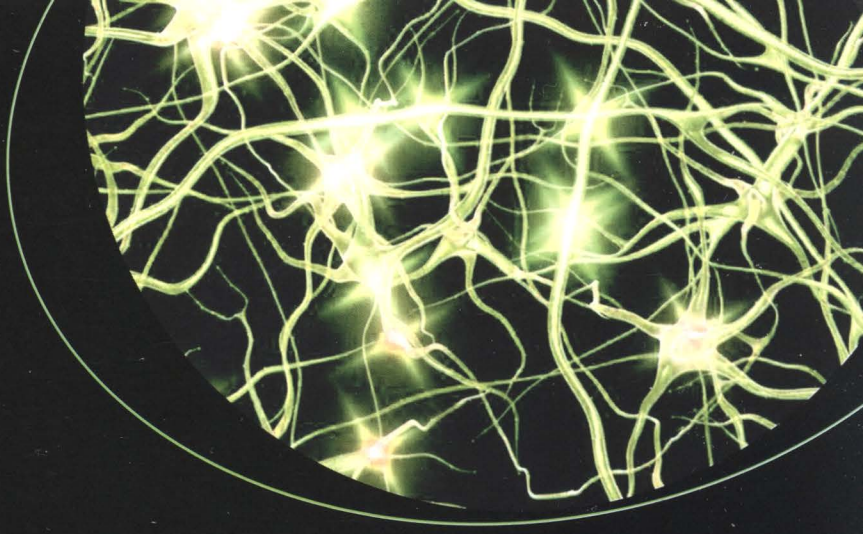


Умберто Микелуччи

Прикладное глубокое обучение

Подход к пониманию глубоких нейронных сетей на основе метода кейсов

bhv®

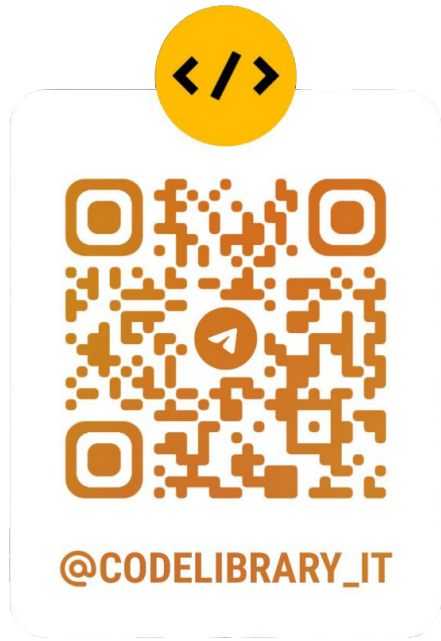
Apress®

Applied Deep Learning

A Case-Based Approach to Understanding Deep Neural Networks

Umberto Michelucci

Apress®


@CODELIBRARY_IT

Прикладное глубокое обучение

Подход к пониманию глубоких нейронных сетей на основе метода кейсов

Умберто Микелуччи

Санкт-Петербург
«БХВ-Петербург»
2020

УДК 004.85
ББК 32.973.26-018
М59

Микелуччи У.

М59 Прикладное глубокое обучение. Подход к пониманию глубоких нейронных сетей на основе метода кейсов: Пер. с англ. — СПб.: БХВ-Петербург, 2020. — 368 с.: ил.

ISBN 978-5-9775-4118-3

Затронуты расширенные темы глубокого обучения: оптимизационные алгоритмы, настройка гиперпараметров, отсев и анализ ошибок, стратегии решения типичных задач во время тренировки глубоких нейронных сетей. Описаны простые активационные функции с единственным нейроном (ReLu, сигмоида и Swish), линейная и логистическая регрессии, библиотека TensorFlow, выбор стоимостной функции, а также более сложные нейросетевые архитектуры с многочисленными слоями и нейронами. Показана отладка и оптимизация расширенных методов отсева и регуляризации, настройка проектов машинного обучения, ориентированных на глубокое обучение с использованием сложных наборов данных. Приведены результаты анализа ошибок нейронной сети с примерами решения проблем, возникающих из-за дисперсии, смещения, переподгонки или разрозненных наборов данных. По каждому техническому решению даны примеры решения практических задач.

Для разработчиков систем глубокого обучения

УДК 004.85
ББК 32.973.26-018

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Сависте*
Перевод с английского	*Андрея Логунова*
Компьютерная верстка	*Ольги Сергиенко*
Оформление обложки	*Карины Соловьевой*

Подписано в печать 30.09.19.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 29,67.
Тираж 1200 экз. Заказ № 10023.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано с готового оригинал-макета
ООО "Принт-М", 142300, М.О., г. Чехов, ул. Полиграфистов, д. 1

ISBN 978-1-4842-3789-2 (англ.)
ISBN 978-5-9775-4118-3 (рус.)

Оглавление

Об авторе **11**

О рецензенте **13**

Признательности **15**

Комментарии переводчика **17**

Введение **19**

Глава 1. Вычислительные графы и TensorFlow **25**
Настройка среды программирования на языке Python 25
Создание среды 27
Установка библиотеки TensorFlow 31
Блокноты Jupyter 33
Элементарное введение в TensorFlow 35
Вычислительные графы 35
Тензоры 38
Создание и выполнение вычислительного графа 39
Вычислительный граф с типом *tf.constant* 40
Вычислительный граф с типом *tf.Variable* 40
Вычислительный граф с типом *tf.placeholder* 42
Различия между *run* и *eval* 44
Зависимости между узлами 45
Советы по созданию и закрытию сеанса 46

Глава 2. Один-единственный нейрон **49**
Структура нейрона 49
Матричное обозначение 52
Совет по реализации на языке Python: циклы и NumPy 53
Активационные функции 55
Активационная функция тождественного отображения 55
Активационная функция сигмоидальная 56
Активационная функция *tanh* (гиперболический тангенс) 58
Активационная функция ReLU (выпрямленного линейного элемента) 58

Активационная функция ReLU с утечкой ... 60
Активационная функция Swish ... 61
Другие активационные функции ... 62
Стоимостная функция и градиентный спуск: причуды темпа заучивания ... 63
Темп заучивания на практическом примере ... 65
Пример линейной регрессии в TensorFlow ... 70
Набор данных для линейной регрессионной модели ... 72
Нейрон и стоимостная функция для линейной регрессии ... 75
Разумно достаточный и оптимизационный метрический показатель ... 79
Пример логистической регрессии ... 81
Стоимостная функция ... 81
Активационная функция ... 82
Набор данных ... 82
Реализация в TensorFlow ... 86
Ссылки на литературу ... 90

Глава 3. Нейронные сети прямого распространения ... 93
Сетевая архитектура ... 94
Выход нейронов ... 96
Сводка матричных размерностей ... 97
Пример: уравнения для сети с тремя слоями ... 97
Гиперпараметры в полносвязных сетях ... 98
Функция softmax для многоклассовой классификации ... 99
Краткое отступление: переподгонка ... 100
Практический пример переподгонки ... 100
Простой анализ ошибок ... 106
Набор данных Zalando ... 108
Построение модели с помощью TensorFlow ... 111
Сетевая архитектура ... 111
Модификация меток для функции softmax — кодировка с одним активным состоянием ... 113
Модель TensorFlow ... 116
Варианты градиентного спуска ... 119
Пакетный градиентный спуск ... 119
Стохастический градиентный спуск ... 120
Мини-пакетный градиентный спуск ... 121
Сравнение вариантов градиентного спуска ... 123
Примеры неправильных предсказаний ... 127
Инициализация весов ... 128
Эффективное добавление многочисленных слоев ... 130
Преимущества дополнительных скрытых слоев ... 132
Сравнение разных сетей ... 133
Советы по выбору правильной сети ... 137

Глава 4. Тренировка нейронных сетей ... 139
Динамическое ослабление темпа заучивания ... 139
Итерации или эпохи? ... 141
Ступенчатое ослабление ... 142
Пошаговое ослабление ... 143

Обратно-временное ослабление ... 146
Экспоненциальное ослабление ... 149
Естественное экспоненциальное ослабление ... 150
Реализация в TensorFlow ... 155
Применение описанных методов к набору данных Zalando ... 159
Широко используемые оптимизаторы ... 160
Экспоненциально взвешенные средние ... 160
Momentum ... 164
RMSProp ... 167
Adam ... 170
Какой оптимизатор следует использовать? ... 172
Пример оптимизатора собственной разработки ... 173

Глава 5. Регуляризация ... 179
Сложные сети и переподгонка ... 179
Что такое регуляризация? ... 183
О сетевой сложности ... 185
Норма l_p ... 185
Регуляризация l_2 ... 185
Теоретическое обеспечение регуляризации l_2 ... 185
Реализация в TensorFlow ... 187
Регуляризация l_1 ... 196
Теоретическое обеспечение регуляризации l_1 и ее реализации в TensorFlow ... 196
Веса действительно сходятся к нулю? ... 198
Отсев ... 200
Досрочная остановка ... 203
Дополнительные методы ... 204

Глава 6. Метрический анализ ... 207
Человеческая результативность и байесова ошибка ... 208
Краткая история человеческой результативности ... 211
Человеческая результативность на наборе данных MNIST ... 213
Смещение ... 214
Диаграмма метрического анализа ... 215
Переподгонка к тренировочному набору данных ... 216
Тестовый набор ... 217
Как подразделить набор данных ... 219
Распределение несбалансированных классов: что может произойти ... 223
Метрики прецизионности, полноты и F1 ... 227
Наборы данных с разными распределениями ... 232
K-блочная перекрестная проверка ... 239
Ручной метрический анализ: пример ... 247

Глава 7. Гиперпараметрическая настройка ... 253
Черно-ящичная оптимизация ... 253
Замечания по черно-ящичным функциям ... 255
Задача гиперпараметрической настройки ... 256
Образец черно-ящичной задачи ... 257
Решеточный поиск ... 258

Случайный поиск ... 262
Оптимизация с переходом "от крупнозернистости к мелкозернистости" ... 265
Байесова оптимизация ... 269
Регрессия Надарая – Ватсона ... 269
Гауссов процесс ... 270
Стационарный процесс ... 271
Предсказание с помощью гауссовых процессов ... 271
Функция обнаружения ... 277
Верхняя доверительная граница ... 277
Пример ... 278
Отбор образцов на логарифмической шкале ... 285
Гиперпараметрическая настройка с набором данных Zalando ... 287
Краткое замечание о радиальной базисной функции ... 294

Глава 8. Сверточные и рекуррентные нейронные сети ... 297
Ядра и фильтры ... 297
Свертка ... 298
Примеры свертки ... 306
Сведение ... 311
Заполнение ... 314
Строительные блоки CNN-сети ... 315
Сверточные слои ... 315
Сводящие слои ... 316
Стековая укладка слоев ... 317
Пример CNN-сети ... 317
Введение в RNN-сети ... 322
Обозначения ... 323
Основная идея RNN-сетей ... 324
Почему именно "рекуррентная" сеть? ... 325
Учимся считать ... 325

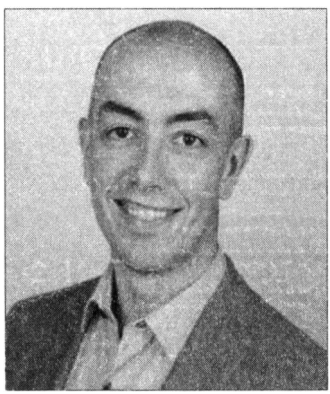
Глава 9. Исследовательский проект ... 331
Описание задачи ... 331
Математическая модель ... 334
Регрессионная задача ... 335
Подготовка набора данных ... 340
Тренировка модели ... 347

Глава 10. Логистическая регрессия с нуля ... 353
Математический каркас логистической регрессии ... 354
Реализация на Python ... 357
Тестирование модели ... 359
Подготовка набора данных ... 359
Выполнение теста ... 360
Заключение ... 361

Предметный указатель ... 362

Посвящаю эту книгу моей дочери Катерине и жене Франческе. Спасибо за вдохновение, мотивацию и счастье, которые вы приносите в мою жизнь каждый день. Без вас это было бы невозможно

Об авторе



Умберто Микелуччи в настоящее время работает в области инноваций и искусственного интеллекта (ИИ) в ведущей медицинской страховой компании Швейцарии. Он возглавляет несколько стратегических инициатив, связанных с ИИ, новыми технологиями, машинным обучением и исследовательским сотрудничеством с университетами. Ранее он работал в качестве исследователя данных и ведущего моделиста для нескольких крупных проектов в области здравоохранения и приобрел большой практический опыт в программировании и разработке алгоритмов. Он руководил проектами в области бизнес-аналитики и хранения данных по реализации управляемых данными решений в сложных производственных средах. В последнее время Умберто активно работал с нейронными сетями и занимался применением глубокого обучения к нескольким задачам, связанным со страхованием, поведением клиентов (например, уходом клиентов) и сенсорной наукой. Он изучал теоретическую физику в Италии, США и Германии, где также работал исследователем. Высшее образование он получил в Великобритании.

Регулярно представляет научные результаты на конференциях, публикует научные статьи в рецензируемых журналах.

О рецензенте

Джоджо Мулайл является профессионалом в области искусственного интеллекта, глубокого обучения, машинного обучения и теории принятия решений с более чем пятилетним производственным стажем. Он автор книги "Smarter Decisions — The Intersection of IoT and Decision Science" ("Более умные решения — на стыке Интернет вещей (IoT) и теории принятия решений"). Работал с несколькими лидерами индустрии в области высокоэффективных и критически важных проектов науки о данных и машинного обучения по нескольким вертикалям. В настоящее время его имя ассоциируют с General Electric, первопроходцем и лидером в области науки о данных для промышленного IoT. Живет в Бенгалуру — Кремниевой долине Индии.

Джоджо родился и вырос в Пуне (Индия), окончил университет Пуны по специализации в области информационных технологий. Свою карьеру он начал в Mu Sigma Inc., крупнейшем в мире поставщике "чистокровной" аналитики, и работал со многими ведущими клиентами списка Fortune 50. Будучи одним из первых энтузиастов, рискнувших вложиться в IoT-аналитику, он объединил свои познания из теории принятия решений для того, чтобы привнести каркасы решения задач и свои познания из науки о данных и теории принятия решений в IoT-аналитику.

Для того чтобы укрепить свои познания науки о данных для промышленного Интернета вещей и масштабировать воздействие экспериментов с применением теории принятия решений, он присоединился к быстро растущему стартапу в области IoT-аналитики под названием Flutura, базирующемуся в Бангалоре и со штабквартирой в долине. После непродолжительной работы с Flutura, Джоджо перешел на работу с лидерами производственного IoT — General Electric, в Бангалоре, где и сосредоточился на работе с задачами принятия решений в промышленных IoT-приложениях. В рамках своей роли в General Electric Джоджо также фокусируется на развитии продуктов и платформ науки о данных и принятия решений для промышленного Интернета вещей.

Помимо написания книг по теории принятия решений и IoT, Джоджо также выступал техническим рецензентом различных книг по машинному обучению, глубокому обучению и бизнес-аналитике с публикациями в издательствах Apress и Packt. Он является действующим преподавателем науки о данных и ведет блог по адресу **http://www.jojomoolayil.com/web/blog/**.

Профиль:

- **http://www.jojomoolayil.com/**;
- **https://www.linkedin.com/in/jojo62000**.

"Хочу поблагодарить свою семью, друзей и наставников".

— Джоджо Мулайл

Признательности

Было бы несправедливо, если бы я не поблагодарил всех тех людей, которые помогли мне с этой книгой. Во время написания книги я обнаружил, что совершенно ничего не знаю о книгоиздании, а также что даже когда вы думаете, что знаете что-то хорошо, положить это на бумагу — совершенно другая история. Невероятно, как якобы ясный ум человека искажается при изложении мыслей на бумаге. Это предприятие было одним из самых трудных, которые я когда-либо начинал, но это событие также было одним из самых полезных в моей жизни.

Во-первых, я должен поблагодарить мою любимую жену Франческу Вентурини (Francesca Venturini), которая проводила бесчисленные часы ночью и по выходным, читая текст. Без нее книга не была бы такой ясной. Я должен также поблагодарить Селестина Суреша Джона (Celestin Suresh John), который поверил в мою идею и дал мне возможность написать эту книгу. Адите Мираши (Aditee Mirashi) — самый терпеливый редактор, каких я когда-либо встречал. Она всегда была рядом, чтобы ответить на все мои вопросы, а у меня их было немало, и не все хорошие. Я особенно хотел бы поблагодарить Мэтью Муди (Matthew Moodie), у которого хватило терпения прочитать каждую главу. Я никогда не встречал никого, кто мог бы предложить так много хороших предложений. Спасибо, Мэтт, я перед тобой в долгу. У Джоджо Мулайла (Jojo Moolayil) хватило терпения проверить каждую строчку кода и убедиться в правильности каждого объяснения. И когда я говорю "каждую", я это имею в виду. Нет, правда, я серьезно. Спасибо, Джоджо, за твои отзывы и твою поддержку. Это действительно много значило для меня.

Наконец, я бесконечно благодарен моей любимой дочери Катерине (Caterina) за ее терпение, когда я писал, и за то, что она каждый день напоминала мне, как важно следовать своим мечтам. И конечно, я должен поблагодарить родителей, которые всегда поддерживали мои решения, какими бы они ни были.

Комментарии переводчика

Данная книга послужит ценным справочным руководством по разработке нейросетевых приложений вообще и с использованием библиотеки TensorFlow в частности. В ней содержится ряд ценных сведений и советов, которые трудно найти в Интернете, среди прочих касающиеся особенностей метрического анализа, предсказания с помощью гауссовых процессов, функции обнаружения, регрессии Надарая – Ватсона и байесовой оптимизации. Подробно описаны принципы работы полносвязных, сверточных и рекуррентных сетей и применение нейросетей в сенсорных приложениях. На YouTube-канале по адресу **https://www.youtube.com/channel/UC0JFHky44E1oYJWDVJuZf7g** можно найти несколько видеороликов автора, посвященных продвинутым темам.

Кодовая база книги была протестирована в среде Windows 10. При тестировании исходного кода за основу взят Python версии 3.7.0 (время перевода — март 2019 г.). *Главы 1, 2, 7* и *8* были проверены полностью, остальные — выборочно. Следует отметить, что исходные коды книги, размещенные в хранилище GitHub по адресу **https://github.com/Apress/applied-deep-learning**, в ряде случаев слегка отличаются от исходного кода в печатном издании, и это говорит только в пользу автора, который регулярно обновляет свой код после публикации книги.

Введение

Зачем нужна еще одна книга по прикладному глубокому обучению? Именно этот вопрос я задал себе перед тем, как приступить к написанию настоящей книги. Ведь стоит только погуглить по данной теме, и вы будете ошеломлены огромным числом результатов. Однако я столкнулся с проблемой, которая состояла в том, что я нашел материал для реализации только очень простых моделей на очень простых наборах данных. Снова и снова вам предлагают одни и те же задачи, одни те же подсказки и советы. Если вы хотите научиться классифицировать набор данных MNIST[1], который состоит из десяти рукописных цифр, то вам повезло. (Почти любой, у кого есть блог, уже это сделал, в основном копируя исходный код на вебсайте TensorFlow). Ищете что-то, чтобы понять, как работает логистическая регрессия? Не так-то просто. Хотите узнать, как подготовить набор данных для выполнения интересной бинарной классификации? Еще труднее. Я чувствовал, что есть необходимость заполнить этот пробел. К примеру, я потратил часы, пытаясь отладить модели, которые не работали по таким глупым причинам, как неправильные метки. В частности, вместо меток 0 и 1 у меня были метки 1 и 2, но ни один блог меня об этом не предупредил. Во время разработки моделей важно проводить правильный метрический анализ, но никто не демонстрирует, как это делать (по крайней мере, не на легкодоступном материале). Этот пробел необходимо было заполнить. По моему мнению, охват более сложных примеров, от подготовки данных до анализа ошибок, является очень эффективным и интересным способом изучения правильных технических решений. В этой книге я все время старался охватывать полные и сложные примеры с целью объяснить понятия, которые не так легко усвоить каким-либо другим способом. Невозможно понять важность подбора правильного темпа заучивания, если вы не видите, что может произойти при выборе неправильного его значения. Поэтому я все время объясняю понятия на реальных примерах и на полноценном и протестированном исходном Python-коде, который можно использовать многократно. Обратите внимание, что цель этой книги вовсе не в том, чтобы сделать из вас эксперта в программировании на языке Python или библиотеке

---

[1] Модифицированный набор данных института NIST (National Institute of Standards and Technology), Национального института стандартов и технологий, США. — *Прим. пер.*

TensorFlow, или кем-то, кто может разрабатывать новые сложные алгоритмы. Python и TensorFlow — это просто инструменты, которые очень хорошо подходят для разработки моделей и быстрого получения результатов. Поэтому я ими и пользуюсь. Я мог бы использовать и другие инструменты, но эти как раз те, которые практики применяют чаще всего, и поэтому вполне резонно было выбрать именно их. Если вам еще предстоит только научиться, то лучше, когда вы работаете именно с тем, что можете использовать в собственных проектах и для своей карьеры.

Цель этой книги — дать вам увидеть более продвинутый материал новой точки зрения. Я освещаю математические основы максимально глубоко в той мере, в какой, по моим ощущениям, это необходимо для полного понимания трудностей и рассуждений, стоящих за многими понятиями. Невозможно понять, почему большой темп заучивания будет побуждать вашу модель (строго говоря, стоимостную функцию) расходиться, если вы не знаете, каким образом алгоритм градиентного спуска работает математически. Во всех реальных проектах вам не придется рассчитывать частные производные или компле́ксные суммы, но вы должны их понимать для того, чтобы уметь оценить то, что может работать, а что — нет (и в особенности, почему). По достоинству оценить, почему библиотека TensorFlow упрощает вам жизнь, можно только в том случае, если вы попытаетесь разработать с нуля тривиальную модель с одним-единственным нейроном. Это очень показательная вещь, и я покажу вам, как это сделать, в *главе 10*. Сделав это один раз, вы запомните весь ход работы навсегда и по-настоящему оцените важность таких библиотек, как TensorFlow.

Я предлагаю вам попытаться действительно разобраться в математических основах (хотя это и не является строго необходимым для того, чтобы извлечь выгоду из этой книги), потому что они позволят вам усвоить многие понятия, которые в противном случае невозможно усвоить полностью. Машинное обучение — очень сложный предмет, и утопично думать, что его можно освоить полностью без хорошего понимания математического каркаса или Python. В каждой главе я даю важные советы по эффективной разработке на Python. В этой книге нет такого утверждения, которое не подкреплялось бы конкретными примерами и воспроизводимым исходным кодом. Я не буду ничего обсуждать, не приведя соответствующих примеров из реальной жизни. Благодаря этому все сразу обретет смысл, и вы это запомните.

Потратьте время на изучение исходного кода, который вы найдете в этой книге, и пробуйте сами. Каждый хороший преподаватель знает, что усвоение учебного материала лучше всего проходит тогда, когда студенты пытаются решать задачи самостоятельно. Старайтесь, ошибайтесь и учитесь. Прочитайте главу, наберите исходный код и попробуйте его модифицировать. Например, в *главе 2* я покажу вам, как выполнять распознавание на основе бинарной классификации между двумя написанными от руки цифрами: 1 и 2. Возьмите исходный код и попробуйте две другие цифры. Играйте с кодом, экспериментируйте и получайте удовольствие.

По замыслу исходный код, который вы найдете в этой книге, написан максимально просто. Он не оптимизирован, и я знаю, что можно написать гораздо эффективнее,

но поступая так, я бы пожертвовал ясностью и удобочитаемостью. Цель этой книги не в том, чтобы научить вас писать высокооптимизированный исходный код на Python, а в том, чтобы вы вникли в фундаментальные алгоритмические понятия и их ограничения и получили прочную основу для продолжения своего обучения в этой области. Несмотря на это, я, безусловно, укажу на важные детали реализации на Python, например, что вы должны как можно чаще избегать стандартных циклов Python.

Весь код в этой книге написан в поддержку учебных целей, которые я поставил для каждой главы. Такие библиотеки, как NumPy и TensorFlow, были рекомендованы, поскольку они позволяют переводить математические формулировки непосредственно на язык Python. Я в курсе о существовании других программных библиотек, таких как TensorFlow Lite, Keras и многих других, которые могут облегчить вам жизнь, но это всего лишь инструменты. Существенная разница заключается в вашей способности понимать концепции, лежащие в основе методов. Если вы понимаете их правильно, то можете выбрать любой инструмент и сумеете добиться хорошей реализации. Если вы не понимаете того, как работают алгоритмы, то независимо от инструмента, вы не сможете выполнить надлежащую реализацию или надлежащий анализ ошибок. Я ярый противник концепции науки о данных для всех. Наука о данных и машинное обучение являются трудными и многосложными предметами, которые требуют глубокого понимания математики и стоящих за ними тонкостей.

Я надеюсь, что вы получите удовольствие от чтения этой книги (у меня точно его было вдоволь при написании этой книги) и найдете примеры и исходный код полезными. Я надеюсь, что у вас будет много моментов озарения, когда вы, наконец, поймете, почему что-то работает так, как вы и ожидаете (или почему — нет). Надеюсь, вы найдете полнофункциональные примеры интересными и полезными. И даже если я помогу вам разобраться лишь в одном термине, который раньше для вас был непонятен, то буду просто счастлив.

Несколько глав этой книги математически являются более продвинутыми. Например, в *главе 2* вычисляются частные производные. Но не переживайте. Если вы их не понимаете, то уравнения можно пропустить. Я сделал так, чтобы основные понятия легко воспринимались без большинства математических деталей. Тем не менее вы все-таки должны знать, что такое матрица, как умножать матрицы, что такое транспонирование матрицы и т. д. В принципе, у вас должно быть хорошее понимание линейной алгебры. Если у вас его нет, то предлагаю вам ознакомиться с какой-нибудь вводной книгой по линейной алгебре и только потом приступать к чтению данной книги. Если у вас есть прочный фундамент в линейной алгебре и дифференциальном исчислении, то я настоятельно рекомендую не пропускать математические разделы. Они действительно помогут понять, почему мы делаем что-то тем или иным образом. Например, это очень поможет вам понять причуды темпа заучивания или то, как работает алгоритм градиентного спуска. Вы также не должны бояться более сложных обозначений и чувствовать себя уверенно с таким сложным уравнением, как то, которое приведено ниже. (Это среднеквадратическая

ошибка, которую мы будем использовать для алгоритма линейной регрессии, и она будет подробно объяснена позже, поэтому не переживайте, если на данный момент вы не знаете, что означают эти символы.)

$$J(w_0, w_1) = \frac{1}{m}\sum_{i=1}\left(y_i - f\left(w_0, w_1, x^{(i)}\right)\right)^2 .$$

Вы должны разбираться и чувствовать себя уверенно с такими понятиями, как сумма или математический ряд. Если вы в них не уверены, то перед началом чтения книги проведите ревизию своих знаний по этим темам; в противном случае вы пропустите некоторые важные понятия, о которых должны иметь четкое представление с целью продолжения своей карьеры в сфере глубокого обучения. Цель этой книги не в том, чтобы дать вам прочную математическую основу. Полагаю, она у вас есть. Глубокое обучение и нейронные сети (в общем смысле, машинное обучение) — очень сложны, и тот, кто пытается убедить вас в обратном, обманывает или просто их не понимает.

Я не буду тратить время на обоснование или математическое выведение алгоритмов или уравнений. Вам придется мне довериться. Кроме того, я не буду обсуждать применимость конкретных уравнений. Для тех из вас, кто хорошо разбирается в дифференциальном исчислении, например, я не буду обсуждать вопрос дифференцируемости функций, для которых мы вычисляем производные. Просто будем считать, что вы можете применять формулы, которые я вам даю. Как показали многие годы их практической реализации сообществом глубокого обучения, эти методы и уравнения работают, как и ожидалось, и могут применяться на практике. Такого рода продвинутые темы, о которых упомянуто выше, потребуют отдельной книги.

В *главе 1* вы узнаете, как настроить среду Python и что такое вычислительный граф. Я расскажу о некоторых основных примерах математических расчетов, выполняемых с помощью TensorFlow. В *главе 2* мы рассмотрим, что можно делать с одним-единственным нейроном. Я расскажу, что такое активационная функция и какие их типы применяются на практике наиболее часто, в частности, такие как сигмоида, ReLU или tanh. Я покажу вам, как работает градиентный спуск и как реализовать логистическую и линейную регрессию с одним нейроном и с помощью TensorFlow. В *главе 3* мы рассмотрим полносвязную сеть. Я расскажу о размерностях матрицы, о том, что такое переподгонка, и познакомлю вас с набором данных Zalando. Затем мы построим первую реальную сеть с помощью TensorFlow и начнем рассматривать более сложные варианты алгоритмов градиентного спуска, такие как мини-пакетный градиентный спуск. Мы также рассмотрим различные способы инициализации весов и то, как сравнивать различные сетевые архитектуры. В *главе 4* мы рассмотрим алгоритмы динамического ослабления темпа заучивания, такие как ступенчатое, пошаговое или экспоненциальное ослабление, а затем обсудим передовые оптимизаторы, такие как Momentum, RMSProp и Adam. Я также дам вам несколько советов о том, как с помощью TensorFlow можно разрабатывать пользовательские оптимизаторы. В *главе 5* я расскажу о регуляризации, включая такие из-

вестные методы, как l_1, l_2, отсев и досрочная остановка. Мы рассмотрим алгоритмы этих методов и способы их реализации в TensorFlow. В *главе 6* мы рассмотрим такие понятия, как человеческая результативность и байесова ошибка. Затем я представлю рабочий процесс метрического анализа, который позволит выявлять проблемы, связанные с набором данных. Кроме того, мы рассмотрим *k*-блочную перекрестную проверку как инструмент валидации ваших результатов. В *главе 7* мы рассмотрим класс черно-ящичных задач и что собой представляет гиперпараметрическая настройка. Мы рассмотрим такие алгоритмы, как решеточный и случайный поиск, и выясним, какие из них эффективнее и почему. Затем мы рассмотрим некоторые приемы, такие как оптимизация с переходом от крупнозернистости к мелкозернистости. Я посвятил большую часть этой главы байесовой оптимизации — тому, как ее использовать и что такое функция обнаружения. Я дам несколько советов относительно того, например, как настраивать гиперпараметры на логарифмической шкале, а затем мы выполним гиперпараметрическую настройку на наборе данных Zalando с целью продемонстрировать то, как это может работать на практике. В *главе 8* мы рассмотрим сверточные и рекуррентные нейронные сети. Я покажу, что подразумевается под сверткой и сведением, и покажу базовую реализацию обеих архитектур в TensorFlow. В *главе 9* я расскажу вам о реальном исследовательском проекте, над которым работаю в Цюрихском университете прикладных наук в Винтертуре, и о том, как глубокое обучение может использоваться менее стандартным способом. Наконец, в *главе 10* я покажу вам, как выполнять логистическую регрессию с одним-единственным нейроном на Python — без использования TensorFlow — совершенно с нуля.

Очень надеюсь, что эта книга вам понравится, и вы получите от нее удовольствие.

ГЛАВА 1

Вычислительные графы и TensorFlow

Прежде чем погрузиться в изучение расширенных примеров далее в этой книге, вам потребуются среда программирования на языке Python и рабочие знания платформы машинного обучения TensorFlow. И данная глава поможет вам подготовить среду программирования на Python к выполнению исходного кода этой книги. После того как все будет готово, мы рассмотрим основы библиотеки машинного обучения TensorFlow.

Настройка среды программирования на языке Python

Весь исходный код в этой книге был разработан с использованием дистрибутива Python Anaconda и блокнотов Jupyter. Для того чтобы настроить дистрибутив Anaconda, сначала скачайте и установите его для своей операционной системы. (В книге использовалась ОС Windows 10, но указанный исходный код от этой операционной системы не зависит. Если хотите, то вы легко можете использовать его версию для Mac.) Вы можете получить дистрибутив Anaconda, обратившись по адресу **https://anaconda.org/**.

В правой части указанной веб-страницы вы найдете ссылку на скачивание дистрибутива Anaconda (рис. 1.1, справа вверху).

РИС. 1.1. На веб-сайте Anaconda в правом верхнем углу страницы вы найдете ссылку для скачивания необходимого программного обеспечения

Просто следуйте инструкциям по его установке. Когда после установки вы его запустите, вам будет представлен экран, показанный на рис. 1.2. В случае если этот экран вы не видите, щелкните по ссылке **Home** (Главная) на панели навигации слева.

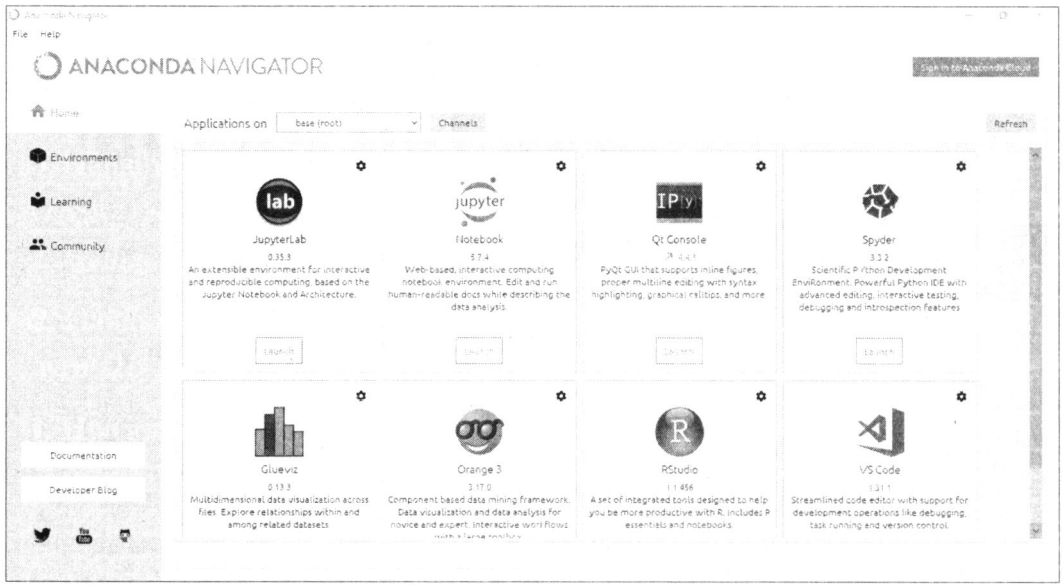


РИС. 1.2. Экран, который вы видите при запуске дистрибутива Anaconda

Программные Python-пакеты (в частности, NumPy) обновляются регулярно и очень часто. Возможно, новая версия пакета приведет к тому, что ваш код перестанет работать. Время от времени функции объявляются устаревшими, удаляются и добавляются новые. Для решения этой проблемы в Anaconda можно создавать так называемую среду. Это контейнер, который содержит определенную версию Python и конкретные версии пакетов, которые вы решили установить. Благодаря ей, например, вы можете организовать контейнер для Python 2.7 и NumPy 1.10 и еще один с Python 3.6 и NumPy 1.13. Возможно, вам придется работать с существующим кодом, который был разработан на основе Python 2.7, и, следовательно, у вас должен быть контейнер с правильной версией Python. В то же время вашим собственным проектам может потребоваться Python 3.6. С помощью контейнеров вы можете обеспечить все это одновременно. Иногда разные пакеты конфликтуют друг с другом, поэтому следует быть осторожным и избегать установки в своей среде всех тех пакетов, которые вы посчитаете интересными, в особенности, если вы разрабатываете пакеты в условиях предельного срока. Нет ничего хуже, чем обнаружить, что ваш программный код перестанет работать, и вы не знаете почему.

ПРИМЕЧАНИЕ. При определении среды старайтесь устанавливать только те программные пакеты, которые вам действительно нужны, и будьте внимательны, когда их обновляете с целью обеспечения того, чтобы никакое обновление

не нарушило ваш исходный код. (Функции объявляются устаревшими, удаляются, добавляются или часто изменяются.) Перед обновлением проверьте документацию относительно обновлений и делайте это только в том случае, если вам действительно нужен обновленный функционал.

Создать среду можно из командной строки с помошью команды `conda`, но для подготовки среды для исходного кода книги все можно проделать из графического интерфейса. Здесь будет рассмотрен именно этот метод, потому что он самый простой. Рекомендуется прочитать приведенную далее страницу из документации дистрибутива Anaconda, где подробно объясняются особенности работы внутри его среды: **https://conda.io/docs/user-guide/tasks/manage-environments.html**.

Создание среды

Давайте начнем. Сначала щелкните по ссылке **Environments** (Среды) (с маленьким значком в виде куба) на левой навигационной панели (рис. 1.3).

РИС. 1.3. Для того чтобы создать новую среду, сначала необходимо перейти в раздел **Environments** приложения, щелкнув по соответствующей ссылке на левой навигационной панели (обозначена на рисунке кубиком)

Затем нажмите кнопку **Create** (Создать) в средней навигационной панели (как показано на рис. 1.4).

После нажатия кнопки **Create** (Создать) откроется небольшое окно (рис. 1.5).

Вы можете выбрать любое имя. В этой книге использовалось имя tensorflow. После того как вы наберете имя, кнопка **Create** станет активной (и зеленой). Нажмите ее и подождите несколько минут до тех пор, пока не будут установлены все необходимые пакеты. Иногда может появиться всплывающее окно, сообщающее о том, что имеется новая версия дистрибутива Anaconda, и спрашивающее о том, не хотите ли вы обновить программное обеспечение. Смело нажмите кнопку **Yes**. Следуйте

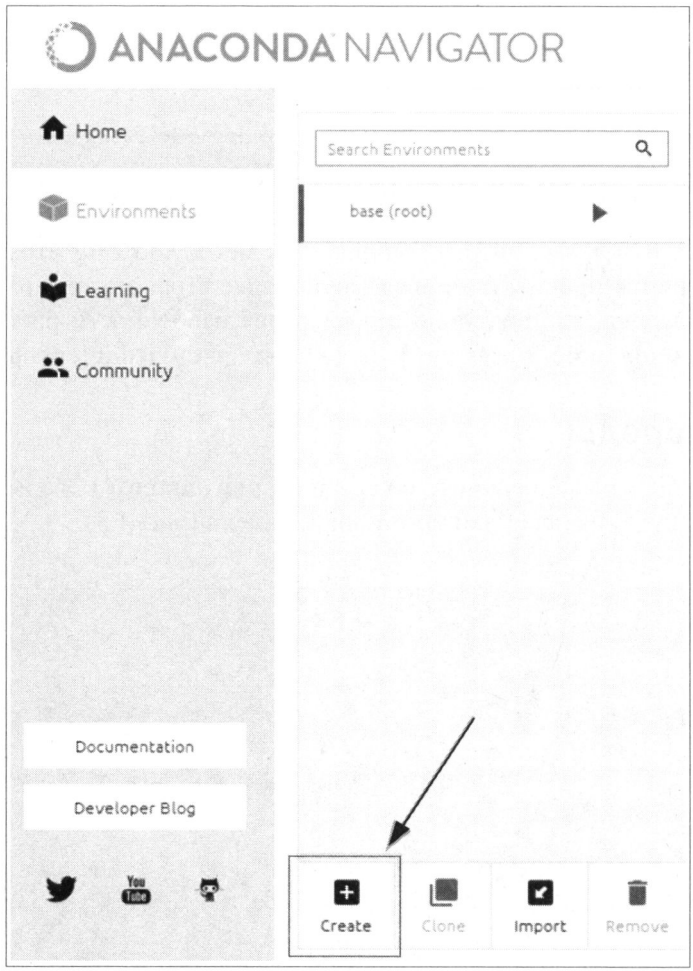


РИС. 1.4. Для того чтобы создать новую среду, необходимо нажать кнопку **Create** (обозначенную значком "плюс") на средней навигационной панели. На рисунке стрелка указывает на расположение данной кнопки

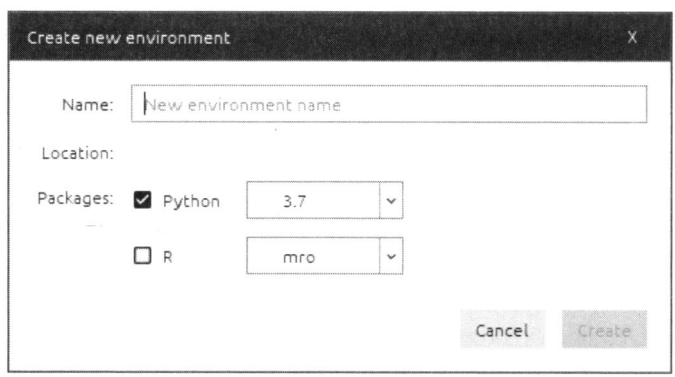


РИС. 1.5. Окно, которое вы увидите, когда нажмете кнопку **Create**, показанную на рис. 1.4

инструкциям на экране до тех пор, пока навигатор дистрибутива Anaconda не запустится снова, в случае если вы получили это сообщение и нажали кнопку **Yes**.

Мы еще не закончили. Снова щелкните по ссылке **Environments** на левой навигационной панели (см. рис. 1.3), затем щелкните на имени вновь созданной среды. Если до настоящего момента вы следовали инструкциям, то должны увидеть среду с именем "tensorflow". Через несколько секунд на правой панели вы увидите список всех установленных Python-пакетов, которые будут в вашем распоряжении в этой среде. Теперь мы должны установить несколько дополнительных пакетов: NumPy, matplotlib, TensorFlow и Jupyter. Для начала в раскрывающемся списке выберите пункт **Not installed** (Не установлены), как показано на рис. 1.6.

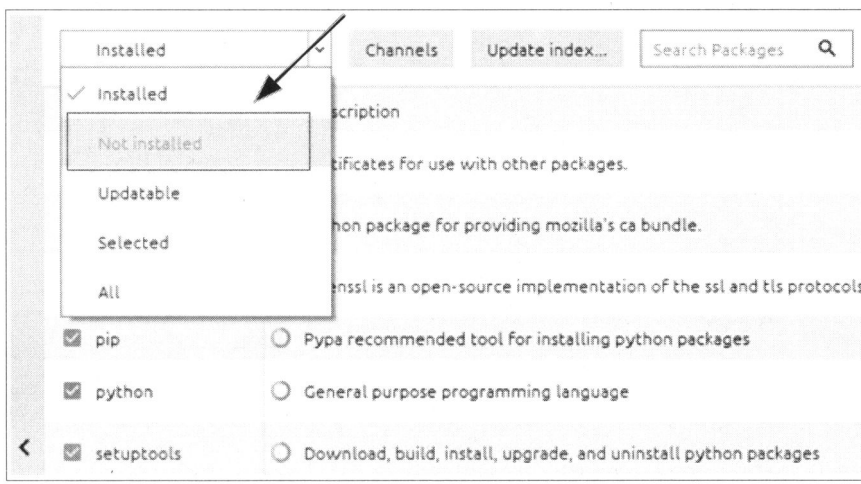


РИС. 1.6. Выбор пункта **Not installed** в раскрывающемся списке

Затем в поле **Search Packages** (Поиск пакетов) введите имя пакета, который вы хотите установить (на рис. 1.7 показано, что выбран элемент `numpy`).

Навигатор Anaconda автоматически покажет вам все пакеты, в названии или описании которых есть слово *numpy*. Щелкните на маленьком квадратике слева от имени пакета с именем *numpy*. Он станет небольшой стрелкой вниз (обозначающей, что он отмечен для установки). Затем можно нажать зеленую кнопку **Apply** (Применить) в правом нижнем углу интерфейса (рис. 1.8).

Навигатор Anaconda достаточно умен, чтобы определить, нужны ли пакету NumPy другие пакеты. Вы можете получить дополнительное окно с вопросом, можно ли установить дополнительные пакеты. Просто нажмите кнопку **Apply**. На рис. 1.9 показано, как это окно выглядит.

Для выполнения исходного кода этой книги необходимо установить следующие пакеты. (В скобках указаны версии, которые использовались для тестирования исходного кода в этой книге; никаких проблем, если это будут последующие версии.)

- numpy (1.13.3): для выполнения численных расчетов.
- matplotlib (2.1.1): для создания качественных графиков, как те, которые вы увидите в этой книге.
- scikit-learn (0.19.1): этот пакет содержит все библиотеки, связанные с машинным обучением, которые мы используем, например, для загрузки наборов данных.
- jupyter (1.0.0): позволяет использовать блокноты Jupyter.

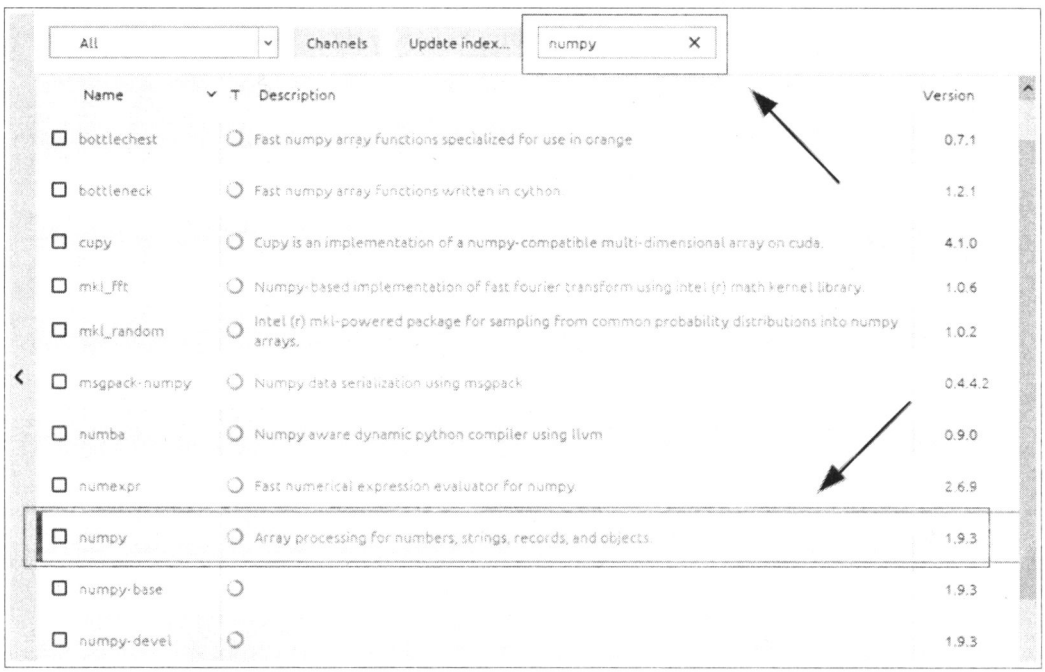


РИС. 1.7. Наберите *numpy* в поле поиска для того, чтобы включить этот пакет в репозиторий

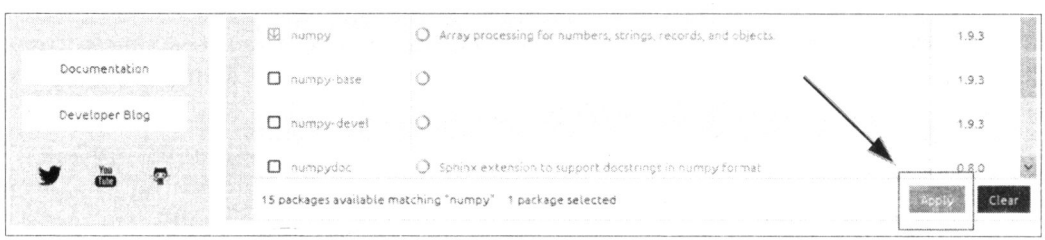


РИС. 1.8. После выбора программного пакета NumPy для установки нажмите зеленую кнопку **Apply**. Кнопка находится в правом нижнем углу интерфейса

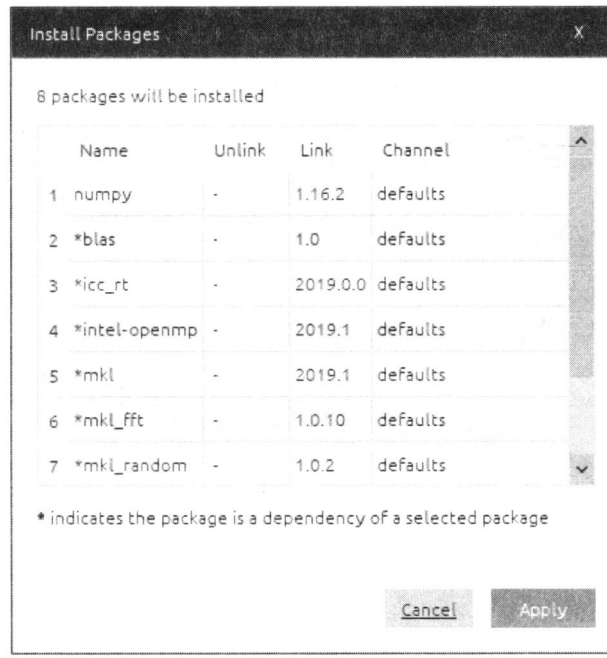


РИС. 1.9. Во время установки программного пакета навигатор Anaconda проверяет, зависит ли то, что вы хотите установить, от других еще не установленных пакетов. В таком случае будет предложено установить отсутствующие (но необходимые) пакеты из дополнительного окна. В нашем случае библиотека NumPy потребовала установки 52 дополнительных пакетов в недавно установленной системе. Просто нажмите кнопку **Apply** для того, чтобы установить их все

Установка библиотеки TensorFlow

Установка библиотеки TensorFlow проходит немного сложнее. Лучший способ сделать это — следовать инструкциям команды разработчиков TensorFlow, которые размещены по следующему адресу: **www.tensorflow.org/install/**.

На этой странице выберите вашу операционную систему, и вы получите всю необходимую информацию. Здесь будут приведены инструкции для Windows, но то же самое можно сделать, используя системы macOS или Ubuntu (Linux). Установка с Anaconda официально не поддерживается, но работает отлично (поддерживается сообществом) и представляет собой самый простой способ подготовки к работе и проверке исходного кода этой книги. В случае более продвинутых приложений можно рассмотреть другие варианты установки. (Для этого вам нужно будет обратиться на веб-сайт TensorFlow.) Для начала перейдите в меню **Пуск** в Windows и введите *anaconda*. В разделе **Лучшее соответствие** вы должны увидеть пункт **Anaconda Prompt** (Консоль Anaconda), как показано на рис. 1.10.

Запустите консоль Anaconda. В результате должен появиться интерфейс командной строки (рис. 1.11). Разница между этой и обычной консолью командной строки

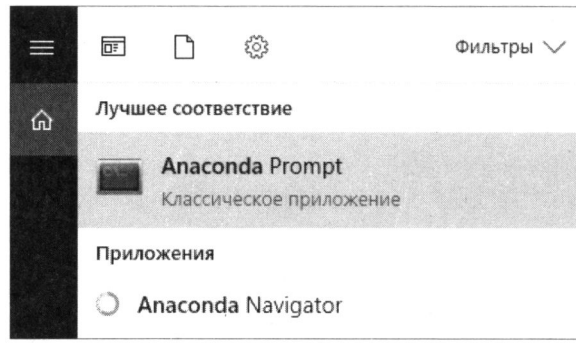


РИС. 1.10. Если в поле поиска меню **Пуск** в Windows 10 набрать *anaconda*, то можно увидеть по крайней мере две записи: **Anaconda Navigator**, где вы создали среду TensorFlow, и **Anaconda Prompt**

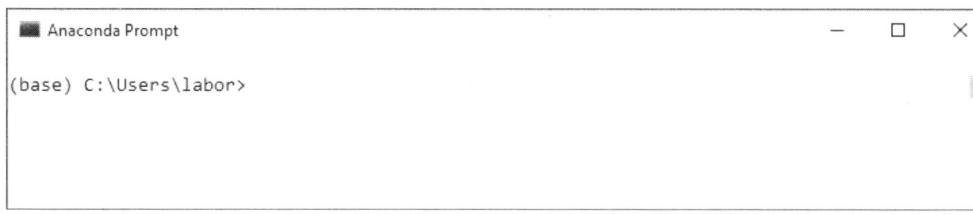


РИС. 1.11. Это то, что вы должны увидеть при выборе консоли **Anaconda Prompt**. Обратите внимание, что ваше пользовательское имя будет отличаться от имени на рисунке. Вы увидите не labor (мое пользовательское имя), а ваше собственное пользовательское имя

`cmd.exe` состоит в том, что здесь все команды Anaconda распознаются без настройки какой-либо переменной среды Windows.

В командной строке сначала необходимо активировать новую среду tensorflow. Это необходимо для того, чтобы дать установленной версии Python знать, в какой среде вы хотите установить TensorFlow. Для этого просто наберите следующую команду: `activate tensorflow`. Приглашение на ввод команды в вашей консоли должно измениться и выглядеть так:

```
(tensorflow) C:\Users\labor>
```

Ваше пользовательское имя будет отличаться (в приглашении на ввод команды вы увидите не labor, а ваше пользовательское имя). Будем считать, что вы установите стандартную версию TensorFlow, которая использует только CPU (а не GPU). Просто наберите следующую команду:

```
pip install --ignore-installed --upgrade tensorflow
```

Теперь дайте системе установить все необходимые пакеты. Это может занять несколько минут (в зависимости от таких факторов, как аппаратное обеспечение вашего компьютера или подключение к Интернету). Вы не должны получать никаких сообщений об ошибках. Поздравляю! Теперь у вас есть среда, в которой вы можете выполнять исходный код с использованием TensorFlow.

Блокноты Jupyter

Последний шаг, который даст возможность набирать и исполнять исходный код, состоит в запуске блокнота Jupyter. Блокнот Jupyter можно описать (согласно официальному веб-сайту) следующим образом:

> "Блокнот Jupyter — это веб-приложение с открытым исходным кодом, которое позволяет создавать и совместно использовать документы, содержащие „живой" исходный код, уравнения, визуализации и повествовательный текст. Варианты его использования включают очистку и преобразование данных, численную симуляцию, статистическое моделирование, визуализацию данных, машинное обучение и многое другое".

В сообществе машинного обучения он применяется очень широко, и неплохо научиться его использовать. Обратитесь к веб-сайту проекта Jupyter по адресу **http://jupyter.org/**.

Этот сайт очень поучителен и содержит много примеров того, что можно делать. Весь исходный код, который вы найдете в этой книге, был разработан и протестирован с использованием блокнотов Jupyter.

Будем считать, что у вас уже есть некоторый опыт работы с этой средой веб-разработки. В случае если вам нужно освежить свои знания, рекомендуется обратиться к документации, которую можно найти на веб-сайте проекта Jupyter по следующему адресу: **http://jupyter.org/documentation.html**.

Для того чтобы запустить блокнот в вашей новой среде, необходимо вернуться в навигатор дистрибутива Anaconda в раздел **Environments** (см. рис. 1.3). Щелкните на треугольнике справа от среды "tensorflow" (если вы использовали другое имя, вам придется щелкнуть на треугольнике справа от вашей новой среды), как показано на рис. 1.12. Затем выберите пункт **Open with Jupyter Notebook** (Открыть с помощью блокнота Jupyter).

РИС. 1.12. Для того чтобы запустить записную книжку Jupyter в вашей новой среде, щелкните на треугольнике справа от имени среды "tensorflow" и выберите пункт **Open with Jupyter Notebook**

Ваш браузер запустится со списком всех папок в вашей пользовательской папке. (Если вы используете Windows, то они обычно находятся по маршруту c:\Users\ <*Имя_пользователя*>, в котором вы должны заменить <*Имя_пользователя*> на ваше собственное пользовательское имя.) Оттуда следует перейти к папке, в которой вы хотите хранить файлы ваших блокнотов и из которой вы можете создать новый блокнот, нажав кнопку **New** (Новый), как показано на рис. 1.13.

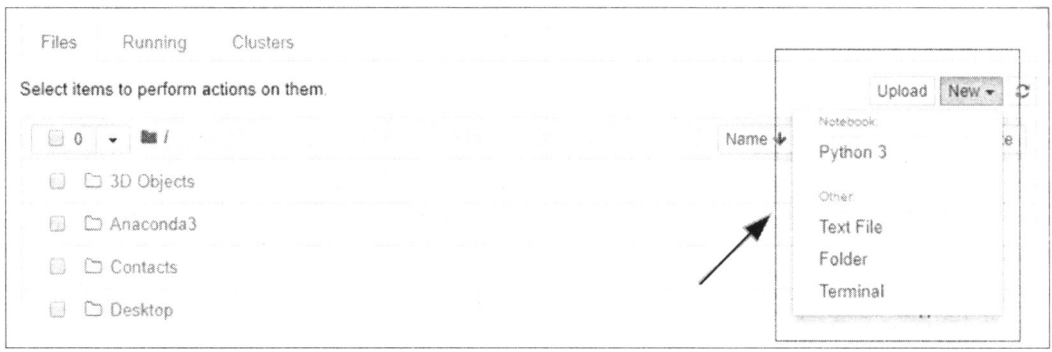


РИС. 1.13. Для того чтобы создать новый блокнот, нажмите кнопку **New** в правом верхнем углу страницы и выберите **Python 3**

Откроется новая страница, которая должна выглядеть так, как показано на рис. 1.14.

Например, вы можете набрать следующие ниже строки кода в первую "ячейку" (прямоугольное поле, в котором можно набирать текст).

```
a=1
b=2
print(a+b)
```

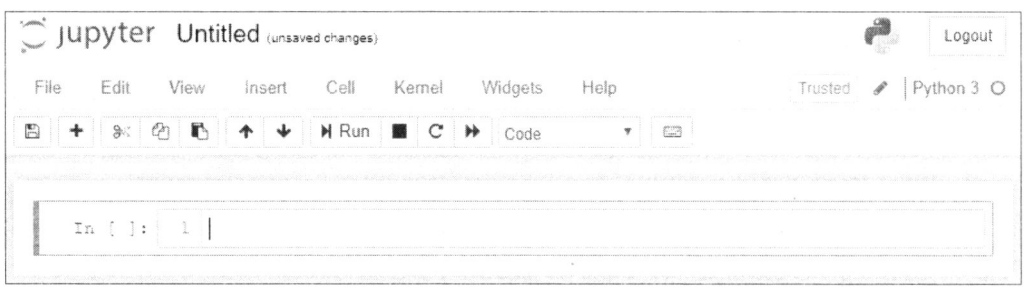


РИС. 1.14. При создании пустого блокнота откроется пустая страница, которая должна выглядеть так, как показано здесь

Для того чтобы вычислить этот исходный код, просто нажмите комбинацию клавиш <Shift>+<Enter>, и вы сразу увидите результат (3) (рис. 1.15).

Приведенный выше исходный код даст результат выражения $a+b$, т. е. 3. Новая пустая ячейка для ввода создается автоматически после получения результата. Для получения дополнительной информации о том, как добавлять комментарии, урав-

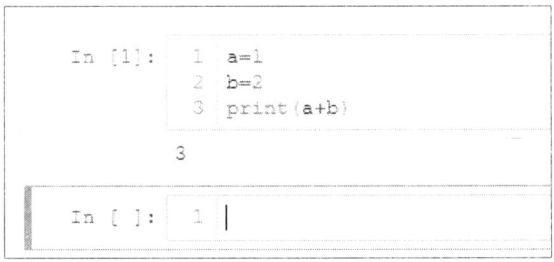


РИС. 1.15. После набора некоторого исходного кода в ячейку нажатие сочетания клавиш <Shift>+<Enter> вычислит исходный код в этой ячейке

нения, встроенные графики и многое другое, рекомендуется посетить веб-сайт Jupyter и обратиться к предоставленной там документации.

ПРИМЕЧАНИЕ. В случае если вы забыли имя папки, в которой находится блокнот, взгляните на URL-адрес страницы. Например, в моем случае, у меня **http://localhost:8888/notebooks/Documents/Data%20Science/Projects/Applied%20advanced%20deep%20learning%20(book)/chapter%201/AADL%20-%20Chapter%201%20-%20Introduction.ipynb**. Вы заметите, что URL-адрес является просто конкатенацией имен папок, в которых находится записная книжка, разделенных косой чертой. Символ %20 означает пробел. В данном случае мой блокнот находится в папке: Documents/Data Science/Projects/... и т. д. Я часто работаю с несколькими записными книжками одновременно, и полезно знать, где находится каждая из них, в случае если вы забыли ее расположение (иногда это бывает).

