство вычисляется в два этапа, которые описаны далее.

Сначала мы вычисляем скалярное произведение между текущим входным элементом $x^{(i)}$ и другим элементом входной последовательности $x^{(j)}$:

$$\omega_{ij} = x^{(i)\top} x^{(j)}.$$

Прежде чем нормализовать значения ω_{ij} для получения весов внимания a_{ij} , проиллюстрируем практическое вычисление значения ω_{ij} на примере кода. Для этого предположим, что у нас есть входное предложение «can you help me to translate this sentence» («могли бы вы помочь мне перевести это предложение»), которое уже преобразовано в целочисленное представление через словарь, как показано в главе 15:

```
>>> import torch
>>> sentence = torch.tensor(
>>>     [0, # can
>>>     7, # you
>>>     1, # help
>>>     2, # me
>>>     5, # to
>>>     6, # translate
```

```
>>>      4, # this
>>>      3] # sentence
>>> )
>>> sentence
tensor([0, 7, 1, 2, 5, 6, 4, 3])
```

Давайте также условимся, что мы уже закодировали это предложение в векторное представление вещественных чисел через слой встраивания. Здесь наш размер встраивания равен 16, и мы предполагаем, что размер словаря равен 10. Следующий код создаст встраивания наших восьми слов:

```
>>> torch.manual_seed(123)
>>> embed = torch.nn.Embedding(10, 16)
>>> embedded_sentence = embed(sentence).detach()
>>> embedded_sentence.shape
torch.Size([8, 16])
```

Теперь мы можем вычислить ω_{ij} как скалярное произведение i -го и j -го встраиваний слов. Для всех значений ω_{ij} это можно сделать следующим образом:

```
>>> omega = torch.empty(8, 8)
>>> for i, x_i in enumerate(embedded_sentence):
>>> for j, x_j in enumerate(embedded_sentence):
>>> omega[i, j] = torch.dot(x_i, x_j)
```

Хотя приведенный код легко читать и понимать, циклы `for` могут быть очень неэффективными, поэтому лучше вместо циклов использовать матричное умножение:

```
>>> omega_mat = embedded_sentence.matmul(embedded_sentence.T)
```

Мы можем воспользоваться функцией `torch.allclose`, чтобы проверить, дает ли это умножение матриц ожидаемые результаты. Если два тензора содержат одинаковые значения, `torch.allclose` возвращает `True`:

```
>>> torch.allclose(omega_mat, omega)
True
```

Мы научились вычислять основанные на сходстве веса для i -го входа и всех входов в последовательности (от $x^{(1)}$ до $x^{(T)}$) — т. е. «сырые» веса (от ω_{i1} до ω_{iT}). А веса внимания α_{ij} мы можем получить, нормализовав значения ω_{ij} с помощью знакомой функции `softmax`:

$$\alpha_{ij} = \frac{\exp(\omega_{ij})}{\sum_{j=1}^T \exp(\omega_{ij})} = \text{softmax}\left(\left[\omega_{ij}\right]_{j=1 \dots T}\right).$$

Заметьте, что знаменатель включает сумму по всем входным элементам ($1 \dots T$). Следовательно, из-за применения этой функции `softmax` веса после нормализации будут в сумме равны 1, т. е.

$$\sum_{j=1}^T \alpha_{ij} = 1.$$

Вычислить веса внимания мы можем, используя функцию `softmax` PyTorch:

```
>>> import torch.nn.functional as F
>>> attention_weights = F.softmax(omega, dim=1)
```

```
>>> attention_weights.shape
torch.Size([8, 8])
```

Здесь `attention_weights` — это матрица 8×8 , где каждый элемент представляет вес внимания α_{ij} . Например, если мы обрабатываем i -е входное слово, i -я строка этой матрицы содержит соответствующие веса внимания для всех слов в предложении. Эти веса внимания показывают, насколько релевантно каждое слово i -му слову. Следовательно, столбцы в этой матрице внимания должны в сумме равняться 1, что мы можем подтвердить с помощью следующего кода:

```
>>> attention_weights.sum(dim=1)
tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
```

Теперь, когда мы рассмотрели вычисление весовых коэффициентов внимания, кратко резюмируем три основных шага, лежащих в основе операции самовнимания:

1. Для заданного входного элемента $x^{(i)}$ и каждого j -го элемента в наборе $\{1, \dots, T\}$ вычисляем скалярное произведение $x^{(i)\top} x^{(j)}$.
2. Получаем вес внимания α_{ij} путем нормализации скалярных произведений с помощью функции `softmax`.
3. Вычисляем вывод $z^{(i)}$ как взвешенную сумму по всей входной последовательности:

$$z^{(i)} = \sum_{j=1}^T \alpha_{ij} x^{(j)}.$$

Эти шаги дополнительно показаны на рис. 16.4.

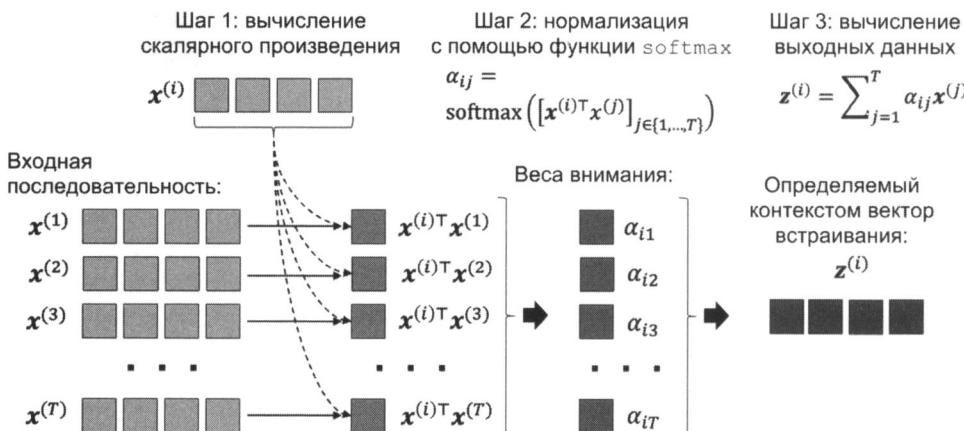


Рис. 16.4. Работа базового механизма самовнимания

В завершение рассмотрим пример кода для вычисления векторов контекста $z^{(i)}$ как взвешенной по вниманию суммы входов (шаг 3 на рис. 16.4). В частности, предположим, что мы вычисляем вектор контекста для второго входного слова — т. е. $z^{(2)}$:

```
>>> x_2 = embedded_sentence[1, :]
>>> context_vec_2 = torch.zeros(x_2.shape)
>>> for j in range(8):
...     x_j = embedded_sentence[j, :]
...     context_vec_2 += attention_weights[1, j] * x_j
```

```
>>> context_vec_2
tensor([-9.3975e-01, -4.6856e-01, 1.0311e+00, -2.8192e-01, 4.9373e-01,
-1.2896e-02, -2.7327e-01, -7.6358e-01, 1.3958e+00, -9.9543e-01,
-7.1288e-04, 1.2449e+00, -7.8077e-02, 1.2765e+00, -1.4589e+00,
-2.1601e+00])
```

Опять же, мы можем сделать это более эффективно, используя матричное умножение. С помощью следующего кода мы вычисляем векторы контекста для всех восьми входных слов:

```
>>> context_vectors = torch.matmul(
... attention_weights, embedded_sentence)
```

Подобно встраиванию входных слов, хранящимся в `embedded_sentence`, матрица `context_vectors` имеет размерность 8×8 . Вторая строка в этой матрице содержит вектор контекста для второго входного слова, и мы можем снова проверить реализацию, используя `torch.allclose()`:

```
>>> torch.allclose(context_vec_2, context_vectors[1])
True
```

Как видим, использование цикла `for` и матричных вычислений для второго контекстного вектора дало одинаковые результаты.

В этом подразделе была реализована базовая форма самовнимания, а в следующем мы изменим эту реализацию, используя обучаемые матрицы параметров, которые можно оптимизировать во время обучения нейронной сети.

16.2.2. Параметризация механизма самовнимания: взвешенное скалярное произведение

Теперь, когда вам стала ясна основная идея самовнимания, мы рассмотрим более продвинутый механизм самовнимания, называемый *взвешенным скалярным вниманием* (scaled dot-product attention), который используется в архитектуре Transformer. Как вы помните, в предыдущем подразделе мы не использовали какие-либо обучаемые параметры при вычислении выходных данных. Другими словами, модель трансформера, использующая базовый механизм самовнимания, весьма ограничена в возможностях обновлять или изменять значения внимания во время оптимизации под заданную последовательность. Чтобы сделать механизм самовнимания более гибким и поддающимся оптимизации, мы введем три дополнительные весовые матрицы, которые можно использовать в качестве параметров модели во время обучения: U_q , U_k и U_v . Они задействуются для проецирования входных данных в элементы последовательности запросов, ключей и значений следующим образом:

- ◆ последовательность запросов: $q^{(i)} = U_q x^{(i)}$ где $i \in [1, T]$;
- ◆ последовательность ключей: $k^{(i)} = U_k x^{(i)}$ где $i \in [1, T]$;
- ◆ последовательность значений: $v^{(i)} = U_v x^{(i)}$ где $i \in [1, T]$.

На рис. 16.5 показано, как с помощью этих компонентов производится вычисление контекстно-зависимого вектора встраивания, соответствующего второму входному элементу.

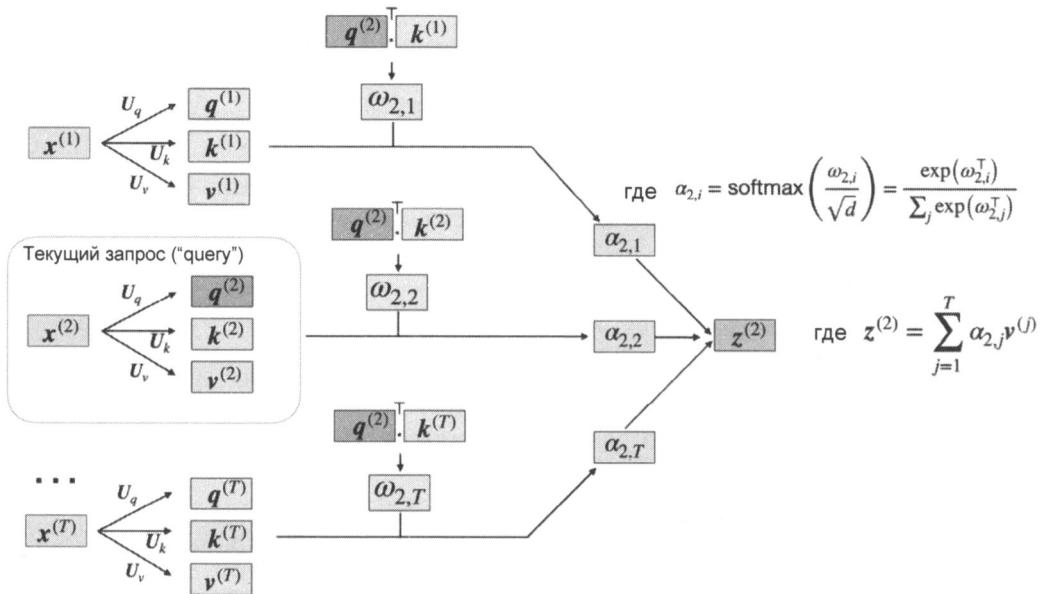


Рис. 16.5. Вычисление контекстно-зависимого вектора встраивания второго элемента последовательности



Термины «запрос», «ключ» и «значение»

Термины «запрос», «ключ» и «значение», использованные в упомянутой ранее оригинальной статье о трансформерах, позаимствованы из систем поиска информации и баз данных. Например, если мы вводим запрос, он сопоставляется с ключами, для которых извлекаются определенные значения.

Здесь и $q^{(i)}$, и $k^{(i)}$ являются векторами размера d_k . Следовательно, матрицы проекций U_q и U_k имеют форму $d_k \times d$, а U_v имеет форму $d_v \times d$. (Здесь d — это размерность каждого вектора слов $x^{(i)}$.) Для простоты мы можем построить эти векторы так, чтобы они имели одинаковый размер — например, $d_k = d_v = d$. Следующий код поможет вам лучше разобраться в матрицах проекций:

```
>>> torch.manual_seed(123)
>>> d = embedded_sentence.shape[1]
>>> U_query = torch.rand(d, d)
>>> U_key = torch.rand(d, d)
>>> U_value = torch.rand(d, d)
```

Используя матрицу проекции запроса, мы можем вычислить последовательность запроса. Продолжая наш пример, рассмотрим второй элемент ввода $x^{(i)}$ в качестве запроса, как показано на рис. 16.5:

```
>>> x_2 = embedded_sentence[1]
>>> query_2 = U_query.matmul(x_2)
```

Аналогичным образом мы можем вычислить последовательности ключей $k^{(i)}$ и значений $v^{(i)}$:

```
>>> key_2 = U_key.matmul(x_2)
>>> value_2 = U_value.matmul(x_2)
```

Однако, как видно из рис. 16.5, нам также нужны последовательности ключей и значений для всех остальных входных элементов, которые мы можем вычислить следующим образом:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> values = U_value.matmul(embedded_sentence.T).T
```

В матрице ключей i -я строка соответствует последовательности ключа i -го входного элемента, и то же самое относится к матрице значений. Мы можем убедиться в этом, снова используя `torch.allclose()`, который должен вернуть `True`:

```
>>> keys = U_key.matmul(embedded_sentence.T).T
>>> torch.allclose(key_2, keys[1])
>>> values = U_value.matmul(embedded_sentence.T).T
>>> torch.allclose(value_2, values[1])
```

В предыдущем подразделе мы вычислили ненормализованные веса ω_{ij} как попарное скалярное произведение между заданным элементом входной последовательности $x^{(i)}$ и j -м элементом последовательности $x^{(j)}$. Далее, в этой параметризованной версии самовнимания мы вычисляем ω_{ij} как скалярное произведение между запросом и ключом:

$$\omega_{ij} = \mathbf{q}^{(i)\top} \mathbf{k}^{(j)}.$$

Например, следующий код вычисляет ненормализованный вес внимания ω_{23} — т. е. скалярное произведение между нашим запросом и третьим элементом входной последовательности:

```
>>> omega_23 = query_2.dot(keys[2])
>>> omega_23
tensor(14.3667)
```

Имеет смысл нормализовать эти вычисления для всех ключей, поскольку они понадобятся нам позже:

```
>>> omega_2 = query_2.matmul(keys.T)
>>> omega_2
tensor([-25.1623, 9.3602, 14.3667, 32.1482, 53.8976, 46.6626, -1.2131, -32.9391])
```

Следующим шагом в развитии самовнимания является переход от ненормализованных весов внимания к нормализованным весам α_{ij} с помощью функции `softmax`. Затем мы можем использовать $1/\sqrt{m}$ для масштабирования ω_{ij} , прежде чем нормализовать его с помощью функции `softmax`:

$$\alpha_{ij} = \text{softmax}\left(\frac{\omega_{ij}}{\sqrt{m}}\right).$$

Нормализация ω_{ij} с помощью $1/\sqrt{m}$, где обычно $m = dk$, гарантирует, что евклидова длина весовых векторов будет примерно в том же диапазоне.

Следующий код выполняет эту нормализацию для вычисления весовых коэффициентов внимания для всей входной последовательности по отношению ко второму входному элементу в качестве запроса:

```
>>> attention_weights_2 = F.softmax(omega_2 / d**0.5, dim=0)
>>> attention_weights_2
tensor([2.2317e-09, 1.2499e-05, 4.3696e-05, 3.7242e-03, 8.5596e-01, 1.4025e-01,
       8.8896e-07, 3.1936e-10])
```

Наконец, выход представляет собой средневзвешенное последовательностей значений $z^{(i)} = \sum_{j=1}^T \alpha_{ij} v^{(j)}$, которое можно получить следующим образом:

```
>>> context_vector_2 = attention_weights_2.matmul(values)
>>> context_vector_2
tensor([-1.2226, -3.4387, -4.3928, -5.2125, -1.1249, -3.3041,
       -1.4316, -3.2765, -2.5114, -2.6105, -1.5793, -2.8433, -2.4142,
       -0.3998, -1.9917, -3.3499])
```

В этом разделе мы рассмотрели механизм самовнимания с обучаемыми параметрами, который позволяет нам вычислять контекстно-зависимые векторы встраивания, включая все входные элементы, взвешенные по их соответствующим показателям внимания. Далее речь пойдет об архитектуре Transformer — нейронной сети, основанной на механизме самовнимания, о котором вы узнали в этом разделе.

16.3. Внимание — это все, что нам нужно: знакомство с архитектурой Transformer

Интересно, что оригинальная архитектура Transformer основана на механизме внимания, впервые использованном в RNN. Первоначально механизм внимания был предназначен для улучшения возможностей RNN по генерации текста при работе с длинными предложениями. Однако всего через несколько лет экспериментов с механизмами внимания для RNN исследователи обнаружили, что языковая модель, основанная на внимании, становится еще более мощной, когда удаляются рекуррентные слои. Это привело к разработке архитектуры Transformer, которая является основной темой остальных разделов этой главы.

Архитектура Transformer впервые была предложена А. Васвани и его коллегами в статье для NeurIPS (2017) «Attention Is All You Need» (<https://arxiv.org/abs/1706.03762>). Благодаря механизму самовнимания, модель-трансформер может улавливать долгосрочные зависимости между элементами входной последовательности — в контексте NLP, например, это помогает модели лучше «понимать» значение входного предложения.

Хотя архитектура изначально была разработана для языкового перевода, ее можно обобщить и для других задач — таких как синтаксический анализ естественного языка, генерация и классификация текста. Позже мы обсудим популярные языковые модели, такие как BERT и GPT, которые являются потомками исходной архитектуры Transformer. Рис. 16.6, который мы адаптировали из оригинальной статьи о трансформерах, демонстрирует ключевые компоненты архитектуры, которые мы обсудим в этом разделе.

Далее мы шаг за шагом рассмотрим базовую модель трансформера, разбив ее на два основных блока: кодировщик и декодер. Кодировщик получает исходную последовательность и кодирует встраивания с помощью многоголового модуля самовнимания.

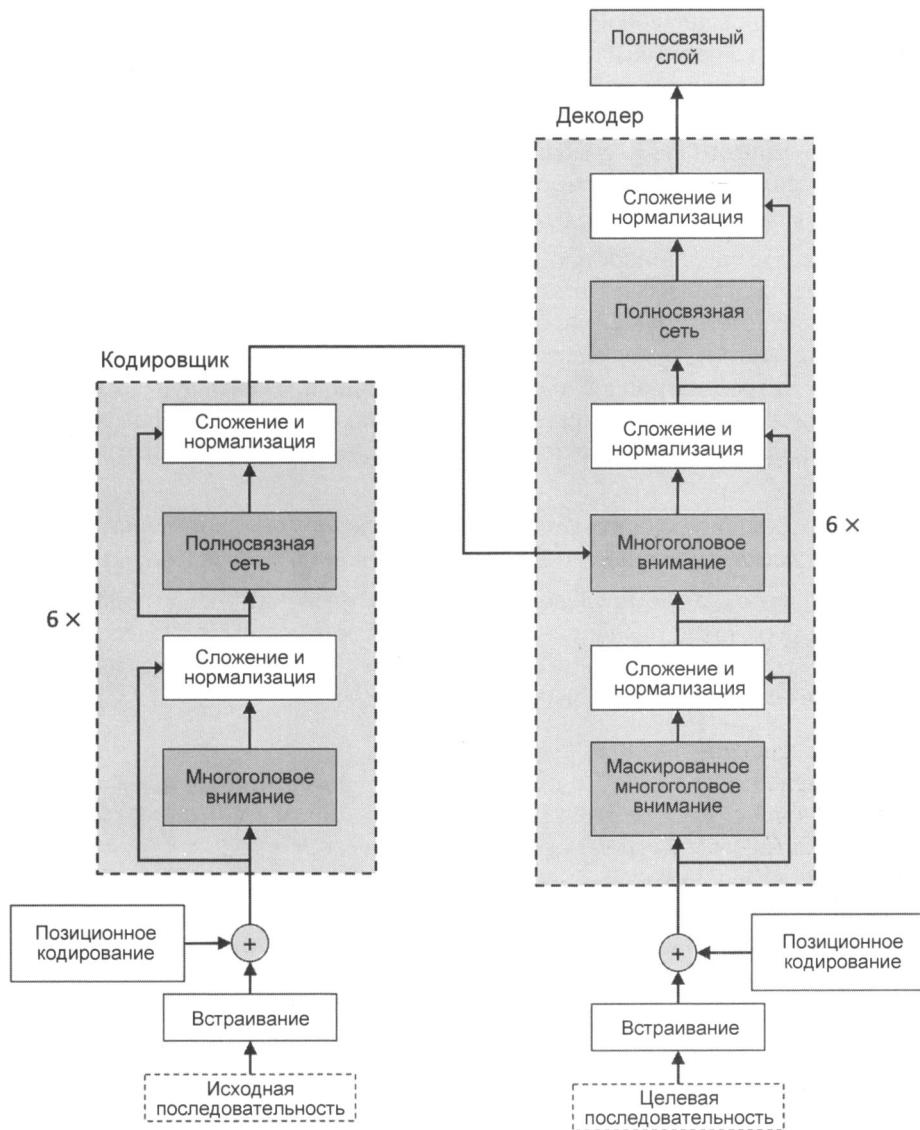


Рис. 16.6. Исходная архитектура Transformer

Декодер получает обработанный ввод и выводит результирующую последовательность (например, переведенное предложение), используя *маскированную* форму самовнимания.

16.3.1. Кодирование контекстных встраиваний с помощью многоголового внимания

Единственное назначение блока *кодировщика* заключается в получении входной последовательности $\mathbf{X} = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$ и отображении ее в непрерывное представление $\mathbf{Z} = (z^{(1)}, z^{(2)}, \dots, z^{(T)})$, поступающее на декодер.

Кодировщик представляет собой стек из шести одинаковых слоев. «Шесть» здесь не магическое число, а просто выбор гиперпараметра, сделанный в оригинальной статье о трансформерах. Вы можете выбрать другое количество слоев в зависимости от производительности модели. Внутри каждого из этих идентичных слоев есть два подслоя: один вычисляет многоголовое самовнимание, о котором мы поговорим чуть позже, а другой — полно связанный слой, с которым вы уже сталкивались в предыдущих главах.

Давайте сначала поговорим о *многоголовом самовнимании* (multi-head self-attention), которое представляет собой простую модификацию взвешенного скалярного внимания, описанного ранее в этой главе. Во взвешенном скалярном внимании мы использовали три матрицы (соответствующие запросу, значению и ключу) для преобразования входной последовательности. В контексте многоголового внимания мы можем рассматривать этот набор из трех матриц как одну *голову* внимания. Как следует из названия, при многоголовом внимании у нас имеется несколько таких голов (наборов матриц запросов, значений и ключей), подобно тому, как сверточные нейронные сети могут иметь несколько ядер.

Чтобы более подробно объяснить понятие многоголового самовнимания с h головами, разобьем его на несколько этапов.

Сначала прочитаем входную последовательность $X = (x^{(1)}, x^{(2)}, \dots, x^{(T)})$. Предположим, что каждый элемент представлен вектором встраивания длины d . Далее входные данные могут быть встроены в матрицу $T \times d$. Затем мы создаем h наборов матриц обучаемых параметров запроса, ключа и значения:

- ◆ U_{q1}, U_{k1}, U_{v1} ;
- ◆ U_{q2}, U_{k2}, U_{v2} ;
- ◆ ...
- ◆ U_{qh}, U_{kh}, U_{vh} .

Поскольку мы используем эти весовые матрицы при проецировании каждого элемента $x^{(i)}$ для требуемого совпадения размеров при умножении матриц, то и U_{qi} , и U_{ki} имеют размер $d_k \times d$, а U_{vi} имеет размер $d_v \times d$. В итоге обе результирующие последовательности: запроса и ключа — имеют длину d_k , а результирующая последовательность значения имеет длину d_v . На практике для простоты часто выбирают $d_k = d_v = m$.

Чтобы проиллюстрировать реализацию стека многоголового самовнимания в коде, сначала вспомним, как мы создали матрицу проекции одного запроса в разд. 16.2.2:

```
>>> torch.manual_seed(123)
>>> d = embedded_sentence.shape[1]
>>> one_U_query = torch.rand(d, d)
```

Теперь предположим, что у нас есть восемь голов внимания — как в исходном трансформере, т. е. $h = 8$:

```
>>> h = 8
>>> multihead_U_query = torch.rand(h, d, d)
>>> multihead_U_key = torch.rand(h, d, d)
>>> multihead_U_value = torch.rand(h, d, d)
```

Как видно из этого кода, можно добавить несколько голов внимания, просто добавив дополнительные измерения.



Разделение данных между несколькими головами внимания

На практике вместо того, чтобы иметь отдельную матрицу для каждой головы внимания, различные реализации трансформера используют единую матрицу для всех голов внимания. Затем головы внимания организуются в логически обособленные области в такой матрице, доступ к которым можно получить с помощью булевых масок. Это позволяет более эффективно реализовывать многоголовое внимание, потому что вместо нескольких матричных умножений можно реализовать одно матричное умножение. Однако для простоты мы опускаем здесь эту деталь реализации.

После инициализации матриц проекций мы можем вычислить спроектированные последовательности аналогично тому, как это делается во взвешенном скалярном внимании. Только вместо вычисления одного набора последовательностей запросов, ключей и значений нам нужно вычислить h их наборов. С формальной точки зрения, например, вычисление проекции запроса для i -й точки данных в j -й голове можно записать следующим образом:

$$\mathbf{q}_j^{(i)} = \mathbf{U}_{\mathbf{q}_j} \mathbf{x}^{(i)}.$$

Затем мы повторяем это вычисление для всех голов.

В коде для второго входного слова в качестве запроса это выглядит следующим образом:

```
>>> multihead_query_2 = multihead_U_query.matmul(x_2)
>>> multihead_query_2.shape
torch.Size([8, 16])
```

Матрица `multihead_query_2` состоит из восьми строк, где каждая строка соответствует j -й голове внимания.

Аналогично мы можем вычислить последовательности ключей и значений для каждой головы:

```
>>> multihead_key_2 = multihead_U_key.matmul(x_2)
>>> multihead_value_2 = multihead_U_value.matmul(x_2)
>>> multihead_key_2[2]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778, 0.6763, 0.6547,
1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

Выход этого кода показывает вектор ключа второго входного элемента через третью голову внимания.

Однако помните, что нам надо повторить вычисления ключа и значения для всех элементов входной последовательности, а не только для `x_2` — это нужно для последующего вычисления самовнимания. Простой и наглядный способ сделать это — расширить встраивания входной последовательности до размера 8 в качестве первого измерения, которое является количеством голов внимания. Для этого мы воспользуемся методом `.repeat()`:

```
>>> stacked_inputs = embedded_sentence.T.repeat(8, 1, 1)
>>> stacked_inputs.shape
torch.Size([8, 16, 8])
```

Затем может следовать пакетное умножение матриц через `torch.bmm()` с головами внимания для вычисления всех ключей:

```
>>> multihead_keys = torch.bmm(multihead_U_key, stacked_inputs)
>>> multihead_keys.shape
torch.Size([8, 16, 8])
```

В этом коде у нас теперь есть тензор, первое измерение которого относится к восьми головам внимания. Второе и третье измерения относятся к размеру встраивания и количеству слов соответственно. Мы поменяем местами второе и третье измерения, чтобы ключи имели более интуитивное представление, т. е. ту же размерность, что и исходная входная последовательность `embedded_sentence`:

```
>>> multihead_keys = multihead_keys.permute(0, 2, 1)
>>> multihead_keys.shape
torch.Size([8, 8, 16])
```

После перестановки мы можем получить доступ ко второму ключевому значению во второй голове внимания следующим образом:

```
>>> multihead_keys[2, 1]
tensor([-1.9619, -0.7701, -0.7280, -1.6840, -1.0801, -1.6778, 0.6763, 0.6547,
1.4445, -2.7016, -1.1364, -1.1204, -2.4430, -0.5982, -0.8292, -1.4401])
```

Мы видим, что это то же значение ключа, которое мы получили ранее через `multihead_key_2[2]`, что подтверждает правильность наших сложных матричных манипуляций и вычислений. Итак, давайте повторим это для последовательностей значений:

```
>>> multihead_values = torch.matmul(
    multihead_U_value, stacked_inputs)
>>> multihead_values = multihead_values.permute(0, 2, 1)
```

Мы следуем шагам расчета одноголового внимания, чтобы вычислить векторы контекста, как описано в разд. 16.2.2. Для краткости мы пропустим промежуточные шаги и предположим, что вычислили векторы контекста для второго элемента ввода в качестве запроса и восьми различных заголовков внимания, которые мы представляем как `multihead_z_2` со случайными данными:

```
>>> multihead_z_2 = torch.rand(8, 16)
```

Первое измерение указывает на восемь голов внимания, а векторы контекста, аналогичные входным предложениям, являются 16-мерными. Если это кажется сложным, рассматривайте `multihead_z_2` как восемь копий $z^{(2)}$, показанных на рис. 16.5, — т. е. у нас есть по одному $z^{(2)}$ на каждую из восьми голов внимания.

Затем мы объединяем эти векторы в один вектор длиной $d_v \times d$ и используем линейную проекцию (через полностью связанный слой), чтобы отобразить его обратно на вектор длины d_v . Этот процесс показан на рис. 16.7.

В коде мы можем реализовать конкатенацию и сжатие следующим образом:

```
>>> linear = torch.nn.Linear(8*16, 16)
>>> context_vector_2 = linear(multihead_z_2.flatten())
>>> context_vector_2.shape
torch.Size([16])
```

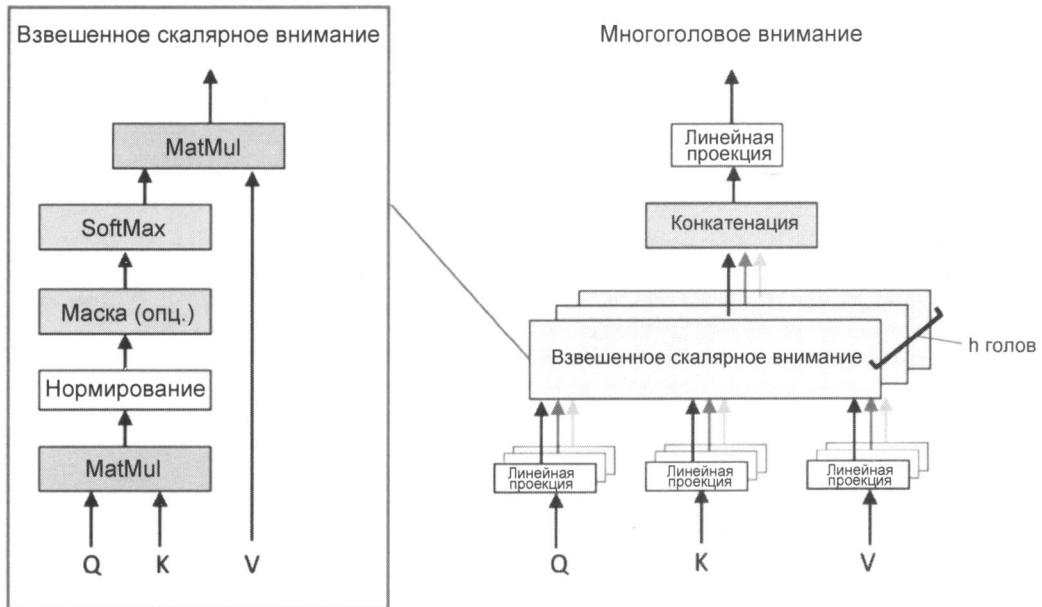


Рис. 16.7. Конкатенация масштабированных векторов внимания в один вектор и его прохождение через линейную проекцию

Подводя итог, можно сказать, что многоголовое самовнимание несколько раз параллельно повторяет вычисление взвешенного скалярного внимания и объединяет результаты. На практике это работает очень хорошо, потому что несколько голов помогают модели собирать информацию из разных частей ввода, что очень похоже на то, как несколько ядер создают несколько каналов в сверточной сети, где каждый канал может собирать информацию о разных признаках. Наконец, несмотря на то, что многоголовое внимание кажется дорогостоящим в вычислительном отношении, нужно иметь в виду, что все вычисления могут выполняться параллельно, поскольку между головами нет зависимостей.

16.3.2. Обучение языковой модели: декодер и маскированное многоголовое внимание

Подобно кодировщику, декодер также содержит несколько повторяющихся слоев. Помимо двух подслоев, которые мы уже представили в предыдущем подразделе про кодировщик (слой многоголового самовнимания и полно связанный слой), каждый повторяющийся слой также содержит маскированный подслой многоголового внимания.

Маскированное внимание — это разновидность исходного механизма внимания, при котором маскированное внимание передает в модель только ограниченную входную последовательность, «маскируя» определенное количество слов. Например, если мы строим модель языкового перевода с помеченным набором данных, в позиции последовательности i во время процедуры обучения мы подаем только правильные выходные слова из позиций $1, \dots, i - 1$. Все остальные слова (например, те, что идут после текущей позиции) скрыты от модели, чтобы модель не «подсматривала». Это также согласуется

с природой генерации текста: хотя правильно переведенные слова известны во время обучения, мы ничего не знаем о правильности на практике. Таким образом, мы можем передать модели только решения того, что она уже сгенерировала для позиции i .

На рис. 16.8 показано, как слои расположены в блоке декодера.

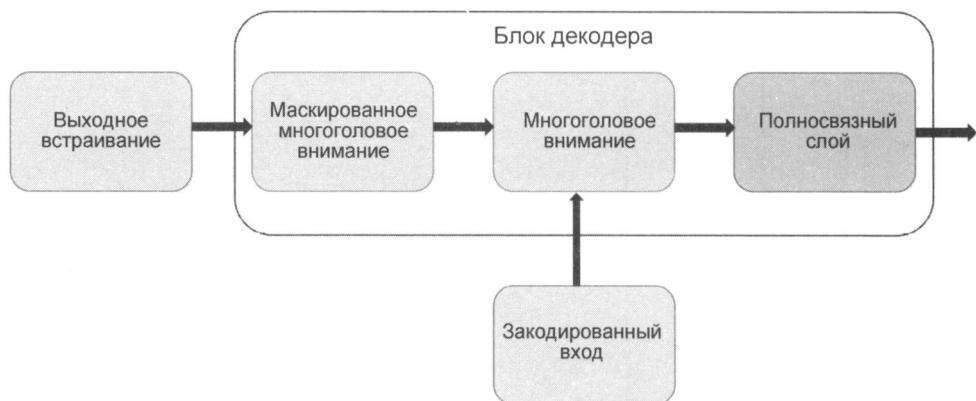


Рис. 16.8. Расположение слоев в блоке декодера

Сначала предыдущие выходные слова (выходные встраивания) передаются в маскированный слой многоголового внимания. Затем следующий слой многоголового внимания получает как закодированные входные данные из блока кодировщика, так и выходные данные уровня маскированного многоголового внимания. Наконец, мы передаем выходные данные слоя многоголового внимания в полносвязный слой, который генерирует общий вывод модели: вектор вероятности, соответствующий выходным словам.

Далее мы можем использовать функцию `argmax` для получения предсказанных слов из этих вероятностей слов аналогично подходу, который мы использовали в рекуррентной нейронной сети в главе 15.

Если сравнивать декодер с кодировщиком, основное различие заключается в диапазоне элементов последовательности, которые может обрабатывать модель. В кодировщике для каждого заданного слова внимание рассчитывается по всем словам в предложении, что можно рассматривать как форму двунаправленного синтаксического анализа ввода. Декодер также получает двунаправленно проанализированные входные данные от кодировщика. Однако когда дело доходит до выходной последовательности, декодер рассматривает только те элементы, которые предшествуют текущей входной позиции, что можно рассматривать как форму одностороннего синтаксического анализа ввода.

16.3.3. Детали реализации: позиционное кодирование и нормализация слоев

В этом подразделе мы более подробно обсудим некоторые детали реализации трансформеров, которые до сих пор рассматривали лишь поверхностно.

Начнем с позиционного кодирования, которое было частью исходной архитектуры трансформера, показанной на рис. 16.6. Позиционное кодирование помогает получать

информацию о порядке входной последовательности и является важной частью трансформера, поскольку как слои взвешенного скалярного внимания, так и полносвязные слои инвариантны к перестановкам. Это означает, что без позиционного кодирования порядок слов игнорируется и не имеет никакого значения для кодирования на основе внимания. Однако мы знаем, что порядок слов важен для понимания предложения. Например, рассмотрим следующие два предложения:

- ◆ Мэри дарит Джону цветок.
- ◆ Джон дарит Мэри цветок.

Эти предложения состоят из одинаковых слов, но их смысл существенно различается.

Трансформеры позволяют одним и тем же словам в разных позициях иметь немного разные кодировки, добавляя вектор небольших значений ко входным встраиваниям в начале блоков кодировщика и декодера. В частности, исходная архитектура Transformer использует так называемое *синусоидальное кодирование*:

$$PE_{(i,2k)} = \sin(pos/10000^{2k/d_{\text{model}}});$$

$$PE_{(i,2k+1)} = \cos(pos/10000^{2k/d_{\text{model}}}).$$

Здесь i — позиция слова, а k — длина вектора кодирования, который имеет тот же размер, что и встраивания входных слов, чтобы позиционные коды и встраивания можно было сложить. Синусоидальные функции применяются для устранения слишком больших позиционных кодов. Например, если бы мы использовали абсолютные позиции 1, 2, 3, ..., n в качестве позиционных кодировок, они бы доминировали над кодировкой слов и сделали бы значения встраивания слов незначительными.

В общем случае существуют два типа позиционного кодирования: абсолютное (как показано в предыдущей формуле) и относительное. Первое отмечает абсолютное положение слов и чувствительно к сдвигу слов в предложении. Другими словами, абсолютные позиционные коды представляют собой фиксированные векторы для каждой заданной позиции. С другой стороны, относительные коды сохраняют только относительное положение слов и инвариантны к сдвигу предложения.

Рассмотрим механизм нормализации слоев, впервые представленный Ба, Киросом и Хинтоном в 2016 году в статье «Layer Normalization»⁵. В то время как пакетная нормализация, которую мы обсудим более подробно в главе 17, приобрела популярность в области компьютерного зрения, послойная нормализация является предпочтительным выбором в контексте NLP, где длина предложений может меняться. На рис. 16.9 показаны основные различия между пакетной и послойной нормализацией.

В то время как пакетная нормализация традиционно выполняется для всех элементов того или иного примера и независимо для каждого примера, послойная нормализация слоев, используемая в трансформерах, расширяет эту концепцию и вычисляет статистику нормализации для всех значений признаков независимо для каждого обучающего примера.

Поскольку при послойной нормализации вычисляют среднее значение и стандартное отклонение для каждого обучающего примера, она ослабляет ограничения размера или

⁵ См. <https://arxiv.org/abs/1607.06450>.

зависимости мини-пакета. Таким образом, в отличие от пакетной нормализации, послойная нормализация способна учиться на данных с мини-пакетами небольшого размера и различной длины. Однако обратите внимание, что исходная архитектура Transformer не имеет входных данных переменной длины (предложения дополняются при необходимости), и, в отличие от RNN, в модели нет рекуррентности. Но почему тогда вместо пакетной нормализации применяют послойную? Трансформеры обычно обучают на очень больших текстовых корпусах, что требует параллельных вычислений, — их бывает сложно выполнить при пакетной нормализации, которая имеет зависимость между обучающими примерами. Послойная нормализация не имеет такой зависимости и поэтому является более естественным выбором для трансформеров.

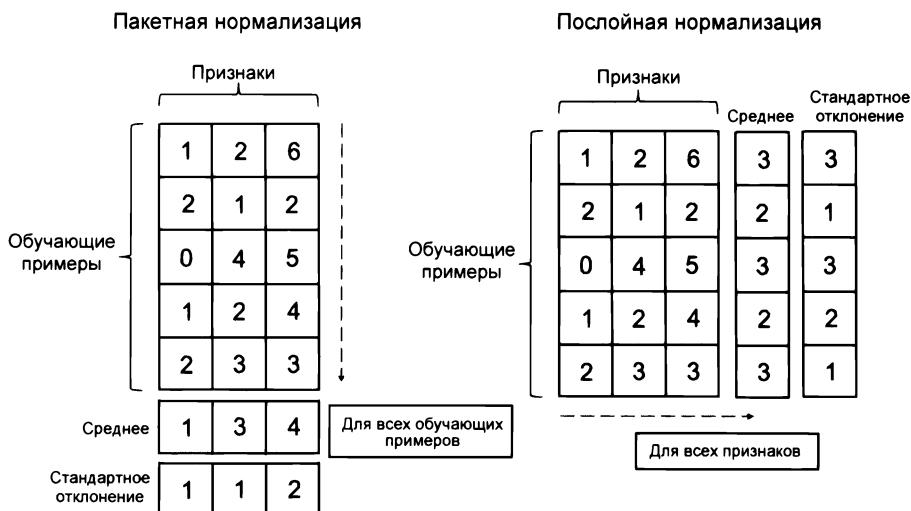


Рис. 16.9. Сравнение пакетной и послойной нормализации

16.4. Построение больших языковых моделей с использованием неразмеченных данных

В этом разделе мы обсудим популярные большие модели-трансформеры, ведущие свое происхождение от оригинальной архитектуры Transformer. Одной из общих черт этих трансформеров является то, что они предварительно обучаются на очень больших неразмеченных наборах данных, а затем точно настраиваются для соответствующих целевых задач. Сначала мы представим общую процедуру обучения современного трансформера и объясним, чем она отличается от обучения исходной архитектуры. Затем перейдем к популярным большим языковым моделям — таким как GPT (Generative Pre-trained Transformer, генеративный предварительно обученный трансформер), BERT (Bidirectional Encoder Representations from Transformers, двунаправленные кодирующие представления из трансформеров) и BART (Bidirectional and Auto-Regressive Transformer, двунаправленный и авторегрессивный трансформер).

16.4.1. Предварительное обучение и тонкая настройка моделей трансформеров

В предыдущем разделе мы обсуждали, как исходную архитектуру Transformer можно использовать для языкового перевода. Языковой перевод — это задача обучения с учителем, требующая помеченного набора данных, получение которого может быть очень дорогим. Отсутствие больших помеченных наборов данных — давняя проблема глубокого обучения, особенно для таких моделей, как трансформеры, которые еще более требовательны к данным, чем другие архитектуры глубокого обучения. Но поскольку каждый день генерируются огромные объемы текстовых данных (книги, веб-сайты и сообщения в социальных сетях), возникает интересный вопрос использования таких неразмеченных данных для улучшения обучения модели.

Ответ на вопрос, можно ли использовать неразмеченные данные в преобразователях, — да, и хитрость заключается в процессе, называемом *самообучением* (*self-supervised learning*), — мы можем генерировать «метки» в результате обучения с учителем из самого простого текста. Например, берем большой неразмеченный текстовый корпус и обучаем модель прогнозировать следующее слово, что позволяет модели изучать распределение вероятностей слов и со временем превратиться в мощную языковую модель.

Самообучение традиционно также называют *предварительным обучением без учителя*, и оно необходимо для успешного обучения современных моделей-трансформеров. «Без учителя» в этом случае относится к тому факту, что мы используем неразмеченные данные, — но раз мы задействуем структуру данных для создания меток (например, в задаче предсказания следующего слова, упомянутой ранее), это нельзя назвать обучением совсем без учителя.

Чтобы более понятно рассказать о том, как работает предварительное обучение без учителя и предсказание следующего слова, если у нас есть предложение, содержащее *i* слов, процедуру предварительного обучения можно разбить на следующие три шага:

1. На первом шаге введите эталонные слова $1, \dots, i - 1$.
2. Попросите модель предсказать слово в позиции i и сравните его с эталонным словом i .
3. Обновите модель и временной шаг $i := i + 1$. Вернитесь к шагу 1 и повторяйте, пока не будут обработаны все слова.

Следует отметить, что на следующей итерации мы всегда подаем модели верное (правильное) слово вместо того, что модель сгенерировала в предыдущем раунде.

Основная идея предварительного обучения состоит в том, чтобы использовать обычный текст, а затем настраивать модель для выполнения более конкретных задач, для которых доступен (меньший) помеченный набор данных. Существует множество различных видов предварительного обучения. Например, ранее упомянутую задачу прогнозирования следующего слова можно рассматривать как односторонний подход к предварительному обучению. Позже мы рассмотрим дополнительные методы предварительного обучения, которые используются в разных языковых моделях для реализации различных функций.

Полная процедура обучения модели-трансформера состоит из двух этапов: (1) предварительное обучение на большом неразмеченном наборе данных и (2) окончательное

обучение (т. е. тонкая настройка) модели для конкретных последующих задач с использованием размеченного набора данных. На первом этапе предварительно обученная модель не предназначена для какой-либо конкретной задачи, а скорее обучается как «общая» языковая модель. После этого на втором этапе модель можно перенести на любую индивидуальную задачу посредством обычного обучения с учителем на размеченном наборе данных.

С точки зрения представлений, которые можно получить из предварительно обученной модели, в основном существуют две стратегии переноса и адаптации модели к конкретной задаче: (1) подход, основанный на признаках, и (2) подход, основанный на точном обучении. (Здесь мы можем рассматривать представления как скрытые активации последних слоев модели.)

Подход, основанный на признаках, использует предварительно обученные представления в качестве дополнительных признаков к помеченному набору данных. Для этого мы должны научиться извлекать признаки предложения из предварительно обученной модели. Одна из первых моделей, которая получила известность благодаря этому подходу к извлечению признаков, — это ELMo (Embeddings from Language Models, встраивания из языковых моделей)⁶. ELMo представляет собой предварительно обученную двунаправленную языковую модель, которая маскирует определенную долю слов. В частности, она случайным образом маскирует 15% входных слов во время предварительного обучения, а задача моделирования состоит в том, чтобы заполнить эти пробелы, т. е. предсказать недостающие (замаскированные) слова. В этом заключается отличие от упоминавшегося ранее одностороннего обучения, при котором маскируют все будущие слова на временном шаге i . Двунаправленная маскировка позволяет модели учиться с обеих сторон и, таким образом, собирать более целостную информацию о предложении. Предварительно обученная модель ELMo может генерировать высококачественные представления предложений, которые впоследствии служат входными признаками для частных задач. Иными словами, мы можем рассматривать подход на основе признаков как метод извлечения признаков на базе модели, аналогичный анализу главных компонентов, с которым вы познакомились в главе 5.

С другой стороны, в соответствии с подходом, основанным на тонкой настройке, мы обновляем параметры предварительно обученной модели в обычном режиме обучения с учителем посредством обратного распространения ошибки. В отличие от метода, основанного на признаках, к предварительно обученной модели обычно добавляют полносвязный слой для выполнения определенных задач — таких как классификация, а затем обновляют всю модель с использованием потерь прогнозирования на размеченном обучающем наборе. Одной из популярных моделей, использующих этот подход, является BERT, крупномасштабная модель-трансформер, предварительно обученная как двунаправленная языковая модель. Немного позже мы поговорим о BERT более подробно. Кроме того, в последнем разделе этой главы будет приведен пример кода, демонстрирующий тонкую настройку предварительно обученной модели BERT для классификации эмоциональной тональности текста с использованием набора данных отзывов о фильмах, с которым мы уже работали в главах 8 и 15.

⁶ Предложена Петерсон и его коллегами в 2018 году в статье «Deep Contextualized Word Representations», <https://arxiv.org/abs/1802.05365>.

Прежде чем перейти к следующему разделу и начать обсуждение популярных языковых моделей на основе архитектуры Transformer, взгляните на рис. 16.10, где показаны два этапа обучения трансформеров и отмечена разница между подходами на основе признаков и тонкой настройки.

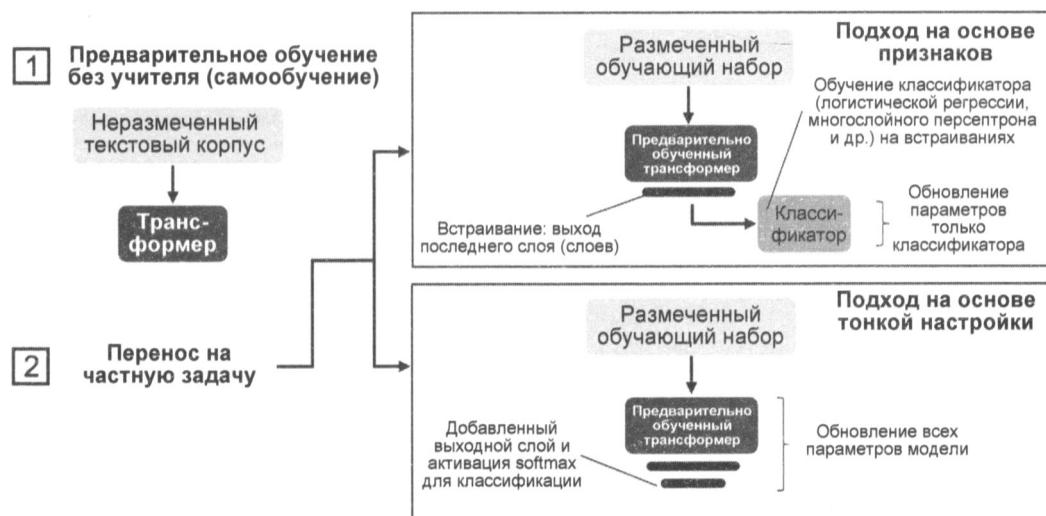


Рис. 16.10. Два основных способа использования предварительно обученного трансформера для последующих задач

16.4.2. Использование неразмеченных данных с помощью GPT

Генеративные предварительно обученные трансформеры (Generative Pre-trained Transformer, GPT) — это популярная серия больших языковых моделей для генерации текста, разработанная OpenAI. Самая последняя модель, GPT-3, которая была выпущена в мае 2020 года (анонсирована в статье «Language Models are Few-Shot Learners»), дает поразительные результаты. Тексты, созданные моделью GPT-3, очень трудно отличить от текстов, созданных человеком. В этом подразделе мы собираемся обсудить, как модели GPT удается работать на столь высоком уровне и как она развивалась с годами.

Как можно видеть в табл. 16.1, одним из наиболее явных усовершенствований в серии моделей GPT является количество параметров.

Но не будем забегать вперед и рассмотрим сначала модель GPT-1, выпущенную в 2018 году. Процедуру ее обучения можно разложить на два этапа:

1. Предварительное обучение на большом количестве неразмеченного простого текста.
2. Тонкое обучение с учителем.

Как показано на рис. 16.11, мы можем рассматривать GPT-1 как трансформер, состоящий из (1) декодера (без блока кодировщика) и (2) дополнительного слоя, который добавляется позже для обучения с учителем при подготовке к решению конкретных задач.

Таблица 16.1. Обзор моделей GPT

Модель	Год выпуска	Кол-во параметров	Название статьи	Ссылка на статью
GPT-1	2018	110 млн	Improving Language Understanding by Generative Pre-Training	https://www.cs.ubc.ca/~amuhamed/LING530/papers/radford2018improving.pdf
GPT-2	2019	1.5 млрд	Language Models are Unsupervised Multitask Learners	https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe
GPT-3	2020	175 млрд	Language Models are Few-Shot Learners	https://arxiv.org/pdf/2005.14165.pdf

Обратите внимание, что если нашей задачей является предсказание текста (точнее, предсказание следующего слова), то модель готова к использованию сразу после этапа предварительного обучения. В противном случае, например, если наша задача связана с классификацией или регрессией, то требуется дополнительное обучение с учителем.

Во время предварительного обучения GPT-1 использует структуру декодера трансформера, когда, находясь на текущей позиции в предложении, модель полагается только на предшествующие слова для предсказания следующего слова. GPT-1 задействует одонаправленный механизм внутреннего внимания, в отличие от двунаправленного, как в BERT (о котором мы поговорим в этой главе позже), потому что модель GPT-1 ориентирована на создание текста, а не на классификацию. Во время генерации текста она создает слова одно за другим с естественным направлением слева направо. Здесь стоит выделить еще один аспект: во время процедуры обучения для каждой позиции мы всегда подаем модели правильные слова из предыдущих позиций. Однако во время логического вывода мы просто подаем в модель любые слова, которые она сгенерировала, чтобы продолжать генерировать новые тексты.

Получив предварительно обученную модель (блок **Трансформер** на рис. 16.11), мы затем вставляем ее между входным блоком предварительной обработки и линейным слоем, который служит выходным слоем (как в глубоких нейросетевых моделях, которые мы обсуждали ранее в этой книге). Для задач классификации тонкая настройка очень проста — сначала токенизация входных данных, а затем передача их в предварительно обученную модель и недавно добавленный линейный слой, за которым следует функция активации softmax. Однако для более сложных задач, таких как ответы на вопросы, входные данные организованы в определенном формате, который не обязательно соответствует предварительно обученной модели, что требует дополнительного этапа обработки, настроенного для каждой задачи. Читателям, интересующимся конкретными модификациями, рекомендуется ознакомиться с подробным описанием GPT-1 для получения дополнительной информации (ссылка приведена в табл. 16.1).