Элементарное введение в TensorFlow

Перед началом использования библиотеки TensorFlow вы должны понять ее философию.

Данная библиотека в значительной степени основана на концепции вычислительных графов, и если вы не понимаете, как они работают, то не сможете разобраться в том, как использовать эту библиотеку. Далее будет дано краткое введение в вычислительные графы и показано, как реализовывать простые вычисления с помощью TensorFlow. В конце следующего раздела вы должны понимать, как работает библиотека и как мы будем ее использовать в этой книге.

Вычислительные графы

Для понимания того, как работает библиотека TensorFlow, необходимо понять, что такое вычислительный граф. Вычислительный граф — это граф, в котором каждый узел соответствует операции или переменной. Переменные могут передавать свои значения в операции, а операции могут передавать свои результаты в другие операции. Обычно узлы изображают в виде круга (или многоточия) с именами перемен-

ных или операций внутри, и когда значение одного узла является входом для другого узла, стрелка проходит от одного узла к другому. Простейший граф, который только может быть, — это граф с единственным узлом, который представляет собой одну-единственную переменную. (Узел может быть переменной или операцией.) Граф на рис. 1.16 вычисляет значение переменной x.

РИС. 1.16. Простейший граф, который только можно построить, показывающий простую переменную

Не очень-то интересно. Теперь рассмотрим нечто посложнее, например сумму двух переменных x и y: $z = x + y$. Ее можно представить так, как на графе (рис. 1.17).

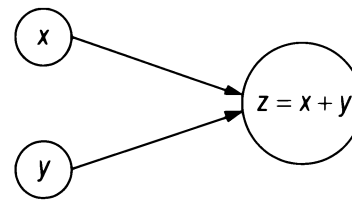


РИС. 1.17. Элементарный вычислительный граф для суммы двух переменных

Узлы слева на рис. 1.17 (узлы с x и y внутри) являются переменными, в то время как больший узел обозначает сумму двух переменных. Стрелки показывают, что две переменные, x и y, являются входами для третьего узла. Граф должен быть прочитан (и вычислен) в топологическом порядке, т. е. вы должны следовать по стрелкам, которые будут указывать, в каком порядке вы должны вычислять различные узлы. Стрелки также будут сообщать о зависимостях между узлами. Для того чтобы вычислить z, сначала необходимо вычислить x и y. Можно также сказать, что узел, выполняющий суммирование, зависит от входных узлов.

Важным для понимания аспектом является то, что такой граф определяет только те операции (в данном случае суммирование), которые нужно выполнить над двумя входными значениями (в данном случае x и y) для получения результата (в данном случае z). Таким образом определяется ответ на вопрос "как?". Вы должны закрепить значения за входами x и y, а затем выполнить суммирование и на выходе получить z. Данный граф даст результат, только когда вы вычислите все узлы.

ПРИМЕЧАНИЕ. В этой книге под словом "конструирование" графа подразумевается ситуация, когда мы будем определять то, что делает каждый узел, а под словом "оценивание" графа — ситуация, когда мы будем фактически вычислять участвующие операции.

Этот аспект очень важен для понимания. Обратите внимание, что входные переменные не обязательно должны быть вещественными числами. Они могут быть

матрицами, векторами и т. д. (В данной книге мы будем в основном использовать матрицы.) Пример немного посложнее можно найти на рис. 1.18. В нем используется граф для вычисления величины $A \cdot (x + y)$ с учетом трех входных величин: x, y и A.

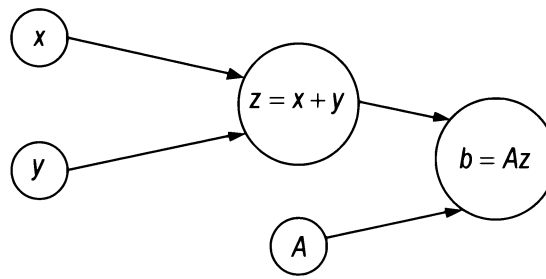


РИС. 1.18. Вычислительный граф для вычисления величины $A \cdot (x + y)$ с учетом трех входных величин: x, y и A

Мы можем вычислить этот граф, закрепив значения за входными узлами (в данном случае x, y и A) и вычислив узлы графа. Например, если взять граф на рис. 1.18 и назначить $x = 1$, $y = 3$ и $A = 5$, то мы получим результат $b = 20$ (рис. 1.19).

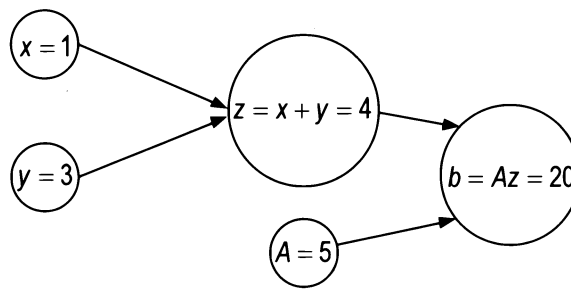


РИС. 1.19. Для того чтобы вычислить граф на рис. 1.18, необходимо закрепить значения за входными узлами x, y и A, а затем вычислить узлы графа

Нейронная сеть, в сущности, представляет собой очень сложный вычислительный граф, в котором каждый нейрон составлен из нескольких узлов, и эти узлы подают свой выход на вход определенного числа других нейронов до тех пор, пока не будет достигнуто определенное выходное значение. В следующем разделе мы построим простейшую из всех нейронных сетей: нейросеть с одним-единственным нейроном. Даже такая простая сеть способна делать несколько довольно забавных вещей.

TensorFlow позволяет очень легко создавать весьма сложные вычислительные графы. При этом по своей конструкции он отделяет их вычисление от конструирования. (Для того чтобы вычислить результат, необходимо закрепить значения и вы-

числить все узлы.) В следующих далее разделах будет продемонстрировано то, как это все делается: как строить вычислительные графы и как их вычислять.

ПРИМЕЧАНИЕ. TensorFlow сначала строит вычислительный граф (в так называемой фазе конструирования), но при этом не вычисляет его автоматически. Данная библиотека разделяет эти два шага, благодаря чему можно вычислять граф несколько раз, например, с разными входами.

Тензоры

В библиотеке TensorFlow за основную единицу обрабатываемых данных принят — попробуйте угадать по названию библиотеки — тензор.

Тензор — это просто коллекция примитивных типов (таких как числа с плавающей точкой) в виде *n*-мерного массива. Вот несколько примеров тензоров (с относительными определениями на Python):

- 1 — скаляр;
- [1, 2, 3] — вектор;
- [[1, 2, 3], [4, 5, 6]] — матрица или двумерный массив.

Тензор имеет статический тип и динамические размерности. Его тип нельзя изменить во время вычисления, но размерности могут быть изменены динамически перед его вычислением. (В сущности, тензоры объявляются без указания некоторых размерностей, и TensorFlow логически выводит размерности из входных значений.) Обычно говорят о ранге тензора, который представляет собой не что иное, как число размерностей тензора (тогда как скаляру вменяется ранг 0). Таблица 1.1 поможеть понять, какими бывают ранги тензоров.

Таблица 1.1. *Примеры тензоров с рангами 0, 1, 2 и 3*

Ранг	Математическая сущность	Пример на Python
0	Скаляр (например, длина или вес)	`L=30`
1	Вектор (например, скорость объекта в двумерной плоскости)	`S=[10.2, 12.6]`
2	Матрица	`M=[[23.2, 44.2], [12.2, 55.6]]`
3	Трехмерная матрица (с тремя размерностями)	`C=[[[1], [2]], [[3], [4]], [[5], [6]]]`

Предположим, вы импортируете библиотеку TensorFlow с помощью инструкции `import tensorflow as tf`. Базовой сущностью, тензором, является класс `tf.tensor`. Класс `tf.tensor` имеет два свойства:

- тип данных (например, `float32`);
- форму (например, [2, 3], т. е. тензор с двумя строками и тремя столбцами).

Важным аспектом является то, что каждый элемент тензора всегда имеет один и тот же тип данных, в то время как форма не обязательно определена во время объявления. (Это будет яснее на практических примерах в последующих главах.) Основными типами тензоров (их существует больше), которые мы увидим в этой книге, являются:

- переменная `tf.Variable`;
- константа `tf.constant`;
- заполнитель `tf.placeholder`.

Значения с типом `tf.constant` и `the tf.placeholder` во время односеансового прогона (подробнее об этом позже) не могут мутировать. После того как они получают значение, они больше не изменяются. Например, значение с типом `tf.placeholder` может содержать набор данных, который вы хотите использовать для тренировки нейронной сети. После того как за ним закрепляется то или иное значение, в вычислительной фазе он не изменится. Значение с типом `tf.Variable` может содержать веса нейронных сетей. Они будут меняться во время тренировки, отыскивая свои оптимальные значения для конкретной задачи. Наконец, константа `tf.constant` никогда не изменится. В следующем разделе будет показано, как использовать эти три разных типа тензоров и какой аспект следует учитывать при разработке своих моделей.

Создание и выполнение вычислительного графа

Давайте приступим к использованию библиотеки TensorFlow для создания вычислительного графа.

ПРИМЕЧАНИЕ. Мы всегда отделяем фазу конструирования (когда мы определяем, что граф должен делать) от его оценивания (когда мы выполняем вычисления). TensorFlow следует той же философии: сначала вы конструируете граф, а затем его вычисляете.

Возьмем что-нибудь очень простое: сумму двух тензоров

$$x_1 + x_2,$$

которую можно рассчитать с помощью вычислительного графа (рис. 1.20).

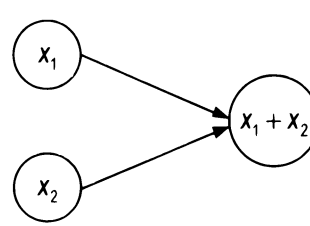


РИС. 1.20. Вычислительный граф для суммы двух тензоров

Вычислительный граф с типом *tf.constant*

Как уже отмечалось, сначала мы должны создать этот вычислительный граф с помощью TensorFlow. (Мы начинаем с фазы конструирования.) Давайте начнем использовать тензорный тип `tf.constant`. Нам нужны три узла: два для входных переменных и один для суммы. Это можно сделать с помощью следующих строк кода:

```
x1 = tf.constant(1)
x2 = tf.constant(2)
z = tf.add(x1, x2)
```

Приведенный выше фрагмент кода создает вычислительный граф на рис. 1.20 и в то же самое время сообщает библиотеке TensorFlow, что `x1` должен иметь значение 1 (в объявлении значение находится в скобках), а `x2` — значение 2. Теперь, для того чтобы вычислить этот исходный код, мы должны создать то, что в библиотеке TensorFlow именуется *сеансом* (в котором происходит фактическое вычисление), а затем, используя следующие ниже строки кода, мы можем запросить сеансовый класс выполнить наш граф:

```
sess = tf.Session()
print(sess.run(z))
```

В итоге вы получите вычисленный результат z, который, как и ожидалось, равен 3. Эта версия кода достаточно простая и не требует многого, но она не очень гибкая. `x1` и `x2` фиксированы и, например, не могут быть изменены во время вычисления.

ПРИМЕЧАНИЕ. В TensorFlow сначала необходимо создать вычислительный граф, затем создать сеанс и только потом выполнить свой граф. Эти три шага вычисления графа всегда должны выполняться последовательно.

Напомним, что можно также запросить TensorFlow вычислить только промежуточный шаг. Например, вы вполне можете вычислить `x1` (что в данном случае не очень интересно, но существует много случаев, когда это будет полезно, например, когда вы захотите вычислить свой граф и одновременно с этим точность и стоимостную функцию модели) следующим образом: `sess.run(x1)`.

Вы получите результат[1], как и ожидалось (ведь, вы ожидали этого, разве нет?). В конце не забудьте закрыть сеанс с помощью инструкции `sess.close()` для того, чтобы высвободить используемые ресурсы.

Вычислительный граф с типом *tf.Variable*

Тот же самый вычислительный граф (см. рис. 1.20) можно создать с переменными, но это потребует немного больше работы. Давайте воссоздадим наш вычислительный граф.

[1] Для того чтобы узнать о переменных больше, ознакомьтесь с официальной документацией по адресу: **www.tensorflow.org/versions/master/api_docs/python/tf/Variable**.

```
x1 = tf.Variable(1)
x2 = tf.Variable(2)
z = tf.add(x1,x2)
```

Мы хотим инициализировать переменные значениями 1 и 2, как и раньше. Проблема только в том, что при выполнении графа, как мы делали раньше, с использованием исходного кода

```
sess = tf.Session()
print(sess.run(z))
```

вы получите сообщение об ошибке. Данное сообщение очень длинное, но ближе к концу вы увидите следующее:

```
FailedPreconditionError (see above for traceback): Attempting to use
uninitialized value Variable
```

что в переводе будет гласить:

```
Ошибка невыполненного предусловия (см. выше в обратной трассировке): попытка
использовать неинициализированное значение с типом Variable)
```

Данная ошибка возникает потому, что TensorFlow не инициализирует переменные автоматически. Для этого вы могли бы применить следующий подход:

```
sess = tf.Session()
sess.run(x1.initializer)
sess.run(x2.initializer)
print(sess.run(z))
```

Теперь все работает без ошибок. Строка `sess.run(x1.initializer)` инициализирует переменную `x1` значением 1, а строка `sess.run(x2.initializer)` инициализирует переменную `x2` значением 2. Но это довольно громоздко. (Вы же не хотите набирать строку для каждой инициализируемой переменной.) Гораздо более оптимальный подход заключается в добавлении в вычислительный граф специального узла, предназначенного для инициализации всех переменных, определенных в графе, с помощью инструкции

```
init = tf.global_variables_initializer()
```

и затем снова в создании и запуске сеанса путем выполнения этого узла (`init`) перед вычислением `z`.

```
sess = tf.Session()
sess.run(init)
print(sess.run(z))
sess.close()
```

Это сработает и даст результат 3, как и ожидалось.

ПРИМЕЧАНИЕ. Во время работы с переменными в самом начале всегда добавляйте глобальный инициализатор (`tf.global_variables_initializer()`) и выполняйте узел в сеансе перед любым другим вычислением. Мы увидим, как это работает, на многих примерах из книги.

Вычислительный граф с типом *tf.placeholder*

Теперь давайте объявим `x1` и `x2` в качестве заполнителей.

```
x1 = tf.placeholder(tf.float32, 1)
x2 = tf.placeholder(tf.float32, 1)
```

Обратите внимание, что в объявлении никакого значения предоставлено не было. Мы должны будем закрепить значение за `x1` и `x2` во время вычисления. Это главное различие между заполнителями и двумя другими тензорными типами. Сумма, опять же, задается строкой кода

```
z = tf.add(x1,x2)
```

Заметим, что если вы попытаетесь взглянуть, что находится в `z`, используя, например, `print(z)`, то получите

```
Tensor("Add:0", shape=(1,), dtype=float32)
```

Почему такой странный результат? Во-первых, потому что мы не предоставили библиотеке TensorFlow значения для `x1` и `x2`, и во-вторых, потому что она еще не выполнила никаких вычислений. Напомним, что конструирование и вычисление графа — это два отдельных шага. Теперь, как и раньше, давайте создадим сеанс в TensorFlow.

```
sess = tf.Session()
```

Сейчас мы можем выполнить фактическое вычисление, но для этого сначала мы должны иметь какой-то способ закрепить значения за двумя входами `x1` и `x2`. Это можно осуществить с помощью словаря Python, который в качестве ключей содержит все имена заполнителей и назначает им значения. В этом примере мы закрепляем за `x1` значение 1, за `x2` — значение 2[2].

```
feed_dict={ x1: [1], x2: [2]}
```

Передача этого кода в сеанс TensorFlow выполняется следующей ниже инструкцией:

```
print(sess.run(z, feed_dict))
```

Вы, наконец, получите ожидаемый результат: 3. Обратите внимание, что библиотека TensorFlow довольно умна и может обрабатывать более сложные входы. Давайте переопределим наш заполнитель так, чтобы можно было использовать массивы с двумя элементами. (Для того чтобы было проще проследить работу примера, здесь мы приводим исходный код целиком.)

```
x1 = tf.placeholder(tf.float32, [2])
x2 = tf.placeholder(tf.float32, [2])
```

---

[2] Рекомендуется всегда обращаться к официальной документации по типам данных: **www.tensorflow.org/versions/master/api_docs/python/tf/placeholder**.

```
z = tf.add(x1,x2)
feed_dict={ x1: [1,5], x2: [1,1]}

sess = tf.Session()
sess.run(z, feed_dict)
```

На этот раз на выходе вы получите массив с двумя элементами.

```
array([ 2., 6.], dtype=float32)
```

Напомним, что суммирование выполняется поэлементно, т. е. `x1=[1,5]` и `x2=[1,1]` означает `z=x1+x2=[1,5]+[1,1]=[2,6]`.

Подводя итог, приведем несколько ориентиров по поводу того, когда и какой тензорный тип использовать.

- Использовать `tf.placeholder` для сущностей, которые не изменяются в каждой фазе оценивания. Обычно это входные значения или параметры, которые вы хотите оставлять фиксированными во время вычисления, но которые могут изменяться с каждым прогоном. (Вы увидите несколько примеров далее в книге.) Примеры включают входной набор данных, темп заучивания и т. д.
- Использовать `tf.Variable` для сущностей, которые будут изменяться во время вычисления, например, веса нейронных сетей, как вы увидите далее в книге.
- Использовать `tf.constant` для сущностей, которые никогда не изменятся, например, закрепить модельные значения, которые вы больше не хотите изменять.

На рис. 1.21 показан несколько более сложный пример: вычислительный граф для расчета выражения $x_1w_1 + x_2w_2$.

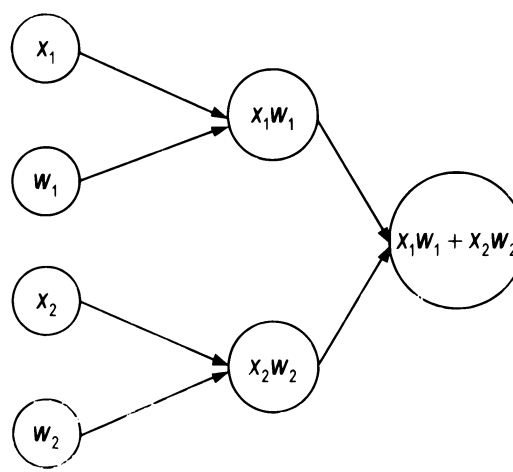


РИС. 1.21. Вычислительный граф для расчета выражения $x_1w_1 + x_2w_2$

В данном случае x_1, x_2, w_1 и w_2 определены как заполнители (они будут входами), содержащие скаляры (при определении заполнителей в качестве второго входного параметра следует всегда передавать размерности, в данном случае 1).

```
x1 = tf.placeholder(tf.float32, 1)
w1 = tf.placeholder(tf.float32, 1)
x2 = tf.placeholder(tf.float32, 1)
w2 = tf.placeholder(tf.float32, 1)

z1 = tf.multiply(x1,w1)
z2 = tf.multiply(x2,w2)
z3 = tf.add(z1,z2)
```

Выполнить расчеты означает просто (как и прежде) определить словарь с входными значениями, создать сеанс и затем его выполнить.

```
feed_dict={x1: [1], w1:[2], x2:[3], w2:[4]}
sess = tf.Session()
sess.run(z3, feed_dict)
```

Как и ожидалось, вы получите следующий результат:

```
array([ 14.], dtype=float32)
```

Здесь вычисляется выражение $1 \cdot 2 + 3 \cdot 4 = 2 + 12 = 14$ (т. к. на предыдущем шаге мы подали значения 1, 2, 3 и 4 в словарь `feed_dict`). В *главе 2* мы нарисуем вычислительный граф для одного-единственного нейрона и применим то, что мы узнали в этой главе, к очень практическому случаю. С помощью такого графа мы сможем выполнять линейную и логистическую регрессию на реальном наборе данных. Как всегда, не забудьте закрыть сеанс инструкцией `sess.close()`, когда закончите.

ПРИМЕЧАНИЕ. В TensorFlow бывают ситуации, когда один и тот же фрагмент кода выполняется несколько раз, и в итоге вы можете получить вычислительный граф с несколькими копиями одного и того же узла. Очень распространенный способ избежать такой проблемы — выполнить инструкцию `tf.reset_default_graph()` перед исходным кодом, который выполняет конструирование графа. Обратите внимание, что если вы надлежаще отделяете свой конструирующий код от вычислительного кода, то сможете избежать таких проблем. Мы увидим, как это работает, позже на многочисленных примерах книги.

Различия между *run* и *eval*

Если вы заглянете в блоги и книги, то найдете два способа вычисления вычислительного графа в TensorFlow. До сего момента мы использовали `sess.run()`, в котором метод `run` требует в качестве аргумента имя узла, подлежащего вычислению. Мы выбрали этот способ, потому что он имеет одно хорошее преимущество. Для его понимания давайте рассмотрим следующий ниже фрагмент кода (тот же, который вы видели ранее):

```
x1 = tf.constant(1)
x2 = tf.constant(2)
z = tf.add(x1, x2)
```

```
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
sess.run(z)
```

Он дает вам только один вычисленный узел `z`, но вы также можете вычислять несколько узлов одновременно, используя следующий код:

```
sess.run([x1,x2,z])
```

В результате получится

```
[1, 2, 3]
```

И это очень полезно, как станет ясно в следующем разделе, посвященном жизненному циклу узла. Кроме того, одновременное вычисление нескольких узлов сделает код короче и более удобочитаемым.

Второй способ вычисления узла в графе предусматривает использование вызова метода `eval()`.

Строка кода

```
z.eval(session=sess)
```

вычислит `z`. Но на этот раз вы должны явно сообщить библиотеке TensorFlow, какой сеанс хотите использовать (у вас может быть определено их несколько). Это не очень практично, и рекомендуется использовать метод `run()`, что позволяет получать несколько результатов одновременно (например, стоимостную функцию, точность и оценку F1). Еще одна причина, почему следует предпочесть именно первый способ, связана с производительностью, как описано в следующем разделе.

Зависимости между узлами

Как уже отмечалось ранее, библиотека TensorFlow вычисляет граф в топологическом порядке. Это означает, что, когда вы поручаете ей вычислить узел, она автоматически определяет все узлы, которые необходимы для вычисления того, что вы поручаете, и сначала вычисляет их. Проблема в том, что TensorFlow может вычислять некоторые узлы многократно. К примеру, рассмотрим следующий фрагмент кода:

```
c = tf.constant(5)
x = c + 1
y = x + 1
z = x + 2
sess = tf.Session()
print(sess.run(y))
print(sess.run(z))
sess.close()
```

Данный фрагмент кода построит и вычислит вычислительный граф, как на рис. 1.22.

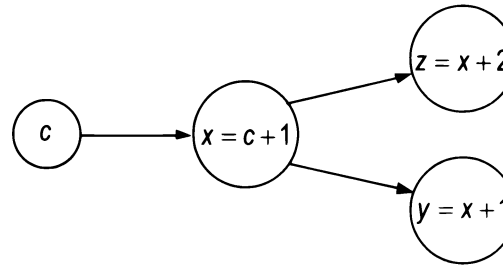


РИС. 1.22. Вычислительный граф, который строится фрагментом кода в начале этого раздела

Как видите, и z, и y зависят от x. Проблема с этим фрагментом кода, как мы уже отметили, состоит в том, что библиотека TensorFlow не будет использовать результат предыдущего вычисления c и x. Это означает, что она будет вычислять узел для x один раз во время вычисления z и снова во время вычисления y. В этом случае, например, инструкция `yy, zz = sess.run([y,z])` вычислит y и z в одном прогоне и вычислит x только один раз.

Советы по созданию и закрытию сеанса

До этого места было показано, как создавать сеанс с помощью шаблона:

```
sess = tf.Session()
# Код, который что-то делает
```

В конце следует всегда закрывать сеанс, высвобождая используемые ресурсы. Синтаксис закрытия довольно прост:

```
sess.close()
```

Имейте в виду, что в момент закрытия сеанса вы не сможете вычислять что-либо еще. Вы должны создать новый сеанс и снова выполнить вычисление. В среде Jupyter этот метод имеет то преимущество, что позволяет подразделять вычисляющий код на несколько ячеек, а затем закрывать сеанс в самом конце. Но полезно знать, что существует несколько более компактный способ открытия и использования сеанса, используя следующий ниже шаблон:

```
with tf.Session() as sess:
# Код, который что-то делает
```

Например, фрагмент кода

```
sess = tf.Session()
print(sess.run(y))
print(sess.run(z))
sess.close()
```

из предыдущего раздела можно было бы записать как

```
with tf.Session() as sess:
    print(sess.run(y))
    print(sess.run(z))
```

В этом случае сеанс будет автоматически закрыт в конце блока `with`. Применение этого метода упрощает использование метода `eval()`. Например, фрагмент кода

```
sess = tf.Session()
print(z.eval(session=sess))
sess.close()
```

с блоком `with` будет выглядеть так:

```
with tf.Session() as sess:
    print(z.eval())
```

В некоторых случаях предпочтение отдается явному объявлению сеанса.

Например, довольно часто пишут функцию, которая выполняет фактическое вычисление графа и возвращает сеанс, благодаря чему дополнительное вычисление (например, точности или аналогичных метрических показателей) может быть выполнено после завершения базового процесса тренировки. В этом случае нельзя использовать вторую версию, т. к. она закроет сеанс сразу после завершения вычисления, что сделает невозможным проведение дополнительных вычислений с результатами сеанса.

ПРИМЕЧАНИЕ. Если вы работаете в интерактивной среде блокнота Jupyter и хотите подразделить вычислительный код на несколько ячеек блокнота, то проще объявить сеанс как `sess = tf.Session()`, выполнить необходимые вычисления, а затем, в конце, закрыть его. Благодаря этому вы можете перемежать вычисления, графики и текст. В случае если вы пишете неинтерактивный код, то иногда предпочтительнее (и менее подвержено ошибкам) использовать вторую версию с целью обеспечения того, чтобы сеанс в конце был закрыт. Кроме того, при использовании второго способа нет необходимости указывать сеанс при применении метода `eval()`.

Материал, описанный в этой главе, должен дать вам все, что нужно для построения своих нейронных сетей с помощью библиотеки TensorFlow. То, что было здесь представлено, никоим образом не является полным или исчерпывающим. Вам действительно следует взять небольшую паузу, перейти на официальный веб-сайт TensorFlow и поизучать учебные руководства и другие материалы.

ПРИМЕЧАНИЕ. В этой книге используется "ленивый" подход к программированию. Он означает, что объясняется только то, что необходимо донести до вашего понимания, не более того. Это вызвано стремлением побудить вас сосредоточиться на учебных целях каждой главы, и мне бы не хотелось, чтобы вы отвлекались на сложности, которые кроются позади методов или программных функций. Разобравшись в том, что я пытаюсь объяснить, вам следует потратить некоторое время и углубленно заняться методами и библиотеками с привлечением официальной документации.

ГЛАВА 2

Один-единственный нейрон

В этой главе будет показано, что такое нейрон и каковы его компоненты. В ней будут даны разъяснения необходимых математических обозначений и рассмотрены многие активационные функции, которые сегодня используются в нейронных сетях. Подробно будет рассмотрена оптимизация на основе градиентного спуска, введено понятие темпа заучивания и объяснены его особенности. Для того чтобы чтение было чуть-чуть увлекательнее, мы применим один-единственный нейрон для выполнения линейной и логистической регрессии на реальных наборах данных. Затем будет показано и разъяснено, как реализовывать эти два алгоритма с помощью TensorFlow.

Для того чтобы глава была концентрированной и учебный эффект высоким, несколько вещей были намеренно исключены. Например, мы не будем разбивать набор данных на тренировочную и тестовую части. Мы будем использовать все данные целиком. Использование обеих частей вынудило бы нас провести надлежащий анализ, отвлекло бы от главной цели этой главы и сделало бы ее слишком длинной. Далее в книге будет проведен надлежащий анализ последствий использования нескольких наборов данных и показано, как это делать правильно, в особенности в контексте глубокого обучения. Эта тема требует отдельной главы.

С помощью глубокого обучения вы можете делать замечательные, удивительные и забавные вещи. Давайте же начнем получать интеллектуальное удовольствие!

Структура нейрона

Глубокое (само)обучение основывается на крупных и сложных сетях, состоящих из большого числа простых вычислительных единиц. Компании на переднем крае исследований имеют дело с сетями со 160 млрд параметров [1]. Если говорить о более широком контексте, то это число вдвое меньше количества звезд в нашей галактике или в 1,5 раза больше числа когда-либо живших людей.

На базовом уровне нейронные сети представляют собой большое множество поразному взаимосвязанных единиц, каждая из которых выполняет определенное (и обычно относительно простое) вычисление. Они напоминают игрушки LEGO, с помощью которых можно строить очень сложные предметы, используя очень простые и базовые блоки. Нейронные сети выглядят похоже. Используя относительно простые вычислительные единицы, вы можете строить очень сложные системы. Мы можем варьировать базовые единицы, изменяя то, как они вычисляют результат, как они связаны друг с другом, как они используют входные значения и т. д. Все эти аспекты определяют так называемую сетевую архитектуру. Изменив архитектуру, мы изменим то, как сеть учится, насколько точными будут предсказания и т. д.

Эти базовые единицы именуются благодаря биологической параллели с мозгом [2], нейронами. Каждый нейрон делает очень простую вещь: берет определенное число входов (вещественных чисел) и вычисляет выход (тоже вещественное число). В этой книге наши входы будут обозначаться как $x_i \in \mathbb{R}$ (вещественные числа) с $i = 1, 2, ..., n_x$, где $i \in \mathbb{N}$ — это целое число и n_x — число входных атрибутов (часто именуемых признаками). В качестве примера входных признаков можно представить возраст и массу тела человека (и значит, у нас будет $n_x = 2$). x_1 может быть возрастом, а x_2 — массой. В реальной жизни число признаков легко может вырастать до больших значений. В наборе данных, который мы намерены использовать для нашего примера логистической регрессии далее в этой главе, у нас будет $n_x = 784$.

Существует несколько видов нейронов. И все они были тщательно изучены. В этой книге мы сосредоточимся на наиболее часто используемом. Нейрон, который нас интересует, просто берет функцию и применяет ее к линейной комбинации всех входов. В более математической форме, с учетом n_x, вещественных параметров $w_i \in \mathbb{R}$ (с $i = 1, 2, ..., n_x$) и константы $b \in \mathbb{R}$ (обычно именуемой смещением) нейрон сначала вычислит то, что обычно указывается в литературе и в книгах как z.

$$z = w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b .$$

Затем он применит функцию f к z, давая выход $\hat{y}$.

$$\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b\right).$$

ПРИМЕЧАНИЕ. На практике в основном используется следующая номенклатура: w_i обозначает веса, b — смещение, x_i — входные признаки и f — активационную функцию.

Отдавая дань биологической параллели, функцию f называют *активационной функцией нейрона* (иногда также *передаточной функцией*), о которой подробно пойдет речь в следующих разделах.

Давайте снова подытожим вычислительные шаги нейрона.

1. Линейно скомбинировать все входы x_i, вычислив

$$z = w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b .$$

2. Применить f к z, произведя выход

$$\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b\right).$$

Как известно, в *главе 1* мы обсуждали вычислительные графы. На рис. 2.1 представлен граф описанного выше нейрона.

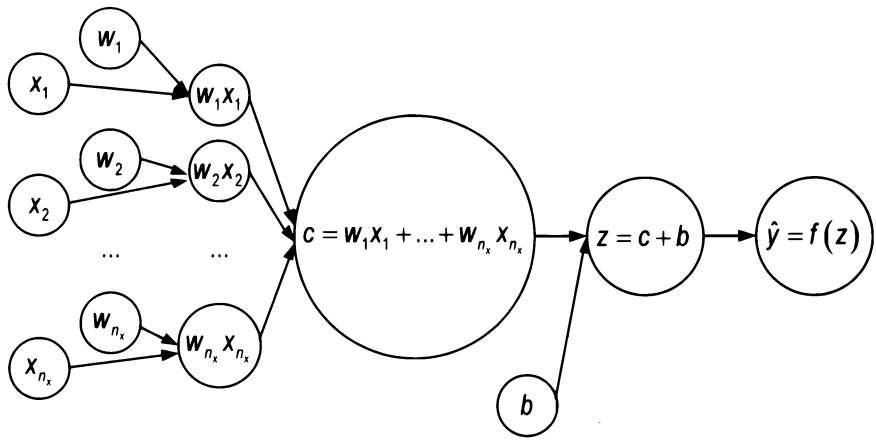


РИС. 2.1. Вычислительный граф для нейрона, описанного в тексте

Это совсем не та форма представления, которую обычно можно найти в блогах, книгах и учебных пособиях. Она довольно сложна и не очень практична в использовании, в особенности если вы хотите рисовать сети с большим числом нейронов. Вообще в литературе можно найти многочисленные формы представления нейронов. В этой книге мы будем использовать ту, что показана на рис. 2.2, потому что она широко используется и понятна.

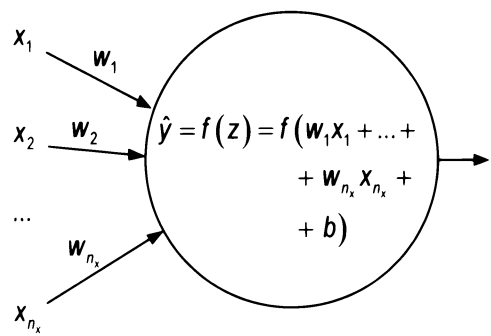


РИС. 2.2. Форма представления нейронов, используемая на практике чаще всего

Рисунок 2.2 следует интерпретировать следующим образом.

- Входы не размещаются в круге. Это делается просто для того, чтобы отличать их от узлов, выполняющих фактические вычисления.
- Имена весов записываются вдоль стрелки. Это означает, что перед подачей входов в центральный круг (или узел) вход сначала будет умножен на относительный вес, как помечено на стрелке. Первый вход x_1 будет умножен на w_1, x_2 на w_2 и т. д.
- Центральный круг (или узел) будет последовательно выполнять несколько вычислений. Сначала он просуммирует входы ($x_i w_i$ для $i = 1, 2, ..., n_x$), затем приплюсует к результату смещение b и, наконец, применит к результирующему значению активационную функцию.

Все нейроны, с которыми мы будем работать в этой книге, будут иметь именно такую структуру. Очень часто используется еще более простая форма представления (рис. 2.3). В таком случае, если не указано иное, подразумевается, что выходом является

$$\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b\right).$$

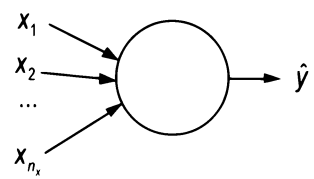


РИС. 2.3. Данная форма представления является упрощенной версией рис. 2.2. Если не указано иное, то обычно подразумевается, что выходом является $\hat{y} = f(z) = f\left(w_1 x_1 + w_2 x_2 + ... + w_{n_x} x_{n_x} + b\right)$. В форме представления нейрона веса часто не указываются явно

Матричное обозначение

Во время работы с большими наборами данных число признаков является большим (n_x будет большим), поэтому для признаков и весов лучше всего использовать векторное обозначение:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_{n_x} \end{bmatrix}$$

где мы обозначили вектор жирным шрифтом $\mathbf{x}$. То же самое обозначение используеся и для весов:

$$\mathbf{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_{n_x} \end{bmatrix}.$$

С целью обеспечения согласованности с формулами, которые мы будем использовать в дальнейшем, перемножая **x** и **w**, мы будем использовать обозначение матричного умножения и, следовательно, напишем:

$$\mathbf{w}^T\mathbf{x} = \begin{bmatrix} w_1 & \dots & w_{n_x} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_{n_x} \end{bmatrix} = w_1 x_1 + w_2 x_2 + \dots + w_{n_x} x_{n_x},$$

где $\mathbf{w}^T$ обозначает транспонированную матрицу **w**. Тогда z можно записать с помощью векторного обозначения как

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

и выход $\hat{y}$ нейрона как

$$\hat{y} = f(z) = f\left(\mathbf{w}^T\mathbf{x} + b\right). \qquad (*)$$

Теперь подытожим разные компоненты, которые задают нейрон, и обозначения, которые мы будем использовать в этой книге:

- $\hat{y}$ — выход нейрона;
- $f(z)$ — активационная функция (или передаточная функция), применяемая к z;
- **w** — веса (вектор с n_x компонентами);
- b — смещение.

Совет по реализации на языке Python: циклы и NumPy

Вычисление, которое мы описали в уравнении (*), может быть выполнено на Python с использованием стандартных списков и циклов, но они, как правило, становятся очень медленными по мере роста числа переменных и наблюдений. Хорошее эмпирическое правило состоит в том, чтобы избегать циклов всегда, когда это возможно, и как можно чаще использовать методы библиотеки NumPy (или TensorFlow, как мы увидим позже).

Можно без труда увидеть, насколько быстрым является вычисление в NumPy (и насколько медленными являются циклы). Начнем с создания двух стандартных списков случайных чисел на Python со 107 элементами в каждом.

```
import random
lst1 = random.sample(range(1, 10**8), 10**7)
lst2 = random.sample(range(1, 10**8), 10**7)
```

Фактические значения не имеют никакого отношения к нашим целям. Нас просто интересует, насколько быстро Python может перемножить два списка поэлементно. Сообщаемые времена были измерены на ноутбуке Microsoft Surface 2017 года и будут сильно различаться в зависимости от оборудования, на котором этот исход-

ный код выполняется. Нас интересуют не абсолютные значения, а только то, насколько NumPy быстрее по сравнению со стандартными циклами Python. Для хронометрирования Python-кода в блокноте Jupyter мы можем применить так называемую "волшебную команду". В блокноте Jupyter эти команды обычно начинаются с символов %% или %. Неплохая идея — проверить официальную документацию по таким командам, доступную по адресу **http://ipython.readthedocs.io/en/stable/interactive/magics.html**, которая поможет лучше разобраться в том, как они работают.

Вернемся к нашему тесту. Давайте измерим продолжительность выполнения поэлементного перемножения двух списков с помощью стандартных циклов. Используя фрагмент кода

```
%%timeit
ab = [lst1[i]*lst2[i] for i in range(len(lst1))]
```

мы получим следующий результат (отметим, что на вашем компьютере этот результат будет отличаться):

```
2.06 s ± 326 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

(В переводе: `2.06 с ± 326 мс на цикл (среднее значение ± стандартное отклонение из 7 прогонов, 1 цикл каждый).`)

На семь прогонов этот исходный код потребовал в среднем около двух секунд. Теперь давайте попробуем сделать то же самое перемножение, но на этот раз используя библиотеку NumPy, где мы сначала конвертировали два списка в массивы NumPy, с помощью следующего ниже фрагмента кода:

```
import numpy as np
list1_np = np.array(lst1)
list2_np = np.array(lst2)

%%timeit
Out2 = np.multiply(list1_np, list2_np)
```

На этот раз мы получим такой результат:

```
20.8 ms ± 2.5 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

(В переводе: `20.8 мс ± 2.5 мс на цикл (среднее значение ± стандартное отклонение из 7 прогонов, 10 циклов каждый).`)

Исходный код с использованием NumPy потребовал всего 21 мс или, другими словами, был примерно в 100 раз быстрее, чем исходный код со стандартными циклами. Библиотека NumPy работает быстрее по двум причинам: базовые процедуры написаны на C, и с целью ускорить вычисления на больших объемах данных она максимально использует векторизованный код.

ПРИМЕЧАНИЕ. Векторизованный код относится к операциям, которые выполняются над многочисленными компонентами вектора (или матрицы) одновременно (в одной инструкции). Передача матриц функциям NumPy является

хорошим примером векторизованного кода. NumPy будет выполнять операции над большими фрагментами данных одновременно, показывая гораздо более высокую производительность по сравнению со стандартными циклами Python, которым приходится работать с одним элементом за раз. Обратите внимание, что хорошая производительность, которую показывает NumPy, также обусловлена тем, что базовые подпрограммы написаны на C.

Во время тренировки глубоко обучающихся моделей вы обнаружите, что выполняете такие операции снова и снова, и, следовательно, такой прирост скорости составит разницу между моделью, которую можно натренировать, и моделью, которая никогда не даст вам результата.

Активационные функции

В нашем распоряжении существует целый ряд активационных функций, которые изменяют выход нейрона. Напомним, что активационная функция — это просто математическая функция, которая преобразует z в выход $\hat{y}$. Давайте посмотрим, какие из них используются наиболее часто.

Активационная функция тождественного отображения

Это самая элементарная функция из всех, которую вы можете использовать. Обычно она обозначается как $I(z)$. Она просто возвращает входное значение без изменений. Математически мы записываем это так:

$$f(z) = I(z) = z\,.$$

Эта простая функция пригодится, когда позже в этой главе будет обсуждаться линейная регрессия с одним-единственным нейроном. На рис. 2.4 показано, как эта функция выглядит.

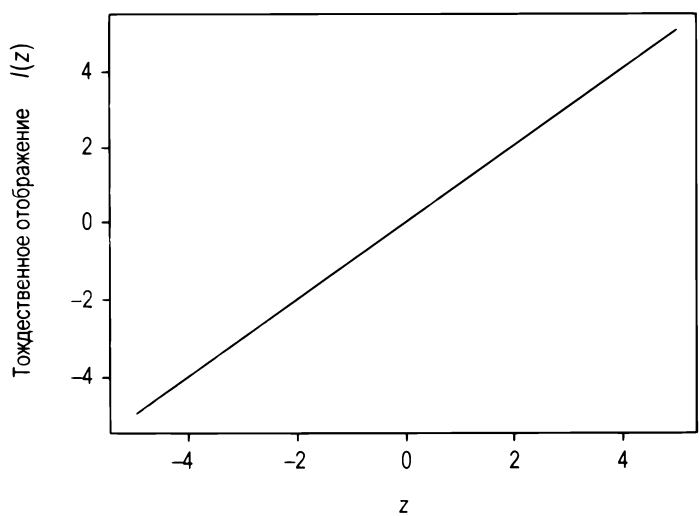


РИС. 2.4. Активационная функция в виде тождественного отображения

Тождественное отображение реализуется на Python исключительно просто:

```
def identity(z):
    return z
```

Активационная функция сигмоидальная

Данная функция используется очень часто и производит значения исключительно между 0 и 1. Обычно она обозначается как $\sigma(z)$.

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

Она особенно часто используется в моделях, в которых на выходе нам требуется предсказать вероятность (напомним, что вероятность может принимать значения исключительно между 0 и 1). Вы увидите ее форму на рис. 2.5. Обратите внимание, что если z является достаточно большим, то в Python может случиться так, что из-за ошибок округления функция вернет ровно 0 или 1 (в зависимости от знака z). В классификационных задачах мы очень часто будем вычислять $\log\sigma(z)$ или $\log(1-\sigma(z))$, и, следовательно, этот факт может стать источником ошибок на Python, потому что он попытается вычислить $\log 0$, который не определен. Например, при вычислении стоимостной функции вы можете начать получать `nan`[1] (подробнее об этом позже). Мы увидим практический пример этого явления позже в этой главе.

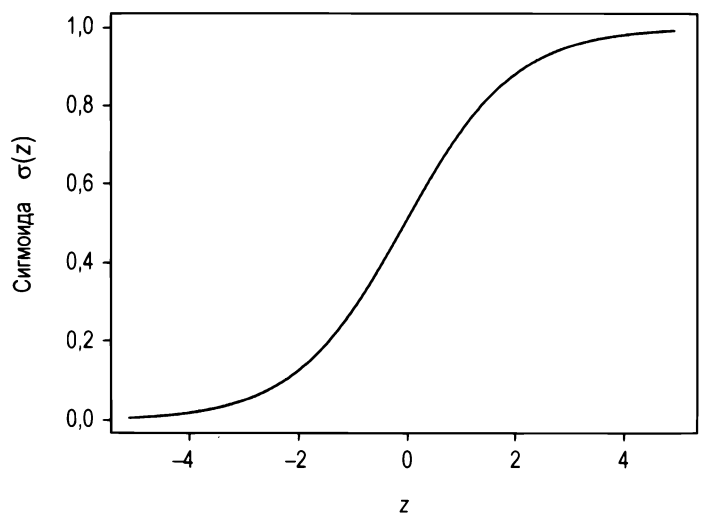


РИС. 2.5. Сигмоидальная активационная функция представляет собой s-образную функцию, которая изменяется в интервале между 0 и 1

[1] Аббревиатура nan означает not a number, т. е. не число. — *Прим. пер.*

ПРИМЕЧАНИЕ. Несмотря на то, что сигмоида $\sigma(z)$ никогда не должна равняться 0 или 1, во время программирования на Python реальность может оказаться совсем другой. Из-за очень большого z (положительного или отрицательного) Python может округлить результаты ровно до 0 или 1, а это может привести к ошибкам при расчете стоимостной функции (ее подробное объяснение и практический пример будут даны позже в этой главе) для классификации, потому что нам нужно будет вычислять $\log\sigma(z)$ и $\log(1-\sigma(z))$, и, следовательно, Python попытается вычислить логарифм $\log 0$, который не определен. Это может произойти, например, если мы неправильно выполним нормализацию входных данных или инициализацию весов. На данный момент важно помнить об одном: хотя математически все будет выглядеть, будто всё под контролем, во время программирования реальность может быть гораздо сложнее. И данный факт неплохо бы учитывать во время отладки моделей, которые, например, дают `nan` в качестве результата стоимостной функции.

Поведение с z можно увидеть на рис. 2.5. Его расчет может быть записан в следующей форме с использованием функций NumPy:

```
s = np.divide(1.0, np.add(1.0, np.exp(-z)))
```

ПРИМЕЧАНИЕ. Очень полезно знать, что если имеются два массива NumPy, A и B, то A/B эквивалентно `np.divide(A,B)`, $A+B$ эквивалентно `np.add(A,B)`, $A-B$ эквивалентно `np.subtract(A,B)` и $A \cdot B$ эквивалентно `np.multiply(A,B)`. Если вы знакомы с объектно-ориентированным программированием, то мы говорим, что в NumPy элементарные операции, такие как `/`, `*`, `+` и `-`, перегружены. Также обратите внимание на то, что эти четыре элементарные операции в NumPy действуют поэлементно.

Сигмоидальная функция может быть написана в удобочитаемой (по крайней мере, для людей) форме следующим образом:

```
def sigmoid(z):
    s = 1.0 / (1.0 + np.exp(-z))
    return s
```

Как говорилось ранее, выражение `1.0 + np.exp(-z)` эквивалентно выражению `np.add(1.0, np.exp(-z))`, а выражение `1.0/(np.add(1.0, np.exp(-z)))` эквивалентно выражению `np.divide(1.0, np.add(1.0, np.exp(-z)))`. Хотелось бы обратить внимание еще на один пункт в формуле. `np.exp(-z)` будет иметь размерность `z` (обычно вектор, длина которого будет равна числу наблюдений), тогда как `1.0` является скаляром (одномерной сущностью). Каким образом Python их суммирует? То, что происходит, называется *транслированием*[2]. Python, с учетом определенных ограни-

[2] Более подробное объяснение того, как NumPy использует механизм транслирования, можно найти в официальной документации по адресу:
https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html.

чений, будет "транслировать" меньший массив (в данном случае `1.0`) на больший, благодаря чему оба в конце будут иметь одинаковые размерности. В этом случае `1.0` становится массивом той же размерности, что и `z`, заполненным `1.0`. Это понятие важно понимать, т. к. оно очень полезно. Например, вам не нужно преобразовывать числа в массивы. Python позаботится об этом за вас. Правила работы механизма транслирования в других случаях достаточно сложны и выходят за рамки данной книги. Однако важно знать, что Python делает что-то на заднем плане.

Активационная функция *tanh* (гиперболический тангенс)

Гиперболический тангенс представляет собой s-образную кривую, которая изменяется в интервале между −1 и 1.

$$f(z) = \tanh(z).$$

На рис. 2.6 показана ее форма. На языке Python ее можно легко реализовать следующим образом:

```
def tanh(z):
    return np.tanh(z)
```

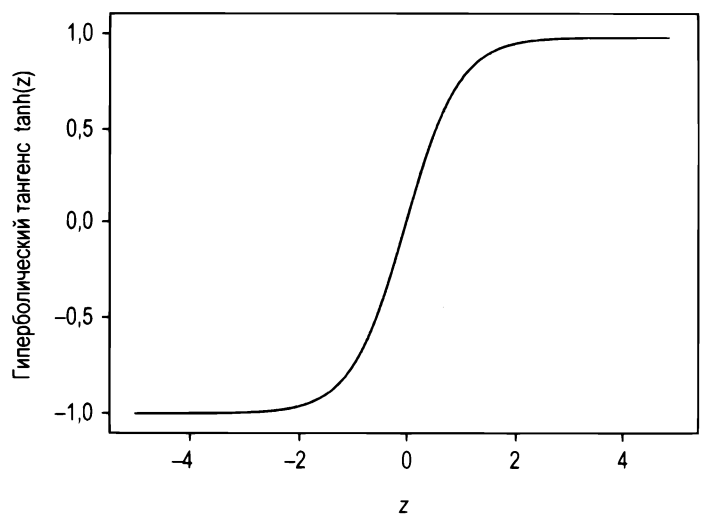


РИС. 2.6. Функция tanh (или гиперболическая функция) представляет собой s-образную кривую, которая проходит от −1 до 1

Активационная функция ReLU (выпрямленного линейного элемента)

Функция ReLU[3] (рис. 2.7) имеет следующую формулу:

$$f(z) = \max(0, z).$$

[3] Функция ReLU (rectified linear unit) также носит название пилообразной функции и аналогична полупериодному выпрямителю в электротехнике. — *Прим. пер.*

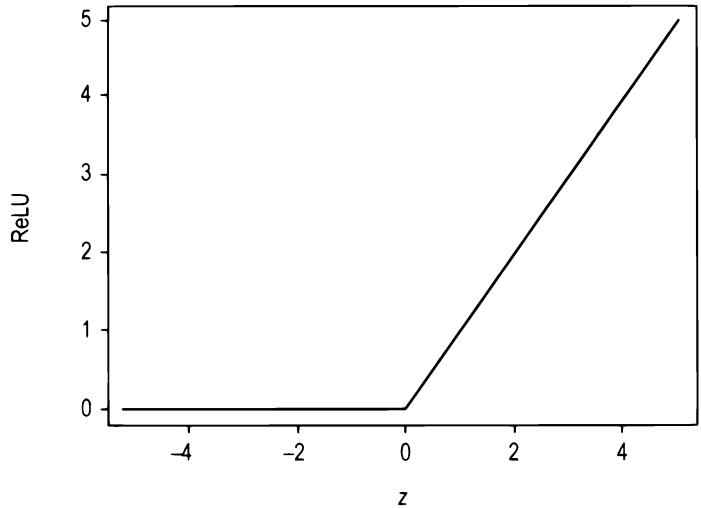


РИС. 2.7. Функция ReLU

Полезно потратить несколько минут на изучение того, как грамотно реализовывать функцию ReLU на языке Python. Обратите внимание: когда мы начнем использовать библиотеку TensorFlow, она уже будет реализована за нас, но очень поучительно наблюдать, как разные ее Python-реализации могут повлиять на реализацию сложных глубоко обучающихся моделей.

На Python функцию ReLU можно реализовать несколькими способами. Ниже перечислены четыре разных способа. (Прежде чем продолжить, постарайтесь разобраться в том, почему они работают.)

```
np.maximum(x, 0, x)
np.maximum(x, 0)
x * (x > 0)
(abs(x) + x) / 2
```

Эти четыре способа реализации имеют очень разные скорости исполнения. Давайте создадим массив NumPy со 108 элементами, как показано ниже:

```
x = np.random.random(10**8)
```

Теперь измерим время, необходимое четырем разным версиям функции ReLU при ее применении. Выполним следующий ниже фрагмент кода:

```
x = np.random.random(10**8)
print("Метод 1:")
%timeit -n10 np.maximum(x, 0, x)

print("Метод 2:")
%timeit -n10 np.maximum(x, 0)

print("Метод 3:")
%timeit -n10 x * (x > 0)
```

```
print("Метод 4:")
%timeit -n10 (abs(x) + x) / 2
```

Результаты будут следующими:

```
Метод 1:
2.66 ms ± 500 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Метод 2:
6.35 ms ± 836 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Метод 3:
4.37 ms ± 780 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
Метод 4:
8.33 ms ± 784 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Разница ошеломляющая. Метод 1 в четыре раза быстрее метода 4. Библиотека NumPy сильно оптимизирована в силу большого числа подпрограмм, написанных на C. Но знание того, как эффективно программировать, все же имеет решающее значение и может иметь большое влияние. Почему `np.maximum(x, 0, x)` быстрее, чем `np.maximum(x, 0)`? Первая версия обновляет `x` прямо на месте, не создавая новый массив. Это может сэкономить много времени, в особенности если массивы являются большими. Если вы не хотите (или не можете) обновлять входной вектор прямо на месте, то все равно можете использовать версию `np.maximum(x, 0)`.

Реализация данной функции может выглядеть следующим образом:

```
def relu(z):
    return np.maximum(z, 0)
```

ПРИМЕЧАНИЕ. Напомним, что во время оптимизации исходного кода даже небольшие изменения могут иметь огромную разницу. В глубоко обучающихся программах один и тот же фрагмент кода будет повторяться миллионы и миллиарды раз, и поэтому даже небольшое улучшение будет иметь огромное влияние в долгосрочной перспективе. Временны́е затраты на оптимизацию исходного кода представляют собой необходимый шаг, который обязательно окупится.

Активационная функция ReLU с утечкой

Активационная функция ReLU с утечкой (также именуемая параметрическим выпрямленным линейным элементом) задается следующей формулой:

$$f(z) = \begin{cases} \alpha z, \text{если } z < 0; \\ z, \quad \text{если } z \geq 0, \end{cases}$$

где параметр α обычно имеет порядок 0,01. На рис. 2.8 показан пример для $\alpha = 0,05$. Это значение было выбрано для того, чтобы сделать разницу между $x > 0$ и $x < 0$ отчетливее. Обычно для α используются меньшие значения, но для того чтобы отыскать лучшее значение, требуется провести тестирование модели.

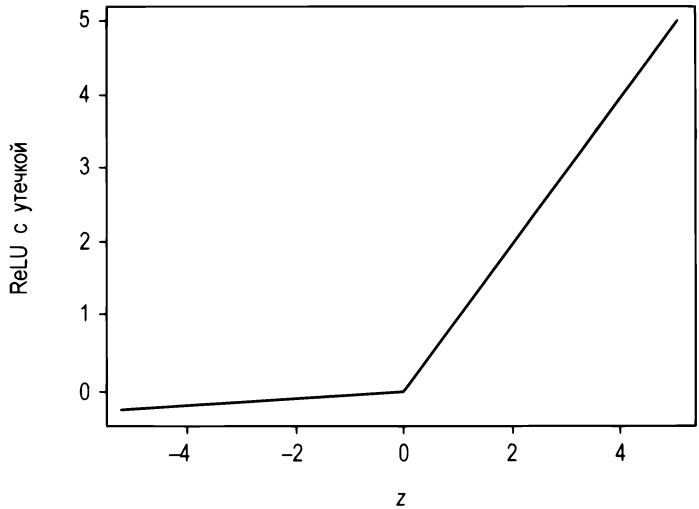


РИС. 2.8. Активационная функция ReLU с утечкой с $\alpha = 0{,}05$

Например, на Python ее можно реализовать, если функция `relu(z)` уже определена, в виде

```
def lrelu(z, alpha):
    return relu(z) - alpha * relu(-z)
```

Активационная функция Swish

Недавно Рамачандран (Ramachandran), Зопф (Zopf) и Ле (Le) в Google Brain [4] провели исследование новой активационной функции, именуемой Swish[4], которая демонстрирует большие перспективы в сфере глубокого обучения. Она определяется как

$$f(z) = z\sigma(\beta z),$$

где β — это заучиваемый параметр[5]. На рис. 2.9 показано, как эта активационная функция выглядит для трех значений параметра β: 0,1; 0,5 и 10,0. Исследования данной команды разработчиков показали, что простая замена функций активации ReLU на Swish повышает точность классифицирования на данных ImageNet на

---

[4] См. статью с каламбурным названием "Is it Time to Swish?" (Не пора ли вилять хвостом?) о сравнении активации Swish с десятью другими активациями по адресу **http://aclweb.org/anthology/D18-1472**. — *Прим. пер.*

[5] Swish — это гладкая, немонотонная активационная функция. Ее имя может быть связано с английским выражением "swish the tail" — вилять хвостом, по характерному ее поведению после нулевой точки. Иногда она имеет форму $f(x) = x \cdot \text{sigmoid}(x)$. Эксперименты показывают, что активация Swish, как правило, работает лучше, чем ReLU на более глубоких моделях в таких областях, как классификация изображений и машинный перевод. См. **https://medium.com/@neuralnets/swish-activation-function-by-google-53e1ea86f820**. — *Прим. пер.*

0,9%. В современном мире глубокого обучения это довольно много. Более подробную информацию о базе данных ImageNet можно найти по адресу **www.image-net.org/**.

ImageNet — это большая база данных изображений, которая часто используется для тестирования новых сетевых архитектур или алгоритмов, таких как сети с другой активационной функцией.

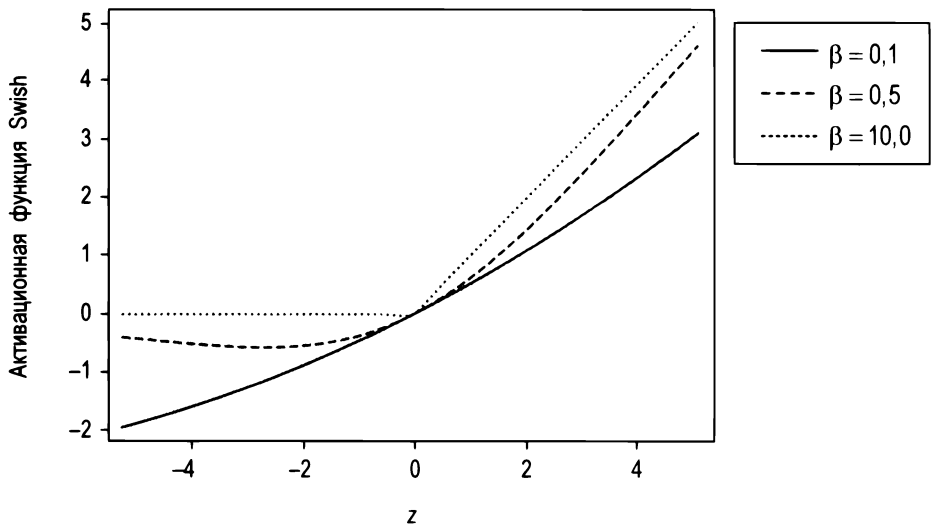


РИС. 2.9. Активационная функция Swish для трех разных значений параметра β

Другие активационные функции

Существует целый ряд других активационных функций, но они используются редко. Для справки далее приведено несколько дополнительных функций. Данный список отнюдь не является исчерпывающим, и его единственная цель — дать вам представление о разнообразии активационных функций, которые могут применяться во время разработки нейронных сетей.

- Активационная функция ArcTan (арктангенсная):

$$f(z) = \tan^{-1} z\,.$$

- Активационная функция ELU (экспоненциальный линейный элемент):

$$f(z) = \begin{cases} \alpha\left(e^{z} - 1\right), & \text{если } z < 0; \\ z, & \text{если } z \geq 0. \end{cases}$$

- Активационная функция Softplus:

$$f(z) = \ln\left(1 + e^{z}\right).$$

ПРИМЕЧАНИЕ. На практике почти всегда используются только две активационные функции: сигмоида и ReLU (ReLU, наверное, даже чаще). С обеими можно достичь хороших результатов, и, учитывая достаточно сложную сетевую архитектуру, обе способны аппроксимировать любую нелинейную функцию [5, 6]. Напомним, что во время использования библиотеки TensorFlow вам не придется реализовывать функции самостоятельно. Библиотека TensorFlow предложит в ваше распоряжение эффективную реализацию. Но важно знать, как ведет себя каждая активационная функция, и разбираться в том, когда и какую из них использовать.

Стоимостная функция и градиентный спуск: причуды темпа заучивания

Теперь, когда вы ясно понимаете, что такое нейрон, следует разъяснить, что имеется в виду, когда говорят, что нейрон (и вообще нейронная сеть) учится. Это позволит нам ввести такие понятия, как гиперпараметры и темп заучивания. Почти во всех нейросетевых задачах под заучиванием подразумевается не что иное, как отыскание весов (напомним, что нейронная сеть состоит из многочисленных нейронов, и каждый нейрон имеет собственное множество весов) и смещений сети, минимизирующих выбранную функцию, которая обычно именуется стоимостной функцией и, как правило, обозначается буквой J.

В дифференциальном исчислении существует несколько методов отыскания минимума заданной функции аналитически. К сожалению, во всех нейросетевых приложениях число весов настолько велико, что использовать эти методы невозможно и необходимо полагаться на численные методы, наиболее известным из которых является градиентный спуск. Градиентный спуск — это самый простой для понимания метод, и он даст вам идеальную основу для понимания более сложных алгоритмов, которые вы увидите позже в книге. В связи с этим приведем краткий обзор того, как он работает, потому что это один из лучших алгоритмов в машинном обучении, который позволяет познакомиться с понятием темпа заучивания и его причудами.

С учетом обобщенной функции $J(\mathbf{w})$, где $\mathbf{w}$ — весовой вектор, минимальное местоположение в весовом пространстве (т. е. значение $\mathbf{w}$, для которого $J(\mathbf{w})$ имеет минимум) можно найти с помощью алгоритма, основанного на следующих ниже шагах:

1. Итерация 0: выдвинуть случайную первоначальную догадку $\mathbf{w}_0$.
2. Итерация $n+1$ (где n начинается с 0): веса $\mathbf{w}_{n+1}$ на итерации $n+1$ будут обновлены из предыдущих значений $\mathbf{w}_n$ итерации n с помощью формулы:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma \nabla J(\mathbf{w}_n).$$

Через $\nabla J(\mathbf{w})$ мы обозначили градиент стоимостной функции, т. е. вектор, компоненты которого являются частными производными стоимостной функции по всем компонентам весового вектора $\mathbf{w}$, следующим образом:

$$\nabla J(\mathbf{w}) = \begin{bmatrix} \partial J(\mathbf{w}) / \partial w_1 \\ \vdots \\ \partial J(\mathbf{w}) / \partial w_{n_x} \end{bmatrix}.$$

Для принятия решения о том, когда следует остановиться, мы могли бы проверять, когда стоимостная функция $J(\mathbf{w})$ перестанет изменяться слишком сильно, или, другими словами, мы могли бы задать порог ε и остановиться на любой итерации $q > k$ (где k — это целое число, которое нужно найти), которая удовлетворяет выражению $\left|J\left(w_{q+1}\right) - J\left(w_q\right)\right| < \varepsilon$ | для всех $q < k$. Проблема с этим подходом заключается в том, что он является сложным и очень дорогим в плане производительности при реализации на языке Python (вам придется выполнять этот шаг очень большое число раз), поэтому, как правило, алгоритму дается время для работы в течение фиксированного большого числа итераций, после чего проверяют окончательные результаты. Если результат не тот, который ожидается, то фиксированное большое число итераций увеличивают. Насколько оно должно быть большим? Дело в том, что это зависит от вашей задачи. Вы выбираете определенное число итераций (например, 10 000 или 1 000 000) и запускаете алгоритм. В то же самое время вы строите график стоимостной функции в сопоставлении с числом итераций и проверяете разумность выбранного числа итераций. Позже в этой главе вы увидите практический пример, в котором будет показано, как проверять достаточность выбранного большого числа. А пока нужно запомнить, что вы останавливаете алгоритм после фиксированного числа итераций.

ПРИМЕЧАНИЕ. Ответ на вопрос "Почему этот алгоритм сходится к минимуму (и как это математически продемонстрировать)?" выходит за рамки данной книги, сделал бы эту главу слишком длинной и отвлечет читателя от основной учебной цели, которая заключается в том, чтобы вы поняли характер эффекта в зависимости от выбора конкретного темпа заучивания и каковы последствия выбора слишком большого или слишком малого темпа.

Здесь мы исходим из того, что стоимостная функция является дифференцируемой. Обычно это не так, но обсуждение этого вопроса выходит далеко за рамки данной книги. Люди склонны принимать в этом деле практический подход. Реализации очень хорошо работают, и поэтому такого рода теоретические вопросы обычно на практике игнорируются большинством разработчиков. Напомним, что в глубоко обучающихся моделях стоимостная функция становится невероятно сложной, и исследовать ее практически невозможно.

После разумного числа итераций числовой ряд $\mathbf{w}_n$, надо надеяться, сойдется к минимальному местоположению. Параметр γ называется темпом заучивания[6] и явля-

---

[6] Английский термин rate of learning часто переводится как "скорость самообучения", "скорость усвоения", что тоже верно. В данной книге со словом "скорость" используется ряд других понятий: скорость схождения, скорость ослабления, скорость вариации и пр., иногда в одном предложении или абзаце. И чтобы выделить этот термин и не вносить путаницы, он переведен как "темп заучивания". — *Прим. пер.*

ется одним из самых важных параметров, необходимых в процессе самообучения нейронной сети.

ПРИМЕЧАНИЕ. В отличие от весов темп заучивания называется гиперпараметром. Мы столкнемся со многими другими. Гиперпараметр — это параметр, значение которого не определяется в результате тренировки нейросети, а обычно устанавливается до начала процесса ее самообучения. Напротив, значения параметров $\mathbf{w}$ и b выводятся в результате тренировки.

Словосочетание "надо надеяться" было выбрано не случайно. Не исключено, что алгоритм к минимуму не сойдется. Возможно даже, что числовой ряд $\mathbf{w}_n$ будет осциллировать между значениями, не сходясь вовсе либо откровенно расходиться. Стоит только выбрать γ слишком большим либо слишком малым, и ваша модель сходиться не будет (либо будет сходиться слишком медленно). Для того чтобы понять причину, давайте рассмотрим практический случай и посмотрим, как этот метод работает при выборе разных темпов заучивания.

Темп заучивания на практическом примере

Рассмотрим набор данных `y`, образованный в результате $m = 30$ наблюдений, сгенерированный следующим фрагменом кода:

```
m = 30
w0 = 2
w1 = 0.5
x = np.linspace(-1,1,m)
y = w0 + w1 * x
```

В качестве стоимостной функции мы выбираем классическую среднеквадратическую ошибку (mean squared error, MSE):

$$J(w_0, w_1) = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - f\left(w_0, w_1, x^{(i)}\right)\right)^2,$$

где верхним индексом (i) мы обозначили i-е наблюдение. Напомним, что с помощью нижнего индекса i (x_i) мы обозначили i-й признак. То есть форма записи $x_j^{(i)}$ обозначает j-й признак и i-е наблюдение. В приведенном здесь примере у нас имеется только один признак, поэтому нижний индекс j нам не нужен. Стоимостная функция может быть легко реализована на Python следующим образом:

```
np.average((y-hypothesis(x, w0, w1))**2, axis=2)/2
```

где мы определили

```
def hypothesis(x, w0, w1):
    return w0 + w1*x
```

Наша цель — найти значения w_0 и w_1, которые минимизируют $J(w_0, w_1)$.

Для того чтобы применить метод градиентного спуска, мы должны вычислить числовые ряды для $w_{0,n}$ и $w_{1,n}$. У нас есть следующие уравнения:

$$\begin{cases} w_{0,n+1} = w_{0,n} - \gamma \dfrac{\partial J\left(w_{0,n}, w_{1,n}\right)}{\partial w_0} = w_{0,n} + \gamma \dfrac{1}{m} \sum_{i=1}^{m} 2\left(y_i - f\left(w_{0,n}, w_{1,n}, x_i\right)\right) \dfrac{\partial f\left(w_0, w_1, x_i\right)}{\partial w_0}; \\ w_{1,n+1} = w_{1,n} - \gamma \dfrac{\partial J\left(w_{0,n}, w_{1,n}\right)}{\partial w_1} = w_{1,n} + \gamma \dfrac{1}{m} \sum_{i=1}^{m} 2\left(y_i - f\left(w_{0,n}, w_{1,n}, x_i\right)\right) \dfrac{\partial f\left(w_0, w_1, x_i\right)}{\partial w_1}. \end{cases}$$

Упростив уравнения, вычислив частные производные, получим:

$$\begin{cases} w_{0,n+1} = w_{0,n} + \dfrac{\gamma}{m} \sum_{i=1}^{m} \left(y_i - f\left(w_{0,n}, w_{1,n}, x_i\right)\right) = w_{0,n}\left(1-\gamma\right) + \dfrac{\gamma}{m} \sum_{i=1}^{m} \left(y_i - w_{1,n} x_i\right); \\ w_{1,n+1} = w_{1,n} + \dfrac{\gamma}{m} \sum_{i=1}^{m} \left(y_i - f\left(w_{0,n}, w_{1,n}, x_i\right)\right) x_i = w_{1,n} - \gamma w_{0,n} + \dfrac{\gamma}{m} \sum_{i=1}^{m} \left(y_i - w_{1,n} x_i\right) x_i. \end{cases} \quad (**)$$

Поскольку $\partial f\left(w_0, w_1, x_i\right)/\partial w_0 = 1$ и $\partial f\left(w_0, w_1, x_i\right)/\partial w_1 = x_i$, приведенные выше уравнения должны быть реализованы на Python, в случае если мы хотим запрограммировать алгоритм градиентного спуска самостоятельно.

ПРИМЕЧАНИЕ. Математическое выведение уравнений в формуле (**) имеет целью показать, насколько быстро уравнения градиентного спуска становятся очень сложными, даже для очень легкого случая. В следующем разделе мы построим первую модель с использованием TensorFlow. Одним из лучших аспектов данной библиотеки является то, что все эти формулы вычисляются автоматически, и вам не нужно ничего вычислять самостоятельно. Реализация уравнений, подобных приведенным здесь, и их отладка могут занять довольно много времени и оказаться невозможными в тот момент, когда вы столкнетесь с большими нейронными сетями взаимосвязанных нейронов.

В этой книге полная реализация данного примера на Python опущена, потому что потребовала бы слишком много места.

Поучительно выполнять сопоставление результативности работы модели, варьируя темп заучивания. На рис. 2.10–2.12 начерчены контурные линии[7] стоимостных функций, и поверх них в виде точек был нанесен числовой ряд ($w_{0,n}$, $w_{1,n}$) с целью визуализации схождения (или несхождения) числового ряда. На рисунках минимум обозначен окружностью, расположенной приблизительно в центре. Рассмотрим значения $\gamma = 0{,}8$ (см. рис. 2.10), $\gamma = 2$ (см. рис. 2.11) и $\gamma = 0{,}05$ (см. рис. 2.12). Разные оценки, $\mathbf{w}_n$, обозначены точками. Минимум обозначен окружностью приблизительно посередине изображения.

[7] Контурная линия функции — это кривая, вдоль которой функция имеет постоянное значение.

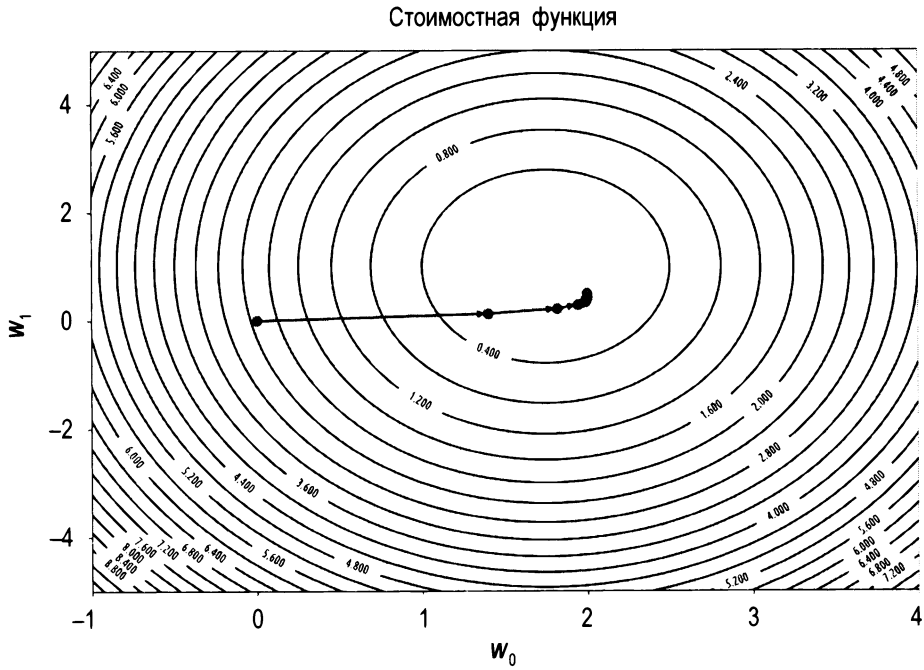


РИС. 2.10. Иллюстрация алгоритма градиентного спуска с благополучным схождением

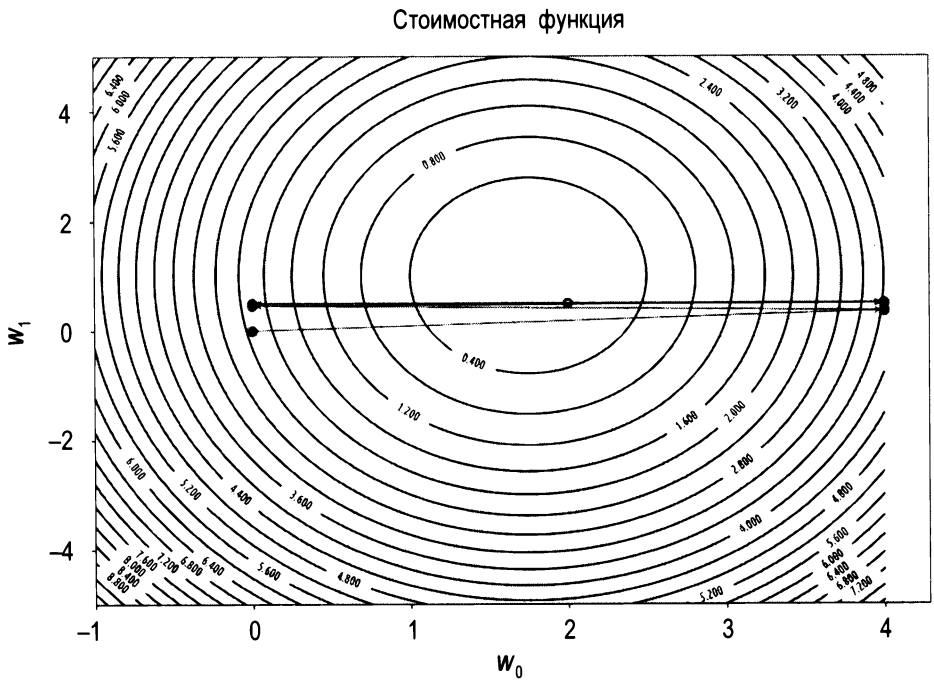


РИС. 2.11. Иллюстрация алгоритма градиентного спуска, когда темп заучивания является слишком большим. Метод не способен сойтись к минимуму

В первом случае (см. рис. 2.10) мы видим благополучное схождение, и метод сходится к минимуму всего за восемь шагов. При $\gamma = 2$ (см. рис. 2.11) метод делает слишком большие шаги (шаги задаются выражением $-\gamma\nabla J(\mathbf{w})$, и, следовательно, чем больше γ, тем больше шаги) и не способен приблизиться к минимуму. Метод продолжает осциллировать вокруг минимума, его не достигая. В этом случае модель никогда не сойдется. В последнем случае, при $y = 0{,}05$ (см. рис. 2.12), заучивание идет настолько медленно, что для приближения к минимуму потребуется еще много шагов. В некоторых случаях стоимостная функция может быть настолько плоской вокруг минимума, что для схождения методу требуется такое большое число итераций, что практически вы не сможете приблизиться к реальному минимуму за разумное время. На рис. 2.12 показано 300 итераций, но метод даже и не приблизился к минимуму.

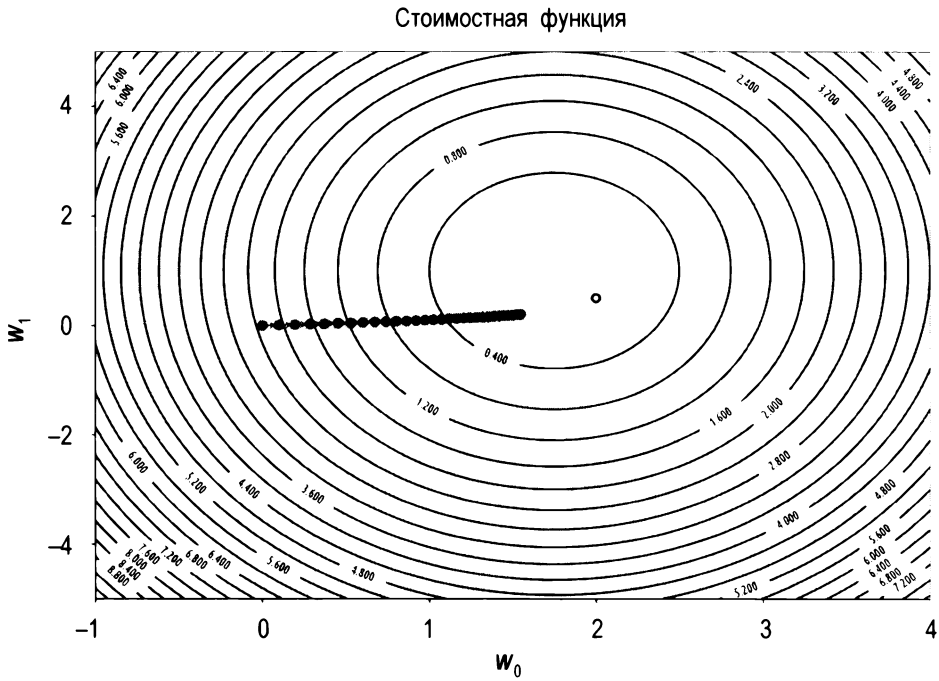


РИС. 2.12. Иллюстрация алгоритма градиентного спуска, когда темп заучивания является слишком малым. Метод настолько медленный, что для приближения к минимуму потребуется огромное число итераций

ПРИМЕЧАНИЕ. Правильный выбор темпа заучивания имеет первостепенное значение во время программирования самообучающейся части нейронной сети. Стоит лишь выбрать слишком большой темп, и метод просто будет скакать вокруг минимума, никогда его не достигая. Стоит выбрать слишком малый темп, и алгоритм может стать настолько медленным, что вам не удастся отыскать минимум за разумное время (или число итераций). Типичным признаком слишком

большого темпа заучивания является то, что стоимостная функция может стать nan ("не числом" на Python-жаргоне). Вывод на печать стоимостной функции через регулярные промежутки времени в тренировочном процессе является хорошим способом проверки наличия такого рода проблем. Это даст вам возможность останавливать процесс и избегать временны́х потерь (в случае, если вы увидите появление nan). Конкретный пример приведен далее в этой главе.

В задачах глубокого обучения каждая итерация будет стоить времени, и вам придется выполнять этот процесс несколько раз. Правильный выбор темпа заучивания является ключевой частью построения хорошей модели, потому что он значительно ускорит тренировку модели (либо сделает тренировку невозможной).

Иногда целесообразно менять темп заучивания в процессе тренировки. Вы начинаете с более крупного значения с целью быстрее приблизиться к минимуму, а затем постепенно его уменьшаете, чтобы как можно ближе приблизиться к реальному минимуму. В дальнейшем в этой книге данный подход будет рассмотрен подробнее.

ПРИМЕЧАНИЕ. Какие-то фиксированные правила в отношении того, как выбирать правильный темп заучивания, отсутствуют. Все зависит от модели, от стоимостной функции, от первоначальной точки и т. д. Хорошее эмпирическое правило состоит в том, чтобы начать с $\gamma = 0{,}05$, а затем смотреть на поведение стоимостной функции. Довольно распространенной практикой является построение графика $J(\mathbf{w})$ в сопоставлении с числом итераций с целью проверки ее уменьшения и динамики ее уменьшения.

Хороший способ проверить сходимость — построить график стоимостной функции в сопоставлении с числом итераций. Благодаря этому вы имеете возможность проверить ее поведение. На рис. 2.13 показан внешний вид стоимостной функции при трех темпах заучивания для приведенного выше примера. Ясно видно, насколько быстро случай с $\gamma = 0{,}8$ обнуляется, указывая на то, что мы достигли минимума. Случай с $\gamma = 2$ даже не начинает идти вниз. Он по-прежнему остается почти на том же первоначальном значении. И наконец, случай с $\gamma = 0{,}05$ начинает идти вниз, но гораздо медленнее, чем в первом случае.

Итак, вот выводы, которые мы должны сделать из рис. 2.13 для трех случаев.

- $\gamma = 0{,}05$ — J уменьшается, что хорошо, но после восьми итераций мы не достигли плато, поэтому мы должны задействовать еще много итераций до тех пор, пока не увидим, что J больше не изменяется.
- $\gamma = 2$ — J не уменьшается. Мы должны проверить темп заучивания на предмет его модификации. Применение меньших значений было бы хорошей отправной точкой.
- $\gamma = 0{,}8$ — стоимостная функция довольно быстро уменьшается, а затем остается постоянной. Это хороший знак и говорит о том, что мы достигли минимума.

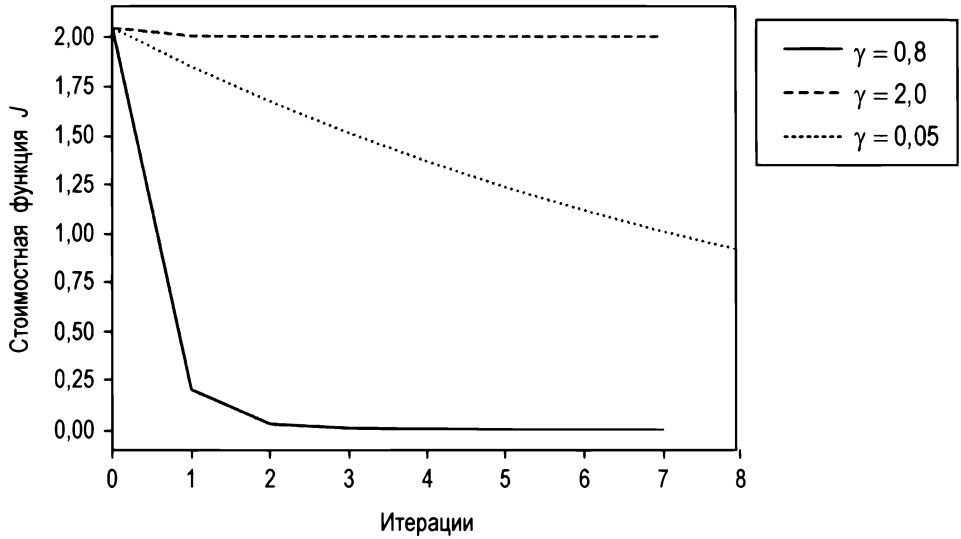


РИС. 2.13. Стоимостная функция в сопоставлении с числом итераций (рассматриваются только первые восемь)

Также напомним, что абсолютное значение темпа заучивания не играет роли. Что важно, так это поведение. Мы можем умножить стоимостную функцию на константу, и это никак не повлияет на самообучение модели. Не смотрите на абсолютные значения; проверьте скорость и характер поведения стоимостной функции. Кроме того, стоимостная функция почти никогда не достигнет нуля, поэтому не стоит этого ожидать. Значение J при ее минимуме почти никогда не равно нулю (оно зависит от самих функций). В разделе о линейной регрессии вы найдете пример, в котором стоимостная функция не достигнет нуля.

ПРИМЕЧАНИЕ. Во время тренировки моделей всегда проверяйте стоимостную функцию в сравнении с числом итераций (либо числом проходов по всему тренировочному набору, именуемых эпохами). Это даст вам эффективный способ оценить результативность тренировки, если она вообще работает, и советы о том, как ее оптимизировать.

Теперь, когда мы определили основу, воспользуемся нейроном для решения двух простых задач машинного обучения: линейной и логистической регрессии.

Пример линейной регрессии в TensorFlow

Первый тип регрессии даст возможность понять то, как строится модель в TensorFlow. Для объяснения того, каким образом эффективно выполнять линейную регрессию с одним-единственным нейроном, сначала необходимо объяснить некоторые дополнительные обозначения. В предыдущих разделах рассматривались входы $x = (x_1, x_2, ..., x_{n_x})$. Это так называемые признаки, которые описывают на-

блюдения. Обычно наблюдений бывает много. Как уже отмечалось ранее, для обозначения разных наблюдений мы будем использовать верхний индекс между скобками. Наше i-е наблюдение будет обозначено через $x^{(i)}$, а j-й признак i-го наблюдения будет обозначен через $x_j^{(i)}$. Число наблюдений мы будем обозначать буквой m.

ПРИМЕЧАНИЕ. В этой книге m — это число наблюдений, а n_x — число признаков. Наш j-й признак i-го наблюдения будет обозначаться через $x_j^{(i)}$. В проектах глубокого обучения чем больше m, тем лучше. Поэтому будьте готовы иметь дело с огромным числом наблюдений.

Как уже отмечалось ранее, библиотека NumPy оптимизирована для выполнения многочисленных параллельных операций одновременно. Для того чтобы получить максимально возможную производительность, важно записывать наши уравнения в матричной форме и передавать матрицы в NumPy. Благодаря этому наш код будет максимально эффективным. Напомним, что следует любой ценой избегать циклов, когда это возможно. Теперь давайте потратим некоторое время на написание всех уравнений в матричной форме. В результате этого наша реализация на Python будет намного проще позднее.

Все множество входов (признаков и наблюдений) может быть записано в матричной форме. Мы будем использовать следующее обозначение:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(m)} \\ \vdots & \ddots & \vdots \\ x_{n_x}^{(1)} & \dots & x_{n_x}^{(m)} \end{bmatrix},$$

где каждый столбец — это наблюдение, а каждая строка — признак в матрице $\mathbf{X}$, имеющей размерность $n_x \times m$. Выходные значения $\hat{y}^{(i)}$ можно тоже записать в матричной форме. Как вы помните из обсуждения темы нейронов, мы определили $z^{(i)} = \mathbf{w}^T \mathbf{x}^{(i)} + b$ для одного наблюдения i. Поместив каждое наблюдение в столбец, мы можем применить следующую форму записи:

$$\mathbf{z} = \begin{bmatrix} z^{(1)} & z^{(2)} & \dots & z^{(m)} \end{bmatrix} = \mathbf{w}^T \mathbf{X} + \mathbf{b},$$

где мы имеем $\mathbf{b} = \begin{bmatrix} b & b & \dots & b \end{bmatrix}$. Определим $\hat{\mathbf{y}}$ как

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \dots & \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} f\left(z^{(1)}\right) & f\left(z^{(2)}\right) & \dots & f\left(z^{(m)}\right) \end{bmatrix} = f(\mathbf{z}),$$

где под $f(z)$ мы имеем в виду, что функция f применяется к матрице $\mathbf{z}$ поэлементно.

ПРИМЕЧАНИЕ. Хотя $\mathbf{z}$ имеет размерность $1 \times m$, для него мы будем использовать термин "матрица", а не вектор. Это делается для того, чтобы имена в книге оставались непротиворечивыми. Это также будет вам напоминать о том, что мы всегда должны использовать матричные операции. Для наших целей $\mathbf{z}$ — это просто матрица всего с одной строкой.

Из *главы 1* вы знаете, что в TensorFlow необходимо объявлять размерности матриц (или тензоров) явным образом, поэтому неплохо держать их под строгим контролем. Приведем обзор размерностей всех векторов и матриц, которые мы будем использовать:

- **X** имеет размерность $n_x \times m$;
- **z** имеет размерность $1 \times m$;
- $\hat{\mathbf{y}}$ имеет размерность $1 \times m$;
- **w** имеет размерность $n_x \times 1$;
- **b** имеет размерность $1 \times m$.

Теперь, когда формализм ясен, мы подготовим набор данных.

Набор данных для линейной регрессионной модели

Для того чтобы было чуть поинтереснее, давайте воспользуемся реальным набором данных. Мы будем использовать так называемый бостонский набор данных[8]. В нем содержится информация, собранная Бюро переписи населения США относительно жилья в окрестностях города Бостона. Каждая запись в базе данных описывает Бостон или его пригород. Данные были взяты из стандартной агломерации Бостона (Standard Metropolitan Statistical Area, SMSA) в 1970 году. Атрибуты данных определены следующим образом [3]:

- CRIM — уровень преступности на душу населения в разбивке по населенным пунктам;
- ZN — доля жилых земельных участков, зонированных для лотов свыше 25 тыс. кв. футов;
- INDUS — доля акров нерозничных предприятий в расчете на населенный пункт;
- CHAS — фиктивная переменная реки Чарльз (равна 1, если участок ограничивает реку; 0 в противном случае);
- NOX — концентрация оксидов азота (частей на 10 млн);
- RM — среднее число комнат в доме;
- AGE — доля занимаемых владельцами квартир, построенных до 1940 года;
- DIS — взвешенные расстояния до пяти бостонских центров занятости;
- RAD — индекс доступности радиальных магистралей;
- TAX — полная ставка налога на имущество на 10 тыс. долларов;

---

[8] Данные для оцениваний процесса заучивания в действующих экспериментах (Delve от англ. Data for Evaluating Learning in Valid Experiments) под названием "The Boston Housing Dataset" (Бостонский набор данных с ценами на жилую недвижимость), **www.cs.toronto.edu/~delve/data/boston/bostonDetail.html**, 1996.

- PTRATIO — соотношение ученик-учитель в разбивке по населенным пунктам;
- B – 1000(Bk – 0.63)^2 – Bk — доля афроамериканцев в разбивке по населенным пунктам;
- LSTAT — процент населения с низким статусом;
- MEDV — медианная стоимость занимаемых владельцами домов, измеряемая в тысячах долларов.

Наша целевая переменная MEDV, которую мы хотим предсказать, — это медианная цена дома, измеряемая в 1000 долларов для каждого пригорода. В нашем примере нам не нужно разбираться или изучать признаки. Цель здесь — показать, как строить линейную регрессионную модель, основываясь на тех знаниях, которые вы уже получили. В проекте машинного обучения обычно сначала исследуются входные данные, проверяются их распределение, качество, пропущенные значения и т. д. Однако эта часть будет пропущена для того, чтобы сосредоточиться на реализации ваших знаний о библиотеке TensorFlow.

ПРИМЕЧАНИЕ. В машинном обучении переменная, которую мы хотим предсказать, обычно называется целевой переменной.

Давайте импортируем обычные библиотеки, включая `sklearn.datasets`. Импортировать данные и получить признаки и целевое значение очень легко с помощью программного пакета `sklearn.datasets`. Вам не нужно загружать CSV-файлы и импортировать их. Просто выполните следующий фрагмент кода:

```
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
import numpy as np
from sklearn.datasets import load_boston

boston = load_boston()
features = np.array(boston.data)
labels = np.array(boston.target)
```

Каждый набор данных в пакете `sklearn.datasets` сопровождается описанием. Вы можете обратиться к нему с помощью инструкции:

```
print(boston["DESCR"])
```

Теперь давайте взглянем, сколько у нас наблюдений и признаков.

```
n_training_samples = features.shape[0]
n_dim = features.shape[1]
print('Набор данных имеет ',n_training_samples,' тренировочных образцов.')
print('Набор данных имеет ',n_dim,' признаков.')
```

Если соотнести математические обозначения с Python-кодом, то `n_training_samples` равно m, `n_dim` равно n_x. Указанный фрагмент кода даст следующие результаты:

```
Набор данных имеет 506 тренировочных образов.
Набор данных имеет 13 признаков.
```

Рекомендуется нормализовать каждый числовой признак, определив нормализованные признаки $x^{(i)}_{\text{норм},\,j}$ по формуле:

$$x^{(i)}_{\text{норм},\,j} = \frac{x^{(i)}_j - \left\langle x^{(i)}_j \right\rangle}{\sigma^{(i)}_j},$$

где $\left\langle x^{(i)}_j \right\rangle$ — среднее значение j-го признака; $\sigma^{(i)}_j$ — его стандартное отклонение.

Нормализация легко вычисляется в NumPy с помощью следующей функции:

```
def normalize(dataset):
    mu = np.mean(dataset, axis = 0)
    sigma = np.std(dataset, axis = 0)
    return (dataset-mu)/sigma
```

Для того чтобы нормализовать массив NumPy, мы должны лишь вызвать функцию `features_norm = normalize(features)`. Теперь каждый признак, содержащийся в массиве NumPy `features_norm`, будет иметь нулевое среднее значение и единичное стандартное отклонение.

ПРИМЕЧАНИЕ. Обычно рекомендуется нормализовывать признаки так, чтобы их среднее значение равнялось нулю, а стандартное отклонение — единице. Иногда величина некоторых признаков намного больше других и может оказывать более сильное влияние на модель, тем самым приводя к неправильным предсказаниям. Особая осторожность с целью обеспечения непротиворечивых нормализаций требуется, когда набор данных разбивается на тренировочный и тестовый наборы данных.

В этой главе для тренировки будут использоваться все данные целиком. Это будет сделано для того, чтобы сосредоточиться на деталях реализации.

```
train_x = np.transpose(features_norm)
train_y = np.transpose(labels)

print(train_x.shape)
print(train_y.shape)
```

Последние две инструкции печати покажут нам размерности новых матриц.

```
(13, 506)
(506,)
```

Массив `train_x` имеет размерность (13, 506), и это именно то, что мы ожидаем. Как мы уже обсуждали, **X** имеет размерность $n_x \times m$.

Тренировочная цель `train_y` имеет размерность `(506,)`. Именно так библиотека NumPy описывает одномерные массивы. Библиотека TensorFlow требует размер-

ность `(1, 506)` (вспомните наше предыдущее обсуждение), поэтому необходимо реформировать массив (с использованием метода NumPy `reshape`) следующим образом:

```
train_y = train_y.reshape(1,len(train_y))
print(train_y.shape)
```

и инструкция печати дает нам то, что нам нужно:

```
(1, 506)
```

Нейрон и стоимостная функция для линейной регрессии

Нейрон, который может выполнять линейную регрессию, использует активационную функцию тождественного отображения. Подлежащая минимизации стоимостная функция представлена среднеквадратической ошибкой MSE, которая может быть записана как

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \mathbf{w}^T\mathbf{x}^{(i)} - b\right)^2,$$

где суммирование проводится по всем m наблюдениям.

Исходный код TensorFlow для построения этого нейрона и определения стоимостной функции на самом деле очень прост.

```
X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())
W = tf.Variable(tf.ones([n_dim,1]))
b = tf.Variable(tf.zeros(1))

init = tf.global_variables_initializer()
y_ = tf.matmul(tf.transpose(W),X)+b
cost = tf.reduce_mean(tf.square(y_-Y))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Обратите внимание, что в TensorFlow нет необходимости явно объявлять число наблюдений. В исходном коде можно использовать `None`. Благодаря этому вы можете выполнять модель на любом наборе данных независимо от числа наблюдений без модификации исходного кода.

В исходном коде мы указали выход $\hat{\mathbf{y}}$ нейрона как `y_`, потому что в Python отсутствует возможность обозначать переменные с крышкой. Отдельно остановимся на разъяснении того, какая строка кода, что конкретно делает.

- `X = tf.placeholder(tf.float32, [n_dim, None])` содержит матрицу $\mathbf{X}$, которая должна иметь размерность $n_x \times m$. Напомним, что в нашем коде `n_dim` — это n_x и что в TensorFlow m не объявляется явно. Вместо него мы используем `None`.

- `Y = tf.placeholder(tf.float32, [1, None])` содержит выходные значения $\hat{\mathbf{y}}$, которые должны иметь размерность $1 \times m$. Здесь это значит, что вместо m мы применяем `None`, потому что хотим использовать одну и ту же модель для разных наборов данных (которые будут иметь разное число наблюдений).
- `learning_rate = tf.placeholder(tf.float32, shape=())` содержит темп заучивания как параметр, а не константа, благодаря чему мы можем запускать одну и ту же самую модель, варьируя темп, без необходимости каждый раз создавать новый нейрон.
- `W = tf.Variable(tf.zeros([n_dim, 1]))` определяет и инициализирует нулями веса $\mathbf{w}$. Напомним, что веса $\mathbf{w}$ должны иметь размерность $n_x \times 1$.
- `b = tf.Variable(tf.zeros(1))` определяет и инициализирует нулями смещение b.

Как вы помните, в TensorFlow заполнитель — это тензор, который не будет изменяться в фазе заучивания, тогда как переменная — это тензор, который будет изменяться. Веса $\mathbf{w}$ и смещение b будут обновляться во время заучивания. Теперь мы должны определиться с тем, что делать со всеми этими величинами. Напомним: мы должны вычислить $\mathbf{z}$. В качестве активационной функции выбрано тождественное отображение, поэтому $\mathbf{z}$ также будет выходом нашего нейрона.

- `init = tf.global_variables_initializer()` создает часть графа, которая инициализирует переменную и добавляет ее в граф.
- `y_ = tf.matmul(tf.transpose(W),X)+b` вычисляет выход нейрона. Выход нейрона равен $\hat{\mathbf{y}} = f(\mathbf{z}) = f\left(\mathbf{w}^T\mathbf{X} + b\right)$. Поскольку активационной функцией линейной регрессии является тождественное отображение, выход равен $\hat{\mathbf{y}} = \mathbf{w}^T\mathbf{X} + b$. То, что b является скаляром, не является проблемой. Python-овский механизм транслирования об этом позаботится, расширив его до нужных размерностей, сделая возможным суммирование вектора $\mathbf{w}^T\mathbf{X}$ и скаляра b.
- `cost = tf.reduce_mean(tf.square(y_-Y))` определяет стоимостную функцию. TensorFlow обеспечивает простой и эффективный способ расчета среднего значения — `tf.reduce_mean()`, который выполняет суммирование всех элементов тензора и делит сумму на число элементов.
- `training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)` сообщает библиотеке TensorFlow, какой алгоритм использовать для минимизации стоимостной функции. На языке библиотеки TensorFlow алгоритмы, используемые для минимизации стоимостной функции, называются оптимизаторами. Сейчас мы используем градиентный спуск с заданным темпом заучивания. Далее в книге будут подробно рассмотрены и другие оптимизаторы.

Из *главы 1* вы помните, что в приведенном выше исходном коде никакая модель выполняться не будет. Он просто определяет вычислительный граф. Давайте определим функцию, которая будет выполнять фактическое заучивание и выполнять нашу модель. Проще определить выполнение модели в функции. Это даст возможность ее перезапускать, к примеру, изменяя темп заучивания или число итераций, которые мы захотим использовать.

```
def run_linear_model(learning_r, training_epochs, train_obs, train_labels,
debug = False):
  sess = tf.Session()
  sess.run(init)

  cost_history = np.empty(shape=[0], dtype = float)

  for epoch in range(training_epochs+1):
    sess.run(training_step, feed_dict = {X: train_obs, Y: train_labels,
                                         learning_rate: learning_r})
    cost_ = sess.run(cost, feed_dict={X:train_obs, Y: train_labels,
                                      learning_rate: learning_r})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 1000 == 0) & debug:
      print("Достигнута эпоха",epoch,"стоимость J =",
            str.format('{0:.6f}', cost_))
```

Давайте пройдемся по исходному коду построчно.

- `sess = tf.Session()` создает сеанс TensorFlow.
- `sess.run(init)` запускает инициализацию другого элемента графов.
- `cost_history = np.empty(shape=[0], dtype = float)` создает пустой вектор (на данный момент с нулевыми элементами), в котором хранится значение стоимостной функции на каждой итерации.
- `for loop...` — в этом цикле TensorFlow выполняет шаги градиентного спуска, которые мы обсуждали ранее, и обновляет веса и смещение. Кроме того, TensorFlow каждый раз будет сохранять в массиве `cost_history` значение стоимостной функции: `cost_history = np.append(cost_history, cost_)`.
- `if (epoch % 1000 == 0)...` — каждые 1000 эпох мы будем печатать значение стоимостной функции. Это простой способ проверки того, что стоимостная функция на самом деле уменьшается или что появляются значения `nan`. Если вы выполняете какие-то предварительные тесты в интерактивной среде (например, в блокноте Jupyter), то можете остановить процесс, если увидите, что стоимостная функция ведет себя не так, как ожидалось.
- `return sess, cost_history` возвращает сеанс (если вы хотите вычислить что-то еще) и массив со значениями стоимостной функции (мы будем использовать этот массив для построения графика).

Выполнение модели сводится к простому вызову.

```
sess, cost_history = run_linear_model(learning_r = 0.01,
                                      training_epochs = 10000,
                                      train_obs = train_x,
                                      train_labels = train_y,
                                      debug = True)
```

Выходом данной инструкции будет стоимостная функция каждые 1000 эпох (в определении функции обратитесь к инструкции `if`, начиная с инструкции `if (epoch % 1000 == 0)`).

```
Достигнута эпоха 0 стоимость J = 613.947144
Достигнута эпоха 1000 стоимость J = 22.131165
Достигнута эпоха 2000 стоимость J = 22.081099
Достигнута эпоха 3000 стоимость J = 22.076544
Достигнута эпоха 4000 стоимость J = 22.076109
Достигнута эпоха 5000 стоимость J = 22.07606
Достигнута эпоха 6000 стоимость J = 22.076057
Достигнута эпоха 7000 стоимость J = 22.076059
Достигнута эпоха 8000 стоимость J = 22.076059
Достигнута эпоха 9000 стоимость J = 22.076054
Достигнута эпоха 10000 стоимость J = 22.076054
```

Стоимостная функция явно уменьшается, а затем достигает значения и остается почти постоянной. Ее график представлен на рис. 2.14. Это хороший знак, указывающий на то, что стоимостная функция достигла минимума. Но это не означает, что наша модель является хорошей или даст хорошие предсказания, а просто говорит о том, что заучивание работает эффективно.

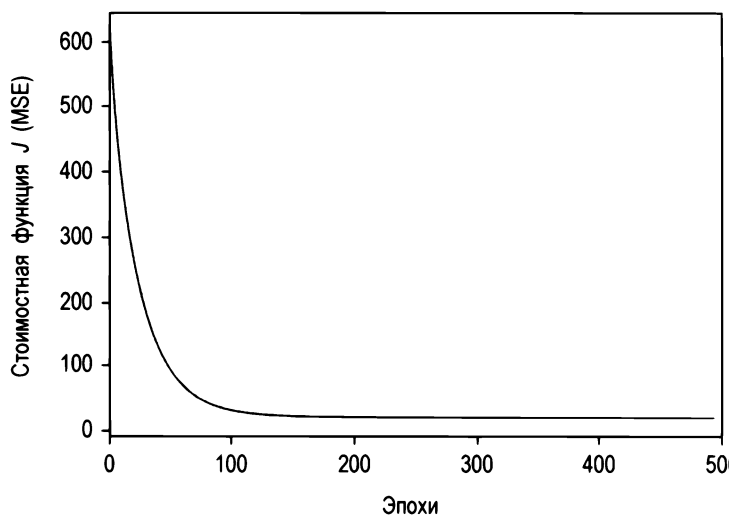


РИС. 2.14. Стоимостная функция, полученная в нашей модели, применительно к бостонскому набору данных с темпом заучивания $\gamma = 0{,}01$. Мы выводим на график только первые 500 эпох, т. к. стоимостная функция уже почти достигла своего окончательного значения

Было бы неплохо иметь возможность графически визуализировать качество нашей подгонки. Поскольку у нас 13 признаков, построить график цены в сопоставлении с другими признаками не выйдет. Тем не менее полезно получить представление о том, насколько хорошо модель предсказывает наблюдаемые значения. Это можно

сделать, выведя на график предсказываемую целевую переменную в сопоставлении с наблюдаемой (рис. 2.15). Если мы можем идеально предсказать нашу целевую переменную, то все точки на графике должны быть на диагональной линии. Чем больше точек разбросано вокруг линии, тем хуже модель предсказывает. Давайте проверим, как наша модель работает.

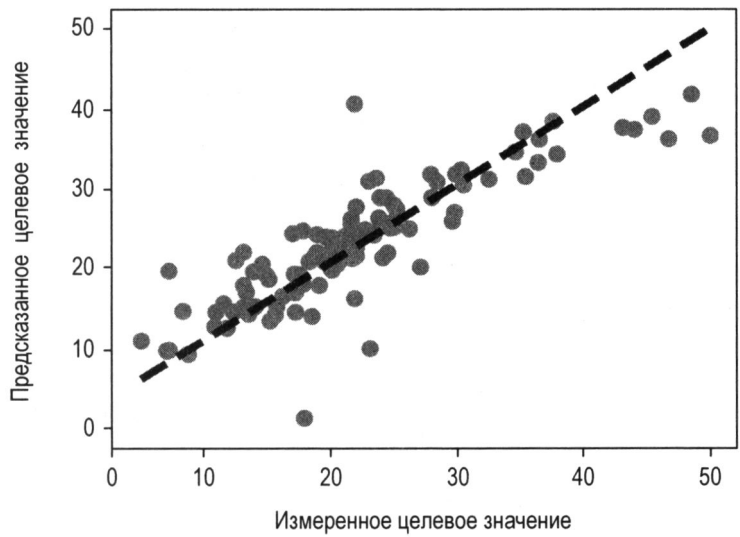


РИС. 2.15. Предсказанное целевое значение в сопоставлении с измеренным целевым значением в нашей модели применительно к тренировочным данным

Точки лежат достаточно хорошо вокруг линии, поэтому, по всей видимости, мы можем в определенной мере предсказать нашу цену. Более качественный метод оценивания точности регрессии — сама среднеквадратическая ошибка (которая в нашем случае является стоимостной функцией). Ответ на вопрос "Является ли значение, которое мы получаем (22,08 в тысячах долларов), достаточно хорошим?" зависит от задачи, которую вы пытаетесь решить, или от ограничения и требования, которые вам были заданы.

Разумно достаточный и оптимизационный метрический показатель

Мы видели, что ответить на вопрос "Является ли модель хорошей?" совсем не просто. Рисунок 2.15 не позволит нам дать качественное описание того, насколько наша модель хороша (или не хороша). Для этого мы должны определить метрический показатель.

Проще всего настроить так называемый одиночный оценочный метрический показатель, т. е. вы вычисляете одно-единственное число и основываете оценивание модели на этом числе. Этот подход являетя легким и очень практичным. Например, в случае классификации вы могли бы применить показатель точности или оценку

F1, а в случае регрессии — среднеквадратическую ошибку MSE. Обычно в реальной жизни в модели присутствуют цели и ограничения. Например, компании вполне может понадобиться получить предсказание цены на жилье с MSE < 20 (в тысячах долларов), и при этом ваша модель должна работать на iPad или менее чем за 1 секунду. Поэтому полезно различать два типа метрических показателей[9].

- **Разумно достаточный метрический показатель** — поиск среди имеющихся альтернатив до тех пор, пока не будет достигнут порог разумной достаточности, например время работы RT (run time) исходного кода, которое минимизирует стоимостную функцию при условии, что RT менее 1 секунды, или выбор среди режимов такого, который имеет RT менее 1 секунды.
- **Оптимизационный метрический показатель** — поиск среди имеющихся альтернатив с целью максимизации конкретного метрического показателя, например выбор модели (или гиперпараметров), которая обеспечивает максимальную точность.

ПРИМЕЧАНИЕ. Если у вас несколько метрик, то следует всегда выбирать один оптимизационный показатель, а все остальные будут разумно достаточными.

Мы написали исходный код так, чтобы он мог выполнять модель с разными параметрами. Характер поведения стоимостной функции в условиях трех разных темпов заучивания: 0,1; 0,01 и 0,001, будет очень показательным. Разные варианты ее поведения представлены на рис. 2.16.

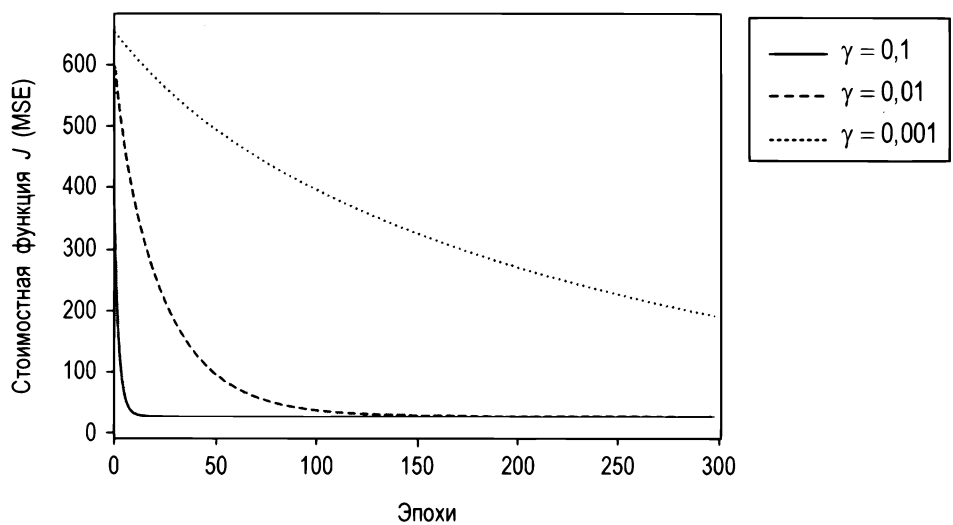


РИС. 2.16. Стоимостная функция для линейной регрессии применительно к бостонскому набору данных для трех темпов заучивания: 0,1 (сплошная линия), 0,01 (пунктирная линия) и 0,001 (точечная линия). Чем меньше темп заучивания, тем медленнее процесс заучивания

[9] В оригинале используются широко применяемые термины satisfycing и optimizing, соответствующие "достаточно хорошей" и оптимальной величинам показателя. — *Прим. пер.*

Как и ожидалось, для очень малых темпов заучивания (0,001) алгоритм градиентного спуска отыскивает минимум очень медленно, тогда как с более крупным значением (0,1) этот метод работает быстро. Этот вид графика очень полезен тем, что он дает представление о том, как быстро и насколько хорошо проходит процесс заучивания. В дальнейшем мы рассмотрим случаи, где стоимостная функция ведет себя гораздо менее благополучно. Например, при применении регуляризации методом отсева стоимостная функция больше не будет гладкой.

Пример логистической регрессии

Логистическая регрессия — это классический классификационный алгоритм. Мы не будем усложнять и рассмотрим здесь бинарную классификацию, т. е. мы будем иметь дело с задачей распознавания только двух классов, которые назовем 0 или 1. Нам понадобится активационная функция, отличная от той, которую мы использовали для линейной регрессии, другая стоимостная функция для минимизации и небольшая модификация выхода нейрона. Наша цель состоит в том, чтобы построить модель, способную предсказывать принадлежность того или иного нового наблюдения к одному из двух классов.

На выходе нейрон должен давать вероятность $P(y=1 \mid x)$ того, что вход x принадлежит к классу 1. Тогда мы сможем отнести наше наблюдение к классу 1, если $P(y=1 \mid x) > 0{,}5$, либо к классу 0, если $P(y=1 \mid x) < 0{,}5$.

Стоимостная функция

В качестве стоимостной функции мы будем использовать перекрестную энтропию[10]. Ее функция для одного наблюдения равна

$$L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\left(y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}^{(i)}\right)\right).$$

При наличии более одного наблюдения стоимостная функция является суммой всех наблюдений:

$$J(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right).$$

В *главе 10* будет приведен полный математический каркас логистической регрессии с нуля, а пока обо всех деталях позаботится TensorFlow — производные, реализации градиентного спуска и т. д. Нам только нужно построить правильный нейрон, и мы пойдем дальше.

---

[10] Анализ смысла перекрестной энтропии выходит за рамки этой книги. Хорошее введение в данную тему можно найти по адресу **https://rdipietro.github.io/friendly-intro-to-crossentropy-loss** и во многих вводных книгах по машинному обучению.

Активационная функция

Напомним: мы хотим, чтобы наш нейрон выводил вероятность того, что наше наблюдение относится к классу 0 либо классу 1. Поэтому нам нужна активационная функция, которая допускает значения только между 0 и 1. В противном случае мы не сможем рассматривать их как вероятность. Для логистической регрессии в качестве активационной функции будем использовать сигмоиду.

Набор данных

Для построения интересной модели мы будем использовать модифицированную версию набора данных MNIST. Вы найдете всю необходимую информацию о нем по следующей ссылке: **http://yann.lecun.com/exdb/mnist/**.

База данных MNIST представляет собой крупную коллекцию рукописных цифр, которую можно использовать для тренировки модели. База данных MNIST содержит 70 тыс. изображений. "Исходные черно-белые (двухуровневые) изображения из NIST были нормализованы по размеру и помещены в прямоугольник размером 20×20 пикселов с сохранением пропорций. Результирующие изображения содержат уровни серого, получившиеся в результате применения метода сглаживания алгоритмом нормализации. Изображения были центрированы в поле 28×28 путем вычисления центра масс пикселов и транслирования изображения таким образом, чтобы расположить эту точку в центре поля 28×28" (источник: **http://yann.lecun.com/exdb/mnist/**).

Наши признаки будут иметь значение серого для каждого пиксела, поэтому в общей сложности будет $28 \times 28 = 784$ признака, значения которых будут располагаться в интервале от 0 до 255 (значения серого). Набор данных содержит все десять цифр от 0 до 9. С помощью представленного далее фрагмента кода можно подготовить данные для использования в последующих разделах.

ПРИМЕЧАНИЕ. В оригинале книги импортирование набора данных MNIST делалось с помощью функции `fetch_mldata` пакета `sklearn.datasets`:

```
from sklearn.datasets import fetch_mldata
X, y = mnist["data"], mnist["target"]
```

Однако в новых версиях пакета `sklearn.datasets` функция `fetch_mldata` по всей видимости снята с эксплуатации, а вместо нее используется новая функция `fetch_openml` с параметрами. См. **https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_openml.html#sklearn.datasets.fetch_openml**.

```
X_mnist, y_mnist = fetch_openml('mnist_784', version=1,
                                return_X_y=True)
X, y = X_mnist, y_mnist.astype(float)
```

Вместе с тем новый файл данных MNIST из этого источника несколько отличается, в частности по порядку следования образцов. Для того чтобы получить точно такие

же результаты, что и в книге, ниже приведено обходное техническое решение с извлечением данных из репозитория Github функцией `workaround`:

```
from six.moves import urllib
from scipy.io import loadmat

def workaround():
  mnist_alternative_url =
   ("https://github.com/amplab/datascience-sp14/"
    "raw/master/lab7/mldata/mnist-original.mat")
    mnist_path = "./mnist-original.mat"

    response = urllib.request.urlopen(mnist_alternative_url)

    with open(mnist_path, "wb") as f:
        content = response.read()
        f.write(content)

    mnist_raw = loadmat(mnist_path)
    mnist = {
        "data": mnist_raw["data"].T,
        "target": mnist_raw["label"][0],
        "COL_NAMES": ["label", "data"],
        "DESCR": "mldata.org dataset: mnist-original",
    }
    return mnist
```

Таким образом, имеется три варианта работы с данными MNIST.

Вариант 1 (вероятно, уже не поддерживаемый):

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X, y = mnist["data"], mnist["target"]
```

Вариант 2 (обновленный):

```
from sklearn.datasets import fetch_openml
X_mnist, y_mnist = fetch_openml('mnist_784', version=1,
                                return_X_y=True)
X, y = X_mnist, y_mnist.astype(float)
```

Вариант 3 (обходной):

```
mnist = workaround()
X, y = mnist["data"], mnist["target"]
```

Как обычно, сначала давайте, используя стандартный способ, импортируем необходимую библиотеку.

```
from sklearn.datasets import fetch_openml  # функция fetch_mldata устарела
```

Затем загрузим данные.

```
X_mnist, y_mnist = fetch_openml('mnist_784', version=1 , return_X_y=True)
X, y = X_mnist, y_mnist.astype(float)
```

Либо с использованием обходного технического решения, что в данном случае предпочтительнее, вызовем функцию `workaround`:

```
mnist = workaround()
X, y = mnist["data"], mnist["target"]
```

Теперь `X` содержит входные изображения, а `y` — целевые метки (напомним, что на жаргоне машинного обучения значение, которое мы хотим предсказать, называется целью, `target`). Просто набрав `X.shape`, вы получите форму `X`: (70000, 784). Обратите внимание, что `X` имеет 70 тыс. строк (каждая строка является изображением) и 784 столбца (каждый столбец является признаком, или в нашем случае значением пиксела с оттенком серого). Давайте проверим, сколько цифр у нас в наборе данных.

```
for i in range(10):
    print ("цифра", i, "появляется", np.count_nonzero(y == i), "раз")
```

В результате получим следующее:

```
цифра 0 появляется 6903 раз
цифра 1 появляется 7877 раз
цифра 2 появляется 6990 раз
цифра 3 появляется 7141 раз
цифра 4 появляется 6824 раз
цифра 5 появляется 6313 раз
цифра 6 появляется 6876 раз
цифра 7 появляется 7293 раз
цифра 8 появляется 6825 раз
цифра 9 появляется 6958 раз
```

Полезно определить функцию визуализации цифр для того, чтобы иметь представление о том, как они выглядят.

```
def plot_digit(some_digit):
    some_digit_image = some_digit.reshape(28,28)

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

Например, мы можем показать одну из них, выбрав любой индекс (рис. 2.17).

```
plot_digit(X[36003])
```

Модель, которую мы хотим здесь реализовать, представляет собой простую логистическую регрессию для бинарной классификации, поэтому набор данных должен

быть сведен к двум классам или, в данном случае, к двум цифрам. Мы выбираем единицы и двойки. Давайте извлечем из набора данных только те изображения, которые представляют 1 или 2. Наш нейрон попытается определить принадлежность данного изображения классу 0 (цифра 1) либо классу 1 (цифра 2).

```
X_train = X[np.any([y == 1,y == 2], axis = 0)]
y_train = y[np.any([y == 1,y == 2], axis = 0)]
```

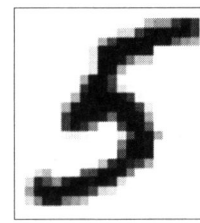


РИС. 2.17. 36003-я цифра в наборе данных. Она легко узнаваема, как 5

Далее входные наблюдения должны быть нормализованы. (Напомним, что при использовании сигмоидальной активационной функции вы не хотите, чтобы ваши входные данные были слишком большими, потому что у вас их 784.)

```
X_train_normalised = X_train/255.0
```

Мы выбрали 255, потому что каждый объект является значением пиксела с оттенком серого в изображении, а уровни серого в исходных изображениях находятся в интервале от 0 до 255. Позже в книге будет подробно рассмотрена причина, почему нужно нормализовывать входные признаки. А пока просто поверьте, что этот шаг необходим. В каждом столбце мы хотим иметь входное наблюдение, и каждая строка должна представлять признак (значение пиксела с оттенком серого), поэтому мы должны реформировать тензоры:

```
X_train_tr = X_train_normalised.transpose()
y_train_tr = y_train.reshape(1,y_train.shape[0])
```

и мы можем задать переменную `n_dim`, назначив ей число признаков:

```
n_dim = X_train_tr.shape[0]
```

Сейчас наступает очень важный момент. Метки в импортированном наборе данных будут равны 1 либо 2 (они просто указывают на то, какая цифра представлена изображением). Однако мы будем строить стоимостную функцию, исходя из допущения, что наши метки равны 0 и 1, поэтому мы должны перешкалировать массив `y_train_tr`.

```
y_train_shifted = y_train_tr - 1
```

ПРИМЕЧАНИЕ. При выполнении бинарной классификации не забудьте проверить значения меток, которые вы используете для тренировки модели. Иногда использование неправильных меток (не 0 и 1) приводит к довольно большим временны́м издержкам на то, чтобы понять причину, почему модель не работает.

Теперь все изображения, представляющие 1, будут иметь метку 0, и все изображения, представляющие 2, — метку 1. Наконец, давайте применим имена собственные для Python-переменных.

```
Xtrain = X_train_tr
ytrain = y_train_shifted
```

На рис. 2.18 показано несколько цифр, с которыми мы имеем дело.