GPT-1 также удивительно хорошо справляется с *задачами без знакомых примеров* (zero-shot task), что доказывает ее способность быть общей языковой моделью, которую можно настраивать для различных типов задач с минимальной тонкой настройкой для конкретной задачи. Обучение без знакомых примеров обычно описывает особую ситуацию в машинном обучении, когда во время тестирования и вывода модель должна

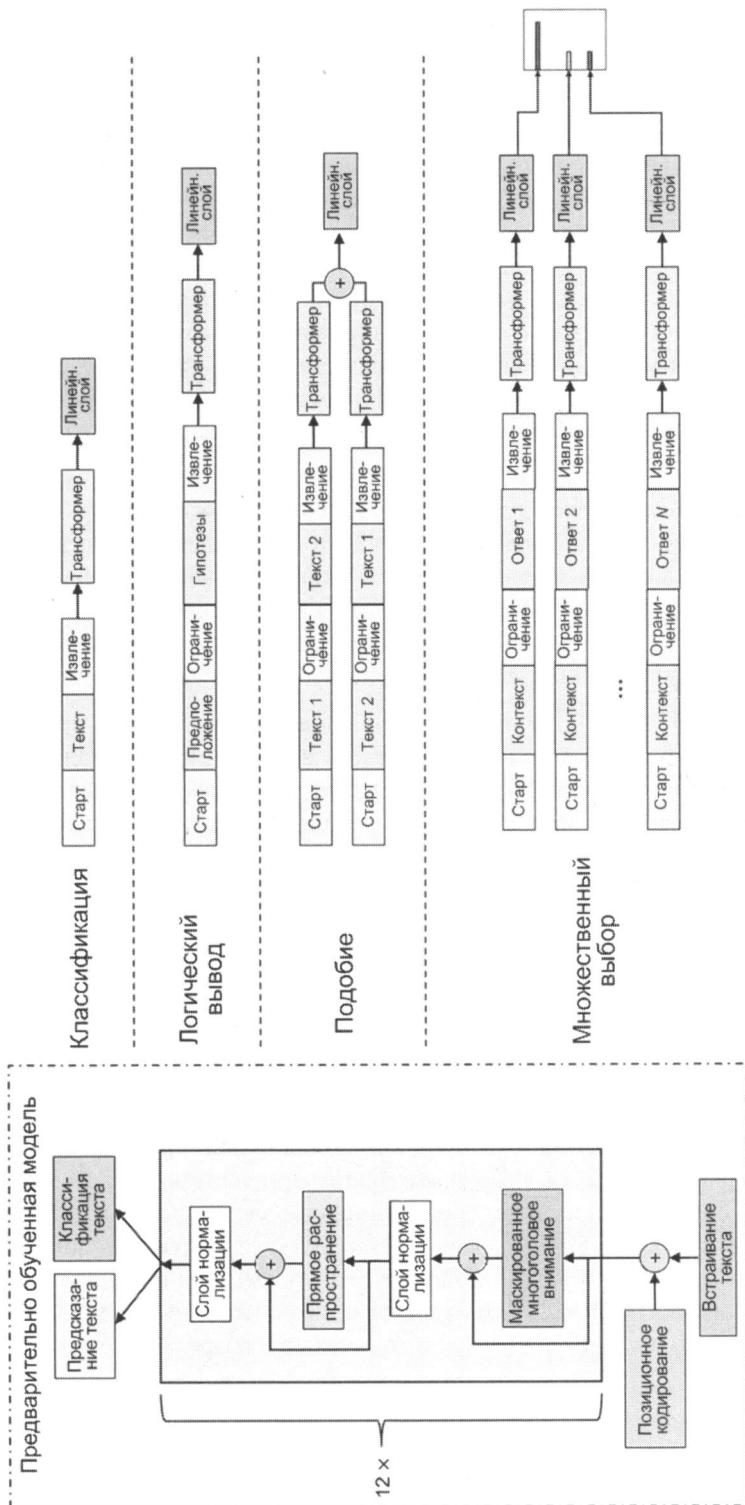


Рис. 16.11. Структура трансформера GPT-1 (подготовлено на основе оригинальной статьи о GPT-1)

классифицировать образцы из классов, которые не наблюдались во время обучения. В контексте GPT эта ситуация означает работу с незнакомыми задачами.

Вдохновившись высокой адаптивностью GPT, исследователи избавились от ввода данных для конкретных задач и настройки модели, что привело к разработке GPT-2. В отличие от своей предшественницы, GPT-2 больше не требует дополнительной модификации на этапах ввода или тонкой настройки. Вместо того, чтобы переупорядочивать последовательности в соответствии с требуемым форматом, мы можем полагаться на способность GPT-2 различать разные типы входных данных и выполнять соответствующие последующие задачи с второстепенными подсказками — так называемыми *контекстами*. Это достигается путем моделирования выходных вероятностей, обусловленных как входными данными, так и типом задачи $p(\text{вывод}|\text{ввод, задача})$, вместо того, чтобы ограничиваться только входными данными. Например, ожидается, что модель распознает задачу перевода, если контекст содержит «перевод на французский», «английский текст», «французский текст».

Такая модель выглядит гораздо более «интеллектуально», чем базовая GPT, и способность распознавать контекст действительно является наиболее заметным ее улучшением, помимо размера модели. Как видно из названия соответствующей статьи («Language Models are Unsupervised Multitask Learners»⁷), обучаемая без учителя языковая модель может служить ключом к обучению без примеров, и GPT-2 в полной мере использует перенос задач без примеров для создания многозадачного ученика.

По сравнению с GPT-2, GPT-3 менее «амбициозна» в том смысле, что она смещает акцент с обучения с нуля на обучение в контексте на одном или нескольких примерах. Хотя предоставление обучающих примеров для конкретных задач кажется слишком строгим требованием, обучение с использованием нескольких примеров не только более реалистично, но и больше похоже на обучение человека: людям обычно нужно увидеть несколько примеров, чтобы иметь возможность изучить новую задачу. Как следует из названия статьи, презентующей модель (см. табл. 16.1), обучение на нескольких примерах означает, что модель видит несколько примеров задачи, в то время как обучение на одном примере ограничено ровно одним примером.

На рис. 16.12 показана разница между процедурами обучения без примеров, на одном примере, на нескольких примерах и с тонкой настройкой.

Архитектура модели GPT-3 практически такая же, как и GPT-2, за исключением стократного увеличения количества параметров и использования разреженного трансформера. В первоначальном (плотном) механизме внимания, который мы обсуждали ранее, каждый элемент обращает внимание на все остальные элементы на входе, что увеличивает сложность в $O(n^2)$ раз. *Разреженное внимание* (sparse attention) повышает эффективность за счет внимания только к подмножеству элементов ограниченного размера, обычно пропорционального $n^{1/p}$. Заинтересованные читатели могут узнать больше о выборе конкретного подмножества, прочитав статью о разреженных трансформерах⁸.

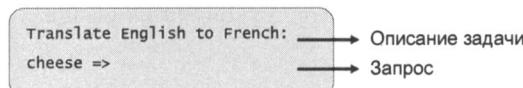
⁷ «Языковые модели — это многозадачные ученики, обучаемые без учителя».

⁸ См. «Generating Long Sequences with Sparse Transformers» by Rewon Child et al., 2019, <https://arxiv.org/abs/1904.10509>.

Три варианта, рассматриваемые при контекстном обучении

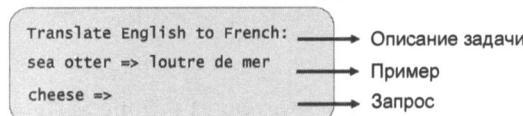
1. Без примеров (Zero-shot)

Модели дается только описание задачи на естественном языке. Обновления веса не выполняются.



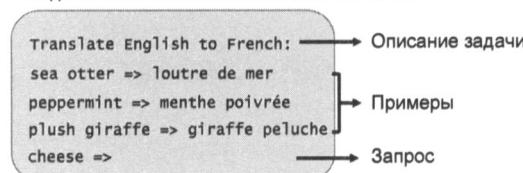
2. Один пример (One-shot)

Помимо описания задачи, модели также предоставляют простой пример задачи. Обновления веса не выполняются.



3. Несколько примеров (Few-shot)

Помимо описания задачи, модели также предоставляют несколько простых примеров задачи. Обновления веса не выполняются.



Традиционная тонкая настройка (не для GPT-3)

Тонкая настройка

Модель обучается на большом корпусе примеров задач

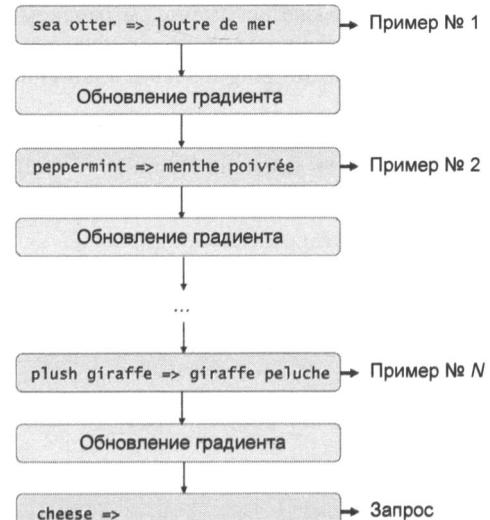


Рис. 16.12. Сравнение обучения без примеров, на одном примере и на нескольких примерах

16.4.3. Использование GPT-2 для создания нового текста

Прежде чем мы перейдем к следующему варианту трансформерной архитектуры, давайте посмотрим, как последние модели GPT применяются для создания нового текста. Когда мы работали над этой книгой, GPT-3 все еще оставалась относительно новой моделью и была доступна только в виде бета-версии через API OpenAI по адресу: <https://openai.com/blog/openai-api/>. Однако была свободно доступна реализация GPT-2, предоставленная Hugging Face (популярная компания NLP и машинного обучения, <http://huggingface.co>), которой мы и воспользовались.

Hugging Face создала достаточно обширную библиотеку transformers на языке Python, предоставляющую доступ к различным моделям на основе трансформеров для предварительной их подготовки и тонкой настройки⁹.



Установка библиотеки transformers версии 4.9.1

Поскольку этот пакет очень быстро развивается, не исключено, что, установив текущую версию библиотеки, вы не сможете воспроизвести примеры, рассмотренные

⁹ Присоединившись к сообществу пользователей этой библиотеки на форуме по адресу: <https://discuss.huggingface.co>, все желающие могут обсуждать готовые модели и делиться своими собственными вариантами моделей.

в следующих разделах. В этой книге мы используем версию 4.9.1, выпущенную в июне 2021 года. Чтобы установить эту версию, выполните следующую команду в своем терминале, чтобы установить ее из PyPI:

```
pip install transformers==4.9.1
```

Мы также рекомендуем ознакомиться с актуальными инструкциями на официальной странице установки по адресу:

<https://huggingface.co/transformers/installation.html>.

Установив библиотеку `transformers`, попробуйте запустить следующий код, чтобы импортировать предварительно обученную модель GPT, которая может генерировать новый текст:

```
>>> from transformers import pipeline, set_seed
>>> generator = pipeline('text-generation', model='gpt2')
```

Затем мы можем предоставить модели запрос в виде фрагмента текста и попросить ее сгенерировать новый текст на основе этого входного фрагмента:

```
>>> set_seed(123)
>>> generator("Hey readers, today is",
... max_length=20,
... num_return_sequences=3)
[{'generated_text': "Hey readers, today is not the last time we'll be seeing one
of our favorite indie rock bands"},

{'generated_text': 'Hey readers, today is Christmas. This is not Christmas,
because Christmas is so long and I hope'},

{'generated_text': "Hey readers, today is CTA Day!\n\nWe're proud to be
hosting a special event"}]
```

Как видно из вывода, модель сгенерировала три относительно осмысленных предложения на основе нашего текстового фрагмента. Если вы хотите получить больше примеров, попробуйте изменять случайное начальное число и максимальную длину последовательности.

Кроме того, как ранее показано на рис. 16.10, мы можем использовать трансформер, чтобы создавать признаки для обучения других моделей. Следующий код показывает, как мы можем использовать GPT-2 для создания признаков на основе входного текста:

```
>>> from transformers import GPT2Tokenizer
>>> tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
>>> text = "Let us encode this sentence"
>>> encoded_input = tokenizer(text, return_tensors='pt')
>>> encoded_input
{'input_ids': tensor([[ 5756, 514, 37773, 428, 6827]]), 'attention_mask':
tensor([[1, 1, 1, 1, 1]])}
```

Здесь мы закодировали в токенизованный формат текст входного предложения для модели GPT-2. Как видите, модель сопоставила строки с целочисленным представлением и задала маску внимания из всех единиц, — т. е. все слова будут обработаны, когда мы передадим закодированный ввод в модель:

```
>>> from transformers import GPT2Model
>>> model = GPT2Model.from_pretrained('gpt2')
>>> output = model(**encoded_input)
```

Переменная `output` хранит последнее скрытое состояние, т. е. кодировку признаков входного предложения на основе GPT-2:

```
>>> output['last_hidden_state'].shape
torch.Size([1, 5, 768])
```

Чтобы скрыть подробный вывод, мы показали только форму тензора. Его первое измерение — это размер пакета (у нас есть только один входной текст), за которым следует длина предложения и размер кодировки признака. Здесь каждое из пяти слов закодировано как 768-мерный вектор.

Теперь мы можем применить это кодирование признаков к заданному набору данных и обучить последующий классификатор на основе представления признаков, созданных GPT-2, вместо использования модели набора слов, как обсуждалось в главе 8.

Кроме того, альтернативным подходом к использованию больших предварительно обученных языковых моделей является их тонкая настройка, как мы обсуждали ранее. Позже в этой главе вы увидите пример тонкой настройки.

Если вас интересуют дополнительные сведения об использовании GPT-2, мы рекомендуем ознакомиться со следующими источниками:

- ◆ <https://huggingface.co/gpt2>;
- ◆ https://huggingface.co/docs/transformers/model_doc/gpt2.

16.4.4. Двунаправленное предварительное обучение модели BERT

Аббревиатура BERT расшифровывается так: Bidirectional Encoder Representations from Transformers (двунаправленные кодирующие представления из трансформеров). Она была создана исследовательской группой Google в 2018 году¹⁰. Для справки: хотя мы не можем напрямую сравнивать GPT и BERT, поскольку это разные архитектуры, BERT имеет 345 млн параметров (что делает ее лишь немного больше, чем GPT-1, и составляет лишь $\frac{1}{5}$ от GPT-2).

Как следует из названия, в основе BERT лежит структура трансформера-кодировщика, в которой используется двунаправленная процедура обучения. (Или, точнее, мы можем рассматривать обучение BERT как «ненаправленный» процесс, потому что она считывает все входные элементы одновременно.) В этом случае кодирование определенного слова зависит как от предыдущего слова, так и от последующего. Напомним, что в GPT входные элементы считаются в естественном порядке слева направо, что помогает сформировать мощную генеративную языковую модель. Двунаправленное обучение лишает BERT способности генерировать предложение слово за словом, но обеспечивает входные кодировки более высокого качества для других задач — таких как классификация, поскольку теперь модель может обрабатывать информацию в обоих направлениях.

Напомним также, что в кодировщике трансформера кодирование токена представляет собой суммирование позиционного кодирования и встраиваний токенов. В кодировщи-

¹⁰ См. «BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding» by J. Devlin, M. Chang, K. Lee, and K. Toutanova, <https://arxiv.org/abs/1810.04805>.

ке BERT предусмотрено дополнительное встраивание сегмента, указывающее, к какому сегменту принадлежит тот или иной токен. Это означает, что каждое представление токена содержит три компонента, как показано на рис. 16.13.

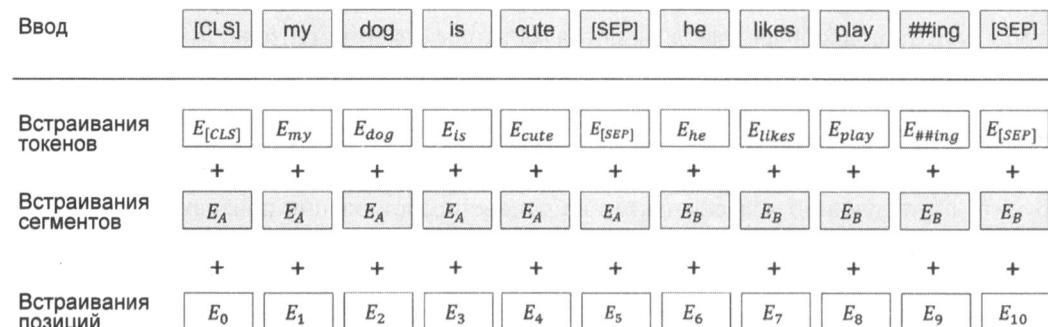


Рис. 16.13. Подготовка ввода для кодировщика BERT

Зачем нам нужна в BERT эта дополнительная информация о сегменте? Потребность в информации о сегменте возникла из-за специальной задачи предварительного обучения BERT, называемой *прогнозированием следующего предложения* (*next-sentence prediction*). В этой предварительной обучающей задаче каждый обучающий пример включает два предложения и, следовательно, требует специальной нотации сегмента, чтобы указать, принадлежит ли он к первому или ко второму предложению.

Теперь рассмотрим задачи предварительного обучения BERT более подробно. Подобно всем другим языковым моделям на основе трансформеров, BERT имеет два этапа обучения: предварительное обучение и тонкую настройку. Предварительное обучение включает в себя две задачи без учителя: маскированное языковое моделирование и предсказание следующего предложения.

В задаче *маскированного языкового моделирования* (Masked Language Model, MLM) токены случайным образом заменяются так называемыми *лексемами маски* [MASK], и модель должна предсказать эти скрытые слова. По сравнению с предсказанием следующего слова в GPT, задача MLM в BERT больше похожа на «заполнение пробелов», потому что модель может обрабатывать все токены в предложении (кроме замаскированных). Однако простое маскирование слов может привести к несоответствию между предварительным обучением и тонкой настройкой, поскольку токены [MASK] не появляются в обычных текстах. Чтобы облегчить задачу, в слова, выбранные для маскирования, вносят дополнительные изменения. Допустим, что 15% слов в BERT выбраны для маскирования. Затем эти 15% случайно выбранных слов обрабатываются следующим образом:

1. Не меняют слово в 10% случаев.
2. Заменяют исходный токен слова случайным словом в 10% случаев.
3. Заменяют токен исходного слова токеном маски [MASK] в 80% случаев.

Помимо устранения упомянутого несоответствия между предварительным обучением и тонкой настройкой при введении токенов [MASK] в процедуру обучения, эти модификации также имеют другие преимущества. Во-первых, неизменяемые слова включают

в себя возможность сохранения информации исходного токена, — в противном случае модель может учиться только на контексте и ничего не извлекает из замаскированных слов. Во-вторых, 10% случайных слов не дают модели стать «ленивой» — например, не учиться ничему, кроме как возвращать то, что поступило на ввод. Вероятности маскирования, рандомизации и оставления слов без изменений были выбраны с помощью исследования аблляции (см. статью про GPT-2, упомянутую в табл. 16.1). Авторы статьи протестирували различные настройки и обнаружили, что эта комбинация работает лучше всего.

На рис. 16.14 показан пример, когда слово *лиса* замаскировано и с определенной вероятностью остается неизменным или заменяется на [MASK] или *кофе*. Затем модель должна предсказать, что представляет собой замаскированное (выделенное) слово.



Рис. 16.14. Пример задачи MLM

Прогнозирование следующего предложения является естественной модификацией задачи прогнозирования следующего слова с использованием двунаправленного кодирования BERT. На самом деле, многие важные задачи NLP — такие как ответы на вопросы, зависят от взаимосвязи двух предложений в документе. Такого рода отношения трудно зафиксировать с помощью обычных языковых моделей, потому что обучение прогнозированию следующего слова обычно происходит на уровне одного предложения из-за ограничений длины входных данных.

В задаче прогнозирования следующего предложения модели даются два предложения: А и В — в следующем формате:

[CLS] А [SEP] В [SEP].

[CLS] — это токен классификации, который служит заполнителем для прогнозируемой метки в выходных данных декодера, а также токеном, обозначающим начало предложения. Токен [SEP], в свою очередь, применяется для обозначения конца каждого предложения. Затем модель должна классифицировать, является ли предложение В продолжением предложения А («IsNext») или нет. Чтобы обеспечить модель сбалансированным набором данных, 50% обучающих примеров помечены как «IsNext» (является продолжением), а остальные образцы помечены как «NotNext» (не является продолжением).

Модель BERT предварительно обучена этим двум задачам: предсказанию маскированных слов и предсказанию следующего предложения одновременно. Цель обучения BERT состоит в том, чтобы минимизировать комбинированную функцию потерь обеих задач.

При наличии предварительно обученной модели на этапе тонкой настройки необходимо внести определенные модификации для различных последующих задач. Каждый пример ввода должен соответствовать определенному формату — например, он должен начинаться с токена [CLS] и отделяться токенами [SEP], если состоит из более чем одного предложения.

Упрощенно говоря, BERT можно настроить на четыре категории задач: (a) классификация пар предложений, (b) классификация одного предложения, (c) ответы на вопросы, (d) присвоение метки одному предложению.

Среди них (a) и (b) — задачи классификации на уровне последовательности, которые требуют только добавления дополнительного слоя softmax к выходному представлению токена [CLS]. Задачи (c) и (d) представляют собой классификацию на уровне токенов. Это означает, что модель передает выходные представления всех связанных токенов на слой softmax, чтобы предсказать метку класса для каждого отдельного токена.



Ответ на вопрос

Задача (c) — ответ на вопрос — обсуждается реже по сравнению с другими популярными задачами классификации, такими как классификация тональности или присвоение метки. При ответе на вопрос каждый входной пример можно разделить на две части: вопрос и абзац, помогающий ответить на вопрос. Модель должна указывать как начальный, так и конечный токен в абзаце, который образует правильный ответ на вопрос. Это означает, что модель должна генерировать тег для каждого отдельного токена в абзаце, указывающий, является ли этот токен начальным или конечным токеном, или ни тем ни другим. В качестве примечания стоит упомянуть, что вывод может содержать конечный токен, который появляется перед стартовым токеном, что приведет к конфликту при генерации ответа. Этот вид вывода будет распознан как «No Answer» (Нет ответа).

Как показано на рис. 16.15, тонкая настройка модели имеет очень простую структуру: входной кодировщик подключается к предварительно обученной модели BERT, а для классификации добавляется слой softmax. Все параметры этой структуры будут корректироваться в процессе обучения.

16.4.5. Лучшее из двух миров: BART

Двунаправленный и авторегрессионный трансформер BART (Bidirectional and Auto-Regressive Transformer) был разработан исследователями Facebook AI Research в 2019 году¹¹. Как вы помните, в предыдущих подразделах мы говорили о том, что GPT использует структуру декодера-трансформера, а BERT — структуру кодировщика-трансформера. Как следствие, эти две модели способны хорошо выполнять разные задачи: специализация GPT — генерация текста, тогда как BERT лучше справляется с задачами классификации. Модель BART можно рассматривать как своего рода объединение GPT и BERT. Как следует из названия этого раздела, BART способен выполнять обе задачи, генерируя и классифицируя текст. Причина, по которой он хорошо справляется с обеими задачами, заключается в том, что модель оснащена двунаправленным кодировщиком, а также авторегрессионным декодером слева направо.

¹¹ См. «BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension», Lewis and colleagues, <https://arxiv.org/abs/1910.13461>.

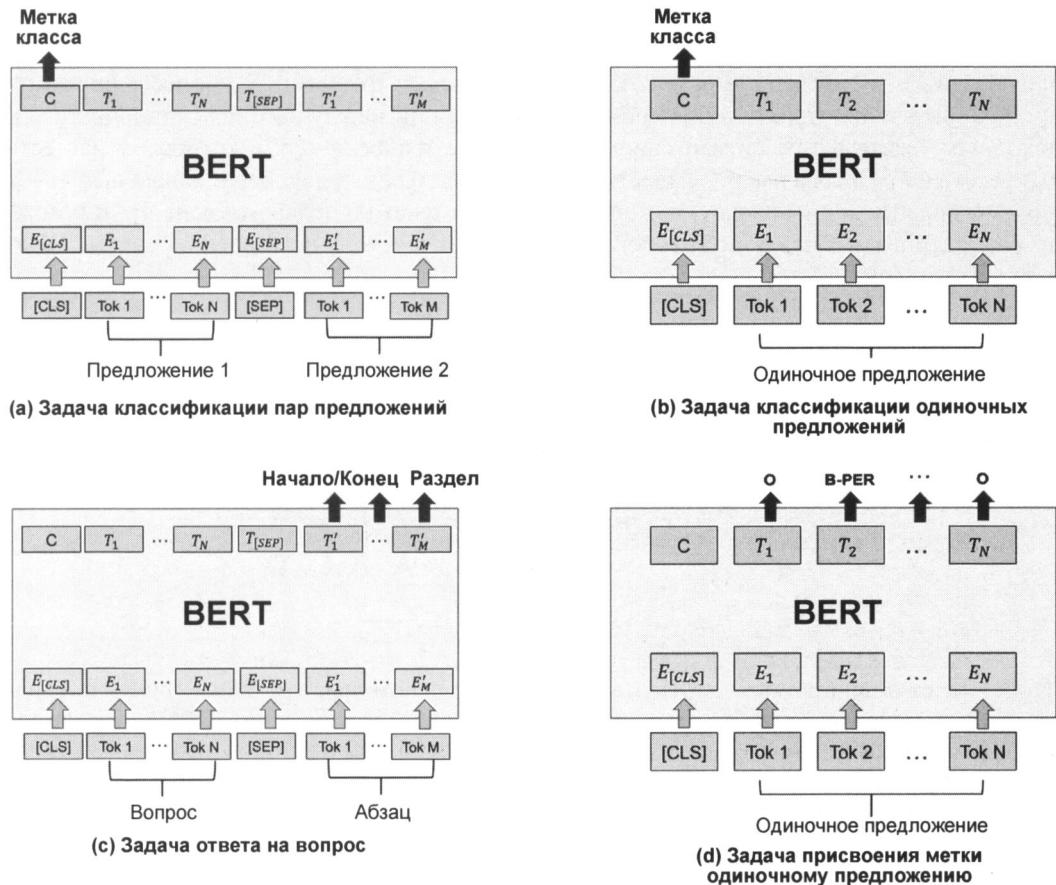


Рис. 16.15. Тонкая настройка модели BERT под различные языковые задачи

Вы можете спросить, в чем тогда отличие BART от обычного трансформера? Действительно, в нем лишь сделано несколько изменений в размере модели, а также внесены некоторые незначительные структурные различия — такие как выбор функции активации. Однако одно из наиболее интересных отличий заключается в том, что BART работает с разными входными данными. Исходный трансформер был разработан для языкового перевода, поэтому у него есть два входа данных: текст для перевода (исходная последовательность) для кодировщика и перевод (целевая последовательность) для декодера. Кроме того, декодер также получает закодированную исходную последовательность, как было показано ранее на рис. 16.6. Однако в BART входной формат обобщен таким образом, что в качестве входных данных используется только исходная последовательность. BART, таким образом, может выполнять более широкий спектр задач, включая языковой перевод, где целевая последовательность по-прежнему требуется для вычисления потерь и тонкой настройки модели, но нет необходимости вводить ее непосредственно в декодер.

Теперь давайте подробнее рассмотрим структуру модели BART. Как упоминалось ранее, BART состоит из двунаправленного кодера и авторегрессионного декодера. При

получении обучающего примера в виде обычного текста входные данные сначала будут «искажены», а затем закодированы кодировщиком. Потом эти входные кодировки будут переданы декодеру вместе со сгенерированными токенами. В процессе обучения происходит вычисление потери перекрестной энтропии между выходом кодировщика и исходным текстом и ее оптимизация. Представьте трансформер, в котором у нас есть два текста на разных языках в качестве входных данных для декодера: начальный текст для перевода (исходный текст) и сгенерированный текст на целевом языке. BART можно рассматривать как замену первого искаженным текстом, а второго — самим входным текстом (рис. 16.16).

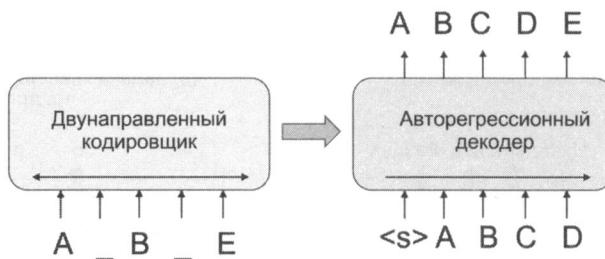


Рис. 16.16. Структура модели BART

Чтобы более подробно объяснить этап искажения, вспомним, что BERT и GPT предварительно обучаются путем восстановления замаскированных слов: BERT «заполняет пробелы», а GPT «предсказывает следующее слово». Эти основанные на предварительном обучении задачи также можно рассматривать как восстановление искаженных предложений, потому что маскирование слов — это один из способов искажения предложения. BART допускает следующие методы искажения, которые можно применить к исходному тексту:

- ◆ маскировка токена;
- ◆ удаление токена;
- ◆ текстовое заполнение;
- ◆ перестановка предложений;
- ◆ вращение документа.

К одному и тому же предложению можно применить один или несколько из указанных приемов — в худшем случае, когда вся информация загрязнена и искажена, текст становится бесполезным. Следовательно, кодировщик имеет ограниченную полезность, и только с правильно работающим модулем декодера выход модели по существу станет более похожим на однонаправленный язык.

После тонкой настройки BART можно применять для решения широкого круга задач, включая: (а) классификацию последовательностей, (б) классификацию токенов, (с) генерацию последовательности и (д) машинный перевод. Как и в случае с BERT, в соответствии с конкретной задачей необходимо будет внести небольшие изменения во входные данные.

В задаче классификации последовательности к входным данным необходимо прикрепить дополнительный токен, который будет служить сгенерированным токеном метки,

аналогичным токену [CLS] в BERT. Кроме того, в этом случае как в кодировщик, так и в декодер подают неискаженный входной текст, чтобы модель могла использовать его полностью.

В задаче классификации токенов дополнительные токены становятся ненужными, и модель может напрямую использовать сгенерированное представление для каждого токена.

Генерация последовательности в BART немного отличается от GPT из-за наличия кодировщика. В отличие от генерации текста с нуля, задачи генерации последовательности с помощью BART более сравнимы с резюмированием (кратким изложением смысла), когда модели дается корпус контекстов и предлагается создать резюме или абстрактный ответ на определенные вопросы. Для этого целевые входные последовательности подаются в кодировщик, в то время как декодер авторегрессивно генерирует выходные данные.

Наконец, если учесть сходство между BART и исходным трансформером, машинный перевод является для BART естественной задачей. Однако вместо того, чтобы следовать той же процедуре, что и для обучения исходного трансформера, исследователи предусмотрели возможность использования всей модели BART в качестве предварительно обученного декодера. Для построения модели машинного перевода в качестве дополнительного кодировщика добавляют новый набор случайно инициализируемых параметров. Затем выполняют два этапа тонкой настройки:

1. Сначала замораживают все параметры, кроме кодировщика.
2. Затем обновляют все параметры в модели.

BART испытали на нескольких эталонных наборах данных для различных задач, и эта модель получила очень конкурентоспособные результаты по сравнению с другими известными языковыми моделями, такими как BERT. В частности, в задачах генерации текста, включая ответы на абстрактные вопросы, ведение диалога и резюмирование, BART достиг самых передовых результатов.

16.5. Тонкая настройка модели BERT в PyTorch

Теперь, когда вы усвоили все необходимые понятия и теорию оригинальной архитектуры Transformer и популярных моделей на ее основе, пришло время практических занятий! В этом разделе вы узнаете, как настроить модель BERT для *классификации эмоциональной окраски текста* в PyTorch.

Нужно сказать, что, несмотря на существование множества других трансформеров, BERT обеспечивает хороший баланс между потенциалом и разумным размером модели, чтобы ее можно было точно настроить на одном графическом процессоре. Также заметим, что предварительное обучение модели BERT с нуля — очень трудоемкая и абсолютно лишняя процедура, учитывая доступность предоставляемого Hugging Face пакета Python `transformers`, включающего в себя набор предварительно обученных моделей, готовых к тонкой настройке.

В следующих подразделах вы узнаете, как подготовить и разметить набор данных обзора фильмов IMDb, а также настроить дистиллированную модель BERT (DistilBERT) для выполнения классификации тональности (эмоциональной окраски) текста. Мы на-

меренно выбрали классификацию тональности в качестве простого, но классического примера, хотя есть много других интересных применений языковых моделей. Кроме того, используя знакомый набор данных обзора фильмов IMDb, мы можем получить хорошее представление о прогностической эффективности модели BERT, сравнив ее с моделью логистической регрессии, представленной в главе 8, и RNN, рассмотренной в главе 15.

16.5.1. Загрузка набора данных обзора фильмов IMDb

Начнем с загрузки необходимых пакетов и набора данных, разделенного на поднаборы для обучения, валидации и тестирования.

При работе с BERT мы будем в основном ориентироваться на созданную Hugging Face библиотеку `transformers` с открытым исходным кодом (<https://huggingface.co/transformers/>), которую установили в разд. 16.4.3.

Модель DistilBERT, которую мы здесь задействуем, представляет собой облегченную модель-трансформер, созданную путем дистилляции предварительно обученной базовой модели BERT. Исходная базовая модель BERT содержит более 110 млн параметров, в то время как у DistilBERT параметров на 40% меньше. Кроме того, DistilBERT работает на 60% быстрее и сохраняет 95% производительности BERT в teste на понимание языка GLUE.

Следующий код импортирует все пакеты, которые нам понадобятся здесь для подготовки данных и тонкой настройки модели DistilBERT:

```
>>> import gzip
>>> import shutil
>>> import time

>>> import pandas as pd
>>> import requests
>>> import torch
>>> import torch.nn.functional as F
>>> import torchtext

>>> import transformers
>>> from transformers import DistilBertTokenizerFast
>>> from transformers import DistilBertForSequenceClassification
```

Далее мы задаем некоторые общие настройки — в том числе: количество эпох, на которых мы обучаем сеть, спецификацию устройства и случайное начальное число. Для воспроизводимости результатов обязательно используйте конкретное случайное начальное число — например, 123:

```
>>> torch.backends.cudnn.deterministic = True
>>> RANDOM_SEED = 123
>>> torch.manual_seed(RANDOM_SEED)
>>> DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

>>> NUM_EPOCHS = 3
```

Как уже отмечалось, мы воспользуемся набором данных обзора фильмов IMDb, с которым уже работали в главах 8 и 15. Следующий код извлекает сжатый набор данных и распаковывает его:

```
>>> url = ("https://github.com/rasbt/"
...         "machine-learning-book/raw/"
...         "main/ch08/movie_data.csv.gz")
>>> filename = url.split("/")[-1]
>>> with open(filename, "wb") as f:
...     r = requests.get(url)
...     f.write(r.content)
>>> with gzip.open('movie_data.csv.gz', 'rb') as f_in:
...     with open('movie_data.csv', 'wb') as f_out:
...         shutil.copyfileobj(f_in, f_out)
```

Если у вас на жестком диске остался файл `movie_data.csv` из главы 8, вы можете пропустить эту процедуру загрузки и распаковки.

Затем мы загружаем данные в DataFrame pandas и проверяем, все ли в порядке (вывод показан на рис. 16.17):

```
>>> df = pd.read_csv('movie_data.csv')
>>> df.head(3)
```

	review	sentiment
0	In 1974, the teenager Martha Moxley (Maggie Gr...	1
1	OK... so... I really like Kris Kristofferson a...	0
2	***SPOILER*** Do not read this, if you think a...	0

Рис. 16.17. Первые три строки набора данных обзора фильмов IMDb

На следующем шаге производится разделение набора данных на отдельные сегменты: для обучения, валидации и тестирования. Здесь мы используем 70% обзоров для обучения, 10% — для валидации и оставшиеся 20% — для тестирования:

```
>>> train_texts = df.iloc[:35000]['review'].values
>>> train_labels = df.iloc[:35000]['sentiment'].values

>>> valid_texts = df.iloc[35000:40000]['review'].values
>>> valid_labels = df.iloc[35000:40000]['sentiment'].values

>>> test_texts = df.iloc[40000:]['review'].values
>>> test_labels = df.iloc[40000:]['sentiment'].values
```

16.5.2. Токенизация набора данных

К этому моменту у нас есть тексты и метки для обучающего, валидационного и тестового наборов. На следующем шаге мы разобьем тексты на отдельные токены слов, используя реализацию токенизатора, унаследованную от класса предварительно обученной модели:

```
>>> tokenizer = DistilBertTokenizerFast.from_pretrained(
...     'distilbert-base-uncased'
... )

>>> train_encodings = tokenizer(list(train_texts), truncation=True, padding=True)
>>> valid_encodings = tokenizer(list(valid_texts), truncation=True, padding=True)
>>> test_encodings = tokenizer(list(test_texts), truncation=True, padding=True)
```



Выбор разных токенизаторов

Если вы хотите испытать различные типы токенизаторов, попробуйте воспользоваться пакетом токенизаторов (<https://huggingface.co/docs/tokenizers/python/latest/>), который также создан и поддерживается Hugging Face. Однако унаследованные токенизаторы обеспечивают согласованность между предварительно обученной моделью и набором данных, что избавляет нас от дополнительных усилий по поиску конкретного токенизатора, соответствующего модели. Другими словами, использование унаследованного токенизатора является рекомендуемым подходом, если вы хотите точно настроить предварительно обученную модель.

Наконец, загрузим все данные в класс `IMDbDataset` и создадим соответствующие загрузчики данных. Такой самоопределяемый класс набора данных позволяет нам настраивать все связанные признаки и функции для нашего пользовательского набора данных обзора фильмов в формате `DataFrame`:

```
>>> class IMDbDataset(torch.utils.data.Dataset):
...     def __init__(self, encodings, labels):
...         self.encodings = encodings
...         self.labels = labels
...
...     def __getitem__(self, idx):
...         item = {key: torch.tensor(val[idx])
...                 for key, val in self.encodings.items()}
...         item['labels'] = torch.tensor(self.labels[idx])
...         return item
...
...     def __len__(self):
...         return len(self.labels)

>>> train_dataset = IMDbDataset(train_encodings, train_labels)
>>> valid_dataset = IMDbDataset(valid_encodings, valid_labels)
>>> test_dataset = IMDbDataset(test_encodings, test_labels)

>>> train_loader = torch.utils.data.DataLoader(
...     train_dataset, batch_size=16, shuffle=True)
>>> valid_loader = torch.utils.data.DataLoader(
...     valid_dataset, batch_size=16, shuffle=False)
>>> test_loader = torch.utils.data.DataLoader(
...     test_dataset, batch_size=16, shuffle=False)
```

Хотя в целом код загрузчика данных должен быть вам знаком из предыдущих глав, стоит отметить одну деталь — переменную `item` в методе `__getitem__`. Кодировки, кото-

рые мы создали ранее, хранят много информации о токенизованных текстах. С помощью свертывания словаря, которое мы используем для присвоения словаря переменной `item`, мы извлекаем только наиболее важную информацию. Например, результирующие словарные записи содержат `input_ids` (уникальные целые числа из словаря, соответствующие токенам), `labels` (метки класса) и `attention_mask`. Здесь `attention_mask` — это тензор с двоичными значениями (0 и 1), обозначающий, какие токены должна учитывать модель. В частности, нули соответствуют токенам, используемым для заполнения последовательности до равной длины, и игнорируются моделью, а единицы соответствуют фактическим текстовым токенам.

16.5.3. Загрузка и тонкая настройка предварительно обученной модели BERT

Итак, мы позаботились о подготовке данных, и сейчас займемся загрузкой предварительно обученной модели DistilBERT и настройкой ее с помощью только что созданного набора данных. Код для загрузки предварительно обученной модели выглядит следующим образом:

```
>>> model = DistilBertForSequenceClassification.from_pretrained(  
...      'distilbert-base-uncased')  
>>> model.to(DEVICE)  
>>> model.train()  
  
>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)
```

Фрагмент `DistilBertForSequenceClassification` указывает на нижестоящую задачу, для которой мы хотим точно настроить модель, — в нашем случае это классификация последовательностей. Как упоминалось ранее, '`distilbert-base-uncased`' — это облегченная версия базовой регистрационезависимой модели BERT с приемлемым размером и хорошей производительностью. *Регистрационезависимость* означает, что модель не различает прописные и строчные буквы.



Использование других предварительно обученных трансформеров

Пакет `transformers` также содержит множество других предварительно обученных моделей и различные задачи для тонкой настройки. Ознакомьтесь с их перечнем на странице: <https://huggingface.co/transformers/>.

Теперь пришло время обучить модель. Мы можем разбить обучение на две части. Сначала нужно определить функцию точности для оценки производительности модели. Обратите внимание, что эта функция точности вычисляет обычную точность классификации. Почему мы заостряем на этом внимание? Здесь мы загружаем набор данных партиями, чтобы обойти ограничения ОЗУ или памяти графического процессора (VRAM) при работе с большой моделью глубокого обучения:

```
>>> def compute_accuracy(model, data_loader, device):  
...     with torch.no_grad():  
...         correct_pred, num_examples = 0, 0  
...         for batch_idx, batch in enumerate(data_loader):  
...             ### Подготовка данных  
             input_ids = batch['input_ids'].to(device)
```

```

...
    attention_mask = batch['attention_mask'].to(device)
    labels = batch['labels'].to(device)

...
    outputs = model(input_ids, attention_mask=attention_mask)
    logits = outputs['logits']
    predicted_labels = torch.argmax(logits, 1)
    num_examples += labels.size(0)
    correct_pred += (predicted_labels == labels).sum()
...
    return correct_pred.float() / num_examples * 100

```

В функции `compute_accuracy` мы загружаем текущий пакет, а затем получаем предсказанные метки из выходных данных. При этом мы отслеживаем общее количество примеров через `num_examples`. Точно так же мы отслеживаем количество правильных прогнозов с помощью переменной `correct_pred`. Наконец, после прохода по всему набору данных мы вычисляем точность как долю правильно предсказанных меток.

В принципе, с помощью функции `compute_accuracy` вы уже можете получить представление о том, как использовать модель трансформера для получения меток классов. То есть мы передаем модели `input_ids` вместе с информацией `attention_mask`, которая здесь указывает, с чем мы имеем дело: с фактическим текстовым токеном или токеном для заполнения последовательностей до равной длины. Затем вызов `model` возвращает выходные данные, представляющие собой объект трансформера `SequenceClassifierOutput`, специфический для этой библиотеки. Затем из этого объекта мы получаем логиты, которые преобразуем в метки классов с помощью функции `argmax`, аналогично тому, как это делали в предыдущих главах.

Наконец, перейдем к основной части: циклу обучения (точнее, тонкой настройки). Как вы заметили, тонкая настройка модели из библиотеки `transformers` очень похожа на обучение модели в чистом коде PyTorch с нуля:

```

>>> start_time = time.time()

>>> for epoch in range(NUM_EPOCHS):

...
    model.train()

...
    for batch_idx, batch in enumerate(train_loader):

...
        ### Подготовка данных
        input_ids = batch['input_ids'].to(DEVICE)
        attention_mask = batch['attention_mask'].to(DEVICE)
        labels = batch['labels'].to(DEVICE)

...
        ### Прямой проход
        outputs = model(input_ids,
                        attention_mask=attention_mask,
                        labels=labels)
        loss, logits = outputs['loss'], outputs['logits']

...
        ### Обратный проход
        optim.zero_grad()

```

```
...         loss.backward()
...
...         optim.step()

...
    ### Журналирование
    if not batch_idx % 250:
        print(f'Эпоха: {epoch+1:04d}/{NUM_EPOCHS:04d}'
              f' | Пакет'
              f'(batch_idx:04d)/'
              f'(len(train_loader):04d) | '
              f'Потери: {loss:.4f}'

...
    model.eval()

...
    with torch.set_grad_enabled(False):
        print(f'Точность при обучении: '
              f'{compute_accuracy(model, train_loader, DEVICE):.2f}%'
              f'\nТочность при валидации: '
              f'{compute_accuracy(model, valid_loader, DEVICE):.2f}%')

...
    print(f'Времени прошло: {(time.time() - start_time)/60:.2f} min')

...
    print(f'Общее время обучения: {(time.time() - start_time)/60:.2f} min')
    print(f'Точность при тестировании: {compute_accuracy(model, test_loader, DEVICE):.2f}%')
```

Вывод, созданный этим кодом, приведен далее (учтите, что код не является полностью детерминированным, поэтому результаты, которые вы получите, могут немного отличаться):

```
Эпоха: 0001/0003 | Пакет 0000/2188 | Потери: 0.6771
Эпоха: 0001/0003 | Пакет 0250/2188 | Потери: 0.3006
Эпоха: 0001/0003 | Пакет 0500/2188 | Потери: 0.3678
Эпоха: 0001/0003 | Пакет 0750/2188 | Потери: 0.1487
Эпоха: 0001/0003 | Пакет 1000/2188 | Потери: 0.6674
Эпоха: 0001/0003 | Пакет 1250/2188 | Потери: 0.3264
Эпоха: 0001/0003 | Пакет 1500/2188 | Потери: 0.4358
Эпоха: 0001/0003 | Пакет 1750/2188 | Потери: 0.2579
Эпоха: 0001/0003 | Пакет 2000/2188 | Потери: 0.2474
Точность при обучении: 96.32%
Точность при валидации: 92.34%
Времени прошло: 20.67 min
Эпоха: 0002/0003 | Пакет 0000/2188 | Потери: 0.0850
Эпоха: 0002/0003 | Пакет 0250/2188 | Потери: 0.3433
Эпоха: 0002/0003 | Пакет 0500/2188 | Потери: 0.0793
Эпоха: 0002/0003 | Пакет 0750/2188 | Потери: 0.0061
Эпоха: 0002/0003 | Пакет 1000/2188 | Потери: 0.1536
Эпоха: 0002/0003 | Пакет 1250/2188 | Потери: 0.0816
Эпоха: 0002/0003 | Пакет 1500/2188 | Потери: 0.0786
Эпоха: 0002/0003 | Пакет 1750/2188 | Потери: 0.1395
Эпоха: 0002/0003 | Пакет 2000/2188 | Потери: 0.0344
```

```
Точность при обучении: 98.35%
Точность при валидации: 92.46%
Времени прошло: 41.41 min
Эпоха: 0003/0003 | Пакет 0000/2188 | Потери: 0.0403
Эпоха: 0003/0003 | Пакет 0250/2188 | Потери: 0.0036
Эпоха: 0003/0003 | Пакет 0500/2188 | Потери: 0.0156
Эпоха: 0003/0003 | Пакет 0750/2188 | Потери: 0.0114
Эпоха: 0003/0003 | Пакет 1000/2188 | Потери: 0.1227
Эпоха: 0003/0003 | Пакет 1250/2188 | Потери: 0.0125
Эпоха: 0003/0003 | Пакет 1500/2188 | Потери: 0.0074
Эпоха: 0003/0003 | Пакет 1750/2188 | Потери: 0.0202
Эпоха: 0003/0003 | Пакет 2000/2188 | Потери: 0.0746
Точность при обучении: 99.08%
Точность при валидации: 91.84%
Времени прошло: 62.15 min
Общее время обучения: 62.15 min
Точность при тестировании: 92.50%
```

В этом коде мы перебираем несколько эпох. В каждой эпохе выполняются следующие шаги:

1. Загрузка ввода в устройство, на котором выполняются вычисления (GPU или CPU).
2. Вычисление вывода и потери модели.
3. Подгонка весовых параметров путем обратного распространения потерь.
4. Оценка производительности модели как на обучающем, так и на валидационном наборе.

Продолжительность обучения зависит от вычислительной платформы. После трех эпох точность на тестовом наборе данных достигает примерно 93%, что является существенным улучшением по сравнению с 85-процентной точностью, достигнутой RNN в [главе 15](#).

16.5.4. Удобная тонкая настройка трансформера с помощью API Trainer

В предыдущем подразделе мы вручную реализовали цикл обучения в PyTorch, чтобы проиллюстрировать, что тонкая настройка модели трансформера на самом деле не сильно отличается от обучения модели RNN или CNN с нуля. Однако надо отметить, что библиотека transformers содержит несколько удобных дополнительных инструментов, и в частности Trainer API, который мы здесь рассмотрим.

API Trainer, предоставляемый Hugging Face, оптимизирован для моделей-трансформеров и оснащен широким спектром вариантов обучения и различными встроенными функциями. При использовании этого API мы можем отказаться от самостоятельного написания обучающих циклов, а обучение или тонкая настройка модели преобразователя так же просты, как вызов функции (или метода). Давайте посмотрим, как это работает на практике.

Загрузим предварительно обученную модель:

```
>>> model = DistilBertForSequenceClassification.from_pretrained(  
...     'distilbert-base-uncased')  
>>> model.to(DEVICE)  
>>> model.train();
```

Цикл обучения из предыдущего подраздела можно заменить следующим кодом:

```
>>> optim = torch.optim.Adam(model.parameters(), lr=5e-5)  
  
>>> from transformers import Trainer, TrainingArguments  
>>> training_args = TrainingArguments(  
...     output_dir='./results',  
...     num_train_epochs=3,  
...     per_device_train_batch_size=16,  
...     per_device_eval_batch_size=16,  
...     logging_dir='./logs',  
...     logging_steps=10,  
... )  
  
>>> trainer = Trainer(  
...     model=model,  
...     args=training_args,  
...     train_dataset=train_dataset,  
...     optimizers=(optim, None) # optim and learning rate scheduler  
... )
```

Здесь мы сначала определили аргументы обучения, которые достаточно очевидны и относятся к расположению входных и выходных файлов, количеству эпох и размеру пакетов. Мы постарались сделать настройки максимально простыми, однако доступно множество дополнительных настроек, и мы рекомендуем ознакомиться с информацией о них на странице описания `TrainingArguments`: https://huggingface.co/transformers/main_classes/trainer.html#trainingarguments.

Затем мы передали эти настройки `TrainingArguments` классу `Trainer` для создания экземпляра нового объекта `trainer`. После инициализации `trainer` с настройками, моделью, которая нуждается в тонкой настройке, и наборами для обучения и оценки, мы можем обучить модель, вызвав метод `trainer.train()` (мы будем использовать этот метод в дальнейшем). Вот и все — задействовать API `Trainer` так же просто, что и показано в приведенном коде, и никакого дополнительного рутинного кода не требуется.

Однако вы, возможно, заметили, что в этих фрагментах кода тестовый набор данных не использовался и мы не указали какие-либо оценочные показатели. Это связано с тем, что API `Trainer` показывает только потери при обучении и по умолчанию не обеспечивает оценку модели в процессе обучения. Есть два способа отобразить окончательную производительность модели, которые мы продемонстрируем далее.

Первый метод оценки окончательной модели заключается в определении функции оценки в качестве аргумента `compute_metrics` для другого экземпляра `Trainer`. Функция `compute_metrics` работает с тестовыми прогнозами моделей в виде логитов (которые

являются выходными данными модели по умолчанию) и тестовыми метками. Чтобы создать экземпляр этой функции, мы рекомендуем установить библиотеку datasets от Hugging Face с помощью команды `pip install datasets` и использовать ее следующим образом:

```
>>> from datasets import load_metric
>>> import numpy as np
>>> metric = load_metric("accuracy")
>>> def compute_metrics(eval_pred):
...     logits, labels = eval_pred
...     # важно: logits это массив numpy, а не тензор pytorch
...     predictions = np.argmax(logits, axis=-1)
...     return metric.compute(
...         predictions=predictions, references=labels)
```

Обновленный экземпляр Trainer (теперь включающий в себя `compute_metrics`) выглядит так:

```
>>> trainer=Trainer(
...     model=model,
...     args=training_args,
...     train_dataset=train_dataset,
...     eval_dataset=test_dataset,
...     compute_metrics=compute_metrics,
...     optimizers=(optim, None) # optim and learning rate scheduler
... )
```

Теперь обучим модель (опять же, отметим, что код не полностью детерминирован, поэтому вы можете получить немного другие результаты):

```
>>> start_time = time.time()
>>> trainer.train()

***** Запуск обучения *****
Num examples = 35000
Num Epochs = 3
Размер пакета на устройство = 16
Общий размер обучающего пакета (параллель, распределение и накопление) = 16
Шагов накопления градиента = 1
Всего шагов оптимизации = 6564
```

Шаг Потери при обучении

10	0.705800
20	0.684100
30	0.681500
40	0.591600
50	0.328600
60	0.478300
...	

```
>>> print(f'Полное время обучения: '
...       f'{(time.time() - start_time)/60:.2f} min')
Полное время обучения: 45.36 min
```

После завершения обучения, которое может занять до часа в зависимости от вашего графического процессора, мы можем вызвать `trainer.evaluate()`, чтобы получить производительность модели на тестовом наборе:

```
>>> print(trainer.evaluate())

***** Запуск оценки *****
Кол-во примеров = 10000
Размер пакета = 16
100% |██████████| 625/625 [10:59<00:00, 1.06s/
it]
{'eval_loss': 0.30534815788269043,
 'eval_accuracy': 0.9327,
 'eval_runtime': 87.1161,
 'eval_samples_per_second': 114.789,
 'eval_steps_per_second': 7.174,
 'epoch': 3.0}
```

Как можно видеть, оценочная точность модели составляет около 94%, что аналогично разработанному нами ранее собственному циклу обучения PyTorch. Здесь мы пропустили этап обучения, потому что модель уже прошла тонкую настройку после предыдущего вызова `trainer.train()`. Существует небольшое расхождение между нашим подходом к собственному коду и использованием класса Trainer, потому что класс Trainer применяет некоторые отличающиеся и дополнительные настройки.