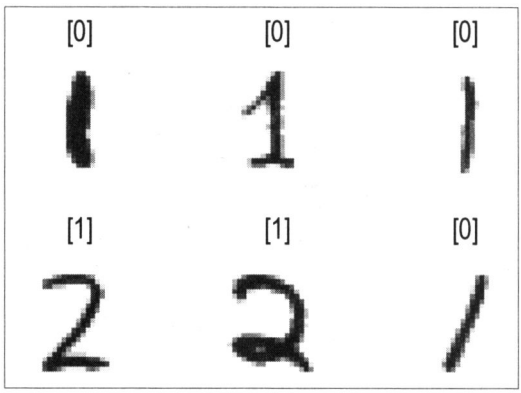


РИС. 2.18. Шесть случайных цифр, отобранных из набора данных. Относительные перешкалированные метки (метки в нашем наборе данных теперь равны 0 или 1) приведены в скобках

Реализация в TensorFlow

Реализация в TensorFlow не является сложной и почти такая же, как и для линейной регрессии. Сначала определим заполнители и переменные.

```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n_dim]))
b = tf.Variable(tf.zeros(1))

init = tf.global_variables_initializer()
```

Обратите внимание, что исходный код тот же, что мы использовали для линейной регрессионной модели. Однако мы должны определить другую стоимостную функцию (как обсуждалось ранее) и другой выход нейрона (сигмоидальную функцию).

```
y_ = tf.sigmoid(tf.matmul(W,X)+b)
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Мы использовали сигмоидальную функцию для выхода нейрона, используя `tf.sigmoid()`. Исходный код, который будет выполнять модель, совпадает с тем, который мы использовали для линейной регрессии. Мы изменили только имя функции.

```
def run_logistic_model(learning_r, training_epochs, train_obs, train_labels,
debug = False):
  sess = tf.Session()
  sess.run(init)

  cost_history = np.empty(shape=[0], dtype = float)

  for epoch in range(training_epochs+1):
    sess.run(training_step, feed_dict = {X: train_obs, Y: train_labels,
        learning_rate: learning_r})
    cost_ = sess.run(cost, feed_dict={X:train_obs, Y: train_labels,
        learning_rate: learning_r})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 500 == 0) & debug:
       print("Достигнута эпоха",epoch,"стоимость J =",
            str.format('{0:.6f}', cost_))

  return sess, cost_history
```

Давайте выполним модель и посмотрим на результаты. Мы решили начать с темпа заучивания 0.01.

```
sess, cost_history = run_logistic_model(learning_r=0.01,
                                        training_epochs=5000,
                                        train_obs=Xtrain,
                                        train_labels=ytrain,
                                        debug=True)
```

Результат работы модели (остановленной после 3000 эпох) будет следующим:

```
Достигнута эпоха 0 стоимость от J = 0.678598
Достигнута эпоха 500 стоимость J = 0.108655
Достигнута эпоха 1000 стоимость J = 0.078912
Достигнута эпоха 1500 стоимость J = 0.066786
Достигнута эпоха 2000 стоимость J = 0.059914
Достигнута эпоха 2500 стоимость J = 0.055372
Достигнута эпоха 3000 стоимость J = nan
```

Что случилось? Внезапно, в какой-то момент, наша стоимостная функция принимает значение `nan` (не число). Похоже, что после определенного момента модель не очень хорошо работает.

Если темп заучивания является слишком большим, либо вы неправильно инициализируете веса, то ваши значения для $\hat{y}^{(i)} = P\left(y^{(i)} = 1 \mid x\right)$ могут сильно приблизить-

ся к нулю или единице (сигмоидальная функция принимает значения очень близкие к 0 или 1 для очень больших отрицательных или положительных значений z). Напомним, что в стоимостной функции имеются два члена — `tf.log(y_)` и `tf.log(1-y_)`, и что логарифмическая функция (log) для нулевого значения не определена. Поэтому если `y_` будет равно 0 или 1, то вы будете получать `nan`, потому что исходный код попытается вычислить `tf.log(0)`. В качестве примера можно привести модель с темпом заучивания 2,0. После только одной эпохи вы уже получите значение стоимостной функции, равное `nan`. И легко понять причину, почему это происходит, если распечатать значение b до и после первого тренировочного шага. Для этого просто видоизмените исходный код модели и примените следующую ниже версию:

```
def run_logistic_model(learning_r, training_epochs, train_obs, train_labels,
                                                                debug = False):
    sess = tf.Session()
    sess.run(init)

    cost_history = np.empty(shape=[0], dtype = float)

    for epoch in range(training_epochs+1):
        print (эпоха: ', epoch)
        print(sess.run(b, feed_dict={X:train_obs, Y: train_labels,
                                     learning_rate: learning_r}))

        sess.run(training_step, feed_dict = {X: train_obs,
                                             Y: train_labels,
                                             learning_rate: learning_r})
        print(sess.run(b, feed_dict={X:train_obs, Y: train_labels,
                                     learning_rate: learning_r}))

        cost_ = sess.run(cost, feed_dict={X:train_obs, Y: train_labels,
                                          learning_rate: learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % 500 == 0) & debug:
            print("Достигнута эпоха",epoch,"стоимость J =",
                  str.format('{0:.6f}', cost_))

    return sess, cost_history
```

Вы получите следующий результат (после остановки тренировки всего через одну эпоху):

```
эпоха: 0
[ 0.]
[-0.05966223]
Достигнута эпоха 0 стоимость J = nan
```

```
эпоха: 1
[-0.05966223]
[ nan]
```

Вы видите, как b идет от 0 к –0,05966223, а затем к nan? Следовательно, $\mathbf{z} = \mathbf{w}^T\mathbf{X} + \mathbf{b}$ превращается в nan, затем $\mathbf{y} = \sigma(z)$ тоже превращается в nan, и, наконец, целевая функция, будучи функцией от $\mathbf{y}$, в результате также будет равна nan. Это происходит лишь потому, что темп заучивания является чересчур большим.

Каким будет решение? Вы должны попробовать другой (читай: гораздо меньший) темп заучивания.

Давайте попробуем и посмотрим, сможем ли мы получить более стабильный результат после 2500 эпох. Мы запускаем модель теперь с вызовом на 5000 эпох, который выглядит так:

```
sess, cost_history = run_logistic_model(learning_r=0.005,
                                        training_epochs=5000,
                                        train_obs=Xtrain,
                                        train_labels=ytrain,
                                        debug=True)
```

В результате исполнения данной инструкции получим:

```
Достигнута эпоха 0 стоимость от J = 0.685799
Достигнута эпоха 500 стоимость J = 0.154386
Достигнута эпоха 1000 стоимость J = 0.108590
Достигнута эпоха 1500 стоимость J = 0.089566
Достигнута эпоха 2000 стоимость J = 0.078767
Достигнута эпоха 2500 стоимость J = 0.071669
Достигнута эпоха 3000 стоимость J = 0.066580
Достигнута эпоха 3500 стоимость J = 0.062715
Достигнута эпоха 4000 стоимость J = 0.059656
Достигнута эпоха 4500 стоимость J = 0.057158
Достигнута эпоха 5000 стоимость J = 0.055069
```

Результат больше не содержит nan. График стоимостной функции показан на рис. 2.19. Для того чтобы оценить нашу модель, мы должны выбрать оптимизационную метрику (как обсуждалось ранее). Для задачи бинарной классификации классической метрикой является точность (обозначаемая буквой a), которая может пониматься как мера разницы между результатом и его "истинным" значением. Математически точность можно рассчитать как

$$a = \frac{\text{число правильно идентифицированных случаев}}{\text{суммарное число случаев}}.$$

Для получения точности мы выполним следующий фрагмент кода. (Напомним, что мы относим наблюдение i к классу 0, если $P\left(y^{(i)} = 1 \mid x^{(i)}\right) < 0,5$, либо к классу 1, если $P\left(y^{(i)} = 1 \mid x^{(i)}\right) > 0,5$.)

```
correct_prediction1 = tf.equal(tf.greater(y_, 0.5), tf.equal(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction1, tf.float32))
print(sess.run(accuracy, feed_dict={X:Xtrain, Y:ytrain, learning_rate:0.05}))
```

С этой моделью мы достигаем точности 98,6%. Неплохо для сети с одним нейроном.

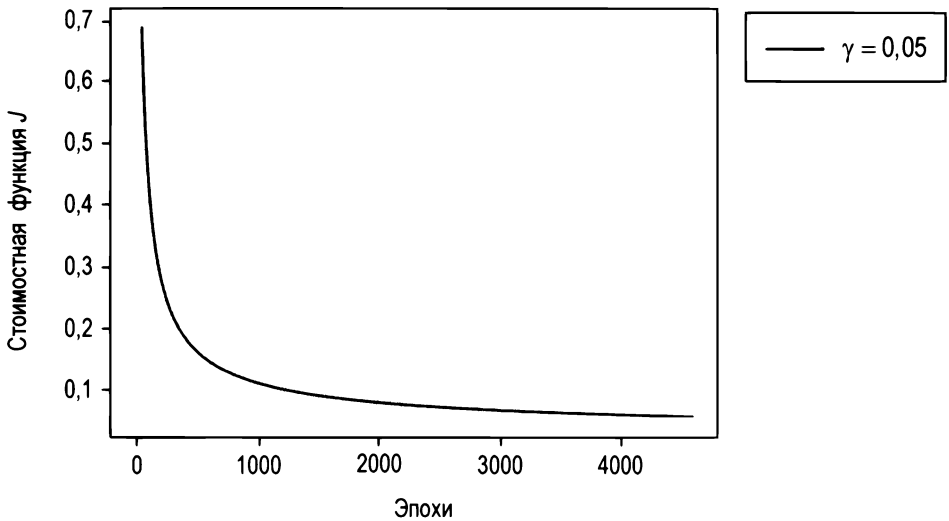


РИС. 2.19. Стоимостная функция в сопоставлении с эпохами для темпа заучивания 0,005

Можно также попробовать выполнить приведенную выше модель (с темпом заучивания 0,005) для большего числа эпох. Вы обнаружите, что примерно при 7000 эпохах значение `nan` появится снова. Решение здесь будет состоять в сокращении темпа заучивания вместе с увеличением числа эпох. Простой подход, такой как сокращение темпа заучивания вдвое каждые 500 эпох, позволит избавиться от значений `nan`. Аналогичный подход будет рассмотрен подробнее далее в книге.

Ссылки на литературу

1. Jeremy Hsu. Biggest Neural Network Ever Pushes AI Deep Learning (Крупнейшая нейронная сеть в мире дает толчок развитию глубокого обучения ИИ). URL: https://spectrum.ieee.org/tech-talk/computing/software/biggest-neural-network-ever-pushes-ai-deeplearning, (2015).
2. Raúl Rojas. Neural Networks: A Systematic Introduction (Нейронные сети: систематическое введение). — Berlin: Springer-Verlag, 1996.
3. Delve (Data for Evaluating Learning in Valid Experiments). The Boston Housing Dataset (Бостонский набор данных с ценами на жилую недвижимость). URL: www.cs.toronto.edu/~delve/data/boston/bostonDetail.html (1996).

4. Prajit Ramachandran, Barret Zoph, Quoc V. Le. Searching for Activation Functions (В поисках активационных функций). arXiv:1710.05941 [cs.NE], 2017.

5. Guido F. Montufar, Razvan Pascanu, Kyunghyun Cho, Yoshua Bengio. On the Number of Linear Regions of Deep Neural Networks (О числе линейных участков глубоких нейронных сетей). URL: https://papers.nips.cc/paper/5422-on-the-numberof-linear-regions-of-deep-neural-networks.pdf (2014).

6. Brendan Fortuner. Can Neural Networks Solve Any Problem? (Могут ли нейронные сети решать какие-либо задачи?). URL: https://towardsdatascience.com/can-neural-networksreally-learn-any-function-65e106617fc6 (2017).

ГЛАВА 3

Нейронные сети прямого распространения

В *главе 2* мы проделали несколько удивительных вещей с одним-единственным нейроном, но этот подход едва ли будет в достаточной мере гибким для решения более сложных случаев. Реальная мощь нейронных сетей проявляется только тогда, когда для решения конкретной задачи друг с другом взаимодействуют несколько (тысяч и даже миллионов) нейронов. Сетевая архитектура (т. е. каким образом нейроны друг с другом связаны, как они себя ведут и т. д.) играет решающую роль в том, насколько эффективно сеть способна автоматически обучаться, насколько хороши ее предсказания и какие задачи она может решать.

Существует много видов архитектур, и все они были широко изучены и являются очень сложными, но с точки зрения поставленной учебной задачи важно начать с самого простого вида нейронной сети с несколькими нейронами. Имеет смысл начать с так называемых нейронных сетей прямого распространения, в которых данные поступают во входной слой и проходят через сеть послойно до тех пор, пока они не дойдут до выходного слоя. (Отсюда и название таких сетей: нейронные сети прямого распространения.) В этой главе мы рассмотрим сети, в которых каждый нейрон в слое получает свой вход от всех нейронов из предыдущего слоя и подает свой выход в каждый нейрон следующего слоя.

Как легко представить, вместе с увеличением сложности возникает больше проблем. Становится труднее достичь быстрого заучивания и хорошей точности; число имеющихся гиперпараметров растет из-за повышенной сложности сети; и простой алгоритм градиентного спуска больше не будет его сокращать при работе с большими наборами данных. Во время разработки моделей с большим числом нейронов нам потребуется иметь в своем распоряжении расширенный набор инструментов, которые позволят справляться со всеми проблемами, которые представляют эти сети. В этой главе мы рассмотрим несколько более продвинутых методов и алгоритмов, которые позволят эффективно работать с большими наборами данных и большими сетями. Такие сложные сети становятся достаточно хорошими для того,

чтобы решать некоторые интересные задачи из категории мультиклассовой классификации. Эта категория задач является одной из самых распространенных из тех, которые обязаны выполнять большие сети (например, распознавание рукописного ввода, распознавание лиц, распознавание изображений и т. д.). В связи с этим был подобран специальный набор данных, который позволит провести ряд интересных мультиклассовых классификаций и изучить трудности данной категории задач.

Данная глава начнется с обсуждения сетевой архитектуры и необходимого матричного формализма. Затем будет предоставлен краткий обзор новых гиперпараметров, которые привносятся этим новым типом сети. Далее будут даны пояснения относительно того, как реализовывать многоклассовую классификацию с помощью функции softmax и какой требуется выходной слой. Затем, прежде чем приступить к программированию на Python, будет сделано краткое отступление для того, чтобы на простом примере подробно объяснить понятие переподгонки и показать процедуру проведения простого анализа ошибок при использовании сложных сетей. Затем мы начнем использовать TensorFlow для построения больших сетей, применяя их к MNIST-подобному набору данных, основанному на изображениях предметов одежды (что будет очень забавно). Мы рассмотрим способы ускорения алгоритма градиентного спуска, описанного в *главе 2*, введя два новых его варианта: стохастический и мини-пакетный градиентный спуск. Затем мы рассмотрим способы эффективного добавления многочисленных слоев и наилучшей инициализации весов и смещения с целью обеспечения быстрой и стабильной тренировки сети. В частности, мы рассмотрим инициализацию Ксавье и Хе (Xavier и He) соответственно для сигмоидальной активационной функции и активационной функции ReLU. Наконец, будет предложено эмпирическое правило относительно того, как сравнивать сложность сетей, выходящих за рамки только числа нейронов. Данная глава завершится несколькими советами по выбору правильных сетей.

Сетевая архитектура

Нейросетевую архитектуру понять довольно просто. Она состоит из входного слоя (входов $x_j^{(i)}$), нескольких слоев (именуемых скрытыми, потому что они зажаты между входным и выходным слоями, поэтому они, как бы, "невидимы" снаружи) и выходного слоя. В каждом слое может быть от одного до нескольких нейронов. Основное свойство такой сети состоит в том, что каждый нейрон получает входы от каждого нейрона в предыдущем слое и передает свои выходы каждому нейрону в следующем слое. На рис. 3.1 показано графическое представление такой сети.

Скачок от одного нейрона, как в *главе 2*, к этой сетевой архитектуре является довольно большим. Для того чтобы построить модель, нам придется работать с матричным формализмом, и, следовательно, мы должны правильно получить все матричные размерности. Сначала поясним новые обозначения:

- L — число скрытых слоев, исключая входной слой, но включая выходной слой;
- n_l — число нейронов в слое l.

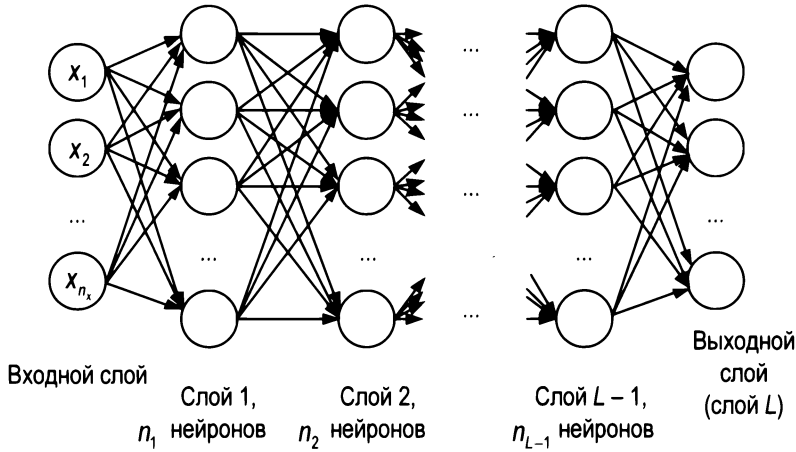


РИС. 3.1. Диаграмма многослойной глубокой нейронной сети прямого распространения, в которой каждый нейрон получает входы от каждого нейрона в предыдущем слое и передает свои выходы каждому нейрону в последующем слое

В такой сети, как на рис. 3.1, мы будем обозначать суммарное число нейронов через $N_{\text{нейронов}}$, которое можно записать как

$$N_{\text{нейронов}} = n_x + \sum_{i=1}^{L} n_i = \sum_{i=0}^{L} n_i ,$$

где по определению мы задали $n_0 = n_x$. Каждая связь между двумя нейронами имеет свой вес. Обозначим вес между нейроном i в слое l и нейроном j в слое $l-1$ через $w_{ij}^{[1]}$. На рис. 3.2 нарисованы только первые два слоя (входной слой и слой 1) обобщенной сети из рис. 3.1, где веса расположены между первым нейроном во входном слое и всеми остальными в слое 1. Все остальные нейроны для ясности показаны серым цветом.

Веса между входным слоем и слоем 1 можно записать в виде матрицы следующим образом:

$$\mathbf{W}^{[1]} = \begin{bmatrix} w_{11}^{[1]} & \cdots & w_{1n_x}^{[1]} \\ \vdots & \ddots & \vdots \\ w_{n_1 1}^{[1]} & \cdots & w_{n_1 n_x}^{[1]} \end{bmatrix}.$$

Это означает, что матрица $\mathbf{W}^{[1]}$ имеет размерность $n_1 \times n_x$. Конечно, это можно обобщить между любыми двумя слоями l и $l-1$, т. е. весовая матрица между двумя соседними слоями l и $l-1$, обозначенная через $\mathbf{W}^{[l]}$, будет иметь размерность $n_l \times n_{l-1}$. По определению $n_0 = n_x$ является числом входных признаков (а не числом наблюдений, которое мы обозначаем через m).

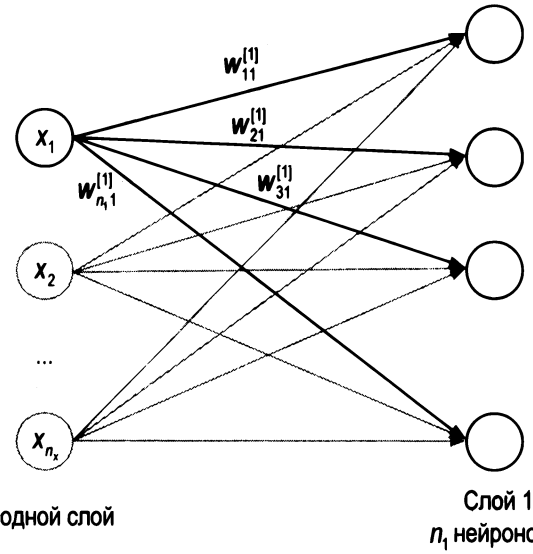


РИС. 3.2. Первые два слоя обобщенной нейронной сети с весами связей между первым нейроном во входном слое и другими в первом слое. Все остальные нейроны и связи показаны светло-серым цветом для того, чтобы сделать диаграмму более четкой

ПРИМЕЧАНИЕ. Весовая матрица между двумя соседними слоями l и $l-1$, которую мы обозначаем через $\mathbf{W}^{[l]}$, будет иметь размерность $n_l \times n_{l-1}$, где по определению $n_0 = n_x$ является числом входных признаков.

Смещение (обозначенное в *главе 2* как b) в этот раз будет матрицей. Напомним, что каждый нейрон, который получает входы, будет иметь собственное смещение, и поэтому при рассмотрении двух слоев, l и $l-1$, нам потребуется n_l разных значений b. Мы обозначим эту матрицу через $\mathbf{b}^{[l]}$, и она будет иметь размерность $n_l \times 1$.

ПРИМЕЧАНИЕ. Матрица смещений для двух соседних слоев l и $l-1$, которую мы обозначаем через $\mathbf{b}^{[l]}$, будет иметь размерность $n_l \times 1$.

Выход нейронов

Теперь приступим к рассмотрению выхода из нейронов. Для начала возьмем i-й нейрон первого слоя (напомним, что входной слой по определению является слоем 0). Обозначим его выход через $\hat{y}_i^{[1]}$ и допустим, что все нейроны в слое l используют одинаковую активационную функцию, которую мы обозначим через $g^{[l]}$. Тогда у нас будет

$$\hat{y}_i^{[1]} = g^{[1]}\left(z_i^{[1]}\right) = g^{[l]}\left(\sum_{j=1}^{n_x}\left(w_{ij}^{[1]}x_j + b_i^{[1]}\right)\right),$$

где, как вы помните из *главы 2*, мы обозначили z_i как

$$z_i^{[1]} = \sum_{i=1}^{n_x} \left(w_{ij}^{[1]} x_j + b_i^{[1]} \right).$$

Как вы понимаете, мы хотим иметь матрицу для всех выходов из слоя 1, поэтому мы будем использовать обозначение

$$\mathbf{Z}^{[1]} = \mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]},$$

где $\mathbf{Z}^{[1]}$ будет иметь размерность $n_1 \times 1$; $\mathbf{X}$ — матрица со всеми наблюдениями (строки для признаков и столбцы для наблюдений), как уже отмечалось в *главе 2*. Здесь мы исходим из того, что все нейроны в слое l будут использовать одинаковую активационную функцию, которую мы обозначим через $g^{[l]}$.

Мы можем легко обобщить предыдущее уравнение для слоя l:

$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]}\mathbf{Z}^{[l-1]} + \mathbf{b}^{[l]},$$

потому что слой l получит свой вход от слоя $l-1$. Нам просто нужно заменить $\mathbf{X}$ на $\mathbf{Z}^{[l-1]}$. $\mathbf{Z}^{[l]}$ будет иметь размерность $n_l \times 1$. Выход в матричной форме будет равен

$$\mathbf{Y}^{[l]} = g^{[l]}\left(\mathbf{Z}^{[l]}\right),$$

где активационная функция действует, как обычно, поэлементно.

Сводка матричных размерностей

Далее представлены размерности всех матриц, которые мы описали к этому моменту:

- $\mathbf{W}^{[l]}$ имеет размерность $n_l \times n_{l-1}$ (где по определению $n_0 = n_x$);
- $\mathbf{b}^{[l]}$ имеет размерность $n_l \times 1$;
- $\mathbf{Z}^{[l-1]}$ имеет размерность $n_{l-1} \times 1$;
- $\mathbf{Z}^{[l]}$ имеет размерность $n_l \times 1$;
- $\mathbf{Y}^{[l]}$ имеет размерность $n_l \times 1$.

В каждом случае l изменяется в интервале от 1 до L.

Пример: уравнения для сети с тремя слоями

Для того чтобы сделать наше обсуждение немного конкретнее, рассмотрим пример сети с тремя слоями (поэтому $L = 3$), где $n_1 = 3$, $n_2 = 2$ и $n_3 = 1$ (рис. 3.3).

В этом случае нам придется рассчитать следующие величины:

- $\hat{\mathbf{Y}}^{[1]} = g^{[1]}\left(\mathbf{W}^{[1]}\mathbf{X} + \mathbf{b}^{[1]}\right)$, где $\mathbf{W}^{[1]}$ имеет размерность $3 \times n_x$, $\mathbf{b}^{[1]}$ — размерность 3×1, $\mathbf{X}$ — размерность $n_x \times m$;

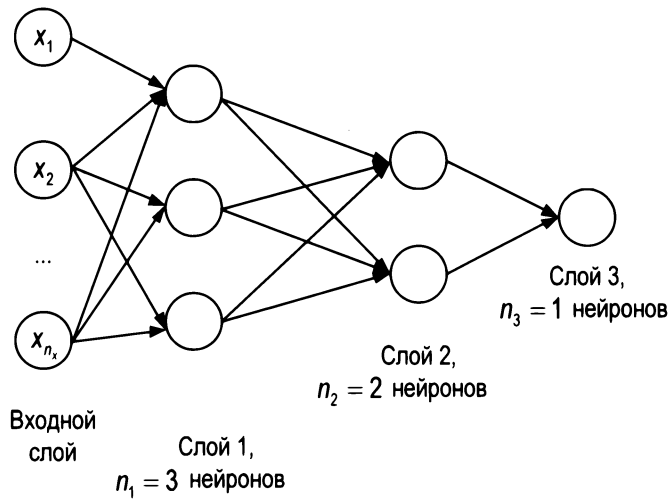


РИС. 3.3. Практический пример нейронной сети прямого распространения

- $\hat{\mathbf{Y}}^{[2]} = g^{[2]}\left(\mathbf{W}^{[2]}\mathbf{Z}^{[1]} + \mathbf{b}^{[2]}\right)$, где $\mathbf{W}^{[2]}$ имеет размерность 2×3, $\mathbf{b}^{[2]}$ — размерность 2×1, $\mathbf{Z}^{[1]}$ — размерность $3 \times m$;
- $\hat{\mathbf{Y}}^{[3]} = g^{[3]}\left(\mathbf{W}^{[3]}\mathbf{Z}^{[2]} + \mathbf{b}^{[3]}\right)$, где $\mathbf{W}^{[3]}$ имеет размерность 1×2, $\mathbf{b}^{[3]}$ — размерность 1×1, $\mathbf{Z}^{[2]}$ — размерность $2 \times m$;

и сетевой выход $\mathbf{Z}^{[3]}$ будет иметь, как и ожидалось, размерность $1 \times m$.

Все это может показаться довольно абстрактным (и, собственно, так оно и есть). Позже в этой главе вы увидите, каким образом это легко реализуется в TensorFlow, просто путем построения правильного вычислительного графа с упором на только что рассмотренные шаги.

Гиперпараметры в полносвязных сетях

В сетях, подобных только что рассмотренным, существует довольно много параметров, которые можно отрегулировать для отыскания наилучшей модели для вашей задачи. Из *главы 2* вы помните, что параметры, которые вы фиксируете в начале, а затем не изменяете в тренировочной фазе, называются гиперпараметрами. Во время работы с сетями прямого распространения вам придется заниматься регулировкой следующих дополнительных гиперпараметров:

- число слоев — L;
- число нейронов в каждом слое — n_i для $i = 1, ..., L$;
- подбор активационной функции для каждого слоя — $g^{[l]}$;

И, конечно же, у вас по-прежнему остаются следующие два гиперпараметра, с которыми вы столкнулись в *главе 2*:

- число итераций (или эпох);
- темп заучивания.

Функция softmax для многоклассовой классификации

Прежде чем приступить к написанию исходного кода TensorFlow, вам не избежать еще немного теории. Описываемые в этой главе виды сетей будут усложняться по мере того, как вы будете учиться выполнять многоклассовую классификацию с разумными результатами. Для этого необходимо сначала ввести активационную функцию softmax.

Математически говоря, функция softmax S — это функция, которая преобразовывает k-мерный вектор вещественных значений в другой k-мерный вектор, каждый элемент которого изменяется в интервале от 0 до 1, а в сумме дают 1. С учетом k вещественных значений z_i для $i = 1, ..., k$ мы определяем вектор $\mathbf{z} = [z_1 \ z_2 \ ... \ z_k]$ и векторную функцию softmax $S(\mathbf{z}) = [S(z_1) \ S(z_2) \ ... \ S(z_k)]$ как

$$S(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

Поскольку знаменатель всегда больше, чем числитель, $S(z)_i < 1$. Кроме того, мы имеем:

$$\sum_{i=1}^{k} S(z)_i = \sum_{i=1}^{k} \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} = \frac{\sum_{i=1}^{k} e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} = 1.$$

Таким образом, $S(z)_i$ ведет себя как вероятность, потому что ее сумма над i равна 1, а все ее элементы меньше 1. Рассмотрим $S(z)_i$ как распределение вероятностей по k возможным исходам. Для нас $S(z)_i$ будет вероятностью того, что входное наблюдение принадлежит к классу i.

Предположим, мы пытаемся классифицировать наблюдение на три класса. Мы можем получить следующие выходы: $S(z)_1 = 0{,}1$, $S(z)_2 = 0{,}6$ и $S(z)_3 = 0{,}3$. Иными словами, наше наблюдение имеет 10%-ю вероятность принадлежать к классу 1, 60%-ю вероятность принадлежать к классу 2 и 30%-ю вероятность принадлежать к классу 3. Обычно принято относить к определенному классу входное наблюдение с более высокой вероятностью, в данном примере к классу 2 с вероятностью 60%.

ПРИМЕЧАНИЕ. Мы будем рассматривать $S(z)_i$ как распределение вероятностей по k с $i = 1, ..., k$ возможными исходами. Для нас $S(z)_i$ представляет собой вероятность того, что входное наблюдение принадлежит к классу i.

Для того чтобы иметь возможность применять функцию softmax для классификации, мы должны использовать специфический выходной слой. Нам придется использовать десять нейронов, каждый из которых будет давать z_i на выходе, а затем еще один нейрон, который будет выводить $S(z)$. У этого нейрона будет активационная функция softmax, и он будет иметь на входе 10 выходов z_i последнего слоя с десятью нейронами. В TensorFlow для этих целей используется функция `tf.nn.softmax`, которая применяется к последнему слою с десятью нейронами. Напомним, что эта функция TensorFlow будет действовать поэлементно. Далее в этой главе вы найдете конкретный пример, показывающий, каким образом это реализуется от начала до конца.

Краткое отступление: переподгонка

Одной из самых распространенных проблем, с которой вы столкнетесь во время тренировки глубоких нейронных сетей, будет переподгонка. Может случиться так, что ваша сеть, в силу своей гибкости, заучит закономерности, которые вызваны шумом, ошибками или просто неправильными данными. Очень важно с самого начала разобраться в том, что такое переподгонка, поэтому здесь приведем практический пример того, что может произойти, дав вам интуитивное понимание этого феномена. Для того чтобы облегчить визуализацию, мы будем работать с простым двухмерным набором данных, который создадим специально для этой цели. Будем надеяться, что в конце следующего раздела у вас сложится четкое представление о том, что такое переподгонка.

Практический пример переподгонки

Сети, описанные в предыдущих разделах, являются довольно сложными и могут легко привести к чрезмерно плотной подгонке к набору данных, т. е. переподгонке. Приведем краткое объяснение понятия переподгонки. Для ее понимания рассмотрим следующую задачу: найти наилучший многочлен, который аппроксимирует некий набор данных. С учетом множества двумерных точек ($x^{(i)}$, $y^{(i)}$) мы хотим найти наилучший многочлен степени K в форме

$$f\left(x^{(i)}\right) = \sum_{j=0}^{K} a_j x^{(i)j},$$

который минимизирует среднеквадратическую ошибку

$$\frac{1}{m}\sum_{j=0}^{m}\left(y^{(i)} - f\left(x^{(i)}\right)\right)^2,$$

где, как обычно, m обозначает число имеющихся точек данных. Требуется не только определить все параметры a_j, но и значение K, которое лучше всего аппроксимирует данные. K в этом случае измеряет сложность модели. Например, для $K = 0$ мы просто имеем $f\left(x^{(i)}\right) = a_0$ (константа), простейший многочлен, который только

можно придумать. Для более высокого K у нас есть многочлены более высокой степени, т. е. наша функция становится сложнее, имея больше предназначенных для тренировки параметров.

Приведем пример функции для $K = 3$:

$$f\left(x^{(i)}\right) = \sum_{j=0}^{K} a_j x^{(i)j} = a_0 + a_1 x^{(i)} + a_2 \left(x^{(i)}\right)^2 + a_3 \left(x^{(i)}\right)^3,$$

где у нас четыре параметра, которые могут быть отрегулированы во время тренировки моделей. Давайте сгенерируем немного данных, начав с многочлена второй степени ($K = 2$)

$$1 + 2x^{(i)} + 3\left(x^{(i)}\right)^2$$

и добавив некую случайную ошибку (это сделает переподгонку заметной). Сначала импортируем стандартные библиотеки с добавлением функции `curve_fit`, которая будет автоматически минимизировать стандартную ошибку и отыскивать наилучшие параметры. Не слишком переживайте по поводу этой функции. Наша цель здесь состоит в том, чтобы показать, что может произойти при использовании слишком сложной модели.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
```

Определим функцию для многочлена второй степени

```
def func_2(p, a, b, c):
    return a+b*p + c*p**2
```

и затем сгенерируем набор данных

```
x = np.arange(-5.0, 5.0, 0.05, dtype = np.float64)
y = func_2(x, 1,2,3)+18.0*np.random.normal(0, 1, size=len(x))
```

Для того чтобы добавить в функцию немного случайного шума, мы воспользовались функцией `np.random.normal(0, 1, size=len(x))`, которая генерирует массив NumPy случайных значений из нормального распределения длины `len(x)` с нулевым средним и единичным стандартным отклонением.

На рис. 3.4 показан внешний вид данных для $a = 1$, $b = 2$ и $c = 3$.

Теперь давайте рассмотрим модель, слишком простую для того, чтобы уловить признак данных, т. е. мы увидим то, к чему приводит модель с высоким смещением[1]. Рассмотрим линейную модель ($K = 1$). Исходный код будет таким:

```
def func_1(p, a, b):
    return a+b*p
popt, pcov = curve_fit(func_1, x, y)
```

[1] Смещение — это мера ошибки, возникающей в моделях, которые слишком просты для того, чтобы улавливать реальные признаки данных.

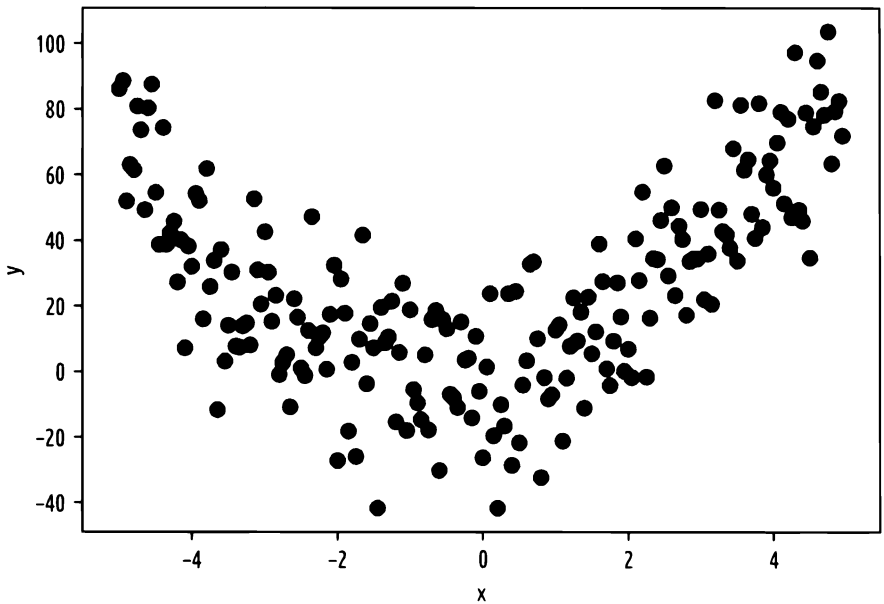


РИС. 3.4. Данные, сгенерированные с $a = 1$, $b = 2$ и $c = 3$, как описано в тексте

В результате получим наилучшие значения для a и b, которые минимизируют стандартную ошибку. На рис. 3.5 вы видите, как эта модель, будучи слишком простой, совершенно упускает главный признак данных.

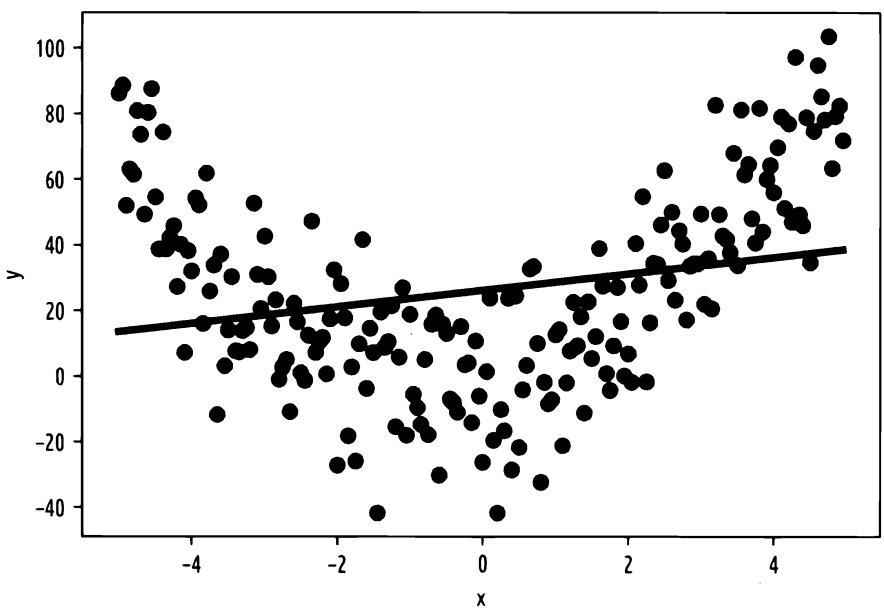


РИС. 3.5. Линейная модель упускает главный признак данных, будучи слишком простой. В этом случае модель имеет высокое смещение

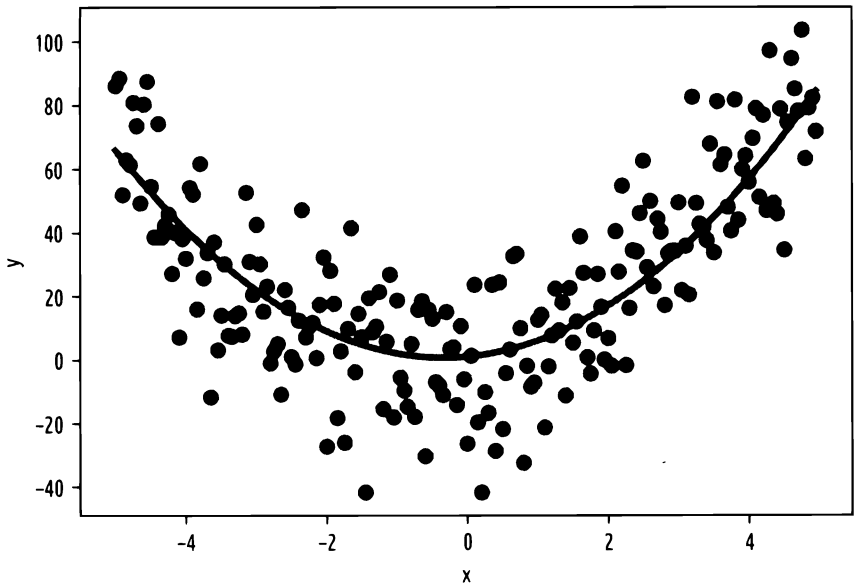


РИС. 3.6. Результаты для многочлена второй степени

Попробуем выполнить подгонку многочлена второй степени ($K = 2$). Результаты представлены на рис. 3.6.

Уже лучше. Эта модель, похоже, улавливает главный признак данных, игнорируя случайный шум. Теперь попробуем очень сложную модель — многочлен 21-й степени ($K = 21$). Результаты показаны на рис. 3.7.

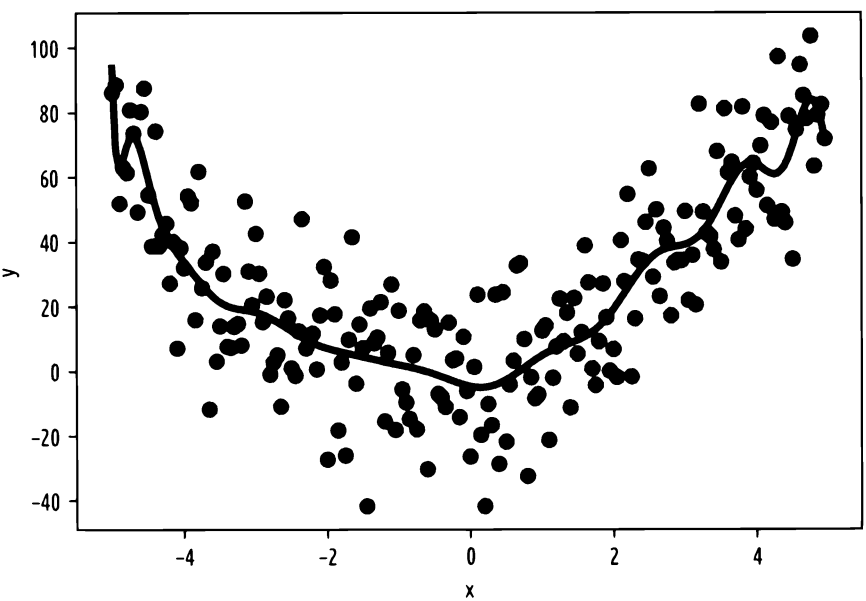


РИС. 3.7. Результаты для модели на основе многочлена 21-й степени

Теперь эта модель показывает признаки, которые, как мы знаем, неверны, потому что данные были созданы нами. Этих признаков нет, но модель настолько гибка, что фиксирует случайную изменчивость, которую мы ввели с помощью шума. Под изменчивостью здесь имеются в виду осцилляции, которые появились в результате использования многочлена такой высокой степени.

В этом случае мы говорим о переподгонке, т. е. нашей моделью мы начинаем улавливать признаки, например, из-за случайной ошибки. Легко понять, что она обобщает довольно плохо. Если мы применим эту 21-степенную многочленную модель к новым данным, то она не будет работать хорошо, потому что в новых данных случайный шум будет отличаться, и поэтому осцилляции, которые мы видим на рис. 3.7, в новых данных не будут иметь смысла. На рис. 3.8 показан график лучших 21-степенных многочленных моделей, полученных путем подгонки к данным, сгенерированным с добавлением 10 разных значений случайного шума. Мы ясно видим, насколько они варьируются. Они нестабильны и сильно зависят от случайного шума. Осцилляции всегда разные! В данном случае речь идет о высокой дисперсии.

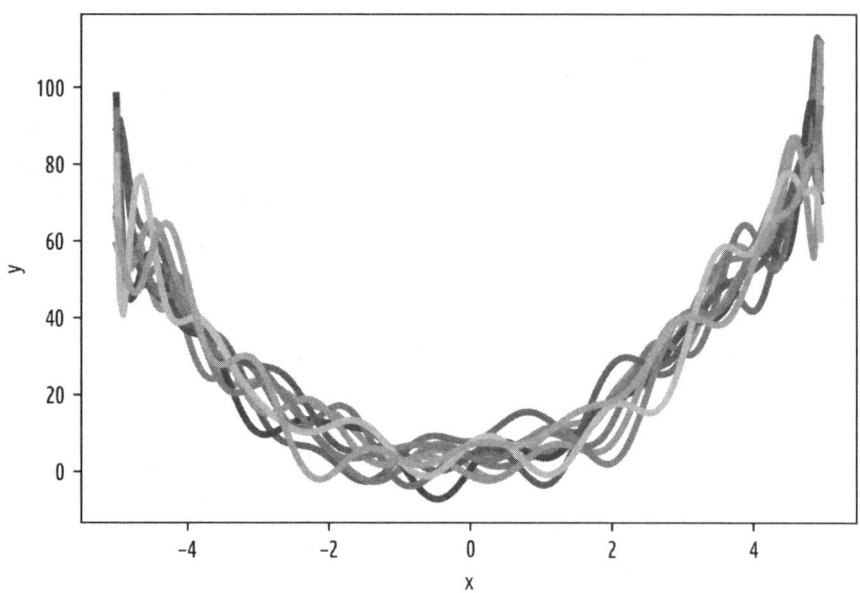


РИС. 3.8. Результат моделей с многочленом 21-й степени, подогнанных к 10 разным наборам данных, сгенерированных с добавлением разных значений случайного шума

Теперь давайте создадим тот же график с линейной моделью, варьируя случайный шум, как мы сделали на рис. 3.8. Вы можете проверить результаты на рис. 3.9.

Вы видите, что наша модель стала гораздо стабильнее. Линейная модель не улавливает никаких признаков, зависящих от шума, но пропускает главный признак наших данных (вогнутую природу). Здесь мы говорим о высоком смещении.

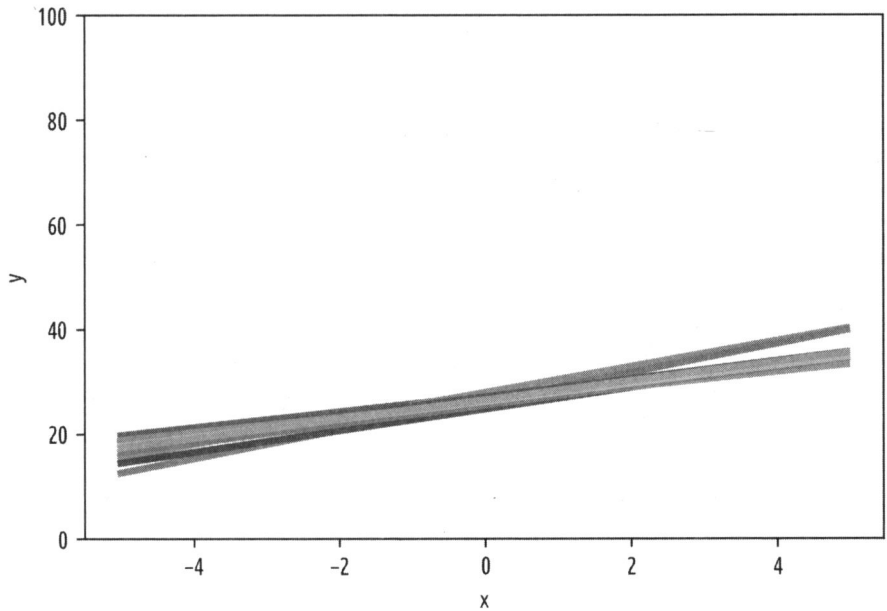


РИС. 3.9. Результаты линейной модели применительно к данным, в которых мы произвольно варьировали случайный шум. Для простоты сравнения с рис. 3.8 использовалась та же самая шкала

Рисунок 3.10 поможет вам получить интуитивное представление о смещении и дисперсии. Смещение — это мера того, насколько близки наши измерения к истинным значениям (центру рисунка), а дисперсия — это мера того, насколько измерения разбросаны вокруг среднего значения (не обязательно истинного значения, как можно увидеть справа).

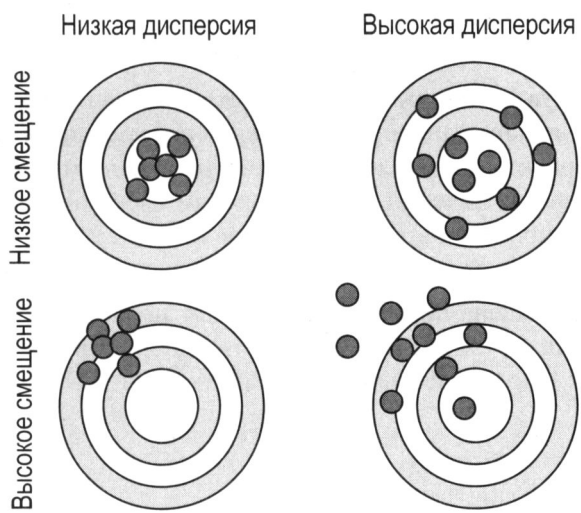


РИС. 3.10. Смещение и дисперсия

В нейронных сетях у нас много гиперпараметров (число слоев, число нейронов в каждом слое, активационная функция и т. д.), и очень трудно понять, в каком режиме мы находимся. Например, как отличить, имеет ли наша модель высокую дисперсию или высокое смещение? Этой теме будет посвящена целая глава, но первый шаг в выполнении этого анализа ошибок состоит в том, чтобы разбить набор данных на два разных. Давайте посмотрим, что это значит и почему мы это делаем.

ПРИМЕЧАНИЕ. Суть переподгонки состоит в том, что невольно извлекается некоторая остаточная дисперсия (т. е. шум), как если бы эта дисперсия представляла собой базовую структуру модели (см. Burnham, K. P., Anderson, D. R. Model Selection and Multimodel Inference[2]. — 2nd ed. — New York; Springer-Verlag, 2002). Противоположный термин называется недоподгонкой — когда модель не способна улавливать структуру данных.

Проблема с переподгонкой и глубокими нейронными сетями заключается в том, что нет способа легко визуализировать результаты, и поэтому для определения того, является ли наша модель переподогнанной, недоподогнанной или же правильной, нам нужен другой подход. Это можно достичь путем разбиения набора данных на разные части и оценивания и сравнения метрик по всем из них. В следующем разделе мы рассмотрим основную идею.

Простой анализ ошибок

Для того чтобы проверить результативность работы нашей модели и сделать правильный анализ ошибок, мы должны разбить набор данных на следующие две части[3].

- **Тренировочный набор данных.** На этом наборе данных модель тренируют, используя входные данные и относительные метки, с помощью алгоритма-оптимизатора, такого как градиентный спуск, как мы делали в *главе 2*. Часто этот набор называется "тренировочным подмножеством".
- **Рабочий (или валидационный) набор.** Натренированная модель будет использоваться на этом наборе данных для проверки ее работы. На этом наборе данных мы будем тестировать разные гиперпараметры. Например, мы можем натренировать две различные модели с разным числом слоев на тренировочном наборе данных и протестировать их на этом наборе данных с целью проверить результативность работы. Часто этот набор называется "рабочим подмножеством".

Анализу ошибок посвящена целая глава, но будет неплохой идеей предложить его обзор с пояснениями, почему важно разбивать набор данных на части. Предположим, что мы имеем дело с классификацией, и что метрический показатель, который

[2] Отбор модели и мультимодельный вывод.

[3] Для проведения правильного анализа ошибок нам потребуется, по крайней мере, три части, возможно, четыре. Но чтобы получить базовое понимание процесса, достаточно двух частей.

мы используем для оценки результативности модели, равен 1 минус точность, или, другими словами, процентное соотношение случаев, когда данные классифицированы неправильно. Рассмотрим следующие три случая (табл. 3.1).

Таблица 3.1. *Четыре разных случая, показывающих, как распознавать переподгонку из ошибки тренировочного и рабочего наборов*

Ошибка	Случай A	Случай B	Случай C	Случай D
Ошибка из тренировочного набора, %	1	15	14	0,3
Ошибка из рабочего набора, %	11	16	32	1,1

- Случай A: здесь у нас переподгонка (высокая дисперсия), потому что мы очень хорошо справляемся с тренировочным набором, но наша модель очень плохо обобщает на рабочем наборе (см. рис. 3.8).
- Случай B: здесь мы видим проблему с высоким смещением, т. е. наша модель не очень хорошо работает на обоих наборах данных (см. рис. 3.9).
- Случай C: здесь мы имеем высокое смещение (модель не очень хорошо предсказывает тренировочный набор) и высокую дисперсию (модель не обобщает достаточно хорошо на рабочем наборе).
- Случай D: здесь все, кажется, в порядке. Ошибка достаточно хороша на тренировочном наборе и также хороша на рабочем наборе. Это хороший кандидат на то, чтобы стать наилучшей моделью.

Позже все эти понятия будут объяснены подробнее, где будут приведены рецепты решения проблем высокого смещения, высоких дисперсий, и того и другого или даже более сложных случаев.

Резюмируя, отметим, что для выполнения очень простого анализа ошибок вам придется разбить набор данных по крайней мере на два подмножества: тренировочный и рабочий наборы данных. Затем следует вычислить метрику для обоих наборов и сравнить их. Цель — добиться модели с низкой ошибкой на тренировочном и рабочем наборах (как в случае D в предыдущем примере), и эти два значения должны быть сопоставимыми.

ПРИМЕЧАНИЕ. Из этого раздела вы должны вынести два ключевых момента: во-первых, для понимания результативности работы модели (переподогнана, недоподогнана или правильная) требуется набор рецептов и ориентиров; во-вторых, для ответа на предыдущие вопросы необходимо разбить набор данных на две части, что даст возможность выполнить релевантный анализ. Далее в книге вы увидите, что можно сделать с набором данных, разбитым на три или даже четыре части.

Набор данных Zalando

Zalando SE — это немецкая компания электронной коммерции, базирующаяся в Берлине. Компания поддерживает кроссплатформенный магазин, который торгует обувью, одеждой и другими модными вещами[4]. Для конкурса kaggle (если вы не знаете, что это такое, посетите веб-сайт **www.kaggle.com**, где вы можете принять участие во многих конкурсах, цель которых — решение задач науки о данных) в компании Zalando подготовили MNIST-подобный набор изображений их одежды, для которого они предоставили 60 тыс. тренировочных изображений и 10 тыс. тестовых изображений. Как и в наборе данных MNIST, каждое изображение имело 28×28-пиксельный формат в оттенках серого. Разработчики компании Zalando сгруппировали все изображения в десять классов и предоставили метки для каждого изображения. Набор данных содержит 785 столбцов. Первый столбец — это метка класса (целое число от 0 до 9), и оставшиеся 784 содержат значение пиксела изображения с оттенком серого (вы можете проверить: $28 \times 28 = 784$), точно так же, как и в *главе 2*, где мы рассматривали родственный набор данных MNIST рукописных цифр.

И тренировочный, и тестовый образец (согласно документации) получает одну из следующих меток:

- 0 — футболка/топ;
- 1 — брюки;
- 2 — пуловер;
- 3 — платье;
- 4 — куртка;
- 5 — сандалии;
- 6 — рубашка;
- 7 — кроссовки;
- 8 — сумка;
- 9 — полусапоги.

На рис. 3.11 показан пример каждого класса, случайно выбранного из набора данных.

Набор данных был предоставлен согласно лицензии MIT[5]. Файл данных можно скачать с веб-сайта kaggle (**www.kaggle.com/zalando-research/fashionmnist/data**) или непосредственно из GitHub (**https://github.com/zalandoresearch/fashion-mnist**). Если вы выберете второй вариант, то вам придется немного повозиться с подготовкой данных. (Их можно преобразовать в CSV с помощью скрипта, расположенного по адресу **https://pjreddie.com/projects/mnist-in-csv/**.) Если вы скачаете его из веб-сайта kaggle, то данные уже будут в правильном формате. На веб-сайте kaggle вы найдете два CSV-файла. После распаковки у вас будет файл fashion-mnist_train.csv с 60 тыс. изображений (примерно 130 Мбайт) и файл fashion-mnist_test.csv с 10 тыс. изображений (примерно 21 Мбайт). Давайте запустим блокнот Jupyter и начнем программировать!

---

[4] Википедия, "Zalando", **https://en.wikipedia.org/wiki/Zalando**.

[5] Википедия, "Лицензия MIT", **https://en.wikipedia.org/wiki/MIT_License**, 2018.

РИС. 3.11. Один пример каждого из десяти классов в наборе данных Zalando

В нашем исходном коде понадобятся следующие инструкции импорта:

```
import pandas as pd
import numpy as np
import tensorflow as tf
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
from random import *
```

Поместите CSV-файлы в тот же каталог, что и записная книжка. Затем вы можете загрузить файлы с помощью функции библиотеки pandas.

```
data_train = pd.read_csv('fashion-mnist_train.csv', header = 0)
```

Вы также можете прочитать файл с помощью стандартных функций NumPy (таких как `loadtxt()`), но использование функции `read_csv()` библиотеки pandas дает более высокую гибкость в нарезке и анализе данных. Кроме того, она намного быстрее. Чтение файла (т. е. примерно 130 Мбайт) с помощью pandas занимает около 10 секунд, в то время как с помощью NumPy на моем ноутбуке это занимает 1 минуту

20 секунд. Поэтому, если вы занимаетесь большими наборами данных, имейте это в виду. Использование библиотеки pandas для чтения и подготовки данных — обычная практика. Если вы не знакомы с pandas, то не переживайте. Все, что вам нужно понять, будет объяснено подробно.

ПРИМЕЧАНИЕ. Напомним, что не следует сосредотачивать свое внимание на деталях реализации на языке Python. Сосредоточьтесь на модели, на концепциях, лежащих в основе реализации. Вы можете достичь тех же результатов, используя pandas, Numpy или даже C. Попробуйте сосредоточиться на том, как подготовить данные, как нормализовать их, как проверить результаты тренировки и т. д.