Второй метод, который мы могли бы использовать для вычисления точности окончательной модели на тестовом наборе, — это повторное использование нашей функции `compute_accuracy`, определенной нами в предыдущем подразделе. Мы можем напрямую оценить производительность настроенной модели на тестовом наборе данных, выполнив следующий код:

```
>>> model.eval()
>>> model.to(DEVICE)

>>> print(f'Точность при тестировании: {compute_accuracy(model, test_loader,
DEVICE):.2f}%)')

Точность при тестировании: 93.27%
```

На самом деле если вы хотите регулярно проверять производительность модели во время обучения, то можете потребовать от класса Trainer выводить оценку модели после каждой эпохи, определив аргументы обучения следующим образом:

```
>>> from transformers import TrainingArguments

>>> training_args = TrainingArguments("test_trainer",
... evaluation_strategy="epoch", ...)
```

Однако, если вы планируете изменить или оптимизировать гиперпараметры и повторить процедуру тонкой настройки несколько раз, мы рекомендуем использовать для этой цели валидационный набор, чтобы сохранить независимость тестового набора. Добиться этого можно, создав экземпляр Trainer с помощью `valid_dataset`:

```
>>> trainer=Trainer(  
...     model=model,  
...     args=training_args,  
...     train_dataset=train_dataset,  
...     eval_dataset=valid_dataset,  
...     compute_metrics=compute_metrics,  
... )
```

Итак, в этом разделе вы ознакомились с процессом тонкой настройки модели BERT для классификации. Этот подход отличается от использования других архитектур глубокого обучения, таких как RNN, которые мы обычно обучаем с нуля. Однако, если вы не занимаетесь научными исследованиями и не пытаетесь разработать новую архитектуру модели (что очень дорого), предварительное обучение трансформеров не требуется. Поскольку модели-трансформеры обучаются на общих неразмеченных наборах данных, их самостоятельное предварительное обучение может оказаться нецелесообразным с точки зрения использования времени и ресурсов, поэтому тонкая настройка — это как раз то, что нам нужно.

16.6. Заключение

В этой главе представлена совершенно новая архитектура модели для обработки естественного языка — Transformer. Эта архитектура основана на концепции, называемой *самовниманием*, и мы пришли к ней не сразу. Сначала мы рассмотрели RNN, снабженную механизмом внимания, улучшающим ее способность переводить длинные предложения. Затем мы постепенно ввели понятие самовнимания и объяснили, как оно используется в модуле внимания с несколькими головами внутри трансформера.

С момента публикации статьи об оригинальной архитектуре Transformer в 2017 году появилось и развивалось множество различных производных этой архитектуры, которые теперь принято называть просто *трансформерами*. В этой главе мы сосредоточились на некоторых самых популярных вариантах — семействе моделей GPT, BERT и BART. GPT — это односторонняя модель, которая особенно хороша для создания нового текста. BERT использует двунаправленный подход, который лучше подходит для других типов задач — например, для классификации. Наконец, BART сочетает в себе двунаправленный кодировщик от BERT и односторонний декодер от GPT. Заинтересованные читатели могут узнать о других вариантах трансформеров из следующих двух обзорных статей:

- ◆ «Pre-trained Models for Natural Language Processing», A Survey by Qiu and colleagues, 2020. Доступна по адресу: <https://arxiv.org/abs/2003.08271>;
- ◆ «AMMUS: A Survey of Transformer-based Pretrained Models in Natural Language Processing» by Kayan and colleagues, 2021. Доступна по адресу: <https://arxiv.org/abs/2108.05542>.

Трансформеры, как правило, более требовательны к данным, чем RNN, и требуют больших объемов данных для предварительного обучения. На этапе предварительного обучения используются большие объемы неразмеченных данных для построения общей языковой модели, которую затем можно адаптировать для конкретных задач путем тонкой настройки на небольших размеченных наборах данных.

Чтобы увидеть, как это работает на практике, мы загрузили предварительно обученную модель BERT из библиотеки `transformers` от Hugging Face и выполнили ее тонкую настройку для задачи классификации эмоциональной окраски текста в наборе данных обзора фильмов IMDb.

В следующей главе мы обсудим генеративно-состязательные сети. Как следует из их названия, *генеративно-состязательные сети* — это аналогичные моделям GPT, обсуждаемым нами в этой главе, модели, которые можно использовать для генерации новых данных. Однако сейчас мы оставляем позади тему моделирования естественного языка и рассмотрим генеративно-состязательные сети в контексте компьютерного зрения и генерации новых изображений — задачи, для которой эти сети изначально были разработаны.

17

Генеративно-состязательные сети и синтез новых данных

В предыдущей главе мы сосредоточились на оригинальной архитектуре Transformer и познакомились с некоторыми из наиболее значимых моделей NLP, основанных на ней. В этой главе мы откроем для себя *генеративно-состязательные сети* (Generative Adversarial Networks, GAN) и продемонстрируем их применение в синтезе новых выборок данных. GAN принято считать одним из самых важных прорывов в области глубокого обучения, потому что они позволяют компьютерам генерировать новые данные (например, новые изображения).

Здесь будут рассмотрены следующие темы:

- ◆ знакомство с генеративными моделями для синтеза новых данных;
- ◆ автокодировщики, вариационные автокодировщики и их связь с GAN;
- ◆ основные функциональные компоненты GAN;
- ◆ пример простой модели GAN для генерации рукописных цифр;
- ◆ транспонированная свертка и пакетная нормализация;
- ◆ улучшенные GAN с использованием глубокой свертки и расстояния Вассерштейна.

17.1. Знакомство с генеративно-состязательными сетями

В общем случае назначение GAN состоит в том, чтобы синтезировать новые данные, которые имеют то же распределение, что и набор обучающих данных. Следовательно, GAN в их исходной форме считаются задачами машинного обучения без учителя, поскольку им не требуются маркованные данные. Однако стоит отметить, что расширенные варианты исходной GAN могут относиться к задачам частичного или полного обучения с учителем.

Общая концепция GAN впервые была предложена в 2014 г. Яном Гудфеллоу и его коллегами в качестве метода синтеза новых изображений с использованием глубоких нейронных сетей¹. Хотя первоначальная архитектура GAN, представленная в этой статье,

¹ См. «Generative Adversarial Nets, in Advances in Neural Information Processing Systems» by I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, p. 2672–2680, 2014.

была основана на полностью связанных слоях — аналогично многоуровневой архитектуре персептрона, и была обучена генерировать рукописные цифры с низким разрешением, подобные MNIST, она служила скорее для демонстрации осуществимости новой идеи.

Однако позднее авторы оригинальной модели, а также другие исследователи предложили многочисленные ее улучшения и способы применения в разных областях науки и техники — например, в компьютерном зрении GAN используют для преобразования изображения в изображение (обучение тому, как преобразовать входное изображение в выходное изображение), повышения разрешения изображения (создание изображения с высоким разрешением из его версии с низким разрешением), раскрашивания изображений (обучение восстановлению отсутствующих частей изображения) и многих других подобных задач. Так, последние достижения в исследованиях GAN позволили создать модели, способные генерировать новые изображения лиц с высоким разрешением. Примеры таких изображений с высоким разрешением можно найти на сайте <https://www.thispersondoesnotexist.com/>, где представлены синтетические изображения лиц, сгенерированные GAN.

17.1.1. Начнем с автокодировщиков

Прежде чем обсуждать принцип работы GAN, необходимо разобраться, как работают *автокодировщики* (autoencoder), которые могут сжимать и распаковывать обучающие данные. Хотя обычные автокодировщики не могут генерировать новые данные, понимание их устройства поможет вам быстрее освоить GAN, о которой пойдет речь в следующем разделе.

Автокодировщики состоят из двух объединенных сетей: *кодировщика* и *декодера*. Сеть кодировщика получает d -мерный вектор входных признаков, ассоциированный с примером x (т. е. $x \in R^d$), и кодирует его в p -мерный вектор z (т. е. $z \in R^p$). Другими словами, задача кодировщика состоит в том, чтобы научиться моделировать функцию $z = f(x)$. Закодированный вектор z также называется *скрытым вектором*, или *представлением скрытых признаков*. Как правило, размерность скрытого вектора меньше, чем у входных примеров, — другими словами: $p < d$. Следовательно, можно сказать, что кодировщик действует как функция сжатия данных. Затем декодер распаковывает \hat{x} из скрытого вектора меньшей размерности z , где мы можем рассматривать декодер как функцию $\hat{x} = g(z)$. Архитектура простейшего автокодировщика показана на рис. 17.1, где кодировщик и декодер состоят только из одного полносвязного слоя каждый.



Связь между автокодировщиком и уменьшением размерности

В главе 5 вы узнали о методах уменьшения размерности — таких как *метод главных компонент* (PCA) и *линейный дискриминантный анализ* (LDA). Автокодировщики тоже можно использовать для уменьшения размерности. Фактически, когда в подсетях кодировщика и декодера нет нелинейности, применение автокодировщика почти идентично методу PCA.

В этом случае, если предположить, что веса однослойного кодировщика (без скрытого слоя и без нелинейной функции активации) обозначены матрицей U , то модели кодировщика можно представить как $z = U^T x$. Точно так же однослойный линейный декодер моделирует $\hat{x} = U z$. Объединяя эти два компонента, мы получаем $\hat{x} = UU^T z$. Это именно то, что делает PCA, разве что PCA имеет дополнительное ортонормированное ограничение: $UU^T = I_{n \times n}$.

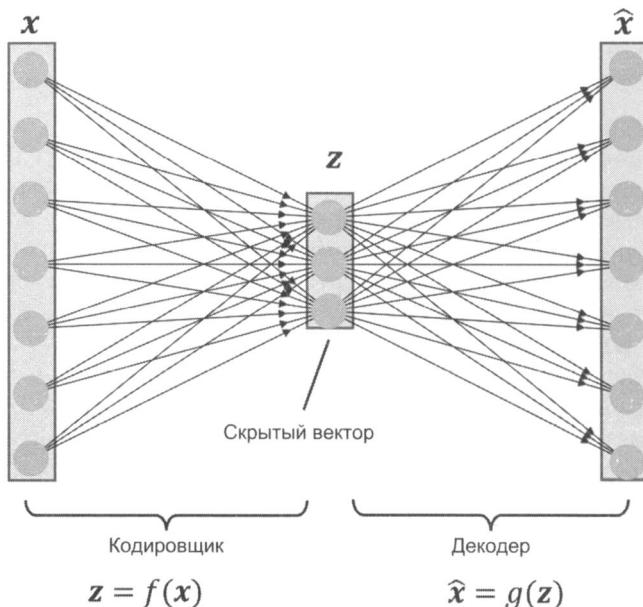


Рис. 17.1. Архитектура простейшего автокодировщика

Хотя на рис. 17.1 показан автокодировщик без скрытых слоев внутри кодировщика и декодера, мы, конечно, можем добавить несколько скрытых слоев с нелинейностями (как в многослойной нейросети), чтобы создать глубокий автокодировщик, способный обучаться более эффективным функциям сжатия и восстановления данных. Также нужно заметить, что автокодировщик, представленный в этом разделе, использует полносвязные слои. Однако когда мы работаем с изображениями, мы можем заменить полносвязные слои сверточными, как было показано в главе 14.



Другие типы автокодировщиков в зависимости от размера скрытого пространства

Как упоминалось ранее, размерность скрытого пространства автокодировщика обычно ниже, чем размерность входных данных ($p < d$), что позволяет использовать автокодировщики для уменьшения размерности. По этой причине скрытый вектор также часто считают «узким местом», а эту конкретную конфигурацию автокодировщика также называют *недополненной* (undercomplete), или *сжатой*. Однако существует другая категория автокодировщиков, называемых *сверхполными* (overcomplete), где размерность скрытого вектора z больше, чем размерность входных примеров ($p > d$).

При обучении сверхполного автокодировщика существует тривиальное решение, когда кодировщик и декодер могут просто научиться копировать (запоминать) входные признаки в свой выходной слой. Очевидно, что это решение не очень полезно. Однако с некоторыми изменениями в процедуре обучения сверхполные автокодировщики можно применять для *подавления шума*.

В этом случае во время обучения к входным примерам добавляется случайный шум ϵ , и сеть учится восстанавливать чистый пример x из зашумленного сигнала $x + \epsilon$. Затем в рабочем режиме мы предоставляем новые примеры, которые являются

ся естественными шумами (т. е. шум уже присутствует, так что дополнительный искусственный шум ϵ не добавляется), и модель должна удалить существующий шум из этих примеров. Эта архитектура автокодировщика в сочетании с методом обучения называется *шумоподавляющим автокодировщиком*.

Если вы хотите узнать больше об этой теме, рекомендуем прочесть исследовательскую статью «Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion» by Pascal Vincent and colleagues, 2010, <http://www.jmlr.org/papers/v11/vincent10a.html>.

17.1.2. Генеративные модели для синтеза новых данных

Автокодировщик — это *детерминированная* модель, а это означает, что после обучения при заданных входных данных x она сможет восстановить входные данные из сжатой версии в пространстве меньшей размерности. Следовательно, она не может генерировать новые данные и способна только восстанавливать входные данные путем преобразования сжатого представления.

С другой стороны, генеративная модель может сгенерировать *новый* пример \tilde{x} из случайного вектора z (соответствующего скрытому представлению). Схематическое представление генеративной модели показано на рис. 17.2. Случайный вектор z берется из распределения с полностью известными характеристиками, поэтому мы можем легко сделать выборку из такого распределения. Например, каждый элемент z может быть выбран из равномерного распределения в диапазоне $[-1, 1]$ (мы можем записать это так: $z_i \sim \text{Uniform}(-1, 1)$) или из стандартного нормального распределения (в этом случае мы пишем: $z_i \sim \text{Normal}(\mu = 0, \sigma^2 = 1)$).

Поскольку мы переключили внимание с автокодировщиков на генеративные модели, вы, возможно, заметили, что декодирующая часть автокодировщика имеет некоторое

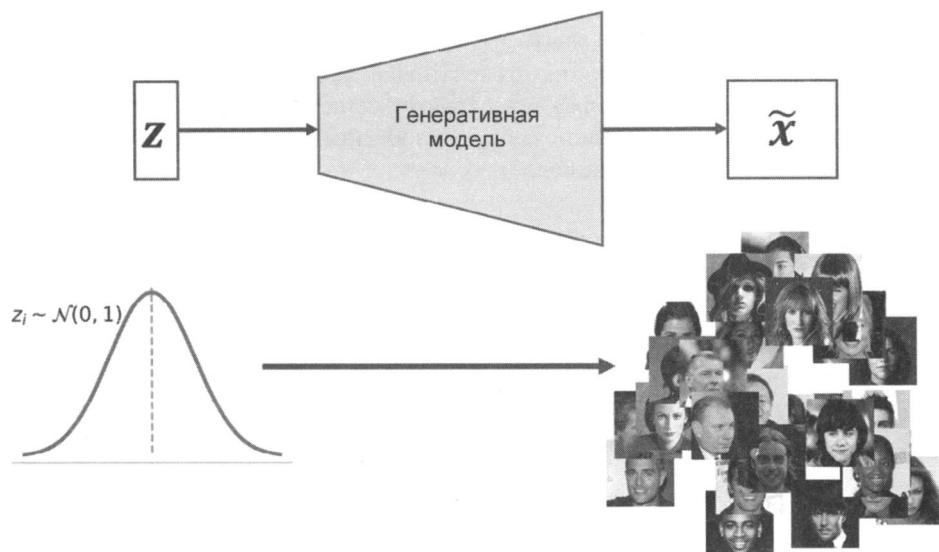


Рис. 17.2. Генеративная модель

сходство с генеративной моделью. В частности, они обе получают скрытый вектор z в качестве входных данных и возвращают выходные данные в том же пространстве, что и x . (Для автокодировщика \hat{x} — это восстановленный ввод x , а для генеративной модели \hat{x} — это синтезированный экземпляр.)

Однако основное различие между ними заключается в том, что мы не знаем распределения z в автокодировщике, в то время как в генеративной модели распределение z полностью характеризуемо. Тем не менее автокодировщик можно обобщить в генеративную модель — одним из подходов такого плана является *вариационный автокодировщик* (Variational Autoencoder, VAE).

В VAE, получающем входной пример x , сеть кодировщика модифицирована таким образом, что она вычисляет два момента распределения скрытого вектора: среднее значение μ и дисперсию σ^2 . Во время обучения VAE сеть вынуждена сопоставлять эти моменты с моментами стандартного нормального распределения (т. е. с нулевым средним и единичной дисперсией). Затем, после обучения модели VAE, кодировщик отбрасывается, и мы можем использовать декодер для создания новых экземпляров \hat{x} путем подачи случайных векторов z из «выученного» распределения Гаусса.

Помимо VAE, существуют и другие типы генеративных моделей — например, *авторегрессионные модели* (autoregressive model) и *нормализующие потоковые модели* (normalizing flow models). Однако в этой главе мы ограничимся только моделями GAN, которые представляют собой один из самых новых и самых популярных типов генеративных моделей в глубоком обучении.



Что такое генеративная модель?

Генеративные модели традиционно определяют как алгоритмы, которые моделируют распределения входных данных $p(x)$ или совместные распределения входных данных и связанных целей $p(x, y)$. По определению эти модели также способны производить выборку из некоторого признака x_i , обусловленного другим признаком x_j , что принято называть *условным выводом* (conditional inference). Однако в контексте глубокого обучения термин «генеративная модель» обычно используется для обозначения моделей, которые генерируют реалистично выглядящие данные. Это означает, что мы можем делать выборку из входных распределений $p(x)$, но не обязаны выполнять условный вывод.

17.1.3. Генерация новых экземпляров данных с помощью GAN

Чтобы кратко пояснить, что делают GAN, давайте сначала предположим, что у нас есть сеть, которая получает случайный вектор z , выбранный из известного распределения, и генерирует выходное изображение x . Мы будем называть эту сеть *генератором* (G) и использовать запись $\hat{x} = G(z)$ для обозначения сгенерированного вывода. Допустим, нашей целью является создание некоторых изображений — например, лиц, зданий, животных или даже рукописных цифр, таких как MNIST.

Как всегда, инициализируем эту сеть случайными весами. Следовательно, первые выходные изображения, пока эти веса не подвергнутся корректировке, будут выглядеть

как белый шум. Теперь представьте, как хорошо бы было, если бы мы обладали функцией, способной оценивать качество изображений (назовем ее функцией-оценщиком).

Если бы такая функция существовала, мы могли бы использовать обратную связь от этой функции, чтобы сообщить нашему генератору, как скорректировать его веса, чтобы улучшить качество сгенерированных изображений. Таким образом, мы могли бы — на основе обратной связи от функции-оценщика — обучить генератор, создавать выходные данные в виде реалистично выглядящих изображений.

Хотя упомянутая функция-оценщик очень упростила бы задачу создания изображений, вопрос заключается в том, существует ли такая универсальная функция для оценки качества изображения, и если да, то как она определяется. Очевидно, что мы, люди, можем легко оценить качество выходных изображений, просто разглядывая результат работы сети, но, увы, не в состоянии (пока) передать оценку из нашего мозга в сеть. Возникает вопрос: если наш мозг способен оценивать качество синтезированных изображений, можем ли мы разработать нейросетевую модель, которая будет делать то же самое? Фактически в этом и заключается самая общая идея GAN.

Как показано на рис. 17.3, модель GAN содержит дополнительную нейросеть, так называемый *дискриминатор* (D), который фактически представляет собой классификатор, обучающийся отличать синтезированное изображение \hat{x} от реального изображения x .

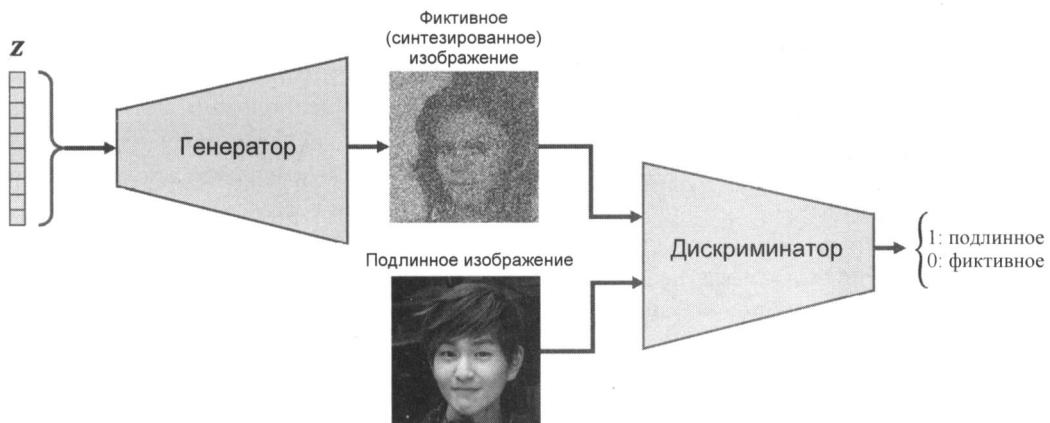


Рис. 17.3. Задача дискриминатора — отличить реальное изображение от изображения, созданного генератором

В модели GAN обе сети: генератор и дискриминатор — обучаются вместе. Сначала, после инициализации весов модели, генератор создает изображения, которые не выглядят реалистично. Точно так же дискриминатор плохо различает реальные и синтезированные изображения. Но со временем (т. е. посредством обучения) обе сети становятся лучше, поскольку взаимодействуют друг с другом. На самом деле две эти сети «играют» в состязательную игру, в которой генератор учится улучшать свои выходные данные, чтобы иметь возможность обмануть дискриминатор. В то же время дискриминатор учится лучше распознавать синтезированные изображения.

17.1.4. Функции потерь сетей генератора и дискриминатора в модели GAN

Целевая функция GAN, описанная в оригинальной статье², выглядит следующим образом:

$$V(\theta^{(D)}, \theta^{(G)}) = E_{x \sim p_{data}(x)} [\log D(x)] + E_{z \sim p_z(z)} [\log (1 - D(G(z)))].$$

Здесь $V(\theta^{(D)}, \theta^{(G)})$ называется *функцией ценности* (value function), которую можно рассматривать как компромисс: мы стремимся максимизировать ее значение по отношению к дискриминатору (D), минимизируя при этом ее значение по отношению к генератору (G), т. е. $\min_G \max_D V(\theta^{(D)}, \theta^{(G)})$. $D(x)$ — это вероятность, указывающая, является ли входной экземпляр x реальным или фиктивным (т. е. сгенерированным). Выражение $E_{x \sim p_{data}(x)} [\log D(x)]$ характеризует ожидаемое количественное значение величины в скобках по отношению к примерам из распределения данных (распределение реальных примеров), а выражение $E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$ характеризует ожидаемое количественное значение величины относительно распределения входных векторов z .

Один шаг обучения модели GAN с такой функцией ценности требует двух шагов оптимизации: (1) максимизация выигрыша для дискриминатора и (2) минимизация выигрыша для генератора. На практике обучение GAN заключается в чередовании этих двух шагов оптимизации: (1) зафиксировать (заморозить) параметры одной сети и оптимизировать веса другой, и (2) зафиксировать вторую сеть и оптимизировать первую. Этот процесс следует повторять на каждом проходе обучения. Предположим, что мы зафиксировали сеть генератора и намереваемся оптимизировать дискриминатор. Оба слагаемых в функции ценности $V(\theta^{(D)}, \theta^{(G)})$ вносят свой вклад в оптимизацию дискриминатора, где первое слагаемое соответствует потерям, связанным с реальными примерами, а второе слагаемое — потерям для сгенерированных (фиктивных) примеров. Поэтому, когда мы фиксируем G , наша цель состоит в том, чтобы *максимизировать* $V(\theta^{(D)}, \theta^{(G)})$, что соответствует улучшению способности дискриминатора различать реальные и сгенерированные изображения.

После оптимизации дискриминатора с использованием членов потерь для реальных и поддельных примеров мы фиксируем дискриминатор и начинаем оптимизировать генератор. В этом случае вклад в градиенты генератора дает только второй член в $V(\theta^{(D)}, \theta^{(G)})$. Следовательно, при фиксированном дискриминаторе наша цель состоит в том, чтобы *минимизировать* $V(\theta^{(D)}, \theta^{(G)})$, что можно записать так: $\min_G E_{z \sim p_z(z)} [\log (1 - D(G(z)))]$. Как сказано в упомянутой ранее статье про GAN, на

ранних этапах обучения функция $\log (1 - D(G(z)))$ страдает от исчезающего градиента. Причина этого в том, что выходные данные $G(z)$ в начале процесса обучения совсем не похожи на реальные примеры, и поэтому значение $D(G(z))$ с высокой достоверностью будет близко к нулю. Это явление называется *насыщением* (saturation). Чтобы ре-

² См. «Generative Adversarial Nets» by I. Goodfellow et al, <https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>.

шить эту проблему, мы можем переформулировать цель минимизации $\min_i E_{z \sim p_z(z)} [\log(1 - D(G(z)))]$, записав ее так: $\max_i E_{z \sim p_z(z)} [\log(D(G(z)))]$.

Эта замена означает, что для обучения генератора мы можем поменять местами метки реальных и поддельных примеров и провести обычную минимизацию функции. Другими словами, даже несмотря на то, что примеры, синтезированные генератором, являются поддельными и поэтому помечены 0, мы можем поменять местами метки, назначив этим примерам метку 1, и минимизировать потерю бинарной перекрестной энтропии с этими новыми метками вместо максимизации $\max_z E_{z \sim p_g(z)} [\log(D(G(z)))]$.

Теперь, когда мы рассмотрели в общих чертах процедуру оптимизации для обучения моделей GAN, давайте подумаем, какие метки данных мы можем использовать при обучении. Поскольку дискриминатор является бинарным классификатором (метки классов равны 0 и 1 для фиктивных и реальных изображений соответственно), мы можем использовать функцию потерь бинарной перекрестной энтропии. Следовательно, мы можем так определить эталонные метки для потерь дискриминатора:

$$\text{Эталонные метки для дискриминатора} = \begin{cases} 1: \text{ для реальных изображений, т. е. } x \\ 0: \text{ для выходов } G, \text{ т. е. } G(z) \end{cases}.$$

А что насчет меток для обучения генератора? Поскольку нам нужно, чтобы генератор синтезировал реалистичные изображения, мы должны наказывать генератор, когда его выходные данные дискриминатор не воспринимает как реальное изображение. Следовательно, мы станем предполагать, что при вычислении функции потерь для генератора эталонные метки для вывода генератора равны 1.

В качестве промежуточного итога на рис. 17.4 показаны отдельные этапы построения простой модели GAN.

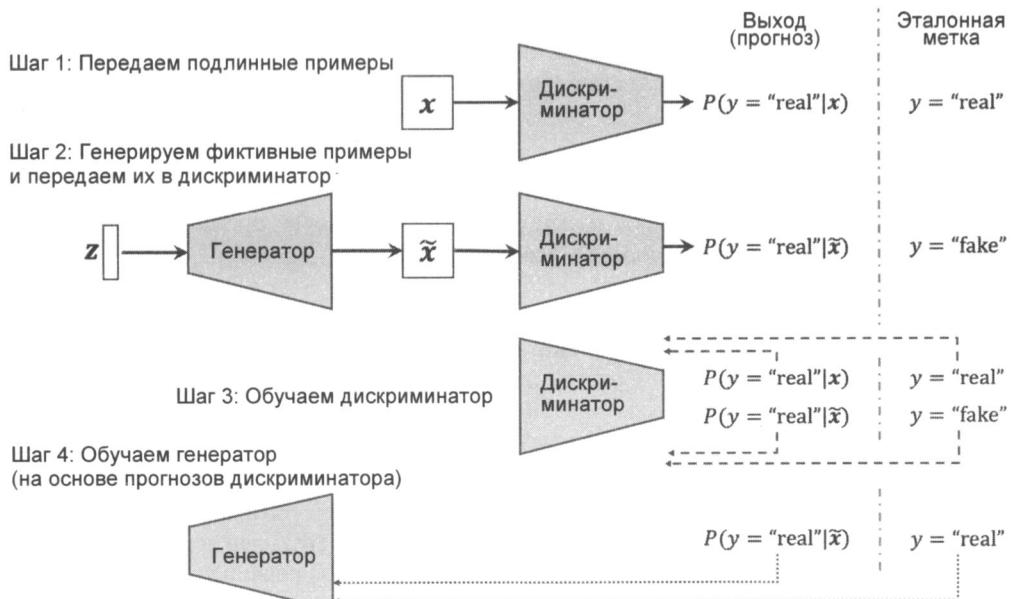


Рис. 17.4. Этапы построения модели GAN

В следующем разделе мы построим с нуля собственную GAN для генерации новых рукописных цифр.

17.2. Строим собственную GAN с нуля

В этом разделе мы расскажем, как реализовать и обучить модель GAN для генерации новых изображений — таких как цифры MNIST. Поскольку обучение модели на обычном компьютере может занять много времени, мы расскажем, как настроить среду облачных вычислений Google Colab, которая позволяет выполнять вычисления на графических процессорах (GPU).

17.2.1. Обучение моделей GAN в Google Colab

Для некоторых примеров кода в этой главе могут потребоваться значительные вычислительные ресурсы, которые выходят за рамки возможностей обычного ноутбука или настольного компьютера, не обладающего мощным графическим процессором. Впрочем, если у вас есть доступ к компьютеру с поддержкой графического процессора NVIDIA, и на нем установлены библиотеки CUDA и cuDNN, вы можете воспользоваться им для ускорения вычислений.

Тем не менее, поскольку многие из нас не имеют доступа к высокопроизводительным вычислительным ресурсам, мы воспользуемся средой Google Colaboratory (часто называемой Google Colab) — бесплатной службой облачных вычислений, доступной в большинстве стран.

Google Colab предоставляет экземпляры блокнотов Jupyter Notebook, работающие в облаке. Блокноты эти можно сохранить на Google Drive или в репозитории GitHub. Работая с различными вычислительными ресурсами платформы Google Colab — такими как процессоры, графические процессоры и даже блоки тензорной обработки (TPU), — важно помнить, что время выполнения задачи в настоящее время ограничено 12 часами. Следовательно, выполнение кода любого блокнота, продлившееся более 12 часов, будет прервано.

Выполнение примеров кода в этой главе занимает максимум два-три часа, так что для вас это не будет проблемой. Однако, если вы решите использовать Google Colab для других проектов, которые занимают более 12 часов машинного времени, обязательно сохраняйте промежуточные контрольные точки.



Jupyter Notebook

Jupyter Notebook — это графический пользовательский интерфейс (GUI) для интерактивного запуска кода и чередования его с текстовой документацией и рисунками. Благодаря своей универсальности и простоте использования он стал одним из самых популярных инструментов в науке о данных.

Дополнительные сведения об общем графическом интерфейсе Jupyter Notebook вы найдете в официальной документации по адресу: <https://jupyter-notebook.readthedocs.io/en/stable/>. Весь код этой книги также доступен в виде блокнотов Jupyter, а краткое введение можно найти в каталоге кодов первой главы.

Наконец, мы настоятельно рекомендуем прочитать статью Адама Рула и его коллег «Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks»

об эффективном использовании Jupyter Notebook в научно-исследовательских проектах, которая находится в свободном доступе по адресу: <https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007007>.

Доступ к Google Colab очень прост. Перейдите по адресу <https://colab.research.google.com>, после чего автоматически откроется окно подсказки (рис. 17.5), в котором вы сможете увидеть свои существующие блокноты Jupyter. В этом окне выберите вкладку **Google Drive** — здесь будут сохраняться ваши блокноты на вашем Google Drive.

Для создания нового блокнота нажмите ссылку **New notebook** (Новый блокнот) в нижней части окна подсказки.

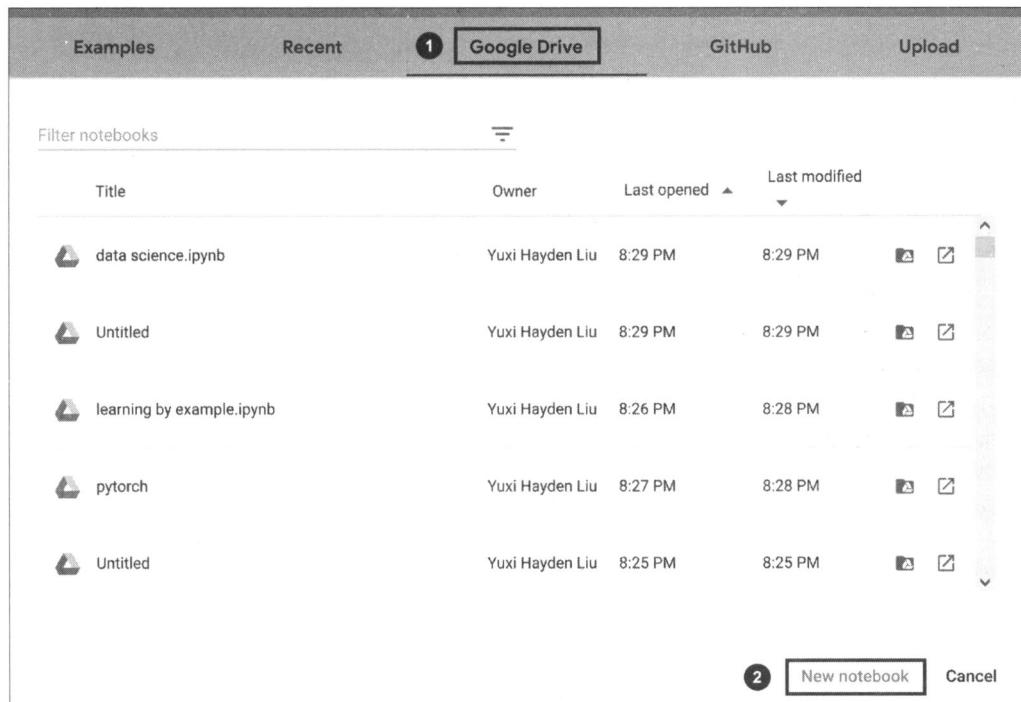


Рис. 17.5. Создание нового блокнота Python в Google Colab

Это действие создаст и откроет для вас новый блокнот. Все примеры кода, которые вы в нем введете, будут автоматически сохранены, и позже вы сможете получить доступ к блокноту в своей учетной записи Google Drive в каталоге **Colab Notebooks** (Блокноты Colab).

Для запуска примеров кода из этого блокнота мы будем использовать графические процессоры. Для этого в строке меню текущего блокнота в разделе **Runtime** (Среда выполнения) нажмите **Change runtime type** (Изменить тип среды выполнения) и выберите **GPU** (Графический процессор), как показано на рис. 17.6.

На последнем шаге нам просто нужно установить пакеты Python, которые понадобятся для реализации кода этой главы. Среда Colab Notebooks уже поставляется с некоторы-

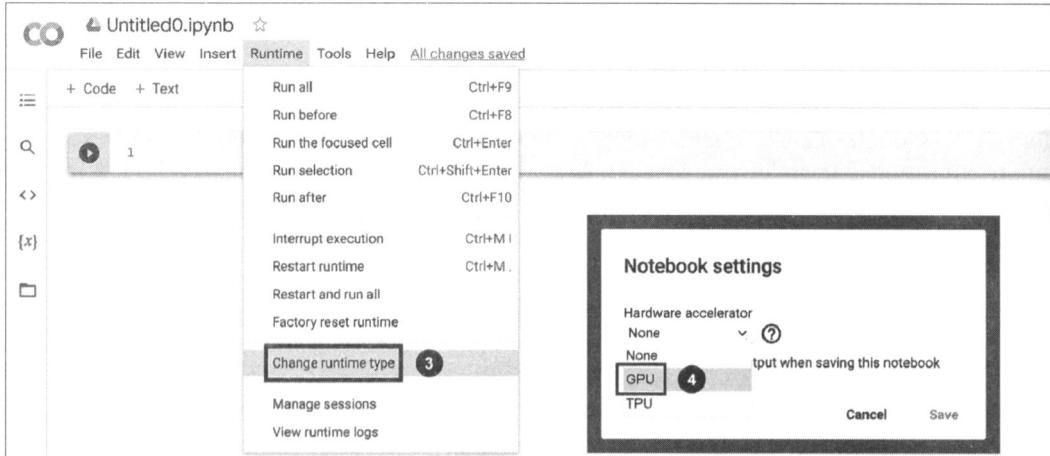


Рис. 17.6. Использование графических процессоров в Google Colab

ми пакетами — такими как NumPy, SciPy и последней стабильной версией PyTorch. На момент подготовки книги последней стабильной версией в Google Colab была PyTorch 1.9. Теперь мы можем протестировать рабочую среду и убедиться, что GPU доступен, используя следующий код:

```
>>> import torch
>>> print(torch.__version__)
1.9.0+cu111
>>> print("Доступность GPU:", torch.cuda.is_available())
Доступность GPU: True
>>> if torch.cuda.is_available():
...     device = torch.device("cuda:0")
... else:
...     device = "cpu"
>>> print(device)
cuda:0
```

Кроме того, если вы хотите сохранить модель на своем личном Google Drive, передать кому-то или загрузить другие файлы, вам необходимо подключить Google Drive. Для этого выполните в новой ячейке блокнота следующие команды:

```
>>> from google.colab import drive
>>> drive.mount('/content/drive/')
```

Эти команды предоставят ссылку для аутентификации блокнота Colab при доступе к вашему Google Drive. Следуя инструкциям по аутентификации, вам нужно скопировать код доступа и вставить его в назначенное поле ввода под ячейкой, которую вы только что выполнили. Ваш Google Drive будет смонтирован и доступен в /content/drive/My Drive. Кроме того, вы можете смонтировать его через графический интерфейс, как показано на рис. 17.7.

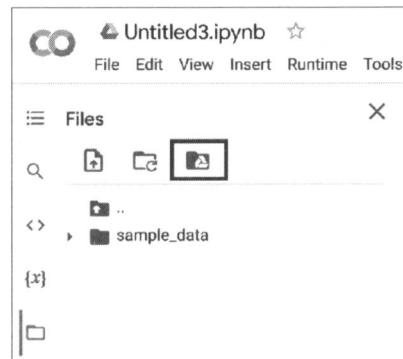


Рис. 17.7. Подключение вашего Google Drive

17.2.2. Реализация сетей генератора и дискриминатора

Итак, приступим к реализации нашей первой модели GAN с генератором и дискриминатором в виде двух полно связанных сетей с одним или несколькими скрытыми слоями.

На рис. 17.8 схематически представлена исходная GAN, основанная на полно связанных слоях, которую мы будем называть *обычной GAN*.

В этой модели для каждого скрытого слоя применяется функция активации ReLU с утечкой (leaky ReLU). Использование ReLU приводит к разреженным градиентам, что

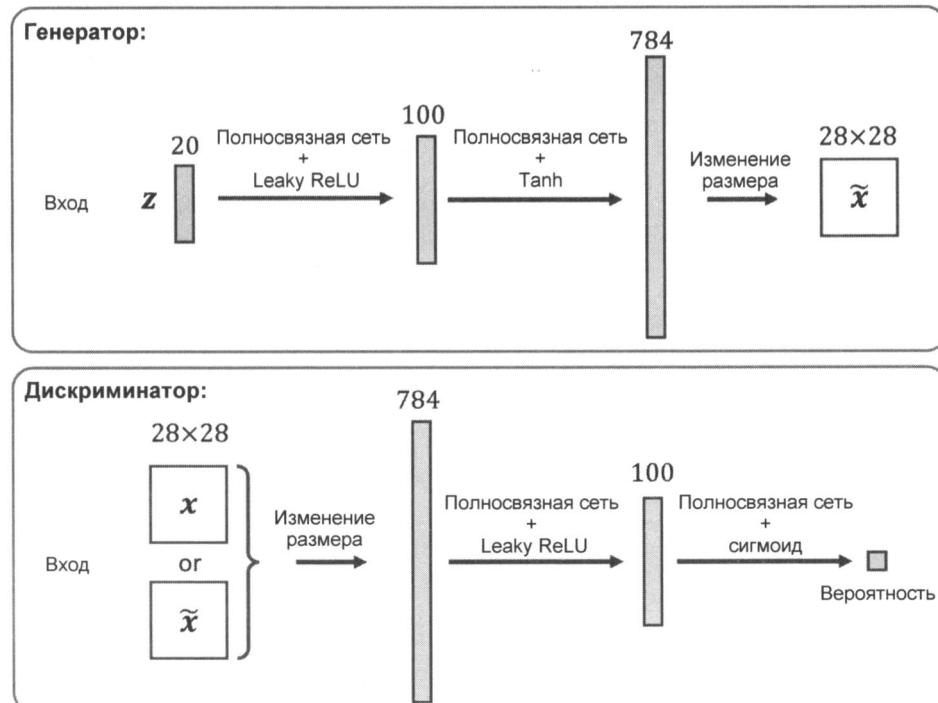


Рис. 17.8. Модель GAN с генератором и дискриминатором в виде двух полно связанных сетей

неприемлемо, если мы хотим иметь градиенты для всего диапазона входных значений. В сети дискриминатора за каждым скрытым слоем также следует выпадающий слой. Кроме того, выходной слой генератора использует функцию активации гиперболического тангенса (\tanh). (Для сети генератора рекомендуется использовать активацию \tanh , т. к. это помогает в обучении.)

Выходной слой в дискриминаторе не имеет функции активации (т. е. применяется линейная активация). В качестве альтернативы мы можем использовать сигмоидную функцию активации, чтобы получить на выходе вероятности.



Функция спрямленной линейной активации с утечкой (leaky ReLU)

В главе 12 мы рассмотрели различные нелинейные функции активации, которые можно использовать в нейросетевой модели. Если вы помните, активация ReLU была определена как функция $\sigma(z) = \max(0, z)$, которая подавляет отрицательные входные значения, т. е. отрицательные значения устанавливаются равными нулю. Следовательно, использование функции активации ReLU может привести к разреженным градиентам во время обратного распространения. Разреженные градиенты не всегда вредны и даже могут принести пользу классифицирующим моделям. Однако в некоторых приложениях, таких как GAN, может быть полезно получить градиенты для всего диапазона входных значений. Мы можем добиться этого, внеся незначительную модификацию в функцию ReLU, чтобы она выдавала небольшие значения для отрицательных входных данных. Эта модифицированная версия функции ReLU также известна как ReLU с утечкой (leaky ReLU). Коротко говоря, функция активации ReLU с утечкой допускает ненулевые градиенты для отрицательных входных данных, и в результате это делает сети в целом более выразительными.

Функция активации ReLU с утечкой определяется следующим образом (рис. 17.9). Здесь α определяет наклон для отрицательных (предактивационных) входных данных.

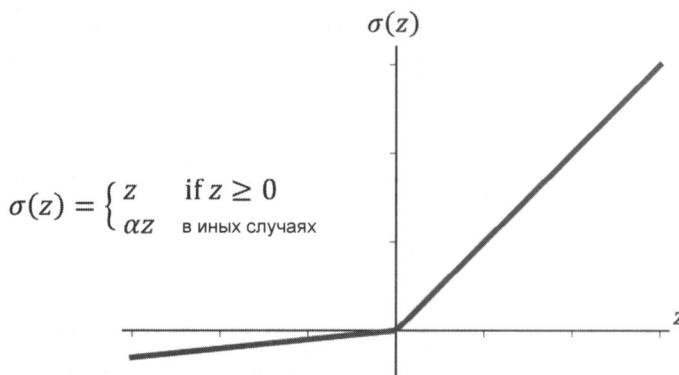


Рис. 17.9. Функция активации ReLU с утечкой

Итак, мы определяем для каждой из двух сетей две вспомогательные функции, создаем экземпляр модели из класса PyTorch `nn.Sequential` и добавляем упомянутые слои:

```
>>> import torch.nn as nn
>>> import numpy as np
```

```

>>> import matplotlib.pyplot as plt
>>> ## определение функции генератора:
>>> def make_generator_network(
...         input_size=20,
...         num_hidden_layers=1,
...         num_hidden_units=100,
...         num_output_units=784):
...     model = nn.Sequential()
...     for i in range(num_hidden_layers):
...         model.add_module(f'fc_g{i}',
...                         nn.Linear(input_size, num_hidden_units))
...         model.add_module(f'relu_g{i}', nn.LeakyReLU())
...         input_size = num_hidden_units
...     model.add_module(f'fc_g{num_hidden_layers}',
...                     nn.Linear(input_size, num_output_units))
...     model.add_module('tanh_g', nn.Tanh())
...     return model
>>>
>>> ## определение функции дискриминатора:
>>> def make_discriminator_network(
...         input_size,
...         num_hidden_layers=1,
...         num_hidden_units=100,
...         num_output_units=1):
...     model = nn.Sequential()
...     for i in range(num_hidden_layers):
...         model.add_module(
...             f'fc_d{i}',
...             nn.Linear(input_size, num_hidden_units, bias=False)
...         )
...         model.add_module(f'relu_d{i}', nn.LeakyReLU())
...         model.add_module('dropout', nn.Dropout(p=0.5))
...         input_size = num_hidden_units
...     model.add_module(f'fc_d{num_hidden_layers}',
...                     nn.Linear(input_size, num_output_units))
...     model.add_module('sigmoid', nn.Sigmoid())
...     return model

```

Затем указываем для модели параметры обучения. Как вы помните из предыдущих глав, размер изображения в наборе данных MNIST составляет 28×28 пикселов. (Есть только один цветовой канал, потому что MNIST содержит лишь изображения в градациях серого.) Далее мы указываем размер входного вектора z равным 20. Поскольку мы в качестве демонстрации подхода реализуем очень простую модель GAN и используем полностью связанные слои, мы задействуем в каждой сети только один скрытый слой со 100 узлами. В следующем коде мы определяем и инициализируем обе сети и выводим их сводную информацию:

```

>>> image_size = (28, 28)
>>> z_size = 20
>>> gen_hidden_layers = 1

```

```
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> disc_hidden_size = 100
>>> torch.manual_seed(1)
>>> gen_model = make_generator_network(
...     input_size=z_size,
...     num_hidden_layers=gen_hidden_layers,
...     num_hidden_units=gen_hidden_size,
...     num_output_units=np.prod(image_size)
... )
>>> print(gen_model)
Sequential(
  (fc_g0): Linear(in_features=20, out_features=100, bias=False)
  (relu_g0): LeakyReLU(negative_slope=0.01)
  (fc_g1): Linear(in_features=100, out_features=784, bias=True)
  (tanh_g): Tanh()
)

>>> disc_model = make_discriminator_network(
...     input_size=np.prod(image_size),
...     num_hidden_layers=disc_hidden_layers,
...     num_hidden_units=disc_hidden_size
... )
>>> print(disc_model)
Sequential(
  (fc_d0): Linear(in_features=784, out_features=100, bias=False)
  (relu_d0): LeakyReLU(negative_slope=0.01)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc_d1): Linear(in_features=100, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

17.2.3. Определение набора обучающих данных

На следующем шаге мы загрузим набор данных MNIST из PyTorch и выполним необходимую предварительную обработку. Поскольку выходной слой генератора использует функцию активации `tanh`, значения пикселов синтезированных изображений будут находиться в диапазоне $(-1, 1)$. Однако входные пиксели изображений MNIST расположены в диапазоне $[0, 255]$ (тип данных `PIL.Image.Image`), поэтому на этапах предварительной обработки мы применим функцию `torchvision.transforms.ToTensor` для преобразования пикселов входного изображения в тензор. В результате, помимо изменения типа данных, вызов этой функции также изменит диапазон интенсивностей входных пикселов на $[0, 1]$. После чего мы можем сдвинуть их на -0.5 и масштабировать с коэффициентом 0.5, чтобы интенсивность пикселов попала в диапазон $[-1, 1]$, поскольку это может улучшить обучение на основе градиентного спуска:

```
>>> import torchvision
>>> from torchvision import transforms
```

```
>>> image_path = './'
>>> transform = transforms.Compose([
...     transforms.ToTensor(),
...     transforms.Normalize(mean=(0.5), std=(0.5)),
... ])
>>> mnist_dataset = torchvision.datasets.MNIST(
...     root=image_path, train=True,
...     transform=transform, download=False
... )
>>> example, label = next(iter(mnist_dataset))
>>> print(f'Min: {example.min()} Max: {example.max()}')
>>> print(example.shape)
Min: -1.0 Max: 1.0
torch.Size([1, 28, 28])
```

Кроме того, мы также создадим случайный вектор z на основе желаемого случайного распределения (в этом примере кода — равномерного или нормального, которые являются наиболее распространенными вариантами):

```
>>> def create_noise(batch_size, z_size, mode_z):
...     if mode_z == 'uniform':
...         input_z = torch.rand(batch_size, z_size)*2 - 1
...     elif mode_z == 'normal':
...         input_z = torch.randn(batch_size, z_size)
...     return input_z
```

Теперь проверим созданный нами объект набора данных. В следующем коде мы возьмем один пакет примеров и выведем размер массива этого примера входных векторов и изображений. Кроме того, чтобы лучше понять общий поток данных нашей модели GAN, в этом же коде мы обработаем прямой проход для нашего генератора и дискриминатора.

Сначала мы передадим набор входных векторов z в генератор и получим его выходные данные g_{output} . Это будет пакет фиктивных примеров, который мы затем передадим в модель дискриминатора, чтобы получить вероятности для пакета фиктивных примеров $d_{\text{proba_fake}}$. Кроме того, обработанные изображения, которые мы получим из объекта набора данных, тоже будут переданы в модель дискриминатора, что даст нам вероятности для реальных примеров $d_{\text{proba_real}}$:

```
>>> from torch.utils.data import DataLoader
>>> batch_size = 32
>>> dataloader = DataLoader(mnist_dataset, batch_size, shuffle=False)
>>> input_real, label = next(iter(dataloader))
>>> input_real = input_real.view(batch_size, -1)
>>> torch.manual_seed(1)
>>> mode_z = 'uniform' # 'uniform' или 'normal'
>>> input_z = create_noise(batch_size, z_size, mode_z)
>>> print('вход-z -- размеры:', input_z.shape)
>>> print('вход-реальн. -- размеры:', input_real.shape)
вход-z -- размеры: torch.Size([32, 20])
вход-реальн. -- размеры: torch.Size([32, 784])
```

```
>>> g_output = gen_model(input_z)
>>> print('Выход G -- размеры:', g_output.shape)
Выход G -- размеры: torch.Size([32, 784])

>>> d_proba_real = disc_model(input_real)
>>> d_proba_fake = disc_model(g_output)
>>> print('Дискр. (реальный) -- размеры:', d_proba_real.shape)
>>> print('Дискр. (фiktивный) -- размеры:', d_proba_fake.shape)
Дискр. (реальный) -- размеры: torch.Size([32, 1])
Дискр. (фiktивный) -- размеры: torch.Size([32, 1])
```

Две вероятности: `d_proba_fake` и `d_proba_real` — понадобятся для вычисления функций потерь при обучении модели.

17.2.4. Обучение модели GAN

На следующем шаге мы создадим экземпляр `nn.BCELoss`, который будет нашей функцией потерь, и используем его для вычисления бинарных потерь перекрестной энтропии для генератора и дискриминатора, связанных с пакетами, которые мы только что обработали. Для этого нам также нужны метки истинности для каждого выхода. Для генератора мы создадим состоящий из единиц вектор того же размера, что и вектор, содержащий предсказанные вероятности для сгенерированных изображений `d_proba_fake`. Для потери дискриминатора у нас есть два члена: потеря при обнаружении фiktивных примеров на основе `d_proba_fake` и потеря при обнаружении реальных примеров на основе `d_proba_real`.

Эталонные метки для фiktивных изображений будут представлять собой вектор из нолей, который мы можем сгенерировать с помощью `torch.zeros()` (или `torch.zeros_like()`). Точно так же мы можем сгенерировать эталонные метки для реальных изображений с помощью функции `torch.ones()` (или `torch.ones_like()`), которая создает вектор из единиц:

```
>>> loss_fn = nn.BCELoss()
>>> ## Потери генератора
>>> g_labels_real = torch.ones_like(d_proba_fake)
>>> g_loss = loss_fn(d_proba_fake, g_labels_real)
>>> print(f'Потери генератора: {g_loss:.4f}')
Потери генератора: 0.6863

>>> ## Потери дискриминатора
>>> d_labels_real = torch.ones_like(d_proba_real)
>>> d_labels_fake = torch.zeros_like(d_proba_fake)
>>> d_loss_real = loss_fn(d_proba_real, d_labels_real)
>>> d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
>>> print(f' Потери дискриминатора: Реальн. {d_loss_real:.4f} Поддельн. {d_loss_fake:.4f}')
Потери дискриминатора: Реальн. 0.6226 Поддельн. 0.7007
```

В этом примере кода показано пошаговое вычисление различных членов потерь для лучшего понимания общей концепции обучения модели GAN. Следующий код настроит модель GAN и реализует цикл обучения, где мы включим эти вычисления в цикл `for`.

Начнем с настройки загрузчика данных для реального набора данных, модели генератора и дискриминатора, а также отдельного оптимизатора Adam для каждой из двух моделей:

```
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> np.random.seed(1)
>>> mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
...                         shuffle=True, drop_last=True)

>>> gen_model = make_generator_network(
...     input_size=z_size,
...     num_hidden_layers=gen_hidden_layers,
...     num_hidden_units=gen_hidden_size,
...     num_output_units=np.prod(image_size)
... ).to(device)
>>> disc_model = make_discriminator_network(
...     input_size=np.prod(image_size),
...     num_hidden_layers=disc_hidden_layers,
...     num_hidden_units=disc_hidden_size
... ).to(device)
>>> loss_fn = nn.BCELoss()
>>> g_optimizer = torch.optim.Adam(gen_model.parameters())
>>> d_optimizer = torch.optim.Adam(disc_model.parameters())
```

Кроме того, вычислим градиенты потерь по отношению к весам модели, оптимизируем параметры генератора и дискриминатора с помощью двух отдельных оптимизаторов Adam и напишем две служебные функции для обучения дискриминатора и генератора:

```
>>> ## Обучение дискриминатора
>>> def d_train(x):
...     disc_model.zero_grad()
...     # Обучение дискриминатора на реальном пакете
...     batch_size = x.size(0)
...     x = x.view(batch_size, -1).to(device)
...     d_labels_real = torch.ones(batch_size, 1, device=device)
...     d_proba_real = disc_model(x)
...     d_loss_real = loss_fn(d_proba_real, d_labels_real)
...     # Обучение дискриминатора на поддельном пакете
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_proba_fake = disc_model(g_output)
...     d_labels_fake = torch.zeros(batch_size, 1, device=device)
...     d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
...     # Обр. распр. градиента и оптимизация ТОЛЬКО параметров D
...     d_loss = d_loss_real + d_loss_fake
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item(), d_proba_real.detach(), \
...           d_proba_fake.detach()
>>>
```

```
>>> ## Обучение генератора
>>> def g_train(x):
...     gen_model.zero_grad()
...     batch_size = x.size(0)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_labels_real = torch.ones(batch_size, 1, device=device)
...
...     g_output = gen_model(input_z)
...     d_proba_fake = disc_model(g_output)
...     g_loss = loss_fn(d_proba_fake, g_labels_real)
...     # Обр. распр. градиента и оптимизация ТОЛЬКО параметров G
...     g_loss.backward()
...     g_optimizer.step()
...     return g_loss.data.item()
```

Далее мы начнем чередовать обучение генератора и дискриминатора сериями по 100 эпох. Для каждой эпохи мы будем записывать потери для генератора, потери для дискриминатора и потери для реальных данных и поддельных данных соответственно. Кроме того, после каждой эпохи мы станем генерировать несколько примеров из фиксированного ввода шума, используя текущую модель генератора и вызывая функцию `create_samples()`. Синтезированные изображения будут храниться в списке Python:

```
>>> fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
>>> def create_samples(g_model, input_z):
...     g_output = g_model(input_z)
...     images = torch.reshape(g_output, (batch_size, *image_size))
...     return (images+1)/2.0
>>>
>>> epoch_samples = []
>>> all_d_losses = []
>>> all_g_losses = []
>>> all_d_real = []
>>> all_d_fake = []
>>> num_epochs = 100
>>>
>>> for epoch in range(1, num_epochs+1):
...     d_losses, g_losses = [], []
...     d_vals_real, d_vals_fake = [], []
...     for i, (x, _) in enumerate(mnist_dl):
...         d_loss, d_proba_real, d_proba_fake = d_train(x)
...         d_losses.append(d_loss)
...         g_losses.append(g_train(x))
...         d_vals_real.append(d_proba_real.mean().cpu())
...         d_vals_fake.append(d_proba_fake.mean().cpu())
...
...     all_d_losses.append(torch.tensor(d_losses).mean())
...     all_g_losses.append(torch.tensor(g_losses).mean())
...     all_d_real.append(torch.tensor(d_vals_real).mean())
...     all_d_fake.append(torch.tensor(d_vals_fake).mean())
```

```

...
    print(f'Эпоха {epoch:03d} | Средн. потери >>
...
        f' G/D {all_g_losses[-1]:.4f}/{all_d_losses[-1]:.4f}''
...
        f' [D-Real: {all_d_real[-1]:.4f}]'
...
        f' D-Fake: {all_d_fake[-1]:.4f})')
...
epoch_samples.append(
    create_samples(gen_model, fixed_z).detach().cpu().numpy()
)

```

Эпоха 001 | Средн. потери >> G/D 0.9546/0.8957 [D-Real: 0.8074 D-Fake: 0.4687]
 Эпоха 002 | Средн. потери >> G/D 0.9571/1.0841 [D-Real: 0.6346 D-Fake: 0.4155]
 Эпоха ...
 Эпоха 100 ! Средн. потери >> G/D 0.8622/1.2878 [D-Real: 0.5488 D-Fake: 0.4518]

При использовании GPU в Google Colab процесс обучения, который мы реализовали в приведенном блоке кода, должен быть завершен менее чем за час. (Он может выполниться даже быстрее на вашем персональном компьютере, если в нем установлены новые и мощные CPU и GPU.) После завершения обучения модели часто бывает полезно построить график потерь дискриминатора и генератора, чтобы проанализировать поведение обеих подсетей и оценить, сошлись ли они.

Также полезно построить график средних вероятностей пакетов реальных и фиктивных примеров, вычисленных дискриминатором на каждой итерации. Мы ожидаем, что эти вероятности будут около 0.5, а это означает, что дискриминатор не может уверенно различать настоящие и фиктивные изображения:

```

>>> import itertools
>>> fig = plt.figure(figsize=(16, 6))
>>> ## Построение графика потерь
>>> ax = fig.add_subplot(1, 2, 1)
>>> plt.plot(all_g_losses, label='Потери генератора')
>>> half_d_losses = [all_d_loss/2 for all_d_loss in all_d_losses]
>>> plt.plot(half_d_losses, label='Потери дискриминатора')
>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Итерация', size=15)
>>> ax.set_ylabel('Потери', size=15)
>>>
>>> ## Построение выхода дискриминатора
>>> ax = fig.add_subplot(1, 2, 2)
>>> plt.plot(all_d_real, label='Real: $D(\mathbf{x})$')
>>> plt.plot(all_d_fake, label='Fake: $D(G(\mathbf{z}))$')
>>> plt.legend(fontsize=20)
>>> ax.set_xlabel('Итерация', size=15)
>>> ax.set_ylabel('Выход дискриминатора', size=15)
>>> plt.show()

```

Как следует из выходных данных дискриминатора, показанных на рис. 17.10, на ранних этапах обучения дискриминатор смог быстро научиться довольно точно различать настоящие и фиктивные изображения, — т. е. фиктивные примеры имели вероятности, близкие к 0, а реальные — близкие к 1. Причина этого заключалась в том, что фиктивные изображения не были похожи на настоящие, поэтому отличить подлинник от подделки было довольно легко. По мере дальнейшего обучения генератор будет лучше

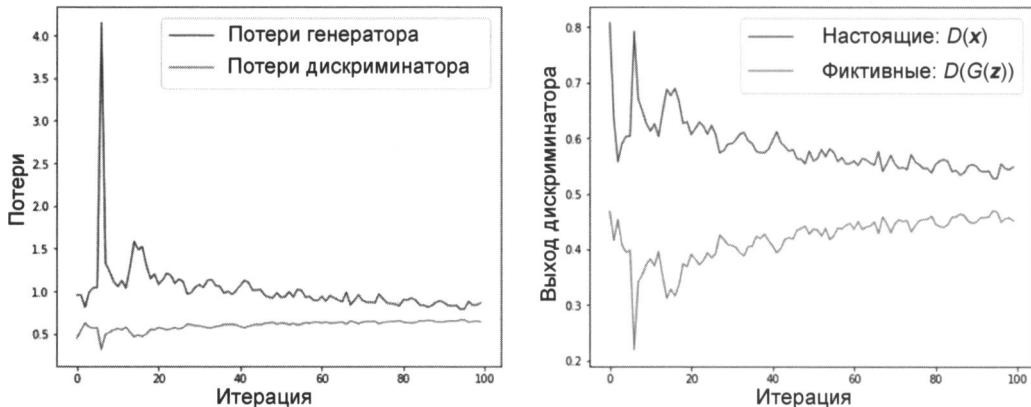


Рис. 17.10. Производительность дискриминатора

синтезировать реалистичные изображения, что приведет к тому, что вероятности как реальных, так и фиктивных примеров будут близки к 0.5.

Кроме того, мы также можем видеть, как выходные данные генератора, т. е. синтезированные изображения, меняются во время обучения. В следующем коде мы визуализируем некоторые изображения, созданные генератором на определенных эпохах:

```
>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
...         if j == 0:
...             ax.text(
...                 -0.06, 0.5, f'Epoch {e}',
...                 rotation=90, size=18, color='red',
...                 horizontalalignment='right',
...                 verticalalignment='center',
...                 transform=ax.transAxes
...             )
...
...         image = epoch_samples[e-1][j]
...         ax.imshow(image, cmap='gray_r')
...
>>> plt.show()
```

Как можно видеть на рис. 17.11, сеть генератора по мере обучения выдает все более реалистичные изображения. Однако даже после 100 эпох полученные изображения все еще сильно отличаются от рукописных цифр, содержащихся в наборе данных MNIST.

В этом разделе мы разработали очень простую модель GAN только с одним полностью связанным скрытым слоем как для генератора, так и для дискриминатора. После обучения модели GAN на наборе данных MNIST мы смогли добиться многообещающих, хотя и неудовлетворительных, результатов генерации новых рукописных цифр.

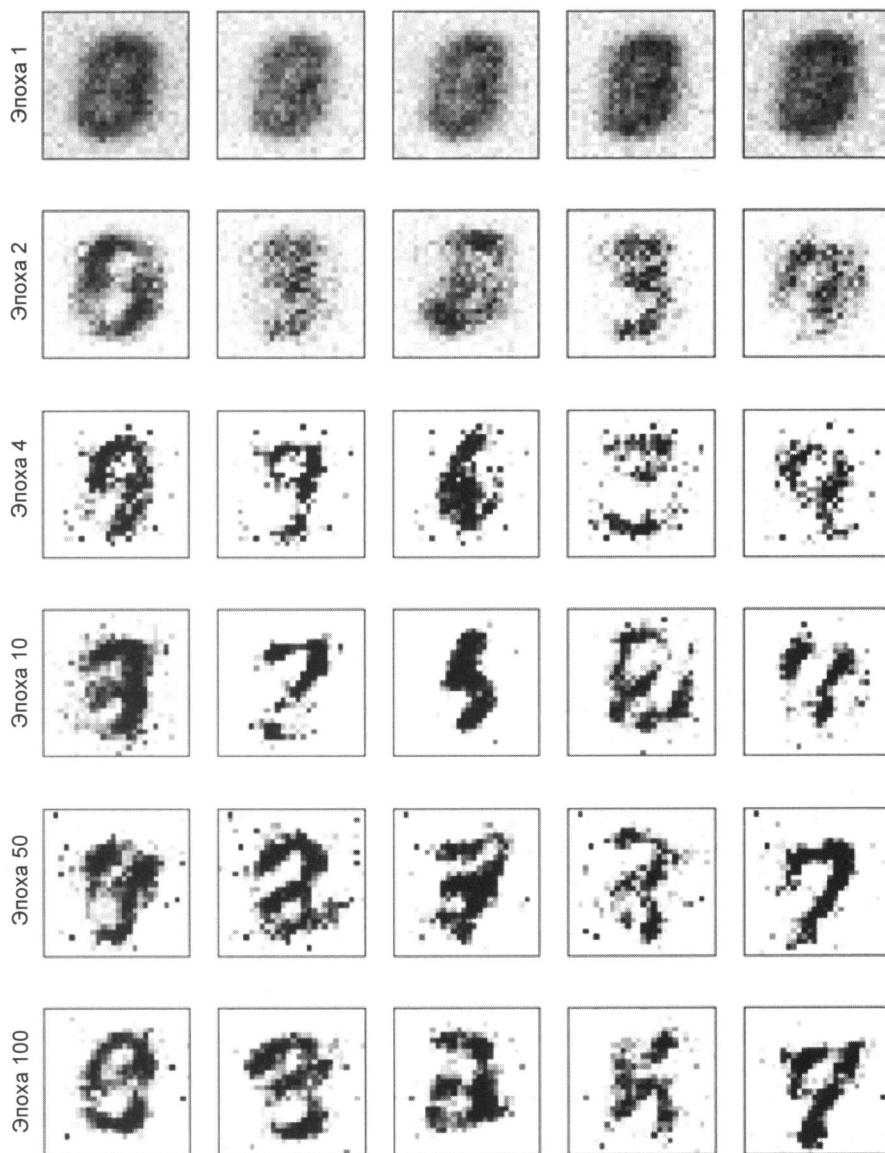


Рис. 17.11. Изображения, созданные генератором

Как вы узнали из главы 14, архитектуры нейронных сетей со сверточными слоями имеют несколько преимуществ по сравнению с полно связанными слоями, когда речь идет о классификации изображений. Аналогичным образом, добавление сверточных слоев в нашу модель GAN для работы с данными изображений может улучшить результат. В следующем разделе мы построим глубокую сверточную GAN (Deep Convolution GAN, DCGAN), которая использует сверточные слои как в генераторе, так и в дискриминаторе.

17.3. Улучшение качества синтезированных изображений с помощью сверточной GAN и метрики Вассерштейна

В этом разделе мы реализуем DCGAN, что позволит нам улучшить производительность, которую мы получили в предыдущем примере GAN. Кроме того, мы кратко поговорим о методе дополнительных ключей, или *GAN Вассерштейна* (Wasserstein GAN, WGAN).

Здесь будут рассмотрены следующие методы:

- ◆ транспонированная свертка;
- ◆ пакетная нормализация (BatchNorm);
- ◆ WGAN.

Архитектура DCGAN была предложена в 2016 году³. В своей статье исследователи предложили использовать сверточные слои как для сетей генератора, так и для сетей дискриминатора. Начиная со случайного вектора z , DCGAN сначала использует полностью связанный слой для проецирования z в новый вектор нужного размера, чтобы его можно было преобразовать в представление пространственной свертки ($h \times w \times c$), которое меньше, чем размер выходного изображения. Затем — для повышения разрешения карт признаков до желаемого размера выходного изображения — применяется ряд сверточных слоев так называемой *транспонированной свертки* (transposed convolution).

17.3.1. Транспонированная свертка

В главе 14 было рассказано об операции свертки в одномерном и двумерном пространствах. В частности, вы узнали, как выбор величины заполнения и шага влияет на выходные карты признаков. В то время как операция свертки обычно используется для понижения разрешения пространства признаков (например, путем установки шага равным 2 или за счет добавления слоя объединения после слоя свертки), операция *транспонированной свертки*, как правило, применяется для *повышения* разрешения пространства признаков.

Чтобы понять суть операции транспонированной свертки, давайте проведем простой мысленный эксперимент. Предположим, что у нас есть входная карта признаков размером $n \times n$. Применим к этому входу $n \times n$ операцию двумерной свертки с определенными параметрами заполнения и шага, в результате чего получим выходную карту признаков размером $m \times m$. Теперь вопрос заключается в том, как применить другую операцию свертки, чтобы получить карту признаков с начальной размерностью $n \times n$ из этой выходной карты признаков $m \times m$, сохраняя при этом связность между входом и выходом? Нужно подчеркнуть, что восстанавливается только размерность входной матрицы $n \times n$, а не фактические значения матрицы. Принцип работы транспонированной свертки показан на рис. 17.12.

³ См. «Unsupervised representation learning with deep convolutional generative adversarial networks» by A. Radford, L. Metz, and S. Chintala, <https://arxiv.org/pdf/1511.06434.pdf>.

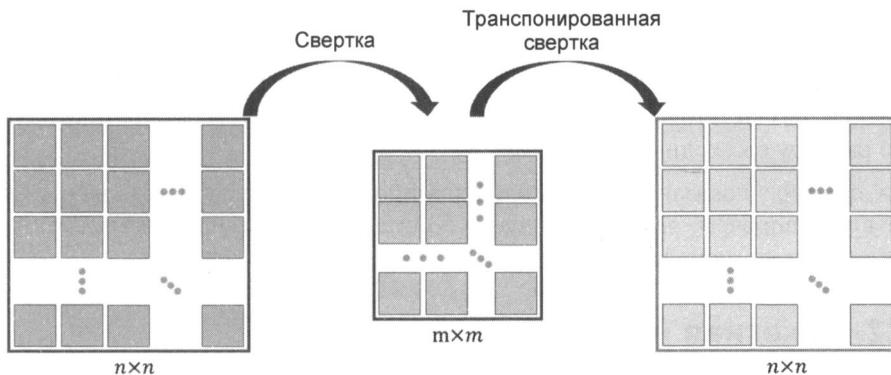


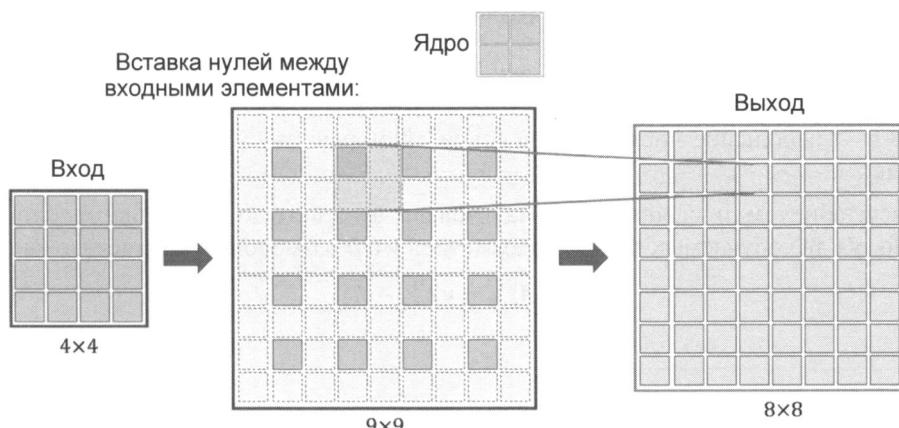
Рис. 17.12. Транспонированная свертка



Транспонированная свертка или обращенная свертка?

Транспонированная свертка (transposed convolution) также называется *сверткой с дробным шагом* (fractionally strided convolution). В литературе по глубокому обучению другим распространенным термином, который используется для обозначения транспонированной свертки, является *обращенная свертка* (deconvolution, деконволюция). Однако обращенная свертка изначально была определена как обратная операция свертки f на карте признаков x с весовыми параметрами w , создающая карту признаков x' , $f_w(x) = x'$. Отсюда функция обращенной свертки f^{-1} может быть определена как $f_w^{-1}(f(x)) = x$. В свою очередь, транспонированная свертка просто сосредоточена на восстановлении размерности пространства признаков, а не фактических значений.

Повышение размерности карт признаков с использованием транспонированной свертки происходит путем вставки нулей между элементами входных карт признаков. На рис. 17.13 показан пример применения транспонированной свертки к входным данным размером 4×4 с шагом 2×2 и размером ядра 2×2 . Матрица размером 9×9 в центре пока-

Рис. 17.13. Применение транспонированной свертки ко входу 4×4

зывает результат вставки нулей во входную карту объектов. Затем выполнение обычной свертки с использованием ядра 2×2 с шагом 1 дает выходную матрицу размером 8×8 . Мы можем проверить результат, применив к этой матрице обычную свертку с шагом 2, в результате чего получается карта признаков размером 4×4 , которая совпадает по размеру с исходными данными.

В этом примере показан обобщенный принцип работы транспонированной свертки. Существуют, впрочем, различные случаи, когда размер входных данных, размер ядра, шаги и варианты заполнения могут изменить выходные данные⁴.

17.3.2. Пакетная нормализация

Метод *пакетной нормализации* BatchNorm был представлен в 2015 году⁵. Одна из основных идей BatchNorm заключается в нормализации входных данных слоя и предотвращении изменений в их распределении во время обучения, что обеспечивает более быструю и лучшую сходимость.

BatchNorm преобразует мини-пакет признаков, исходя из вычисленной статистики. Предположим, что у нас есть действующие карты признаков предварительной активации, полученные после сверточного слоя, в четырехмерном тензоре Z размером $[m \times c \times h \times w]$, где m — количество примеров в пакете (т. е. размер пакета); $h \times w$ — пространственный размер карт признаков, а c — количество каналов. Процедуру BatchNorm можно разделить на три этапа (рис. 17.14):

1. Вычисляем среднее значение и стандартное отклонение действующих входных данных для каждого мини-пакета:

$$\begin{aligned}\mu_B &= \frac{1}{m \times h \times w} \sum_{i,j,k} Z^{[i,j,k]} \\ \sigma_B^2 &= \frac{1}{m \times h \times w} \sum_{i,j,k} (Z^{[i,j,k]} - \mu_B)^2,\end{aligned}$$

где μ_B и σ_B^2 имеют размер c .

2. Стандартизуем действующие входные данные для всех примеров в пакете:

$$Z_{\text{std}}^{[i]} = \frac{Z^{[i]} - \mu_B}{\sigma_B + \epsilon},$$

где ϵ — небольшое число для числовой стабильности (чтобы избежать деления на ноль).

3. Масштабируем и сдвигаем нормализованные действующие входные данные, используя два обучаемых вектора параметров γ и β размером c (количество каналов):

$$A_{\text{pre}}^{[i]} = \gamma Z_{\text{std}}^{[i]} + \beta.$$

⁴ Если вы хотите узнать больше обо всех этих случаях, обратитесь к руководству «Guide to Convolution Arithmetic for Deep Learning» by Vincent Dumoulin and Francesco Visin, 2018, <https://arxiv.org/pdf/1603.07285.pdf>.

⁵ См. «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift» by Sergey Ioffe and Christian Szegedy, <https://arxiv.org/pdf/1502.03167.pdf>.

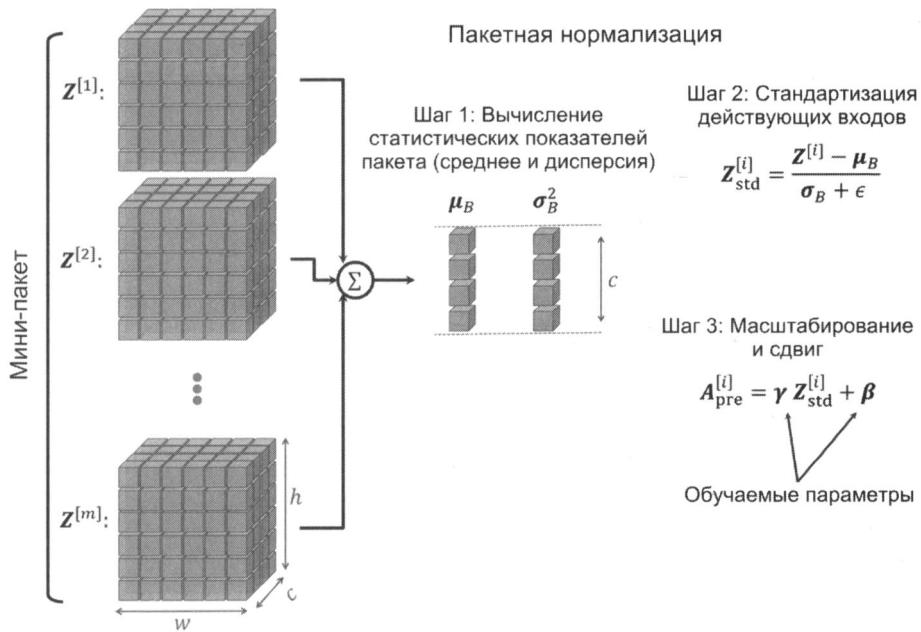


Рис. 17.14. Процесс пакетной нормализации

На первом этапе BatchNorm вычисляются среднее значение μ_B и стандартное отклонение σ_B мини-пакета. И тот и другой являются векторами размером c (где c — количество каналов). Затем эти статистические данные используются на шаге 2 для масштабирования примеров в каждом мини-пакете с помощью нормализации z -оценки (стандартизации), в результате чего получаются стандартизованные действующие входные данные $Z_{\text{std}}^{(i)}$. После нормализации эти входные данные центрированы по среднему и имеют единичную дисперсию, что обычно полезно для оптимизации на основе градиентного спуска. С другой стороны, постоянная нормализация действующих входных данных таким образом, чтобы они имели одинаковые свойства в разных мини-пакетах (которые могут различаться), способна серьезно повлиять на репрезентативную возможность нейросети. Причину этого можно понять, рассмотрев признак $x \sim N(0,1)$, который после сигмоидной активации $\sigma(x)$ образует линейную область для значений, близких к 0. Следовательно, на шаге 3 обучаемые параметры β и γ , которые являются векторами размером c (количество каналов), позволяют BatchNorm управлять смещением и разбросом нормализованных признаков.

Во время обучения вычисляются скользящие средние μ_B и текущая дисперсия σ_B^2 , которые используются вместе с настроенными параметрами β и γ для нормализации тестовых примеров при оценке.



Почему BatchNorm способствует оптимизации?

Изначально BatchNorm разрабатывали для уменьшения так называемого *внутреннего сдвига переменных* (internal covariance shift), который определяется как изменения, происходящие в распределении активаций слоя из-за обновленных параметров сети во время обучения.

Чтобы объяснить это на простом примере, рассмотрим фиксированный пакет, который проходит через сеть в эпоху 1. Запишем активации каждого слоя для этого пакета. После прохода по всему обучающему набору данных и обновления параметров модели мы начинаем вторую эпоху, когда через сеть проходит ранее фиксированный пакет. Затем мы сравниваем активации слоя из первой и второй эпох. Поскольку параметры сети изменились, мы наблюдаем, что активации также изменились. Это явление и называется *внутренним сдвигом переменных*, который, как считалось, замедляет обучение нейронной сети.

Однако в 2018 году С. Сантуркар, Д. Ципрас, А. Ильяс и А. Мадри дополнительно исследовали природу эффективности BatchNorm. В своем исследовании ученые установили, что влияние BatchNorm на внутренний сдвиг переменных незначительно. Основываясь на результатах своих экспериментов, они предположили, что эффективность BatchNorm вместо этого обусловлена более гладкой поверхностью функции потерь, что делает невыпуклую оптимизацию более надежной.

Если вам интересно узнать больше об этих исследованиях, прочтите исходную статью «How Does Batch Normalization Help Optimization?», которая находится в свободном доступе по адресу: <http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf>.

API PyTorch предоставляет класс `nn.BatchNorm2d()` (или `nn.BatchNorm1d()` для одномерного ввода), который мы можем использовать в качестве слоя при определении наших моделей, — он выполнит все шаги процедуры пакетной нормализации BatchNorm. Заметим, что поведение алгоритма при обновлении параметров γ и β зависит от того, обучается ли модель. Эти параметры изучаются только во время обучения и затем используются для нормализации во время оценки.

17.3.3. Реализация генератора и дискриминатора

К настоящему времени мы рассмотрели все основные компоненты модели DCGAN, которую теперь реализуем в коде.

Архитектура сети генератора (карты признаков после каждого слоя) показана на рис. 17.15. На вход генератора поступает вектор z размером 100. Затем последователь-

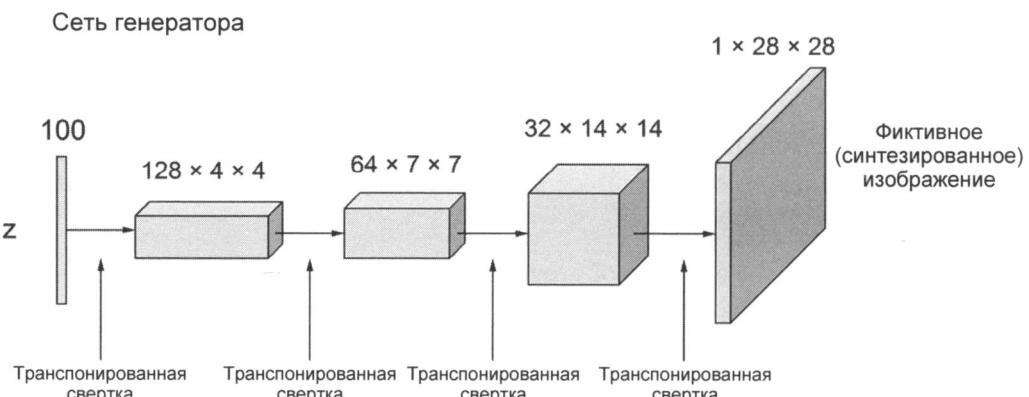


Рис. 17.15. Архитектура сети генератора

ность транспонированных сверток с использованием метода `nn.ConvTranspose2d()` повышает размер карт признаков до тех пор, пока пространственный размер результирующих карт признаков не достигнет 28×28 . Количество каналов уменьшается вдвое после каждого слоя транспонированной свертки, кроме последнего, который использует только один выходной фильтр для генерации изображения в градациях серого. За каждым слоем транспонированной свертки, кроме последнего, следуют BatchNorm и функция активации ReLU с утечкой. Последний слой использует активацию `tanh` (без BatchNorm).

Архитектура сети дискриминатора показана на рис. 17.16. Дискриминатор получает изображения размером $1 \times 28 \times 28$, которые проходят через четыре слоя свертки. Первые три слоя уменьшают пространственную размерность на 4, увеличивая при этом количество каналов карт признаков. За каждым слоем свертки также следует активация BatchNorm и ReLU с утечкой. Последний слой свертки использует ядра размером 4×4 и один фильтр для уменьшения пространственной размерности вывода до $1 \times 1 \times 1$. В конце за выводом свертки следует сигмоидная функция и сжатие до одного измерения.

Сеть дискриминатора

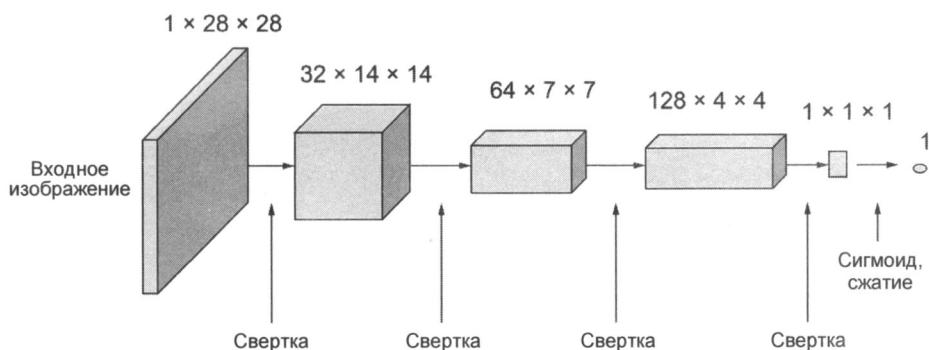


Рис. 17.16. Архитектура сети дискриминатора



Принципы проектирования архитектуры сверточных сетей GAN

Обратите внимание, что количество карт признаков в генераторе и дискриминаторе меняется по разным законам. В генераторе мы начинаем с большого количества карт признаков и уменьшаем их по мере продвижения к последнему слою. И наоборот, в дискриминаторе мы начинаем с небольшого количества каналов и увеличиваем его к последнему слою. Это важный момент, о котором нужно помнить при проектировании сверточных сетей: количество карт признаков обратно пропорционально их пространственному размеру. Когда пространственный размер карт признаков увеличивается, количество карт признаков уменьшается, и наоборот.

Кроме того, отметим, что обычно не рекомендуется использовать смещение в слое, который следует за слоем BatchNorm. Использование смещения в этом случае было бы излишним, т. к. BatchNorm уже имеет параметр смещения β . Вы можете убрать смещение для определенного слоя, просто задав параметр `bias=False` в `nn.ConvTranspose2d` или `nn.Conv2d`.

Код вспомогательной функции для создания класса генератора и дискриминатора выглядит следующим образом:

```
>>> def make_generator_network(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4, 1, 0, bias=False),
...         nn.BatchNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*4, n_filters*2, 3, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*2, n_filters, 4, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
...         nn.Tanh()
...     )
...     return model
>>>
>>> class Discriminator(nn.Module):
...     def __init__(self, n_filters):
...         super().__init__()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters, n_filters*2, 4, 2, 1, bias=False),
...             nn.BatchNorm2d(n_filters * 2),
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1, bias=False),
...             nn.BatchNorm2d(n_filters*4),
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...             nn.Sigmoid()
...         )
...
...     def forward(self, input):
...         output = self.network(input)
...         return output.view(-1, 1).squeeze(0)
```

С помощью вспомогательной функции и класса вы можете построить модель DCGAN и обучить ее, используя тот же объект набора данных MNIST, который мы инициализировали в предыдущем разделе, когда создавали простую полносвязную GAN. Мы можем создать сеть генератора, применив вспомогательную функцию, и вывести ее архитектуру следующим образом:

```
>>> z_size = 100
>>> image_size = (28, 28)
>>> n_filters = 32
>>> gen_model = make_generator_network(z_size, n_filters).to(device)
>>> print(gen_model)
```

```

Sequential(
  (0): ConvTranspose2d(100, 128, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): LeakyReLU(negative_slope=0.2)
  (3): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): LeakyReLU(negative_slope=0.2)
  (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): LeakyReLU(negative_slope=0.2)
  (9): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (10): Tanh()
)

```

Точно так же мы можем сгенерировать сеть дискриминатора и просмотреть ее архитектуру:

```

>>> disc_model = Discriminator(n_filters).to(device)
>>> print(disc_model)
Discriminator(
  (network): Sequential(
    (0): Conv2d(1, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(128, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)

```

Кроме того, мы можем использовать те же функции потерь и оптимизаторы, что и в разд. 17.2.4:

```

>>> loss_fn = nn.BCELoss()
>>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0003)
>>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)

```

Мы внесем несколько небольших изменений в процедуру обучения. В частности, изменим функцию `create_noise()` для генерации случайного ввода, чтобы выводить четырехмерный тензор вместо вектора:

```

>>> def create_noise(batch_size, z_size, mode_z):
...     if mode_z == 'uniform':
...         input_z = torch.rand(batch_size, z_size, 1, 1)*2 - 1
...     elif mode_z == 'normal':
...         input_z = torch.randn(batch_size, z_size, 1, 1)
...     return input_z

```

Функция d_train() для обучения дискриминатора не требует изменения размера входного изображения:

```
>>> def d_train(x):
...     disc_model.zero_grad()
...     # Обучение дискриминатора на реальном пакете
...     batch_size = x.size(0)
...     x = x.to(device)
...     d_labels_real = torch.ones(batch_size, 1, device=device)
...     d_proba_real = disc_model(x)
...     d_loss_real = loss_fn(d_proba_real, d_labels_real)
...     # Обучение дискриминатора на фиктивном пакете
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_proba_fake = disc_model(g_output)
...     d_labels_fake = torch.zeros(batch_size, 1, device=device)
...     d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
...     # обр. распр. градиента и оптимизация ТОЛЬКО
...     # параметров дискриминатора
...     d_loss = d_loss_real + d_loss_fake
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item(), d_proba_real.detach(), d_proba_fake.detach()
```

Далее мы начнем чередовать обучение генератора и дискриминатора сериями по 100 эпох. После каждой эпохи будем генерировать несколько примеров из фиксированного шумового ввода, используя текущую модель генератора с помощью вызова функции `create_samples()`:

```
>>> fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
>>> epoch_samples = []
>>> torch.manual_seed(1)
>>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     for i, (x, _) in enumerate(mnist_dl):
...         d_loss, d_proba_real, d_proba_fake = d_train(x)
...         d_losses.append(d_loss)
...         g_losses.append(g_train(x))
...         print(f'Эпоха {epoch:03d} | Средн. потери >>
...             f' G/D {torch.FloatTensor(g_losses).mean():.4f}'
...             f'/{torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples.append(
...         create_samples(
...             gen_model, fixed_z
...         ).detach().cpu().numpy()
...     )
Эпоха 001 | Средн. потери >> G/D 4.7016/0.1035
Эпоха 002 | Средн. потери >> G/D 5.9341/0.0438
...
Эпоха 099 | Средн. потери >> G/D 4.3753/0.1360
Эпоха 100 | Средн. потери >> G/D 4.4914/0.1120
```

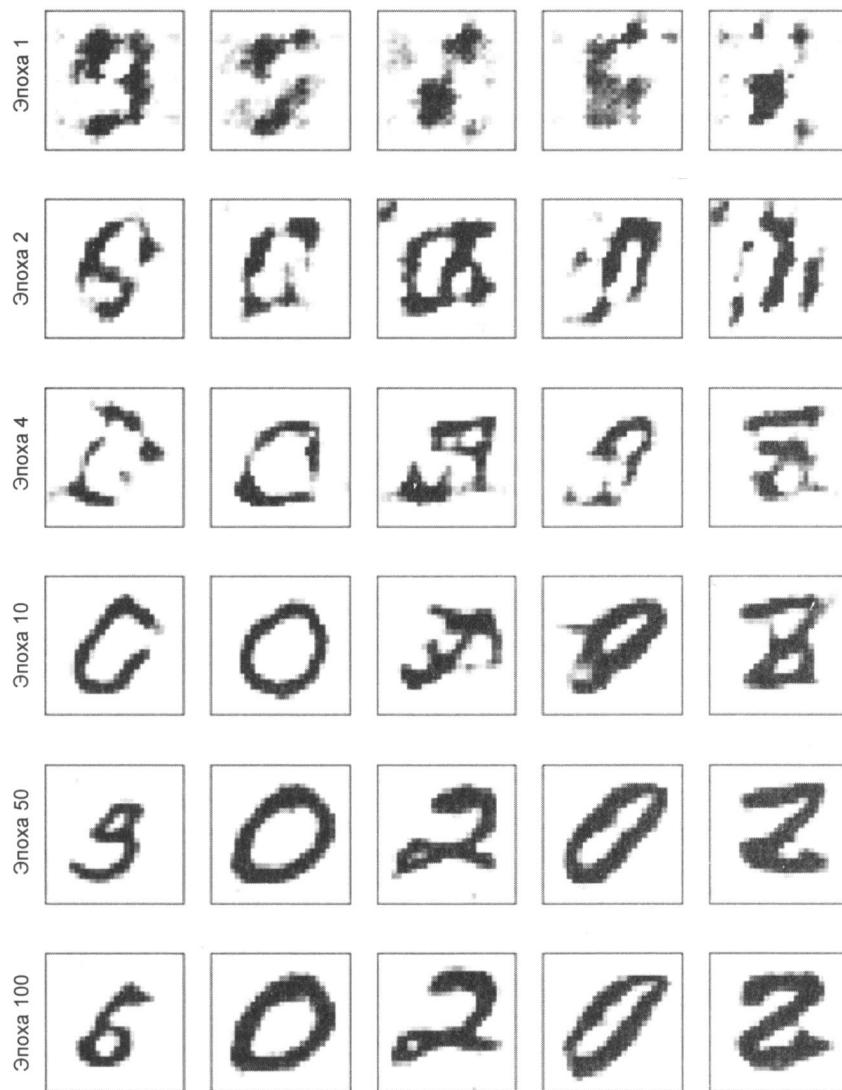


Рис. 17.17. Изображения, сгенерированные DCGAN

Наконец, выведем на экран сохраненные примеры некоторых эпох, чтобы увидеть, как обучается модель и как меняется качество синтезированных примеров в ходе обучения (рис. 17.17):

```
>>> selected_epochs = [1, 2, 4, 10, 50, 100]
>>> fig = plt.figure(figsize=(10, 14))
>>> for i,e in enumerate(selected_epochs):
...     for j in range(5):
...         ax = fig.add_subplot(6, 5, i*5+j+1)
...         ax.set_xticks([])
...         ax.set_yticks([])
```

```

...
    if j == 0:
...
        ax.text(-0.06, 0.5, f'Epoch {e}',
                 rotation=90, size=18, color='red',
                 horizontalalignment='right',
                 verticalalignment='center',
                 transform=ax.transAxes)

...
    image = epoch_samples[e-1][j]
...
    ax.imshow(image, cmap='gray_r')
>>> plt.show()

```

Для визуализации результатов мы использовали тот же код, что и в разделе, посвященном простой GAN. Сравнение тех результатов с новыми показывает, что DCGAN может генерировать изображения гораздо более высокого качества.

У вас может возникнуть вопрос: как нам оценивать результаты работы генераторов GAN? Самый простой подход — визуальная оценка экспертом-человеком качества синтезированных изображений в контексте целевой области и цели проекта. Кроме того, было предложено несколько более сложных методов оценки, которые менее субъективны и менее ограничены знанием предметной области⁶.

Существует теоретический аргумент в пользу того, что обучение генератора должно стремиться минимизировать расхождение между распределениями, наблюдаемыми в реальных данных и в синтезированных примерах. Следовательно, наша текущая архитектура заведомо не будет работать очень хорошо при использовании перекрестной энтропии в качестве функции потерь.

В следующем подразделе мы рассмотрим архитектуру WGAN, которая использует для повышения эффективности обучения модифицированную функцию потерь, основанную на так называемом *расстоянии Вассерштейна-1* между распределениями реальных и фиктивных изображений.

17.3.4. Меры несходства между двумя распределениями

Сначала мы взглянем на различные показатели, используемые для вычисления расхождения между двумя распределениями. Затем вы узнаете, какой из этих показателей уже встроен в исходную модель GAN. Наконец, замена этого показателя в GAN приведет нас к реализации WGAN.

Как упоминалось в начале этой главы, цель генеративной модели — научиться синтезировать новые выборки, которые имеют то же распределение, что и набор обучающих данных. Пусть $P(x)$ и $Q(x)$ представляют распределение случайной величины x , как показано на рис. 17.18, справа.

Рассмотрим сейчас некоторые способы, показанные на рис. 17.18, слева, подходящие для измерения различий между распределениями P и Q .

⁶ Подробный обзор таких методов оценки приведен в статье «Pros and Cons of GAN Evaluation Measures: New Developments», <https://arxiv.org/abs/2103.09396>. В ней рассмотрена как количественная, так и качественная оценка.

Показатели	Математическая запись
Полная вариация (Total Variation, TV)	$TV(P, Q) = \sup_x P(x) - Q(x) $
Расхождение Кульбака — Лейблера (KL)	$KL(P Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Расхождение Дженсена — Шеннона (JS)	$JS(P, Q) = \frac{1}{2} \left(KL\left(P \frac{P+Q}{2}\right) + KL\left(Q \frac{P+Q}{2}\right) \right)$
Траншейное расстояние (Earth Mover, EM)	$EM(P, Q) = \inf_{\gamma \in \Pi(P, Q)} E_{(u, v) \in \gamma} (\ u - v\)$

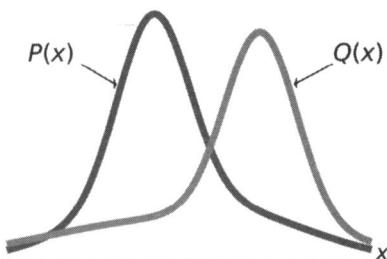


Рис. 17.18. Методы измерения различий между распределениями P и Q

Верхний предел (супремум) функции $\sup(S)$, используемый в показателе *полной вариации* (Total Variation, TV), относится к наименьшему значению, которое больше, чем все элементы S . Другими словами, $\sup(S)$ является наименьшей верхней границей для S . И наоборот, нижний предел (инфимум) функции $\inf(S)$, который присутствует в EM-расстоянии, относится к наибольшему значению, которое меньше всех элементов S (наибольшая нижняя граница).

Давайте разберемся в этих показателях, кратко указав, чего они пытаются достичь:

- ◆ первый из них: TV-расстояние, измеряет наибольшую разницу между двумя распределениями в каждой точке;
- ◆ EM-расстояние можно интерпретировать как минимальный объем работы, необходимый для «перемещения земли»⁷ — преобразования одного распределения в другое. Инфимум в EM-расстоянии берется по $\Pi(P, Q)$ — набору всех совместных распределений, чьи пределы равны P или Q . Тогда $\gamma(u, v)$ представляет собой план перемещения, который указывает, как мы перераспределяем «землю» из местоположения u в v — с учетом некоторых ограничений для поддержания действительных распределений после таких перемещений. Вычисление EM-расстояния само по себе является задачей оптимизации, заключающейся в поиске оптимального плана перемещения $\gamma(u, v)$;
- ◆ показатели расхождения Кульбака — Лейблера (KL) и Дженсена — Шеннона (JS) заимствованы из области теории информации. Обратите внимание, что расхождение KL, в отличие от расхождения JS, не является симметричным, т. е. $KL(P||Q) \neq KL(Q||P)$.

Уравнения несходства, представленные на рис. 17.18, соответствуют непрерывным распределениям, но могут быть расширены на дискретные случаи. Пример расчета раз-

⁷ На английском языке это расстояние называется *earth mover's distance* — расстояние землеройной машины (траншеекопателя). Для краткости мы будем называть его *траншейным расстоянием*. — Прим. пер.

личных показателей несходства с двумя простыми дискретными распределениями показан на рис. 17.19.

Обратите внимание, что в случае ЕМ-расстояния для этого простого примера $Q(x)$ при $x = 2$ имеет избыточное значение $0.5 - \frac{1}{3} = 0.166$, а значение Q при двух других значениях x меньше $\frac{1}{3}$. Следовательно, минимальный объем работы — это перенос значения при $x = 2$ в $x = 1$ и $x = 3$, как показано на рис. 17.19. На этом простом примере легко увидеть, что такие перемещения приведут к минимальному объему работы из всех возможных перемещений. Однако в более сложных случаях найти оптимальное решение иногда бывает невозможным.

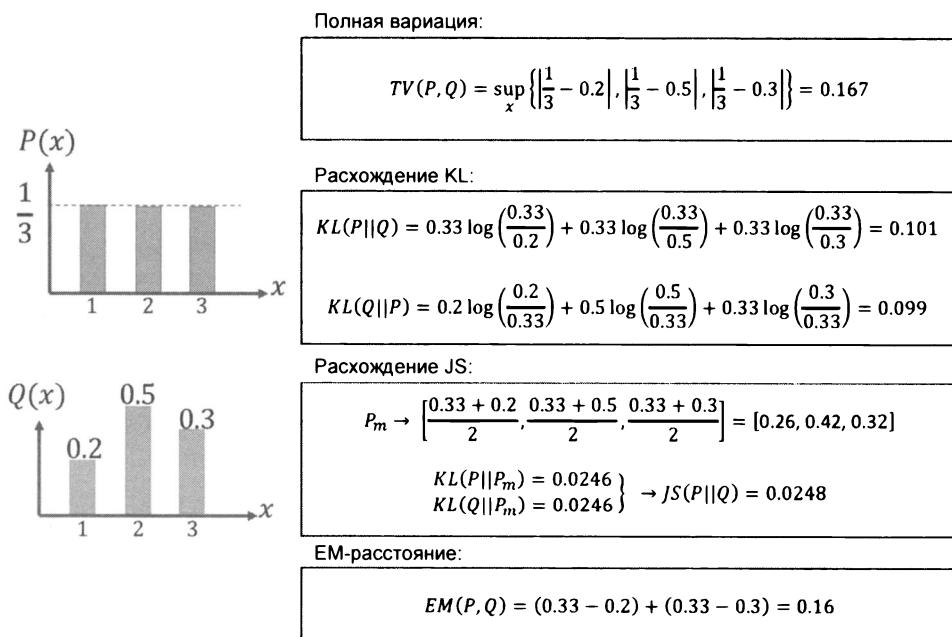


Рис. 17.19. Пример расчета различных показателей несходства



Связь между расхождением KL и перекрестной энтропией

Расхождение $KL(P||Q)$ измеряет энтропию распределения P по отношению к эталонному распределению Q . Формулу расхождения KL в развернутом виде можно записать так:

$$KL(P||Q) = - \int P(x) \log(Q(x)) dx - \left(- \int P(x) \log(P(x)) \right).$$

Кроме того, для дискретных распределений расхождение KL примет вид:

$$KL(P||Q) = - \sum_i P(x_i) \log \left(\frac{P(x_i)}{Q(x_i)} \right).$$

Эту формулу можно аналогично записать в развернутом виде:

$$KL(P||Q) = - \sum_i P(x_i) \log(Q(x_i)) - \left(- \sum_i P(x_i) \log(P(x_i)) \right).$$

Исходя из развернутой записи (дискретной или непрерывной), расхождение KL можно рассматривать как перекрестную энтропию между P и Q (первый член в предыдущем уравнении), вычтенную из собственной энтропии P (второй член), т. е.

$$KL(P||Q) = H(P, Q) - H(P).$$

Теперь, возвращаясь к нашему обсуждению GAN, давайте посмотрим, как эти различные меры расстояния связаны с функцией потерь для GAN. Можно математически показать, что функция потерь в оригинальной GAN действительно минимизирует расхождение JS между распределением реальных и фиктивных примеров. Но, как обсуждалось в статье Мартина Аржовски и его коллег⁸, расхождение JS имеет проблемы с обучением модели GAN, поэтому, чтобы улучшить обучение, исследователи предложили использовать EM-расстояние как меру несходства между распределением реальных и фиктивных примеров.



В чем преимущество использования EM-расстояния?

Чтобы ответить на этот вопрос, мы можем рассмотреть пример из упомянутой ранее статьи Мартина Аржовски и его коллег. Для простоты предположим, что у нас есть два распределения: P и Q , которые представляют собой две параллельные линии. Одна линия зафиксирована в точке $x = 0$, а другая может перемещаться вдоль оси x , но изначально находится в точке $x = \theta$, где $\theta > 0$.

Можно показать, что меры несходства KL, TV и JS равны соответственно:

$$KL(P||Q) = +\infty, TV(P, Q) = 1 \text{ и } JS(P, Q) = \frac{1}{2} \log 2. \text{ Ни одна из этих мер несходства не}$$

является функцией параметра θ , и, следовательно, их нельзя дифференцировать по параметру θ , чтобы сделать распределения P и Q похожими друг на друга. С другой стороны, EM-расстояние равно $EM(P, Q) = |\theta|$, градиент которого относительно θ существует и может способствовать смещению Q к P .

Теперь сосредоточимся на том, как можно использовать EM-расстояние для обучения модели GAN. Предположим, что P_r — это распределение реальных примеров, а P_g — распределение фиктивных (сгенерированных) примеров. Подставим P_r и P_g вместо P и Q в уравнение EM-расстояния. Как упоминалось ранее, вычисление EM-расстояния само по себе является задачей оптимизации — оно трудновыполнимо с вычислительной точки зрения, особенно если мы хотим повторять это вычисление на каждой итерации цикла обучения GAN. Однако, к счастью, вычисление EM-расстояния можно упростить с помощью *теоремы двойственности Канторовича — Рубинштейна*, а именно:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} E_{u \in P_r}[f(u)] - E_{v \in P_g}[f(v)].$$

Здесь верхняя граница берется по всем 1-липшицевым отображениям, обозначаемым $\|f\|_L \leq 1$.

⁸ См. «Wasserstein Generative Adversarial Networks», <http://proceedings.mlr.press/v70/arjovsky17a/arjovsky17a.pdf>.



Липшицево отображение

Основываясь на 1-липшицевом отображении, функция f должна удовлетворять следующему свойству:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|.$$

Кроме того, действительная функция $f: R \rightarrow R$, удовлетворяющая свойству:

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|,$$

называется *K-липшицевым отображением* (*K-Lipschitz continuous*).

17.3.5. Практическое применение ЕМ-расстояния для построения GAN

Теперь возникает вопрос, как нам найти такое 1-липшицево отображение для вычисления расстояния Вассерштейна между распределением реальных (P_r) и фиктивных (P_g) выходных данных для GAN? Хотя теоретические соображения, лежащие в основе подхода WGAN, поначалу выглядят сложными, ответ на этот вопрос проще, чем может показаться. Вспомните, что мы рассматриваем глубокие нейронные сети как универсальные аппроксиматоры функций. Это означает, что мы можем просто обучить нейросетевую модель аппроксимировать функцию расстояния Вассерштейна. Как вы видели в предыдущем разделе, простая GAN использует дискриминатор в форме классификатора. Для WGAN дискриминатор можно изменить, чтобы он вел себя как *критик*, который возвращает скалярную оценку вместо значения вероятности. Мы можем интерпретировать эту оценку как меру реалистичности входных изображений (как мнение искусствоведа, оценивающего произведения искусства в галерее).

При обучении GAN с использованием расстояния Вассерштейна потери для дискриминатора D и генератора G определяются следующим образом. Критик (т. е. сеть дискриминатора) возвращает свои выходные данные для пакета примеров реальных изображений и пакета синтезированных примеров. Обозначим их как $D(x)$ и $D(G(z))$ соответственно.

Тогда можно определить следующие составляющие потерь:

- ◆ потери дискриминатора на реальных примерах:

$$L_{\text{real}}^D = -\frac{1}{N} \sum_i D(\mathbf{x}_i);$$

- ◆ потери дискриминатора на фиктивных примерах:

$$L_{\text{fake}}^D = \frac{1}{N} \sum_i D(G(\mathbf{z}_i));$$

- ◆ потери генератора:

$$L^G = -\frac{1}{N} \sum_i D(G(\mathbf{z}_i)).$$

Это все, что требуется для WGAN, за исключением того, что нам нужно обеспечить сохранение свойства 1-липшицева переноса функции критика во время обучения. Для этого в статье про WGAN было предложено зафиксировать веса в небольшой области — например, $[-0.01, 0.01]$.

17.3.6. Штраф за градиент

В упомянутой ранее статье Аржовски и его коллег для сохранения свойства 1-липшицева переноса дискриминатора (или критика) предлагается отсечение веса. Однако в другой статье⁹ было показано, что отсечение весов может вызвать взрыв и исчезновение градиентов. Кроме того, отсечение веса также может привести к недостаточному использованию емкости модели, а это означает, что сеть критика ограничена изучением лишь некоторых простых, а не более сложных функций. Поэтому вместо того, чтобы урезать веса, Ишаан Гулраджани и его коллеги предложили в качестве альтернативного решения использовать *штраф за градиент* (Gradient Penalty, GP). В результате у нас получается *WGAN со штрафом за градиент* (WGAN-GP).

Процедуру GP, которая добавляется в каждый проход, можно кратко представить следующей последовательностью шагов:

1. Для каждой пары реальных и фиктивных примеров $(x^{[i]}, \tilde{x}^{[i]})$ в каждой партии выберите случайное число $\alpha^{[i]}$, извлеченное из равномерного распределения, т. е. $\alpha^{[i]} \in U(0, 1)$.
2. Вычислите интерполяцию между реальными и поддельными примерами $\bar{x}^{[i]} = \alpha x^{[i]} + (1 - \alpha)\tilde{x}^{[i]}$, в результате чего получится пакет интерполированных примеров.
3. Вычислите выход дискриминатора (критика) для всех интерполированных примеров $D(\bar{x}^{[i]})$.
4. Рассчитайте градиенты выхода критика по отношению к каждому интерполированному примеру, т. е. $\nabla_{\bar{x}^{[i]}} D(\bar{x}^{[i]})$.
5. Вычислите GP как

$$L_{gp}^D = \frac{1}{N} \sum_i \left(\left\| \nabla_{\bar{x}^{[i]}} D(\bar{x}^{[i]}) \right\|_2 - 1 \right)^2.$$

Тогда общие потери для дискриминатора будут следующими:

$$L_{\text{total}}^D = L_{\text{real}}^D + L_{\text{fake}}^D + \lambda L_{gp}^D.$$

Здесь λ — настраиваемый гиперпараметр.

17.3.7. Реализация WGAN-GP для обучения модели DCGAN

Мы уже определили вспомогательную функцию и класс, которые создают сети генератора и дискриминатора для DCGAN (`make_generator_network()` и `Discriminator()`). В WGAN рекомендуется использовать послойную нормализацию вместо пакетной. Нормализация по слоям нормализует входные данные по признакам, а не по пакетному измерению, как при пакетной нормализации. Код для построения модели WGAN выглядит следующим образом:

⁹ См. «Improved Training of Wasserstein GANs» by Ishaan Gulrajani et. al., 2017, <https://arxiv.org/pdf/1704.00028.pdf>.