С помощью инструкции

```
data_train.head()
```

можно увидеть первые пять строк набора данных (рис. 3.12).

```
In [16]: 1 data_train.head()
Out[16]:
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0

5 rows × 785 columns

РИС. 3.12. С помощью инструкции `data_train.head()` можно проверить первые пять строк набора данных

Вы увидите, что у каждого столбца есть имя. Библиотека pandas извлекает его из первой строки файла. Проверив имя столбца, вы сразу узнаете, что и в каком столбце находится. Например, в первом столбце у нас находится метка класса. Теперь мы должны создать массив с метками и массив с 784 признаками (напомним, что в качестве признаков у нас все пиксельные значения оттенков серого). Для этого можно написать следующее:

```
labels = data_train['label'].values.reshape(1, 60000)
train = data_train.drop('label', axis=1).transpose()
```

Давайте кратко остановимся на том, что именно данный фрагмент кода делает, начиная с меток. В pandas каждый столбец имеет имя (как вы видите на рис. 3.12), которое в нашем случае автоматически выводится из первой строки CSV-файла. Первый столбец ("label") содержит метку класса, целое число от 0 до 9. В pandas для выбора только этого столбца можно использовать следующую синтаксическую конструкцию:

```
data_train['label']
```

предоставив имя столбца в квадратных скобках.

Если проверить форму этого массива с помощью

```
data_train['label'].shape
```

то, как и ожидалось, вы получите значение `(60000)`. Как мы уже видели в *главе 2*, нам нужен тензор для меток с размерностью $1 \times m$, где m — это число наблюдений (в данном случае `60000`). Поэтому мы должны его реформировать с помощью инструкции

```
labels = data_train['label'].values.reshape(1, 60000)
```

Теперь тензорные метки имеют размерность (1, 60 000), как мы и хотим.

Тензор, который должен содержать признаки, должен содержать все столбцы, кроме меток. Поэтому мы просто удаляем столбец label, используя для этого метод `drop('label', axis=1)`, берем все остальное, а затем транспонируем тензор. На самом деле, `data_train.drop('label', axis=1)` имеет размерность (60 000, 784), а мы хотим тензор с размерностью $n_x \times m$, где $n_x = 784$ — это число признаков. Ниже приводится сводная информация о наших тензорах на данный момент.

- **Метки.** Массив меток имеет размерность $1 \times m$ $(1 \times 60\,000)$ и содержит метки классов (целые числа от 0 до 9).
- **Тренировочные данные** имеют размерность $n_x \times m$ $(784 \times 60\,000)$ и содержат признаки, в которых каждая строка хранит значение оттенка серого для одного пиксела на изображении (напомним, $28 \times 28 = 784$).

Взгляните еще раз на рис. 3.11 для получения представления о том, как выглядят изображения. Наконец, давайте нормализуем входные данные для того, чтобы вместо значений в интервале от 0 до 255 (значения оттенков серого) они имели значения только в интервале от 0 до 1. Это очень легко сделать с помощью следующей инструкции:

```
train = np.array(train / 255.0)
```

Построение модели с помощью TensorFlow

Теперь самое время расширить то, что мы делали с TensorFlow в *главе 2* с одним-единственным нейроном, до сетей с многочисленными слоями и нейронами. Для начала рассмотрим сетевую архитектуру и выясним, какой выходной слой нам нужен, а затем построим модель с помощью TensorFlow.

Сетевая архитектура

Начнем с сети с одним скрытым слоем. У нас будет входной слой с 784 признаками, далее скрытый слой (в котором мы будем варьировать число нейронов), затем выходной слой из десяти нейронов, который будет запитывать их выход в нейрон с активационной функцией softmax. Взгляните сначала на рис. 3.13 с графическим

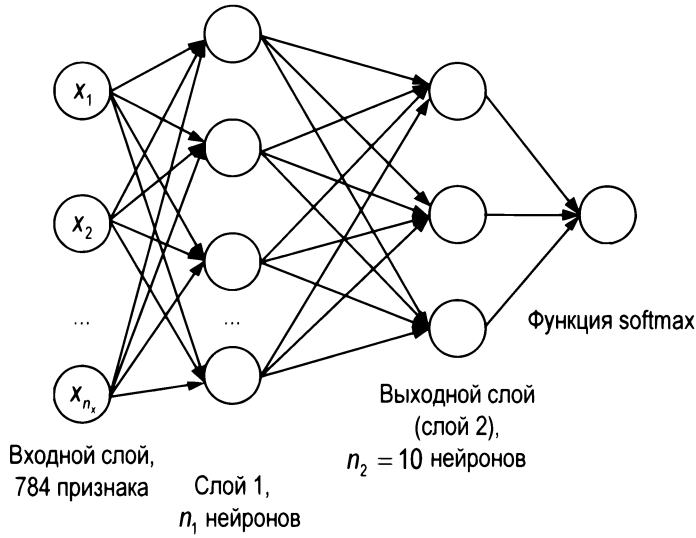


РИС. 3.13. Архитектура сети с одним скрытым слоем. Во время нашего анализа мы будем варьировать число нейронов n_1 в скрытом слое

представлением сети. Затем будут даны пояснения по поводу различных частей, в особенности выходных слоев.

Следует объяснить, почему существует этот странный выходной слой с десятью нейронами и почему существует потребность в дополнительном нейроне для функции softmax. Напомним, что мы хотим иметь возможность определять, к какому классу принадлежит каждое изображение. Для этого, как пояснялось при рассмотрении функции softmax, нам придется сделать 10 выходов для каждого наблюдения: каждый из них является вероятностью того, что изображение принадлежит к одному из классов. Поэтому с учетом входа $\mathbf{x}^{(i)}$ нам понадобятся 10 значений: $P\left(y^{(i)}=1 \mid \mathbf{x}^{(i)}\right)$, $P\left(y^{(i)}=2 \mid \mathbf{x}^{(i)}\right)$, ..., $P\left(y^{(i)}=10 \mid \mathbf{x}^{(i)}\right)$ (вероятность того, что класс наблюдения $y^{(i)}$ является одной из десяти возможностей при заданном входе $\mathbf{x}^{(i)}$), или, другими словами, наш выход должен быть тензором размерности 1×10 в форме

$$\hat{\mathbf{y}} = \left[P\left(y^{(i)}=1 \mid \mathbf{x}^{(i)}\right) \; P\left(y^{(i)}=2 \mid \mathbf{x}^{(i)}\right) \; \ldots \; P\left(y^{(i)}=10 \mid \mathbf{x}^{(i)}\right) \right].$$

И поскольку наблюдение должно принадлежать единственному классу, условие

$$\sum_{j=1}^{10} P\left(y^{(i)}=j \mid \mathbf{x}^{(i)}\right) = 1$$

должно быть удовлетворено. Это можно понять следующим образом: наблюдение имеет 100%-ю вероятность принадлежать к одному из десяти классов, или, другими словами, все вероятности в сумме должны давать 1. Мы решаем эту задачу в два этапа:

1. Создаем выходной слой с десятью нейронами. Благодаря этому на выходе мы получим десять значений.
2. Затем подаем десять значений в новый нейрон (назовем его нейроном "softmax"), который возьмет десять входов и даст на выходе десять значений, которые все меньше 1 и в сумме дают 1.

На рис. 3.14 подробно показан нейрон "softmax".

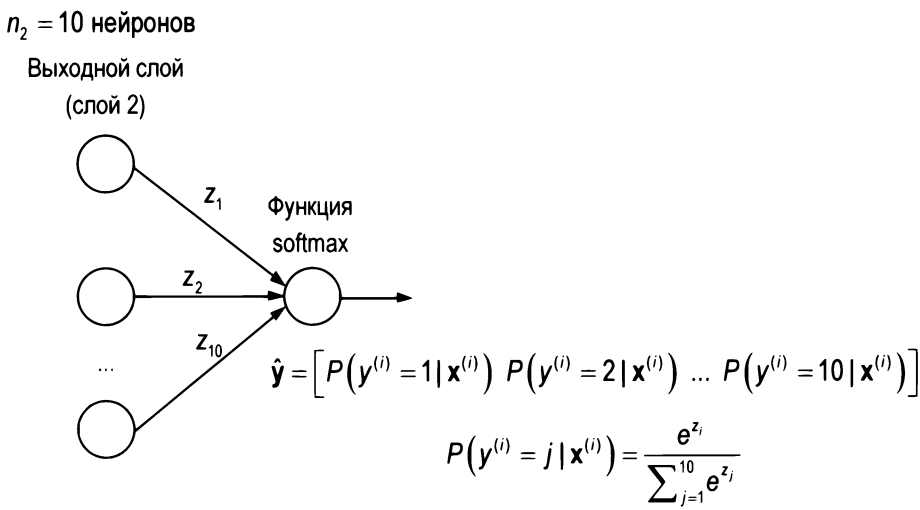


РИС. 3.14. Последний нейрон в сети, преобразующий десять входов в вероятности

Называя z_i выходом i-го нейрона в последнем слое (где i меняется в интервале от 1 до 10), мы будем иметь

$$P\left(y^{(i)} = j \mid \mathbf{x}^{(i)}\right) = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}}.$$

Это именно то, что делает функция TensorFlow `tf.nn.softmax()`. Она принимает на входе тензор и возвращает тензор той же размерности, что и вход, но "нормализованный", как обсуждалось ранее. Другими словами, если мы подадим в функцию $\mathbf{z} = [z_1 \; z_2 \; \ldots \; z_{10}]$, то она вернет тензор той же размерности, что и $\mathbf{z}$, т. е. 1×10, где каждый элемент является приведенным выше уравнением.

Модификация меток для функции softmax — кодировка с одним активным состоянием

Прежде чем развить нашу сеть, мы должны решить еще одну проблему. Из *главы 2* вы помните, что при классифицировании мы будем использовать следующую стоимостную функцию:

```
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
```

где `Y` содержит метки, а `y_` — результат сети. Поэтому два тензора должны иметь одинаковые размерности. В нашем случае сеть дает на выходе вектор с десятью элементами, в то время как метка в наборе данных является просто скаляром. Следовательно, `y_` имеет размерность (10, 1), а `Y` — размерность (1, 1). Это не сработает, если мы не сделаем что-то умное. Мы должны преобразовать метки в тензор, который имеет размерность (10, 1). Также требуется вектор со значением для каждого класса. Но какое значение мы должны использовать?

Мы должны сделать то, что называется *"кодированием с одним активным состоянием"*[6, 7]. Под ним подразумевается, что мы преобразовываем метки (целые числа от 0 до 9) в тензоры с размерностью (1, 10) по следующему алгоритму: вектор, кодированный с одним активным состоянием, будет иметь все нули, кроме индекса метки. Например, для метки 2 тензор 1×10 будет иметь все нули, кроме позиции индекса 2, или, другими словами, он будет (0, 0, 1, 0, 0, 0, 0, 0, 0, 0). Посмотрите другие примеры (табл. 3.2), и данная идея сразу станет ясной.

Таблица 3.2. *Примеры работы кодирования с одним активным состоянием (напомним, что метки находятся на интервале от 0 до 9 как индексы)*

Метка	Метка, кодированная с одним активным состоянием
0	(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
2	(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)
5	(0, 0, 0, 0, 0, 1, 0, 0, 0, 0)
7	(0, 0, 0, 0, 0, 0, 0, 1, 0, 0)

На рис. 3.15 показано графическое представление процесса кодирования метки с одним активным состоянием. Здесь две метки (2 и 5) кодированы с одним активным состоянием в два тензора. Элемент тензора с серым фоном (в данном случае одномерного вектора) становится единицей, а элементы с белым фоном остаются нулями.

Библиотека Scikit-learn предоставляет несколько способов сделать это автоматически (попробуйте, например, функцию `OneHotEncoder()`), но, полагаем, что поучительно провести этот процесс вручную, чтобы действительно увидеть, как это делается. Разобравшись в том, зачем вам это нужно и в каком формате вам это нужно, вы можете применять ту функцию, которая вам понравится больше всего. Python-код для этого кодирования очень прост (последняя строка преобразует кадр данных pandas в массив NumPy):

---

[6] В качестве примечания, это техническое решение часто используется для подачи категориальных переменных в машинно-обучающиеся алгоритмы.

[7] Словосочетание *кодирование с одним активным состоянием* (англ. one-hot encoding) пришло из терминологии цифровых интегральных микросхем, где оно описывает конфигурацию микросхемы, в которой допускается, чтобы только один бит был положительным (активным). — *Прим. пер.*

```
labels_ = np.zeros((60000, 10))
labels_[np.arange(60000), labels] = 1
labels_ = labels_.transpose()
labels_ = np.array(labels_)
```

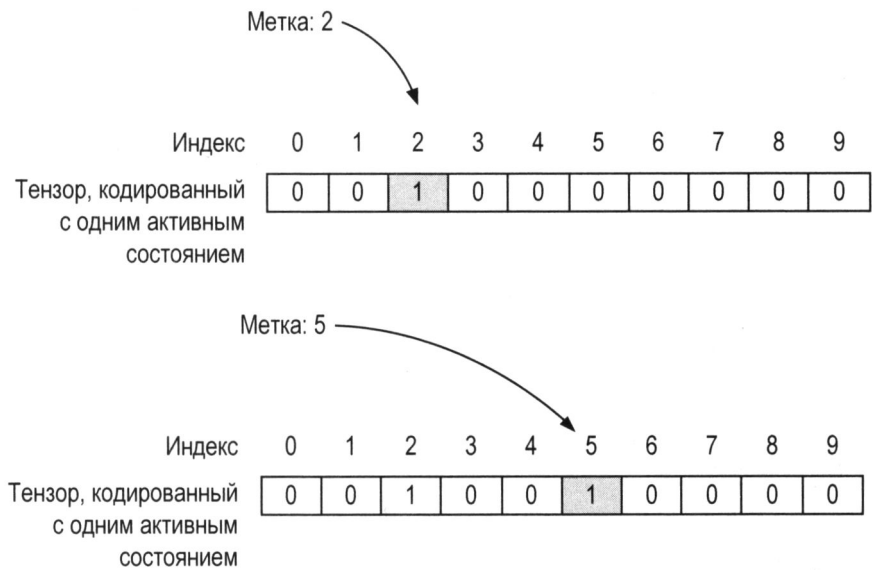


РИС. 3.15. Графическое представление процесса кодирования метки с одним активным состоянием

Сначала создается новый массив с правильной размерностью: (60 000, 10), затем он заполняется нулями с помощью функции `np.zeros((60000,10))`. Далее, назначается 1 только тем столбцам, которые относятся к метке, посредством функциональных возможностей библиотеки pandas по нарезке кадров: `labels_[np.arange(60000), labels] = 1`. И наконец, массив транспонируется для получения нужной нам размерности: (10, 60000), где каждый столбец указывает на другое наблюдение.

Теперь в исходном коде можно, наконец, сравнить `Y` и `y_`, потому что оба теперь имеют размерность (10, 1) для одного наблюдения или, если рассматривать весь тренировочный набор данных, (10, 60 000). Каждая строка в `y_` теперь будет представлять вероятность того, что наше наблюдение относится к определенному классу. В самом конце, при расчете точности модели, мы будем назначать каждому наблюдению класс с наибольшей вероятностью.

ПРИМЕЧАНИЕ. Нейросеть даст нам десять вероятностей того, что наблюдение принадлежит к одному из десяти классов. В конце мы назначим наблюдению класс, который имеет наибольшую вероятность.

Модель TensorFlow

Теперь самое время построить модель с помощью TensorFlow. Следующий ниже фрагмент кода выполнит эту работу:

```
n_dim = 784
tf.reset_default_graph()

# Число нейронов в слоях
n1 = 5  # Число нейронов в слое 1
n2 = 10 # Число нейронов в выходном слое

cost_history = np.empty(shape=[1], dtype = float)
learning_rate = tf.placeholder(tf.float32, shape=())

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [10, None])
W1 = tf.Variable(tf. truncated_normal ([n1, n_dim], stddev=.1))
b1 = tf.Variable(tf.zeros([n1,1]))
W2 = tf.Variable(tf. truncated_normal ([n2, n1], stddev=.1))
b2 = tf.Variable(tf.zeros([n2,1]))

# Построим нашу сеть...
Z1 = tf.nn.relu(tf.matmul(W1, X) + b1)
Z2 = tf.nn.relu(tf.matmul(W2, Z1) + b2)
y_ = tf.nn.softmax(Z2,0)

cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()
```

Мы не будем перебирать каждую строку кода, потому что вы уже должны понимать, что такое заполнитель или переменная. Однако на несколько деталей исходного кода стоило бы обратить внимание.

- При инициализации весов используется код `tf.Variable(tf.truncated_normal([n1, n_dim], stddev=.1))`. Функция `truncated_normal()` будет возвращать значения из нормального распределения с той особенностью, что значения, которые больше чем на 2 стандартных отклонения от среднего, будут отброшены и отобраны заново. Причина выбора небольшого `stddev`, равного 0,1, состоит в том, чтобы избежать слишком большого выхода активационной функции ReLU, что в результате приведет к появлению `nan`, поскольку Python не в состоянии правильно вычислять слишком большие числа. Позже в этой главе будет рассмотрен более оптимальный способ выбора правильного `stddev`.
- Последний нейрон будет использовать активационную функцию softmax: `y_=tf.nn.softmax(Z2,0)`. Напомним, что `y_` будет не скаляром, а тензором той же

размерности, что и `Z2`. Второй параметр, `0`, сообщает TensorFlow о том, что мы хотим применить функцию softmax вдоль вертикальной оси (строк).

- Два параметра, `n1` и `n2`, определяют число нейронов в разных слоях. Напомним: для того чтобы иметь возможность использовать функцию softmax, второй (выходной) слой должен иметь десять нейронов. Но мы поиграем со значением `n1`. Увеличение `n1` усложнит сеть.

Теперь давайте попробуем выполнить тренировку, как мы делали в *главе 2*. Мы можем повторно применить уже написанный код. Попробуйте выполнить следующий исходный код на ноутбуке:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

training_epochs = 5000
cost_history = []

for epoch in range(training_epochs+1):
    sess.run(optimizer, feed_dict={X:train, Y:labels_, learning_rate:
    0.001})
    cost_ = sess.run(cost, feed_dict={X:train, Y:labels_, learning_rate:
    0.001})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 20 == 0):
        print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Вы должны сразу заметить одну вещь: он очень медленный. Если у вас нет очень мощного процессора или не установлен TensorFlow с поддержкой GPU, и у вас нет мощной графической карты, то на ноутбуке 2017 года этот код займет несколько часов (от пары до нескольких, в зависимости от оборудования, которое у вас есть). Проблема заключается в том, что модель в том виде, в каком мы ее запрограммировали, создаст для всех наблюдений (т. е. 60 000) огромную матрицу, а затем будет модифицировать веса и смещение только после полного прохода по всем наблюдениям. Для этого требуется довольно много ресурсов памяти и процессора. Если бы это был наш единственный вариант, то мы были бы обречены. Имейте в виду, что в сфере глубокого обучения 60 000 примеров из 784 признаков — это совсем не большой набор данных. Поэтому мы должны найти способ дать нашей модели обучиться быстрее.

Последний необходимый вам фрагмент кода можно использовать для вычисления точности модели. Вы легко можете сделать это с помощью следующих строк:

```
correct_predictions = tf.equal(tf.argmax(y_,0), tf.argmax(Y,0))
accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"))
print ("Точность:", accuracy.eval({X:train, Y:labels_, learning_rate:0.001},
                                                        session = sess))
```

Функция `tf.argmax()` возвращает индекс с наибольшим значением по осям тензора. Как вы помните, при обсуждении функции softmax было отмечено, что мы будем относить наблюдение к классу, который имеет наибольшую вероятность (`y_` является тензором с десятью значениями, каждое из которых содержит вероятность того, что наблюдение принадлежит одному из классов). Поэтому `tf.argmax(y_,0)` даст нам наиболее вероятный класс для каждого наблюдения. `tf.argmax(Y,0)` сделает то же самое для наших меток. Напомним, что мы закодировали наши метки, используя одно активное состояние, так что, например, класс 2 теперь будет (0, 0, 2, 0, 0, 0, 0, 0, 0). Следовательно, `tf.argmax([0,0,2,0,0,0,0,0,0],0)` вернет 2 (индекс с наибольшим значением, в данном случае единственным отличающимся от нуля).

Ранее было показано, как загрузить и подготовить тренировочный набор данных. Для того чтобы выполнить простой анализ ошибок, вам также понадобится тестовый набор данных. Ниже приведен код, который вы можете использовать. Этот исходный код не требует разъяснений, т. к. он в точности такой же, как тот, который мы использовали для тренировочного набора данных.

```
data_dev = pd.read_csv('fashion-mnist_test.csv', header = 0)
labels_dev = data_test['label'].values.reshape(1, 10000)

labels_dev_ = np.zeros((10000, 10))
labels_dev_[np.arange(10000), labels_dev] = 1
labels_dev_ = labels_dev_.transpose()

dev = data_dev.drop('label', axis=1).transpose()
```

Не смущайтесь тем, что имя файла содержит слово "test". Иногда рабочий набор данных (dev) называется тестовым набором данных. Далее в книге при обсуждении анализа ошибок мы будем использовать три набора данных: тренировочный (train), рабочий (dev) и тестовый (test). Для того чтобы оставаться последовательным на протяжении всей книги, предпочтение будет отдаваться имени "рабочий" (dev), чтобы не путать вас разными именами в разных главах.

Наконец, для того чтобы вычислить точность на рабочем наборе данных, вы просто повторно используете тот же исходный код, который был предоставлен ранее.

```
correct_predictions = tf.equal(tf.argmax(y_,0), tf.argmax(Y,0))
accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"))
print ("Точность:", accuracy.eval({X:dev, Y:labels_dev_, learning_rate:0.001},
                                                    session = sess))
```

Хорошим упражнением было бы попробовать включить его вычисление в модель, благодаря чему функция `model()` смогла бы автоматически возвращать два значения.

Варианты градиентного спуска

В *главе 2* было дано описание очень простого алгоритма градиентного спуска (также именуемого пакетным градиентным спуском). Этот алгоритм не является самым умным способом найти минимальную стоимостную функцию. Давайте посмотрим на варианты, которые вам нужно знать, и сравним их эффективность, используя набор данных Zalando.

Пакетный градиентный спуск

Алгоритм градиентного спуска, описанный в *главе 2*, вычисляет дисперсии весов и смещения для каждого наблюдения, но выполняет заучивание параметров (обновление весов и смещений) только после того, как были вычислены все наблюдения, или, другими словами, после так называемой эпохи. (Напомним, что цикл по всему набору данных называется эпохой.)

Преимущество: меньшее число обновлений весов и смещения означает более стабильный градиент, что обычно приводит к более стабильному схождению.

Недостатки:

- обычно этот алгоритм реализован таким образом, что все наборы данных должны находиться в памяти, что требует довольно интенсивных вычислений;
- этот алгоритм, как правило, является очень медленным для очень больших наборов данных.

Его возможная реализация может выглядеть следующим образом:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

training_epochs = 100
cost_history = []

for epoch in range(training_epochs+1):
    sess.run(optimizer, feed_dict={X:train, Y:labels_, learning_rate:0.01})
    cost_ = sess.run(cost, feed_dict={X:train, Y:labels_,
                                      learning_rate:0.01})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 50 == 0):
        print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Выполнение этого фрагмента кода для 100 эпох даст результат, подобный следующему:

```
Достигнута эпоха 0 стоимость J = 0.331401
Достигнута эпоха 50 стоимость J = 0.329093
Достигнута эпоха 100 стоимость J = 0.327383
```

Этот фрагмент кода работал примерно 2,5 минуты, но стоимостная функция почти не изменилась. Для того чтобы увидеть убывание стоимостной функции, вы должны выполнять тренировку в течение нескольких тысяч эпох, а это потребует довольно много времени. С помощью следующих ниже строк кода мы можем рассчитать точность:

```
correct_predictions = tf.equal(tf.argmax(y_,0), tf.argmax(Y,0))
accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"))
print("Точность:", accuracy.eval({X:train, Y:labels_, learning_rate:0.001},
                                                        session = sess))
```

После 100 итераций мы достигли лишь 16%-й точности на тренировочном наборе!

Стохастический градиентный спуск

Стохастический[8] градиентный спуск (stochastic gradient descent, SGD) вычисляет градиент стоимостной функции, а затем обновляет веса и смещения для каждого наблюдения в наборе данных.

Преимущества:

- частые обновления позволяют легко проверять ход самообучения модели (вам не нужно ждать до тех пор, пока все наборы данных будут рассмотрены);
- в некоторых задачах этот алгоритм работает быстрее, чем пакетный градиентный спуск;
- модель имеет собственный шум, что позволяет избежать локальных минимумов при попытке найти абсолютный минимум стоимостной функции.

Недостатки:

- в больших наборах данных этот метод является довольно медленным, поскольку из-за постоянных обновлений он требует интенсивных вычислений;
- тот факт, что алгоритм является шумным, может затруднить ему установиться на минимуме стоимостной функции, и схождение может оказаться не таким стабильным, как ожидалось.

Его возможная реализация может выглядеть следующим образом:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

cost_history = []

for epoch in range(100+1):
    for i in range(0, features.shape[1], 1):
        X_train_mini = features[:,i:i + 1]
        y_train_mini = classes[:,i:i + 1]
```

[8] Термин "стохастический" означает, что обновления имеют случайное распределение вероятностей и не могут быть предсказаны точно.

```
sess.run(optimizer, feed_dict = {X:X_train_mini, Y: y_train_mini,
                                 learning_rate: 0.0001})
cost_ = sess.run(cost, feed_dict={X:features, Y:classes,
                                  learning_rate:0.0001})
cost_history = np.append(cost_history, cost_)

if (epoch % 50 == 0):
    print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Если вы выполните этот исходный код, то получите результат, который должен выглядеть следующим образом (точные числа всякий раз будут отличаться, потому что мы инициализируем веса и смещения случайно, но динамика уменьшения должна быть одинаковой):

```
Достигнута эпоха 0 стоимость J = 0.31713
Достигнута эпоха 50 стоимость J = 0.108148
Достигнута эпоха 100 стоимость J = 0.0945182
```

Как уже отмечалось, этот метод может быть весьма нестабильным. Например, использование темпа заучивания `1e-3` (0,001) побудит `nan` появиться до достижения 100-й эпохи. Попробуйте поиграть с темпом заучивания и посмотреть, что произойдет. Вам потребуется довольно малое значение для того, чтобы этот метод сходился прилично. По сравнению с более высокими темпами заучивания (например, 0,05) такой метод, как пакетный градиентный спуск, сходится без проблем. Как уже отмечалось ранее, данный метод является в достаточной мере вычислительно интенсивным, и на моем ноутбуке на 100 эпох требуется примерно 35 минут. С таким вариантом градиентного спуска мы бы уже достигли точности 80% всего после 100 эпох. С этим вариантом заучивание, с точки зрения эпох, является очень эффективным, но и очень медленным.

Мини-пакетный градиентный спуск

При таком варианте градиентного спуска наборы данных разбиваются на определенное число малых (отсюда и термин "мини") групп наблюдений (именуемых пакетами), а веса и смещения обновляются только после того, как каждый пакет был подан в модель. Этот метод используется в сфере глубокого обучения наиболее часто.

Преимущества:

- частота обновления модели выше, чем при пакетном градиентном спуске, но ниже чем в стохастическом градиентном спуске. Следовательно, этот вариант допускает более надежное схождение;
- этот метод является вычислительно намного эффективнее, чем пакетный градиентный спуск или стохастический градиентный спуск, поскольку требуется меньше вычислений и ресурсов;
- этот вариант, безусловно, является самым быстрым (как мы увидим позже) из трех.

Среди его *недостатков*: использование этого варианта градиентного спуска вводит новый гиперпараметр, который необходимо отрегулировать — размер пакета (число наблюдений в мини-пакете).

Для пакета размером 50 наблюдений возможная реализация данного варианта градиентного спуска может выглядеть следующим образом:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

cost_history = []

for epoch in range(100+1):
    for i in range(0, features.shape[1], 50):
        X_train_mini = features[:,i:i + 50]
        y_train_mini = classes[:,i:i + 50]

        sess.run(optimizer, feed_dict = {X:X_train_mini, Y:y_train_mini,
                                         learning_rate:0.001})
        cost_ = sess.run(cost, feed_dict={X:features, Y:classes,
                                          learning_rate:0.001})
        cost_history = np.append(cost_history, cost_)

        if (epoch % 50 == 0):
            print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Обратите внимание, что исходный код является тем же, что и для стохастического градиентного спуска. Единственное различие — это размер пакета. В этом примере каждый раз перед обновлением весов и смещений мы используем 50 наблюдений. Выполнение данного примера даст вам результат, который должен выглядеть следующим образом (ваши числа будут отличаться из-за случайной инициализации весов и смещений):

```
Достигнута эпоха 0 стоимость J = 0.322747
Достигнута эпоха 50 стоимость J = 0.193713
Достигнута эпоха 100 стоимость J = 0.141135
```

В этом случае мы использовали темп заучивания `1e-3` — намного больше, чем в стохастическом градиентном спуске — и достигли значения стоимостной функции 0,14 — большего значения, чем 0,094, достигнутое с помощью стохастического градиентного спуска, но гораздо меньше, чем значение 0,32, достигнутое с помощью пакетного градиентного спуска — и на это уходит всего 2,5 минуты. Таким образом, он быстрее, чем стохастический градиентный спуск в 14 раз. После 100 эпох мы достигли точности 66%.

Сравнение вариантов градиентного спуска

В табл. 3.3 приведена сводка результатов по трем вариантам градиентного спуска для 100 эпох.

Таблица 3.3. *Сводка результатов по трем вариантам градиентного спуска для 100 эпох*

Вариант градиентного спуска	Время работы, мин	Окончательное значение стоимостной функции	Точность, %
Пакетный градиентный спуск	2,5	0,323	16
Мини-пакетный градиентный спуск	2,5	0,14	66
Стохастический градиентный спуск (SGd)	35	0,094	80

Теперь вы видите, что алгоритм стохастического градиентного спуска достигает самого низкого значения стоимостной функции с тем же числом эпох, хотя он и является самым медленным. Для того чтобы стоимостная функция в мини-пакетном градиентном спуске достигла значения 0,094, требуется 450 эпох и примерно 11 минут. Тем не менее это огромное улучшение по сравнению со стохастическим градиентным спуском — 31% времени для тех же результатов.

На рис. 3.16 показана разница в том, как уменьшается стоимостная функция при разных размерах мини-пакета. Понятно, что по отношению к числу эпох чем мень-

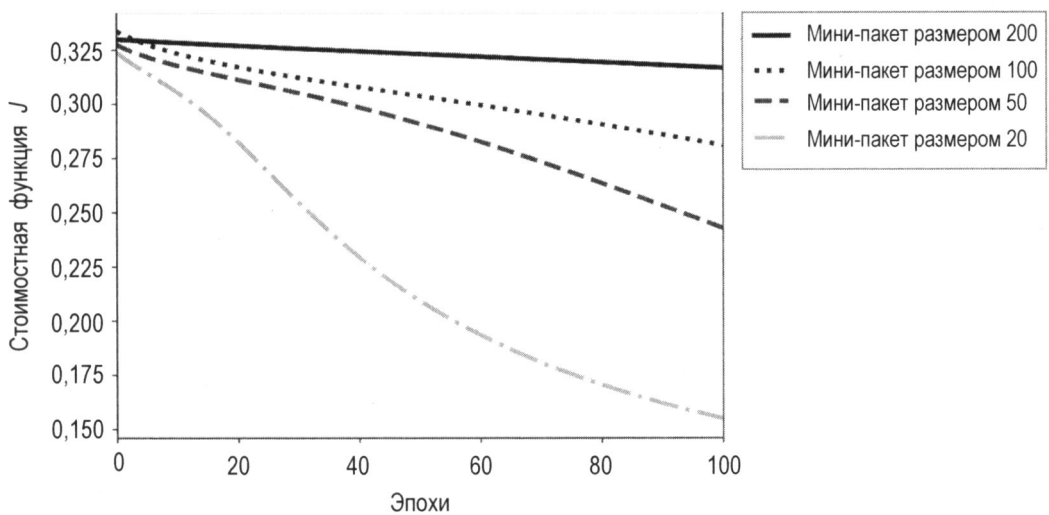


РИС. 3.16. Сравнение скорости схождения алгоритма мини-пакетного градиентного спуска с разными размерами мини-пакета

ше размер мини-пакета, тем быстрее уменьшение (хотя и не по времени). Для этого рисунка использовался темп заучивания $\gamma = 0,001$. Обратите внимание, что время, необходимое в каждом случае, не одинаково, и чем меньше размер мини-пакета, тем больше алгоритму требуется времени.

ПРИМЕЧАНИЕ. Наилучший компромисс между временем работы и скоростью схождения (по числу эпох) достигается мини-пакетным градиентным спуском. Оптимальный размер мини-пакета зависит от вашей задачи, но, как правило, малые числа, такие как 30 или 50, являются хорошим вариантом. Вы найдете компромиссное соотношение между временем работы и скоростью схождения.

Рисунок 3.17 дает представление о том, как время работы зависит от значения, которое стоимостная функция достигает после 100 эпох. Каждая точка помечена размером мини-пакета, используемого во время прогона. Обратите внимание, что точки обозначают одиночные прогоны, и график указывает только на зависимость. Время работы и значение стоимостной функции имеет малую дисперсию по нескольким прогонам. Эта дисперсия на графике не показана. Вы видите, что уменьшение размера мини-пакета начиная с 300 наблюдений быстро уменьшает значение J после 100 эпох без значительного увеличения времени работы, и это происходит до тех пор, пока вы не получите размер мини-пакета, который составит около

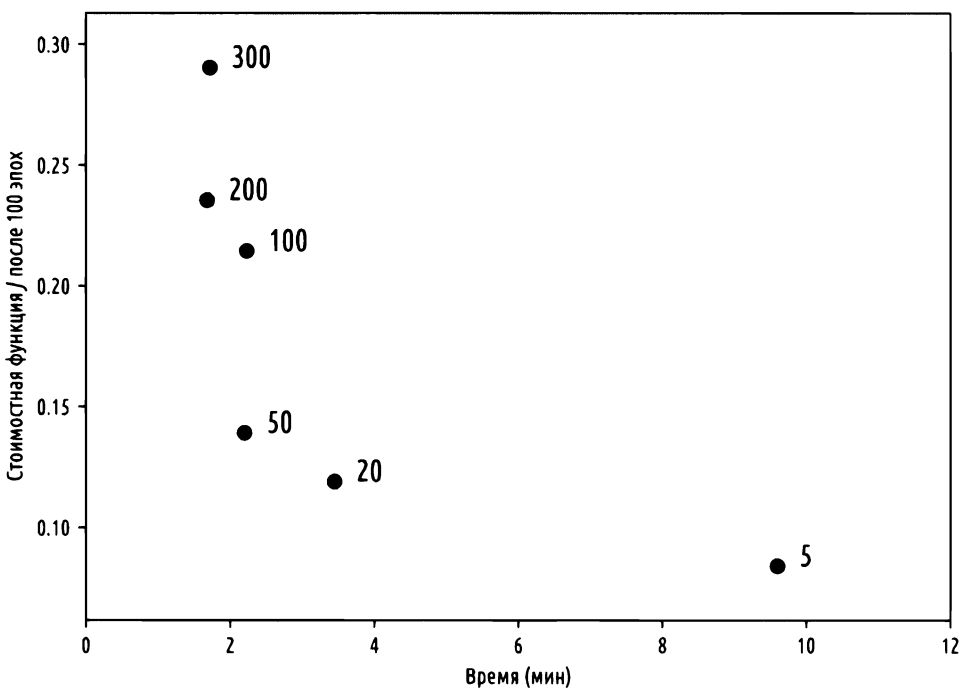


РИС. 3.17. График для набора данных Zalando, показывающий значение стоимостной функции после 100 эпох в сопоставлении со временем работы, необходимым для выполнения 100 эпох

50 наблюдений. В этой точке время начинает быстро увеличиваться, а значение J после 100 эпох больше не уменьшается одинаково быстро и выравнивается. В интуитивном плане, наилучший компромисс состоит в том, чтобы выбрать размер мини-пакета, когда кривая находится ближе к нулю (малое время работы и малое значение стоимостной функции), и это будет при размере мини-пакета, находящемся в интервале между 50 и 30 наблюдениями. Вот почему эти значения выбираются чаще всего. После этой точки время работы начинает быстро расти, и уменьшать размер мини-пакета больше не стоит. Обратите внимание, что для других наборов данных оптимальное значение может сильно отличаться. Поэтому для того чтобы увидеть, какое из них работает лучше всего, стоит попробовать разные их значения. В очень больших наборах данных вы вполне можете попробовать более крупные значения, например 200, 300 или 500 наблюдений. В нашем случае у нас 60 000 наблюдений и мини-пакет размером 50 наблюдений, что дает в итоге 1200 пакетов. Если у вас гораздо больше данных, например `1e6` наблюдений, то размер мини-пакета 50 даст 20 000 пакетов. Имейте это в виду и попробуйте разные значения для выбора наилучшего.

Хорошей практикой программирования является написание функции, которая выполняет ваши вычисления. Благодаря этому можно отрегулировать гиперпараметры (например, размер мини-пакета) без копирования и вставки одного и того же фрагмента кода снова и снова. Следующая функция может использоваться для тренировки модели:

```
def model(minibatch_size, training_epochs, features, classes, logging_step=100,
learning_r = 0.001):
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    cost_history = []

    for epoch in range(training_epochs+1):
        for i in range(0, features.shape[1], minibatch_size):
            X_train_mini = features[:,i:i + minibatch_size]
            y_train_mini = classes[:,i:i + minibatch_size]

            sess.run(optimizer, feed_dict = {X:X_train_mini,
                                             Y:y_train_mini,
                                             learning_rate:learning_r})
        cost_ = sess.run(cost, feed_dict={X:features, Y:classes,
                                          learning_rate:learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % logging_step == 0):
            print("Достугнута эпоха",epoch,"стоимость J =", cost_)

    return sess, cost_history
```

Функция `model()` принимает следующие параметры.

- `minibatch_size` — число наблюдений, которые мы хотим иметь в каждом пакете. Обратите внимание, что, если для этого гиперпараметра мы выберем число q, которое не является делителем числа m (числа наблюдений), или, другими словами, m/q не является целым числом, то у нас последний мини-пакет будет иметь другое число наблюдений, чем все остальные. Но это не станет проблемой для тренировки. Например, предположим, что у нас имеется гипотетический набор данных с $m = 100$, и вы решили использовать мини-пакет размером 32 наблюдения. Тогда при $m = 100$ у вас будет 3 полных мини-пакета с 32 наблюдениями и 1 с 4, т. к. $100 = 3 \cdot 32 + 4$. Теперь вы можете задаться вопросом: что произойдет с такой строкой кода, как

```
X_train_mini = features[:,i:i + 32]
```

  когда $i = 96$, и `features` имеет только 100 наблюдений? Разве мы не выйдем за пределы массива? К счастью, Python благосклонен к программистам и об этом позаботится. Рассмотрим следующий фрагмент кода:

```
l = np.arange(0,100)
for i in range (0, 100, 32):
    print (l[i:i+32])
```

  Результат будет таким:

```
[ 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31]

[32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59 60 61 62 63]

[64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81
82 83 84 85 86 87 88 89 90 91 92 93 94 95]

[96 97 98 99]
```

  Как видите, в последнем пакете имеется всего четыре наблюдения, и мы не получаем ошибок. Так что, вы не должны об этом беспокоиться и можете выбрать любой размер мини-пакета, который лучше всего работает для вашей задачи.

- `training_epochs` — число необходимых нам эпох.
- `features` — тензор с признаками.
- `classes` — тензор с метками.
- `logging_step` — сообщает функции о том, что необходимо печатать значения стоимостной функции каждую `logging_step` эпоху.
- `learning_r` — темп заучивания, который мы хотим использовать.

ПРИМЕЧАНИЕ. Написание функции с гиперпараметрами на входе является обычной практикой. Это позволяет тестировать разные модели с разными значениями гиперпараметров и проверять, какое из них является лучшим.

Примеры неправильных предсказаний

Выполнение модели с пакетным градиентным спуском, одним скрытым слоем с 5 нейронами в течение 1000 эпох и темпом заучивания 0,001 даст нам точность на тренировочном наборе 82,3%. Точность можно увеличить, используя больше нейронов в скрытом слое. Например, использование 50 нейронов, 1000 эпох и темпа заучивания 0,001 позволит достичь 86,4% на тренировочном наборе и 86,1% на тестовом наборе. Интересно проверить несколько примеров неправильно классифицированных изображений для того, чтобы увидеть, можно или нет что-то понять из ошибок. На рис. 3.18 показан пример неправильно классифицированных изображений для каждого класса. Рядом с каждым изображением сообщается истинный класс (помеченный как "Ист.:") и предсказанный класс (помеченный как "Пред.:"). Используемая здесь модель имеет один скрытый слой с пятью нейронами и выполнялась в течение 1000 эпох с темпом заучивания 0,001.

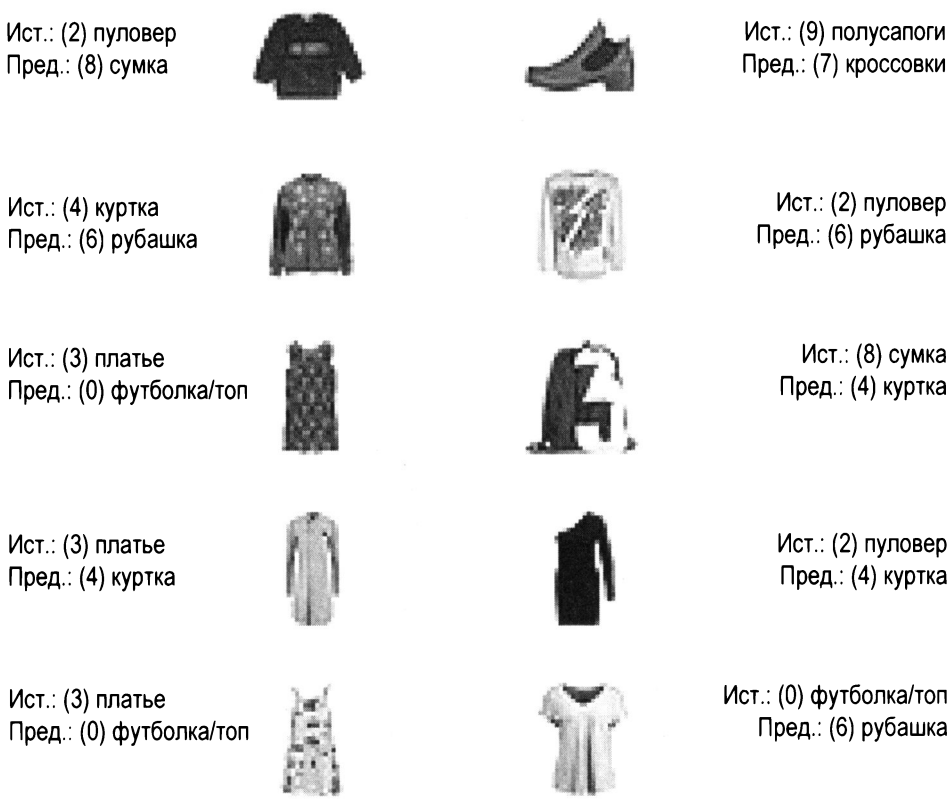


РИС. 3.18. Пример неправильно классифицированных изображений по каждому классу

Некоторые ошибки понятны, например, как в левой верхней части рисунка. Рубашка ошибочно классифицируется как куртка. На этом изображении действительно трудно определить, какая вещь чем является, и вы могли бы точно так же легко сделать ту же самую ошибку. С другой стороны, неправильно классифицированная сумка легко распознается человеком.

Инициализация весов

Если вы попробовали выполнить этот исходный код, то вы уже поняли, что схождение алгоритма сильно зависит от способа инициализации весов. Вы помните, что для инициализации весов мы используем следующую строку:

```
W1 = tf.Variable(tf.truncated_normal([n1, n_dim], stddev=.1))
```

Но почему стандартное отклонение было выбрано равным 0,1?

Вы наверняка задавались таким вопросом. В предыдущих разделах все было направлено на то, чтобы сосредоточить все усилия на понимании того, как такая сеть работает, не отвлекаясь на дополнительную информацию, но сейчас самое время взглянуть на эту проблему повнимательнее, потому что она играет фундаментальную роль для многих слоев. В сущности, мы инициализируем веса с небольшим стандартным отклонением для того, чтобы алгоритм градиентного спуска не взорвался и не начал возвращать `nan`. Например, в нашем первом слое для i-го нейрона мы должны будем вычислить активационную функцию ReLU передаваемой величины (обратитесь к началу данной главы, если вы забыли, почему) следующим образом:

$$z_i = \sum_{j=1}^{n_x} \left(w_{ij}^{[1]} x_j + b_i^{[1]} \right).$$

Обычно в глубокой сети число весов является довольно большим, поэтому вы легко можете себе представить, что если веса $w_{ij}^{[1]}$ являются большими, то величина z_i тоже может быть довольно большой, и активационная функция ReLU может вернуть значение `nan`, потому что аргумент является слишком большим для того, чтобы Python смог его вычислить надлежаще. Поэтому существует потребность в том, чтобы величина z_i была достаточно малой с целью избежать взрывного роста выходов нейронов и была достаточно большой с целью предотвратить затухание выходов, и следовательно, делая процесс схождения очень медленным.

Данная проблема была тщательно исследована[9], и в зависимости от используемой активационной функции существуют разные стратегии инициализации. Некоторые из них приведены в табл. 3.4, в которой принято допущение, что вес будет инициа-

[9] См., например, Xavier Glorot, Yoshua Bengio. Understanding the Difficulty of Training Deep Feedforward Neural Networks (Ксавье Глоро, Джошуа Беджо. Понимание трудности тренировки глубоких нейронных сетей прямого распространения), доступная по адресу **https://goo.gl/bHB5BM**.

лизирован нормальным распределением со средним значением, равным 1, и стандартным отклонением (при этом стандартное отклонение будет зависеть от активационной функции, которую вы хотите использовать).

Таблица 3.4. *Разные стратегии инициализации в зависимости от активационной функции*

Активационная функция	**Стандартное отклонение σ для данного слоя**	
Сигмоида	$\sigma = \sqrt{\frac{2}{n_{вх} + n_{вых}}}$	Обычно называется инициализацией Ксавье
ReLU	$\sigma = \sqrt{\frac{4}{n_{вх} + n_{вых}}}$	Обычно называется инициализацией Хе

В слое l число входов (вх) будет равняться числу нейронов предыдущего слоя $l-1$, а число выходов (вых) — числу нейронов в следующем слое: $l+1$. Таким образом, мы будем иметь

$$n_{вх} = n_{l-1}$$

и

$$n_{вых} = n_{l+1}.$$

Очень часто глубокие сети, подобные той, которая обсуждалась ранее, будут иметь несколько слоев, все с одинаковым числом нейронов. Следовательно, для большинства слоев у вас будет $n_{l-1} = n_{l+1}$, и значит, при инициализации Ксавье у вас будет следующее:

$$\sigma_{Ксавье} = \sqrt{1/n_{l+1}} \text{ или } \sigma_{Ксавье} = \sqrt{1/n_{l-1}}$$

Для активационных функций ReLU инициализация Хе будет:

$$\sigma_{Хе} = \sqrt{2/n_{l+1}} \text{ или } \sigma_{Хе} = \sqrt{2/n_{l-1}}$$

Рассмотрим активационную функцию ReLU (ту, которую мы использовали в этой главе). Каждый слой, как уже говорилось, будет иметь n нейронов. Способ инициализации весов для слоя 3, например, был бы

```
stddev = 2 / np.sqrt(n4+n2)
W3=tf.Variable(tf.truncated_normal([n3,n2], stddev = stddev)
```

Либо, если все слои имеют одинаковое число нейронов и, следовательно, `n2=n3=n4`, то можно было бы использовать следующее:

```
stddev = 2 / np.sqrt(2.0*n2)
W3=tf.Variable(tf.truncated_normal([n3,n2], stddev = stddev)
```

Как правило, для упрощения вычисления и построения сетей наиболее типичной используемой формой инициализации является та, которая применяется для активационной функции ReLU:

$$\sigma_{Xe} = \sqrt{2/n_{l-1}}$$

и

$$\sigma_{Ксавье} = \sqrt{1/n_{l-1}}\ .$$

Для сигмоидальной активационной функции, например, исходный код инициализации весов для сети, которую мы использовали ранее с одним слоем, будет выглядеть следующим образом:

```
W1 = tf.Variable(tf.random_normal([n1, n_dim], stddev= 2.0 /
np.sqrt(2.0*n_dim)))
b1 = tf.Variable(tf.ones([n1,1]))
W2 = tf.Variable(tf.random_normal([n2, n1], stddev= 2.0 / np.sqrt(2.0*n1)))
b2 = tf.Variable(tf.ones([n2,1]))
```

Применение этой инициализации может значительно ускорить тренировку и является стандартным способом инициализации весов, используемым во многих библиотеках (например, в библиотеке Caffe).

Эффективное добавление многочисленных слоев

Необходимость многократно набирать весь этот код немного утомляет и приводит к ошибкам. Обычно мы определяем функцию, которая создает слой. Это можно легко сделать с помощью вот этого фрагмента кода:

```
def create_layer (X, n, activation):
    ndim = int(X.shape[0])
    stddev = 2 / np.sqrt(ndim)
    initialization = tf.truncated_normal((n, ndim), stddev = stddev)
    W = tf.Variable(init)
    b = tf.Variable(tf.zeros([n,1]))
    Z = tf.matmul(W,X)+b
    return activation(Z)
```

Давайте пройдемся по этому фрагменту кода.

1. Сначала мы получаем размерность входов, которая необходима для того, чтобы определить правильную весовую матрицу.
2. Потом мы инициализируем веса с помощью инициализации Xe, описанной в предыдущем разделе.
3. Далее мы создаем веса `W` и смещение `b`.
4. Затем мы вычисляем величину `Z` и возвращаем активационную функцию, вычисленную на `Z`. (Обратите внимание, что в Python можно передавать функции

в качестве аргументов другим функциям. В этом случае `activation` может равняться `tf.nn.relu`.)

Таким образом, для создания сетей можно просто написать конструкционный код (в данном примере, с двумя слоями) следующим образом:

```
n_dim = 784
n1 = 300
n2 = 300
n_outputs = 10

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [10, None])

learning_rate = tf.placeholder(tf.float32, shape=())

hidden1 = create_layer (X, n1, activation = tf.nn.relu)
hidden2 = create_layer (hidden1, n2, activation = tf.nn.relu)
outputs = create_layer (hidden2, n3, activation = tf.identity)
y_ = tf.nn.softmax(outputs)

cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Для выполнения модели мы снова определяем функцию `model()`, как уже обсуждалось ранее.

```
def model(minibatch_size, training_epochs, features, classes,
          logging_step=100, learning_r = 0.001):
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    cost_history = []

    for epoch in range(training_epochs+1):
        for i in range(0, features.shape[1], minibatch_size):
            X_train_mini = features[:,i:i + minibatch_size]
            y_train_mini = classes[:,i:i + minibatch_size]

            sess.run(optimizer, feed_dict = {X:X_train_mini,
                                             Y:y_train_mini,
                                             learning_rate:learning_r})
        cost_ = sess.run(cost, feed_dict={X:features, Y:classes,
                                          learning_rate:learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % logging_step == 0):
            print("Достигнута эпоха",epoch,"стоимость J =", cost_)

    return sess, cost_history
```

Теперь исходный код намного легче понять, и вы можете использовать его для создания сетей такими большими, какими захотите.

С приведенными выше функциями очень легко выполнять несколько моделей и сравнивать их, как было сделано на рис. 3.19, который иллюстрирует пять разных протестированных моделей.

- Один слой и десять нейронов в каждом слое.
- Два слоя и десять нейронов в каждом слое.
- Три слоя и десять нейронов в каждом слое.
- Четыре слоя и десять нейронов в каждом слое.
- Четыре слоя и 100 нейронов в каждом слое.

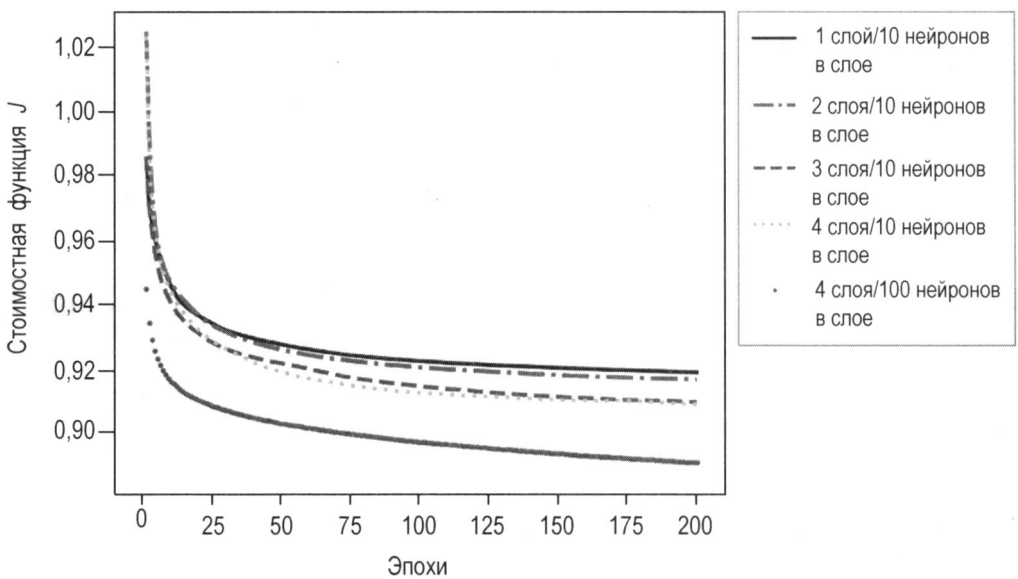


РИС. 3.19. Стоимостная функция в сопоставлении с эпохами для пяти моделей, как описано в легенде

В случае если вам интересно, модель с четырьмя слоями со 100 нейронами каждый, которая выглядит намного лучше, чем другие, начинает двигаться в режиме переподгонки, с точностью 94% на тренировочном наборе и 88% на рабочем наборе (всего после 200 эпох).

Преимущества дополнительных скрытых слоев

Рекомендуется поиграть с моделями. Попробуйте поварьировать число слоев, число нейронов, способы инициализации весов и т. д. Если вы потратите на это некоторое время, то за несколько минут работы сможете достичь точности более 90%,

но это требует некоторого навыка. Если вы попробуете несколько моделей, то сможете понять, что в этом случае использование нескольких слоев, по всей видимости, не накапливает преимуществ по сравнению с сетью только с одним слоем. Зачастую так и бывает.

Теоретически, однослойная сеть может аппроксимировать любую функцию, которую вы можете себе представить, но число необходимых нейронов может быть очень большим, и, следовательно, модель становится гораздо менее полезной. Загвоздка заключается в том, что способность аппроксимировать функцию не означает, что сеть способна научиться делать это, например, из-за огромного числа задействованных нейронов или требуемого времени.

Эмпирически было показано, что сети с большим числом слоев требуют гораздо меньшего числа нейронов для достижения тех же результатов и обычно лучше обобщают на неизвестных данных.

ПРИМЕЧАНИЕ. В сетях совсем не обязательно иметь многочисленные слои, но часто, на практике, вам все же следует их иметь. Почти всегда хорошо попробовать сеть из нескольких слоев с несколькими нейронами в каждом, вместо сети с одним слоем, заполненным огромным числом нейронов. Какого-то фиксированного правила, декларирующего, сколько нейронов или слоев лучше, нет. Начните с малого числа слоев и нейронов, а затем увеличивайте их до тех пор, пока ваши результаты не перестанут улучшаться.

Вдобавок наличие большего числа слоев позволит вашей сети заучивать разные аспекты входных данных. Например, один слой может научиться распознавать вертикальные края изображения, а другой — горизонтальные. Напомним, что в этой главе рассматривались сети, в которых каждый слой идентичен (вплоть до числа нейронов) всем остальным. Позже, в *главе 4*, вы увидите, как создавать сети, в которых каждый слой выполняет очень разные задачи, а также структурирован посвоему, что делает этот вид сети гораздо более мощным при решении определенных задач, которые обсуждались ранее в этой главе.

Возможно, вы помните, что в *главе 2* мы пытались предсказать продажные цены на дома в окрестностях Бостона. В этом случае сеть с несколькими слоями может показывать больше информации о том, как признаки связаны с ценой. Например, первый слой может выявлять базовые связи, в частности, о том, что большие дома имеют более высокие цены. Но второй слой может выявлять более сложные связи, например, о том, что большие дома с меньшим числом ванных комнат имеют более низкие продажные цены.

Сравнение разных сетей

Теперь вы должны знать, как строить нейронные сети с огромным числом слоев и нейронов. Однако относительно легко затеряться в лесу возможных моделей, не зная, какие из них стоит попробовать. Предположим, вы начинаете с сети (как

делалось в предыдущих разделах) с одним скрытым слоем из пяти нейронов, одним слоем из десяти нейронов (для функции softmax) и нейроном "softmax". Предположим, вы достигли некоторой точности и хотели бы попробовать разные модели. Сначала вы должны попытаться увеличить число нейронов в скрытых слоях для того, чтобы увидеть, чего вы можете достичь. На рис. 3.20 построен график стоимостной функции по мере того, как она уменьшается для разных чисел нейронов. Расчеты проводились с помощью мини-пакетного градиентного спуска с пакетом размером 50 наблюдений, одного скрытого слоя соответственно с 1, 5, 15 и 30 нейронами и темпом заучивания 0,05. Вы видите, как переход от одного нейрона к пяти сразу ускоряет схождение. Но дальнейшее увеличение числа нейронов к значительному улучшению не приводит. Например, увеличение нейронов с 15 до 30 практически улучшению не способствовало.

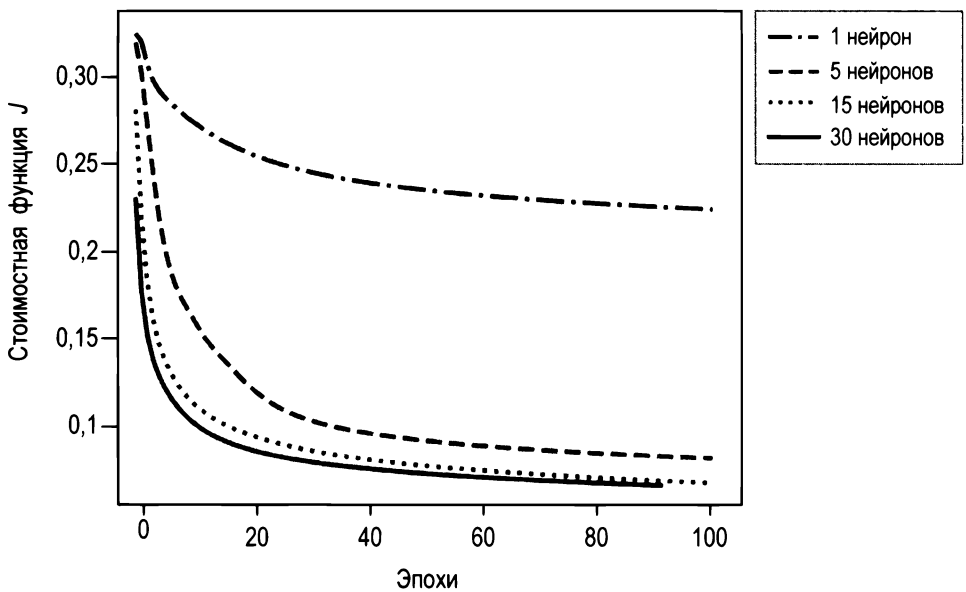


РИС. 3.20. Снижение стоимостной функции в сопоставлении с эпохами для нейронной сети с одним скрытым слоем соответственно с 1, 5, 15 и 30 нейронами, как указано в легенде. Расчеты были проведены с помощью мини-пакетного градиентного спуска, с пакетом размером 50 наблюдений и темпом заучивания 0,05

Давайте сначала попробуем найти способ сравнения этих сетей. Сравнение только числа нейронов может ввести в заблуждение, как будет показано в ближайшее время. Напомним, что алгоритм пытается найти наилучшие сочетания весов и смещений с целью минимизации стоимостной функции. Но сколько в модели имеется заучиваемых параметров? У нас есть веса и смещения. Из теоретических выкладок вы помните, что с каждым слоем мы можем ассоциировать определенное число весов, и число заучиваемых параметров в слое l, которое мы обозначим через $Q^{[l]}$, задается суммарным числом элементов матрицы $\mathbf{W}^{[l]}$, т. е. $n_l n_{l-1}$ (где по определе-

нию мы получаем $n_0 = n_x$), плюс число имеющихся смещений (в каждом слое у нас будет n_l смещений). Число $Q^{[l]}$ может быть записано как

$$Q^{[l]} = n_l n_{l-1} + n_l = n_l \left(n_{l-1} + 1\right).$$

Таким образом, суммарное число заучиваемых параметров в сети (обозначенное здесь через Q) может быть записано как:

$$Q = \sum_{j=1}^{L} n_l \left(n_{l-1} + 1\right),$$

где по определению $n_0 = n_x$. Обратите внимание, что параметр Q сети сильно зависит от архитектуры. Давайте посчитаем его на нескольких примерах для того, чтобы вы поняли, что имеется в виду (табл. 3.5).

Таблица 3.5. *Сравнение значений Q для разных сетевых архитектур*

Сетевая архитектура	**Параметр Q (число заучиваемых параметров)**	**Число нейронов**
Сеть A: 784 признаков, 2 слоя: $n_1 = 15$, $n_2 = 10$	$Q_A = 15 \cdot (784 + 1) + 10 \cdot (15 + 1) = 11\,935$	25
Сеть B: 784 признаков, 16 слоев: $n_1 = n_2 = \ldots = n_{15} = 1$, $n_{16} = 10$	$Q_B = 1 \cdot (784 + 1) + 1 \cdot (1 + 1) + \ldots + 10 \cdot (1 + 1) = 923$	25
Сеть C: 784 признаков, 3 слоя: $n_1 = 10$, $n_2 = 10$, $n_3 = 10$	$Q_C = 10 \cdot (784 + 1) + 10 \cdot (10 + 1) + 10 \cdot (10 + 1) = 8070$	30

Хотелось бы обратить ваше внимание на сети *A* и *B*. У обеих по 25 нейронов, но параметр Q_A гораздо больше (более чем в 10 раз), чем Q_B. Вы можете легко представить, что сеть *A* будет гораздо гибче в заучивании, чем сеть *B*, даже если число нейронов будет одинаковым.