```
>>> def make_generator_network_wgan(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4, 1, 0, bias=False),
...         nn.InstanceNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*4, n_filters*2, 3, 2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*2, n_filters, 4, 2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
...         nn.Tanh()
...     )
...     return model
>>>
>>> class DiscriminatorWGAN(nn.Module):
...     def __init__(self, n_filters):
...         super().__init__()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters, n_filters*2, 4, 2, 1, bias=False),
...             nn.InstanceNorm2d(n_filters * 2),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1, bias=False),
...             nn.InstanceNorm2d(n_filters*4),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...             nn.Sigmoid()
...         )
...
...     def forward(self, input):
...         output = self.network(input)
...         return output.view(-1, 1).squeeze(0)
```

Теперь мы можем инициализировать сети и их оптимизаторы:

```
>>> gen_model = make_generator_network_wgan(
...     z_size, n_filters
... ).to(device)
>>> disc_model = DiscriminatorWGAN(n_filters).to(device)
>>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0002)
>>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

Далее мы определим функцию для вычисления компонента GP:

```
>>> from torch.autograd import grad as torch_grad
>>> def gradient_penalty(real_data, generated_data):
...     batch_size = real_data.size(0)
...
...
...     # Вычисление интерполяции
...
...     alpha = torch.rand(real_data.shape[0], 1, 1, 1,
...                        requires_grad=True, device=device)
...     interpolated = alpha * real_data + (1 - alpha) * generated_data
...
...     # Вычисление вероятности интерполированных примеров
...     proba_interpolated = disc_model(interpolated)
...
...     # Вычисление градиентов вероятностей
...     gradients = torch_grad(
...         outputs=proba_interpolated, inputs=interpolated,
...         grad_outputs=torch.ones(proba_interpolated.size(), device=device),
...         create_graph=True, retain_graph=True
...     )[0]
...
...     gradients = gradients.view(batch_size, -1)
...     gradients_norm = gradients.norm(2, dim=1)
...     return lambda_gp * ((gradients_norm - 1)**2).mean()
```

WGAN-версия функций обучения дискриминатора и генератора выглядит так:

```
>>> def d_train_wgan(x):
...     disc_model.zero_grad()
...
...     batch_size = x.size(0)
...     x = x.to(device)
...
...     # Вычисление вероятностей реальных и сгенерированных данных
...     d_real = disc_model(x)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_generated = disc_model(g_output)
...     d_loss = d_generated.mean() - d_real.mean() + \
...     gradient_penalty(x.data, g_output.data)
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item()
>>>
>>> def g_train_wgan(x):
...     gen_model.zero_grad()
...
...     batch_size = x.size(0)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...
...
```

```

...
    d_generated = disc_model(g_output)
    g_loss = -d_generated.mean()

...
...
    # обратное распространение градиента и оптимизация
    # ТОЛЬКО параметров генератора
    g_loss.backward()
    g_optimizer.step()
...
    return g_loss.data.item()

```

Затем мы обучим модель в течение 100 эпох и запишем выходные данные генератора для фиксированного входного шума:

```

>>> epoch_samples_wgan = []
>>> lambda_gp = 10.0
>>> num_epochs = 100
>>> torch.manual_seed(1)
>>> critic_iterations = 5
>>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     d_losses, g_losses = [], []
...     for i, (x, _) in enumerate(mnist_dl):
...         for _ in range(critic_iterations):
...             d_loss = d_train_wgan(x)
...             d_losses.append(d_loss)
...             g_losses.append(g_train_wgan(x))
...
...     print(f'Эпохи {epoch:03d} | Потери D >>
...           f' {torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples_wgan.append(
...         create_samples(
...             gen_model, fixed_z
...         ).detach().cpu().numpy()
...     )
...

```

Наконец, выведем на экран сохраненные примеры для некоторых эпох — чтобы увидеть, как обучается модель WGAN и как меняется качество синтезированных примеров в ходе обучения. На рис. 17.20 показаны результаты, которые демонстрируют несколько лучшее качество изображения по сравнению с моделью DCGAN.

17.3.8. Схлопывание мод распределения

Из-за состязательного характера моделей GAN их, как известно, сложно обучать. Одной из частых причин сбоев при обучении GAN является то, что генератор застревает в небольшом подпространстве и учится генерировать примеры, похожие один на другой. Это явление, пример которого показан на рис. 17.21, называется *схлопыванием моды распределения* (mode collapse).

Синтезированные примеры на этом рисунке не являются случайными и независимыми. Это говорит о том, что генератор не смог изучить все распределение данных и вместо этого лениво сосредоточился на ограниченной области пространства.

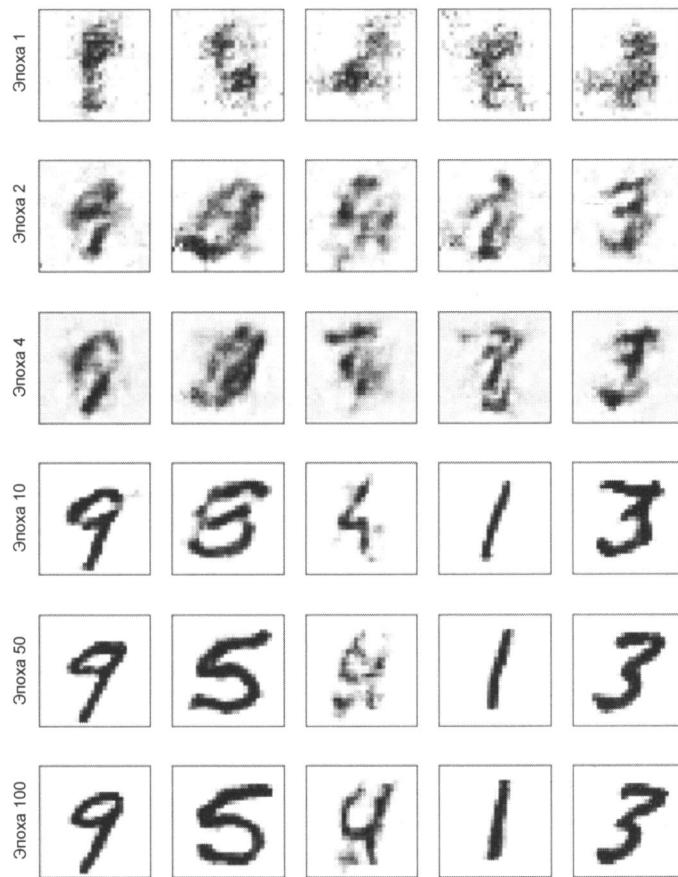


Рис. 17.20. Изображения, сгенерированные с использованием WGAN

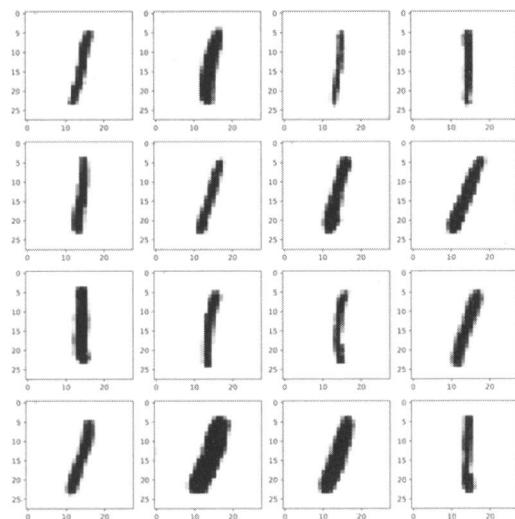


Рис. 17.21. Пример схлопывания моды распределения

Помимо проблем исчезающего и взрывающегося градиента, которые мы упоминали ранее, есть некоторые дополнительные аспекты, которые также могут затруднить обучение моделей GAN (на самом деле, это своего рода искусство). Вот несколько советов от «художников» GAN.

Один из подходов называется *мини-пакетной дискриминацией* (mini-batch discrimination) и основан на том факте, что пакеты, состоящие только из реальных или фиктивных примеров, подаются на дискриминатор по отдельности. При классификации мини-пакетов мы позволяем дискриминатору сравнивать примеры в этих пакетах, чтобы определить, является ли пакет реальным или фиктивным. Разнообразие пакета, состоящего только из реальных примеров, скорее всего, выше, чем разнообразие пакета фиктивных примеров, особенно если модель страдает схлопыванием моды.

Другой метод, который обычно используется для стабилизации обучения GAN, — это *сопоставление признаков* (feature matching). При сопоставлении признаков мы вносим небольшие изменения в целевую функцию генератора, добавляя дополнительный член, который минимизирует разницу между исходными и синтезированными изображениями на основе промежуточных представлений (карт признаков) дискриминатора¹⁰.

Во время обучения модель GAN также может застрять в нескольких модах и просто переключаться между ними. Чтобы избежать такого поведения, вы можете сохранить некоторые старые примеры и передать их дискриминатору, чтобы генератор не возвращался к предыдущим модам. Этот метод называется *воспроизведением опыта* (experience replay). Кроме того, можно обучать несколько GAN с разными случайными начальными значениями, чтобы их комбинация покрывала более обширную область данных, чем любая из них по отдельности.

17.4. Другие применения GAN

В этой главе мы в основном сосредоточились на создании примеров изображений с использованием GAN и рассмотрели несколько приемов и методов для улучшения качества синтезированных выходных данных. Диапазон применений GAN быстро расширяется, в том числе в компьютерном зрении, машинном обучении и даже в других областях науки и техники. Хороший список различных моделей GAN и областей применения можно найти здесь: <https://github.com/hindupuravinash/the-gan-zoo>.

Стоит также отметить, что мы рассматривали обучение GAN без учителя, — т. е. в моделях, описанных в этой главе, не использовалась информация о метках классов. Однако методологию GAN можно распространить и на частичное или полное обучение с учителем. Например, *условная GAN* (conditional GAN, cGAN)¹¹ использует информацию о метках классов и учится синтезировать новые изображения, обусловленные предоставленной меткой, т. е. $\tilde{x} = G[z | y]$ — применительно к MNIST. Это позволяет нам выборочно генерировать разные цифры в диапазоне от 0 до 9. Кроме того, условные

¹⁰ Мы рекомендуем вам прочитать больше об этом методе в статье Тинг-Чун Вана и его коллег под названием «High Resolution Image Synthesis and Semantic Manipulation with Conditional GANs», которая свободно доступна по адресу: <https://arxiv.org/pdf/1711.11585.pdf>.

¹¹ Предложена Мехди Мирзой и Саймоном Осиндеро в статье «Conditional Generative Adversarial Nets», 2014, <https://arxiv.org/pdf/1411.1784.pdf>.

GAN дают возможность выполнять преобразование изображения в изображение, т. е. научиться преобразовывать входное изображение из одной предметной области в другую. В этом контексте заслуживает внимания алгоритм Pix2Pix, представленный в статье Филипа Изола и его коллег¹². Стоит отметить, что в алгоритме Pix2Pix дискriminator выдает прогнозы «реальность/подделка» для нескольких участков изображения, а не один прогноз для всего изображения.

CycleGAN — еще одна интересная модель GAN, построенная на основе cGAN и также предназначена для преобразования изображения в изображение. Однако в CycleGAN обучающие примеры из двух областей непарные, а это означает, что между входными и выходными данными нет однозначного соответствия. Например, с помощью CycleGAN мы могли бы изменить время года на фотоснимке с лета на зиму. В статье Джун-Ян Чжу и коллег «Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks», 2020 г. (<https://arxiv.org/pdf/1703.10593.pdf>), приведен впечатляющий пример того, как лошади превращаются в зебр.

17.5. Заключение

В этой главе вы сначала узнали о генеративных моделях глубокого обучения и их общей цели: синтезе новых данных. Затем мы увидели, как модели GAN используют сети генератора и дискриминатора, конкурирующие друг с другом в условиях состязательного обучения, чтобы взаимно улучшать свое качество, и реализовали простую модель GAN, используя только полносвязные слои как для генератора, так и для дискриминатора.

Мы также рассмотрели способы улучшения модели GAN. Во-первых, освоили модель DCGAN, которая использует глубокие сверточные сети как для генератора, так и для дискриминатора. Попутно ознакомились с двумя новыми понятиями: транспонированной свертки (для повышения пространственной размерности карт объектов) и пакетной нормализации BatchNorm (для улучшения сходимости во время обучения).

В завершение мы изучили модель WGAN, которая использует ЕМ-расстояние для измерения расхождения между распределениями реальных и фиктивных выборок, и обсудили WGAN с применением GP вместо отсечки весов для сохранения 1-липшицевости.

В следующей главе мы рассмотрим графовые нейронные сети. Ранее мы работали с наборами данных в виде таблиц и изображений. Графовые нейронные сети предназначены для данных, структурированных в виде графов, что позволяет нам работать с наборами данных, которые широко распространены в социальных науках, технике и биологии. К распространенным примерам данных графовой структуры относятся графы социальных сетей и молекулы, состоящие из атомов, соединенных ковалентными связями.

¹² См. «Image-to-Image Translation with Conditional Adversarial Networks», 2018, <https://arxiv.org/pdf/1611.07004.pdf>.

18

Графовые нейронные сети: выявление зависимостей в структурированных графовых данных

В этой главе вы познакомитесь с особым классом моделей глубокого обучения, предназначенных для работы с графовыми данными, — с *графовыми нейронными сетями* (Graph Neural Networks, GNN). Эти сети особенно быстро развивались в последние годы. Согласно отчету о состоянии ИИ за 2021 год¹, GNN превратились «из нишевой темы в одну из самых популярных областей исследований ИИ».

GNN нашли применение в различных областях, в том числе следующих:

- ◆ классификация текстов (<https://arxiv.org/abs/1710.10903>);
- ◆ рекомендательные системы (<https://arxiv.org/abs/1704.06803>);
- ◆ прогнозирование трафика (<https://arxiv.org/abs/1707.01926>);
- ◆ разработка лекарств (<https://arxiv.org/abs/1806.02473>).

В этой быстро развивающейся области невозможно подробно описать каждую идею, но вы прочно усвоите основные принципы работы GNN и основные приемы их реализации. Кроме того, вы познакомитесь с библиотекой PyTorch Geometric, которая позволяет оперировать графовыми данными для глубокого обучения, а также с реализациями различных типов графовых слоев, которые вы сможете использовать в своих моделях глубокого обучения.

В этой главе будут представлены:

- ◆ введение в графовые данные и способы их представления для использования в глубоких нейронных сетях;
- ◆ объяснение графовых сверток — основного функционального компонента обычных GNN;
- ◆ пример разработки GNN для предсказания свойств молекул с помощью PyTorch Geometric;
- ◆ обзор передовых методов в области GNN.

¹ См. <https://www.stateof.ai/2021-report-launch.html>.

18.1. Введение в графовые данные

В широком смысле графы представляют собой определенный способ описания и представления взаимосвязей в данных. *Графы* — это особый тип структуры данных, который является нелинейным и абстрактным. А поскольку графы являются абстрактными объектами, необходимо определить конкретное представление, чтобы с графиками можно было работать. Кроме того, графы могут иметь определенные свойства, поэтому нам могут понадобиться разные представления. На рис. 18.1 в обобщенном виде показаны распространенные типы графов, которые мы обсудим более подробно в этой главе.

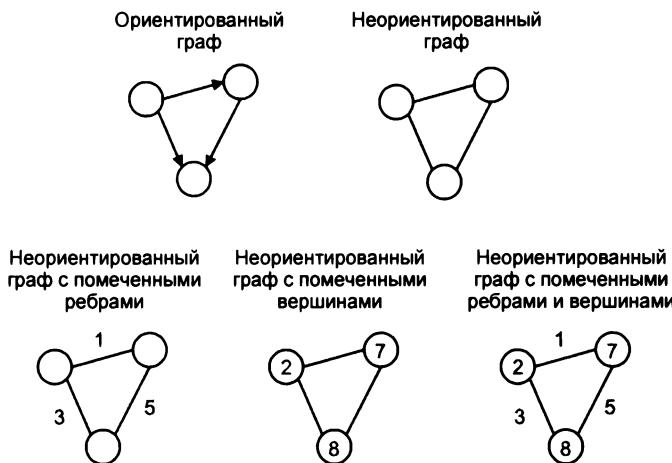


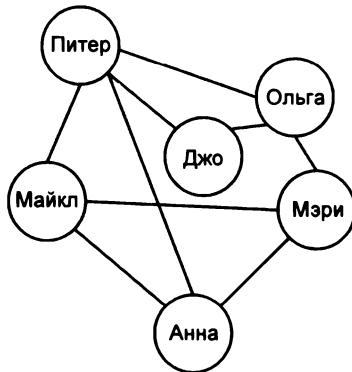
Рис. 18.1. Наиболее распространенные типы графов

18.1.1. Неориентированные графы

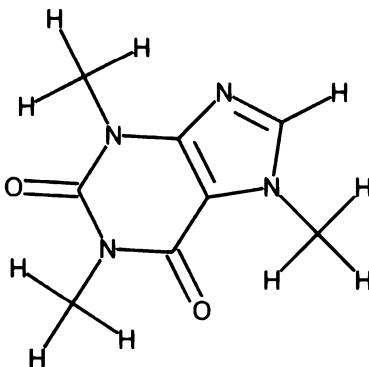
Неориентированный граф (undirected graph) состоит из узлов (node), в теории графов также часто называемых *вершинами* (verticle), соединенных ребрами, где порядок узлов и их связь не имеют значения. На рис. 18.2 показаны два типичных примера неориентированных графов: граф дружеских отношений и граф химической молекулы, состоящей из атомов, соединенных химическими связями (более подробно такие молекулярные графы мы обсудим в следующих разделах).

К другим распространенным примерам данных, которые могут быть представлены в виде неориентированных графов, относятся изображения, сети межблковых взаимодействий и облака точек.

С математической точки зрения неориентированный граф G — это пара (V, E) , где V — множество узлов графа, а E — множество ребер, составляющих парные узлы. Следовательно, граф можно представить как *матрицу смежности* (adjacency matrix) A размером $|V| \times |V|$. Каждый элемент x_{ij} в матрице A равен либо 1, либо 0, где 1 обозначает ребро между узлами i и j (и наоборот, 0 обозначает отсутствие ребра). Поскольку граф неориентированный, дополнительным свойством матрицы A является то, что $x_{ij} = x_{ji}$.



Граф дружеских отношений



Граф молекулы кофеина

Рис. 18.2. Два примера неориентированных графов

18.1.2. Ориентированные графы

В ориентированных графах (directed graph), в отличие от неориентированных, обсуждавшихся в предыдущем подразделе, узлы соединяются ориентированными (направленными) ребрами. Математически они определяются так же, как и неориентированные графы, за исключением того, что множество ребер E представляет собой набор упорядоченных пар. Следовательно, элемент x_{ij} из матрицы A может не равняться x_{ji} .

Примером ориентированного графа может служить сеть цитирования, где узлами являются публикации, а ребра от узлов направлены к узлам статей, на которые ссылается та или иная статья.

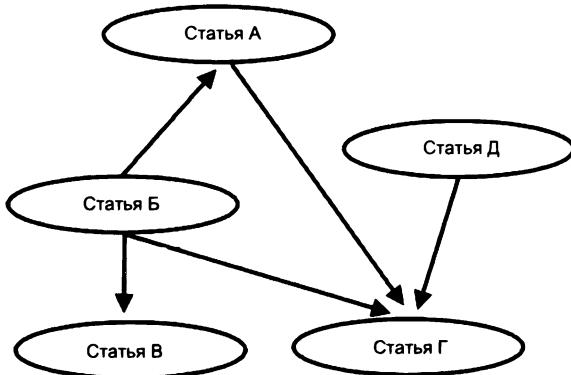


Рис. 18.3. Пример ориентированного графа

18.1.3. Помеченные графы

Многие графы, с которыми нам приходится встречаться на практике, имеют дополнительную информацию, связанную с каждым из их узлов и ребер. Например, если взять молекулу кофеина, показанную на рис. 18.2, то ее можно представить в виде графа, где

каждый узел — это атом химического элемента (например, атом кислорода — O, углерода — C, азота — N или водорода — H), а каждое ребро определяет тип связи (например, одинарная или двойная связь) между двумя его узлами. Эти признаки узлов и ребер должны быть закодированы, что требует хранилища информации нужной емкости. Для заданного графа G , определенного множествами узлов и ребер (V, E), мы определяем матрицу X признаков узлов $|V| \times f_V$, где f_V — длина вектора меток каждого узла. А для меток ребер определяем матрицу X_E признаков ребер $|E| \times f_E$, где f_E — длина вектора меток каждого ребра.

Молекулы — отличный пример данных, которые можно представить в виде *помеченного графа*, поэтому мы будем работать с молекулярными данными на протяжении всей этой главы. И сейчас мы воспользуемся возможностью, чтобы в следующем подразделе подробно рассмотреть графовое представление молекул.

18.1.4. Представление молекул в виде графов

С точки зрения химии молекулы можно рассматривать как группы атомов, удерживающих вместе химическими связями. Существуют разные атомы, представляющие собой разные химические элементы — например, к наиболее распространенным элементам относятся углерод (C), кислород (O), азот (N) и водород (H). Также существуют различные виды связей между атомами — например, одинарные или двойные связи.

Мы можем рассматривать молекулу как неориентированный граф с матрицей меток узлов, где каждая строка представляет собой унитарный код типа атома соответствующего узла. Кроме того, существует матрица меток ребер, где каждая строка представля-

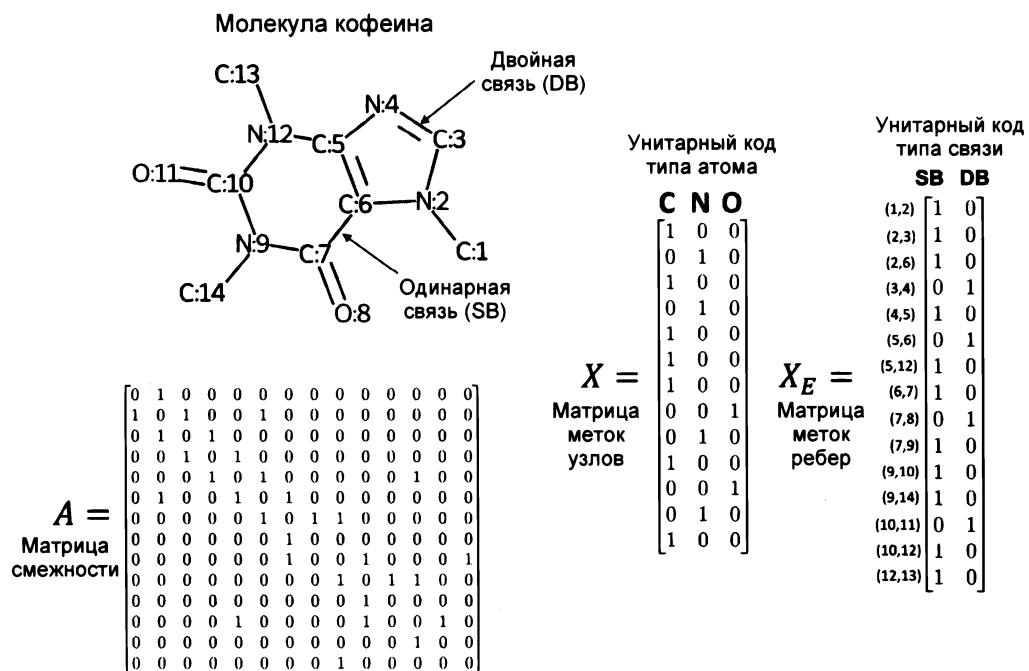


Рис. 18.4. Графовое представление молекулы кофеина

ет собой унитарный код типа связи соответствующего ребра. Чтобы упростить это представление, атомы водорода иногда делают неявными, поскольку их местоположение можно вывести с помощью основных химических правил. Пример графового представления молекулы кофеина с неявными атомами водорода показан на рис. 18.4.

18.2. Что такое графовая свертка?

В предыдущем разделе были показаны различные способы представления графовых данных. Следующим логическим шагом будет обсуждение доступных инструментов, которые могут эффективно использовать эти представления.

В этом разделе мы рассмотрим графовые свертки, которые являются ключевым компонентом для построения GNN. Вы узнаете, почему нужно применять механизм свертки к графикам, и какие атрибуты должны иметь эти свертки. Затем вы познакомитесь с примером реализации графовой свертки.

18.2.1. Причина использования графовых сверток

Чтобы лучше разобраться в графовых свертках, давайте кратко вспомним, как свертки используются в сверточных нейронных сетях (CNN), которые мы обсуждали в главе 14. В контексте изображений мы можем рассматривать свертку как процесс скольжения сверточного фильтра по изображению, где на каждом шаге вычисляется взвешенная сумма между фильтром и рецептивным полем (частью изображения, которую он перекрывает).

Как было отмечено в главе 14, фильтр можно рассматривать как детектор определенного признака. Этот подход к обнаружению признаков хорошо пригоден для изображений по нескольким причинам, связанным с характерными особенностями размещения изображения на априорных данных:

- Инвариантность к сдвигу:** мы можем распознать признак на изображении независимо от того, где он находится (например, после сдвига). Кошку можно распознать как кошку независимо от того, находится ли она в левом верхнем углу, правом нижнем углу или в другой части изображения.
- Локальность:** соседние пиксели тесно связаны между собой.
- Иерархия:** большие части изображения часто можно разбить на комбинации связанных меньших частей: у кошки есть голова и ноги, на голове присутствуют глаза и нос, глаза имеют зрачки и радужную оболочку.

Заинтересованные читатели могут найти более формальное описание этих априорных значений изображений и априорных значений, принимаемых GNN, в опубликованной в 2019 году статье².

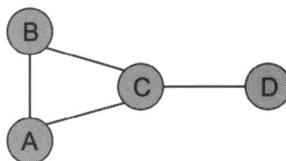
Еще одна причина, по которой свертки хорошо подходят для обработки изображений, заключается в том, что количество обучаемых параметров не зависит от размерности входных данных. Вы можете обучить серию сверточных фильтров 3×3 , например, на изображении 256×256 или 9×9 . (Однако, если одно и то же изображение представлено

² См. «Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology» by N. Dehmamy, A.-L. Barabasi, and R. Yu, <https://arxiv.org/abs/1907.05008>.

в разных разрешениях, рецептивные поля и, следовательно, извлекаемые признаки будут различаться. А для эффективного извлечения полезных признаков из изображений с более высоким разрешением нам может понадобиться выбрать более крупные ядра или добавить дополнительные слои.)

Как и изображения, графы также содержат естественные априорные признаки, которые оправдывают сверточный подход. Оба вида данных: изображения и графы — имеют локализованные признаки, однако у них различается определение локальности: в изображениях преобладает локальность в двумерном пространстве, а в графах — структурная локальность. Например, интуитивно понятно, что узлы, которые расположены на расстоянии одного ребра, с большей вероятностью будут связаны между собой, чем узлы, удаленные на пять ребер. Так, можно допустить, что на графе цитирования непосредственно цитируемая публикация, которая находится на расстоянии одного ребра, с большей вероятностью будет иметь аналогичную тематику, чем публикация, удаленная на несколько ребер.

Строгим априорным фактором для данных графа является инвариантность перестановок — т. е. порядок узлов не влияет на результат. Это показано на рис. 18.5, где изменение порядка узлов графа не меняет структуры графа.



Матрица смежности 1: Матрица смежности 2: Матрица смежности 3:

$$\begin{array}{l}
 \text{Матрица смежности 1:} \\
 \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \\
 \text{Матрица смежности 2:} \\
 \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \\
 \text{Матрица смежности 3:} \\
 \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
 \end{array}$$

Рис. 18.5. Различные матрицы смежности, представляющие один и тот же граф

Поскольку один и тот же граф, как показано на рис. 18.5, может быть представлен несколькими матрицами смежности, следовательно, любая графовая свертка должна быть инвариантной к перестановкам.

Сверточный подход также желателен для графов, поскольку он может работать с фиксированным набором параметров для графов разных размеров. Возможно, это свойство даже важнее для графов, чем для изображений. Например, существует множество наборов данных изображений с фиксированным разрешением, где возможно применение полно связных сетей (например, с использованием многослойного персептрона), как было показано в главе 11. Напротив, большинство наборов графовых данных содержат графы разных размеров.

Хотя операторы свертки изображений стандартизированы, существует множество различных видов графовых сверток, и разработка новых сверток сейчас стала очень активной областью исследований. Мы сосредоточимся на предоставлении общих идей, чтобы читатели могли усовершенствовать собственные GNN. С этой целью в следующем

подразделе будет показана реализация базовой графовой свертки в PyTorch. Затем мы создадим простую GNN в PyTorch с нуля.

18.2.2. Реализация базовой свертки графа

В этом подразделе мы рассмотрим базовую функцию графовой свертки и разберемся в том, что произойдет, когда она будет применена к графу. Взгляните на граф и его представление, показанные на рис. 18.6.

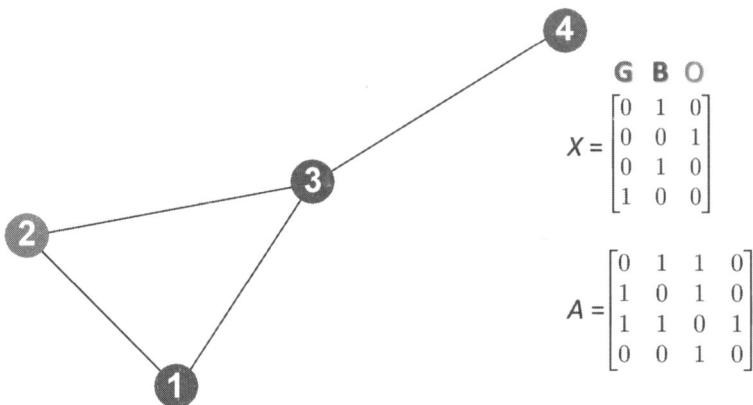


Рис. 18.6. Представление графа

Здесь изображен неориентированный граф с метками узлов, заданными матрицей смежности $n \times n$ A и матрицей признаков узлов $n \times f_m$ X , где единственным признаком является унитарное представление цвета каждого узла: зеленый (G), синий (B) или оранжевый (O).

Одной из самых универсальных библиотек для обработки и визуализации графов является NetworkX³, которой мы и воспользуемся, чтобы проиллюстрировать, как строить графы из матрицы меток X и матрицы узлов A .



Установка NetworkX

NetworkX — это удобная библиотека Python для работы с графами и их визуализации. Ее можно установить через pip:

```
pip install networkx
```

Дополнительную информацию вы найдете на официальном веб-сайте библиотеки: <https://networkx.org>.

Обратившись к NetworkX, мы можем построить граф, показанный на рис. 18.6:

```
>>> import numpy as np
>>> import networkx as nx
>>> G = nx.Graph()
... # Шестнадцатеричные представления цветов в графе
```

³ Для визуального представления графов в этой главе мы использовали версию библиотеки 2.6.2.

```

>>> blue, orange, green = "#1f77b4", "#ff7f0e", "#2ca02c"
>>> G.add_nodes_from([
...     (1, {"color": blue}),
...     (2, {"color": orange}),
...     (3, {"color": blue}),
...     (4, {"color": green})
... ])
>>> G.add_edges_from([(1,2), (2,3), (1,3), (3,4)])
>>> A = np.asarray(nx.adjacency_matrix(G).todense())
>>> print(A)
[[0 1 1 0]
 [1 0 1 0]
 [1 1 0 1]
 [0 0 1 0]]


>>> def build_graph_color_label_representation(G, mapping_dict):
...     one_hot_idxs = np.array([mapping_dict[v] for v in
...         nx.get_node_attributes(G, 'color').values()])
>>> one_hot_encoding = np.zeros(
...     (one_hot_idxs.size, len(mapping_dict)))
>>> one_hot_encoding[
...     np.arange(one_hot_idxs.size), one_hot_idxs] = 1
>>> return one_hot_encoding
>>> X = build_graph_color_label_representation(
...     G, {green: 0, blue: 1, orange: 2})
>>> print(X)
[[0., 1., 0.],
 [0., 0., 1.],
 [0., 1., 0.],
 [1., 0., 0.]]

```

Чтобы вывести на экран график, построенный с помощью этого кода, надо выполнить следующие команды:

```

>>> color_map = nx.get_node_attributes(G, 'color').values()
>>> nx.draw(G, with_labels=True, node_color=color_map)

```

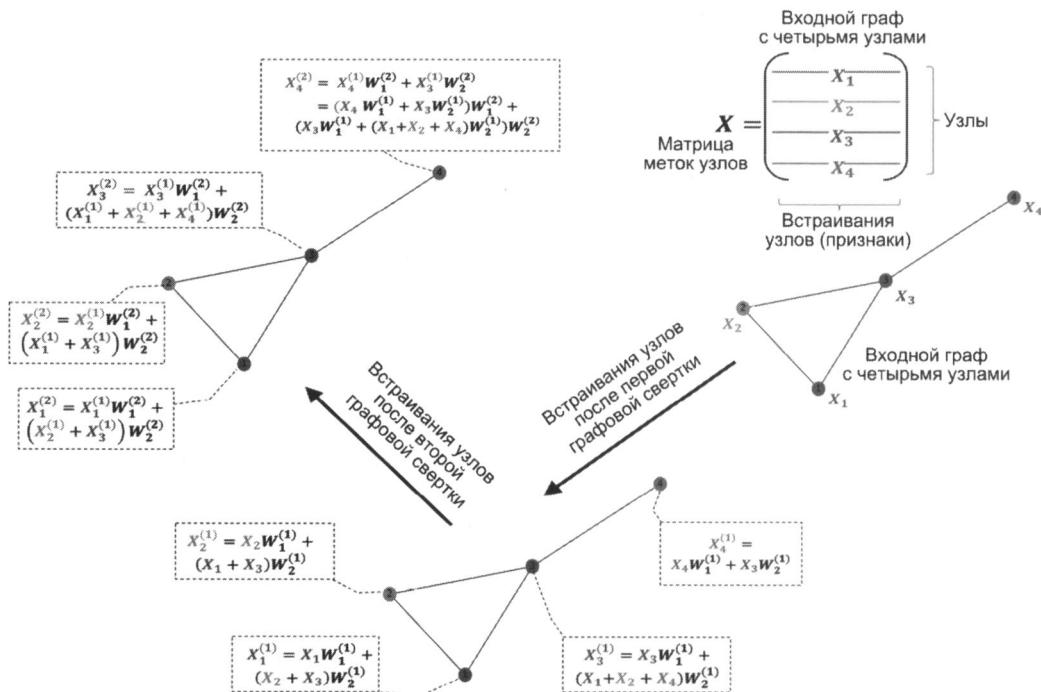
В приведенном примере кода мы сначала инициировали новый объект `Graph` из библиотеки `NetworkX`. Затем добавили узлы с 1-го по 4-й вместе с определениями цветов для визуализации. Добавив узлы, мы указали их соединения (ребра). Используя конструктор `adjacency_matrix` из `NetworkX`, мы создали матрицу смежности A , а наша пользовательская функция `build_graph_color_label_representation` создала матрицу меток узла X из информации, которую мы ранее добавили в объект `Graph`.

С помощью графовых сверток мы можем интерпретировать каждую строку X как встраивание информации, которая хранится в узле, соответствующем этой строке. Графовые свертки обновляют встраивания в каждом узле на основе встраиваний их соседей и самого этого узла. Для нашего примера графовая свертка примет следующий вид:

$$x'_i = x_i w_1 + \sum_{j \in N(i)} x_j w_2 + b.$$

Здесь x'_i — обновленное встраивание для узла i , W_1 и W_2 — матрицы $f_{in} \times f_{out}$ обучаемых весов фильтров, а b представляет собой обучаемый вектор смещения длиной f_{out} .

Обе матрицы весов: W_1 и W_2 — можно рассматривать как банки фильтров, где каждый столбец является отдельным фильтром. Заметим, что эта структура фильтра наиболее эффективна, когда необходимо сохранить априорную информацию в графовых данных. Если значение в одном узле сильно коррелирует со значением в другом узле, удаленном на много ребер, одиночная свертка не зафиксирует эту связь. Стек из нескольких сверток способен охватывать более отдаленные отношения, как показано на рис. 18.7 (для простоты мы установили смещение равным нулю).



Приведенная схема графовой свертки соответствует нашим априорным предположениям о графовых данных, но может быть неясно, как реализовать суммирование по соседям в матричной форме. Здесь мы используем матрицу смежности A . Матричная форма этой свертки: $XW_1 + AXW_2$. Матрица смежности, состоящая из 1 и 0, действует как маска для выбора узлов и вычисления желаемых сумм. В NumPy инициализацию этого слоя и вычисление прямого прохода на рассматриваемом граfe можно записать следующим образом:

```
>>> f_in, f_out = X.shape[1], 6
>>> W_1 = np.random.rand(f_in, f_out)
>>> W_2 = np.random.rand(f_in, f_out)
>>> h = np.dot(X, W_1) + np.dot(np.dot(A, X), W_2)
```

Вычисление прямого прохода графовой свертки отличается своей простотой.

В конечном счете нам нужно, чтобы сверточный слой графа обновлял представление закодированной в X информации об узле, используя структурную (связную) информацию, предоставленную A . Есть много потенциальных способов сделать это, что нашло отражение в многочисленных типах графовых сверток.

Если разнообразных графовых сверток так много, было бы неплохо найти для них объединяющую основу. К счастью, такая основа появилась⁴.

Согласно принципу *передачи сообщений* (message-passing) каждый узел в графе имеет связанное с ним скрытое состояние $h_i^{(t)}$, где i — индекс узла на временному шагу t . Начальное значение $h_i^{(0)}$ определяется как X_i , которое представляет собой строку X , связанную с узлом i .

Каждая графовая свертка может быть разделена на фазу передачи сообщений и фазу обновления узла. Пусть $N(i)$ — соседи узла i . Для неориентированных графов $N(i)$ — это множество узлов, которые имеют общее ребро с узлом i . Для ориентированных графов $N(i)$ — это множество узлов, у которых есть ребро, конечная точка которого — узел i . Фазу передачи сообщений можно записать в виде следующего выражения:

$$m_i = \sum_{j \in N(i)} M_t(h_i^{(t)}, h_j^{(t)}, e_{ij}).$$

Здесь M_t — функция сообщения. В нашем демонстрационном слое мы определяем эту функцию сообщения так: $M_t = h_j^{(t)} W_2$. Фаза обновления узла с функцией обновления U_t , записывается так: $h_i^{(t+1)} = U_t(h_i^{(t)}, m_i)$. Для нашего слоя это обновление принимает вид:

$$h_i^{(t+1)} = h_i^{(t)} W_1 + m_i + b.$$

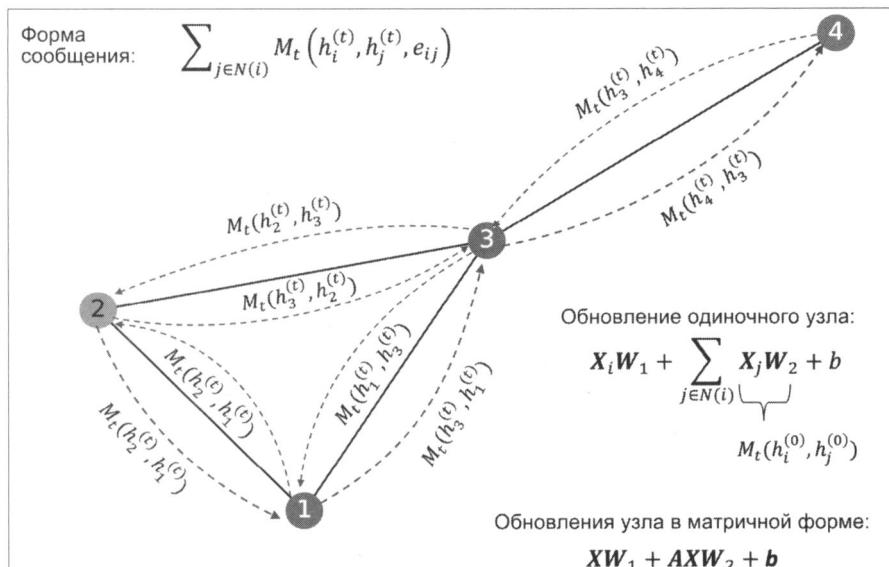


Рис. 18.8. Свертки, показанные на графике и в форме сообщения

⁴ См. книгу «Neural Message Passing for Quantum Chemistry» by Justin Gilmer and colleagues, 2017, <https://arxiv.org/abs/1704.01212>.

На рис. 18.8 схематически представлена идея передачи сообщений и показана реализованная нами свертка.

В следующем разделе мы включим этот слой графовой свертки в модель GNN, реализованную в PyTorch.

18.3. Построение GNN в PyTorch с нуля

Предыдущий раздел был посвящен изучению основ операции графовой свертки. В этом разделе вы познакомитесь с базовой реализацией графовой нейронной сети и научитесь применять эти методы к графикам, начиная с самых первых шагов. Если реализация графовых сетей и графовой свертки вызовет у вас затруднения, не волнуйтесь — GNN и в самом деле являются относительно сложными для реализации моделями. В следующем разделе мы рассмотрим пакет PyTorch Geometric, который предоставляет инструменты для упрощения реализации и управления данными для графовых нейронных сетей.

18.3.1. Определение модели *NodeNetwork*

В этом разделе мы займемся разработкой GNN с нуля с помощью PyTorch. Мы воспользуемся подходом «сверху вниз», начиная с основной модели нейронной сети, которую мы называем *NodeNetwork*, а затем добавим недостающие детали:

```
import networkx as nx
import torch
from torch.nn.parameter import Parameter
import numpy as np
import math
import torch.nn.functional as F

class NodeNetwork(torch.nn.Module):
    def __init__(self, input_features):
        super().__init__()
        self.conv_1 = BasicGraphConvolutionLayer (
            input_features, 32)
        self.conv_2 = BasicGraphConvolutionLayer(32, 32)
        self.fc_1 = torch.nn.Linear(32, 16)
        self.out_layer = torch.nn.Linear(16, 2)

    def forward(self, X, A, batch_mat):
        x = F.relu(self.conv_1(X, A))
        x = F.relu(self.conv_2(x, A))
        output = global_sum_pool(x, batch_mat)
        output = self.fc_1(output)
        output = self.out_layer(output)
        return F.softmax(output, dim=1)
```

Модель *NodeNetwork*, которую мы только что определили, можно кратко описать следующим образом:

1. Выполняем две графовые свертки: `self.conv_1` и `self.conv_2`.
2. Объединяем все встраивания узлов в слое `global_sum_pool`, который мы определим позже.
3. Пропускаем объединенные встраивания через два полносвязных слоя: `self.fc_1` и `self.out_layer`.
4. Выводим вероятность принадлежности к классу через `softmax`.

Структура сети с пояснениями того, что делает каждый уровень, представлена на рис. 18.9.

Отдельные аспекты, такие как слои графовой свертки и глобального объединения, будут рассмотрены в следующих разделах.

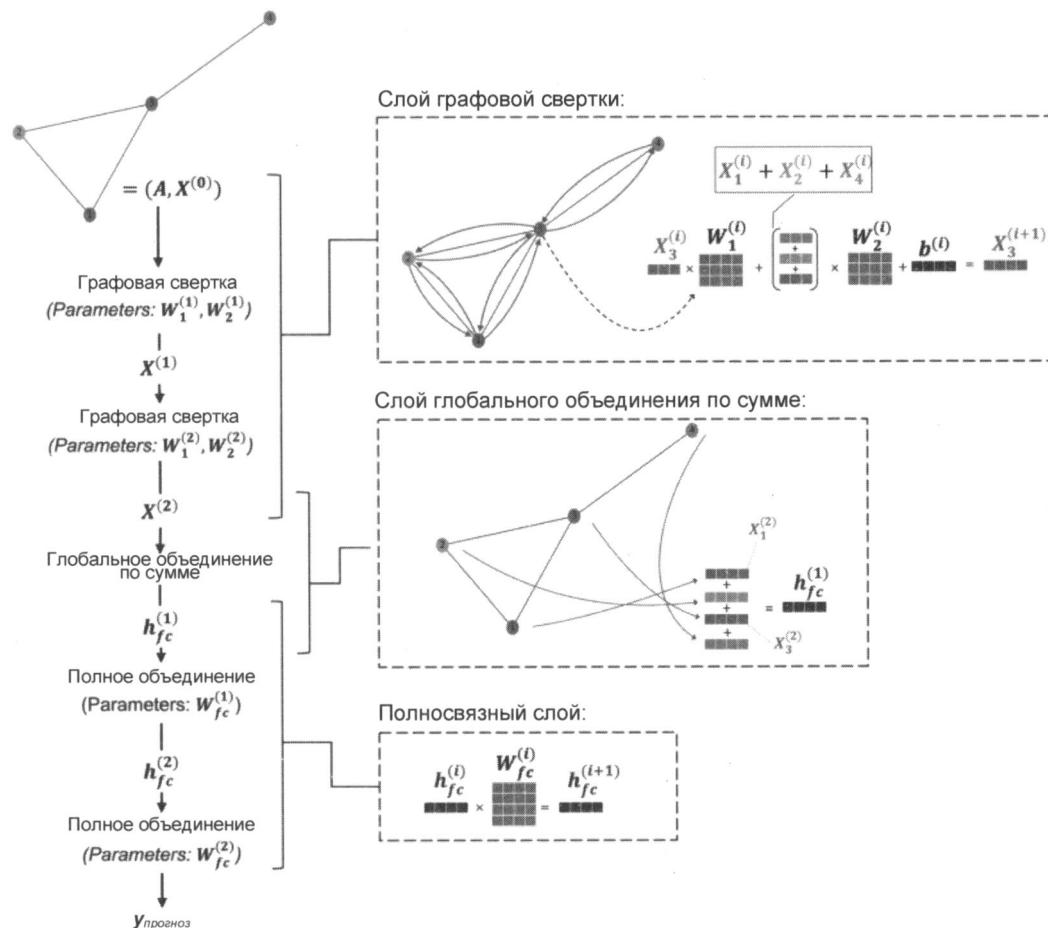


Рис. 18.9. Устройство и принцип работы простой графовой нейронной сети

18.3.2. Реализация слоя графовой свертки *NodeNetwork*

Теперь определим операцию графовой свертки `BasicGraphConvolutionLayer`, которая использовалась внутри уже знакомого вам класса `NodeNetwork`:

```
class BasicGraphConvolutionLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.W2 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))
        self.W1 = Parameter(torch.rand(
            (in_channels, out_channels), dtype=torch.float32))

        self.bias = Parameter(torch.zeros(
            out_channels, dtype=torch.float32))
    def forward(self, X, A):
        potential_msgs = torch.mm(X, self.W2)
        propagated_msgs = torch.mm(A, potential_msgs)
        root_update = torch.mm(X, self.W1)
        output = propagated_msgs + root_update + self.bias
        return output
```

Как и в случае полносвязных слоев и сверточных слоев для работы с изображениями, мы добавляем член смещения, чтобы иметь возможность варьировать пересечение линейной комбинации выходных данных слоя (до применения нелинейности, такой как `ReLU`). Метод `forward()` реализует матричную форму прямого прохода, которую мы обсуждали в предыдущем подразделе, но с добавлением члена смещения.

Чтобы опробовать `BasicGraphConvolutionLayer`, применим его к графу и матрице смежности, которые мы определили ранее в разд. 18.2.2:

```
>>> print('X.shape:', X.shape)
X.shape: (4, 3)

>>> print('A.shape:', A.shape)
A.shape: (4, 4)

>>> basiclayer = BasicGraphConvolutionLayer(3, 8)
>>> out = basiclayer(
...     X=torch.tensor(X, dtype=torch.float32),
...     A=torch.tensor(A, dtype=torch.float32)
... )

>>> print('Размер вывода:', out.shape)
Размер вывода: torch.Size([4, 8])
```

Как следует из вывода, наш `BasicGraphConvolutionLayer` преобразовал граф с четырьмя узлами, состоящий из трех признаков, в представление с восемью признаками.

18.3.3. Добавочный слой глобального объединения для работы с графами разного размера

Теперь определим функцию `global_sum_pool()`, которая использовалась в классе `NodeNetwork`, где она реализует слой глобального объединения. Слои глобального объединения объединяют все встраивания узлов графа в выходные данные фиксированного размера. Как показано на рис. 18.9, функция `global_sum_pool()` суммирует все встраивания узлов графа. Можно заметить, что это глобальное объединение относительно похоже на глобальное объединение по среднему в CNN, которое используется до того, как данные будут пропущены через полно связанные слои, как мы видели в главе 14.

Суммирование всех встраиваний узлов приводит к потере информации, поэтому изменение размерности данных было бы предпочтительнее, но, поскольку графы могут иметь непредсказуемо разные размеры, это невозможно. Глобальное объединение может быть выполнено с любой функцией, инвариантной к перестановке, — например: `sum`, `max` и `mean`. Так выглядит код функции `global_sum_pool()`:

```
def global_sum_pool(X, batch_mat):
    if batch_mat is None or batch_mat.dim() == 1:
        return torch.sum(X, dim=0).unsqueeze(0)
    else:
        return torch.mm(batch_mat, X)
```

Если данные не пакетированы или размер пакета равен единице, эта функция просто суммирует текущие встраивания узлов. В противном случае встраивания умножаются на `batch_mat`, структура которой основана на группировке данных графа.

Когда все данные в наборе данных имеют одинаковую размерность, пакетная обработка данных сводится к добавлению измерения путем суммирования данных. (Примечание: функция, вызываемая по умолчанию в функции пакетной обработки в PyTorch, буквально называется *стеком*: `stack`.) Но когда размеры графов существенно различаются, заполнение может быть неэффективным. Как правило, лучший способ справиться с различными размерами графов — рассматривать каждый пакет как единый граф, где каждый граф в пакете является подграфом, не связанным с остальными. Этот подход показан на рис. 18.10.

Чтобы описать это более формальным математическим языком, предположим, что существуют графы G_1, \dots, G_k размером n_1, \dots, n_k с f признаками на узел. Кроме того, нам даны соответствующие матрицы смежности A_1, \dots, A_k и матрицы признаков X_1, \dots, X_k .

Пусть N — общее количество узлов $N = \sum_{i=1}^k n_i$, $s_1 = 0$ и $s_i = s_{i-1} + n_{i-1}$ при $1 < i \leq k$. Как показано на рис. 18.10, мы определяем граф G_B с матрицей смежности A_B размером $N \times N$ и матрицей признаков X_B размером $N \times f$. В соответствии с нотацией индексов Python $A_B[s_i:s_i + n_i, s_i + n_i] = A_i$, а все остальные элементы A_B за пределами этих наборов индексов равны 0. Кроме того, $X_B[s_i:s_i + n_i, :] = X_i$.

Суть в том, что несвязанные узлы никогда не будут находиться в одном и том же рецептивном поле графовой свертки. В результате при обратном распространении градиентов G_B через графовые свертки градиенты, прикрепленные к каждому графу в пакете, будут независимыми. Это означает, что если рассматривать множество графовых сверток как функцию f , если $h_B = f(X_B, A_B)$ и $h_i = f(X_i, A_i)$, то $h_B[s_i:s_i + n_i, :] = h_i$. Если глобаль-

ное объединение по сумме извлекает суммы каждого h_i из h_B в виде отдельных векторов, прохождение этого стека векторов через полно связные слои будет сохранять градиенты каждого элемента в пакете отдельными на протяжении всего обратного распространения.