ПРИМЕЧАНИЕ. Утверждение, что число Q является мерой того, насколько сеть является сложной или насколько она является хорошей, будет вводить в заблуждение. Это не так, и вполне может случиться, что из всех нейронов лишь немногие будут играть роль. Поэтому показанные тут расчеты не рассказывают всю историю. Существует огромное число исследований по так называемым эффективным степеням свободы глубоких нейронных сетей, но эта тема выходит за рамки данной книги. Тем не менее этот параметр обеспечит хорошее эмпирическое правило при принятии решения о том, находится ли набор моделей, которые вы хотите испытать, в разумной прогрессии сложности.

Проверка Q для модели, которую вы хотите протестировать, может дать вам несколько подсказок, которыми следует пренебречь и которые следует попробовать. Например, рассмотрим тестируемые нами случаи на рис. 3.20 и рассчитаем параметр Q для каждой сети (табл. 3.6).

Таблица 3.6. *Сравнение значений Q для разных сетевых архитектур*

Сетевая архитектура	**Параметр Q**	**Число нейронов**
784 признаков, 1 слой с 1 нейроном, 1 слой с 10 нейронами	$Q = 1 \cdot (784 + 1) + 10 \cdot (1 + 1) = 895$	11
784 признаков, 1 слой с 5 нейронами, 1 слой с 10 нейронами	$Q = 5 \cdot (784 + 1) + 10 \cdot (5 + 1) = 3985$	15
784 признаков, 1 слой с 15 нейронами, 1 слой с 10 нейронами	$Q = 15 \cdot (784 + 1) + 10 \cdot (15 + 1) = 11\,935$	25
784 признаков, 1 слой с 30 нейронами, 1 слой с 10 нейронами	$Q = 30 \cdot (784 + 1) + 10 \cdot (30 + 1) = 23\,860$	40

Из рис. 3.20 давайте предположим, что в качестве кандидата на лучшую модель мы выбираем модель с 15 нейронами. Теперь давайте предположим, что мы хотим попробовать модель с 3 слоями, все с тем же числом нейронов, которые должны конкурировать (и, возможно, будут лучше), чем наша (на данный момент) кандидатная модель с 1 слоем и 15 нейронами. Что мы должны выбрать в качестве отправной точки для числа нейронов в трех слоях? Давайте обозначим моделью A ту, которая имеет 1 слой с 15 нейронами, и моделью B ту, которая имеет 3 слоя с (пока) неизвестным числом нейронов в каждом слое, обозначенным как n_B. Для обеих сетей можно легко рассчитать параметр Q:

$$Q_A = 15 \cdot (784 + 1) + 10 \cdot (15 + 1) = 11\,935$$

и

$$Q_B = n_B \cdot (784 + 1) + n_B \cdot (n_B + 1) + n_B \cdot (n_B + 1) + 10 \cdot (n_B + 1) = 2n_B^2 + 797n_B + 10.$$

Какое значение для n_B даст $Q_B \approx Q_A$? Это уравнение можно легко решить.

$$2n_B^2 + 797n_B + 10 = 11\,935.$$

Вы должны уметь решать квадратные уравнения, поэтому здесь будет приведено только решение (совет: попробуйте его решить). Это уравнение решается для значения $n_B = 14{,}4$, но поскольку мы не можем иметь 14,4 нейронов, нам придется использовать ближайшее целое число, которым будет $n_B = 14$. Для $n_B = 14$ мы будем иметь $Q_B = 11\,560$, значение очень близкое к 11 935.

ПРИМЕЧАНИЕ. Стоит повторить еще раз. Тот факт, что две сети имеют одинаковое число заучиваемых параметров, совсем не означает, что они могут достичь одинаковой точности. Это даже не значит, что если одна учится очень быстро, то и вторая будет учиться!

Однако наша модель с 3 слоями по 14 нейронов может стать хорошей отправной точкой для дальнейшего тестирования.

Также стоит обсудить еще один момент, который очень важен во время работы со сложным набором данных. Рассмотрим первый слой. Предположим, мы рассматриваем набор данных Zalando и создаем сеть с двумя слоями: первый с одним нейроном и второй с многочисленными. Все сложные признаки, которые есть в вашем наборе данных, могут быть потеряны в первом нейроне, потому что он объединит все признаки в одном значении и передаст это значение всем другим нейронам второго слоя.

Советы по выбору правильной сети

Возможно, вы задаетесь вопросом: рассмотрено много случаев, предоставлено много формул, но какие решения необходимо принять при проектировании сети?

К сожалению, фиксированного набора правил нет. Но вы можете учесть следующие советы.

- При рассмотрении набора моделей (или сетевых архитектур), которые вы хотите протестировать, хорошее эмпирическое правило состоит в том, чтобы начать с менее сложных и переходить к более сложным. Еще один способ состоит в оценивании относительной сложности (убедиться, что вы двигаетесь в правильном направлении) с использованием параметра Q.
- В случае если вы не можете достичь хорошей точности, проверьте, имеет ли какой-либо из ваших слоев особенно низкое число нейронов. Этот слой может занизить эффективную емкость заучивания на сложном наборе данных сети. Рассмотрим, например, случай с одним нейроном на рис. 3.20. Модель не может достичь низких значений стоимостной функции, т. к. сеть является слишком простой для того, чтобы обучиться на таком сложном наборе данных, как Zalando.
- Помните, что низкое или высокое число нейронов всегда зависит от числа имеющихся признаков. Если в вашем наборе данных только два признака, одного нейрона вполне может быть достаточно, но если у вас их несколько сотен (как в наборе данных Zalando, где $n_x = 784$), то вы не должны ожидать, что одного нейрона будет достаточно.
- Ответ на вопрос "Какая архитектура вам нужна?" также зависит от того, что вы хотите сделать. Всегда стоит обратиться к онлайновой литературе с целью посмотреть, что было обнаружено другими при решении конкретных задач. На-

пример, хорошо известно, что для распознавания изображений очень хороши сверточные сети, поэтому они будут отличным вариантом для выбора.

ПРИМЕЧАНИЕ. При переходе от модели с L слоями к модели с $L+1$ слоями всегда рекомендуется начинать с новой модели, используя немного меньшее число нейронов в каждом слое, а затем постепенно их увеличивать. Помните, что у большего числа слоев есть шанс эффективнее заучивать сложные признаки, поэтому если вам повезет, то меньшего числа нейронов может быть достаточно. И это стоит попробовать. Всегда следите за своей оптимизационной метрикой (упомянутой в *главе 2*) всех своих моделей. Когда вы больше не получаете каких-то особых улучшений, то вполне стоит попробовать совершенно другие архитектуры (возможно, сверточные нейронные сети и т. д.).

ГЛАВА 4

Тренировка нейронных сетей

Как вы, наверное, уже поняли, строить сложные сети с помощью TensorFlow довольно просто. Для того чтобы построить сеть с тысячами (и даже больше) параметров, достаточно всего нескольких строк кода. Уже сейчас должно быть понятно, что проблемы возникают во время тренировки таких сетей. Тестирование гиперпараметров сопряжено с трудностями, оно нестабильно и проходит медленно, потому что прогон через несколько сотен эпох может занимать несколько часов. Это не только проблема производительности; в противном случае было бы достаточно использовать более быстрое оборудование. Проблема в том, что очень часто процесс схождения (заучивания) не работает вообще. Он останавливается, расходится или никогда не приближается к минимуму стоимостной функции. Нам нужны способы сделать тренировочный процесс эффективным, быстрым и надежным. Вы познакомитесь с двумя главными стратегиями, которые помогут вам во время тренировки сложных сетей: динамическим ослаблением темпа заучивания и оптимизаторам (такими как RMSProp, Momentum и Adam), которые умнее, чем простой градиентный спуск.

Динамическое ослабление темпа заучивания

Уже несколько раз упоминалось, что параметр темпа заучивания γ является крайне важным, и что плохой его подбор приведет к тому, что модель работать не будет. Обратитесь еще раз к рис. 2.12, на котором показано, как при выборе слишком большого темпа заучивания алгоритм градиентного спуска будет отскакивать от минимума и не сходиться. Давайте, не вдаваясь в обсуждение, перепишем уравнения, приведенные в *главе 2* при рассмотрении алгоритма градиентного спуска, описывающие обновление весов и смещения. (Там описывался алгоритм для задачи с двумя весами: w_0 и w_1.)

$$w_{0,[n+1]} = w_{0,[n]} - \gamma \frac{\partial J\left(w_{0,[n]}, w_{1,[n]}\right)}{\partial w_0};$$

$$w_{1,[n+1]} = w_{1,[n]} - \gamma \frac{\partial J\left(w_{0,[n]}, w_{1,[n]}\right)}{\partial w_1}.$$

Напомним обозначения (в случае если вы подзабыли, как работает градиентный спуск, обратитесь к *главе 2* еще раз):

- $w_{0,[n]}$ — вес 0 на итерации n;
- $w_{1,[n]}$ — вес 1 на итерации n;
- $J\left(w_{0,[n]}, w_{1,[n]}\right)$ — стоимостная функция на итерации n;
- γ — темп заучивания.

Для того чтобы показать эффект от действия обсуждаемых далее методов, мы рассмотрим ту же самую задачу, которая была описана в *разд. "Темп заучивания на практическом примере" главы 2*. График весов $w_{0,[n]}$, $w_{1,[n]}$, построенный на контурных линиях стоимостной функции для $\gamma = 2$ (рис. 4.1), показывает (как вы помните из *главы 2*), как значения весов осциллируют вокруг минимума ($w_{0,[n]}$, $w_{1,[n]}$).

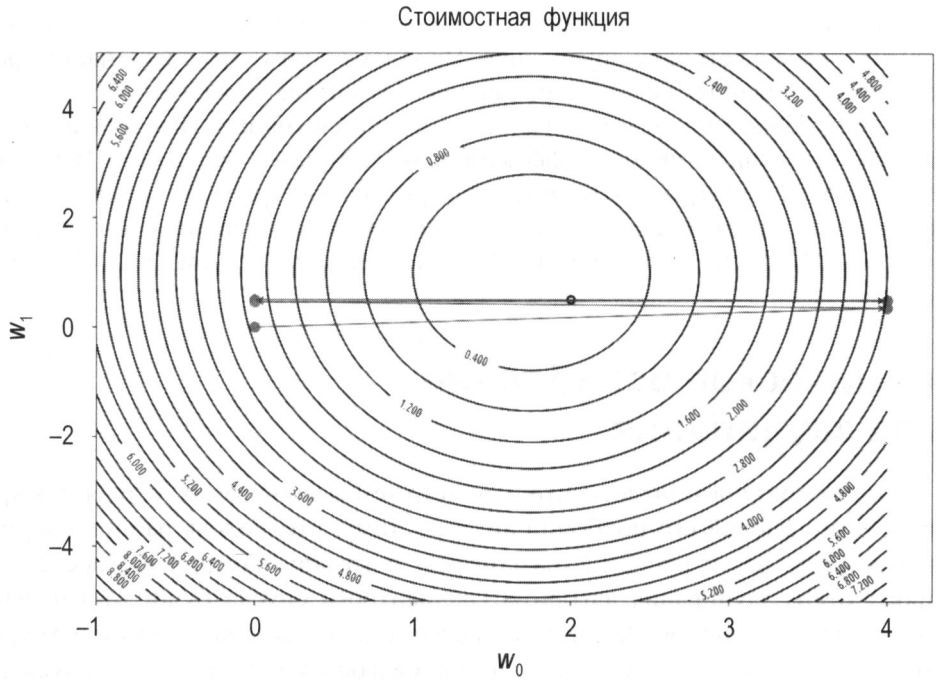


РИС. 4.1. Иллюстрация алгоритма градиентного спуска. Здесь выбран темп заучивания $\gamma = 2$

Здесь ясно видна проблема слишком большого темпа заучивания. Алгоритму не удается сойтись, потому что выполняемые им шаги слишком велики для того, чтобы суметь приблизиться к минимуму. Разные оценки w_n обозначены на рис. 4.1 точками. Минимум обозначен окружностью примерно посередине изображения.

Вы, возможно, заметили, что в нашем алгоритме мы приняли довольно важное решение (не указывая его явно): для каждой итерации мы держим темп заучивания постоянным. Но для этого нет никаких оснований. Наоборот, это довольно плохая идея. Интуитивно большой темп заучивания побудит схождение двигаться быстро в самом начале, но как только вы приблизитесь к минимуму, вы захотите использовать гораздо меньший темп заучивания для того, чтобы алгоритм сходился к минимуму наиболее эффективно. Мы хотим иметь темп заучивания, который начинается (относительно) будучи большим, а затем уменьшается вместе с итерациями. Но как он должен уменьшаться? Сегодня используется несколько методов, и в следующем разделе мы рассмотрим те из них, которые применяются наиболее часто, а также их реализации на Python и в TensorFlow. Мы будем решать ту же задачу, для которой были сгенерированы рис. 4.1 и 2.12, и сравним поведение разных алгоритмов. Прежде чем читать последующие разделы, рекомендуется уделить немного времени на ревизию раздела *главы 2*, посвященного градиентному спуску, с целью освежить этот материал в памяти.

Итерации или эпохи?

Прежде чем перейти к рассмотрению различных методов, следует пролить некоторый свет на вопрос, о каких итерациях мы говорим? Являются они эпохами? Технически, это не так. Итерация — это некий шаг, во время которого вы обновляете веса. Рассмотрим, например, мини-пакетный градиентный спуск. В этом случае итерация выполняется после каждого мини-пакета (при обновлении весов). Рассмотрим набор данных Zalando из *главы 3*: 60 000 тренировочных образцов и мини-пакет размером 50 наблюдений. В этом случае у вас будет 1200 итераций на эпоху. Для снижения темпа заучивания важно не число эпох, а число обновлений, которые вы выполняете на весах. Если вы использовали стохастический градиентный спуск (SGD) на наборе данных Zalando (обновляли веса после каждого наблюдения), то у вас будет 60 000 итераций, и вам может потребоваться уменьшить заучивание больше, чем при мини-пакетном градиентном спуске, потому что он обновляется чаще. В случае пакетного градиентного спуска, когда вы обновляете вес после одного полного прохода по тренировочным данным, вы будете обновлять темп заучивания ровно один раз в каждую эпоху.

ПРИМЕЧАНИЕ. Итерации в динамическом ослаблении темпа заучивания касаются шага в алгоритме, в котором обновляются веса. Например, если вы используете стохастический градиентный спуск на наборе данных Zalando из *главы 3* с мини-пакетом размером 50 наблюдений, то за одну эпоху (один проход по 60 000 тренировочных наблюдений) у вас будет 1200 итераций.

Это очень важно понимать правильно. Если вы это поймете, то сможете надлежаще выбирать параметры разных алгоритмов ослабления темпа заучивания. Если вы выбираете их, думая, что темп заучивания обновляется только после каждой эпохи, то будете получать большие ошибки.

ПРИМЕЧАНИЕ. В каждом алгоритме, который уменьшается динамически, темп заучивания будет вводить новые гиперпараметры, которые вы должны оптимизировать, добавляя некоторую сложность в процесс отбора модели.

Ступенчатое ослабление

Метод ступенчатого ослабления — самый простой в использовании. Он состоит в сокращении темпа заучивания вручную в исходном коде и жесткого кодирования изменений на основе того, что, по мнению разработчика, будет работать. Например, как побудить алгоритм градиентного спуска сходиться на рис. 4.1, начиная с $\gamma = 2$? Рассмотрим следующее ослабление (в котором через j мы обозначили число итераций):

$$\gamma = \begin{cases} 2 & \text{при } j < 4; \\ 0{,}4 & \text{при } j \geq 4. \end{cases}$$

Включив это выражение в Python-код

```
gamma0 = 2.0
if (j < 4):
    gamma = gamma0
elif j>=4:
    gamma = gamma0 /5.0
```

мы получим сходящийся алгоритм (рис. 4.2). Здесь был выбран первоначальный темп заучивания $\gamma_0 = 2$ и, начиная с итерации 4, был использован $\gamma = 0{,}4$. Разные оценки w_n обозначены точками. Минимум обозначен окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью упростить обновление весов.

Первые шаги являются большими, а затем, когда на итерации 4 мы уменьшаем темп заучивания до 0,4, они становятся меньше, и градиентный спуск способен сойтись к минимуму (табл. 4.1). С помощью этой простой модификации мы добились хорошего результата. Проблема, правда, в том, что при работе со сложными наборами данных и моделями (как мы делали в *главе 3*) этот процесс требует (если он работает) большого числа тестов. Вам придется сократить темп заучивания в несколько раз и найти правильную итерацию и значения снижения темпа заучивания, что представляет собой действительно сложную задачу, причем настолько, что она фактически непригодна для использования, если только вы не имеете дело с очень легкими наборами данных и сетями. Этот метод также не очень стабилен и в зави-

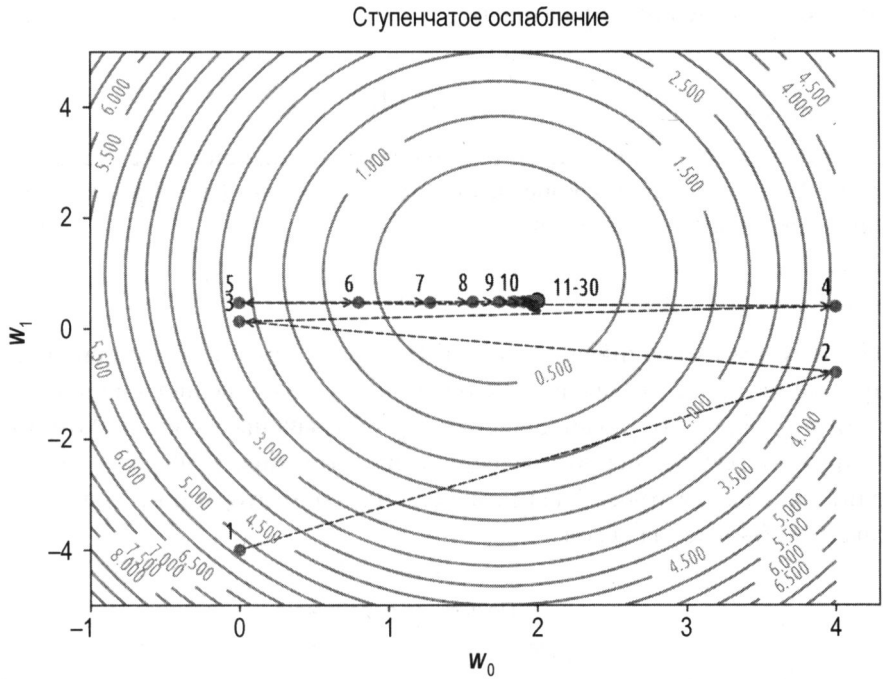


РИС. 4.2. Иллюстрация алгоритма градиентного спуска с методом ступенчатого ослабления

симости от имеющихся данных может потребовать непрерывной регулировки. TL;DR[1]: не используйте его.

Таблица 4.1. *Дополнительные гиперпараметры*

Гиперпараметр	Пример
Итерация, на которой алгоритм обновит темп заучивания	В этом примере мы выбираем номер итерации 4
Темп заучивания после каждого изменения (многочисленные значения)	В этом примере мы имели $\gamma = 2$ с итерации 1 до итерации 3 и $\gamma = 0{,}4$, начиная с итерации 4

Пошаговое ослабление

Пошаговое ослабление представляет собой несколько более автоматический метод. Он уменьшает темп заучивания на постоянный коэффициент через определенное число итераций.

[1] В случае если вы не знаете, TL;DR — это акроним от английского выражения "too long; didn't read", т. е. "слишком длинно; не читал". Аналогичное выражение в русском интернет-жаргоне — "ниасилил, многабукаф". Он предназначен для описания текста, который был проигнорирован, потому что он слишком длинный. (Источник: **https://en.wikipedia.org/wiki/TL;DR**.)

Математически это можно записать как

$$\gamma = \frac{\gamma_0}{\left[j/D+1\right]},$$

где $[a]$ — это целочисленная часть a; D (позже обозначена в исходном коде как `epoch_drop`) — целочисленная константа, которую можно регулировать. Например, с помощью следующих строк кода

```
epochs_drop = 2
gamma = gamma0 / (np.floor(j/epochs_drop)+1)
```

мы снова получим сходящийся алгоритм. На рис. 4.3 был выбран первоначальный темп заучивания $\gamma_0 = 2$, и каждые 2 итерации темп заучивания уменьшается в соответствии с $\gamma_0/\left[j/2+1\right]$. Разные оценки w_n обозначены точками. Минимум обозначен окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью облегчить отслеживание обновления весов.

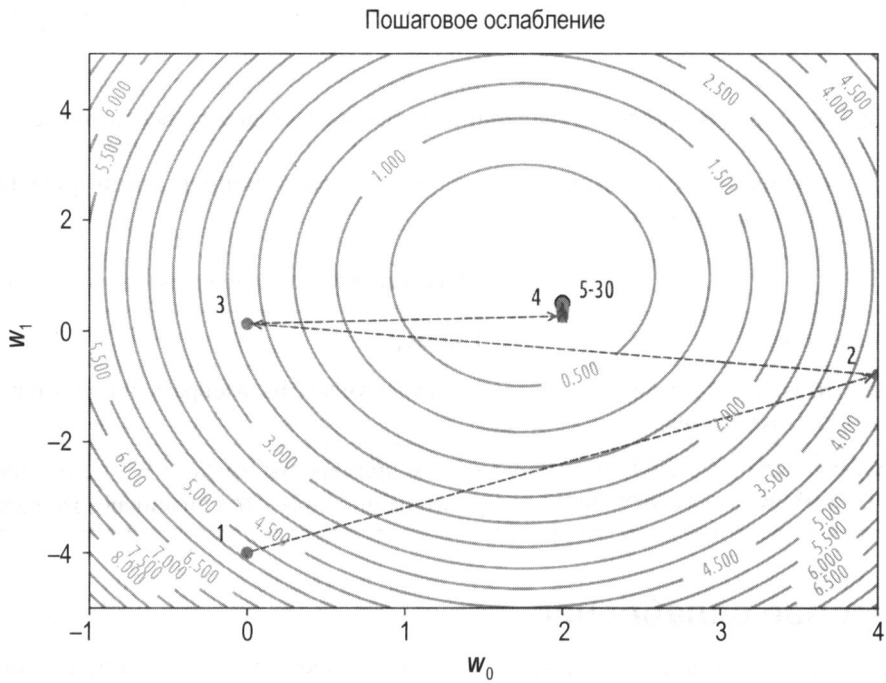


РИС. 4.3. Иллюстрация алгоритма градиентного спуска с пошаговым ослаблением

Важно иметь представление о динамике снижения темпа заучивания. Ведь вы не хотите, чтобы темп заучивания был близок к нулю после нескольких итераций; в противном случае схождение никогда не будет успешным. На рис. 4.4 показано, как быстро (или медленно) снижается темп заучивания для трех значений D.

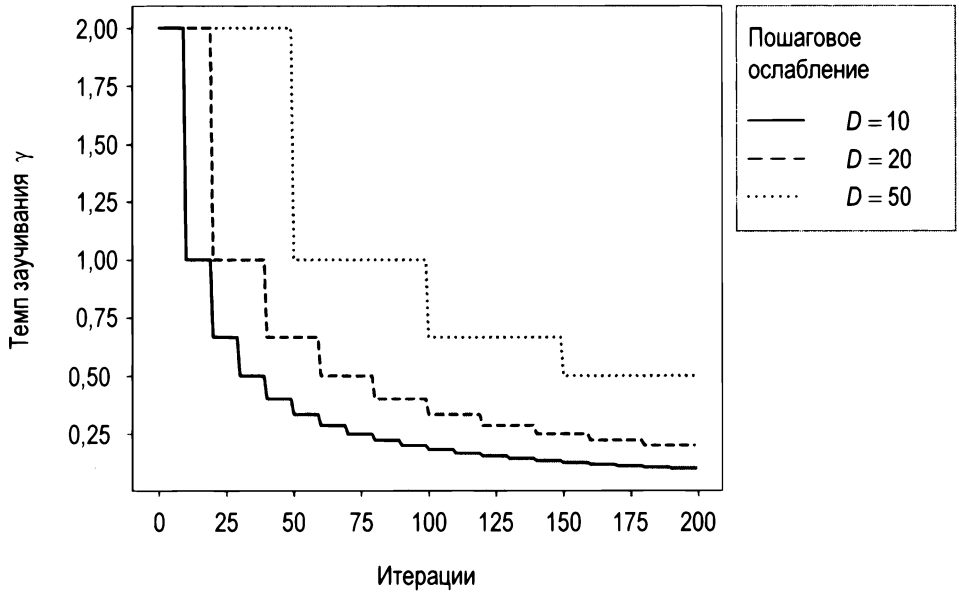


РИС. 4.4. Уменьшение темпа заучивания для трех значений D: 10, 20 и 50 с использованием алгоритма пошагового ослабления

Обратите внимание, например, что при $D = 10$ темп заучивания примерно в 10 раз меньше после 100 итераций! Если вы слишком ускорите снижение темпа заучивания, то увидите, что схождение останавливается после всего нескольких итераций. Всегда старайтесь иметь представление о динамике уменьшения γ.

ПРИМЕЧАНИЕ. Хороший способ прочувствовать динамику снижения темпа заучивания — попытаться определить, через сколько итераций γ будет в 10 раз меньше первоначального значения. Имейте в виду, что если у вас $\gamma = \gamma_0/10$ после $10D$ итераций, то вы получите $\gamma = \gamma_0/100$ только после $100D$ итераций, а $\gamma = \gamma_0/10^3$ только после $10^3 D$ итераций, и т. д. Если это происходит, то на вопрос "Какое значение требуется?" можно ответить, только надлежаще протестировав темп на нескольких значениях D.

Рассмотрим конкретный пример. Предположим, вы тренируете свою модель с 10^5 наблюдениями в течение 5000 эпох, с мини-пакетом размером 50 наблюдений и первоначальным темпом заучивания $\gamma_0 = 0{,}2$. Если вы, не задумываясь, выберете $D = 10$ (табл. 4.2), то у вас будет

$$\gamma = \frac{\gamma_0}{20\,000} = \frac{0{,}2}{20\,000} = 10^{-5}$$

только после 100 эпох, и поэтому от использования 5000 эпох вы особо не выиграете, если сократите темп заучивания так быстро.

Таблица 4.2. *Дополнительные гиперпараметры*

Гиперпараметр	Пример
Параметр D	$D = 10$

Обратно-временное ослабление

Еще один способ обновления темпа заучивания основан на формуле, именуемой обратно-временным ослаблением (inverse time decay):

$$\gamma = \frac{\gamma_0}{1 + vj},$$

где v — это параметр темпа ослабления. На рис. 4.5 показано сравнение снижения темпа заучивания для трех параметров v: 0,01; 0,1 и 0,8. Также видно, как темп заучивания уменьшается для трех разных значений v. Отметим, что ось y была построена на логарифмической шкале для того, чтобы упростить сравнение сущности изменений.

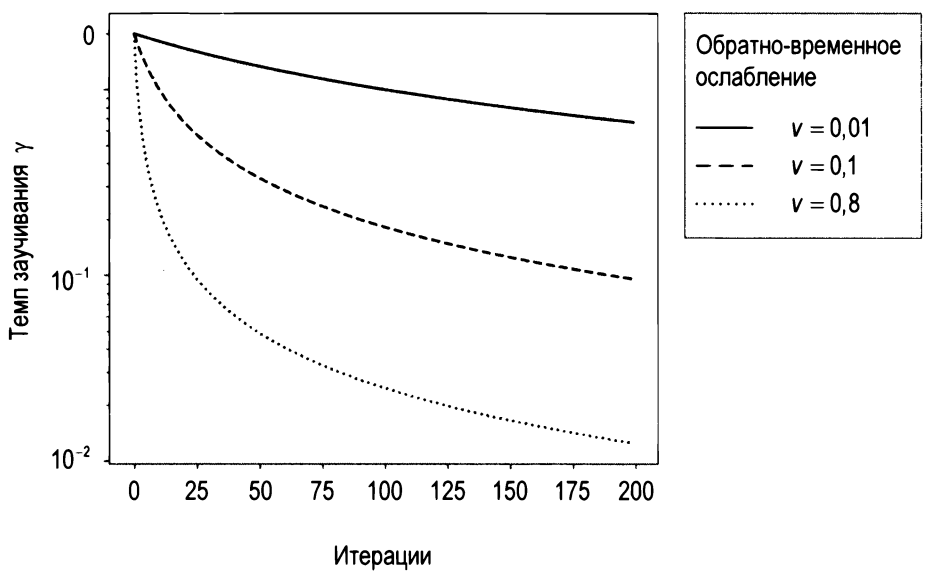


РИС. 4.5. Снижение темпа заучивания для трех значений v: 0,01; 0,1 и 0,8 с использованием алгоритма обратно-временного ослабления

Этот метод тоже побуждает рассмотренный в *главе 2* алгоритм градиентного спуска сходиться. На рис. 4.6 показано, как при выборе $v = 0,2$ веса сходятся к минимальному местоположению после нескольких итераций. На рис. 4.6 выбран первоначальный темп заучивания $\gamma_0 = 2$ и использован алгоритм обратно-временного ослабления с $v = 0,2$. Разные оценки w_n обозначены точками. Минимум обозначен

окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью облегчить отслеживание обновления весов.

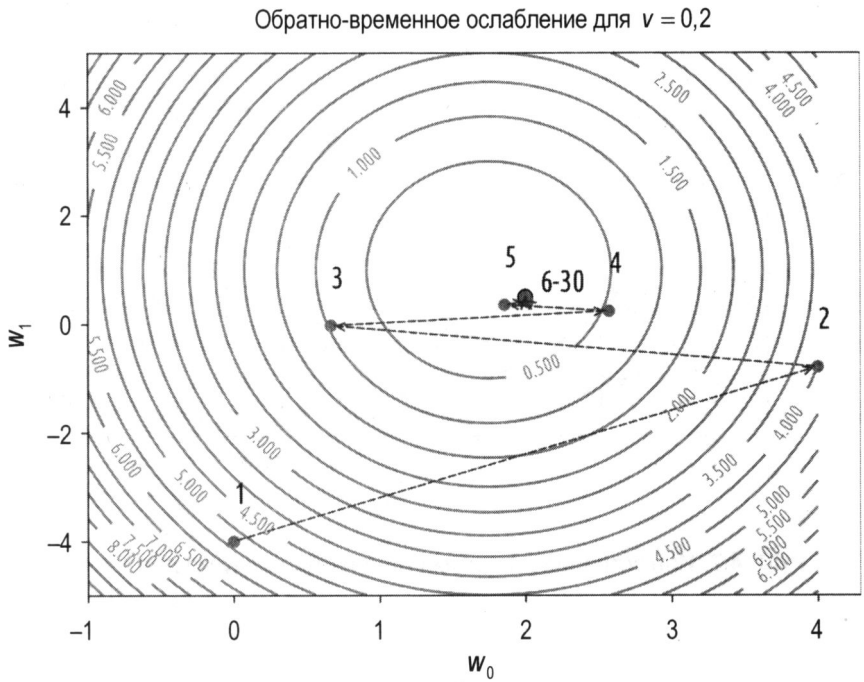


РИС. 4.6. Иллюстрация алгоритма градиентного спуска с обратно-временным ослаблением для $v = 0,2$

Очень интересно посмотреть, что произойдет, если мы возьмем большее значение v. На рис. 4.7 был выбран первоначальный темп заучивания $\gamma_0 = 2$ и был использован алгоритм обратно-временного ослабления с $v = 1,5$. Разные оценки w_n обозначены точками. Минимум обозначен окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью облегчить отслеживание обновления весов.

Ситуация, изображеннная на рис. 4.7, все объясняет. Увеличение v побуждает темп заучивания уменьшаться быстрее, и, следовательно, для достижения минимума требуется больше шагов, потому что темп заучивания становится все меньше по сравнению с тем, что происходит на рис. 4.6. Можно сравнить поведение стоимостных функций для двух значений v. На рис. 4.8, *а* стоимостная функция сопоставлена с числом эпох. На первый взгляд кажется, что они сходятся одинаково быстро. Но давайте увеличим участок вокруг $J = 0$, как показано на рис. 4.8, *б*. Ясно видно, как при $v = 0,2$ (табл. 4.3) схождение происходит гораздо быстрее, потому что темп заучивания больше, чем для $v = 1,5$.

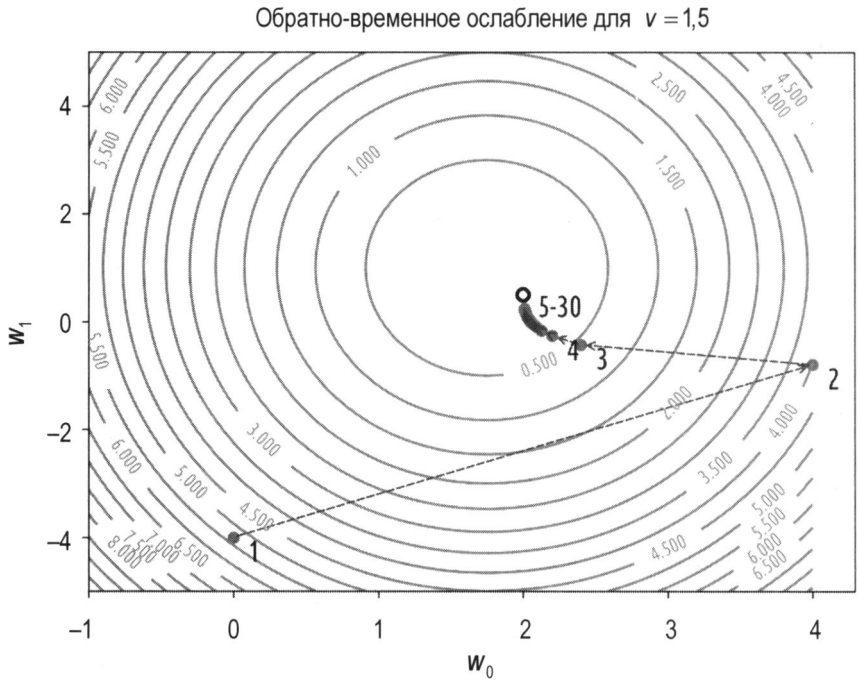


РИС. 4.7. Иллюстрация алгоритма градиентного спуска для $v = 1{,}5$

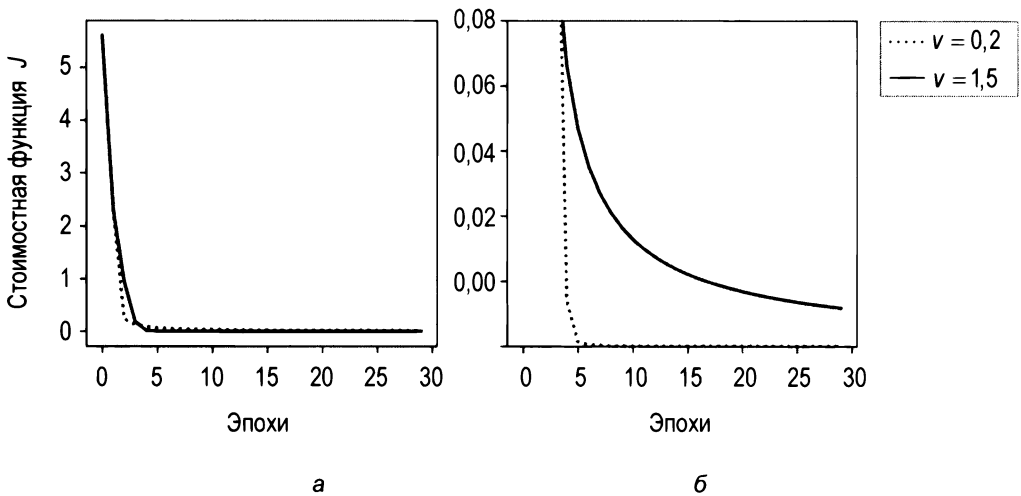


РИС. 4.8. Стоимостная функция в сопоставлении с числом эпох. На графике (*а*) строится весь диапазон значений, которые принимают стоимостные функции. На графике (*б*) участок вокруг $J = 0$ был увеличен для того, чтобы показать, насколько быстрее стоимостные функции уменьшаются при меньших значениях v

Таблица 4.3. *Дополнительные гиперпараметры*

Гиперпараметр	Пример
Темп ослабления v	$v = 0,2$

Экспоненциальное ослабление

Еще один способ сокращения темпа заучивания основывается на формуле экспоненциального ослабления:

$$\gamma = \gamma_0 v^{j/T}.$$

Взгляните на рис. 4.9, чтобы получить представление о темпе заучивания. Отметим, что ось y была построена на логарифмической шкале для того, чтобы упростить сравнение сущности изменений. Обратите внимание, что при $v = 0,01$ после 200 итераций (не эпох) темп заучивания уже в 1000 раз меньше, чем в самом начале!

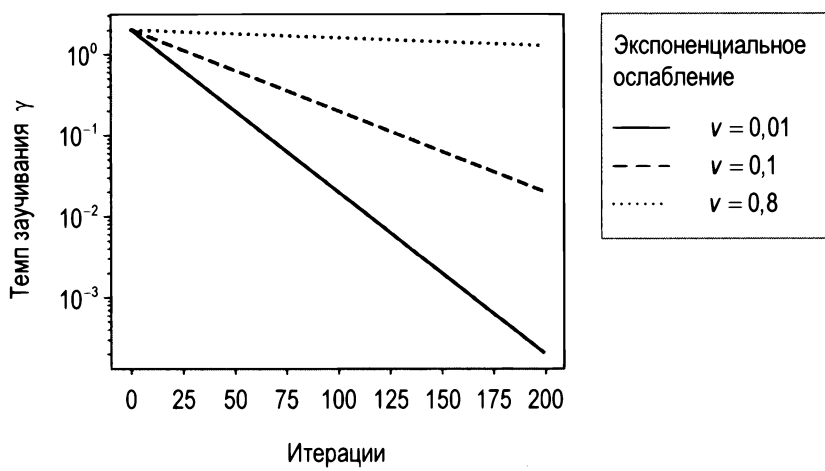


РИС. 4.9. Уменьшение темпа заучивания для трех значений v: 0,01; 0,1; 0,8 и $T = 100$ с использованием алгоритма экспоненциального ослабления

Мы можем применить этот метод к задаче с $v = 0,2$ и $T = 3$ (табл. 4.4), и алгоритм снова сойдется. На рис. 4.10 выбран первоначальный темп заучивания $\gamma_0 = 2$ и использован алгоритм экспоненциального ослабления с $v = 0,2$ и $T = 3$. Разные оценки w_n отмечены точками. Минимум обозначен окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью облегчить отслеживание обновления весов.

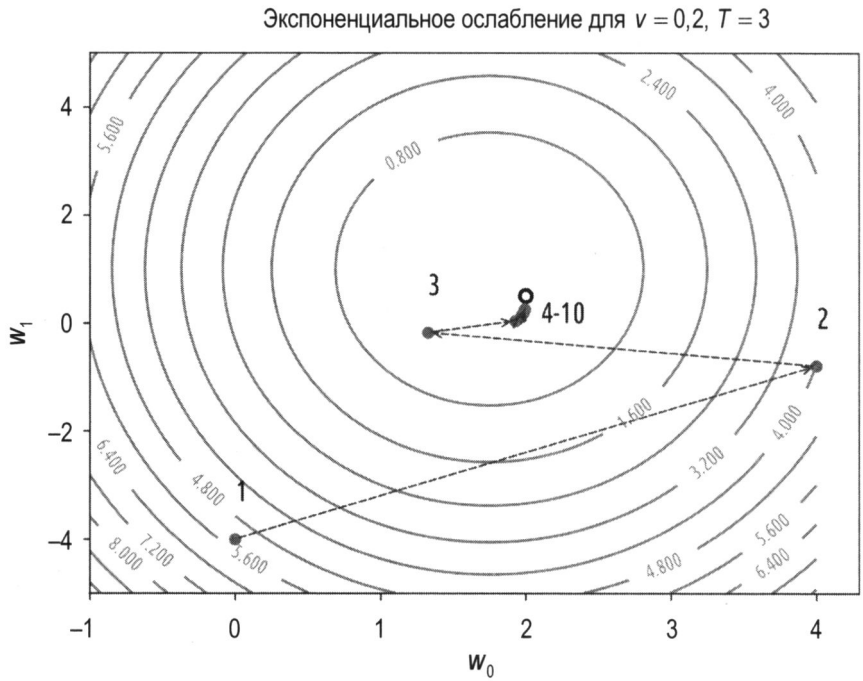


РИС. 4.10. Иллюстрация алгоритма градиентного спуска для экспоненциального ослабления

Таблица 4.4. *Дополнительные гиперпараметры*

Гиперпараметр	**Пример**
Темп ослабления v	$v = 0{,}2$
Шаги ослабления T	$T = 3$

Естественное экспоненциальное ослабление

Еще один способ сокращения темпа заучивания основывается на формуле, именуемой естественным экспоненциальным ослаблением:

$$\gamma = \gamma_0 e^{-vj}.$$

Этот случай особенно интересен тем, что позволяет получать информацию о нескольких важных вещах. Рассмотрим первый рис. 4.11, чтобы сравнить, как разные значения v соотносятся с разными уменьшениями темпа заучивания.

Хотелось бы обратить внимание на значения по оси y (данная шкала является логарифмической). Для $v = 0{,}8$ после 200 итераций темп заучивания составляет 10^{-64} от первоначального значения! Практически нулевой. Это означает, что уже после нескольких итераций обновления больше происходить не смогут, т. к. темп заучива-

ния слишком мал. Для того чтобы вы смогли представить величину 10^{-64}, стоит упомянуть, что размер атома водорода составляет "всего" примерно 10^{-11} м! Поэтому если вы не очень осторожны с выбором v, то уйдете не очень далеко.

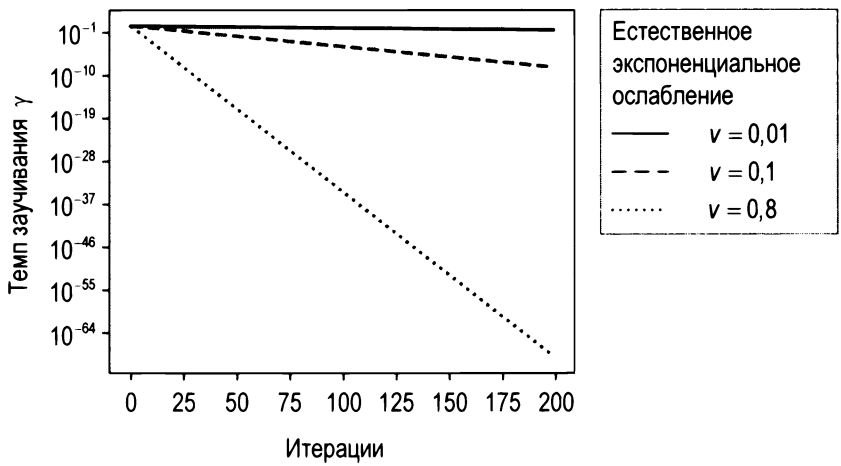


РИС. 4.11. Снижение темпа заучивания при трех значениях v: 0,01; 0,1; 0,8 и $T = 100$ с использованием алгоритма естественного экспоненциального ослабления. Отметим, что ось y была построена на логарифмической шкале для того, чтобы упростить сравнение сущности изменений. Обратите внимание, что для v = 0,8 после 200 итераций (не эпох) темп заучивания уже в 10^{64} раза меньше, чем в самом начале

Рассмотрим рис. 4.12, для которого веса строились по мере того, как они обновлялись с помощью алгоритма градиентного спуска, для двух значений темпа заучивания: 0,2 (пунктирная линия) и 0,5 (непрерывная линия).

Для того чтобы проверить схождение, нам нужно увеличить участок вокруг минимума. Вы увидите это на рис. 4.13. В случае если вас интересует вопрос, почему минимум, похоже, находится в другом месте относительно контурных линий, как на рис. 4.12, то это вызвано тем, что эти контурные линии не те же самые, потому что на рис. 4.13 мы находимся гораздо ближе к минимуму.

Теперь мы видим вещи, которые снова все объясняют. Непрерывная линия относится к $v = 0,5$; следовательно, темп заучивания уменьшается гораздо быстрее, и ему не удается достичь минимума. На самом деле, всего после 7 итераций мы имеем $\gamma = 0,06$, а после 20 итераций у нас $\gamma = 9 \cdot 10^{-5}$. Данное значение является настолько малым, что процессу схождения больше не удается продолжиться с разумной скоростью! Опять же, очень поучительно проверить снижение стоимостной функции для этих двух параметров (рис. 4.14).

На основе графика на рис. 4.14, *б* видно, как стоимостная функция для $v = 0,5$ не достигает нуля и становится практически постоянной, потому что темп заучивания слишком мал. Вы можете подумать, что если применить больше итераций, то данный метод в конечном итоге сойдется. Но это не так. Взгляните на рис. 4.15,

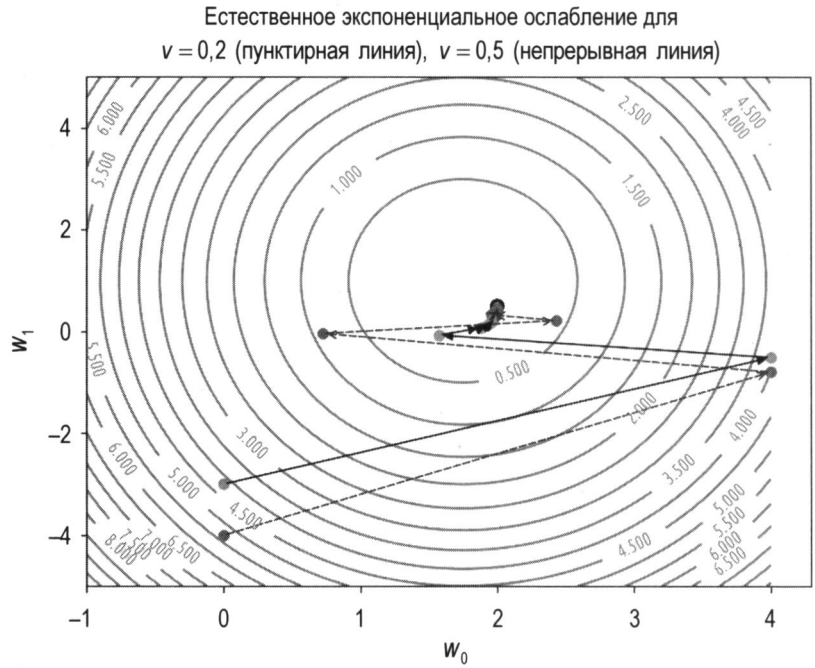


РИС. 4.12. Иллюстрация алгоритма градиентного спуска с естественным экспоненциальным ослаблением

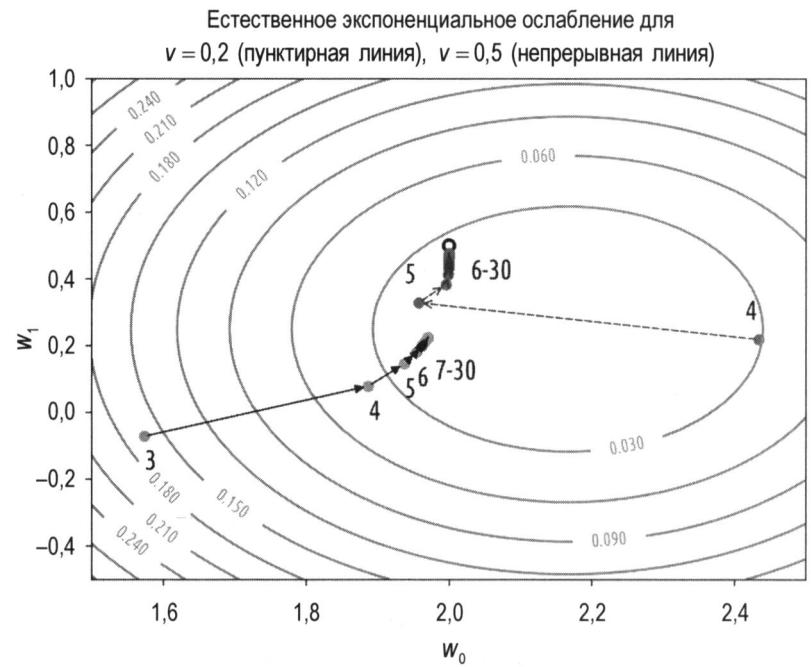


РИС. 4.13. Иллюстрация алгоритма градиентного спуска с увеличением участка вокруг минимума. Здесь использовались те же методы и параметры, что и на рис. 4.12

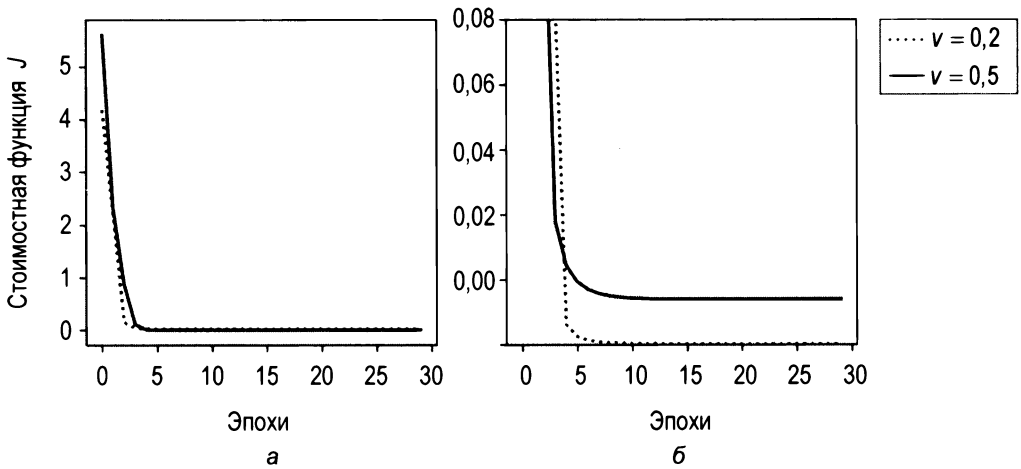


РИС. 4.14. Стоимостная функция в сопоставлении с числом эпох естественного экспоненциального ослабления при двух значениях $v = 0,2$ и $v = 0,5$. На графике (*а*) строится весь диапазон значений, которые принимают стоимостные функции. На графике (*б*) участок вокруг $J = 0$ был увеличен для того, чтобы показать, насколько быстрее стоимостные функции уменьшаются при меньших значениях v

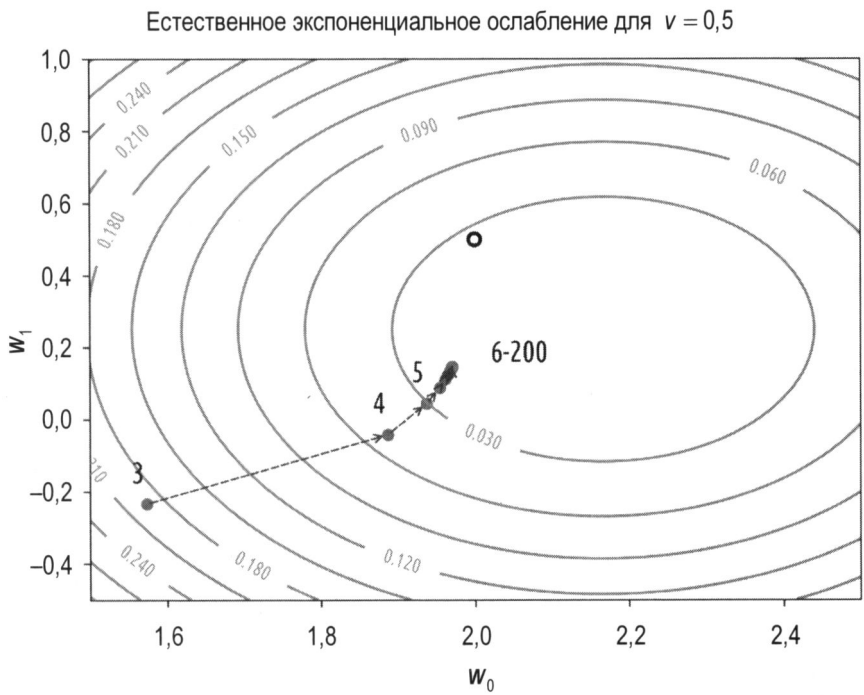


РИС. 4.15. Иллюстрация алгоритма градиентного спуска с увеличенным участком вокруг минимума для 200 итераций

который показывает, что процесс схождения фактически останавливается из-за того, что через некоторое время темп заучивания становится почти нулевым. На рисунке выбран первоначальный темп заучивания $\gamma_0 = 2$ и использован алгоритм естественного экспоненциального ослабления с $v = 0{,}5$. Алгоритму градиентного спуска не удается достичь минимума. Разные оценки w_n обозначены точками. Минимум обозначен окружностью примерно посередине изображения. Алгоритм теперь способен сходиться. Каждая точка была помечена номером итерации с целью облегчить отслеживание обновления весов.

Давайте проверим темп заучивания во время этого процесса для $v = 0{,}5$ (рис. 4.16). Проверьте значения вдоль оси *y*. Темп заучивания достигает 10^{-40} примерно после 175 итераций. Во всех практических аспектах он равен нулю. Алгоритм градиентного спуска обновлять веса больше не будет независимо от того, сколько итераций вы ему дадите.

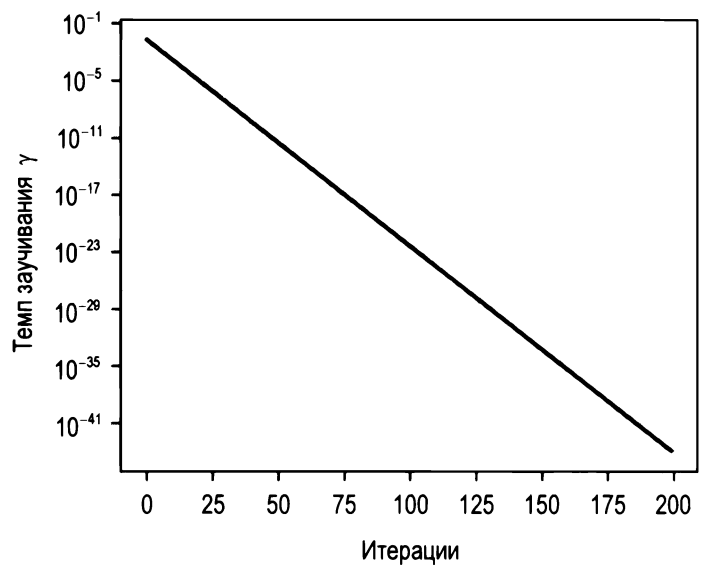


РИС. 4.16. Темп заучивания в сопоставлении с числом итераций с естественным экспоненциальным ослаблением для $v = 0{,}5$. Отметим, что ось *y* является логарифмической для того, чтобы лучше выделить изменение γ

В заключение давайте сравним все методы, поместив их на один график для того, чтобы получить представление об относительном их поведении. На рис. 4.17 показаны три графика, отслеживающие снижение темпа заучивания для каждого метода с разными параметрами.

ПРИМЕЧАНИЕ. Вы должны быть осведомлены о динамике снижения темпа заучивания для того, чтобы избежать ситуаций, когда она становится практически нулевой и полностью останавливает схождение.

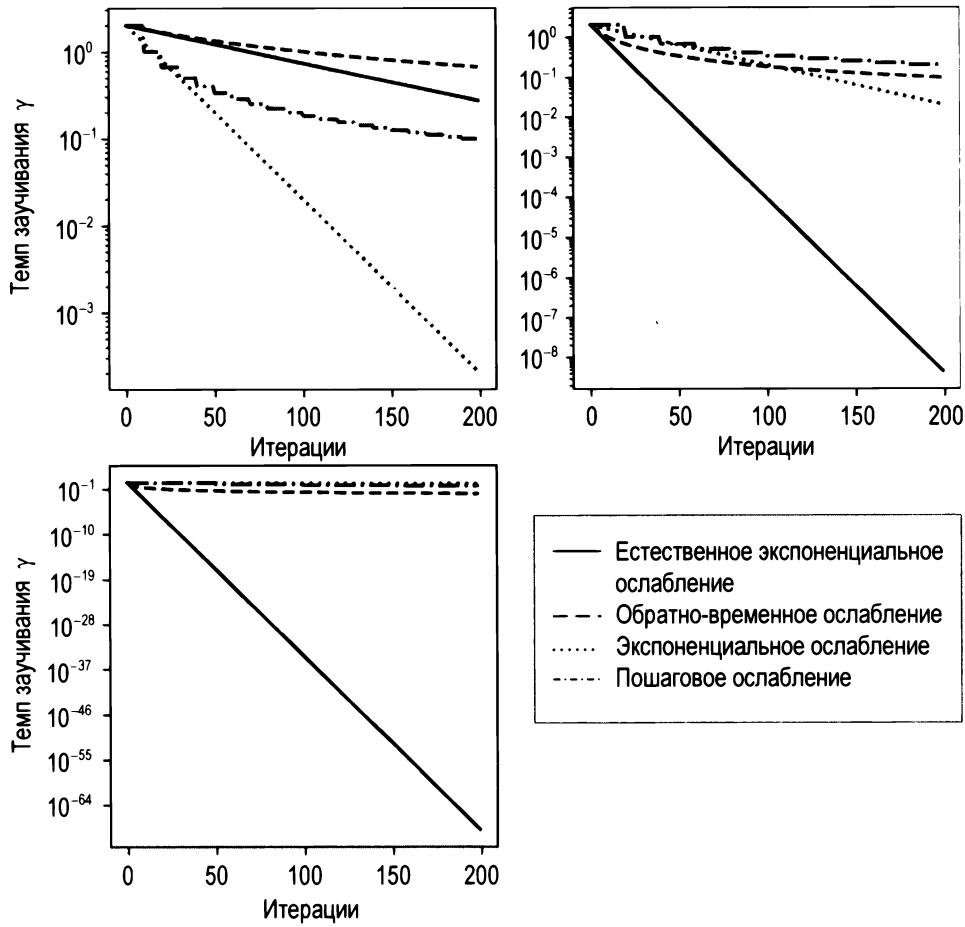


РИС. 4.17. Сравнение разных параметров ослабления темпа заучивания для указанного алгоритма

Реализация в TensorFlow

Необходимо кратко остановиться на том, как в библиотеке TensorFlow реализованы только что разъясненные методы, потому что существует несколько деталей, о которых вы должны знать. В TensorFlow можно найти следующие функции, которые выполняют динамическое ослабление темпа заучивания[2]:

- экспоненциальное ослабление — `tf.train.exponential_decay` (**https://goo.gl/fiE2ML**);
- обратно-временное ослабление — `tf.train.inverse_time_decay` (**https://goo.gl/GXK6MX**);

[2] Обратитесь к соответствующему обзору в официальной документации TensorFlow по адресу: **https://goo.gl/vpFNp7**.

- естественное экспоненциальное ослабление — `tf.train.natural_exp_decay` (**https://goo.gl/cGJe52**);
- пошаговое ослабление — `tf.train.piecewise_constant` (**https://goo.gl/bL47ZD**);
- многочленное ослабление — `tf.train.polynomial_decay` (**https://goo.gl/zuJWNo**).

Многочленное ослабление является немного более сложным способом уменьшения темпа заучивания. Оно не рассматривалось, потому что редко используется, но вы можете прочитать документацию по нему на веб-сайте TensorFlow и получить представление о том, как оно работает.

TensorFlow использует дополнительный параметр, который дает немного больше гибкости. Возьмем, к примеру, метод обратно-временного ослабления. Уравнение для ослабления темпа заучивания было таким:

$$\gamma = \frac{\gamma_0}{1 + vj},$$

где мы имеем два параметра: γ_0 и v. TensorFlow использует три параметра:

$$\gamma = \frac{\gamma_0}{1 + \frac{vj}{v_{ds}}},$$

где v_{ds} в TensorFlow называется шагом ослабления, `decay_step`. В официальной документации TensorFlow вы найдете формулу, которая на языке Python записывается так:

```
decayed_learning_rate = learning_rate / (1 + decay_rate * global_step /
                                                              decay_step)
```

и связывает язык TensorFlow с нашими обозначениями следующим образом:

- `global_step` — j (число итераций);
- `decay_rate` — v (темп ослабления);
- `decay_step` — v_{ds} (шаг ослабления);
- `learning_rate` — γ_0 (первоначальный темп заучивания).

Вы можете задаться вопросом: зачем нужен этот дополнительный параметр? Данный параметр, говоря математически, избыточен. Можно установить v равным тому же значению v/v_{ds} и получить тот же самый результат. Проблема, практически, заключается в том, что j (число итераций) весьма быстро становится очень большим, и, следовательно, для того чтобы иметь возможность получить разумное снижение темпа заучивания, параметру v может потребоваться принять очень малые значения. Цель параметра v_{ds} состоит в шкалировании числа итераций. Например, для этого параметра вы можете задать значение $v_{ds} = 10^5$, что приведет к снижению темпа заучивания в масштабе 10^5 итераций вместо каждой отдельной итерации. Если у вас огромный набор данных с 10^8 наблюдениями, и вы используете мини-

пакет размером 50 наблюдений, то для каждой эпохи вы получите $2 \cdot 10^6$ итераций. Предположим, вы хотите, чтобы после 100 эпох темп заучивания составлял 1/5 от первоначального значения. Для этого вам понадобится $v = 2 \cdot 10^{-8}$, довольно малое значение, которое, что важнее, зависит от размера набора данных и размера мини-пакета. Если вы, условно говоря, выполните "нормализацию" числа итераций, то можете выбрать значение для v, которое может оставаться постоянным, в случае если, к примеру, вы решите изменить размер мини-пакета. Существует дополнительная практическая причина (более важная, чем та, которая обсуждалась только что). Она заключается в следующем: функция TensorFlow имеет дополнительный ступенчатый параметр — `staircase`, который может принимать значение `True` или `False`. Если установлено значение `True`, то используется следующая функция:

$$\gamma = \frac{\gamma_0}{1 + v\left\lfloor \frac{j}{v_{ds}} \right\rfloor}.$$

И, следовательно, вы получаете обновление не непрерывно, а только для каждой итерации v_{ds}. На рис. 4.18 показана разница между $v = 0{,}5$ и $v_{ds} = 20$ для 200 итераций. Вы, возможно, захотите удерживать темп заучивания постоянным в течение десяти эпох и обновлять его только после них.

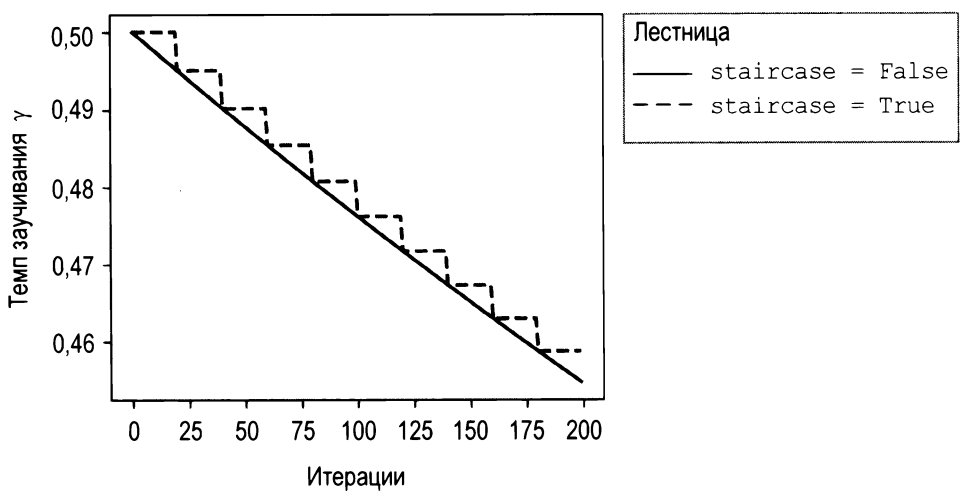


РИС. 4.18. Ослабление темпа заучивания с двумя вариантами, полученными с параметром TensorFlow `staircase`, равным `True` и `False`

Такие же параметры необходимы функциям `tf.train.inverse_time_decay`, `tf.train.natural_exp_decay` и `tf.train.polynomial_decay`. Они работают таким же образом, а предназначение дополнительного параметра только что было описано. В случае если вам понадобится этот дополнительный параметр, то далее для того чтобы не запутаться при реализации этих методов в TensorFlow, будет показано,

как его реализовывать для обратно-временного ослабления, но он работает точно так же для всех других типов. Вам понадобится следующий дополнительный фрагмент кода:

```
initial_learning_rate = 0.1
decay_steps = 1000
decay_rate = 0.1
global_step = tf.Variable(0, trainable = False)
learning_rate_decay = tf.train.inverse_time_decay(initial_learning_rate,
                                    global_step, decay_steps, decay_rate)
```

Затем вы должны изменить строку кода, в которой указывается используемый оптимизатор.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate_decay).minimize(
    cost, global_step = global_step)
```

Единственное отличие — дополнительный параметр в функции `minimize`: `global_step = global_step`. Функция `minimize` при каждом обновлении будет обновлять переменную `global_step` номером итерации. Вот и все. Это работает точно таким же образом и для других функций.

Разница существует только для кусочно постоянной функции `piecewise_constant`, которая требует других параметров: `x`, `boundaries` и `values`. Например (из документации TensorFlow):

> ...использовать темп заучивания 1,0 для первых 100 000 шагов, 0,5 для шагов 100 001–110 000 и 0,1 для любых дополнительных шагов.

Для этого потребуются границы и значения:

```
boundaries = [100000, 110000]
values = [1.0, 0.5, 0.1]
```

Строки кода

```
boundaries = [b1,b2,b3, ..., bn]
values = [l1,l12,l23,l34, ..., ln]
```

дадут темп заучивания `l1` перед итерациями `b1`, `l12` между итерациями `b1` и `b2`, `l23` между итерациями `b2` и `b3` и т. д. Имейте в виду, что с этим методом в исходном коде необходимо вручную задавать все значения и границы. И если вы хотите проверить каждую их комбинацию с целью увидеть, хорошо ли она работает, то для этого потребуется немного терпения. Реализация алгоритма пошагового ослабления в TensorFlow будет выглядеть так:

```
global_step = tf.Variable(0, trainable=False)
boundaries = [100000, 110000]
values = [1.0, 0.5, 0.1]
learning_rate = tf.train.piecewise_constant(global_step, boundaries, values)
```

Применение описанных методов к набору данных Zalando

Давайте попробуем применить методы, с которыми вы только что познакомились, к реалистичному сценарию. Для этого мы воспользуемся набором данных Zalando, используемым в *главе 3*. Обратитесь к *главе 3* еще раз для того, чтобы узнать, как загружать этот набор данных и подготавливать данные к работе. В конце той главы мы написали функции для построения многослойной модели и функцию для ее тренировки. Рассмотрим модель с четырьмя скрытыми слоями с 20 нейронами каждый. Давайте сравним, как модель учится с первоначальным темпом заучивания 0,01, поддержим ее постоянной, а затем применим алгоритм обратно-временного ослабления, начиная с $\gamma_0 = 0{,}1$, $v = 0{,}1$ и $v_{ds} = 10^3$ (рис. 4.19).

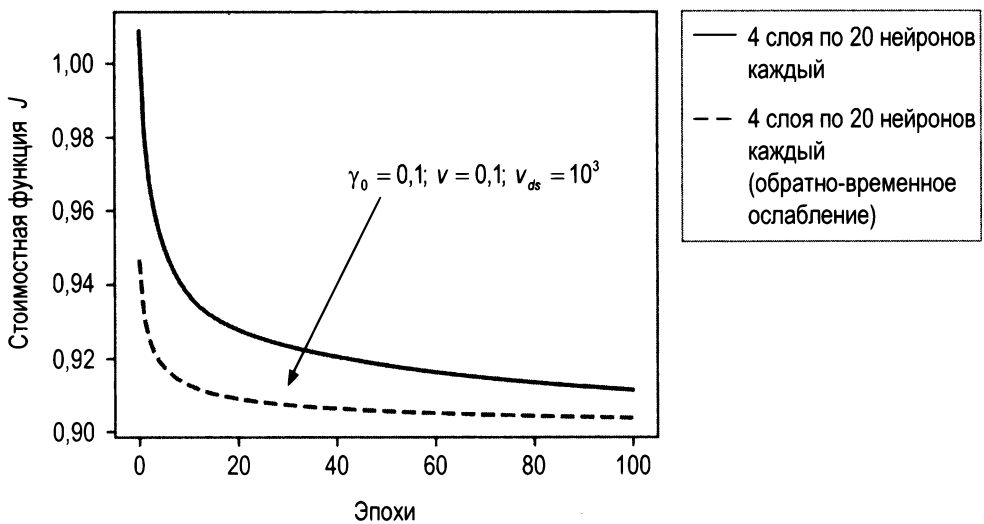


РИС. 4.19. Поведение стоимостной функции для нейронной сети из 4 слоев по 20 нейронов каждый, применительно к набору данных Zalando. Непрерывная линия показывает модель с постоянным темпом заучивания $\gamma_0 = 0{,}01$. Пунктирная линия показывает сеть, в которой мы использовали алгоритм обратно-временного ослабления, с $\gamma_0 = 0{,}1$, $v = 0{,}1$ и $v_{ds} = 10^3$

Таким образом, даже при первоначальном в 10 раз большем темпе заучивания данный алгоритм намного эффективнее. В ряде исследований было доказано, что применение динамического темпа заучивания делает заучивание быстрее и эффективнее, как мы уже заметили в данном случае.

ПРИМЕЧАНИЕ. Если только вы не используете оптимизационные алгоритмы, которые предусматривают изменение темпа заучивания во время тренировки (вы увидите их в следующих разделах), то обычно рекомендуется использовать динамическое ослабление темпа заучивания. Оно стабилизирует и, как правило, ускоряет заучивание. Его недостатком является то, что вам нужно настраивать больше гиперпараметров.

Обычно при использовании динамического ослабления темпа заучивания неплохо начинать с первоначального темпа заучивания γ_0, который больше обычно используемого. Поскольку γ уменьшается, то, как правило, проблемы не возникают, и схождение в самом начале (будем надеяться) происходит быстрее. Как и следовало ожидать, какие-то фиксированные правила улучшения работы метода отсутствуют. Каждый случай и набор данных различаются, и всегда требуется проводить определенное тестирование для того, чтобы увидеть, какое значение параметра дает наилучшие результаты.

Широко используемые оптимизаторы

До сих пор для минимизации стоимостной функции мы применяли градиентный спуск. Этот способ не самый эффективный, и существуют определенные модификации данного алгоритма, которые делают его намного быстрее и эффективнее. Данная область исследований активно развивается, и вы найдете невероятное число алгоритмов на основе разных идей, которые ускоряют заучивание. Здесь будут рассмотрены самые показательные и известные из них: Momentum, RMSProp и Adam. Дополнительный материал, с которым можно ознакомиться с целью изучить самые экзотические алгоритмы, был написан С. Рудером (S. Ruder) в статье "An overview of gradient descent optimization algorithms" ("Обзор оптимизационных алгоритмов градиентного спуска"), которая доступна по адресу **https://goo.gl/KgKVgG**. Указанная статья рассчитана не для начинающих и требует обширной математической подготовки, но в ней дан обзор таких необычных алгоритмов, как Adagrad, Adadelta и Nadam. Кроме того, в ней рассматриваются схемы обновления весов, применимые в распределенных средах, такие как Hogwild!, Downpour SGD и многие другие. Ее чтение, безусловно, стоит вашего времени.

Для того чтобы уяснить основную идею алгоритма Momentum (и частично также RMSProp и Adam), вы сначала должны понять, что такое экспоненциально взвешенные средние.

Экспоненциально взвешенные средние

Предположим, что вы измеряете величину θ (это может быть температура в помещении) во времени, например один раз в день. У вас будет ряд измерений, которые мы можем обозначить с помощью θ_i, где i изменяется на интервале от 1 до определенного числа N. Наберитесь терпения, если в начале вы не увидите большого смысла; тем не менее давайте рекурсивно определим величину v_n как

$$v_0 = 1;$$
$$v_1 = \beta v_0 + (1-\beta)\theta_1;$$
$$v_2 = \beta v_1 + (1-\beta)\theta_2$$

и т. д., где β — это вещественное число, такое, что $0 < \beta < 1$. Как правило, мы могли бы написать *n*-й член как

$$v_n = \beta v_{n-1} + (1-\beta)\theta_n .$$

Теперь запишем все члены, v_1, v_2 и т. д., просто как функцию от β и θ_i (не рекурсивно). Для v_2 мы имеем:

$$v_2 = \beta(\beta v_0 + (1-\beta)\theta_1) + (1-\beta)\theta_2 = \beta^2 + (1-\beta)(\beta\theta_1 + \theta_2),$$

для v_3:

$$v_3 = \beta^3 + (1-\beta)(\beta^2\theta_1 + \beta\theta_2 + \theta_3).$$

Обобщая, получаем:

$$v_n = \beta^n + (1-\beta)(\beta^{n-1}\theta_1 + \beta^{n-2}\theta_2 + ... + \theta_n).$$

Или, более элегантно (без трех точек):

$$v_n = \beta^n + (1-\beta)\sum_{i=1}^{n}\beta^{n-i}\theta_i .$$

Теперь давайте попробуем понять, что означает эта формула. Прежде всего, обратите внимание, что если мы выберем $v_0 = 0$, то член β^n исчезнет. Сделаем это (зададим $v_0 = 0$) и посмотрим, что останется:

$$v_n = (1-\beta)\sum_{i=1}^{n}\beta^{n-i}\theta_i .$$

Следите за моей мыслью? Теперь самое интересное. Определим свертку между двумя последовательностями[3]. Рассмотрим две последовательности: x_n и h_n. Свертка между ними (которую мы обозначаем символом $*$) задается формулой:

$$x_n * h_n = \sum_{k=-\infty}^{\infty} x_k h_{n-k} .$$

Теперь, поскольку число измерений величины θ_i является конечным, мы будем иметь:

$$\theta_k = \theta, \quad k > n,\ k \le 0 .$$

Следовательно, мы можем написать v_n как свертку в виде:

$$v_n = \theta_n * b_n ,$$

где мы определили

$$b_n = (1-\beta)\beta^n .$$

---

[3] Вообще говоря, последовательность — это перечисляемая коллекция объектов.

Для того чтобы получить представление о том, что это значит, давайте построим совместный график θ_n, b_n и v_n. Для этого будем считать, что θ_n имеет гауссову форму (точная форма не имеет значения, пример служит исключительно в иллюстративных целях), и возьмем $\beta = 0{,}9$ (рис. 4.20).

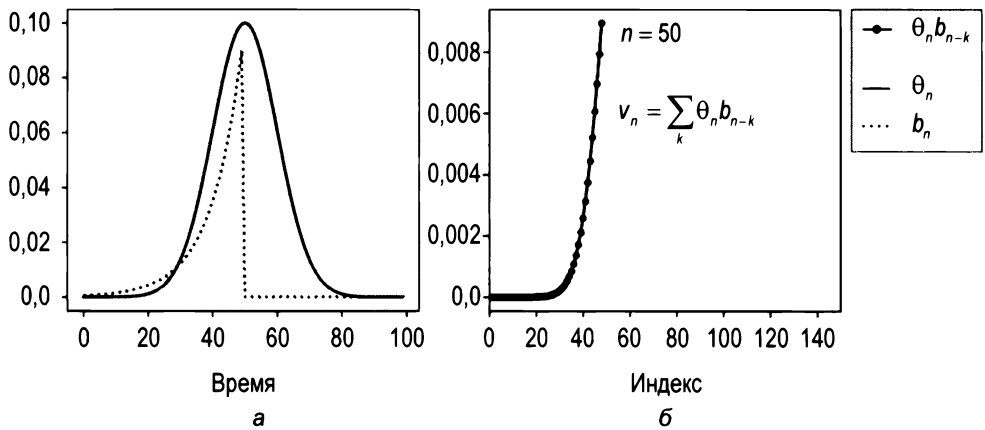


РИС. 4.20. График (*а*), показывающий θ_n (сплошная линия) и b_n (пунктирная линия) вместе, и график (*б*), показывающий точки, которые нужно просуммировать, чтобы получить v_n для $n = 50$

Давайте вкратце рассмотрим рис. 4.20. Гауссова кривая (θ_n) будет свернута с помощью b_n, в результате чего мы получим v_n. Этот результат можно увидеть на рис. 4.20, *б*. Все эти члены, $(1-\beta)\beta^{n-i}\theta_i$ для $i = 1, ..., 50$ (изображены справа), будут просуммированы и дадут v_{50}. Интуитивно v_n — это среднее значение всех θ_n для $n = 1, ..., 50$. Далее каждый член умножается на член b_n, т. е. на 1 для $n = 50$, затем уменьшаясь по мере уменьшения n до 1.[4] В сущности, мы имеем взвешенное среднее с экспоненциально уменьшающимся весом (отсюда и его имя). Члены на удалении от $n = 50$ становятся все менее и менее релевантными, а члены вблизи к $n = 50$ — более релевантными. Экспоненциально взвешенное среднее также является скользящим средним. Для каждого n добавляются все предшествующие члены, каждый из которых умножается на вес (b_n).

Важно понимать, почему существует эта составляющая $1-\beta$ в b_n. Почему бы не выбрать только β^n? Причина очень проста. Сумма b_n по всем положительным n равна 1. Давайте посмотрим, почему. Рассмотрим следующее уравнение:

$$\sum_{k=1}^{\infty} b_k = (1-\beta)\sum_{k=1}^{\infty}\beta^n = (1-\beta)\lim_{N\to\infty}\frac{1-\beta^{N+1}}{1-\beta} = (1-\beta)\frac{1}{1-\beta} = 1,$$

[4] Здесь имеется в виду, что по мере старения данных их весовой коэффициент снижается. В этом состоит суть экспоненциального взвешенного среднего. $n = 50$ — это самая свежая точка данных. — *Прим. пер.*

в котором мы использовали тот факт, что для $i<1$ мы имеем

$$\lim_{N\to\infty}\beta^{N+1}=0$$

и для геометрического ряда мы имеем

$$\sum_{k=1}^{n}ar^{k-1}=\frac{a\left(1-r^{n}\right)}{1-r}.$$

Описанный нами алгоритм вычисления v_n есть не что иное, как свертка величины θ_i с рядом, сумма которого равна 1, и имеет форму $(1-\beta)\beta^i$.

ПРИМЕЧАНИЕ. Экспоненциально взвешенное среднее v_n ряда θ_n является сверткой $v_n=\theta_n * b_n$ величины θ_i с $b_n=(1-\beta)\beta^i$, где b_n имеет свойство, что ее сумма положительных значений n равна 1. Оно имеет интуитивный смысл скользящего среднего, в котором каждый член умножается на веса, заданные последовательностью b_n.

Как видно на рис. 4.21, в котором показан ряд b_n для разных значений i, по мере того, как вы выбираете все меньшие значения β, число точек θ_n, чей вес значительно отличается от нуля, уменьшается.

Этот метод лежит в основе оптимизатора Momentum (импульсного оптимизатора) и более продвинутых самообучающихся алгоритмов, и в следующих разделах вы увидите, как он работает на практике.

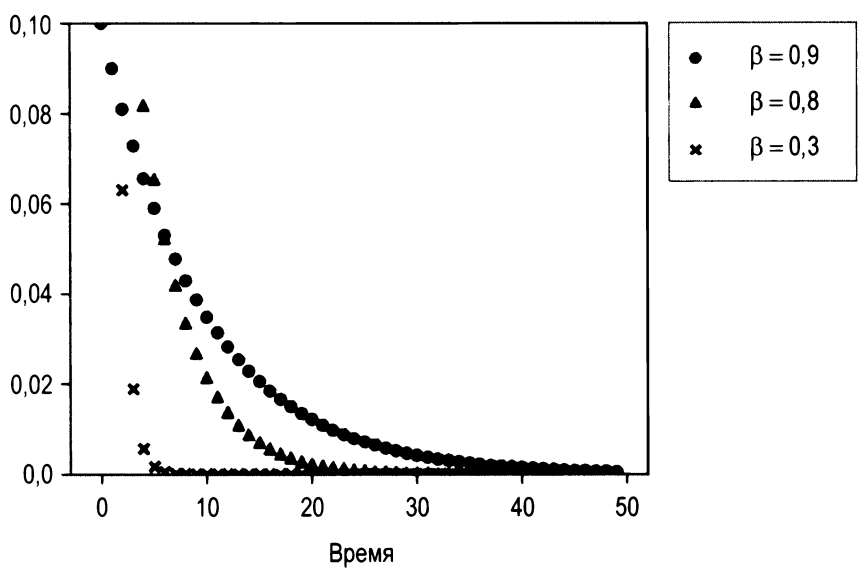


РИС. 4.21. Ряд b_n для трех значений β : 0,9; 0,8 и 0,3. Обратите внимание, что по мере уменьшения β ряд существенно отличается от нуля для возрастающе меньшего числа значений вокруг $n=0$

Momentum

Напомним, что при простом градиентном спуске обновления весов вычисляются с помощью уравнений:

$$\begin{cases} \mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right); \\ b_{[n+1]} = b_{[n]} - \gamma \dfrac{\partial J\left(\mathbf{w}_{[n]}, b_{[n]}\right)}{\partial b}. \end{cases}$$

Идея в основе импульсного оптимизатора Momentum заключается в том, чтобы использовать экспоненциально взвешенные средние поправок градиента, а затем применять их для обновления весов. В более математическом плане мы вычисляем

$$\begin{cases} \mathbf{v}_{w,[n+1]} = \beta \mathbf{v}_{w,[n]} - (1-\beta) \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right); \\ v_{b,[n+1]} = \beta v_{b,[n]} - (1-\beta) \dfrac{\partial J\left(\mathbf{w}_{[n]}, b_{[n]}\right)}{\partial b}, \end{cases}$$

а затем мы будем выполнять обновления с помощью уравнений:

$$\begin{cases} \mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma \mathbf{v}_{w,[n]}; \\ b_{[n+1]} = b_{[n]} - \gamma v_{b,[n]}, \end{cases}$$

где обычно выбираются $\mathbf{v}_{w,[0]} = \mathbf{0}$ и $v_{b,[0]} = 0$. Как теперь понятно из обсуждения экспоненциально взвешенных средних в предыдущем разделе, это означает, что вместо использования производных стоимостных функций относительно весов мы обновляем веса с помощью скользящего среднего производных. Опыт показывает, что, как правило, поправкой смещения теоретически можно пренебречь.

ПРИМЕЧАНИЕ. Для обновления весов алгоритм Momentum использует экспоненциальное взвешенное среднее производных стоимостной функции относительно весов. Благодаря этому используются не только производные на данной итерации, но и учитывается прошлое поведение. Нередко бывает, что алгоритм осциллирует вокруг минимума, а не сходится напрямую. Этот алгоритм может уходить с плато гораздо эффективнее, чем стандартный градиентный спуск.

В книгах или блогах иногда можно найти несколько иную формулировку (для краткости здесь приведено уравнение только для весов $\mathbf{w}$).

$$\mathbf{v}_{w,[n+1]} = \gamma \mathbf{v}_{w,[n]} - \eta \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right).$$

Идея и смысл остаются прежними. Эта математическая формулировка просто немного другая. По моему мнению, описанный метод интуитивно воспринимается легче с помощью понятия свертки последовательности и взвешенных средних, чем эта вторая формулировка. Вы найдете еще одну формулировку (используемую в TensorFlow), которая имеет вид

$$v_{w,[n+1]} = \eta' v_{w,[n]} + \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right),$$

где η' носит название TensorFlow Momentum (верхний индекс t указывает, что эта переменная используется в TensorFlow). В этой формулировке обновление весов принимает вид

$$\mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma' v_{w,[n+1]} = \mathbf{w}_{[n]} - \gamma' \left(\eta' v_{w,[n]} + \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right)\right) =$$
$$= \mathbf{w}_{[n]} - \gamma' \eta' v_{w,[n]} - \gamma' \nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right),$$

где опять же верхний индекс t указывает, что переменная используется в TensorFlow. Хотя эта формулировка выглядит по-иному, она полностью эквивалентна формулировке, которая была дана в начале этого раздела.

$$\mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma\beta v_{w,[n]} - \gamma(1-\beta)\nabla_{\mathbf{w}} J\left(\mathbf{w}_{[n]}, b_{[n]}\right).$$

Формулировка TensorFlow и та, которая обсуждалась ранее, являются эквивалентными, если мы выбираем

$$\begin{cases} \eta = \dfrac{\beta}{1-\beta}; \\ \gamma' = \gamma(1-\beta). \end{cases}$$

Это можно увидеть, сравнив два разных уравнения для обновлений весов. Как правило, в реализациях TensorFlow используются значения около $\eta = 0{,}9$, и они обычно работают хорошо.

Оптимизатор Momentum реализуется в TensorFlow чрезвычайно легко. Просто замените `GradientDescentOptimizer` на

```
tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0.9)
```

Оптимизатор Momentum почти всегда сходится быстрее, чем простой градиентный спуск.

ПРИМЕЧАНИЕ. Было бы неправильно сравнивать разные параметры в различных оптимизаторах. Например, в разных алгоритмах темп заучивания имеет другой смысл. В отличие от этого следует сравнивать наилучшую скорость схождения, которую можно достичь с помощью нескольких оптимизаторов, независимо от выбора параметров. Сравнивать градиентный спуск для темпа заучивания 0,01 с алгоритмом Adam (о нем позже) для того же темпа заучивания не имеет большого смысла. Для того чтобы решить, какой из оптимизаторов использовать, вы должны сравнивать их с параметрами, которые дают вам лучшее и самое быстрое схождение.

На рис. 4.22 показана стоимостная функция задачи, рассмотренной в предыдущем разделе, для простого градиентного спуска (с $\gamma = 0{,}05$) и для оптимизатора Momentum (с $\gamma = 0{,}05$ и $\eta = 0{,}9$). Обратите внимание, как оптимизатор Momentum осциллирует вокруг минимума. Главная трудность — увидеть на шкале y, что с оптимизатором Momentum стоимость J достигает гораздо меньшего значения.

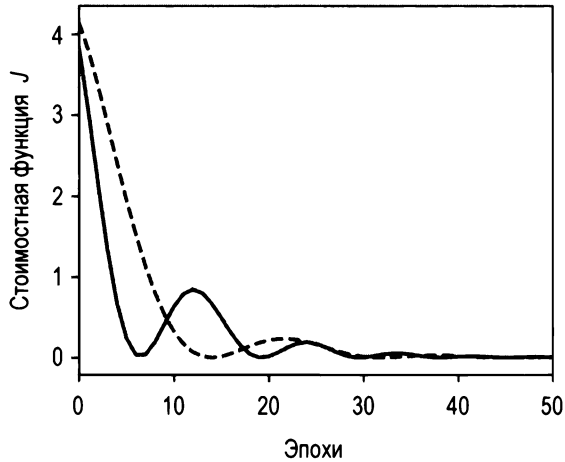


РИС. 4.22. Стоимостная функция в сопоставлении с числом эпох для простого градиентного спуска (с $\gamma = 0{,}05$) и для оптимизатора Momentum (с $\gamma = 0{,}05$ и $\eta = 0{,}9$). Видно, как оптимизатор Momentum немного осциллирует вокруг минимума

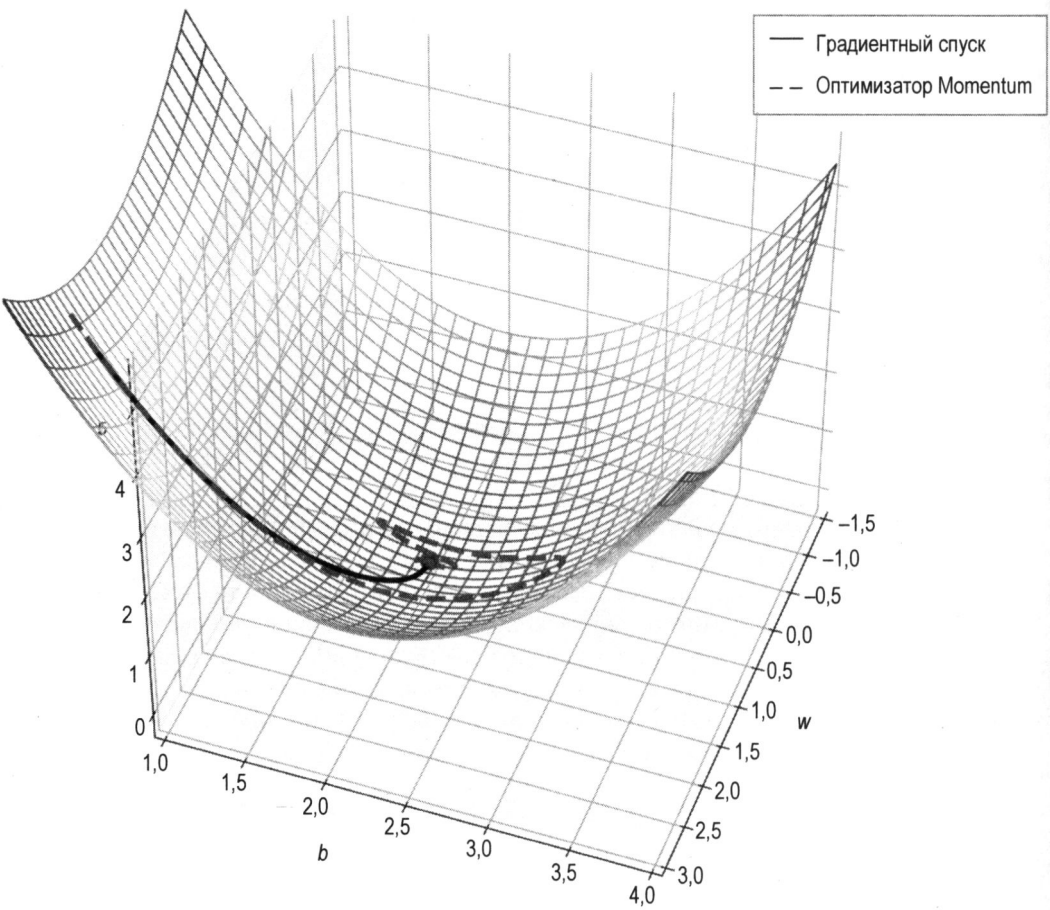


РИС. 4.23. Трехмерный график поверхности стоимостной функции J. Непрерывная линия — это траектория, которую оптимизатор градиентного спуска выбирает — вдоль максимальной крутизны, как и ожидалось. Пунктирная линия — это линия, которую оптимизатор Momentum выбирает по мере осцилляции вокруг минимума

Интереснее проверить, как оптимизатор Momentum выбирает свою траекторию по поверхности стоимостной функции. На рис. 4.23 показан трехмерный график поверхности стоимостной функции. Непрерывная линия — это траектория, которую оптимизатор градиентного спуска выбирает вдоль максимальной крутизны, как и ожидалось. Пунктирная линия — это линия, которую оптимизатор Momentum выбирает по мере осцилляции вокруг минимума.

Далее приведем убедительное доказательство того, что в схождении оптимизатор Momentum работает быстрее и лучше. Для этого давайте проверим, как ведут себя два оптимизатора на весовой плоскости. На рис. 4.24 показаны траектории, выбранные двумя оптимизаторами. На правом графике вы видите увеличение вокруг минимума. Видно, как градиентный спуск после 100 эпох не достигает минимума, хотя кажется, что он выбирает более прямой путь к минимуму. Он приближается очень близко, но недостаточно. Оптимизатор Momentum осциллирует вокруг минимума и достигает его очень эффективно.

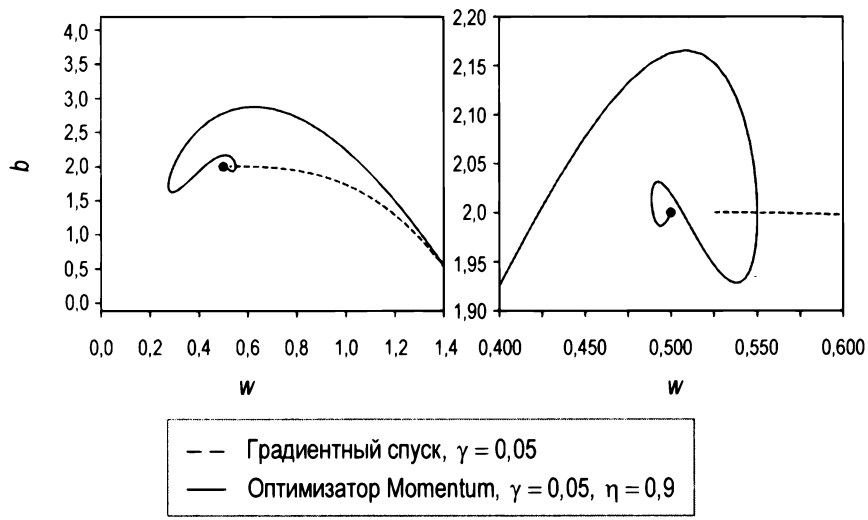


РИС. 4.24. Траектории, выбираемые двумя оптимизаторами. Правый график показывает увеличение вокруг минимума. Видно, как импульс достигает минимума после осциллирования вокруг него, в то время как градиентному спуску не удается достичь его за 100 эпох

RMSProp

Теперь перейдем к более сложному, но обычно более эффективному оптимизатору. Сначала будут представлены математические уравнения, а затем мы их сравним с другими, которые мы видели до этого. На каждой итерации нам нужно вычислять

$$\begin{cases} \mathbf{S}_{\mathbf{w},[n+1]} = \beta_2 \mathbf{S}_{\mathbf{w},[n]} + (1-\beta_2)\nabla_{\mathbf{w}} J(\mathbf{w}, b) \circ \nabla_{\mathbf{w}} J(\mathbf{w}, b); \\ S_{b,[n+1]} = \beta_2 S_{b,[n]} + (1-\beta_2)\dfrac{\partial J(w, b)}{\partial b} \circ \dfrac{\partial J(w, b)}{\partial b}, \end{cases}$$

где символ ∘ обозначает поэлементное произведение. Затем мы обновляем веса с помощью уравнений

$$\begin{cases} \mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \dfrac{\gamma \nabla_{\mathbf{w}} J(\mathbf{w}, b)}{\sqrt{\mathbf{S}_{\mathbf{w},[n+1]} + \varepsilon}}; \\ b_{[n+1]} = b_{[n]} - \gamma \dfrac{\partial J(\mathbf{w}, b)}{\partial b} \dfrac{1}{\sqrt{S_{b,[n]} + \varepsilon}}. \end{cases}$$

Таким образом, сначала вы определяете экспоненциальное взвешенное среднее величин $\mathbf{S}_{\mathbf{w},[n+1]}$ и $S_{b,[n+1]}$, а затем используете их для модифицирования производных, которые вы используете для обновления весов. Величина ε, обычно $\varepsilon = 10^{-8}$, предназначена для того, чтобы не дать знаменателю сходиться к нулю, в случае если величины $\mathbf{S}_{\mathbf{w},[n+1]}$ и $S_{b,[n+1]}$ сходятся к нулю. Интуитивная идея состоит в том, что если производная — большая, то величины $\mathbf{S}$ будут большими; поэтому составляющие

$$1/\sqrt{\mathbf{S}_{\mathbf{w},[n+1]} + \varepsilon}$$

или

$$1/\sqrt{S_{b,[n]} + \varepsilon}$$

будут меньше, и заучивание замедлится. Обратное также верно, поэтому если производные малы, то заучивание ускорится. Этот алгоритм убыстрит заучивание для

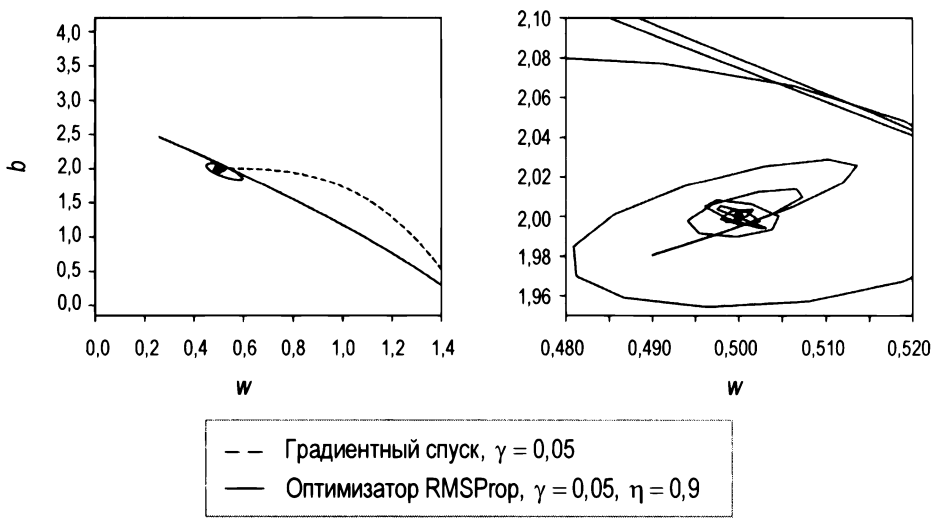


РИС. 4.25. Траектория, выбираемая по направлению к минимуму стоимостной функции простым градиентным спуском и оптимизатором RMSProp. Последний делает петли вокруг минимума и затем его достигает. Градиентный спуск за то же самое число эпох к нему даже не приближается. Обратите внимание на шкалу правого графика. Уровень увеличения очень высок. Мы смотрим на крайний крупный план (траектория градиентного спуска на этой шкале даже не видна) вокруг минимума

параметров, которые его замедляют. В TensorFlow эта оптимизация опять же особенно легко реализуется с помощью следующей строки кода:

```
optimizer = tf.train.RMSPropOptimizer(learning_rate,
                                      momentum = 0.9).minimize(cost)
```

Давайте проверим, какую траекторию выбирает оптимизатор. На рис. 4.25 видно, что RMSProp осциллирует вокруг минимума. В то время как градиентный спуск его не достигает, алгоритм RMSProp перед тем, как его достичь, успевает сделать несколько петель вокруг него.

На рис. 4.26 вы видите ту же самую траекторию по поверхности стоимостной функции в трехмерной среде.

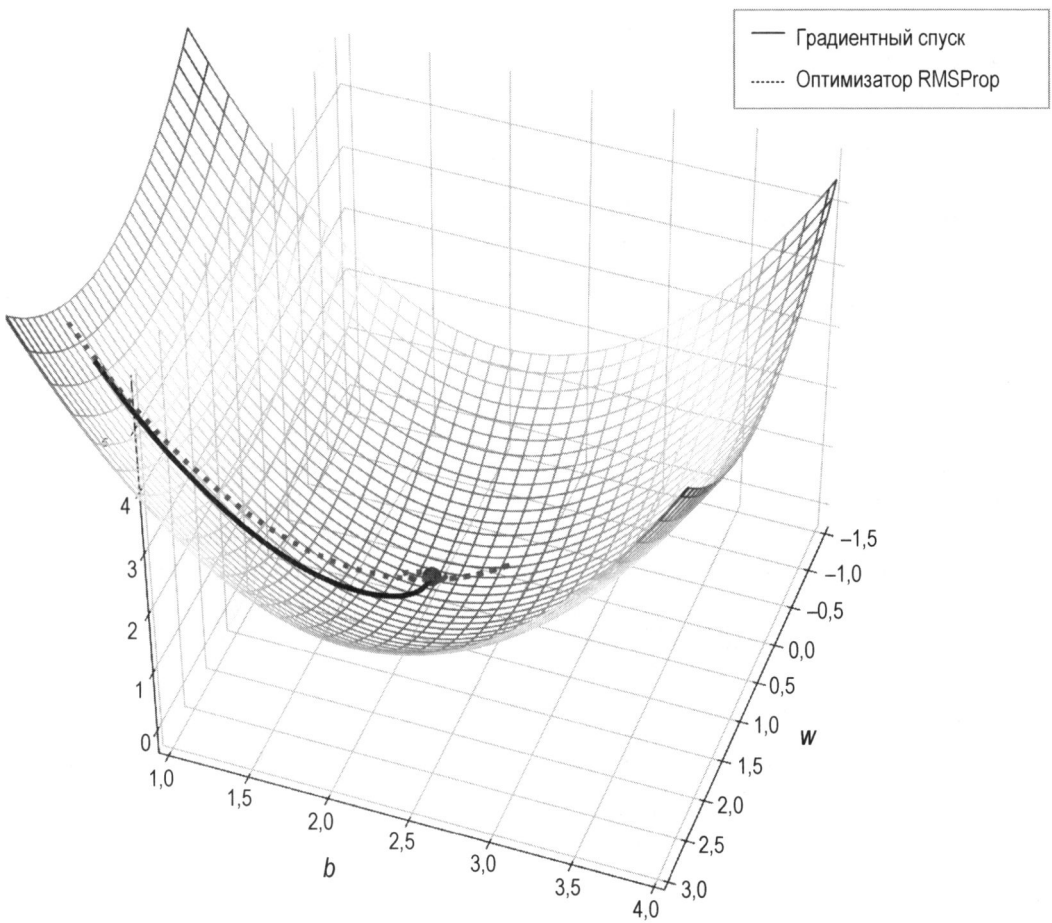


РИС. 4.26. Траектория, выбираемая градиентным спуском ($\gamma = 0{,}05$) и RMSProp ($\gamma = 0{,}05$, $\eta = 0{,}9$, $\varepsilon = 10^{-10}$) по поверхности стоимостной функции. Красная точка обозначает минимум. RMSProp, в особенности в самом начале, выбирает более прямую траекторию к минимуму, чем градиентный спуск

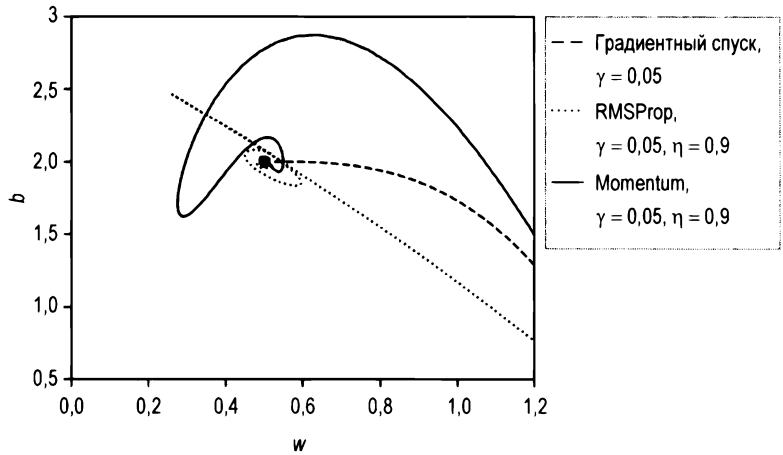


РИС. 4.27. Траектория к минимуму, выбираемая градиентным спуском, оптимизатором RMSProp и оптимизатором Momentum. Видно, как траектория RMSProp к минимуму гораздо прямее. Она очень быстро обходит вокруг него, а затем осциллирует все ближе и ближе

На рис. 4.27 показаны траектории градиентного спуска, оптимизатора RMSProp и оптимизатора Momentum. Вы видите, что траектория RMSProp намного прямее к минимуму. Она приближается к нему очень быстро, а затем осциллирует все ближе и ближе. Она немного перескакивает в самом начале, но затем быстро исправляется и возвращается.

Adam

Последний алгоритм, который мы рассмотрим, называется Adam (adaptive moment estimation, адаптивное оценивание импульса). Он объединяет идеи алгоритмов RMSProp и Momentum в одном оптимизаторе. Как и Momentum, он использует экспоненциально взвешенные средние прошлых производных, и как RMSProp, он использует экспоненциально взвешенные средние прошлых квадратичных производных.

Вам нужно будет рассчитать те же величины, что и для Momentum и RMSProp, а затем необходимо рассчитать следующие величины:

$$\mathbf{v}_{\mathbf{w},[n]}^{\text{скорректированная}} = \frac{\mathbf{v}_{\mathbf{w},[n]}}{1-\beta_1^n};$$

$$v_{b,[n]}^{\text{скорректированная}} = \frac{v_{b,[n]}}{1-\beta_1^n}.$$

Схожим образом необходимо рассчитать

$$\mathbf{S}_{\mathbf{w},[n]}^{\text{скорректированная}} = \frac{\mathbf{S}_{\mathbf{w},[n]}}{1-\beta_2^n};$$

$$S_{b,[n]}^{\text{скорректированная}} = \frac{S_{b,[n]}}{1-\beta_2^n}.$$

Там, где для гиперпараметра мы использовали β_1, мы будем применять его в Momentum, а там, где мы использовали β_2, будем применять в RMSProp. Затем, точно так же как и в RMSProp, мы обновим веса с помощью уравнений

$$\begin{cases} \mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma \dfrac{\mathbf{v}_{\mathbf{w},[n]}^{\text{скорректированная}}}{\sqrt{\mathbf{S}_{\mathbf{w},[n+1]}^{\text{скорректированная}}} + \varepsilon}; \\ b_{[n+1]} = b_{[n]} - \gamma \dfrac{v_{b,[n]}^{\text{скорректированная}}}{\sqrt{S_{b,[n]}^{\text{скорректированная}}} + \varepsilon}. \end{cases}$$

Если применить следующую строку кода, то TensorFlow сделает все за нас:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9, beta2=0.999,
                                   epsilon=1e-8).minimize(cost)
```

где в данном случае были выбраны типичные значения параметров: $\gamma = 0{,}3$, $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ и $\varepsilon = 10^{-8}$. Обратите внимание, что, поскольку этот алгоритм адаптирует темп заучивания к ситуации, для ускорения схождения можно начинать с большего темпа заучивания.

На рис. 4.28 показана траектория вокруг минимума, выбираемого градиентным спуском и оптимизатором Adam. Оптимизатор Adam тоже осциллирует вокруг минимума, но достигает его без проблем. На правом графике (увеличенный участок вокруг минимума) вы видите, как данный алгоритм приближается к минимуму. Для

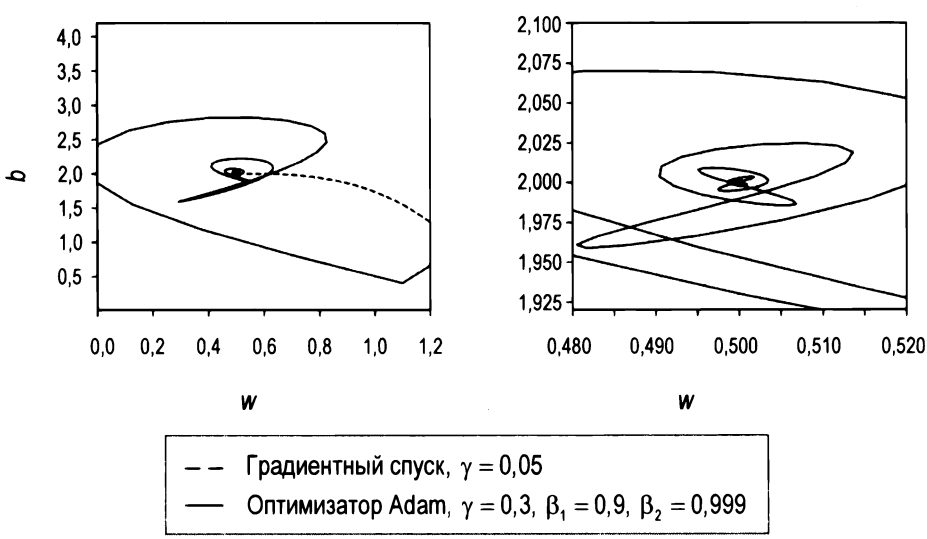


РИС. 4.28. Траектории, выбираемые оптимизаторами на основе градиентного спуска и Adam после 200 эпох. Обратите внимание на число петель, которые требуются оптимизатору Adam вокруг минимума. Несмотря на это, по сравнению с простым градиентным спуском данный оптимизатор реально эффективен

того чтобы иметь представление о качестве данного оптимизатора, веса и смещение доходят до 0,499983; 2,000047 всего после 200 эпох, а это находится по-настоящему близко к минимуму (напомним, что минимум равен $w = 0,5$ и $b = 2,0$).

Здесь показаны не все оптимизаторы, т. к. вы увидели бы много петель, которые ничему бы не научили, а скорее запутали.

Какой оптимизатор следует использовать?

Если коротко, то следует использовать Adam. Он в целом считается быстрее и лучше, чем другие методы. Но это не означает, что так происходит всегда. Недавние исследования показывают, что эти оптимизаторы могут плохо обобщать на новых наборах данных (см., например, Ashia C. Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, Benjamin Recht. The Marginal Value of Adaptive Gradient Methods in Machine Learning[5]. URL: **https://goo.gl/Nzc8bQ**). Существуют и другие работы, которые одобряют использование градиентного спуска с динамическим снижением темпа заучивания. В основном все зависит от вашей задачи. Но, как правило, Adam представляет собой очень хорошую отправную точку.

ПРИМЕЧАНИЕ. Если вы не уверены, с какого оптимизатора начать, используйте Adam. В целом он считается быстрее и лучше, чем другие методы.

Для того чтобы дать вам представление о том, насколько хорошим может быть Adam, давайте применим его к набору данных Zalando. Мы будем использовать сеть с четырьмя скрытыми слоями по 20 нейронов каждый. Модель, которую мы будем рассматривать, обсуждается в самом конце *главы 3*. На рис. 4.29 показано, как стоимостная функция сходится быстрее при использовании оптимизации Adam по сравнению с градиентным спуском. Кроме того, в 100 эпохах градиентный спуск достигает точности 86%, в то время как Adam достигает 90%. Обратите внимание, что в модели ничего не менялось, только оптимизатор! Для Adam использовался следующий фрагмент кода:

```
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9, beta2=0.999,
                                   epsilon=1e-8).minimize(cost)
```

Как и было рекомендовано ранее, при тестировании сложных сетей на больших наборах данных оптимизатор Adam является хорошей отправной точкой. Но вы не должны ограничивать свои тесты только этим оптимизатором. Всегда стоит тестировать другие методы. Возможно, другой подход сработает лучше.

[5] Ашиа С. Уилсон, Ребекка Рулофс, Митчел Стерн, Натан Сребро и Бенджамин Рехт. Предельное значение адаптивных градиентных методов в машинном обучении. — *Прим. пер.*

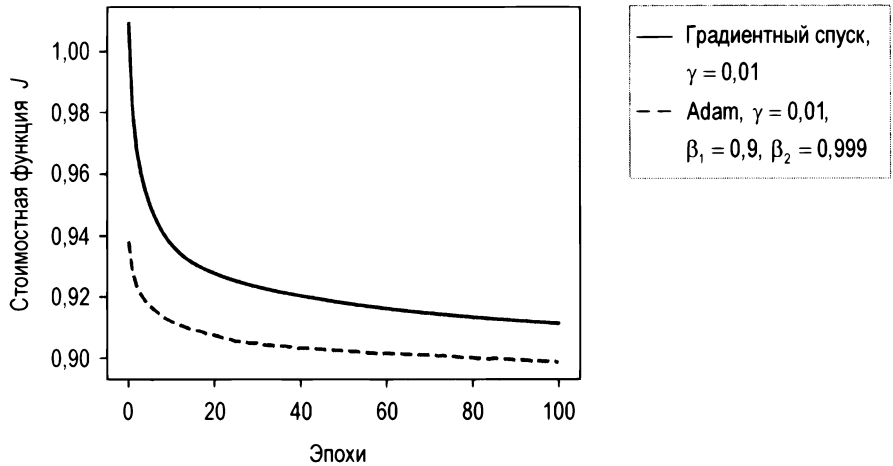


РИС. 4.29. Стоимостная функция набора данных Zalando для сети с 4 скрытыми слоями по 20 нейронов каждый. Непрерывная линия представляет обычный градиентный спуск с темпом заучивания $\gamma = 0{,}01$, и пунктирная линия — оптимизация Adam с $\gamma = 0{,}1$, $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ и $\varepsilon = 10^{-8}$

Пример оптимизатора собственной разработки

Прежде чем завершить эту главу, продемонстрируем разработку собственного оптимизатора с использованием TensorFlow. Это бывает очень полезным, когда вы хотите использовать оптимизатор, который напрямую не доступен. Возьмем, к примеру, исследовательскую работу А. Нилакантана и соавт.[6] В своем исследовании они показывают, как добавление случайного шума к градиентам во время тренировки сложных сетей позволяет сделать простой градиентный спуск очень эффективным. Авторы продемонстрируют, как 20-слойная глубокая сеть может быть эффективно натренирована с помощью стандартного градиентного спуска, даже если начинать с плохой инициализации весов.

Если вы, к примеру, захотите протестировать этот метод, то не сможете использовать функцию `tf.GradientDescentOptimizer`, потому что она реализует простой градиентный спуск без шума, описанного в статье. Для того чтобы его протестировать, вы должны обращаться к градиентам в исходном коде, добавлять в них шум, а затем обновлять веса с помощью измененных градиентов. Мы не будем здесь тестировать их метод; это потребует слишком много времени и выйдет за рамки этой книги, но поучительно будет посмотреть на то, как можно разработать простой

[6] Neelakantan A et al. Adding Gradient Noise Improves Learning for Very Deep Learning (Нилакантан А. Добавление градиентного шума улучшает заучивание в случае очень глубокого обучения). Доклад конференции, представленный на ICLR 2016. **https://arxiv.org/abs/1511.06807**.

градиентный спуск без использования `tf.GradientDescentOptimizer` и без расчета производных вручную.

Перед тем как сконструировать нашу сеть, мы должны знать набор данных, который мы хотим использовать, и представлять, какие задачи (регрессия, классификация и др.) мы хотим решать. Давайте сделаем нечто новое с известным набором данных. Возьмем набор данных MNIST, который мы использовали в *главе 2*, но на этот раз выполним мультиклассовую классификацию с помощью функции `softmax`, как мы сделали на наборе данных Zalando в *главе 3*. В *главе 2* подробно рассматривался вопрос загрузки набора данных MNIST с помощью библиотеки Scikit-learn, поэтому давайте сделаем это по-другому (и эффективнее). В TensorFlow есть способ скачивания набора данных MNIST, в том числе с метками, уже кодированными с одним активным состоянием. Это можно сделать просто с помощью следующих строк:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

В результате получим

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
```

Вы найдете файлы в папке c:\tmp\data (если вы используете Windows). Если вы хотите изменить место расположения файлов, то должны изменить параметр `"/tmp/data"` функции `read_data_sets`. Вероятно, вы помните из *главы 2*, что изображения MNIST имеют размер 28×28 пикселов (всего 784 пикселов), представлены в оттенках серого, поэтому каждый пиксел может принимать значение от 0 до 254. Имея эту информацию, мы можем построить нашу сеть.