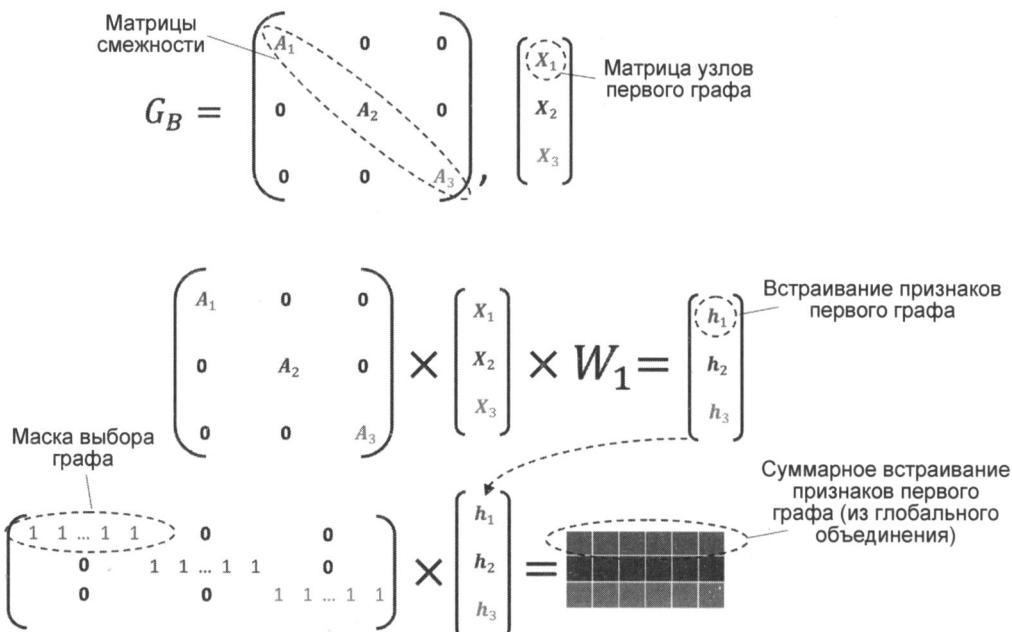


Рис. 18.10. Работа с графами разного размера

В этом и заключается назначение `batch_mat` в `global_sum_pool()` — служить маской выбора графа, которая разделяет графы в пакете. Мы можем сгенерировать эту маску для графов размером n_1, \dots, n_k с помощью следующего кода:

```
def get_batch_tensor(graph_sizes):
    starts = [sum(graph_sizes[:idx])
              for idx in range(len(graph_sizes))]
    stops = [starts[idx] + graph_sizes[idx]
             for idx in range(len(graph_sizes))]
    tot_len = sum(graph_sizes)
    batch_size = len(graph_sizes)
    batch_mat = torch.zeros([batch_size, tot_len]).float()
    for idx, starts_and_stops in enumerate(zip(starts, stops)):
        start = starts_and_stops[0]
        stop = starts_and_stops[1]
        batch_mat[idx, start:stop] = 1
    return batch_mat
```

Таким образом, при заданном размере пакета b матрица `batch_mat` имеет размер $b \times N$, где $batch_mat[i-1, s_i:s_i + n_i] = 1$ при $1 \leq i \leq k$, и где элементы вне этих наборов индексов равны 0. Далее приведен код функции логического умножения для построения представления некоторого G_B соответствующей пакетной матрицы:

```

# пакет представляет собой список словарей, каждый из которых
# содержит представление и метку графа
def collate_graphs(batch):
    adj_mats = [graph['A'] for graph in batch]
    sizes = [A.size(0) for A in adj_mats]
    tot_size = sum(sizes)
    # создаем пакетную матрицу
    batch_mat = get_batch_tensor(sizes)
    # объединяем матрицы признаков
    feat_mats = torch.cat([graph['X'] for graph in batch], dim=0)
    # объединяем метки
    labels = torch.cat([graph['y'] for graph in batch], dim=0)
    # объединяем матрицы смежности
    batch_adj = torch.zeros([tot_size, tot_size], dtype=torch.float32)
    accum = 0
    for adj in adj_mats:
        g_size = adj.shape[0]
        batch_adj[accum:accum+g_size, accum:accum+g_size] = adj
        accum = accum + g_size
    repr_and_label = {'A': batch_adj,
                      'X': feat_mats, 'y': labels,
                      'batch': batch_mat}
    return repr_and_label

```

18.3.4. Подготовка загрузчика данных *DataLoader*

В этом подразделе мы соберем воедино код из предыдущих подразделов. Сначала мы создадим несколько графов и поместим их в набор данных `Dataset PyTorch`. Затем воспользуемся нашей функцией сопоставления в `DataLoader` для создания сети GNN.

Но прежде чем мы определим графы, давайте напишем функцию для построения представления словаря, которое нам пригодится позже:

```

def get_graph_dict(G, mapping_dict):
    # Функция строит словарное представление графа G
    A = torch.from_numpy(
        np.asarray(nx.adjacency_matrix(G).todense())).float()
    # функция build_graph_color_label_representation()
    # рассмотрена в первом примере работы с графиком
    X = torch.from_numpy(
        build_graph_color_label_representation(
            G, mapping_dict)).float()
    # заглушка, т. к. для этого примера нет подходящей задачи
    y = torch.tensor([[1,0]]).float()
    return {'A': A, 'X': X, 'y': y, 'batch': None}

```

Эта функция принимает граф `NetworkX` и возвращает словарь, содержащий его матрицу смежности `A`, матрицу признаков узлов `X` и двоичную метку `y`. Поскольку на самом деле мы не будем обучать эту модель реальной задаче, мы просто устанавливаем метки произвольно. Затем `nx.adjacency_matrix()` берет граф `NetworkX` и возвращает разрежен-

ное представление, которое мы преобразуем в плотную форму `np.array` с помощью `todense()`.

Теперь построим графы и воспользуемся функцией `get_graph_dict` для преобразования графов NetworkX в формат, который может обрабатывать наша сеть:

```
>>> # строим 4 графа в качестве набора данных
>>> blue, orange, green = "#1f77b4", "#ff7f0e", "#2ca02c"
>>> mapping_dict= {green:0, blue:1, orange:2}
>>> G1 = nx.Graph()
>>> G1.add_nodes_from([
...     (1,{"color": blue}),
...     (2,{ "color": orange}),
...     (3,{ "color": blue}),
...     (4,{ "color": green})
... ])
>>> G1.add_edges_from([(1, 2), (2, 3), (1, 3), (3, 4)])
>>> G2 = nx.Graph()
>>> G2.add_nodes_from([
...     (1,{ "color": green}),
...     (2,{ "color": green}),
...     (3,{ "color": orange}),
...     (4,{ "color": orange}),
...     (5,{ "color": blue})
... ])
>>> G2.add_edges_from([(2, 3),(3, 4),(3, 1),(5, 1)])
>>> G3 = nx.Graph()
>>> G3.add_nodes_from([
...     (1,{ "color": orange}),
...     (2,{ "color": orange}),
...     (3,{ "color": green}),
...     (4,{ "color": green}),
...     (5,{ "color": blue}),
...     (6,{ "color": orange})
... ])
>>> G3.add_edges_from([(2,3), (3,4), (3,1), (5,1), (2,5), (6,1)])
>>> G4 = nx.Graph()
>>> G4.add_nodes_from([
...     (1,{ "color": blue}),
...     (2,{ "color": blue}),
...     (3,{ "color": green})
... ])
>>> G4.add_edges_from([(1, 2), (2, 3)])
>>> graph_list = [get_graph_dict(graph, mapping_dict) for graph in
...     [G1, G2, G3, G4]]
```

Графы, сгенерированные этим кодом, показаны на рис. 18.11.

Приведенный блок кода строит четыре графа NetworkX и сохраняет их в списке. Здесь конструктор `nx.Graph()` инициализирует пустой граф, а `add_nodes_from()` добавляет узлы в пустой граф из списка кортежей. Первый элемент в каждом кортеже — это имя узла, а второй элемент — словарь атрибутов этого узла.

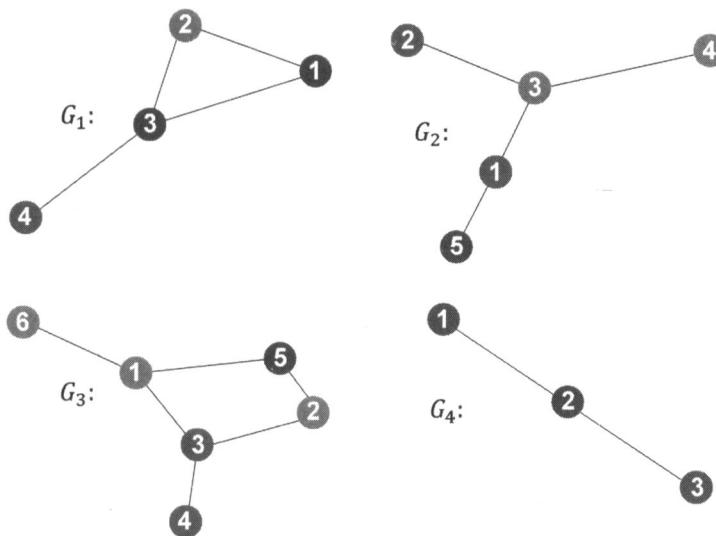


Рис. 18.11. Четыре сгенерированных графа

Метод графа `add_edges_from()` принимает список кортежей, где каждый кортеж определяет ребро между его элементами (узлами). Теперь мы можем создать набор данных PyTorch для этих графов:

```
from torch.utils.data import Dataset
class ExampleDataset(Dataset):
    # Простой набор данных PyTorch, который будет использовать наш список графов
    def __init__(self, graph_list):
        self.graphs = graph_list
    def __len__(self):
        return len(self.graphs)

    def __getitem__(self, idx):
        mol_rep = self.graphs[idx]
        return mol_rep
```

Хотя использование специально созданного `Dataset` может показаться ненужным усилием, это позволяет нам продемонстрировать использование `collate_graphs()` в `DataLoader`:

```
>>> from torch.utils.data import DataLoader
>>> dset = ExampleDataset(graph_list)
>>> # Заметьте, как мы используем нашу функцию collate
>>> loader = DataLoader(
...     dset, batch_size=2, shuffle=False,
...     collate_fn=collate_graphs)
```

18.3.5. Использование *NodeNetwork* для прогнозирования

Когда мы определили все необходимые функции и настроили `DataLoader`, можно инициализировать новую сеть `NodeNetwork` и применить ее к данным нашего графа:

```
>>> node_features = 3
>>> net = NodeNetwork(node_features)

>>> batch_results = []
>>> for b in loader:
...     batch_results.append(
...         net(b['X'], b['A'], b['batch']).detach())
```

Заметим, что для краткости мы не включили в код цикл обучения, но модель GNN можно обучать как обычно, вычисляя потери между предсказанными и истинными метками классов, распространяя потери обратно, используя `.backward()` и обновляя веса модели с помощью оптимизатора на основе градиентного спуска. Мы оставляем обучение модели как дополнительное упражнение для читателя. В следующем разделе мы покажем, как это сделать с помощью реализации GNN от PyTorch Geometric, которая предоставляет более сложный код модели.

В продолжение нашего предыдущего кода подадим на вход модели один граф напрямую — без `DataLoader`:

```
>>> G1_rep = dset[1]
>>> G1_single = net(
...     G1_rep['X'], G1_rep['A'], G1_rep['batch']).detach()
```

Теперь мы можем сравнить результаты применения GNN к одиночному графу (`G1_single`) и к первому графу из `DataLoader` (это будет тоже первый граф `G1`, в чем мы можем быть уверены, поскольку отключили перемешивание, установив `shuffle=False`), чтобы убедиться, что пакетный загрузчик работает корректно. Применив `torch.isclose()` (для учета ошибок округления), мы видим, что результаты эквивалентны, как мы и надеялись:

```
>>> G1_batch = batch_results[0][1]
>>> torch.all(torch.isclose(G1_single, G1_batch))
tensor(True)
```

Поздравляю! Теперь вы знаете, как создавать, настраивать и запускать базовую GNN. Однако из этого введения вы, вероятно, также поняли, что управление графовыми данными и их обработка могут быть весьма трудоемкими. Кроме того, мы даже не построили графовую свертку, использующую метки ребер, что еще больше усложнило бы ситуацию. К счастью, существует PyTorch Geometric — пакет, который значительно упрощает эту задачу, предоставляя реализации многих слоев GNN. В следующем разделе вы познакомитесь с этой геометрической библиотекой на примере реализации и обучения более сложной GNN для работы с молекулярными данными.

18.4. Построение GNN с использованием геометрической библиотеки PyTorch

В этом разделе мы построим GNN с использованием библиотеки PyTorch Geometric, которая упрощает процесс обучения GNN. Затем применим GNN к состоящему из описаний небольших молекул набору данных QM9 — чтобы предсказать изотропную поляризумость, которая является мерой склонности молекулы к искажению собственного заряда под влиянием внешнего электрического поля.



Установка PyTorch Geometric

Библиотеку PyTorch Geometric можно установить с помощью conda или pip. Мы рекомендуем вам посетить официальный веб-сайт ее документации по адресу: <https://pytorch-geometric.readthedocs.io/en/latest/notes/installation.html>, чтобы выбрать способ установки, рекомендованный для вашей операционной системы. В этой главе мы использовали pip для установки версии 2.0.2 вместе с ее зависимостями torch-scatter и torch-sparse:

```
pip install torch-scatter==2.0.9
pip install torch-sparse==0.6.12
pip install torch-geometric==2.0.2
```

Мы начнем с загрузки набора данных о малых молекулах и посмотрим, как PyTorch Geometric хранит данные:

```
>>> # Для всех примеров в этом разделе мы используем импорт, как показано далее
>>> # Учтите, что мы используем DataLoader torch_geometric.
>>> import torch
>>> from torch_geometric.datasets import QM9
>>> from torch_geometric.loader import DataLoader
>>> from torch_geometric.nn import NNConv, global_add_pool
>>> import torch.nn.functional as F
>>> import torch.nn as nn
>>> import numpy as np
>>> # загружаем набор данных малых молекул QM9
>>> dset = QM9('.')
>>> len(dset)
130831
>>> # так torch geometric упаковывает данные в оболочку
>>> data = dset[0]
>>> data
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", pos=[5, 3],
x=[5, 11], y=[1, 19], z=[5])
>>> # к атрибутам можно обращаться напрямую
>>> data.z
tensor([6, 1, 1, 1, 1])
>>> # атомный номер каждого атома может добавлять атрибуты
>>> data.new_attribute = torch.tensor([1, 2, 3])
>>> data
Data(edge_attr=[8, 4], edge_index=[2, 8], idx=[1], name="gdb_1", new_
attribute=[3], pos=[5, 3], x=[5, 11], y=[1, 19], z=[5])
>>> # все атрибуты можно переносить между устройствами
>>> device = torch.device(
... "cuda:0" if torch.cuda.is_available() else "cpu"
... )
>>> data.to(device)
>>> data.new_attribute.is_cuda
True
```

Объект Data — это удобная и гибкая оболочка для графовых данных. Учтите, что многие объекты PyTorch Geometric требуют наличия определенных ключевых слов в объ-

ектах данных для их правильной обработки. В частности, x должен содержать признаки узла, $edge_attr$ — признаки ребер, $edge_index$ — список ребер, а y — метки. Данные QM9 содержат некоторые дополнительные примечательные атрибуты: pos — положение каждого атома молекулы в трехмерной сетке и z — атомный номер каждого атома в молекуле. Метки в QM9 представляют собой набор физических свойств молекул — таких как дипольный момент, свободная энергия, энタルпия или изотропная поляризация. Мы собираемся построить GNN и обучить ее на QM9 для предсказания изотропной поляризации.



Набор данных QM9

Набор данных QM9 содержит описания 133 885 небольших органических молекул по некоторым геометрическим, энергетическим, электронным и термодинамическим параметрам. QM9 — это популярный эталонный набор данных для разработки методов прогнозирования взаимосвязей между химической структурой и свойствами, а также гибридных методов квантовой механики и машинного обучения. Более подробную информацию о наборе данных можно найти по адресу: <http://quantum-machine.org/datasets/>.

Типы связей, задействованные в молекулах, очень важны — например, многие свойства зависят от того, одинарные или двойные связи соединяют атомы в молекуле. Следовательно, нам нужна графовая свертка, которая может использовать признаки ребер, и мы применим слой `torch_geometric.nn.NNConv`⁵.

Эта свертка в слое `NNConv` принимает следующий вид:

$$\mathbf{X}_i^{(t)} = \mathbf{W}\mathbf{X}_i^{(t-1)} + \sum_{j \in N(i)} \mathbf{X}_j^{(t-1)} \cdot h_\Theta(e_{i,j}).$$

Здесь h — нейронная сеть, параметризованная набором весов Θ , а \mathbf{W} — матрица весов для меток узлов. Эта графовая свертка очень похожа на ту, для которой мы недавно разработали реализацию с нуля:

$$\mathbf{X}_i^{(t)} = \mathbf{W}_1 \mathbf{X}_i^{(t-1)} + \sum_{j \in N(i)} \mathbf{X}_j^{(t-1)} \mathbf{W}_2.$$

Единственное реальное отличие состоит в том, что эквивалент \mathbf{W}_2 , нейронная сеть h , параметризуется на основе меток ребер, что позволяет изменять веса для разных меток ребер. С помощью следующего кода мы создадим GNN, используя два слоя графовой свертки (`NNConv`):

```
class ExampleNet(torch.nn.Module):
    def __init__(self, num_node_features, num_edge_features):
        super().__init__()
        conv1_net = nn.Sequential(
            nn.Linear(num_edge_features, 32),
            nn.ReLU(),
            nn.Linear(32, num_node_features*32))
```

⁵ Если вас интересуют подробности реализации, исходный код можно найти по адресу: https://pytorch-geometric.readthedocs.io/en/latest/_modules/torch_geometric/norm/nn_conv/nn_conv.html#NNConv.

```

conv2_net = nn.Sequential(
    nn.Linear(num_edge_features, 32),
    nn.ReLU(),
    nn.Linear(32, 32*16))

self.conv1 = NNConv(num_node_features, 32, conv1_net)
self.conv2 = NNConv(32, 16, conv2_net)
self.fc_1 = nn.Linear(16, 32)
self.out = nn.Linear(32, 1)

def forward(self, data):
    batch, x, edge_index, edge_attr = (
        data.batch, data.x, data.edge_index, data.edge_attr)
    # Первый слой графовой свертки
    x = F.relu(self.conv1(x, edge_index, edge_attr))
    # Второй слой графовой свертки
    x = F.relu(self.conv2(x, edge_index, edge_attr))
    x = global_add_pool(x, batch)
    x = F.relu(self.fc_1(x))
    output = self.out(x)
    return output

```

Мы обучим эту GNN предсказывать изотропную поляризуемость молекулы — меру относительной склонности к искажению заряда молекулы во внешнем электрическом поле. Как принято, мы разделим набор данных QM9 на наборы для обучения, валидации и тестирования и используем `DataLoader` библиотеки PyTorch Geometric. Для этих наборов не требуется специальная функция сопоставления, но объект данных должен иметь атрибуты с соответствующими именами.

Итак, разделим набор данных:

```

>>> from torch.utils.data import random_split
>>> train_set, valid_set, test_set = random_split(
...     dset, [110000, 10831, 10000])
>>> trainloader = DataLoader(train_set, batch_size=32, shuffle=True)
>>> validloader = DataLoader(valid_set, batch_size=32, shuffle=True)
>>> testloader = DataLoader(test_set, batch_size=32, shuffle=True)

```

Следующий код инициализирует и обучает сеть на графическом процессоре (если он доступен):

```

>>> # инициализация сети
>>> qm9_node_feats, qm9_edge_feats = 11, 4
>>> net = ExampleNet(qm9_node_feats, qm9_edge_feats)

>>> # инициализация оптимизатора с разумными параметрами
>>> optimizer = torch.optim.Adam(
...     net.parameters(), lr=0.01)
>>> epochs = 4
>>> target_idx = 1 # индекс положения метки поляризуемости
>>> device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
>>> net.to(device)

```

Цикл обучения, показанный далее, хорошо известен вам из предыдущих глав про PyTorch, поэтому можно не вдаваться в подробности. Однако одна деталь, которую стоит подчеркнуть, заключается в том, что здесь мы вычисляем среднеквадратичную ошибку (Mean Squared Error, MSE) потери вместо перекрестной энтропии, поскольку поляризуемость является непрерывным параметром, а не меткой класса:

```
>>> for total_epochs in range(epochs):
...     epoch_loss = 0
...     total_graphs = 0
...     net.train()
...     for batch in trainloader:
...         batch.to(device)
...         optimizer.zero_grad()
...         output = net(batch)
...         loss = F.mse_loss(
...             output, batch.y[:, target_idx].unsqueeze(1))
...         loss.backward()
...         epoch_loss += loss.item()
...         total_graphs += batch.num_graphs
...         optimizer.step()
...     train_avg_loss = epoch_loss / total_graphs
...     val_loss = 0
...     total_graphs = 0
...     net.eval()
...     for batch in validloader:
...         batch.to(device)
...         output = net(batch)
...         loss = F.mse_loss(
...             output, batch.y[:, target_idx].unsqueeze(1))
...         val_loss += loss.item()
...         total_graphs += batch.num_graphs
...     val_avg_loss = val_loss / total_graphs
...     print(f"Эпоха: {total_epochs} | "
...           f"cp.потеря эпохи: {train_avg_loss:.2f} | "
...           f"cp.потеря валид.: {val_avg_loss:.2f}")
Эпоха: 0 | cp.потеря эпохи: 0.30 | cp.потеря валид.: 0.10
Эпоха: 1 | cp.потеря эпохи: 0.12 | cp.потеря валид.: 0.07
Эпоха: 2 | cp.потеря эпохи: 0.10 | cp.потеря валид.: 0.05
Эпоха: 3 | cp.потеря эпохи: 0.09 | cp.потеря валид.: 0.07
```

В течение первых четырех эпох обучения потери как при обучении, так и при валидации уменьшаются. Набор данных большой, и обучение на обычном процессоре может занять некоторое время, поэтому мы прекращаем обучение после четырех эпох. Однако, если мы будем обучать модель дальше, потери будут продолжать уменьшаться. Вы можете обучить модель на протяжении дополнительных эпох, чтобы увидеть, как это меняет ее производительность.

Следующий код предсказывает значения тестовых данных и собирает истинные метки:

```
>>> net.eval()
>>> predictions = []
```

```
>>> real = []
>>> for batch in testloader:
...     output = net(batch.to(device))
...     predictions.append(output.detach().cpu().numpy())
...     real.append(
...         batch.y[:,target_idx].detach().cpu().numpy())
>>> real = np.concatenate(real)
>>> predictions = np.concatenate(predictions)
```

Теперь мы можем построить диаграмму рассеяния с подмножеством тестовых данных. Поскольку набор тестовых данных относительно велик (10 тыс. молекул), результаты могут быть немного загромождены, и для простоты мы наносим на диаграмму только первые 500 прогнозов и целей:

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(real[:500], predictions[:500])
>>> plt.xlabel('Isotropic polarizability')
>>> plt.ylabel('Predicted isotropic polarizability')
```

Полученная диаграмма показана на рис. 18.12.

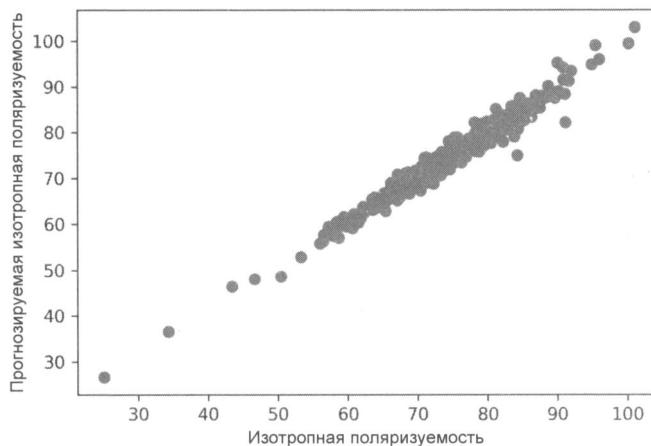


Рис. 18.12. Прогнозируемая изотропная поляризуемость относительно фактической изотропной поляризуемости

Судя по тому, что точки на диаграмме лежат относительно близко к диагонали, наша простая GNN проделала достойную работу по предсказанию значений изотропной поляризуемости молекул даже без настройки гиперпараметров.



TorchDrug — библиотека на основе PyTorch для поиска лекарств

PyTorch Geometric — это всеобъемлющая библиотека общего назначения для работы с графиками, включая молекулы, как вы видели в этом разделе. Если вы нуждаетесь в более углубленной работе с молекулами и поиском лекарств, мы также рекомендуем рассмотреть недавно разработанную библиотеку TorchDrug, которая предлагает множество удобных инструментов для работы с молекулами. Вы можете узнать больше о TorchDrug по адресу: <https://torchdrug.ai/>.

18.5. Другие слои GNN и новейшие разработки

В этом разделе мы обсудим другие слои, которые вы можете использовать в своих GNN, а также в общих чертах рассмотрим последние разработки в этой области. Математические основы идей, о которых пойдет речь, могут показаться вам сложными с математической точки зрения, но не расстраивайтесь — темы эти не являются обязательными, и нет необходимости вникать в тонкости всех показанных реализаций. Общего понимания идей, лежащих в основе дополнительных слоев, будет достаточно, чтобы провести эксперименты с реализациями PyTorch Geometric, на которые мы ссылаемся.

В следующих подразделах мы рассмотрим слои спектральной графовой свертки, слои объединения и слои нормализации для графов. В заключительном подразделе будет представлен обзор некоторых более продвинутых типов графовых нейронных сетей.

18.5.1. Спектральные графовые свертки

Графовые свертки, с которыми мы работали до этого момента, были пространственными по своей природе. Это означает, что они агрегируют информацию на основе топологического пространства, связанного с графом, — что есть лишь причудливый способ сказать, что пространственные свертки работают с локальными соседями узлов. Как следствие, если GNN, использующая пространственные свертки, должна выявлять сложные глобальные закономерности в данных графа, то сети потребуется сложить несколько пространственных сверток. В ситуациях, когда эти глобальные закономерности важны, но необходимо ограничить глубину сети, хорошую альтернативу могут предоставить спектральные графовые свертки.

Спектральные графовые свертки работают иначе, чем пространственные. Они используют *спектр графа* — его набор собственных значений — путем вычисления собственной декомпозиции нормализованной версии матрицы смежности графа, называемой *лапласианом графа*. Последнее предложение может показаться бессмысленным набором слов, поэтому давайте разберем его по частям.

Для неориентированного графа матрица Лапласа (Laplacian matrix) графа определяется как $L = D - A$, где A — матрица смежности графа, а D — степенная матрица. *Степенная матрица* (degree matrix) — это диагональная матрица, где элемент на диагонали в строке с индексом i — это количество ребер, входящих и исходящих из узла, связанного с i -й строкой матрицы смежности.

L — симметричная матрица с вещественными значениями, и было доказано, что такие матрицы можно разложить как $L = Q\Lambda Q^T$, где Q — ортогональная матрица, столбцы которой являются собственными векторами L , а Λ — диагональная матрица, элементы которой являются собственными значениями L . Вы можете рассматривать Q как базовое представление структуры графа. В отличие от пространственных сверток, которые учитывают локальные окрестности графа, определенные A , спектральные свертки используют альтернативное представление структуры из Q для обновления встраиваний узлов.

В показанном далее примере спектральной свертки применяется собственное разложение *симметричного нормализованного лапласиана графа*, которое определяется для графа следующим образом:

$$L_{sym} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}},$$

где I — единичная матрица (матрица тождественности). Эта процедура используется, потому что нормализация лапласiana графа способствует стабилизации процесса обучения на основе градиента, аналогично стандартизации признаков.

Учитывая, что $Q\Lambda Q^T$ является собственным разложением L_{sym} , графовую свертку можно определить следующим образом:

$$X' = Q(Q^T X \odot Q^T W),$$

где W — матрица обучаемых весов. Внутри скобок по существу происходит умножение X и W на матрицу, которая кодирует структурные отношения в графе. Оператор \odot здесь обозначает поэлементное умножение внутренних членов, в то время как внешняя Q отображает результат обратно в исходный базис. У этой свертки есть несколько нежелательных свойств, поскольку вычисление собственного разложения графа имеет вычислительную сложность $O(n^3)$. Это означает, что она медленная, и поскольку она структурирована, W зависит от размера графа. Следовательно, спектральная свертка может применяться только к графикам одинакового размера. Кроме того, рецептивным полем этой свертки является весь график, и его невозможно настроить при текущем подходе. Однако для решения этих проблем были разработаны различные методы и обходные пути.

Например, Бруна и его коллеги⁶ представили метод сглаживания, который учитывает зависимость W от размера графа, аппроксимируя ее набором функций, каждая из которых умножается на собственный скалярный параметр α . То есть задано множество функций f_1, \dots, f_n , $W \approx \sum \alpha_i f_i$. Это множество функций таково, что размерность можно варьировать. Однако, поскольку α остается скалярным, пространство параметров сверток может не зависеть от размера графа.

К другим спектральным сверткам, о которых стоит упомянуть, относятся графовые свертки Чебышева⁷, которые способны аппроксимировать исходную спектральную свертку с меньшей временной сложностью и могут иметь рецептивные поля разных размеров. Кипф и Веллинг⁸ предлагают свертки со свойствами, подобными сверткам Чебышева, но с уменьшенной сложностью параметров. Реализации обеих этих сверток доступны в PyTorch Geometric как `torch_geometric.nn.ChebConv` и `torch_geometric.nn.GNConv` и хорошо подходят для начала, если вы хотите поэкспериментировать со спектральными свертками.

18.5.2. Объединение

В этом подразделе мы кратко обсудим некоторые примеры слоев объединения, которые были разработаны специально для графов. Хотя понижение разрешения за счет слоев объединения было полезным в архитектурах CNN, преимущество понижения разрешения в GNN менее очевидно.

Слои объединения для данных изображения используют пространственную локальность, которой нет у графов. Если предусмотрена кластеризация узлов в графике, мы мо-

⁶ См. <https://arxiv.org/abs/1312.6203>.

⁷ См. <https://arxiv.org/abs/1606.09375>.

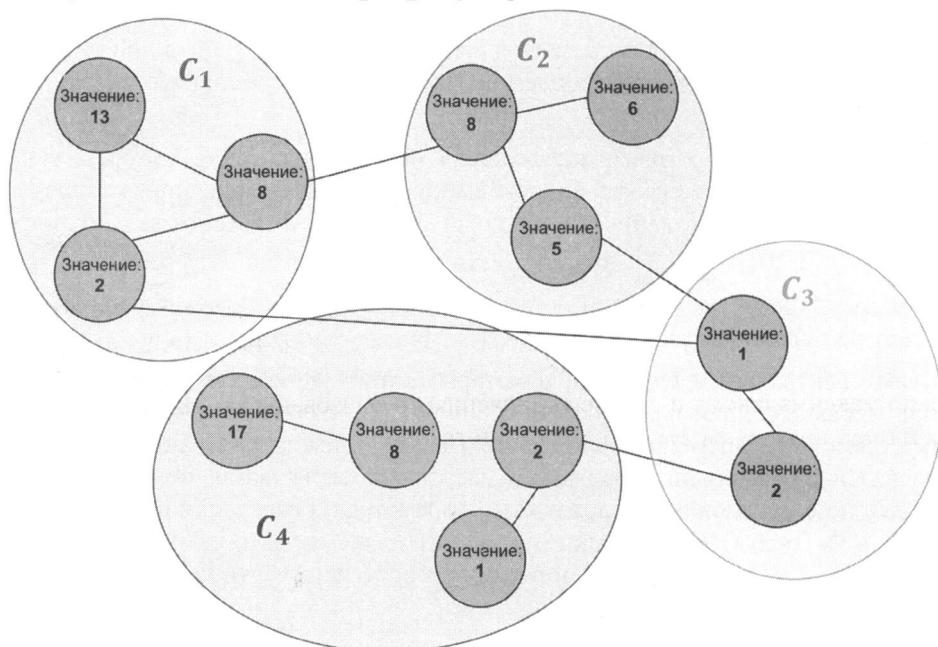
⁸ См. <https://arxiv.org/abs/1609.02907>.

жем определить, как слой объединения должен объединять узлы графа. Однако неясно, как определить оптимальную кластеризацию, и для разных контекстов могут быть предпочтительны разные подходы к кластеризации. И даже после определения кластеризации, если узлы разреженные, неясно, что делать с оставшимися узлами. Хотя все это открытые темы для исследований, мы рассмотрим несколько слоев объединения для графовых сетей и их подходы к упомянутым проблемам.

Как и в случае с CNN, для графовых сетей существуют слои объединения по максимальному и среднему значению. Как показано на рис. 18.13, при кластеризации узлов каждый кластер становится узлом в новом графе.

Встраивание каждого кластера равно среднему или максимальному значению встраиваний узлов в кластере. Для сохранения связности кластеру назначается объединение

Граф G с кластерами C_1, C_2, C_3, C_4 :



Граф G после объединения по максимуму:

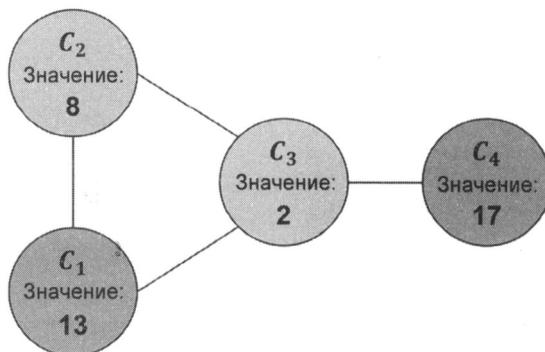


Рис. 18.13. Применение к графу объединения по максимуму

всех индексов ребер в кластере. Например, если узлы i, j, k назначены кластеру c_1 , любой узел или кластер, содержащий узел, который имеет общее ребро с i, j или k , будет иметь общее ребро с c_1 .

Более сложный объединяющий слой DiffPool⁹ пытается одновременно выполнить как кластеризацию, так и понижение разрешения. Этот слой изучает матрицу назначения мягких кластеров $S \in \mathbb{R}^{n \times c}$, которая распределяет встраивания n узлов на c кластеров. (Мы говорили о мягкой и жесткой кластеризации в разд. 10.1.3.) При этом X заменяется на $X' = S^T X$, а A — на $A' = S^T A^T S$. Примечательно, что A' больше не содержит дискретных значений и вместо этого может рассматриваться как матрица весов ребер. Со временем DiffPool сходится к почти жесткой кластеризации с интерпретируемой структурой.

Другой метод объединения — *top-k pooling* (объединение по k -верхним) — удаляет узлы из графа вместо их агрегирования, что позволяет обойти проблемы кластеризации и связности. Хотя такое объединение связано с возможной потерей информации в удаленных узлах, в контексте целой сети, если перед объединением происходит свертка, сеть может научиться избегать этого. Отброшенные узлы выбираются с использованием оценки проекции относительно обучаемого вектора p . Формула для вычисления (X', A') , как отмечено в разделе «О классификаторах разреженных иерархических графов» статьи, приведенной по адресу: <https://arxiv.org/abs/1811.01287>, выглядит следующим образом:

$$y = \frac{X_p}{\|p\|}, \quad i = \text{top-}k(y, k), \quad X' = (X \odot \tanh(y))_i, \quad A' = A_{ii}.$$

Здесь метод top-k выбирает индексы y , при этом верхние значения k и индексный вектор i используются для удаления строк X и A . Объединение по методу top-k реализовано в PyTorch Geometric как модуль `torch_geometric.nn.TopKPooling`. Кроме того, объединения по максимальному и среднему значениям реализованы как `torch_geometric.nn.max_pool_x` и `torch_Geometry.nn.avg_pool_x` соответственно.

18.5.3. Нормализация

Методы нормализации применяются во многих видах нейронных сетей для стабилизации и/или ускорения процесса обучения. Многие подходы, такие как пакетная нормализация (обсуждается в главе 17), могут быть легко перенесены на GNN с небольшими изменениями. В этом подразделе мы кратко опишем некоторые слои нормализации, которые были разработаны специально для графовых данных.

Давайте кратко вспомним, в чем заключается нормализация. В заданном наборе значений признаков x_1, \dots, x_n мы обновляем значения с помощью формулы $\frac{x_i - \mu}{\sigma}$, где μ — среднее значение, а σ — стандартное отклонение набора значений. Как правило, большинство методов нормализации нейронных сетей имеют общий вид $\gamma \frac{x_i - \mu}{\sigma} + \beta$, где γ и β — обучаемые параметры, а разница между методами связана с набором признаков, к которым применяется нормализация.

⁹ См. <https://arxiv.org/abs/1806.08804>.

В статье¹⁰ показано, что статистические данные после агрегирования в графовой свертке могут содержать значимую информацию, поэтому полностью отказываться от них может быть нежелательно. Чтобы решить эту проблему, был предложен метод GraphNorm.

Позаимствуем обозначения из этой статьи. Пусть h будет матрицей встраиваний узлов, а $h_{i,j}$ — j -м значением признака узла v_i , где $i = 1, \dots, n$ и $j = 1, \dots, d$. Метод GraphNorm можно описать следующей формулой:

$$\gamma_j \frac{h_{i,j} - \alpha_j \cdot \mu_j}{\hat{\sigma}_j} + \beta_j.$$

Здесь

$$\mu_j = \frac{\sum_{i=1}^n h_{i,j}}{n}$$

и

$$\hat{\sigma}_j = \sqrt{\frac{\sum_{i=1}^n (h_{i,j} - \alpha_j \mu_j)^2}{n}}.$$

Ключевым дополнением является обучаемый параметр α , от которого зависит, какую часть μ_j отбрасывать.

Другой метод нормализации графов — MsgNorm¹¹, основанный на формуле графовой свертки с передачей сообщений, упомянутой ранее в этой главе. Если использовать обозначения для сети с передачей сообщений (определенные в конце разд. 18.2.2), после того как графовая свертка суммирует M_i и дает m_i , но до обновления встраиваний узлов с U_i , алгоритм MsgNorm нормализует m_i по следующей формуле:

$$m'_i = s \cdot \|h_i\|_2 \cdot \frac{m_i}{\|m_i\|_2}.$$

Здесь s — обучаемый коэффициент масштабирования, и идея, лежащая в основе этого подхода, заключается в том, чтобы нормализовать признаки агрегированных сообщений в графовой свертке. Хотя теории, обосновывающей этот подход к нормализации, не существует, на практике он хорошо зарекомендовал себя.

Все упомянутые здесь слои нормализации реализованы и доступны через PyTorch Geometric как BatchNorm, GroupNorm и MessageNorm¹².

В отличие от слоев объединения графовых моделей, для которых может потребоваться дополнительная настройка кластеризации, слои нормализации легче подключить к существующей модели GNN. Опробование различных методов нормализации во время разработки и оптимизации модели — это разумный общепринятый подход.

¹⁰ См. «GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training» by Tianle Cai et al., 2020, <https://arxiv.org/abs/2009.03294>.

¹¹ Описан в 2020 г. Гонхао Ли и его коллегами в статье «DeeperGCN: All You Need to Train Deeper GCNs», <https://arxiv.org/abs/2006.07739>.

¹² За дополнительной информацией можно обратиться к документации PyTorch Geometric по адресу: <https://pytorch-geometric.readthedocs.io/en/latest/modules/norm.html#normalization-layers>.

18.5.4. Дополнительная литература по графовым нейронным сетям

Область глубокого обучения, ориентированная на графы, быстро развивается, и существует множество методов, которые мы не можем достаточно подробно описать в этой обзорной главе. Поэтому, прежде чем ее завершить, мы хотим предоставить заинтересованным читателям подборку ссылок на заслуживающую внимания литературу для более глубокого изучения этой темы.

Как вы, возможно, помните из главы 16, механизмы внимания могут расширить возможности моделей, предоставляя дополнительные контексты. В связи с этим были разработаны различные методы внимания для GNN. Примером GNN, дополненных механизмом внимания, могут служить графовые сети, предложенные Петаром Величковичем и его коллегами в 2017 г. (см. <https://arxiv.org/abs/1710.10903>) и реляционные графовые сети Дэна Басбриджа и его коллег, предложенные в 2019 г. (см. <https://arxiv.org/abs/1904.05811>).

Недавно эти механизмы внимания нашли применение в графовых трансформерах, предложенных Сонджуном Юном и его коллегами в 2020 г. (см. <https://arxiv.org/abs/1911.06455>), и в трансформере на основе гетерогенных графов, предложенном Зиниу Ху и его коллегами также в 2020 г. (см. <https://arxiv.org/abs/2003.01332>).

Помимо упомянутых графовых трансформеров, специально для графов были разработаны другие глубокие генеративные модели. Существуют графовые вариационные автокодировщики — такие как те, что были представлены в работах «Variational Graph Auto-Encoders», Kipf and Welling, 2016 (см. <https://arxiv.org/abs/1611.07308>), «Constrained Graph Variational Autoencoders for Molecule Design», Qi Liu et al., 2018 (см. <https://arxiv.org/abs/1805.09076>) и «GraphVAE: Towards Generation of Small Graphs Using Variational Autoencoders», Simonovsky and Komodakis, 2018 (см. <https://arxiv.org/abs/1802.03480>).

Еще один известный графовый вариационный автокодировщик, который применялся для генерации молекул, представлен в статье «Junction Tree Variational Autoencoder for Molecular Graph Generation», Wengong Jin et al., 2019 (см. <https://arxiv.org/abs/1802.04364>). Некоторые GAN были разработаны специально для генерации графовых данных, хотя на момент опубликования этой статьи производительность GAN в области графов была гораздо менее впечатляющей, чем в области изображений. Примерами могут служить публикации «GraphGAN: Graph Representation Learning with Generative Adversarial Nets», Hongwei Wang and colleagues, 2017 (см. <https://arxiv.org/abs/1711.08267>) и «MolGAN: An Implicit Generative Model for Small Molecular Graphs», Cao and Kipf, 2018 (см. <https://arxiv.org/abs/1805.11973>).

GNN также были интегрированы в модели глубокого обучения с подкреплением — вы узнаете больше об обучении с подкреплением в следующей главе. К примерам таких моделей относятся «Graph Convolutional Policy Network for Goal-Directed Molecular Graph Generation», Jiaxuan You et al., 2018 (см. <https://arxiv.org/abs/1806.02473>) и глубокая Q-сеть, предложенная в работе «Optimization of Molecules via Deep Reinforcement Learning», Zhenpeng Zhou et al., 2018 (см. <https://arxiv.org/abs/1810.08678>), в которой используется GNN, примененная к задачам генерации молекул.

Наконец, хотя технически это не графовые данные, 3D-облака точек иногда представляют как таковые с использованием отсечек по расстоянию для создания краев. Пример

применения графовых сетей в таком пространстве точек описан в статье «Point-GNN: Graph Neural Network for 3D Object Detection in a Point Cloud», Weijing Shi et al., 2020 (см. <https://arxiv.org/abs/2003.01251>), где нейросеть обнаруживает 3D-объекты в облаке точек лидара. Кроме того, сеть, предложенная в статье «GAPNet: Graph Attention based Point Neural Network for Exploiting Local Feature of Point Cloud», Can Chen et al., 2019 (см. <https://arxiv.org/abs/1905.08705>), была разработана для обнаружения локальных признаков в данных облака точек, что является сложной задачей для других архитектур глубоких нейросетей.

18.6. Заключение

По мере роста количества доступных данных будет расти и наша потребность в понимании взаимосвязей внутри данных. Хотя этого можно достичь разными способами, графы представляют собой дистиллированное представление этих взаимосвязей, поэтому количество доступных графовых данных будет только увеличиваться.

В этой главе мы по шагам и компонентам разобрали графовые нейронные сети, самостоятельно реализовав с нуля слой графовой свертки и GNN. Вы увидели, что реализация GNN из-за особенностей графовых данных на самом деле довольно сложна. Поэтому, чтобы применить GNN к решению реальной задачи, такой как предсказание поляризации молекул, пришлось использовать библиотеку PyTorch Geometric, которая предоставляет готовую реализацию многих необходимых нам функциональных компонентов. Главу завершает список дополнительной литературы для более глубокого погружения в тематику GNN.

Надеюсь, эта глава научила вас использовать глубокое обучение в графовых моделях. Эта область в настоящее время является предметом интенсивных исследований, и многие из упомянутых здесь статей были опубликованы за последние пару лет (перед написанием этой книги). Взяв эту главу за отправную точку, вполне возможно, вы сможете добиться новых успехов в области графовых нейронных сетей.

В следующей главе мы рассмотрим обучение с подкреплением, которое представляет собой совершенно особую категорию машинного обучения по сравнению с ранее упомянутыми в этой книге.

19

Обучение с подкреплением для принятия решений в сложных условиях

Предыдущие главы были посвящены машинному обучению с учителем и без учителя. Вы также узнали, как использовать искусственные нейронные сети и глубокое обучение для решения задач, связанных с этими типами машинного обучения. Как вы помните, целью обучения с учителем является предсказание метки категории или непрерывного значения из заданного входного вектора признаков. Обучение без учителя сосредоточено на извлечении паттернов из данных, что делает его полезным для сжатия данных (*глава 5*), кластеризации (*глава 10*) или аппроксимации распределения обучающей выборки для создания новых данных (*глава 17*).

Эта глава посвящена отдельной категории машинного обучения — *обучению с подкреплением* (Reinforcement Learning, RL), которая отличается от предыдущих категорий тем, что ориентирована на обучение последовательности действий для достижения наибольшего общего вознаграждения — например, победы в шахматной игре.

В этой главе будут рассмотрены следующие темы:

- ◆ основы RL, знакомство с взаимодействием агента и среды и пояснение того, как работает процесс вознаграждения, помогающий принимать решения в сложных условиях;
- ◆ знакомство с различными категориями задач RL, задачами обучения на основе моделей и без моделей, методом Монте-Карло и с алгоритмами обучения на временных разностях;
- ◆ реализация алгоритма Q-обучения в табличном формате;
- ◆ определение аппроксимации функций для решения задач RL и сочетание RL с глубоким обучением путем реализации алгоритма глубокого Q-обучения.

Обучение с подкреплением — сложная и обширная область исследований, и в этой главе основное внимание уделяется ее основам. Поскольку глава представляет собой лишь введение в тему, чтобы сосредоточить ваше внимание на самых важных методах и алгоритмах, мы приведем здесь главным образом простые примеры, иллюстрирующие базовые концепции. Однако ближе к концу главы мы рассмотрим более сложный пример и применим архитектуру глубокого обучения в особом подходе RL — глубоком Q-обучении.

19.1. Введение в обучение на собственном опыте

Сначала мы рассмотрим обучение с подкреплением как разновидность машинного обучения и отметим его основные отличия от других методов. После этого вы познакомитесь с основными компонентами системы RL. Затем мы исследуем математическое представление RL, основанное на марковском процессе принятия решений.

19.1.1. Ключевые идеи обучения с подкреплением

До этого момента мы уделяли основное внимание обучению с учителем и без учителя. Напомним, что в обучении с учителем мы используем помеченные обучающие примеры, которые предоставляются учителем или человеком-экспертом, и цель состоит в том, чтобы обучить на них модель, которая хорошо обобщает полученные знания на незнакомые данные. Это означает, что модель обучения с учителем должна научиться присваивать заданному входному примеру те же метки или значения, что и человек-эксперт (учитель). При обучении без учителя цель состоит в том, чтобы выявить базовые закономерности в наборе данных — например, с помощью методов кластеризации и уменьшения размерности, или научиться создавать новые синтетические обучающие примеры с аналогичным распределением. RL существенно отличается от обучения с учителем и без учителя, поэтому его часто называют третьей категорией машинного обучения.

Ключевое отличие RL от других методов машинного обучения, таких как обучение с учителем и без учителя, заключается в том, что в основе RL лежит идея *обучения через взаимодействие*. Иными словами, RL-модель учится на опыте взаимодействия с окружающей средой, стремясь максимизировать *функцию вознаграждения* (reward function).

При обучении с учителем максимизация функции вознаграждения связана с минимизацией функции потерь, но при обучении с подкреплением правильные метки для обучения последовательности действий не известны и не определены заранее: вместо этого их необходимо найти посредством взаимодействия с окружающей средой для достижения определенного желаемого результата — например, выигрыша в игре. В процессе RL модель (также называемая *агентом*) взаимодействует со своим окружением и тем самым генерирует последовательность взаимодействий, которые вместе называются *эпизодом*. Вследствие этих взаимодействий агент получает ряд вознаграждений, определяемых средой. Эти вознаграждения могут быть положительными или отрицательными, и иногда они не раскрываются агенту до конца эпизода.

Например, представьте, что мы хотим научить компьютер играть в шахматы и выигрывать у людей. Метки (вознаграждения) за каждый отдельный шахматный ход, сделанный компьютером, неизвестны до конца игры, потому что в процессе самой игры мы, как правило, не знаем, приведет ли конкретный ход к победе или поражению в этой партии. Только в самом конце игры определяется обратная связь. Эта обратная связь будет положительной, если компьютер одержал победу, потому что по совокупности действий агент достиг желаемого результата, и наоборот, принесет отрицательное вознаграждение, если компьютер партию проиграл.

Кроме того, в примере с игрой в шахматы входными данными является текущая конфигурация — например, расположение отдельных шахматных фигур на доске. Учитывая большое количество возможных входных данных (состояний системы), невозможно пометить каждую конфигурацию или состояние как положительное или отрицательное. Поэтому, чтобы завершить процесс обучения, мы предоставляем вознаграждение (или штраф) в конце каждой игры, когда становится известно, достигнут ли желаемый результат — выиграли мы игру или нет.

В этом и заключается суть RL: мы не можем или не хотим напрямую учить агента (компьютер или робота), *как* что-то делать, — мы можем только указать, *чего* должен достичь агент. Затем, исходя из результатов конкретного испытания, мы можем определить вознаграждение в зависимости от успеха или неудачи агента. Это делает RL очень привлекательным для принятия решений в *сложных условиях* (complex environment), особенно когда решение задачи требует ряда шагов, которые неизвестны, труднообъяснимы или трудноопределимы.

Помимо применения в играх и робототехнике, примеры RL также можно найти в природе. Так, дрессировка собаки включает в себя подкрепление — мы выдаем собаке вознаграждение (угощение), когда она выполняет определенные желаемые действия. Или возьмем собаку-ассистента, обученную предупреждать своего владельца о приближающемся припадке. В этом случае мы не знаем точного механизма, с помощью которого собака способна обнаружить приближающийся припадок, и мы, конечно же, не смогли бы рассказать о нем собаке, даже если бы у нас были точные знания об этом механизме. Тем не менее мы можем наградить собаку лакомством, если она успешно обнаружит припадок, чтобы закрепить правильное поведение!

Хотя RL обеспечивает мощную основу изучения произвольной последовательности действий для достижения определенной цели, нельзя забывать, что RL все еще является относительно молодой и активной областью исследований со многими нерешенными проблемами. Один аспект, который делает обучение RL-моделей особенно сложным, заключается в том, что последующие входные данные модели зависят от действий, предпринятых ранее. Это может создать различные проблемы и обычно приводит к нестабильному поведению при обучении. Кроме того, зависимость от последовательности действий создает так называемый *отложененный эффект*, когда за действие, предпринятое на временному шаге t , вознаграждение будет выдано через некоторое произвольное количество шагов позже.

19.1.2. Определение интерфейса агент-среда в системе обучения с подкреплением

Во всех примерах RL по определению присутствуют две отдельные сущности: агент и среда. Формально *агент* определяется как сущность, которая учится принимать решения и взаимодействует с окружающей средой, совершая действия. Вследствие выполнения действия агент получает *наблюдения* и сигнал вознаграждения, определяемый средой. *Среда* — это все, что находится за пределами агента. Среда взаимодействует с агентом и определяет сигнал вознаграждения за действие агента, а также формирует его наблюдения.

Сигнал вознаграждения (reward signal) — это обратная связь, которую агент получает от взаимодействия со средой, обычно она представлена в виде скалярной величины и

может быть как положительной, так и отрицательной. Цель вознаграждения — сообщить агенту, насколько хорошо он работает. Частота, с которой агент получает вознаграждение, зависит от поставленной задачи. Например, в шахматах награда становится известна после завершения игры в зависимости от результата всех ходов: выигрыша или проигрыша. С другой стороны, мы могли бы определить задачу так, чтобы вознаграждение определялось после каждого выполненного действия. В такой последовательности действий агент пытается максимизировать накопленные вознаграждения за все время жизни, равное продолжительности эпизода.

На рис. 19.1 показаны взаимодействия и связь между агентом и средой.

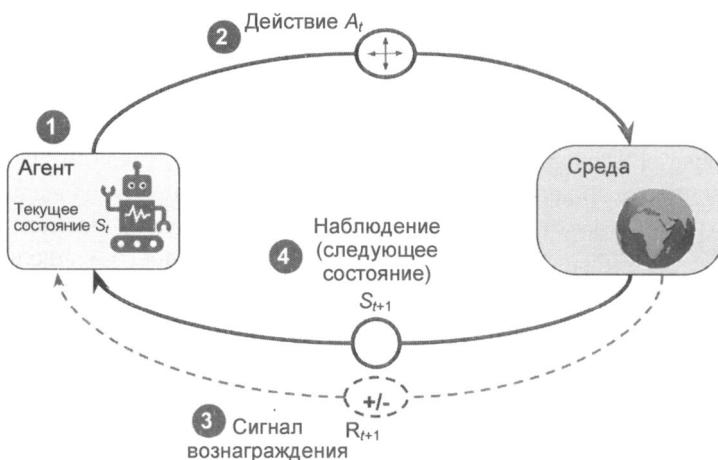


Рис. 19.1. Взаимодействие между агентом и средой

Состояние агента, как можно здесь видеть, представляет собой набор всех его переменных (1). Например, в случае дрона-робота эти переменные могут включать в себя текущее положение дрона (долготу, широту и высоту), оставшееся время автономной работы, скорость вращения каждого пропеллера и т. д. На каждом временному шаге агент взаимодействует со средой через набор доступных действий A_t (2). В зависимости от действия A_t , предпринятоого агентом, когда он находится в состоянии S_t , агент получит сигнал вознаграждения R_{t+1} (3) и перейдет в состояние S_{t+1} (4).

В процессе обучения агент должен пробовать разные действия (*исследование*), чтобы постепенно узнать, какие действия предпочесть и выполнять чаще (*использование*) для максимизации накопительного вознаграждения. Рассмотрим очень простой пример, когда выпускник факультета информатики, специализирующийся на разработке программного обеспечения, задается вопросом, устроиться ли на работу в компанию (*использование знаний*) или пойти в аспирантуру и получить учченую степень, чтобы лучше изучить науку о данных и машинное обучение (*продолжение исследования*). В целом склонность к использованию приведет к выбору действий с большей краткосрочной наградой, тогда как продолжение исследования потенциально может привести к большей общей выгоде в долгосрочной перспективе. Компромисс между исследованием и использованием широко изучен, и все же не существует универсального ответа на эту дилемму принятия решений.

19.2. Теоретические основы RL

Прежде чем перейти к практическим примерам и начать обучение RL-модели, необходимо разобраться в теоретических основах обучения с подкреплением. В следующих разделах мы рассмотрим математическое представление марковских процессов принятия решений, сравним эпизодические и непрерывные задачи, а также познакомимся с ключевой терминологией RL и динамического программирования с использованием уравнения Беллмана. Начнем с марковских процессов принятия решений.

19.2.1. Марковские процессы принятия решений

В общем случае задачи, с которыми имеет дело RL, обычно формулируются как *марковские процессы принятия решений* (Markov Decision Process, MDP). Стандартный подход к решению проблем MDP заключается в использовании динамического программирования, но RL обладает некоторыми ключевыми преимуществами по сравнению с ним.



Динамическое программирование

Динамическим программированием называют набор компьютерных алгоритмов и методов программирования, разработанных Ричардом Беллманом в 1950-х годах. В некотором смысле динамическое программирование связано с рекурсивным решением проблем — решением относительно сложных проблем путем их разбиения на более мелкие подзадачи.

Ключевое различие между рекурсией и динамическим программированием заключается в том, что динамическое программирование сохраняет результаты подзадач (обычно в виде словаря или другой формы справочной таблицы), чтобы к ним можно было получить доступ за постоянное время (вместо их пересчета), если эти подзадачи встречаются в будущем.

К примерам известных задач информатики, которые решаются с помощью динамического программирования, относятся упорядочивание последовательностей и вычисление кратчайшего пути из точки A в точку B.

Однако динамическое программирование неосуществимо на практике, когда размер пространства состояний (т. е. набор возможных конфигураций) относительно велик. В таких случаях обучение с подкреплением считается гораздо более эффективным и практичным альтернативным подходом к решению задач MDP.

Математическая основа марковских процессов принятия решений

Задачи, требующие изучения интерактивного и последовательного процесса принятия решений, когда решение на временному шаге t влияет на последующие ситуации, с математической точки зрения относятся к марковским процессам принятия решений.

В случае взаимодействия агента и среды в RL, если мы обозначим начальное состояние агента как S_0 , взаимодействия между агентом и средой приводят к следующей последовательности:

$$\{S_0, A_0, R_1\}, \{S_1, A_1, R_2\}, \{S_2, A_2, R_3\}, \dots$$

Фигурные скобки в этой записи применяются только для наглядности. Здесь S_t и A_t обозначают состояние и действие, предпринятое на временном шаге t , а R_{t+1} — вознаграждение, полученное от среды после выполнения действия A_t . Кроме того, S_t , R_{t+1} и A_t являются зависящими от времени случайными величинами, которые получают значения из предопределенных конечных множеств, обозначаемых $s \in \hat{S}$, $r \in \hat{R}$ и $a \in \hat{A}$ соответственно. В MDP зависящие от времени случайные величины S_t и R_{t+1} имеют распределения вероятностей, которые зависят только от их значений на предыдущем временном шаге $t - 1$. Распределение вероятностей для $S_{t+1} = s'$ и $R_{t+1} = r$ можно записать как условную вероятность предыдущего состояния (S_t) и предпринятого действия (A_t) следующим образом:

$$p(s', r | s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s', R_{t+1} = r | S_t = s, A_t = a).$$

Это распределение вероятностей полностью определяет *динамику среды* (или модель среды), поскольку на основе этого распределения можно вычислить все вероятности перехода состояния среды. Следовательно, динамика среды является ключевым критерием при классификации различных методов RL. Разновидности методов RL, которые требуют наличия модели среды или пытаются изучить модель среды (т. е. динамику среды), называются *методами, основанными на модели*, в отличие от *методов без моделей*.



RL без моделей и на основе моделей

Когда вероятность $p(s', r | s, a)$ известна, то задача обучения может быть решена с помощью динамического программирования. Но когда динамика среды неизвестна, как это бывает во многих реальных задачах, нам, взаимодействуя с окружающей средой, потребуется получить большое количество образцов, чтобы компенсировать незнание динамики среды.

Двумя основными подходами к решению этой проблемы являются безмодельный метод Монте-Карло (Monte Carlo, MC) и метод временных различий (Temporal Difference, TD). На рис. 19.2 показаны две основные категории и вариации каждого метода.

В этой главе мы рассмотрим различные подходы и их варианты: от теории до практических алгоритмов.

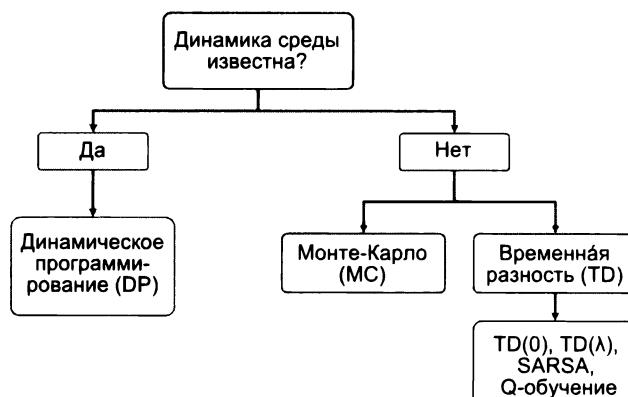


Рис. 19.2. Использование различных моделей в зависимости от динамики среды

Динамику среды можно считать детерминированной, если всегда предпринимаются или никогда не предпринимаются определенные действия для заданных состояний, т. е. $p(s', r|s, a) \in \{0,1\}$. Иначе в более общем случае среда имеет стохастическое поведение.

Чтобы понять, что собой представляет это стохастическое поведение, давайте рассмотрим вероятность наблюдения будущего состояния $S_{t+1} = s'$ при условии, что текущее состояние $S_t = s$ и выполненное действие $A_t = a$. Этую вероятность можно записать так:

$$p(s'|s, a) \stackrel{\text{def}}{=} P(S_{t+1} = s' | S_t = s, A_t = a).$$

Ее можно вычислить как предельную вероятность, взяв сумму по всем возможным вознаграждениям:

$$p(s'|s, a) \stackrel{\text{def}}{=} \sum_{r \in R} p(s', r|s, a).$$

Эта вероятность называется *вероятностью перехода состояния* (state-transition probability). С точки зрения вероятности перехода состояния, если динамика среды детерминирована, это означает, что когда агент совершает действие $A_t = a$ в состоянии $S_t = s$, то переход в следующее состояние $S_{t+1} = s'$ произойдет в 100% случаев, т. е. $p(s'|s, a) = 1$.

Визуальное представление марковского процесса

Марковский процесс можно представить в виде ориентированного циклического графа, в котором узлы графа — это различные состояния среды, а ребра графа (т. е. связи между узлами) — вероятности перехода между состояниями.

В качестве примера возьмем студента, выбирающего между тремя различными ситуациями: (A) готовиться к экзамену дома, (B) играть дома в видеоигры или (C) заниматься в библиотеке. Кроме того, существует конечное состояние (T) — просто лежь спать. Решения принимаются каждый час, и после принятия решения студент остается в выбранной ситуации в течение этого конкретного часа. Допустим, если студент остается дома (состояние A), существует 50-процентная вероятность того, что он переключится на видеоигры. С другой стороны, когда студент находится в состоянии B (играет в видеоигры), существует относительно высокая вероятность (80%) того, что учащийся продолжит играть в видеоигры в последующие часы.

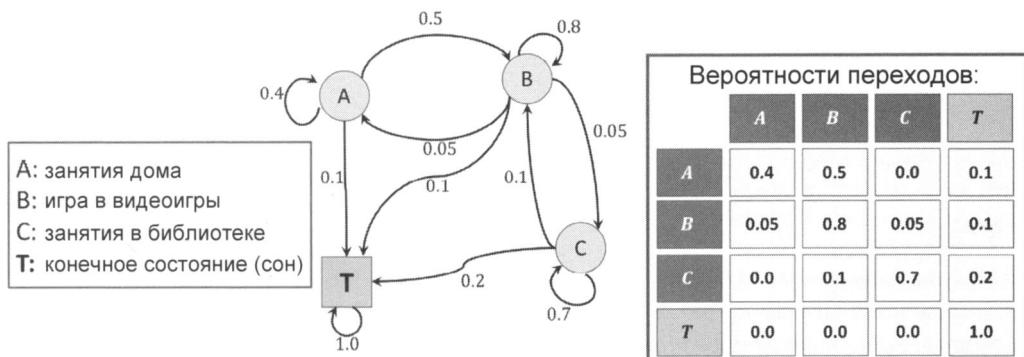


Рис. 19.3. Марковский процесс поведения студента

Динамика поведения студента показана на рис. 19.3 в виде марковского процесса, включающего в себя циклический граф и таблицу переходов.

Значения на ребрах графа представляют собой вероятности перехода состояния студента, и их значения также приведены в таблице справа. Рассматривая строки в таблице, обратите внимание, что вероятности перехода из каждого состояния (узла) всегда в сумме равны 1.

19.2.2. Эпизодические и непрерывные задачи

Когда агент взаимодействует с окружающей средой, последовательность наблюдений или состояний формирует *траекторию*. Существуют два типа траекторий. Если траекторию движения агента можно разделить на части, каждая из которых начинается в момент времени $t = 0$ и заканчивается в конечном состоянии S_T (в момент времени $t = T$), такая задача называется *эпизодической* (episodic task).

С другой стороны, если траектория не имеет разрывов и конечного состояния, такая задача называется *непрерывной* (continuing task).

Задача обучения агента игре в шахматы, является эпизодической, в то время как робот-уборщик, поддерживающий порядок в доме, обычно выполняет непрерывную задачу. В этой главе мы рассматриваем только эпизодические задачи.

В эпизодических задачах эпизод представляет собой последовательность или траекторию, по которой агент переходит из начального состояния S_0 в конечное состояние S_T :

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_t, A_t, R_{t+1}, \dots, S_{t-1}, A_{t-1}, R_t, S_T.$$

Для марковского процесса, показанного на рис. 19.3, где смоделирована задача студента, готовящегося к экзамену, мы можем столкнуться с эпизодами типа следующих трех примеров:

- ◆ эпизод 1: BBCCCCBAT → успех (общее вознаграждение = +1);
- ◆ эпизод 2: ABBBBBBBBBBBT → неудача (общее вознаграждение = -1);
- ◆ эпизод 3: BCCCCCT → успех (общее вознаграждение = +1).

19.2.3. Терминология RL: отдача, стратегия и функция ценности

В этом разделе мы введем дополнительные термины из области обучения с подкреплением, которые понадобятся в оставшейся части этой главы.

Отдача

Так называемая *отдача* (return) в момент времени t — это *накопленное вознаграждение*, полученное за всю продолжительность эпизода. Напомним, что $R_{t+1} = r$ — это *немедленное вознаграждение* (immediate reward), полученное после выполнения действия A_t в момент времени t . За ним могут поступить *последующие вознаграждения* (subsequent reward): R_{t+2}, R_{t+3} и т. д.

Отдачу в момент времени t можно вычислить как из непосредственного вознаграждения, так и из последующих:

$$G_t \stackrel{\text{def}}{=} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

Здесь γ — коэффициент дисконтирования (discount factor) в диапазоне $[0, 1]$. Параметр γ показывает, сколько «стоит» будущее вознаграждение в текущий момент (время t). Если мы задаем $\gamma = 0$, это означает, что нас не волнуют будущие вознаграждения. В этом случае отдача станет равна непосредственному вознаграждению, и агент окажется недальновидным, поскольку игнорирует последующие вознаграждения после $t + 1$. С другой стороны, если $\gamma = 1$, отдача будет представлять собой невзвешенную сумму всех последующих вознаграждений.

Нужно заметить, что уравнение для вычисления отдачи можно записать проще, используя рекурсию:

$$G_t = R_{t+1} + \gamma G_{t+1} = r + \gamma G_{t+1}.$$

Это означает, что отдача в момент времени t равна непосредственному вознаграждению r плюс дисконтированная будущая отдача в момент времени $t + 1$. Это очень важное свойство, которое облегчает расчеты отдачи.



Что такое коэффициент дисконтирования?

Чтобы лучше понять суть коэффициента дисконтирования, взгляните на рис. 19.4, где показана ценность обладания 100-долларовой банкнотой сегодня по сравнению с обладанием ею через год. В определенных экономических ситуациях, таких как инфляция, обладание этой 100-долларовой банкнотой сегодня может иметь для нас большую ценность, чем обладание той же банкнотой в будущем.

Следовательно, мы говорим, что если эта купюра сейчас стоит 100 долларов, то через год она будет стоить 90 долларов с коэффициентом дисконта $\gamma = 0.9$.

Вычислим доходность на разных временных шагах для эпизодов в нашем предыдущем примере со студентом. Предположим, что $\gamma = 0.9$ и что единственное вознаграждение

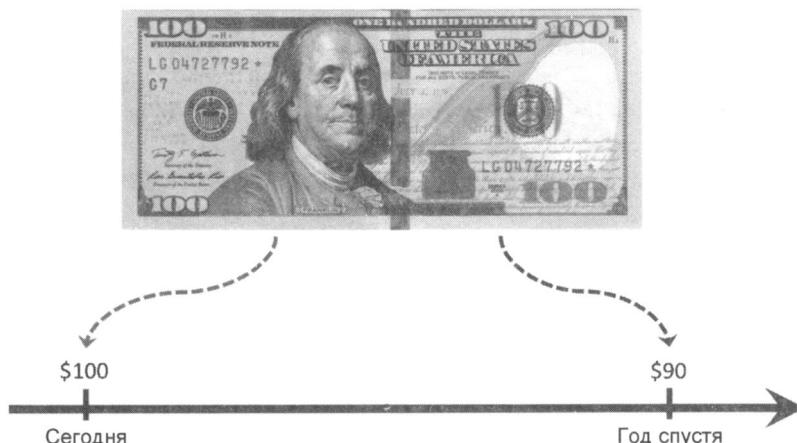


Рис. 19.4. Пример коэффициента дисконтирования, основанного на изменении ценности 100-долларовой банкноты с течением времени

основано на результате экзамена (+1 за сдачу экзамена и -1 за его провал). Награды за промежуточные временные шаги равны 0.

◆ Эпизод 1: ВВCCCCВАТ → успех (общее вознаграждение = +1):

- $t = 0 : G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots + \gamma^6 R_7;$
- $\rightarrow G_0 = 0 + 0 \times \gamma + \dots + 1 \times \gamma^6 = 0.96 \approx 0.531;$
- $t = 1 : G_1 = 1 \times \gamma^5 = 0.590;$
- $t = 2 : G_2 = 1 \times \gamma^4 = 0.656$
- ...
- $t = 6 : G_6 = 1 \times \gamma = 0.9;$
- $t = 7 : G_7 = 1.$

◆ Эпизод 2: АBBBBBBBBBBBT → неудача (общее вознаграждение = -1):

- $t = 0 : G_0 = -1 \times \gamma^8 = -0.430;$
- $t = 1 : G_0 = -1 \times \gamma^7 = -0.478$
- ...
- $t = 8 : G_0 = -1 \times \gamma = -0.9;$
- $t = 9 : G_{10} = -1.$

Мы оставляем вычисление отдачи для третьего эпизода в качестве упражнения для читателя.

Стратегия

Стратегия (policy), обычно обозначаемая как $\pi(a|s)$, представляет собой функцию, определяющую следующее действие, которое необходимо предпринять. Это действие может быть либо детерминированным, либо стохастическим (когда стратегия определяет вероятность выполнения следующего действия). Стохастическая стратегия имеет распределение вероятностей по действиям, которые агент может предпринять в текущем состоянии:

$$\pi(a|s) \triangleq P[A_t = a | S_t = s].$$

В процессе обучения стратегия по мере накопления агентом опыта может меняться. Например, агент может начать с полностью случайной стратегии, где вероятность всех действий одинакова, — а со временем он, как мы надеемся, научится находить оптимальную стратегию. Оптимальная стратегия $\pi_*(a|s)$ приносит наибольшую отдачу.

Функция ценности

Функция ценности (value function), также называемая *функцией ценности состояния* (state-value function), измеряет *благоприятность* (goodness) каждого состояния — другими словами, насколько хорошо или плохо находится в определенном состоянии. Критерием благоприятности состояния является отдача.

Основываясь на отдаче G_t , определим функцию ценности состояния s как ожидаемую отдачу (среднюю доходность по всем возможным эпизодам) после следования стратегии π :

$$v_{\pi}(s) \triangleq E_{\pi}[G_t | S_t = s] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+1} \middle| S_t = s\right].$$

В реальности мы обычно находим значения функции ценности, используя переводные таблицы, поэтому нам не нужно повторно вычислять ее несколько раз. (Это особенность динамического программирования.) Например, на практике, когда мы оцениваем функцию ценности с помощью таких табличных методов, мы сохраняем все значения состояния в таблице, обозначаемой $V(s)$. В коде на языке Python это может быть список или массив NumPy, индексы которого относятся к разным состояниям, или это может быть словарь Python, где ключи словаря сопоставляют состояния с соответствующими значениями.

Кроме того, мы также можем определить значение для каждой пары состояние-действие, которое называется *функцией ценности действия* (action-value function) и обозначается $q_{\pi}(s, a)$. Функция ценности действия соответствует ожидаемой отдаче G_t , когда агент находится в состоянии $S_t = s$ и выполняет действие $A_t = a$.

Распространяя определение функции ценности состояния на пары состояние-действие, мы получаем следующее уравнение:

$$q_{\pi}(s, a) \triangleq E_{\pi}[G_t | S_t = s, A_t = a] = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k+1} R_{t+k+1} \middle| S_t = s, A_t = a\right].$$

Оно аналогично уравнению оптимальной стратегии, поскольку $\pi_*(a | s)$, $v_*(s)$ и $q_*(s, a)$ также обозначают оптимальные функции ценности состояния и ценности действия.

Нахождение функции ценности является важным компонентом методов RL. Позже в этой главе мы рассмотрим различные способы вычисления и оценки функций ценности состояния и ценности действия.



Разница между вознаграждением, отдачей и функцией ценности

Вознаграждение (reward) является следствием действия агента с учетом текущего состояния среды. Другими словами, вознаграждение — это сигнал, который получает агент при выполнении действия по переходу из одного состояния в другое. Однако имейте в виду, что не каждое действие дает положительное или отрицательное вознаграждение, — вспомните наш пример с шахматами, где положительное вознаграждение получается только при победе в игре, а вознаграждение за все промежуточные действия равно нулю.

Состояние само по себе имеет определенную ценность, которую мы ему присваиваем, чтобы измерить, насколько это состояние хорошее или плохое, — здесь в игру вступает *функция ценности* (value function). Как правило, «ценными», или «хорошими», считаются те состояния, которые имеют высокую ожидаемую отдачу (return) и, вероятно, принесут высокое вознаграждение при определенной стратегии.

Например, давайте еще раз обратимся к примеру с компьютером, играющим в шахматы. Положительное вознаграждение может быть дано только в конце игры, и только если компьютер выигрывает игру. Если компьютер проиграет игру, то никакого положительного вознаграждения не будет. Теперь представьте, что компьютер выполняет определенный шахматный ход, который берет ферзя соперника без ка-

ких-либо негативных последствий для компьютера. Поскольку компьютер получает вознаграждение только за победу в игре, он не получает немедленного вознаграждения после хода, позволяющего взять ферзя противника. Однако новое состояние (расстановка фигур после взятия ферзя) может иметь большую ценность, если оно способствует выигрышу компьютера в будущем. Интуитивно мы можем сказать, что высокая ценность, связанная со взятием ферзя противника, связана с тем фактом, что взятие ферзя часто приводит к победе в игре, — и, следовательно, к высокой ожидаемой отдаче или ценности. Однако очевидно, что взятие ферзя противника не всегда приводит к победе — следовательно, агент, скорее всего, получит положительное вознаграждение, но оно не гарантировано.

Короче говоря, отдача — это взвешенная сумма вознаграждений за весь эпизод, которая в нашем примере с шахматами будет равна дисконтируенному финальному вознаграждению (поскольку награда только одна). Функция ценности — это ожидание по всем возможным эпизодам, которое в основном вычисляет, насколько «ценным» в среднем является выполнение определенного хода.

Прежде чем перейти непосредственно к некоторым алгоритмам RL, мы кратко рассмотрим вывод уравнения Беллмана, которое пригодится для реализации оценки стратегии.

19.2.4. Уравнение Беллмана в динамическом программировании

Уравнение Беллмана (Bellman equation) является одним из центральных элементов многих алгоритмов RL. Оно упрощает вычисление функции ценности, потому что вместо суммирования по нескольким временным шагам в нем используется рекурсия, аналогичная рекурсии для вычисления отдачи.

Основываясь на рекурсивном уравнении для общей отдачи: $G_t = r + \gamma G_{t+1}$, мы можем переписать уравнение функции ценности следующим образом:

$$\begin{aligned} v_\pi(s) &\stackrel{\text{def}}{=} E_\pi[G_t | S_t = s] = \\ &= E_\pi[r + \gamma G_{t+1} | S_t = s] = \\ &= r + \gamma E_\pi[G_{t+1} | S_t = s]. \end{aligned}$$

Обратите внимание, что непосредственное вознаграждение r вычитается из ожидания, поскольку оно является постоянной и известной величиной в момент времени t .

Точно так же для функции ценности действия мы могли бы написать:

$$\begin{aligned} q_\pi(s, a) &\stackrel{\text{def}}{=} E_\pi[G_t | S_t = s, A_t = a] = \\ &= E_\pi[r + \gamma G_{t+1} | S_t = s, A_t = a] = \\ &= r + \gamma E_\pi[G_{t+1} | S_t = s, A_t = a]. \end{aligned}$$

Мы можем использовать динамику среды для вычисления ожидания путем суммирования всех вероятностей следующего состояния s' и соответствующих вознаграждений r :

$$v_\pi(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S, r \in R} p(s', r|s, a)[r + \gamma E_\pi[G_{t+1} | S_{t+1} = s']].$$

Теперь мы можем видеть, что ожидание отдачи $E_{\pi}[G_{t+1}|S_t = s']$ по сути является функцией ценности состояния $v_{\pi}(s')$. Это дает нам возможность записать $v_{\pi}(s)$ как функцию $v_{\pi}(s')$:

$$v_{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S, r \in R} p(s', r|s, a) [r + \gamma v_{\pi}(s')].$$

Это и есть уравнение Беллмана, которое связывает функцию ценности состояния s с функцией ценности последующего состояния s' . Оно значительно упрощает вычисление функции ценности, поскольку устраниет повторяющийся цикл по оси времени.

19.3. Алгоритмы обучения с подкреплением

В этом разделе мы рассмотрим несколько алгоритмов обучения с подкреплением. Мы начнем с динамического программирования, которое предполагает, что динамика перехода — или динамика среды, т. е. $p(s', r|s, a)$, — известна. Однако в большинстве задач RL это не так. Чтобы решить проблему неизвестной динамики среды, были разработаны методы RL, которые обучаются через взаимодействие с окружающей средой. К этой категории относятся методы Монте-Карло (MC), обучение на временных разностях (TD) и все более популярные подходы Q-обучения и глубокого Q-обучения.

На рис. 19.5 показана эволюция алгоритмов RL от динамического программирования до Q-обучения.



Рис. 19.5. Различные типы алгоритмов RL

Далее мы рассмотрим каждый из этих алгоритмов. Мы начнем с динамического программирования, затем перейдем к MC и, наконец, к TD и его ответвлениям SARSA (State-Action-Reward-State-Action, состояние-действие-вознаграждение-состояние-действие) и Q-обучению. Вы также познакомитесь с глубоким Q-обучением, когда будете строить некоторые практические модели.

19.3.1. Динамическое программирование

В этом подразделе мы сосредоточимся на решении задач RL при наличии следующих предположений:

- ♦ у нас есть полное знание динамики окружающей среды — т. е. все вероятности перехода $p(s', r|s, a)$ известны;

- ◆ состояние агента обладает марковским свойством — т. е. следующее действие и вознаграждение зависят только от текущего состояния и выбора действия, который мы делаем в этот момент или на текущем временном шаге.

Математическая запись задач RL с использованием марковского процесса принятия решений (MDP) была рассмотрена ранее в этой главе. Если вам нужно освежить знания, обратитесь к разд. «*Математическая основа марковских процессов принятия решений*», в котором представлено формальное определение функции ценности $v_\pi(s)$ в соответствии со стратегией π , и повторите вывод уравнения Беллмана, которое было получено с использованием динамики среды.

Следует подчеркнуть, что динамическое программирование не является практическим подходом к решению задач RL. Проблема с использованием динамического программирования заключается в том, что оно предполагает полное знание динамики среды, что обычно нецелесообразно или неосуществимо для большинства реальных применений. Однако с образовательной точки зрения динамическое программирование помогает упростить изучение RL и служит основой для применения более продвинутых и сложных алгоритмов RL.

Описание задач в следующих подразделах преследует две главные цели:

1. Получить истинную функцию ценности состояния $v_\pi(s)$. Эта задача также известна как задача прогнозирования и решается путем *оценки стратегии* (policy evaluation).
2. Найти оптимальную функцию $v_*(s)$ с помощью итерации по обобщенной стратегии.

Оценка стратегии: прогнозирование функции ценности с помощью динамического программирования

Если известна динамика среды, то на основе уравнения Беллмана можно вычислить функцию ценности для произвольной стратегии π с помощью динамического программирования. Для вычисления этой функции ценности мы можем адаптировать итеративное решение, в котором мы начинаем с функции $v^{(0)}(s)$, инициализированной нулевыми значениями для каждого состояния. Затем на каждой итерации $i + 1$ мы обновляем значения для каждого состояния на базе уравнения Беллмана, которое, в свою очередь, основано на значениях состояний из предыдущей итерации i следующим образом:

$$v^{(i+1)}(s) = \sum_a \pi(a|s) \sum_{s' \in S, r \in R} p(s', r | s, a) [r + \gamma v^{(i)}(s')].$$

Можно показать, что по мере увеличения числа итераций до бесконечности $v^{(i)}(s)$ сходится к истинной функции ценности состояния $v_\pi(s)$.

Не забывайте, что в рассматриваемом случае нам не нужно взаимодействовать с окружающей средой, поскольку мы уже точно знаем динамику окружающей среды. Воспользовавшись этим знанием, легко найти функцию ценности.

После нахождения функции ценности возникает очевидный вопрос: как она может нам пригодиться, если наша стратегия все еще остается случайной? Ответ заключается в том, что мы можем использовать эти вычисленные $v_\pi(s)$ для улучшения стратегии, как будет показано далее.

Улучшение стратегии с помощью известной функции ценности

Итак, мы вычислили функцию ценности $v_\pi(s)$, следуя действующей стратегии π . Теперь можно взять $v_\pi(s)$ и попытаться улучшить действующую стратегию. Это означает, что мы должны найти новую стратегию π' , которая для каждого состояния s давала бы большую или, по крайней мере, равную ценность по сравнению с использованием текущей стратегии π . Математически мы можем записать цель улучшения стратегии следующим образом:

$$v_{\pi'}(s) \geq v_\pi(s) \quad \forall s \in \hat{S}.$$

Напомним, что стратегия π определяет вероятность выбора каждого действия a , пока агент находится в состоянии s . Чтобы найти стратегию π' , которая всегда обеспечивает лучшую или равную ценность для каждого состояния, мы сначала вычисляем функцию ценности действия $q_\pi(s, a)$ для каждого состояния s и действия a на основе ценности состояния, вычисленной с использованием функции ценности $v_\pi(s)$. Мы перебираем все состояния и для каждого состояния s сравниваем ценность следующего состояния s' , которое возникло бы, если выбрать действие a .

Получив наивысшую ценность состояния путем оценивания всех пар состояния-действие с помощью $q_\pi(s, a)$, мы можем сравнить соответствующее действие из этой пары с действием, выбранным текущей стратегией. Если действие, предлагаемое текущей стратегией (т. е. $\arg \max_a \pi(a|s)$), отличается от действия, предлагаемого функци-

ей ценности действия (т. е. $\arg \max_a q_\pi(s, a)$), то мы можем обновить стратегию, изменив вероятности действий, чтобы они соответствовали действию, которое несет наибольшую ценность $q_\pi(s, a)$. Это так называемый *алгоритм совершенствования стратегии*.

Итерация стратегии

Используя алгоритм совершенствования стратегии, можно показать, что он строго приведет к лучшей стратегии, если только текущая стратегия еще не оптимальна (что означает $v_\pi(s) = v_{\pi'}(s) = v_*(s)$ для каждого $s \in \hat{S}$). Следовательно, если мы итеративно выполняем оценку стратегии с последующим улучшением, то гарантированно найдем оптимальную стратегию.



Отметим, что этот метод называется *обобщенной итерацией стратегии* (Generalized Policy Iteration, GPI) и является общим для многих методов RL. Мы воспользуемся GPI в последующих разделах этой главы для методов обучения МС и TD.

Итерация ценности

Мы только что показали, что, повторяя оценку стратегии (вычисляя $v_\pi(s)$ и $q_\pi(s, a)$) и совершенствование стратегии (находя такие π' , что $v_{\pi'}(s) \geq v_\pi(s) \forall s \in \hat{S}$), мы можем достичь оптимальной стратегии. Однако было бы более эффективно объединить обе задачи: оценки и совершенствования стратегии — в один шаг. Следующее уравнение обновляет функцию ценности для итерации $i + 1$ (обозначенной $v^{(i+1)}$) на основе действия,

которое максимизирует взвешенную сумму следующей ценности состояния и его непосредственного вознаграждения ($r + \gamma v^{(i)}(s')$):

$$v^{(i+1)}(s) = \max_a \sum_{s', r} p(s', r | s, a) [r + \gamma v^{(i)}(s')].$$

В этом случае максимальное обновленное значение для $v^{(i+1)}(s)$ достигается за счет выбора наилучшего действия из всех возможных, тогда как при оценке стратегии обновленное значение использовало взвешенную сумму по всем действиям.



Обозначения для табличных оценок функций ценности состояния и ценности действия

В большинстве статей и учебников по RL строчные буквы v_π и q_π используются для обозначения соответственно истинных функций ценности состояния и ценности действия как математических функций в аналитической форме.

В то же время на практике эти функции ценности определяются как переводные таблицы. Табличные оценки этих функций ценности обозначены как $V(S_t = s) \approx v_\pi(s)$ и $Q_\pi(S_t = s, A_t = a) \approx q_\pi(s, a)$. Мы тоже будем использовать эти обозначения далее в главе.

19.3.2. Обучение с подкреплением по методу Монте-Карло

Как было отмечено ранее, динамическое программирование основано на упрощенном предположении, что динамика среды полностью известна. Теперь мы отказываемся от подхода динамического программирования и предполагаем, что у нас нет никаких знаний о динамике среды.

То есть мы не знаем вероятности перехода состояния среды, и нам нужно сделать так, чтобы агент самостоятельно обучался, взаимодействуя с окружающей средой. При использовании методов МС процесс обучения строится на так называемом *симулированном опыте* (*simulated experience*).

Для RL на основе МС мы определяем класс агентов, который следует вероятностной стратегии π , и на основе этой стратегии наш агент выполняет действие на каждом этапе. Последовательность действий составляет симулированный эпизод.

Ранее мы определили функцию ценности состояния, такую, что ценность состояния указывает на ожидаемую отдачу от этого состояния. В динамическом программировании это вычисление опиралось на знание динамики среды, т. е. $p(s', r | s, a)$.

Однако с этого момента мы будем разрабатывать алгоритмы, не требующие динамики среды. Методы на основе МС решают эту задачу, создавая смоделированные эпизоды, в которых агент взаимодействует с окружающей средой. Из этих смоделированных эпизодов мы сможем вычислить среднюю отдачу для каждого состояния, посещенного в этом смоделированном эпизоде.

Нахождение функции ценности состояния с использованием МС

Сначала мы создаем набор эпизодов для каждого состояния s , а затем все эпизоды, которые проходят через заданное состояние s , используются для вычисления ценности

этого состояния. Предположим, что мы задействуем переводную таблицу для получения значения, соответствующего функции ценности $V(S_t = s)$. Обновления МС для оценки функции ценности основаны на общей отдаче, полученной в этом эпизоде, начиная с первого посещения состояния s . Этот алгоритм называется «first-visit Monte Carlo» (метод Монте-Карло с единичным посещением)¹.

Нахождение функции ценности действия с использованием МС

Когда динамика среды известна, мы можем легко вывести функцию ценности действия из функции ценности состояния, просто заглянув на шаг вперед, чтобы найти действие, дающее максимальную ценность, как было показано в разд. 19.3.1. Однако это невозможно, если динамика среды неизвестна.

Чтобы решить эту проблему, мы можем расширить алгоритм нахождения ценности состояния first-visit МС. Например, вычислить предполагаемую отдачу для каждой пары состояния-действие, используя функцию ценности действия. Чтобы получить эту предполагаемую отдачу, мы рассматриваем посещения каждой пары состояния-действие (s, a) , которая относится к посещению состояния s и выполнению действия a .

Однако здесь возникает проблема, поскольку некоторые действия могут никогда не быть выбраны, что приводит к недостаточному исследованию. Есть несколько способов решить эту проблему. Самый простой подход называется *исследовательским стартом* (exploratory start) — он предполагает, что каждая пара состояния-действие имеет ненулевую вероятность в начале эпизода.

Другой подход к решению проблемы отсутствия исследований называется *ϵ -жадной стратегией* (ϵ -greedy policy), о которой будет рассказано далее — в подразделе, посвященном совершенствованию стратегии.

Поиск оптимальной стратегии с помощью МС-управления

МС-управление (MC control) представляет собой процедуру оптимизации для улучшения стратегии. Подобно рассмотренному в разд. 19.3.1 подходу итерации стратегии, мы можем многократно чередовать оценку и улучшение стратегии, пока не достигнем оптимума. То есть, начиная со случайной политики π_0 , процесс чередования оценки политики и улучшения политики можно проиллюстрировать следующим образом:

$$\pi_0 \xrightarrow{\text{Оценка}} q_{\pi_0} \xrightarrow{\text{Улучшение}} \pi_1 \xrightarrow{\text{Оценка}} q_{\pi_1} \xrightarrow{\text{Улучшение}} \pi_2 \dots \xrightarrow{\text{Оценка}} q_* \xrightarrow{\text{Улучшение}} \pi_*$$

Улучшение стратегии: вычисление жадной стратегии на основе функции ценности действия

Используя функцию ценности действия $q(s, a)$, мы можем сгенерировать жадную (детерминированную) стратегию следующим образом:

$$\pi(s) \triangleq \arg \max_a q(s, a).$$

¹ Название этого метода обычно не переводят. Метод Монте-Карло «first-visit» оценивает значение всех состояний как среднее значение результатов после единичных посещений каждого состояния до завершения работы, тогда как метод Монте-Карло «every-visit» усредняет результаты после n посещений. (см. <https://habr.com/ru/companies/otus/articles/477042/>). — Прим. пер.

Чтобы избежать проблемы недостаточного исследования и рассмотреть непосещенные пары состояние-действие, как обсуждалось ранее, мы можем позволить неоптимальным действиям иметь небольшой шанс (ε) быть выбранными. Этот подход называется ε -жадной стратегией, согласно которой все неоптимальные действия в состоянии s имеют минимальную ненулевую вероятность $\frac{\varepsilon}{|A(s)|}$ быть выбранными, а оптимальное действие имеет вероятность $1 - \frac{(|A(s)| - 1) \times \varepsilon}{|A(s)|}$ (вместо 1).

19.3.3. Обучение на временных разностях

Мы уже рассмотрели две фундаментальные методики RL: динамическое программирование и обучение на основе МС. Напомним, что динамическое программирование подразумевает полное и точное знание динамики окружающей среды. С другой стороны, подход на основе МС подразумевает обучение агента на симулированном опыте. В этом подразделе мы представим третий метод RL, называемый обучением на *временных разностях* (Temporal Difference, TD), который можно рассматривать как улучшение или расширение подхода RL на основе МС.

Подобно МС, метод TD также основан на обучении на основе опыта и, следовательно, не требует каких-либо знаний о динамике окружающей среды и вероятностях перехода. Основное различие между методами TD и МС заключается в том, что в МС мы должны дождаться окончания эпизода, чтобы иметь возможность рассчитать общую отдачу.

В то же время при TD-обучении мы можем использовать некоторые изученные свойства для обновления оценочных значений до достижения конца эпизода. Этот прием называется *бутстрэпом* (бустрэп в контексте RL не следует путать с бутстрэп-агрегированием, которое мы использовали в главе 7).

По аналогии с динамическим программированием и обучением на основе МС мы рассмотрим две задачи: нахождение функции ценности (которую также называют *прогнозированием ценности*) и улучшение стратегии (которую также называют *задачей управления*).

Прогнозирование ценности с TD

Давайте сначала вернемся к прогнозированию ценности с МС. В конце каждого эпизода мы можем оценить отдачу G_t для каждого временного шага t и обновить наши оценки посещенных состояний:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)).$$

Здесь G_t используется в качестве *целевой отдачи* (target return) для обновления оценочных значений, а $(G_t - V(S_t))$ является поправочным членом, добавляемым к нашей текущей оценке значения $V(S)$. Множитель α — это гиперпараметр, обозначающий скорость обучения, которая остается постоянной во время обучения.

Напомним, что в поправке МС используется *фактическая* отдача G_t , которая неизвестна до конца эпизода. Для большей ясности мы можем переименовать фактическую отдачу G_t в $G_{t:T}$, где нижний индекс ($t:T$) указывает, что это отдача, полученная на

временным шаге t при рассмотрении всех событий, произошедших с шага t до завершающего шага T .

В TD-обучении мы заменяем фактическую отдачу $G_{t:T}$ новой целевой отдачей $G_{t:t+1}$, что значительно упрощает обновление функции ценности $V(S_t)$. Формула обновления на основе TD-обучения выглядит следующим образом:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_{t:t+1} - V(S_t)).$$

Здесь целевая отдача:

$$G_{t:t+1} \stackrel{\text{def}}{=} R_{t+1} + \gamma V(S_{t+1}) = r + \gamma V(S_{t+1})$$

использует наблюдаемое вознаграждение R_{t+1} и ожидаемую ценность следующего непосредственного шага. Обратите внимание на разницу между МС и TD. В методе МС значение $G_{t:T}$ недоступно до конца эпизода, поэтому мы должны выполнить столько шагов, сколько необходимо, чтобы добраться туда. Наоборот, чтобы получить целевую отдачу в TD нам нужно сделать только один шаг вперед. Поэтому соответствующий алгоритм известен под названием TD(0).

Более того, алгоритм TD(0) можно обобщить до так называемого n -шагового алгоритма TD, который включает в себя большее количество будущих шагов — точнее, взвешенную сумму n будущих шагов. Если мы зададим $n = 1$, то n -шаговая процедура TD идентична TD(0), упомянутой в предыдущем абзаце. Однако, если $n \rightarrow \infty$, n -шаговый алгоритм TD будет таким же, как и алгоритм МС. Правило обновления n -шагового TD выглядит следующим образом:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_{t:t+n} - V(S_t)),$$

где $G_{t:t+n}$ определяется так:

$$G_{t:t+n} \stackrel{\text{def}}{=} \begin{cases} R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) & \text{if } t + n < T \\ G_{t:T} & \text{в ином случае} \end{cases}$$



МС или TD: какой метод сходится быстрее?

Хотя точного ответа на этот вопрос до сих пор нет, на практике эмпирически показано, что TD может сходиться быстрее, чем МС. Если вас интересует этот вопрос, вы можете найти более подробную информацию о сходимости МС и TD в книге Ричарда Саттона и Эндрю Барто «Reinforcement Learning: An Introduction».

Разобравшись с задачей прогнозирования с использованием алгоритма TD, мы можем перейти к задаче управления и рассмотрим два алгоритма управления TD: *on-policy* и *off-policy*². В обоих случаях мы применяем обобщенную итерацию по стратегии (GPI), которую до этого задействовали как в алгоритмах динамического программирования, так и в алгоритмах МС. В алгоритме *on-policy* TD функция ценности обновляется на основе действий из той же стратегии, которой следует агент, в то время как в алгоритме *off-policy* TD функция ценности обновляется на основе действий вне текущей стратегии.

² В российской литературе эти термины не переводят. — Прим. пер.

Алгоритм on-policy TD (SARSA)

Для простоты мы рассматриваем только одношаговый алгоритм TD, или TD(0). Однако алгоритм on-policy TD может быть легко обобщен на n -шаговый TD. Мы начнем с расширения формулы прогнозирования для определения функции ценности состояния, применяемой затем для описания функции ценности действия. Для этого мы воспользуемся переводной таблицей, т. е. табличным двумерным массивом $Q(S_t, A_t)$, который представляет собой функцию ценности действия для каждой пары состояния-действие. В этом случае мы получим следующую формулу обновления:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)].$$

Этот алгоритм часто называют SARSA по буквам членов $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, которые присутствуют в формуле обновления.

Как было сказано в предыдущих разделах, описывающих алгоритмы динамического программирования и МС, мы можем использовать методику GPI и, начав со случайной стратегии, многократно вычислить функцию ценности действия для текущей стратегии, а затем оптимизировать эту стратегию, применив ϵ -жадную стратегию, основанную на текущей функции ценности действия.

Алгоритм off-policy TD (Q-обучение)

При использовании алгоритма on-policy TD наш расчет функции ценности действия основывался на стратегии, применяемой в моделируемом эпизоде. После обновления функции ценности действия делался отдельный шаг по улучшению стратегии путем выполнения действия с более высокой ценностью.

Альтернативный (и лучший) подход состоит в объединении этих двух шагов. Другими словами, представьте, что агент следует политике π , генерируя эпизод с текущей пятеркой переходов $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Но вместо обновления функции ценности действия с использованием ценности действия A_{t+1} , предпринимаемого агентом, мы можем найти наилучшее действие, даже если оно на самом деле не выбрано агентом в соответствии с текущей стратегией. (Вот почему этот алгоритм называется off-policy — буквально «вне стратегии».)

Для этого мы изменим правило обновления, чтобы учитывать максимальное значение Q , варьируя различные действия в следующем непосредственном состоянии. Модифицированное уравнение для обновления значений Q выглядит так:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right].$$

Сравните это правило обновления с правилом алгоритма SARSA. Как видите, мы находим лучшее действие в следующем состоянии S_{t+1} и используем его в поправочном компоненте для обновления нашей оценки $Q(S_t, A_t)$.

Для лучшего понимания этого материала в следующем разделе мы обсудим реализацию алгоритма Q-обучения для решения классической задачи *Grid World* («задача клетчатого мира»).

19.4. Реализация первого алгоритма RL

В этом разделе мы рассмотрим реализацию алгоритма Q-обучения для решения задачи клетчатого мира Grid World (*клетчатый мир* — это двумерная среда, состоящая из квадратных ячеек, в которой агент движется в четырех направлениях, чтобы получить как можно больше вознаграждений)³. Для решения этой задачи мы воспользуемся набором инструментов OpenAI Gym.

19.4.1. Знакомство с набором инструментов OpenAI Gym

OpenAI Gym — это специализированный набор инструментов для облегчения разработки моделей RL. OpenAI Gym поставляется с несколькими предварительно определенными средами. К базовым примерам относятся CartPole (задача с перевернутым маятником) и MountainCar (задача о горном автомобиле), где цель агента соответственно заключается в том, чтобы балансировать перевернутым маятником и помочь автомобилю заехать в гору. Существует также множество продвинутых робототехнических сред для обучения робота брать, толкать и тянуть находящиеся на подставке предметы или обучать роботизированную руку ориентировать блоки, мячи или ручки. Более того, OpenAI Gym предоставляет удобную унифицированную платформу для разработки новых сред⁴.

Чтобы работать с примерами кода OpenAI Gym, приведенными в этом разделе, вам необходимо установить библиотеку gym (на момент подготовки книги была доступна версия 0.20.0), что можно легко сделать с помощью pip⁵:

```
pip install gym==0.20
```

Работа с готовыми средами в OpenAI Gym

В качестве упражнения по работе с готовыми средами Gym давайте создадим среду из CartPole-v1, которая уже существует в OpenAI Gym. В этом примере (рис. 19.6) к *тележке* (cart), которая может двигаться горизонтально, на шарнире прикреплен *обратный маятник* (pole).

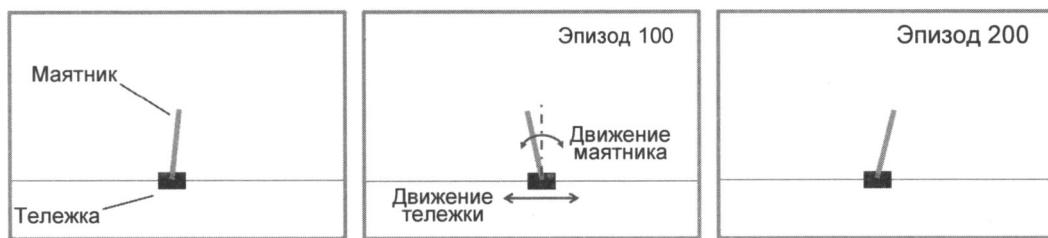


Рис. 19.6. Пример CartPole в Gym

³ В теории принятия решений также часто встречается более сложная версия этой задачи под названием Hex World — мир, составленный из шестиугольных ячеек по типу пчелиных сот (гексамир). В таком мире у агента не четыре, а шесть возможных направлений движения. — Прим. пер.

⁴ Более подробную информацию можно найти на официальном сайте этого набора инструментов по адресу: <https://gym.openai.com/>.

⁵ Если вам нужна дополнительная помощь по установке OpenAI Gym, обратитесь к соответствующему официальному руководству по адресу: <https://gym.openai.com/docs/#installation>.

Движения маятника определяются законами физики, и цель агентов RL — научиться двигать тележку таким образом, чтобы удерживать маятник в вертикальном положении и предотвратить его опрокидывание в любую сторону.

Рассмотрим некоторые свойства среды CartPole в контексте RL — такие как пространство состояния (или наблюдения), пространство действия и способ выполнения действия:

```
>>> import gym
>>> env = gym.make('CartPole-v1')
>>> env.observation_space
Box(-3.4028234663852886e+38, 3.4028234663852886e+38, (4,), float32)
>>> env.action_space
Discrete(2)
```

В этом коде мы создали среду для задачи CartPole. Пространством наблюдения для этой среды является `Box(4,)` (со значениями с плавающей запятой от `-inf` до `inf`), которое представляет собой четырехмерное пространство, соответствующее четырем действительным числам: положение тележки, скорость тележки, угол наклона маятника и скорость кончика маятника. Пространство действий — дискретное пространство `Discrete(2)` с двумя вариантами: толкать тележку либо влево, либо вправо.

Объект среды `env`, созданный вызовом `gym.make('CartPole-v1')`, имеет метод `reset()`, который можно использовать для повторной инициализации среды перед каждым эпизодом. Вызов метода `reset()` в основном устанавливает начальное состояние маятника (S_0):

```
>>> env.reset()
array([-0.03908273, -0.00837535, 0.03277162, -0.0207195])
```

Значения в массиве, возвращаемом вызовом метода `env.reset()`, означают, что начальное положение тележки равно -0.039 , скорость -0.008 , угол маятника равен 0.033 радиана, а угловая скорость его кончика составляет -0.021 . При вызове метода `reset()` эти значения инициализируются случайными значениями с равномерным распределением в диапазоне $[-0.05, 0.05]$.

После сброса среды мы можем взаимодействовать с ней, выбрав и выполнив действие, для чего передадим действие методу `step()`:

```
>>> env.step(action=0)
(array([-0.03925023, -0.20395158, 0.03235723, 0.28212046]), 1.0, False, {})
>>> env.step(action=1)
(array([-0.04332927, -0.00930575, 0.03799964, -0.00018409]), 1.0, False, {})
```

С помощью двух команд: `env.step(action=0)` и `env.step(action=1)` — мы подтолкнули тележку сначала влево (`action=0`) и затем вправо (`action=1`) соответственно. Тележка и ее маятник будут двигаться в зависимости от выбранного действия и в соответствии с законами физики. Каждый раз, когда мы вызываем метод `env.step()`, он возвращает кортеж, состоящий из четырех элементов:

- ◆ массив для нового состояния (или наблюдений);
- ◆ вознаграждение (скалярное значение типа `float`);

- ◆ флаг завершения (True или False).
- ◆ словарь Python, содержащий вспомогательную информацию.

У объекта `env` также есть метод `render()`, который мы можем выполнять после каждого шага (или серии шагов), чтобы визуализировать окружающую среду и движения маятника и тележки во времени.

Эпизод заканчивается, когда угол наклона маятника становится больше 12 градусов (с любой стороны) относительно воображаемой вертикальной оси или когда смещение тележки превышает 2.4 единицы расстояния от центрального положения. Награда, определенная в этом примере, состоит в том, чтобы максимизировать время, в течение которого тележка и маятник стабилизированы в допустимых областях, — другими словами, общая награда (т. е. отдача) может быть максимизирована за счет максимизации продолжительности эпизода.

Пример клетчатого мира

Познакомившись в качестве разминки со средой `CartPole`, для дальнейшей работы с набором инструментов OpenAI Gym мы рассмотрим пример клетчатого мира, представляющего собой упрощенную среду из m строк и n столбцов. Приняв $m = 5$ и $n = 6$, мы можем показать вам эту среду (рис. 19.7).



Рис. 19.7. Пример клетчатого мира

В среде клетчатого мира существуют 30 различных возможных состояний. Четыре из них являются конечными состояниями: горшок с золотом (клад) — в состоянии 16 и три ловушки — в состояниях 10, 15 и 22. Достижение любого из этих четырех конечных состояний завершит эпизод, но с разным результатом: клад или ловушка. Попадание на ячейку с кладом дает положительное вознаграждение +1, тогда как попадание в одно из состояний-ловушек дает отрицательное вознаграждение -1. Все остальные состояния дают нулевое вознаграждение. Агент всегда начинает с состояния 0 — т. е. каждый раз, когда мы сбрасываем среду, агент будет возвращаться в состояние 0. Пространство действий состоит из четырех направлений: вверх, вниз, влево и вправо.

Если агент достигает границы среды, то попытка совершить действие, которое приведет к выходу из среды, не изменит состояние.

Далее мы покажем, как реализовать эту среду на Python с помощью пакета OpenAI Gym.

Реализация среды клетчатого мира в OpenAI Gym

Для экспериментов со средой клетчатого мира через OpenAI Gym настоятельно рекомендуется использовать редактор сценариев или IDE, а не выполнять код в интерактивном режиме.

Сначала мы создадим новый скрипт Python с именем `gridworld_env.py`, а затем импортируем необходимые пакеты и две вспомогательные функции, которые определим для построения визуализации среды.

Для построения визуального представления сред библиотека OpenAI Gym использует библиотеку `pyglet` и предоставляет удобные классы-оболочки и функции. Мы задействуем эти классы-оболочки⁶ для визуализации среды клетчатого мира в следующем примере кода:

```
## Скрипт: gridworld_env.py
import numpy as np
from gym.envs.toy_text import discrete
from collections import defaultdict
import time
import pickle
import os
from gym.envs.classic_control import rendering

CELL_SIZE = 100
MARGIN = 10

def get_coords(row, col, loc='center'):
    xc = (col+1.5) * CELL_SIZE
    yc = (row+1.5) * CELL_SIZE
    if loc == 'center':
        return xc, yc
    elif loc == 'interior_corners':
        half_size = CELL_SIZE//2 - MARGIN
        xl, xr = xc - half_size, xc + half_size
        yt, yb = xc - half_size, xc + half_size
        return [(xl, yt), (xr, yt), (xr, yb), (xl, yb)]
    elif loc == 'interior_triangle':
        x1, y1 = xc, yc + CELL_SIZE//3
        x2, y2 = xc + CELL_SIZE//3, yc - CELL_SIZE//3
        x3, y3 = xc - CELL_SIZE//3, yc - CELL_SIZE//3
        return [(x1, y1), (x2, y2), (x3, y3)]

def draw_object(coords_list):
    if len(coords_list) == 1: # -> круг
        obj = rendering.make_circle(int(0.45*CELL_SIZE))
        obj_transform = rendering.Transform()
```

⁶ Более подробную информацию об этих классах-оболочках можно найти по адресу:

https://github.com/openai/gym/blob/58ed658d9b15fd410c50d1fdb25a7cad9acb7fa4/gym/envs/classic_control/rendering.py.

```

obj.add_attr(obj_transform)
obj_transform.set_translation(*coords_list[0])
obj.set_color(0.2, 0.2, 0.2) # -> черный
elif len(coords_list) == 3: # -> треугольник
    obj = rendering.FilledPolygon(coords_list)
    obj.set_color(0.9, 0.6, 0.2) # -> желтый
elif len(coords_list) > 3: # -> многоугольник
    obj = rendering.FilledPolygon(coords_list)
    obj.set_color(0.4, 0.4, 0.8) # -> синий
return obj

```



Использование OpenAI Gym 0.22 или новее

Имейте в виду, что на момент подготовки книги разработчики проводят внутреннюю реструктуризацию модуля gym. В версии 0.22 и новее вам может потребоваться обновить предыдущий пример кода (из `gridworld_env.py`) и заменить строку:

```
from gym.envs.classic_control import rendering
```

таким кодом:

```
from gym.utils import pyglet_rendering
```

Дополнительные сведения вы найдете в репозитории кода по адресу:
<https://github.com/rasbt/machine-learning-book/tree/main/ch19>.

Первая вспомогательная функция: `get_coords()` — возвращает координаты геометрических фигур, которые мы будем использовать для аннотирования среды клетчатого мира, — например, треугольник для отображения клада или круги для отображения ловушек. Список координат передается функции `draw_object()`, которая решает нарисовать круг, треугольник или многоугольник на основе длины входного списка координат.

Настало время реализовать среду клетчатого мира. В том же файле (`gridworld_env.py`) мы применяем класс с именем `GridWorldEnv`, который наследуется от класса `DiscreteEnv` OpenAI Gym. Наиболее важной функцией этого класса является метод-конструктор `__init__()`, в котором мы определяем пространство действий, указываем роль каждого действия и задаем конечные состояния (клад и ловушки):

```

class GridWorldEnv(discrete.DiscreteEnv):
    def __init__(self, num_rows=4, num_cols=6, delay=0.05):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self.delay = delay
        move_up = lambda row, col: (max(row - 1, 0), col)
        move_down = lambda row, col: (min(row + 1, num_rows - 1), col)
        move_left = lambda row, col: (row, max(col - 1, 0))
        move_right = lambda row, col: (row, min(col + 1, num_cols - 1))
        self.action_defs = {0: move_up, 1: move_right,
                           2: move_down, 3: move_left}
        # Количество состояний/действий
        nS = num_cols * num_rows
        nA = len(self.action_defs)
        self.grid2state_dict = {(s // num_cols, s % num_cols): s
                               for s in range(nS)}

```

```
self.state2grid_dict = {s: (s // num_cols, s % num_cols)
                       for s in range(nS)}

# Состояние: клад
gold_cell = (num_rows // 2, num_cols - 2)

# Состояние: ловушки
trap_cells = [(gold_cell[0] + 1, gold_cell[1]),
               (gold_cell[0], gold_cell[1] - 1),
               (gold_cell[0] - 1, gold_cell[1])]

gold_state = self.grid2state_dict[gold_cell]
trap_states = [self.grid2state_dict[(r, c)]
               for (r, c) in trap_cells]
self.terminal_states = [gold_state] + trap_states
print(self.terminal_states)
# Построение вероятности перехода
P = defaultdict(dict)
for s in range(nS):
    row, col = self.state2grid_dict[s]
    P[s] = defaultdict(list)
    for a in range(nA):
        action = self.action_defs[a]
        next_s = self.grid2state_dict[action(row, col)]
        # Конечное состояние
        if self.is_terminal(next_s):
            r = (1.0 if next_s == self.terminal_states[0]
                 else -1.0)
        else:
            r = 0.0
        if self.is_terminal(s):
            done = True
            next_s = s
        else:
            done = False
        P[s][a] = [(1.0, next_s, r, done)]
# Распределение начального состояния
isd = np.zeros(nS)
isd[0] = 1.0
super().__init__(nS, nA, P, isd)
self.viewer = None
self._build_display(gold_cell, trap_cells)

def is_terminal(self, state):
    return state in self.terminal_states

def _build_display(self, gold_cell, trap_cells):
    screen_width = (self.num_cols + 2) * CELL_SIZE
    screen_height = (self.num_rows + 2) * CELL_SIZE
```

```
self.viewer = rendering.Viewer(screen_width,
                               screen_height)

all_objects = []
# Список координат внешней границы
bp_list = [
    (CELL_SIZE - MARGIN, CELL_SIZE - MARGIN),
    (screen_width - CELL_SIZE + MARGIN, CELL_SIZE - MARGIN),
    (screen_width - CELL_SIZE + MARGIN,
     screen_height - CELL_SIZE + MARGIN),
    (CELL_SIZE - MARGIN, screen_height - CELL_SIZE + MARGIN)
]
border = rendering.PolyLine(bp_list, True)
border.set_linewidth(5)
all_objects.append(border)

# Вертикальные линии
for col in range(self.num_cols + 1):
    x1, y1 = (col + 1) * CELL_SIZE, CELL_SIZE
    x2, y2 = (col + 1) * CELL_SIZE, \
              (self.num_rows + 1) * CELL_SIZE
    line = rendering.PolyLine([(x1, y1), (x2, y2)], False)
    all_objects.append(line)

# Горизонтальные линии
for row in range(self.num_rows + 1):
    x1, y1 = CELL_SIZE, (row + 1) * CELL_SIZE
    x2, y2 = (self.num_cols + 1) * CELL_SIZE, \
              (row + 1) * CELL_SIZE
    line = rendering.PolyLine([(x1, y1), (x2, y2)], False)
    all_objects.append(line)

# Ловушки: --> круги
for cell in trap_cells:
    trap_coords = get_coords(*cell, loc='center')
    all_objects.append(draw_object([trap_coords]))

# Клад: --> треугольник
gold_coords = get_coords(*gold_cell,
                         loc='interior_triangle')
all_objects.append(draw_object(gold_coords))

# Агент: --> квадрат или робот
if (os.path.exists('robot-coordinates.pkl') and CELL_SIZE == 100):
    agent_coords = pickle.load(
        open('robot-coordinates.pkl', 'rb'))
    starting_coords = get_coords(0, 0, loc='center')
    agent_coords += np.array(starting_coords)
else:
    agent_coords = get_coords(0, 0, loc='interior_corners')
```

```

agent = draw_object(agent_coords)
self.agent_trans = rendering.Transform()
agent.add_attr(self.agent_trans)
all_objects.append(agent)

for obj in all_objects:
    self.viewer.add_geom(obj)

def render(self, mode='human', done=False):
    if done:
        sleep_time = 1
    else:
        sleep_time = self.delay
    x_coord = self.s % self.num_cols
    y_coord = self.s // self.num_cols
    x_coord = (x_coord + 0) * CELL_SIZE
    y_coord = (y_coord + 0) * CELL_SIZE
    self.agent_trans.set_translation(x_coord, y_coord)
    rend = self.viewer.render(
        return_rgb_array=(mode == 'rgb_array'))
    time.sleep(sleep_time)
    return rend

def close(self):
    if self.viewer:
        self.viewer.close()
    self.viewer = None

```

Приведенный код формирует среду клетчатого мира, из которой мы можем создавать экземпляры этой среды. После чего с ним можно взаимодействовать так же, как в примере с CartPole. Реализованный класс GridWorldEnv наследует такие методы, как `reset()` (для сброса состояния) и `step()` (для выполнения действия). Обратите внимание на следующие нюансы реализации:

- ◆ мы определили четыре различных действия с помощью лямбда-функций: `move_up()`, `move_down()`, `move_left()` и `move_right()`;
- ◆ массив `isd` NumPy содержит вероятности начальных состояний, так что случайное состояние будет выбрано на основе этого распределения при вызове метода `reset()` (из родительского класса). Поскольку мы всегда начинаем с состояния 0 (нижний левый угол клетчатого мира), мы присваиваем состоянию 0 вероятность 1.0, а всем остальным 29 состояниям — вероятность 0.0;
- ◆ вероятности перехода, определенные в словаре Python `P`, определяют вероятности перемещения из одного состояния в другое при выборе действия. Это позволяет нам получить вероятностную среду, в которой действие может иметь разные результаты в зависимости от стохастичности среды. Для простоты мы просто используем единственный результат, который заключается в изменении состояния в направлении выбранного действия. Наконец, функция `env.step()` использует эти вероятности перехода для определения следующего состояния;

- ◆ функция `_build_display()` создает первоначальное отображение среды, а функция `render()` показывает перемещения агента.



Обратите внимание, что в процессе обучения мы не знаем о вероятностях перехода, и цель состоит в том, чтобы учиться, взаимодействуя с окружающей средой. Следовательно, у нас нет доступа к `P` вне определения класса.

Протестируем этот код, создав новую среду и визуализировав случайный эпизод после выполнения случайных действий в каждом состоянии. Включите следующий код в конец того же скрипта Python (`gridworld_env.py`), а затем запустите скрипт:

```
if __name__ == '__main__':
    env = GridWorldEnv(5, 6)
    for i in range(1):
        s = env.reset()
        env.render(mode='human', done=False)
    while True:
        action = np.random.choice(env.nA)
        res = env.step(action)
        print('Action ', env.s, action, ' -> ', res)
        env.render(mode='human', done=res[2])
        if res[2]:
            break
    env.close()
```

Выполнив скрипт, вы должны увидеть визуальное представление клетчатого мира, как показано на рис. 19.8.

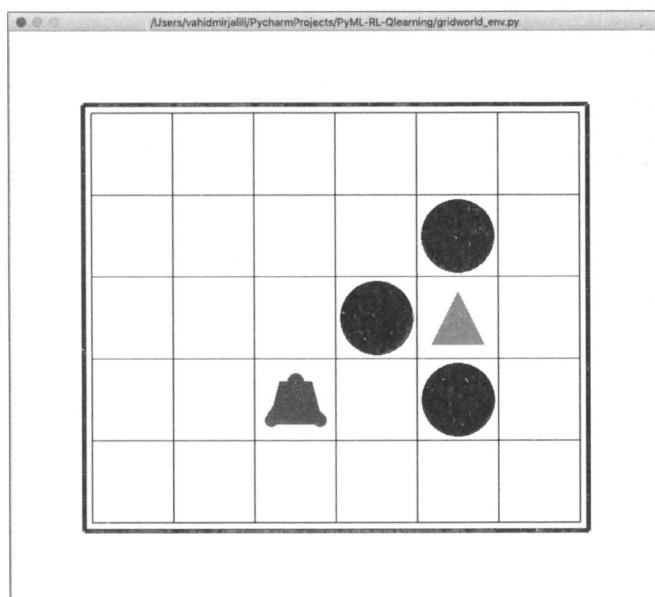


Рис. 19.8. Визуализация среды клетчатого мира

19.4.2. Решение задачи клетчатого мира с помощью Q-обучения

Познакомившись с теорией и процессом разработки алгоритмов RL, а также научившись настраивать среду с помощью набора инструментов OpenAI Gym, мы приступим к реализации самого популярного в настоящее время алгоритма RL — Q-обучения. Для этого воспользуемся примером клетчатого мира, который мы уже реализовали в скрипте `gridworld_env.py`.

Создайте новый скрипт и назовите его `agent.py`. В этом скрипте мы определяем агента для взаимодействия со средой:

```
# Скрипт: agent.py
from collections import defaultdict
import numpy as np

class Agent:
    def __init__(self, env,
                 learning_rate=0.01,
                 discount_factor=0.9,
                 epsilon_greedy=0.9,
                 epsilon_min=0.1,
                 epsilon_decay=0.95):
        self.env = env
        self.lr = learning_rate
        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay

    # Определение q_table
    self.q_table = defaultdict(lambda: np.zeros(self.env.nA))

    def choose_action(self, state):
        if np.random.uniform() < self.epsilon:
            action = np.random.choice(self.env.nA)
        else:
            q_vals = self.q_table[state]
            perm_actions = np.random.permutation(self.env.nA)
            q_vals = [q_vals[a] for a in perm_actions]
            perm_q_argmax = np.argmax(q_vals)
            action = perm_actions[perm_q_argmax]
        return action

    def _learn(self, transition):
        s, a, r, next_s, done = transition
        q_val = self.q_table[s][a]
        if done:
            q_target = r
        else:
            q_target = r + self.gamma * np.max(self.q_table[next_s])
```

```

else:
    q_target = r + self.gamma*np.max(self.q_table[next_s])

    # Обновление q_table
    self.q_table[s][a] += self.lr * (q_target - q_val)

    # Подгонка epsilon
    self._adjust_epsilon()

def _adjust_epsilon(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

```

Конструктор `_init_()` задает различные гиперпараметры — такие как скорость обучения, коэффициент дисконтирования γ и параметры для ϵ -жадной политики. Изначально мы начинаем с высокого значения ϵ , но метод `_adjust_epsilon()` уменьшает его, пока оно не достигнет минимального значения ϵ_{\min} . Метод `choose_action()` выбирает действие на основе ϵ -жадной политики следующим образом: из равномерного распределения выбирается случайное число для определения того, должно ли действие быть выбрано случайным образом или же на основе функции ценности действия. Метод `_learn()` реализует правило обновления для алгоритма Q-обучения. Он получает кортеж для каждого перехода, который состоит из текущего состояния (s), выбранного действия (a), наблюдаемого вознаграждения (r), следующего состояния (s'), а также флага для определения того, достигнут ли конец эпизода. Целевая ценность равна наблюдаемому вознаграждению (r), если установлен флаг конца эпизода, в противном случае она равна:

$$r + \gamma \max_a Q(s', a).$$

В завершение мы можем собрать воедино все компоненты и создать новый скрипт: `qlearning.py` — для обучения агента с помощью алгоритма Q-обучения.

В следующем коде мы определяем функцию `run_qlearning()`, которая реализует алгоритм Q-обучения, имитируя эпизод путем вызова `_choose_action()` агента и выполнения среды. Затем кортеж перехода передается в метод `_learn()` агента для обновления функции ценности действия. Кроме того, для наблюдения за процессом обучения мы также храним итоговое вознаграждение за каждый эпизод (которое может принимать значения -1 или $+1$), а также продолжительность эпизодов (количество ходов, сделанных агентом с начала эпизода и до конца).

После чего список вознаграждений и количество ходов отображаются с помощью функции `plot_learning_history()`:

```

# Скрипт: qlearning.py
from gridworld_env import GridWorldEnv
from agent import Agent
from collections import namedtuple
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(1)

Transition = namedtuple(
    'Transition', ('state', 'action', 'reward', 'next_state', 'done'))

```

```
def run_qlearning(agent, env, num_episodes=50):
    history = []
    for episode in range(num_episodes):
        state = env.reset()
        env.render(mode='human')
        final_reward, n_moves = 0.0, 0
        while True:
            action = agent.choose_action(state)
            next_s, reward, done, _ = env.step(action)
            agent._learn(Transition(state, action, reward,
                                    next_s, done))
            env.render(mode='human', done=done)
            state = next_s
            n_moves += 1
            if done:
                break
        final_reward = reward
        history.append((n_moves, final_reward))
        print(f'Episode {episode}: Reward {final_reward:.2} #Moves {n_moves}')

    return history

def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 10))
    ax = fig.add_subplot(2, 1, 1)
    episodes = np.arange(len(history))
    moves = np.array([h[0] for h in history])
    plt.plot(episodes, moves, lw=4,
             marker="o", markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('# moves', size=20)

    ax = fig.add_subplot(2, 1, 2)
    rewards = np.array([h[1] for h in history])
    plt.step(episodes, rewards, lw=4)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('Final rewards', size=20)
    plt.savefig('q-learning-history.png', dpi=300)
    plt.show()

if __name__ == '__main__':
    env = GridWorldEnv(num_rows=5, num_cols=6)
    agent = Agent(env)
    history = run_qlearning(agent, env)
    env.close()

plot_learning_history(history)
```

Выполнение этого скрипта запустит программу Q-обучения на 50 эпизодов. Поведение агента будет визуализировано, и вы сможете увидеть, что в начале процесса обучения агент относительно часто попадает в состояния ловушки. Но со временем он учится на своих ошибках и в конце концов находит клад (например, впервые в эпизоде 7). На рис. 19.9 показано количество ходов и вознаграждений агента.

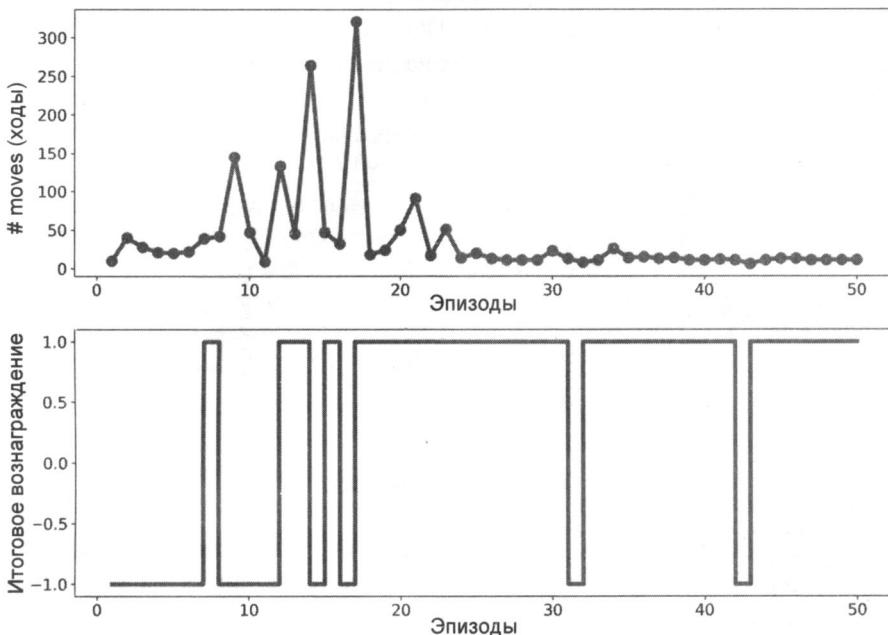


Рис. 19.9. Количество ходов агента и его вознаграждения

Приведенный здесь график обучения демонстрирует, что агент после 30 эпизодов обучения находит короткий путь, чтобы добраться до клада. Поэтому длительности эпизодов после 30-го эпизода более-менее одинаковы, с небольшими отклонениями из-за ϵ -жадной стратегии.

19.5. Обзор глубокого Q-обучения

В коде предыдущего раздела была приведена реализация популярного алгоритма Q-обучения для примера клетчатого мира. Этот пример представлял собой дискретное пространство состояний размером 30, для которого было достаточно хранить Q-значения в словаре Python.

Однако следует отметить, что иногда количество состояний может быть очень большим, возможно, почти бесконечно большим. Кроме того, пространство состояний может оказаться непрерывным, а не дискретным. Некоторые состояния могут вообще не посещаться во время обучения, что способно вызвать проблемы обобщения при последующей работе агента с такими неизведанными состояниями.

Чтобы решить эти проблемы, вместо представления функции ценности в табличном формате, таком как $V(S_t)$ или $Q(S_t, A_t)$, к функции ценности действия применяют метод

аппроксимации. Суть его в том, что мы определяем параметрическую функцию $v_w(x_s)$, которая может научиться аппроксимировать истинную функцию ценности, т. е. $v_w(x_s) \approx v_\pi(s)$, где x_s — набор входных признаков (или «характерных» состояний).

Когда аппроксимирующая функция $q_w(x_s, a)$ представляет собой глубокую нейронную сеть (Deep Neural Network, DNN), полученная модель называется глубокой *Q*-сетью (Deep Q-Network, DQN). Для обучения модели DQN веса обновляются в соответствии с алгоритмом *Q*-обучения. Пример модели DQN приведен на рис. 19.10, где состояния представлены в виде признаков, переданных на первый уровень.

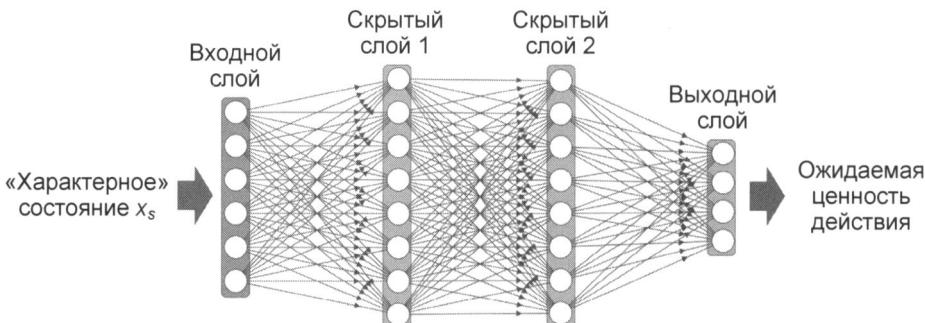


Рис. 19.10. Пример DQN

Теперь давайте посмотрим, как мы можем обучить DQN, используя алгоритм глубокого *Q*-обучения (deep Q-learning). В целом этот подход очень похож на табличный метод Q-learning. Основное же отличие состоит в том, что теперь у нас есть многослойная нейросеть, которая вычисляет ценности действий.

19.5.1. Применение алгоритма *Q*-обучения в моделях DQN

В этом разделе мы опишем процедуру обучения модели DQN с использованием алгоритма *Q*-обучения. Применение глубокого *Q*-обучения требует от нас внесения некоторых изменений в уже готовую реализацию стандартного подхода к *Q*-обучению.

Одно из таких изменений необходимо внести в метод агента `choose_action()`, который в коде для *Q*-обучения, приведенном в предыдущем разделе, просто обращался к ценостям действий, хранящимся в словаре. Теперь эту функцию следует изменить, чтобы выполнить прямой проход нейросетевой модели для вычисления ценностей действий.

Другие изменения, необходимые для алгоритма глубокого *Q*-обучения, представлены далее.

Глобальная память

Используя табличный метод для *Q*-обучения в примере кода из предыдущего раздела, мы могли обновлять ценности для определенных пар состояния-действие, не затрагивая ценности других пар. Однако теперь, когда мы аппроксимируем $q(s, a)$ нейросетевой моделью, обновление весов для пары состояния-действие, вероятно, повлияет и на вы-

ходные данные других состояний. При обучении нейронных сетей с применением стохастического градиентного спуска для задачи обучения с учителем (например, задачи классификации) мы используем несколько эпох для проходов по обучающим данным, пока модель не сойдется.

Однако это невозможно в Q-обучении, т. к. эпизоды в процессе обучения будут меняться и, как следствие, некоторые состояния, посещенные на ранних этапах обучения, станут реже посещаться позже.

Кроме того, другая проблема заключается в том, что при обучении нейронной сети мы предполагаем, что обучающие примеры являются *независимыми и одинаково распределенными* (Independent and Identically Distributed, IID). Однако примеры, взятые из эпизода агента, не являются IID, т. к. они образуют последовательность переходов.

Для решения этих проблем, когда агент взаимодействует со средой и генерирует пятерку переходов $q_w(x_s, a)$, мы сохраняем большое (но конечное) число таких переходов в буфере памяти, часто называемом *глобальной памятью*, или *памятью воспроизведения* (replay memory). После каждого нового взаимодействия (т. е. когда агент выбирает действие и выполняет его в среде) результирующая новая пятерка переходов добавляется в память.

Чтобы не раздувать объем сохраняемых данных до бесконечности, самый старый переход удаляют из памяти (например, если это список Python, мы можем использовать метод `pop(0)` для удаления первого элемента списка). Затем из буфера памяти случайным образом выбирают мини-пакет примеров, которые будут использованы для вычисления потерь и обновления параметров сети. Этот процесс схематически показан на рис. 19.11.

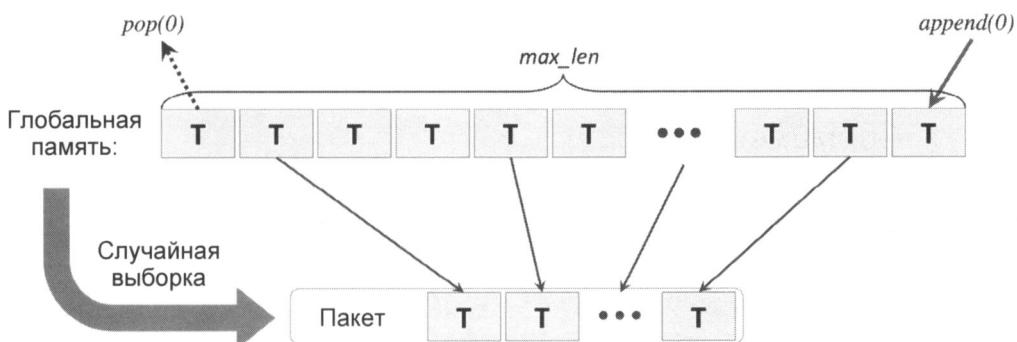


Рис. 19.11. Использование глобальной памяти



Реализация глобальной памяти

Глобальная память может быть реализована с помощью списка Python, однако при каждом добавлении нового элемента в список нам пришлось бы проверять размер списка и при необходимости вызывать метод `pop(0)`.

В качестве альтернативы мы можем использовать структуру данных `deque` из библиотеки Python `collections`, что позволяет нам указать необязательный аргумент `max_len`. Передав аргумент `max_len`, мы получим ограниченную очередь. Поэтому, когда объект заполнен, добавление нового элемента приводит к автоматическому удалению из него самого старого элемента.

Надо сказать, что это более эффективно, чем использование списка Python, поскольку удаление первого элемента списка с помощью `pop(0)` имеет вычислительную сложность $O(n)$, в то время как сложность выполнения `dequeue` составляет $O(1)$. Вы можете узнать больше о реализации `dequeue` из официальной документации, доступной по адресу: <https://docs.python.org/3.9/library/collections.html#collections.deque>.

Определение целей для расчета потерь

Еще одно необходимое изменение по сравнению с табличным методом Q-обучения заключается в способе адаптации правила обновления для обучения параметров модели DQN. Напомним, что пятерка переходов T , хранящаяся в пакете примеров, содержит $(x_s, a, r, x_{s'}, \text{done})$, где `done` — признак завершения.

Как показано на рис. 19.12, мы выполняем два прямых прохода модели DQN. Первый прямой проход использует признаки текущего состояния (x_s). Затем второй прямой проход использует особенности следующего состояния ($x_{s'}$). В результате мы получим расчетные ценности действий $q_w(x_s, :)$ и $q_w(x_{s'}, :)$ из первого и второго прямого проходов соответственно. (Здесь запись $q_w(x_s, :)$ означает вектор Q-ценностей для всех действий в \hat{A} .) Из пятерки переходов мы знаем, что действие a выбирается агентом.

Следовательно, в соответствии с алгоритмом Q-обучения нам необходимо обновить ценность действия, соответствующую паре состояние-действие (x_s, a) , скалярным целевым значением $r + \gamma \max_{a' \in \hat{A}} q_w(x_{s'}, a')$. Вместо формирования скалярного целевого значения мы создадим вектор целевой ценности действия, который сохраняет ценности для других действий $a' \neq a$.

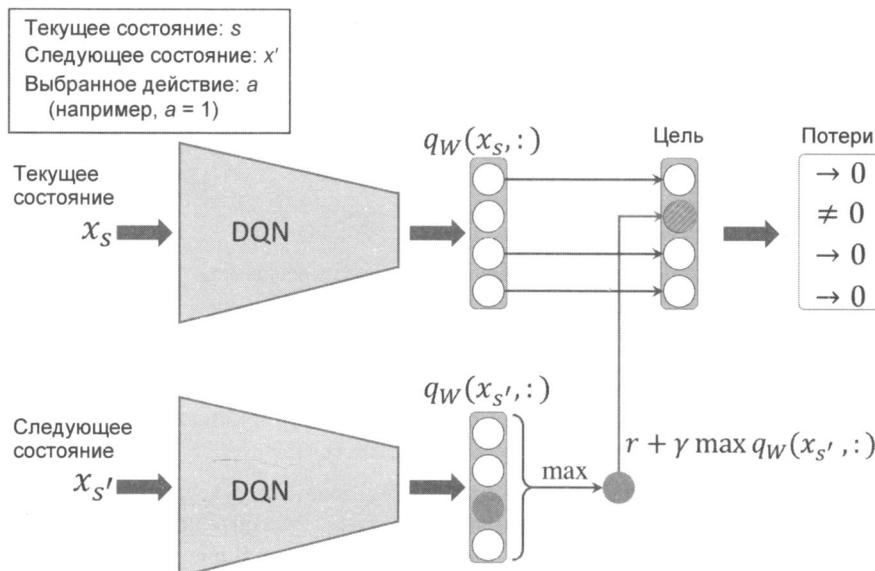


Рис. 19.12. Определение целевой ценности с помощью DQN

Этот процесс можно рассматривать как решение задачи регрессии, в которой задействованы следующие три переменные:

- ◆ текущие прогнозируемые ценности $q_w(x_s, :)$;
- ◆ вектор целевой ценности, как описано ранее;
- ◆ функция потерь на основе стандартной среднеквадратичной ошибки (MSE).

В результате потери будут нулевыми для всех действий, кроме a . Вычисленные потери распространяются обратно для обновления параметров сети.

19.5.2. Реализация алгоритма глубокого Q-обучения

Наконец-то мы можем применить на практике все упомянутые методы для реализации алгоритма глубокого Q-обучения. На этот раз мы используем среду CartPole из библиотеки OpenAI Gym, которая уже была представлена ранее. Напомним, что среда CartPole имеет непрерывное пространство состояний размером 4. В следующем коде мы определяем класс DQNAgent, который строит модель и задает различные гиперпараметры.

Этот класс включает два дополнительных метода по сравнению с предыдущим агентом, основанным на табличном Q-обучении: метод `remember()` добавит новую пятерку переходов в буфер памяти, а метод `replay()` создаст мини-пакет примеров переходов и передаст их методу `_learn()` для обновления весовых параметров сети:

```
import gym
import numpy as np
import torch
import torch.nn as nn
import random
import matplotlib.pyplot as plt
from collections import namedtuple
from collections import deque
np.random.seed(1)
torch.manual_seed(1)

Transition = namedtuple(
    'Transition', ('state', 'action', 'reward',
                  'next_state', 'done'))

class DQNAgent:
    def __init__(self, env, discount_factor=0.95,
                 epsilon_greedy=1.0, epsilon_min=0.01,
                 epsilon_decay=0.995, learning_rate=1e-3,
                 max_memory_size=2000):
        self.env = env
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n
        self.memory = deque(maxlen=max_memory_size)
        self.gamma = discount_factor
        self.epsilon = epsilon_greedy
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
```

```
self.lr = learning_rate
self._build_nn_model()

def _build_nn_model(self):
    self.model = nn.Sequential(nn.Linear(self.state_size, 256),
                               nn.ReLU(),
                               nn.Linear(256, 128),
                               nn.ReLU(),
                               nn.Linear(128, 64),
                               nn.ReLU(),
                               nn.Linear(64, self.action_size))
    self.loss_fn = nn.MSELoss()
    self.optimizer = torch.optim.Adam(
        self.model.parameters(), self.lr)

def remember(self, transition):
    self.memory.append(transition)

def choose_action(self, state):
    if np.random.rand() <= self.epsilon:
        return np.random.choice(self.action_size)
    with torch.no_grad():
        q_values = self.model(torch.tensor(state,
                                             dtype=torch.float32))[0]
    return torch.argmax(q_values).item() # возвращает действие

def _learn(self, batch_samples):
    batch_states, batch_targets = [], []
    for transition in batch_samples:
        s, a, r, next_s, done = transition
        with torch.no_grad():
            if done:
                target = r
            else:
                pred = self.model(torch.tensor(next_s,
                                                dtype=torch.float32))[0]
                target = r + self.gamma * pred.max()
        target_all = self.model(torch.tensor(s,
                                              dtype=torch.float32))[0]
        target_all[a] = target
        batch_states.append(s.flatten())
        batch_targets.append(target_all)
    self._adjust_epsilon()
    self.optimizer.zero_grad()
    pred = self.model(torch.tensor(batch_states,
                                   dtype=torch.float32))
    loss = self.loss_fn(pred, torch.stack(batch_targets))
    loss.backward()
    self.optimizer.step()
    return loss.item()
```

```
def _adjust_epsilon(self):
    if self.epsilon > self.epsilon_min:
        self.epsilon *= self.epsilon_decay

def replay(self, batch_size):
    samples = random.sample(self.memory, batch_size)
    return self._learn(samples)
```

Затем с помощью следующего кода мы обучаем модель в течение 200 эпизодов и в конце визуализируем историю обучения с помощью функции plot_learning_history():

```
def plot_learning_history(history):
    fig = plt.figure(1, figsize=(14, 5))
    ax = fig.add_subplot(1, 1, 1)
    episodes = np.arange(len(history))+1
    plt.plot(episodes, history, lw=4,
             marker='o', markersize=10)
    ax.tick_params(axis='both', which='major', labelsize=15)
    plt.xlabel('Episodes', size=20)
    plt.ylabel('Total rewards', size=20)
    plt.show()

## Общие настройки
EPISODES = 200
batch_size = 32
init_replay_memory_size = 500

if __name__ == '__main__':
    env = gym.make('CartPole-v1')
    agent = DQNAgent(env)
    state = env.reset()
    state = np.reshape(state, [1, agent.state_size])
    ## Заполнение глобальной памяти
    for i in range(init_replay_memory_size):
        action = agent.choose_action(state)
        next_state, reward, done, _ = env.step(action)
        next_state = np.reshape(next_state, [1, agent.state_size])
        agent.remember(Transition(state, action, reward,
                                  next_state, done))
    if done:
        state = env.reset()
        state = np.reshape(state, [1, agent.state_size])
    else:
        state = next_state
total_rewards, losses = [], []
for e in range(EPISODES):
    state = env.reset()
    if e % 10 == 0:
        env.render()
    state = np.reshape(state, [1, agent.state_size])
```

```

for i in range(500):
    action = agent.choose_action(state)
    next_state, reward, done, _ = env.step(action)
    next_state = np.reshape(next_state,
                           [1, agent.state_size])
    agent.remember(Transition(state, action, reward,
                               next_state, done))
    state = next_state
    if e % 10 == 0:
        env.render()
    if done:
        total_rewards.append(i)
        print(f'Эпизод: {e}/{EPISODES}, Вознаграждение: {i}')
        break
    loss = agent.replay(batch_size)
    losses.append(loss)
plot_learning_history(total_rewards)

```

Обучив агента на протяжении 200 эпизодов, мы видим, что он действительно научился увеличивать общее вознаграждение с течением времени, как показано на рис. 19.13.

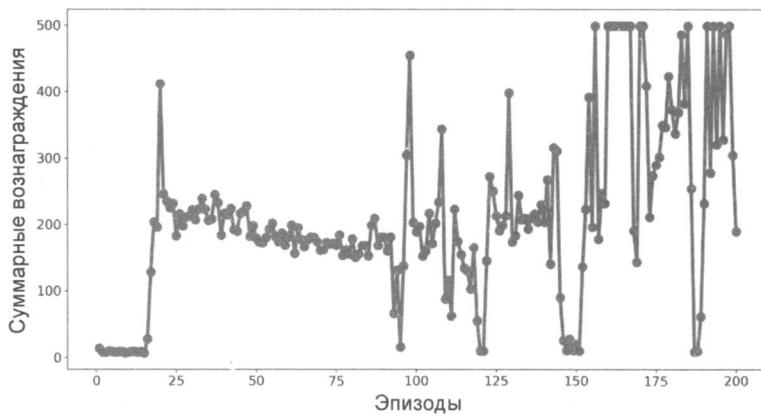


Рис. 19.13. Увеличение вознаграждения агента с течением времени

Обратите внимание, что общее количество вознаграждений, полученных в эпизоде, равно количеству времени, в течение которого агент может сбалансировать обратный маятник. История обучения, представленная на рис. 19.13, показывает, что примерно после 30 эпизодов агент приобретает навык балансировать маятник и удерживать его дольше 200 временных шагов.

19.6. Краткое содержание главы и книги

В этой главе мы рассмотрели основные принципы RL, начиная с самых основ, и принципы поддержки принятия решений с помощью RL в сложных условиях.

Вы узнали о взаимодействии агента со средой и марковских процессах принятия решений (MDP), а также познакомились с тремя основными подходами к решению задач

RL: динамическим программированием, MC-обучением и TD-обучением. Мы отметили тот факт, что алгоритм динамического программирования предполагает наличие полных знаний о динамике окружающей среды, что обычно неверно для большинства реальных задач.

Затем вы увидели, как алгоритмы на основе MC и TD обучаются моделью путем взаимодействия агента с окружающей средой и генерации симулированного опыта. Изучив теоретические основы, мы реализовали алгоритм Q-обучения в качестве подкатегории off-policy алгоритма TD для решения задачи клетчатого мира. Наконец, мы рассмотрели концепцию аппроксимации функций и, в частности, глубокого Q-обучения, которые можно использовать для задач с большими или непрерывными пространствами состояний.

Мы надеемся, что вам понравились эта глава, повествующая о машинном обучении Python, и наш обзор машинного и глубокого обучения. На протяжении всей книги мы старались поднять основные темы, которые наиболее актуальны для указанной области, и теперь вы наверняка готовы применить эти методы на практике для решения реальных задач.

Мы начали наше увлекательное путешествие в *главе 1* с краткого обзора различных типов машинного обучения: с учителем, с подкреплением и без учителя.

Затем в *главе 2* мы обсудили несколько различных алгоритмов обучения, которые можно использовать для классификации, начиная с простых однослойных нейронных сетей.

Мы продолжили обсуждение продвинутых алгоритмов классификации в *главе 3* и рассказали о наиболее важных аспектах конвейера машинного обучения в *главе 4*.

Помните, что даже самый продвинутый алгоритм ограничен информацией, содержащейся в обучающих данных, на которых он учится. Поэтому в *главе 6* мы рассказали о передовых методах построения и оценки прогностических моделей, что является еще одним важным аспектом приложений машинного обучения.

Если одиночный алгоритм обучения не достигает желаемой производительности, иногда может быть полезно создать ансамбль алгоритмов (моделей), чтобы получить совместный прогноз. Мы рассмотрели этот прием в *главе 7*.

Затем в *главе 8* мы применили машинное обучение для анализа одной из самых популярных и интересных форм данных нашего времени, которая наиболее широко представлена на платформах социальных сетей в Интернете, — текстовых документов.

По большей части наше внимание было сосредоточено на алгоритмах классификации, которые, вероятно, являются самым популярным применением машинного обучения. Однако на этом наше путешествие не закончилось! В *главе 9* мы рассмотрели несколько алгоритмов регрессионного анализа для прогнозирования значений непрерывных целевых переменных.

Еще одна захватывающая область машинного обучения — кластерный анализ, который способен найти скрытые закономерности в данных, даже если доступные обучающие данные не дают правильных ответов, на которых можно учиться. Мы рассмотрели эту тему в *главе 10*.

Затем мы переключили внимание на один из самых захватывающих алгоритмов во всей области машинного обучения — искусственные нейронные сети. Мы начали с реализации многослойного персептрона с нуля с помощью NumPy в *главе 11*.

Преимущества PyTorch при реализации моделей глубокого обучения стали очевидны в главе 12, где мы использовали PyTorch для облегчения процесса построения нейросетевых моделей, работали с объектами `Dataset` PyTorch и показали, как применять методы предварительной обработки к набору данных.

Мы углубленно рассмотрели механизм PyTorch в главе 13 и обсудили различные аспекты и методы PyTorch, включая тензорные объекты, вычисление градиентов, а также модуль нейронной сети `torch.nn`.

В главе 14 мы углубились в тематику сверточных нейронных сетей, которые в настоящее время широко используются в компьютерном зрении из-за их высокой производительности в задачах классификации изображений.

В главе 15 рассказано о моделировании последовательностей с использованием RNN.

В главе 16 мы представили механизм внимания для устранения одной из слабых сторон RNN, а именно — невозможности запоминания предыдущих входных элементов при работе с длинными последовательностями. Затем мы исследовали различные виды трансформеров, которые представляют собой варианты архитектуры глубокого обучения, основанные на механизме самовнимания, и относятся к последним достижениям в области больших языковых моделей.

В главе 17 вы научились генерировать новые изображения с помощью GAN, а также узнали об автокодировщиках, пакетной нормализации, транспонированной свертке и GAN Вассерштейна.

Перечисленные главы были посвящены данным, представленным в виде таблиц, текста или изображений. А в главе 18 мы сосредоточились на глубоком обучении для графовых данных, которые обычно применяются для представления структуры социальных сетей и молекул (химических соединений). Кроме того, вы узнали о так называемых графовых нейронных сетях, которые предназначены специально для работы с подобными данными.

Наконец, в этой, последней главе книги мы рассмотрели отдельную категорию задач машинного обучения и показали, как разрабатывать алгоритмы, которые обучаются, взаимодействуя с окружающей средой и получая вознаграждение за удачные действия.

Хотя всестороннее описание глубокого обучения выходит далеко за рамки нашей книги, мы надеемся, что пробудили у вас достаточный интерес, чтобы вы начали следить за самыми последними достижениями в этой области глубокого обучения.

Если вы подумываете о карьере в области машинного обучения или просто хотите быть в курсе последних достижений в этой области, мы рекомендуем вам следить за новыми статьями, опубликованными в этой области. Далее приведены некоторые ресурсы, которые мы считаем особенно полезными:

- ◆ **сабреддит и сообщество, посвященное машинному обучению:**
<https://www.reddit.com/r/learnmachinelearning/>;
- ◆ **ежедневно обновляемый список последних рукописей на тему машинного обучения, загружаемых на сервер препринтов arXiv:** <https://arxiv.org/list/cs.LG/recent>;
- ◆ **механизм рекомендации статей, созданный на основе arXiv:**
<http://www.arxiv-sanity.com>.

Наконец, вы можете узнать больше о текущей деятельности авторов книги на следующих сайтах:

- ◆ Себастьян Рашка: <https://sebastianraschka.com>;
- ◆ Юси (Хейден) Лю: <https://www.mlexample.com/>;
- ◆ Вахид Мирджалили: <http://vahidmirjalili.com>.

Вы всегда можете связаться с нами, если у вас появятся какие-либо вопросы об этой книге или если вам понадобятся общие советы по машинному обучению.

Предметный указатель

Q

- Q-обучение 653
 - ◊ глубокое 653

* * *

A

- Автокодировщик
 - ◊ вариационный 570
 - ◊ недополный 568
 - ◊ сверхполный 568
 - ◊ шумоподавляющий 569
- Автоматическое дифференцирование 357, 405
- Авторегрессия 514
- Агент 30, 642
- Алгоритм
 - ◊ DBSCAN 325
 - ◊ k-ближайших соседей 115
 - ◊ off-policy 659
 - ◊ on-policy 659
 - ◊ глубокого Q-обучения 674
 - ◊ нечетких С-средних 310
 - ◊ обратного распространения 345
 - ◊ одиночной связи 318
 - ◊ ожидания-максимизации 268
 - ◊ полной связи 318
 - ◊ Портера 259
 - ◊ последовательное пошаговое исключение 142
 - ◊ случайного леса 112
 - ◊ совершенствования стратегии 655

Анализ

- ◊ данных
 - исследовательский 277
- ◊ мнений 250
- ◊ регрессионный 272
- ◊ эмоциональной окраски текста 250
- Атрибут 35
- Аутлаер 286

Б

- Благоприятность состояния 650
- Блок управляемый рекуррентный 484, 494
- Бустинг 234
 - ◊ адаптивный 234
- Бутстрэп-выборка 112
- Бэггинг 228

В

- Вектор
 - ◊ признаков
 - разреженный 253
 - ◊ скрытый 567
- Вентиль См. Гейт
- Вероятность перехода состояния 647
- Взвешенное скалярное внимание 527
- Вложенная перекрестная проверка 199
- Внешнее обучение 263
- Внимание
 - ◊ маскированное 535
 - ◊ разреженное 544
- Внутренний сдвиг переменных 591

Вознаграждение 651

- ◊ **накопленное** См. Отдача
- ◊ **непосредственное** 648
- ◊ **последующее** 648
- Воспроизведение опыта** 608
- Выбор модели** 183
- Выборка** 35

Г

Гауссово ядро 101

Гейт 492

- ◊ **входа** 493
- ◊ **выхода** 493
- ◊ **утраты (забывания)** 493

Гиперболический тангенс 393

Гиперпараметры 38, 183

Глобальная память 675

Глубокая Q-сеть 674

Голосование

- ◊ **мажоритарное** 212
- ◊ **мода** 213
- ◊ **относительным большинством** 212

Градиент

- ◊ **взрывающийся** 491
- ◊ **исчезающий** 491

Градиентный спуск 59

- ◊ **мини-пакетный** 68

- ◊ **пакетный** 60

- ◊ **стохастический** 67

Граф 611

- ◊ **вычислений** 401

- ◊ **лапласиан** 634

- ◊ **направленный ациклический** 401

- ◊ **неориентированный** 611

- ◊ **ориентированный** 612

- ◊ **помеченный** 613

- ◊ **спектр** 634

- ◊ **узел (вершина)** 611

Графический процессор 365

Д

Дальнее взаимодействие 491

Данные

- ◊ **временных рядов** 481
- ◊ **входные** 35
- ◊ **выходные** 35
- ◊ **последовательные** 480

Дендрограмма 318

Дерево решений 104

- ◊ **бинарное** 106

Динамика среды 646

Динамическое программирование 645

Дискретная свертка 439

- ◊ **шаг** 442

- ◊ **ядро** 440

Дисперсия 92

Доля объясненной дисперсии 156

Дополнение данных 466

Древовидный оценщик Парзена 198

З

Задача

- ◊ **Grid World** 660

- ◊ **исключающего ИЛИ** 400

- ◊ **классификации** 28

- ◊ **маскированного языкового моделирования** 548

- ◊ **непрерывная** 648

- ◊ **эпизодическая** 648

Запись 35

Заполнение 440

И

Инерция кластера 306

Инлаер 286

Искусственный интеллект 25, 43

Использование 644

Исследование 644

Исследовательский старт 657

Исчезающий градиент 335

К

Классификация 28

- ◊ **без учителя** 32

- ◊ **бинарная** 28

- ◊ **многоклассовая** 29

Кластер 32

- ◊ **выбросы** 315

- ◊ **компактность (связность)** 313

- ◊ **отделимость** 313

Кластеризация 32

- ◊ **жесткая** 310

- ◊ **иерархическая** 305, 318

- ◊ **агломеративная** 318

- ◊ **разделительная** 318

Кластеризация (*прод.*)

- ◊ мягкая 310
- ◊ на основе плотности 305
- ◊ на основе прототипов 305

Ковариата 35

Кодирование

- ◊ двоичное 130
- ◊ позиционное 128
- ◊ пороговое 131
- ◊ унитарное 128
- ◊ частотное 130

Коэффициент

- ◊ дисконтирования 649
- ◊ нечеткости 311
- ◊ смешанной корреляции 292

Коэффициент корреляции Мэтьюза 201

Коэффициент корреляции Пирсона 278

Кривая

- ◊ валидации 188
- ◊ обнаружения характеристическая 205
- ◊ обучения 188
- ◊ точности-полноты 206

Л

Лексема маски 548

Лемма 260

Лемматизация 260

Линейная разделимость 47

Линейный дискриминант Фишера 165

Линейный дискриминантный анализ 164

Линия регрессии 273

Логический вывод 417

Локальное рецептивное поле 438

М

Макроусреднение 208

Марковский процесс принятия решений 645

Матрица

- ◊ диаграмм рассеяния 277
- ◊ несоответствий 201
- ◊ плана 321
- ◊ связей 320
- ◊ смежности 611
- ◊ степенная 634

Машинное обнуление 83

Медианное абсолютное отклонение 287

Медоид 305

Метка 35

Метка класса

- ◊ истинная 47
- ◊ прогнозируемая 47

Метод

- ◊ k-средних 304
- ◊ ансамблевый 212
- ◊ аппроксимации функции 673
- ◊ временных различий 646
- ◊ главных компонент 152
- ◊ локтя 312
- ◊ Монте-Карло 646
- ◊ обобщенной итерации стратегии 655
- ◊ один против всех 335
- ◊ опорных векторов 95
 - ◊ ядерный 99
- ◊ откладывания 183
- ◊ стохастического градиентного спуска 334
- ◊ ядерный 101

Метод LASSO 293

Метод наименьших квадратов

- ◊ обычный 280

Микроусреднение 208

Многослойный персепtron 334

Моделирование тем 266

Модель

- ◊ авторегрессионная 570
- ◊ емкость 410
- ◊ мешка слов 253
- ◊ нормализующая потоковая 570

Н

Наблюдение 35, 643

Насыщение 572

Насыщение модели 293

Недобучение 92

Независимая переменная См. Предиктор

Нейронная сеть 331

- ◊ GAN Вассерштейна 588
- ◊ автокодировщик 567
- ◊ генеративно-состязательная 566
- ◊ генератор 570
- ◊ глубокая 39, 331, 674
- ◊ графовая 610
- ◊ декодер 567
- ◊ дискриминатор 571
- ◊ кодировщик 567
- ◊ рекуррентная 480
- ◊ сверточная 437

Необработанная частота термина 254

Нормализация 134

◊ пакетная 590

О

Области решений 77

Обработка естественного языка 250

Обратный маятник 661

Обучающая подвыборка 184

Обучающий пример 35

Обучение 35

◊ без учителя 26, 304

◊ глубокое 39, 331

◊ машинное 25

◊ на основе многообразий 174

◊ с подкреплением 26, 641

◊ с учителем 26

◊ через взаимодействие 642

Опорный вектор 95

Отдача 648

◊ целевая 658

отклик 29

Отображение К-липшицево 602

Отсроченная обратная связь 31

Отступ 95

Оцениватель 124

Ошибка

◊ классификации 106

◊ прогнозирования 203

◊ систематическая 92

◊ среднеквадратичная 59

Ошибка обобщения 38

П

Память воспроизведения См. Глобальная память

Перекрестная проверка 38

◊ с исключением по одному 186

Переменная 35

◊ зависимая 35

◊ отклика 35

Переменная невязки 96

Переменная отклика 273

Подстановка среднего 123

Поиск

◊ исчерпывающий 195

◊ по сетке 193

◊ рандомизированный 195

Полнота 203

Последовательное обучение 67

Правдоподобие 83

Правильность 203

Предиктор 29, 35, 273

Признак 29, 35

◊ нагрузка 163

Признаки

◊ иерархия 438

◊ извлечение 142, 151

◊ карта 438

◊ масштабирование 134

◊ номинальные 125

◊ отбор 142

◊ порядковые (ординальные) 125

◊ пошаговый отбор 151

Примесь 105

◊ Джини 106

Прирост информации 104, 299

Проклятие размерности 118

Прореживание 453

◊ обратное 455

Прямое распространение 336

Псевдоостаток 243

Псевдоотклик 243

Р

Регрессионный анализ 29

Регрессия 28

◊ гребневая 293

◊ линейная

◊ многофакторная 274

◊ простая 272

◊ логистическая 79

◊ мультиномиальная 80

Регуляризация 93

◊ L2 136

◊ параметр 93

Регуляризация L1 137

Рекуррентное ребро 485

С

Самовнимание 523

◊ многоголовое 532

Самообучение 539

Свертка

◊ обращенная 589

◊ с дробным шагом См. Свертка транспонированная

◊ транспонированная 588

Сигнал вознаграждения 30, 643

Силуэтный

- ◊ анализ 313
- ◊ коэффициент 313

Симулированный опыт 656**Скрытое распределение Дирихле** 266**Сложные условия** 643**Слой объединяющий** 439

- ◊ подвыборки (объединения) 448

Смещение 45**Собственные**

- ◊ векторы 153

- ◊ декомпозиция 153

- ◊ значения 153

Совещательное планирование 30**Состязательный пример** 407**Среда** 643**Среднее гармоническое** 204**Стандартизация** 65, 134**Стекинг** 228**Стекирование** 372**Стемминг** 259**Стратегия** 650

- ◊ ϵ -жадная 657

- ◊ оценка 654

Схлопывание моды распределения 606**Т****Тележка** 661**Теорема двойственности**

Канторовича — Рубинштейна 601

Точка

- ◊ граничная 325

- ◊ окруженная 325

- ◊ шумовая 325

Точность 203**Траектория** 648**Трансформер** 518**У****Уменьшение размерности** 32**Унigramма** 254**Уравнение Беллмана** 652**Условный вывод** 570**Утечка тестовых данных** 183**Ф****Фактический вход** 44**Функция**

- ◊ softmax 395

- ◊ вознаграждения 642

- ◊ единичная ступенчатая 44

- ◊ логарифмического правдоподобия 83

- ◊ логит 80

- ◊ потерь 35

- ◊ радиальная базисная 101

- ◊ сигмоидная 80, 393

- ◊ спрямленная линейная активация 398

- ◊ стоимости 35

- ◊ целевая 58

- ◊ ценности 572

- действия 651

- состояния 650

- ◊ ядерная (керн-функция) 101

Ц**Целевая переменная** См. Признак**Цель** 35**Центроид** 305**Ш****Шанс** 80**Э****Экземпляр** 35**Эластичная сеть** 294**Энтропия** 106**Эпизод** 642**Эталон** 35**Я****Ядерный трюк** 101

МАШИННОЕ ОБУЧЕНИЕ С PYTORCH И SCIKIT-LEARN

Перед вами не только исчерпывающее руководство по машинному и глубокому обучению с использованием Python, фреймворка PyTorch и библиотеки scikit-learn, но и справочник, к которому вы будете постоянно возвращаться при создании систем машинного обучения. Книга подробно описывает все основные методы машинного обучения и содержит четкие пояснения, визуализации и примеры. Авторы стремятся научить читателя принципам самостоятельного создания моделей и приложений, а не просто следовать жестким инструкциям.

Описаны новые дополнения к библиотеке scikit-learn. Рассмотрены различные методы машинного и глубокого обучения для классификации текста и изображений. Рассказано о генеративно-состязательных сетях (GAN) для синтеза новых данных и обучения интеллектуальных агентов. Освещены последние тенденции в области глубокого обучения, включая введение в графовые нейронные сети и крупномасштабные преобразователи, используемые для обработки естественного языка (NLP). Книга будет полезна как начинающим разработчикам на Python, слабо знакомым с машинным обучением, так и опытным, желающим углубить свои знания.

«Я уверен, что эта книга станет для вас бесценным источником теоретических знаний в области машинного обучения и сокровищницей практических идей. Я надеюсь, что она вдохновит вас на новые потрясающие достижения, независимо от того, какие задачи вы решаете».

— Дмитро Джулгаков,
ведущий специалист PyTorch Core

Вы изучите:

- фреймворки, модели и методы машинного обучения, применимые к широкому кругу задач и наборов данных;
- библиотеку scikit-learn для машинного обучения и фреймворк PyTorch для глубокого обучения;
- приемы обучения классификаторов на изображениях, тексте и т. д.;
- средства создания и обучения нейронных сетей, преобразователей и графических нейронных сетей;
- передовые методы оценки и настройки моделей.

Вы сможете глубже понять:

- прогнозирование непрерывных целевых результатов с помощью регрессионного анализа;
- особенности текстовых данных и данных из социальных сетей с помощью тонального анализа.



Архив с цветными иллюстрациями и кодами всех примеров можно скачать по ссылке
<https://tinyurl.com/3f5nfu52>.

ISBN 978-601-11-0034-2



Packt

FOLIANT