```
X = tf.placeholder(tf.float32, [784, None]) # данные mnist имеют форму
28*28=784
Y = tf.placeholder(tf.float32, [10, None]) # распознавание цифр 0-9 => 10
классов
learning_rate_ = tf.placeholder(tf.float32, shape=())
W = tf.Variable(tf.zeros([10, 784]), dtype=tf.float32)
b = tf.Variable(tf.zeros([10,1]), dtype=tf.float32)

y_ = tf.nn.softmax(tf.matmul(W,X)+b)
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))

grad_W, grad_b = tf.gradients(xs=[W, b], ys=cost)
```

```
new_W = W.assign(W - learning_rate_ * grad_W)
new_b = b.assign(b - learning_rate_ * grad_b)
```

Строка

```
grad_W, grad_b = tf.gradients(xs=[W, b], ys=cost)
```

Приведенный выше фрагмент кода дает вам тензоры, которые содержат градиенты узла `cost` соответственно по `W` и `b`. TensorFlow вычисляет их за вас автоматически! Если вам интересно узнать, как это делается, обратитесь к официальной документации по функции `tf.gradients` (**https://goo.gl/XAjRkX**). Теперь необходимо добавить узлы в вычислительный граф, который обновляет веса:

```
new_W = W.assign(W - learning_rate_ * grad_W)
new_b = b.assign(b - learning_rate_ * grad_b)
```

Когда во время сеанса мы просим TensorFlow вычислить узлы `new_W` и `new_b`, веса и смещение обновляются. Наконец, мы должны модифицировать функцию, которая вычисляет граф, используя (для мини-пакетного градиентного спуска) строку кода

```
_, _, cost_ = sess.run([new_W, new_b , cost], feed_dict = {X:X_train_mini,
                                                           Y:y_train_mini,
                                                 learning_rate_:learning_r})
```

Благодаря этому вычисляются новые узлы `new_W` и `new_b`, и TensorFlow обновляет веса и смещение. Следующая строка больше не нужна:

```
sess.run(optimizer, feed_dict = {X:X_train_mini, Y:y_train_mini,
                                 learning_rate_:learning_r})
```

потому что у нас больше нет узла `optimizer`. Ниже приведена вся функция целиком:

```
def run_model_mb(minibatch_size, training_epochs, features, classes,
                 logging_step=100, learning_r=0.001):
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    total_batch = int(mnist.train.num_examples/minibatch_size)

    cost_history = []
    accuracy_history = []
    for epoch in range(training_epochs+1):
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(minibatch_size)
            batch_xs_t = batch_xs.T
            batch_ys_t = batch_ys.T

        _, _, cost_ = sess.run([new_W, new_b, cost],
                               feed_dict={X:batch_xs_t,
                                          Y:batch_ys_t,
                                          learning_rate_:learning_r})
```

```
        cost_ = sess.run(cost, feed_dict={ X:features, Y: classes})
        accuracy_ = sess.run(accuracy, feed_dict={ X:features,
                                                   Y: classes})
        cost_history = np.append(cost_history, cost_)
        accuracy_history = np.append(accuracy_history, accuracy_)

        if (epoch % logging_step == 0):
            print("Достигнута эпоха",epoch,"стоимость J =", cost_)
            print ("Точность:", accuracy_)

    return sess, cost_history, accuracy_history
```

Эта функция немного отличается от тех, которые мы использовали раньше, потому что здесь применено несколько функциональных средств TensorFlow, которые во многом делают нашу жизнь легче. В частности, строка

```
total_batch = int(mnist.train.num_examples/minibatch_size)
```

вычисляет суммарное число имеющихся мини-пакетов, т. к. переменная `mnist.train.num_examples` содержит число наблюдений, которые мы имеем в нашем распоряжении. После этого для получения пакетов мы используем

```
batch_xs, batch_ys = mnist.train.next_batch(minibatch_size)
```

В результате получаем два тензора с тренировочными входными данными (`batch_xs`) и метками, кодированными с одним активным состоянием (`batch_ys`). Затем мы должны их транспонировать, потому что TensorFlow возвращает массив с наблюдениями в виде строк. Мы делаем это с помощью следующих инструкций:

```
batch_xs_t = batch_xs.T
batch_ys_t = batch_ys.T
```

Кроме того, в функцию был добавлен расчет точности для того, чтобы было легче видеть результаты работы. Выполнив модель с помощью Python-вызова

```
sess, cost_history, accuracy_history = run_model(100, 50, X_train_tr,
                            labels_, logging_step=10, learning_r=0.01)
```

мы получим следующий результат:

```
Достигнута эпоха 0 стоимость J = 1.06549
Точность: 0.773786
Достигнута эпоха 10 стоимость J = 0.972171
Точность: 0.853371
Достигнута эпоха 20 стоимость J = 0.961519
Точность: 0.869357
Достигнута эпоха 30 стоимость J = 0.956766
Точность: 0.877814
Достигнута эпоха 40 стоимость J = 0.953982
Точность: 0.883143
Достигнута эпоха 50 стоимость J = 0.952118
Точность: 0.886386
```

Эта модель будет работать точно так же, как с оптимизатором градиентного спуска, предоставляемым библиотекой TensorFlow. Но теперь у вас есть доступ к градиентам, и вы можете изменять их, добавлять в них шум (если хотите попробовать) и т. д. На рис. 4.30 представлено поведение стоимостной функции (*а*) и точность в сопоставлении с эпохами (*б*), которые мы получаем этой моделью.

ПРИМЕЧАНИЕ. TensorFlow — это отличная библиотека, поскольку она обеспечивает гибкость в создании моделей, в сущности, с нуля. Однако важно понимать особенности работы разных методов и уметь в полной мере использовать библиотеку. Вы должны обладать очень хорошим пониманием математического каркаса, лежащего в основе алгоритмов, для того, чтобы иметь возможность их настраивать или разрабатывать свои варианты.

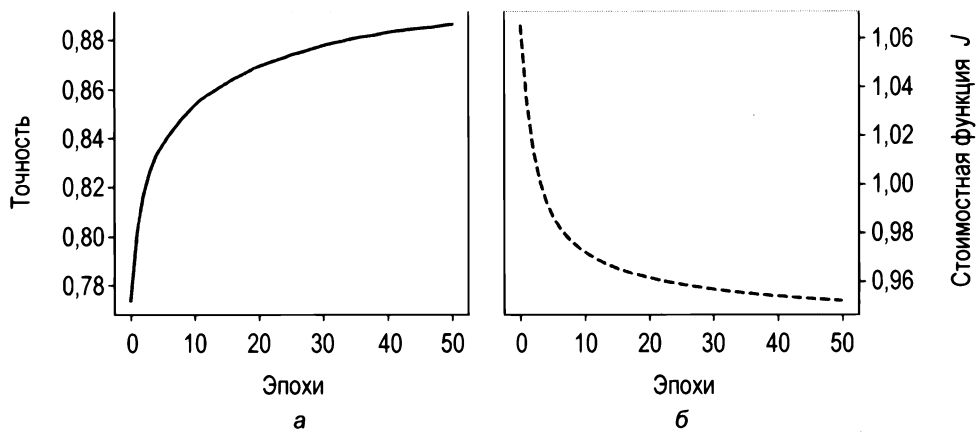


РИС. 4.30. Поведение стоимостной функции (*а*) и точность в сопоставлении с эпохами (*б*) для нейронной сети с одним нейроном и с градиентным спуском, разработанным с помощью функции `tf.gradients`

ГЛАВА 5

Регуляризация

В этой главе мы рассмотрим очень важное техническое решение, часто используемое во время тренировки глубоких сетей: регуляризацию. Вы рассмотрите такие технические решения, как регуляризация l_2 и l_1, отсев и досрочная остановка. Вы увидите, как эти методы при правильном применении помогают избежать проблемы переподгонки и достичь гораздо более высоких результатов от ваших моделей. Мы рассмотрим математический каркас, лежащий в основе этих решений, и их правильную реализацию на Python и в TensorFlow.

Сложные сети и переподгонка

В предыдущих главах вы научились строить и тренировать сложные сети. Одна из наиболее распространенных проблем, с которой вы столкнетесь при использовании сложных сетей, — это переподгонка. Просмотрите *главу 3*, где дается обзор понятия переподгонки. В этой главе вы столкнетесь с крайним случаем переподгонки, и здесь будет продемонстрировано несколько стратегий, которые позволяют ее избежать.

Идеальным набором данных для изучения этой проблемы является бостонский набор данных с ценами на жилую недвижимость, обсуждавшийся в *главе 2*. Давайте еще раз посмотрим, как получать эти данные (более подробное обсуждение *см. в главе 2*). Начнем с необходимых нам программных пакетов.

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

import tensorflow as tf
import numpy as np

from sklearn.datasets import load_boston
import sklearn.linear_model as sk
```

Затем импортируем набор данных.

```
boston = load_boston()
features = np.array(boston.data)
target = np.array(boston.target)
```

Набор данных содержит 13 признаков (в массиве NumPy `features`) и цены домов (в массиве NumPy `target`). Как и в *главе 2*, для нормализации признаков мы будем использовать функцию

```
def normalize(dataset):
    mu = np.mean(dataset, axis = 0)
    sigma = np.std(dataset, axis = 0)
    return (dataset-mu)/sigma
```

В завершение подготовки набора данных давайте его нормализуем, а затем создадим тренировочный (train) и рабочий (dev) наборы данных.

```
features_norm = normalize(features)
np.random.seed(42)
rnd = np.random.rand(len(features_norm)) < 0.8

train_x = np.transpose(features_norm[rnd])
train_y = np.transpose(target[rnd])
dev_x = np.transpose(features_norm[~rnd])
dev_y = np.transpose(target[~rnd])
```

Функция `np.random.seed(42)` необходима для того, чтобы вы всегда получали одни и те же тренировочный и рабочий наборы (благодаря чему ваши результаты будут воспроизводимыми). Теперь давайте реформируем нужные нам массивы.

```
train_y = train_y.reshape(1,len(train_y))
dev_y = dev_y.reshape(1,len(dev_y))
```

И затем построим сложную нейронную сеть с 4 слоями и 20 нейронами в каждом слое. Определим следующую функцию для построения каждого слоя:

```
def create_layer (X, n, activation):
    ndim = int(X.shape[0])
    stddev = 2.0 / np.sqrt(ndim)
    initialization = tf.truncated_normal((n, ndim), stddev = stddev)
    W = tf.Variable(initialization)
    b = tf.Variable(tf.zeros([n,1]))
    Z = tf.matmul(W,X)+b
    return activation(Z), W, b
```

Обратите внимание, что на этот раз мы возвращаем весовой тензор `W` и смещение `b`. Они нам понадобятся при осуществлении регуляризации. Вы уже видели эту функцию в конце *главы 3*, поэтому должны понимать, что она делает. Здесь мы используем инициализацию Xe, потому что будем применять активационные функции ReLU. Сеть создается с помощью следующего фрагмента кода:

```
tf.reset_default_graph()

n_dim = 13
n1 = 20
n2 = 20
n3 = 20
n4 = 20
n_outputs = 1

tf.set_random_seed(5)

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])

learning_rate = tf.placeholder(tf.float32, shape=())

hidden1, W1, b1 = create_layer (X, n1, activation = tf.nn.relu)
hidden2, W2, b2 = create_layer (hidden1, n2, activation = tf.nn.relu)
hidden3, W3, b3 = create_layer (hidden2, n3, activation = tf.nn.relu)
hidden4, W4, b4 = create_layer (hidden3, n4, activation = tf.nn.relu)
y_, W5, b5 = create_layer (hidden4, n_outputs, activation = tf.identity)

cost = tf.reduce_mean(tf.square(y_-Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9, beta2=0.999,
                                   epsilon=1e-8).minimize(cost)
```

В выходном слое находится один нейрон, в котором для регрессии используется активационная функция тождественного отображения. Кроме того, мы используем оптимизатор Adam, как предложено в *главе 4*. Теперь давайте выполним эту модель с помощью следующего ниже фрагмента кода:

```
sess = tf.Session()
sess.run(tf.global_variables_initializer())

cost_train_history = []
cost_dev_history = []

for epoch in range(10000+1):
    sess.run(optimizer, feed_dict = {X:train_x, Y:train_y,
                                     learning_rate:0.001})
    cost_train_ = sess.run(cost, feed_dict={X:train_x, Y:train_y,
                                     learning_rate:0.001})
    cost_dev_ = sess.run(cost, feed_dict={X:dev_x, Y:dev_y,
                                     learning_rate:0.001})
    cost_train_history = np.append(cost_train_history, cost_train_)
    cost_dev_history = np.append(cost_test_history, cost_test_)
```

```
    if (epoch % 1000 == 0):
        print("Достигнута эпоха",epoch,"стоимость J(train) =", cost_train_)
        print("Достигнута эпоха",epoch,"стоимость J(test) =", cost_test_)
```

Как вы могли заметить, тут имеется несколько отличий от того, что мы делали раньше. Для упрощения задачи здесь не используется отдельная функция, и все значения жестко запрограммированы в исходном коде, потому что в этом случае нам не нужно особо настраивать параметры. Здесь не используются мини-пакеты, потому что у нас всего несколько сотен наблюдений, и MSE (среднеквадратическая ошибка) вычисляется как для тренировочного, так и для рабочего наборов данных с помощью следующих ниже строк:

```
cost_train_ = sess.run(cost, feed_dict={X:train_x, Y:train_y,
learning_rate:0.001})
cost_dev_ = sess.run(cost, feed_dict={X:dev_x, Y:dev_y, learning_rate:0.001})
```

Благодаря этому мы можем одновременно проверять, что происходит в обоих наборах данных. Теперь, если выполнить этот исходный код и построить график двух MSE, один для тренировочного набора данных, который будет обозначен как $MSE_{тренировка}$, а другой для рабочего набора, обозначенного как $MSE_{разработка}$, мы получим рис. 5.1.

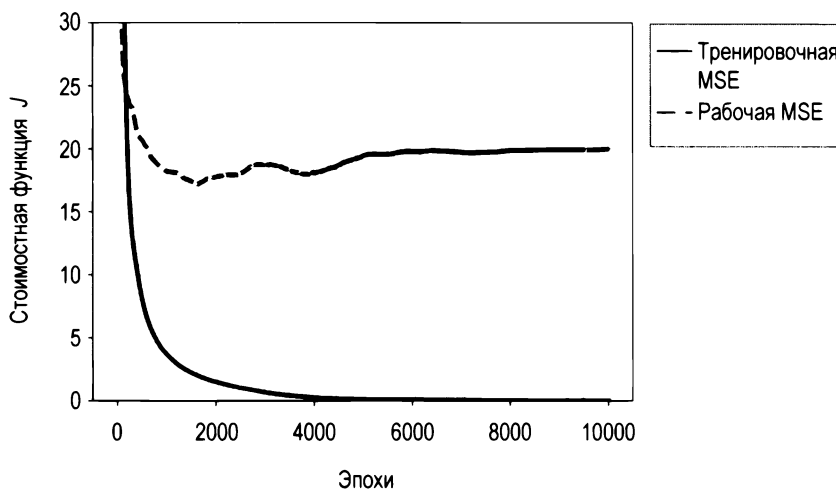


РИС. 5.1. MSE для тренировочного (непрерывная линия) и рабочего (пунктирная линия) наборов данных для нейронной сети с 4 слоями по 20 нейронов каждый

Вы заметите, что тренировочная ошибка опускается до нуля, в то время как рабочая ошибка остается постоянной примерно на уровне 20 после быстрого падения в самом начале. Как вы помните из введения в простой анализ ошибок, это означает, что мы находимся в режиме экстремальной переподгонки (при $MSE_{тренировка} \ll MSE_{разработка}$). Ошибка на тренировочном наборе данных практически равна нулю, в то время как на рабочем наборе данных — нет. Во время применения модели к новым данным она вообще не способна обобщать. На рис. 5.2 показано

предсказанное значение в сопоставлении с реальным значением. Вы заметите, что в графике слева предсказание на тренировочных данных является почти идеальным, в то время как график справа для рабочего набора данных не так хорош. Идеальная модель должна давать предсказанные значения, в точности равные измеренным. Поэтому при построении графика сравнения одного значения с другим все они будут лежать на 45-градусной линии графика, как на рис. 5.2, *б*.

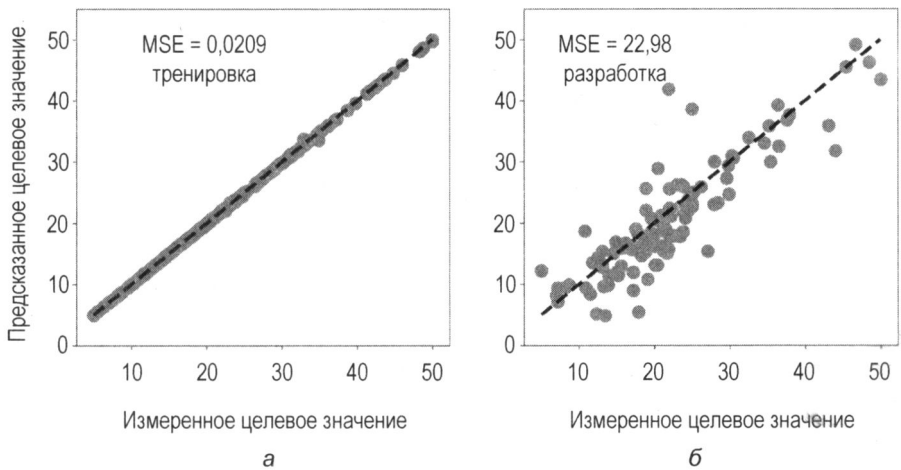


РИС. 5.2. Предсказанное значение в сопоставлении с реальным значением целевой переменной (цена дома). Вы заметите, что на графике (*а*) предсказание на тренировочных данных является почти идеальным, в то время как на графике (*б*) предсказания на рабочем наборе данных разбросаны больше

Что можно сделать в этом случае для того, чтобы избежать проблемы переподгонки? Одним из решений, конечно, было бы уменьшение сложности сети, т. е. уменьшение числа слоев и/или числа нейронов в каждом слое. Но, как вы понимаете, эта стратегия является очень трудоемкой. Вам придется испытывать несколько сетевых архитектур и смотреть на поведение ошибки на тренировочном и рабочем наборах данных. В этом случае это решение будет по-прежнему жизнеспособным, но если вы работаете над задачей, для которой тренировочная фаза занимает несколько дней, то оно может оказаться довольно сложным и чрезвычайно трудоемким. Для решения этой проблемы было разработано несколько стратегий. Наиболее распространенная из них называется регуляризацией и находится в центре внимания этой главы.

Что такое регуляризация?

Прежде чем перейти к рассмотрению разных методов, необходимо вкратце обсудить сам термин "регуляризация", что именно под ним понимается в сообществе глубокого обучения. С течением времени данный термин "глубоко" (каламбур) эволюционировал. Например, в традиционном смысле (с 1990-х годов) этот термин

относился только к штрафному члену в функции потери (Christopher M. Bishop, Neural Networks for Pattern Recognition. — New York: Oxford University Press, 1995[1]). В последнее время этот термин приобрел гораздо более широкий смысл. Например, Иэн Гудфеллоу и соавт. (Ian Goodfellow et al. Deep Learning. — Cambridge, MA, MIT Press, 2016[2]) определяют его как "любую модификацию, вносимую нами в самообучающийся алгоритм, которая предназначена для сокращения ошибки на тестовом наборе, но не ошибки на тренировочном наборе". Исследование Яна Кукачки и соавт. (Jan Kukacka et al. Regularization for deep learning: a taxonomy[3], arXiv:1710.10686v1, доступное на **https://goo.gl/wNkjXz**) обобщает этот термин еще больше и предлагает следующее определение: "регуляризация — это любое дополнительное техническое решение, ориентированное на то, чтобы модель обобщала лучше, т. е. давала более высокие результаты на тестовом наборе". Поэтому будьте внимательны при использовании этого термина и всегда будьте точны в том, что вы имеете в виду.

Возможно, вы также слышали или встречали утверждение о том, что регуляризация была разработана для борьбы с переподгонкой. Это тоже является одним из способов ее понимания. Напомним, что модель, которая слишком плотно прилегает к тренировочному набору данных, плохо обобщает на новых данных. Это определение также можно найти в Интернете наряду со всеми другими. Хотя это всего лишь определения, важно о них знать, чтобы лучше понимать то, что имеется в виду, читая исследовательские работы или книги. Это очень активная область исследований, и, чтобы дать вам представление, Кукачка и соавт. в своем упомянутом выше обзоре, приводят список из 58 разных регуляризационных методов. Да-да, 58; это не опечатка. Но важно понимать, что в их общем определении данного понятия стохастический градиентный спуск (SGD) также считается методом регуляризации, с чем не все согласны. Поэтому необходимо это учитывать, читая исследовательский материал, и проверять, что понимается под термином "регуляризация".

В этой главе вы рассмотрите три наиболее распространенных и известных метода: l_1, l_2 и отсев, а также, в кратком изложении, раннюю остановку, хотя этот метод технически не борется с переподгонкой. Методы l_1 и l_2 достигают так называемого затухания весов путем добавления так называемого регуляризационного члена в стоимостную функцию, в то время как отсев просто случайно удаляет узлы из сети в тренировочной фазе. Для того чтобы надлежаще понять эти три метода, мы должны изучить их в деталях. Начнем, пожалуй, с самого показательного: регуляризации l_2.

В конце главы мы рассмотрим несколько других идей о том, как бороться с переподгонкой и получить модель, способную обобщать лучше. Вместо изменения или модификации модели или самообучающегося алгоритма, мы рассмотрим стратегии

---

[1] Кристофер М. Бишоп. Нейронные сети для распознавания образов. — *Прим. пер.*

[2] Гудфеллоу И. Глубокое обучение. — *Прим. пер.*

[3] Регуляризация для глубокого обучения: таксономия. — *Прим. пер.*

с идеей модификации тренировочных данных с целью сделать заучивание эффективнее.

О сетевой сложности

Потратим несколько минут на то, чтобы кратко рассмотреть термин, который уже использовался неоднократно: сетевая сложность. Вы уже встречали здесь, как, впрочем, и в других текстах, что с помощью регуляризации можно сократить сетевую сложность. Но что это на самом деле значит? Дать определение сетевой сложности довольно трудно настолько, что никто этого не делает. Можно найти несколько научных работ по проблеме модельной сложности (обратите внимание, что речь не идет о сетевой сложности), коренящейся в теории информации. В этой главе вы увидите, как, например, число отличающихся от нуля весов будет драматически изменяться вместе с числом эпох, вместе с оптимизационным алгоритмом и т. д., что, как следствие, ставит это смутное интуитивное понятие сложности в зависимость и от того, как долго вы тренируете свою модель. Короче говоря, термин "сетевая сложность" следует использовать только на интуитивном уровне, поскольку теоретически это понятие является очень сложным. Полное обсуждение этого вопроса выходит далеко за рамки данной книги.

Норма l_p

Прежде чем мы начнем изучать регуляризацию l_1 и l_2, необходимо ввести обозначения нормы l_p. Определим норму l_p вектора $\mathbf{x}$ с x_i компонентами как

$$\left\|\mathbf{x}_p\right\| = \sqrt[p]{\sum_i \left|x_i\right|^p}, \quad p \in \mathbb{R},$$

где суммирование выполняется по всем компонентам вектора $\mathbf{x}$.

Начнем с самой показательной нормы: l_2.

Регуляризация l_2

Один из наиболее распространенных методов регуляризации, регуляризация l_2, состоит в добавлении члена в стоимостную функцию с целью эффективного сокращения способности сети адаптироваться к сложным наборам данных. Давайте сначала рассмотрим математический механизм, лежащий в основе данного метода.

Теоретическое обеспечение регуляризации l_2

Во время выполнения простой регрессии, как вы помните из *главы 2*, нашей стоимостной функцией была простая среднеквадратическая ошибка (MSE):

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(y_i - \hat{y}_i\right)^2,$$

где y_i — измеренная целевая переменная; $\hat{y}_i$ — предсказанное значение; $\mathbf{w}$ — вектор всех весов сети, включая смещение; m — число наблюдений.

Теперь определим новую стоимостную функцию $\tilde{J}(\mathbf{w}, b)$:

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \frac{\lambda}{2m}\|\mathbf{w}\|_2^2.$$

Дополнительный член

$$\frac{\lambda}{2m}\|\mathbf{w}\|_2^2$$

называется регуляризационным членом и является ничем иным, как квадратом нормы l_2 вектора $\mathbf{w}$, умноженным на постоянный множитель $\lambda/(2m)$. Лямбда λ называется регуляризационным параметром.

ПРИМЕЧАНИЕ. Новый регуляризационный параметр λ представляет собой новый гиперпараметр, который вы должны отрегулировать для отыскания оптимального значения.

Теперь попробуем получить интуитивное представление о том, как этот термин влияет на алгоритм градиентного спуска. Рассмотрим обновленное уравнение для веса w_j:

$$w_{j,[n+1]} = w_{j,[n]} - \gamma\frac{\partial\tilde{J}\left(\mathbf{w}_{[n]}\right)}{\partial w_j} = w_{j,[n]} - \gamma\frac{\partial J\left(\mathbf{w}_{[n]}\right)}{\partial w_j} - \frac{\gamma\lambda}{m}w_{j,[n]}.$$

Поскольку

$$\frac{\partial}{\partial w_j}\|\mathbf{w}\|_2^2 = 2w_j,$$

в результате получаем:

$$w_{j,[n+1]} = w_{j,[n]}\left(1 - \frac{\gamma\lambda}{m}\right) - \lambda\frac{\partial J\left(\mathbf{w}_{[n]}\right)}{\partial w_j}.$$

Именно это уравнение мы должны использовать для обновления весов. Разница с уравнением, которое мы уже знаем из обычного градиентного спуска, состоит в том, что теперь вес $w_{j,[n]}$ умножается на константу $1 - \gamma\lambda/m < 1$, и, следовательно, это имеет эффект фактического сдвига весовых значений во время обновления к нулю, что (в интуитивном плане) делает сеть менее сложной, т. е. происходит борьба с переподгонкой. Выясним, что на самом деле происходит с весами, применив данный метод к бостонскому набору данных с ценами на жилую недвижимость.

Реализация в TensorFlow

Реализация в TensorFlow довольно простая. Напомним: мы должны вычислить дополнительный член $\|\mathbf{w}\|_2^2$, а затем добавить его в стоимостную функцию. Конструирование модели практически не изменилось. Это можно сделать с помощью следующего фрагмента кода:

```
tf.reset_default_graph()

n_dim = 13
n1 = 20
n2 = 20
n3 = 20
n4 = 20
n_outputs = 1

tf.set_random_seed(5)

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])

learning_rate = tf.placeholder(tf.float32, shape=())

hidden1, W1, b1 = create_layer (X, n1, activation = tf.nn.relu)
hidden2, W2, b2 = create_layer (hidden1, n2, activation = tf.nn.relu)
hidden3, W3, b3 = create_layer (hidden2, n3, activation = tf.nn.relu)
hidden4, W4, b4 = create_layer (hidden3, n4, activation = tf.nn.relu)
y_, W5, b5 = create_layer (hidden4, n_outputs, activation = tf.identity)

lambd = tf.placeholder(tf.float32, shape=())
reg = tf.nn.l2_loss(W1) + tf.nn.l2_loss(W2) + tf.nn.l2_loss(W3) + \
      tf.nn.l2_loss(W4) + tf.nn.l2_loss(W5)

cost_mse = tf.reduce_mean(tf.square(y_-Y))
cost = tf.reduce_mean(cost_mse + lambd*reg)
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate, beta1=0.9,
                                   beta2=0.999, epsilon=1e-8).minimize(cost)
```

Мы создаем заполнитель для нового регуляризационного параметра λ.

```
lambd = tf.placeholder(tf.float32, shape=())
```

Напомним, что в Python слово `lambda` зарезервировано, т. е. его нельзя применять, и поэтому мы используем `lambd`. Затем мы вычисляем регуляризационный член $\|\mathbf{w}\|_2^2$

```
reg = tf.nn.l2_loss(W1) + tf.nn.l2_loss(W2) + tf.nn.l2_loss(W3) + \
      tf.nn.l2_loss(W4) + tf.nn.l2_loss(W5)
```

с полезной функцией TensorFlow `tf.nn.l2_loss()`, а потом добавляем его в функцию MSE `cost_mse`.

```
cost_mse = tf.reduce_mean(tf.square(y_-Y))
cost = tf.reduce_mean(cost_mse + lambd*reg)
```

Теперь тензор `cost` будет содержать MSE плюс регуляризационный член. Затем необходимо просто натренировать сеть и понаблюдать, что происходит. Для тренировки сети мы используем эту функцию:

```
def model(training_epochs, features, target, logging_step=100,
          learning_r=0.001, lambd_val=0.1):
    sess = tf.Session()
    sess.run(tf.global_variables_initializer())

    cost_history = []

    for epoch in range(training_epochs+1):
        sess.run(optimizer, feed_dict = {X:features, Y:target,
                                         learning_rate:learning_r,
                                         lambd:lambd_val})
        cost_ = sess.run(cost_mse, feed_dict={X:features, Y:target,
                         learning_rate:learning_r, lambd:lambd_val})
        cost_history = np.append(cost_history, cost_)

        if (epoch % logging_step == 0):
            pred_y_test = sess.run(y_, feed_dict={X:test_x, Y:test_y})
            print("Достигнута эпоха",epoch,"стоимость J =", cost_)
            print("Тренировочная MSE = ", cost_)
            print("Рабочая MSE = ", sess.run(cost_mse,
                                             feed_dict={X:test_x, Y:test_y}))
    return sess, cost_history
```

На этот раз печатается ошибка MSE, которая получается из тренировочного ($MSE_{тренировка}$) и рабочего ($MSE_{разработка}$) наборов данных, позволяя проверить ход работы. Как уже упоминалось, применение этого метода вынуждает многочисленные веса опускаться до нуля, фактически сокращая сложность сети и, следовательно, борясь с переподгонкой. Давайте выполним эту модель для $\lambda = 0$ без регуляризации и для $\lambda = 10{,}0$. Мы можем выполнить модель с использованием следующего кода:

```
sess, cost_history = model(learning_r=0.01,
                           training_epochs=5000,
                           features=train_x,
                           target=train_y,
                           logging_step=5000,
                           lambd_val=0.0)
```

В результате получим

```
Достигнута эпоха 0 стоимость J = 238.378
Тренировочная MSE = 238.378
Рабочая MSE = 205.561
Достигнута эпоха 5000 стоимость J = 0.00527479
Тренировочная MSE = 0.00527479
Рабочая MSE = 28.401
```

Как и ожидалось, после 5000 эпох мы находимся в режиме экстремальной переподгонки ($MSE_{тренировка} \ll MSE_{разработка}$). Теперь попробуем с $\lambda = 10$.

```
sess, cost_history = model(learning_r=0.01,
                           training_epochs=5000,
                           features=train_x,
                           target=train_y,
                           logging_step=5000,
                           lambd_val=10.0)
```

В результате получим:

```
Достигнута эпоха 0 стоимость J = 248.026
Тренировочная MSE = 248.026
Рабочая MSE = 214.921
Достигнута эпоха 5000 стоимость J = 23.795
Тренировочная MSE = 23.795
Рабочая MSE = 21.6406
```

Теперь мы больше не находимся в режиме переподгонки, потому что два значения MSE имеют одинаковый порядок величины. Лучший способ проверить, что происходит, — изучить распределение весов каждого слоя. На рис. 5.3 показано распределение весов для первых 4 слоев. Светло-серая гистограмма предназначена для весов без регуляризации, а более темный (и гораздо более концентрированный вокруг нуля) участок — для весов с регуляризацией. Слой 5 был опущен, т. к. он является выходным.

Здесь ясно видно, что, когда мы применяем регуляризацию, веса гораздо более сконцентрированы вокруг нуля, т. е. они намного меньше, чем без регуляризации. Это четко показывает эффект регуляризации на затухание. Пользуясь возможностью, сделаем еще одно краткое отступление от сетевой сложности. Как было отмечено, этот метод уменьшает сетевую сложность. В данной главе уже упоминалось, что число заучиваемых параметров можно рассматривать как показатель сетевой сложности, но при этом высказывалось предупреждение, что этот показатель может вводить в заблуждение. Теперь будет показано, почему это бывает. Из *главы 3* вы помните, что суммарное число заучиваемых параметров, имеющихся в сети, подобной той, которую мы используем здесь, определяется формулой:

$$Q = \sum_{j=1}^{L} n_j \left(n_{j-1} + 1 \right),$$

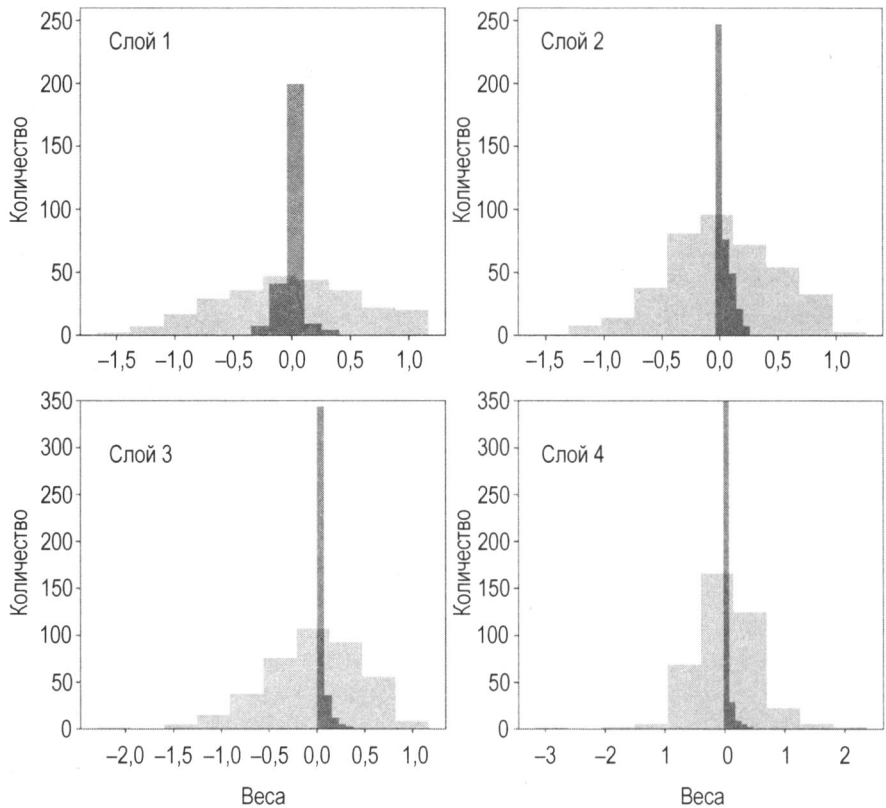


РИС. 5.3. Распределение весов каждого слоя

где n_l — это число нейронов в слое l; L — суммарное число слоев, включая выходной слой. В нашем случае мы имеем входной слой с 13 признаками, далее 4 слоя с 20 нейронами каждый и затем выходной слой с 1 нейроном. Поэтому Q задается выражением

$$Q = 20 \times (13+1) + 20 \times (20+1) + 20 \times (20+1) + 20 \times (20+1) + 1 \times (20+1) = 1561.$$

Величина числа Q — довольно большая. Но уже сейчас, без регуляризации, интересно отметить, что после 10 000 эпох у нас примерно 48% весов меньше 10^{-10}, и поэтому практически равны нулю. Именно по этой причине высказывалось предостережение против того, чтобы говорить о сложности с точки зрения числа заучиваемых параметров. Кроме того, использование регуляризации полностью изменит этот сценарий. Понятие "сложность" определить трудно: оно зависит от многих вещей, в том числе от архитектуры, оптимизационного алгоритма, стоимостной функции и числа эпох во время тренировки.

ПРИМЕЧАНИЕ. Определение понятия сетевой сложности только с точки зрения числа весов является не совсем верным. Суммарное число весов дает некое представление, но оно может ввести в заблуждение, потому что многие из них могут

быть нулевыми после тренировки, практически исчезая из сети и делая ее менее сложной. Правильнее говорить не о сетевой сложности, а о "модельной сложности", поскольку фактически задействуется гораздо больше аспектов, чем просто число нейронов или слоев, которые имеются в сети.

Невероятно, но только половина весов в конечном счете играет роль в предсказаниях. Вот почему в той главе отмечалось, что определение понятия сетевой сложности только параметром Q вводит в заблуждение. С учетом вашей задачи, функции потери и оптимизатора, вы вполне можете получить сеть, которая после ее тренировки будет намного проще, чем в конструкционной фазе. Поэтому в сфере глубокого обучения будьте очень осторожны при использовании термина "сложность". Будьте в курсе всех тонкостей.

Для того чтобы дать вам представление об эффективности данной регуляризации в сокращении весов, взгляните на табл. 5.1, в которой сравнивается процентное соотношение весов меньше 10^{-3} с регуляризацией и без нее после 1000 эпох в каждом слое.

Таблица 5.1. *Процентное соотношение весов менее 10^{-3} с регуляризацией и без нее после 1000 эпох*

Слой	% весов менее 10^{-3} для $\lambda = 0$	% весов менее 10^{-3} для $\lambda = 3$
1	0,0	20,0
2	0,25	41,5
3	0,75	60,5
4	0,25	66,0
5	0,0	35,0

Но как выбрать λ? Для получения представления (повторяйте за мной: в сфере глубокого обучения универсального правила попросту нет) полезно посмотреть, что происходит при варьировании параметра λ в вашей оптимизационной метрике (в данном случае MSE). На рис. 5.4 показано поведение наборов данных $MSE_{тренировка}$ (непрерывная линия) и $MSE_{разработка}$ (пунктирная линия) для нашей сети при варьировании λ после 1000 эпох.

Как вы видите по небольшим значениям λ (практически без регуляризации), мы находимся в режиме переподгонки ($MSE_{тренировка} \ll MSE_{разработка}$): $MSE_{тренировка}$ медленно увеличивается, в то время как $MSE_{разработка}$ остается примерно постоянной. Вплоть до $\lambda \approx 7{,}5$ модель слишком плотно прилегает к тренировочным данным, затем эти два значения пересекаются, и переподгонка заканчивается. После этого они растут вместе, и в этот момент модель больше не может улавливать тонкие структуры данных. После пересечения линий модель становится слишком простой для улавливания признаков задачи, и, следовательно, ошибки растут вместе, при

этом ошибка на тренировочном наборе данных становится больше, потому что модель не аппроксимирует даже тренировочные данные. В этом конкретном случае хорошим значением λ будет около 7,5, почти то значение, когда две линии пересекаются, потому что в той области вы больше не находитесь на участке переподогнанности, т. к. $MSE_{тренировка} \approx MSE_{разработка}$. Напомним, главная цель наличия регуляризационного члена — получить модель, которая обобщает наилучшим образом при ее применении к новым данным. Вы можете посмотреть на это по-другому: значение $\lambda \approx 7,5$ дает вам минимальную $MSE_{разработка}$ за пределами участка переподогнанности (для $\lambda \lesssim 7,5$); поэтому данное значение было бы хорошим вариантом. Обратите внимание, что в своих задачах вы можете наблюдать совсем другое поведение своей оптимизационной метрики, поэтому в каждом конкретном случае вам придется решать, какое значение λ работает для вас лучше.

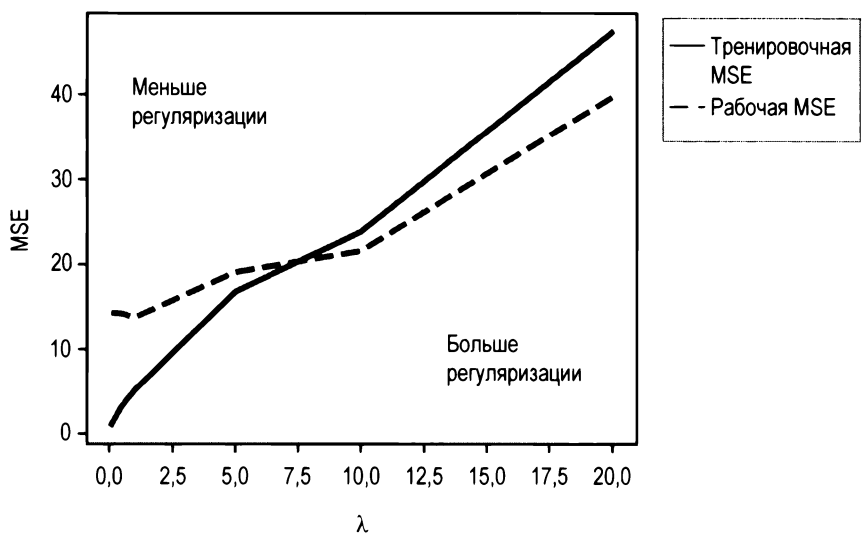


РИС. 5.4. Поведение MSE для тренировочного (непрерывная линия) и рабочего (пунктирная линия) наборов данных для нашей сети при варьировании λ

ПРИМЕЧАНИЕ. Хороший способ оценить оптимальное значение регуляризационного параметра λ состоит в том, чтобы построить график своей оптимизационной метрики (в данном примере MSE) для тренировочного и рабочего наборов данных и понаблюдать за тем, как они ведут себя с разными значениями λ. Затем выбрать то значение, которое дает минимум вашей оптимизационной метрики на рабочем наборе данных и в то же время дает модель, которая больше не прилегает слишком плотно к тренировочным данным.

Теперь необходимо продемонстрировать эффекты регуляризации l_2 еще более наглядно. Рассмотрим набор данных, созданный с помощью следующего фрагмента кода:

```
nobs = 30

np.random.seed(42)

xx1 = np.array([np.random.normal(0.3,0.15) for i in range (0,nobs)])
yy1 = np.array([np.random.normal(0.3,0.15) for i in range (0,nobs)])
xx2 = np.array([np.random.normal(0.1,0.1) for i in range (0,nobs)])
yy2 = np.array([np.random.normal(0.3,0.1) for i in range (0,nobs)])

c1_ = np.c_[xx1.ravel(), yy1.ravel()]
c2_ = np.c_[xx2.ravel(), yy2.ravel()]

c = np.concatenate([c1_,c2_])

yy1_ = np.full(nobs, 0, dtype=int)
yy2_ = np.full(nobs, 1, dtype=int)
yyL = np.concatenate((yy1_, yy2_), axis = 0)

train_x = c.T
train_y = yyL.reshape(1,60)
```

Набор данных имеет два признака: `x` и `y`. Мы генерируем две группы точек, `xx1,yy1` и `xx2,yy2`, из нормального распределения. Первой группе мы назначаем метку 0 (содержащуюся в массиве `yy1_`), а второй группе — метку 1 (в массиве `yy2_`). Теперь воспользуемся сетью, подобной описанной выше (с 4 слоями с 20 нейронами каждый), для того, чтобы выполнить бинарную классификацию на этом наборе данных. Мы можем взять тот же исходный код, что и раньше, изменив выходной слой и стоимостную функцию. Вы помните, что для бинарной классификации в выходном слое нам нужны один нейрон с сигмоидальной активационной функцией

```
y_, W5, b5 = create_layer (hidden4, n_outputs, activation=tf.sigmoid)
```

и следующая ниже стоимостная функция:

```
cost_class = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
cost = tf.reduce_mean(cost_class + lambd*reg)
```

Все остальное остается таким же, как было описано ранее. Построим для этой задачи график границы принятия решения[4]. Это означает, что мы выполним нашу сеть на наборе данных с помощью инструкции

```
sess, cost_history = model(learning_r=0.005,
                           training_epochs=100,
                           features=train_x,
```

[4] В задаче статистической классификации с двумя классами граница принятия решения, или поверхность принятия решения — это поверхность, которая разбивает базовое пространство на два множества, по одному для каждого класса. (Источник: Википедия, **https://goo.gl/E5nELL**).

```
                target=train_y,
                logging_step=10,
                lambd_val=0.0)
```

На рис. 5.5 показаны наборы данных, в которых белые точки относятся к первому классу, а черные — ко второму. Серым цветом обозначена зона, которую сеть классифицирует как один класс, и белая — как другой. Хорошо видно, что эта сеть способна гибко улавливать сложную структуру данных.

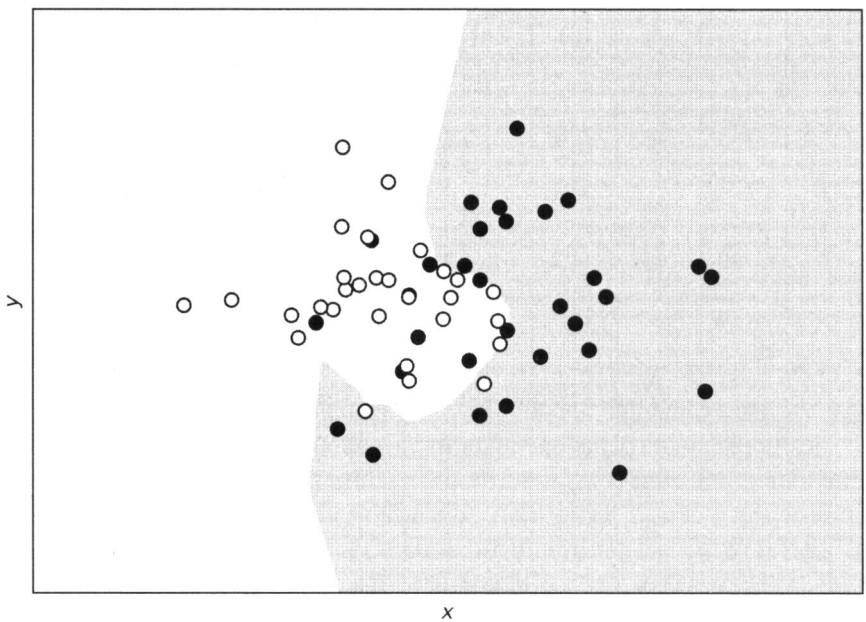


РИС. 5.5. Граница принятия решения без регуляризации. Белые точки представлют первый класс, черные — второй

Теперь давайте применим к сети регуляризацию, точно так же, как мы делали раньше, и посмотрим, как изменяется граница принятия решения. Здесь мы будем использовать регуляризационный параметр $\lambda = 0{,}1$.

Как явствует из рис. 5.6, граница принятия решения является почти линейной и больше не способна улавливать сложную структуру данных. Именно то, что мы и ожидали: регуляризационный член упрощает модель и, следовательно, делает ее менее способной улавливать тонкие структуры.

Интересно сравнить границу принятия решения сети с результатом логистической регрессии только с одним нейроном. Из соображений экономии пространства соответствующий исходный код отсутствует, но если вы сравните две границы принятия решения на рис. 5.7 (та, которая происходит из сети с одним нейроном, является линейной), то увидите, что они почти одинаковы. Регуляризационный член $\lambda = 0{,}1$ дает те же результаты, что и сеть с одним нейроном.

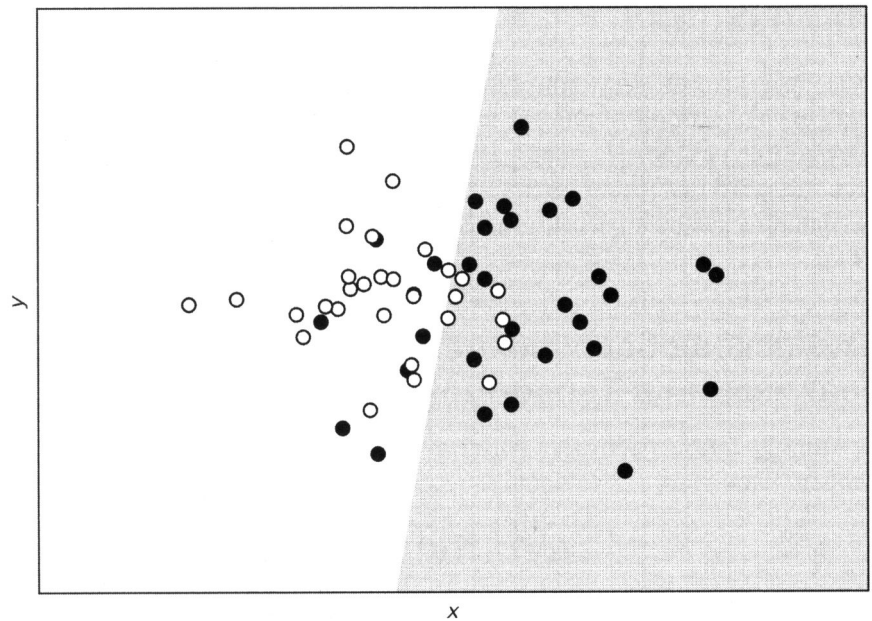


РИС. 5.6. Граница принятия решения, предсказанная сетью с регуляризацией l_2 и регуляризационным параметром $\lambda = 0{,}1$

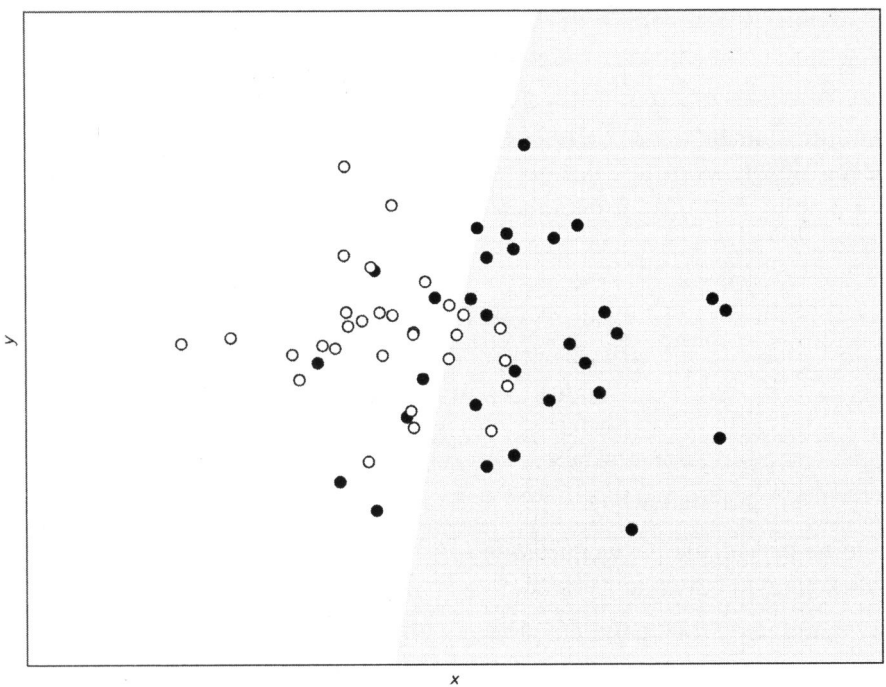


РИС. 5.7. Границы принятия решения для сложной сети с $\lambda = 0{,}1$ и для сети только с одним нейроном. Две границы почти полностью совпадают

Регуляризация l_1

Теперь мы рассмотрим регуляризационное техническое решение, которое очень похоже на регуляризацию l_2. Оно основано на том же принципе, добавлении члена в стоимостную функцию. На этот раз математическая форма добавленного члена отличается, но метод работает очень похоже на то, что объяснялось в предыдущих разделах. Давайте сначала рассмотрим математический механизм в основе данного алгоритма.

Теоретическое обеспечение регуляризации l_1 и ее реализации в TensorFlow

Регуляризация l_1 также работает, когда в стоимостную функцию добавляется дополнительный член

$$\tilde{J}(\mathbf{w}) = J(\mathbf{w}) + \frac{\lambda}{m}\|\mathbf{w}\|_1 .$$

Эффект, который он оказывает на заучивание, фактически такой же, как был описан в случае с регуляризацией l_2. TensorFlow не имеет готовой к использованию функции, как для l_2. Мы должны запрограммировать ее вручную, используя следующие строки кода:

```
reg = tf.reduce_sum(tf.abs(W1))+tf.reduce_sum(tf.abs(W2))+ \
      tf.reduce_sum(tf.abs(W3))+tf.reduce_sum(tf.abs(W4))+ \
      tf.reduce_sum(tf.abs(W5))
```

Остальной исходный код остается без изменений. Можно снова сравнить распределение весов между моделью без регуляризационного члена ($\lambda = 0$) и с регуляризацией ($\lambda = 3$, рис. 5.8). Для расчета мы использовали бостонский набор данных. Мы натренировали модель, вызвав модель следующим образом:

```
sess, cost_history = model(learning_r=0.01,
                           training_epochs=1000,
                           features=train_x,
                           target=train_y,
                           logging_step=1000,
                           lambd_val=3.0)
```

один раз с $\lambda = 0$ и один раз с $\lambda = 3$.

Как видно на графиках, регуляризация l_1 имеет тот же эффект, что и l_2. Она уменьшает эффективную сетевую сложность, сводя многочисленные веса до нуля.

Для получения представления об эффективности данной регуляризации в сокращении весов взгляните на табл. 5.2, в которой сравнивается процентное соотношение весов менее 10^{-3} с регуляризацией и без нее после 1000 эпох.

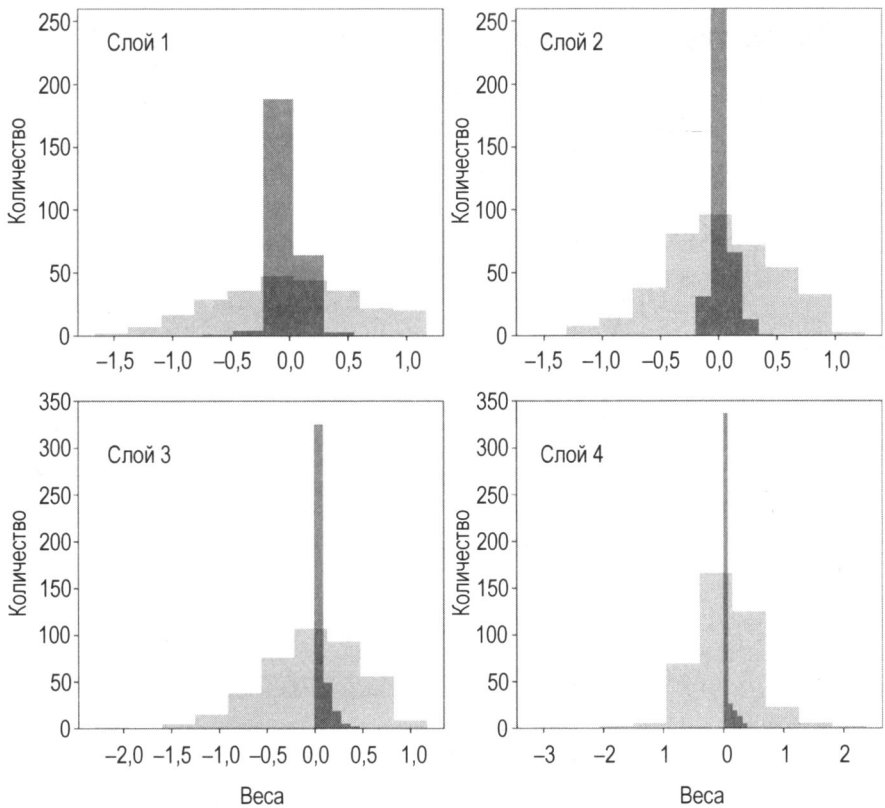


РИС. 5.8. Сравнение распределения весов между моделью без регуляризационного члена l_1 ($\lambda = 0$, светло-серый) и с регуляризацией l_1 ($\lambda = 3$, темно-серый)

Таблица 5.2. *Сравнение процентного соотношения весов менее 10^{-3} с регуляризацией и без нее*

Слой	% весов менее 10^{-3} для $\lambda = 0$	% весов менее 10^{-3} для $\lambda = 3$
1	0,0	52,7
2	0,25	53,8
3	0,75	46,3
4	0,25	45,3
5	0,0	60,0

Веса действительно сходятся к нулю?

Очень поучительно посмотреть на то, как веса сходятся к нулю. На рис. 5.9 показан вес $w_{12,5}^{[3]}$ (из слоя 3) в сопоставлении с числом эпох для искусственного набора данных с двумя признаками, регуляризацией l_2, $\gamma = 10^{-3}$, $\lambda = 0{,}1$, после 1000 эпох. Вы видите, как он быстро уменьшается до нуля. Значение после 1000 эпох равно $2 \cdot 10^{-21}$, поэтому во всех отношениях оно равно нулю.

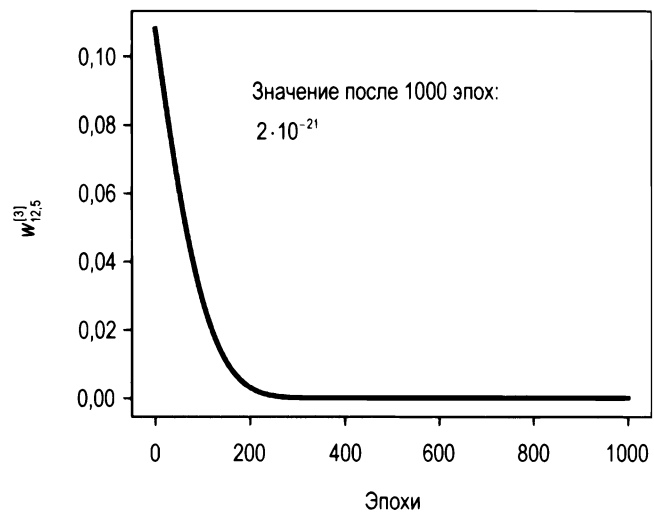


РИС. 5.9. Вес $w_{12,5}^{[3]}$ в сопоставлении с эпохами для искусственного набора данных с двумя признаками, регуляризацией l_2, $\gamma = 10^{-3}$, $\lambda = 0{,}1$, натренированный в течение 1000 эпох

В случае если вам интересно, вес сходится к нулю почти экспоненциально. Понять, почему это происходит, можно следующим образом. Рассмотрим уравнение обновления одного веса.

$$w_{j,[n+1]} = w_{j,[n]}\left(1 - \frac{\gamma\lambda}{m}\right) - \frac{\gamma \partial J\left(\mathbf{w}_{[n]}\right)}{\partial w_j}.$$

Теперь предположим, что мы находимся близко к минимуму, в участке, где производная от стоимостной функции J почти равна нулю, так что мы можем ею пренебречь. Другими словами, предположим

$$\frac{\partial J\left(\mathbf{w}_{[n]}\right)}{\partial w_j} \approx 0.$$

Мы можем переписать уравнение обновления весов как

$$w_{j,[n+1]} - w_{j,[n]} = -w_{j,[n]}\frac{\gamma\lambda}{m}.$$

Теперь уравнение можно прочитать следующим образом: темп вариации веса по числу итераций пропорционален самому весу. Те из вас, кто знает дифференциальные уравнения, могут понять, что мы можем провести параллель со следующим уравнением:

$$\frac{dx(t)}{dt} = -\frac{\gamma\lambda}{m}x(t).$$

Его можно прочитать как темп вариации $x(t)$ по времени пропорционален самой функции. Те, кто знает, как решать это уравнение, могут знать, что обобщенным решением является

$$x(t) = Ae^{-\frac{\gamma\lambda}{m}(t-t_0)}.$$

Теперь вы понимаете, проведя параллель между двумя уравнениями, почему затухание весов будет подобно затуханию экспоненциальной функции. На рис. 5.10 вы видите уже обсуждавшееся затухание весов с чистым экспоненциальным затуханием. Как и ожидалось, две кривые не являются идентичными, потому что, в особенности в самом начале, градиент стоимостной функции, безусловно, не равен нулю. Но сходство — поразительное и дает нам представление о том, как быстро веса могут падать до нуля (читай: очень быстро).

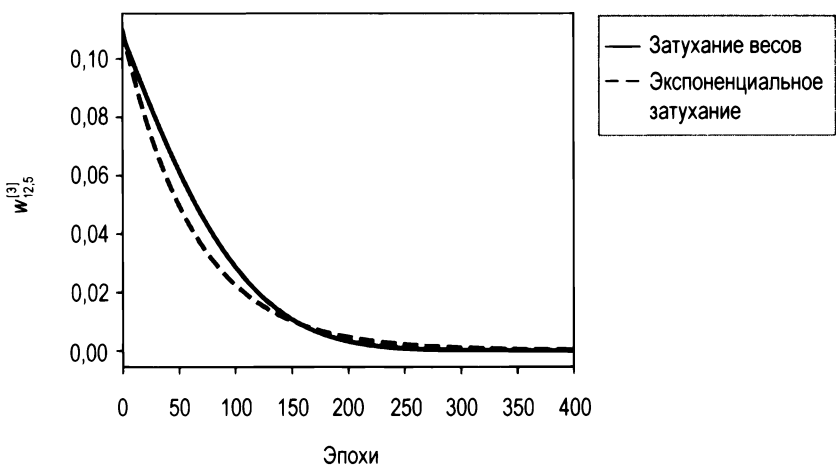


РИС. 5.10. Вес $w^{[3]}_{12,5}$ в сопоставлении с эпохами для искусственного набора данных с двумя признаками, регуляризацией l_2, $\gamma = 10^{-3}$, $\lambda = 0{,}1$, натренированный в течение 1000 эпох (непрерывная линия) вместе с чистым экспоненциальным затуханием (пунктирная линия), предоставленным в качестве иллюстрации

Обратите внимание, что при использовании регуляризации вы получаете тензоры с большим числом нулевых элементов. Такие тензоры называются разреженными. В дальнейшем вы можете извлечь выгоду из специальных процедур, которые чрезвычайно эффективны в работе с разреженными тензорами. Это следует иметь в виду, когда вы начинаете двигаться к более сложным моделям, но данная тема является слишком продвинутой для этой книги и потребует слишком много места.

Отсев

Основная идея отсева отличается: во время тренировочной фазы вы случайно удаляете узлы из слоя l с вероятностью $p^{[l]}$. На каждой итерации вы удаляете разные узлы, фактически тренируя на каждой итерации другую сеть (например, при использовании мини-пакетов вы тренируете другую сеть для каждого пакета). Обычно вероятность (часто называемая в Python `keep_prob`) устанавливается одинаковой для всей сети (но, технически говоря, она может быть специфичной для каждого слоя). В интуитивном плане, давайте рассмотрим выходной тензор `Z` слоя l. В Python можно определить такой вектор, как

```
d = np.random.rand(Z.shape[0], Z.shape[1]) < keep_prob
```

а затем просто умножить выходной слой `Z` на `d`, как показано ниже:

```
Z = np.multiply(Z, d)
```

В результате будут фактически удалены все элементы, вероятность которых меньше `keep_prob`. Очень важно не использовать отсев при выполнении предсказаний на рабочем наборе данных!

ПРИМЕЧАНИЕ. Во время тренировки каждую итерацию отсев случайно удаляет узлы. Но при выполнении предсказаний на рабочем наборе данных должна использоваться вся сеть целиком без отсева. Другими словами, вы должны установить `keep_prob=1`.

Отсев может быть специфичным для каждого слоя. Например, для слоев с большим числом нейронов `keep_prob` может быть небольшим. Для слоев с несколькими нейронами можно установить `keep_prob = 1.0`, фактически оставляя в таких слоях все нейроны нетронутыми.

Реализация в TensorFlow проста. Сначала необходимо определить заполнитель, который будет содержать значение параметра `keep_prob`

```
keep_prob = tf.placeholder(tf.float32, shape=())
```

а затем для каждого слоя добавить регуляризационную операцию. Это делается следующим образом:

```
hidden1, W1, b1 = create_layer (X, n1, activation = tf.nn.relu)
hidden1_drop = tf.nn.dropout(hidden1, keep_prob)
```

Затем, при создании следующего слоя, вместо `hidden1` вы используете `hidden1_drop`. Весь конструкционый исходный код выглядит следующим образом:

```
tf.reset_default_graph()


n_dim = 13
n1 = 20
n2 = 20
```

```
n3 = 20
n4 = 20
n_outputs = 1

tf.set_random_seed(5)

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])

learning_rate = tf.placeholder(tf.float32, shape=())
keep_prob = tf.placeholder(tf.float32, shape=())

hidden1, W1, b1 = create_layer (X, n1, activation = tf.nn.relu)
hidden1_drop = tf.nn.dropout(hidden1, keep_prob)
hidden2, W2, b2 = create_layer (hidden1_drop, n2, activation = tf.nn.relu)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)
hidden3, W3, b3 = create_layer (hidden2, n3, activation = tf.nn.relu)
hidden3_drop = tf.nn.dropout(hidden3, keep_prob)
hidden4, W4, b4 = create_layer (hidden3, n4, activation = tf.nn.relu)
hidden4_drop = tf.nn.dropout(hidden4, keep_prob)
y_, W5, b5 = create_layer (hidden4_drop, n_outputs, activation=tf.identity)

cost = tf.reduce_mean(tf.square(y_-Y))

optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                   beta1=0.9, beta2=0.999,
                                   epsilon=1e-8).minimize(cost)
```

Теперь проанализируем, что происходит со стоимостной функцией при использовании отсева. Давайте выполним модель применительно к бостонскому набору данных для двух значений переменной `keep_prob`: 1,0 (без отсева) и 0,5. На рис. 5.11 видно, что при применении отсева стоимостная функция очень нерегулярна. Она чрезмерно осциллирует. Обе модели были вычислены с помощью вызовов

```
sess, cost_history05 = model(learning_r=0.01,
                             training_epochs=5000,
                             features=train_x,
                             target=train_y,
                             logging_step=1000,
                             keep_prob_val=1.0)
```

для `keep_prob_val = 1.0` и `keep_prob_val = 0.5`.

На рис. 5.12 показана эволюция MSE для тренировочного и рабочего наборов данных в случае отсева (`keep_prob=0.4`).

На рис. 5.13 показан тот же график, но без отсева. Разница довольно разительная. Очень интересным является тот факт, что без отсева $MSE_{разработка}$ растет вместе с эпохами, тогда как используя отсев, она достаточна стабильна.

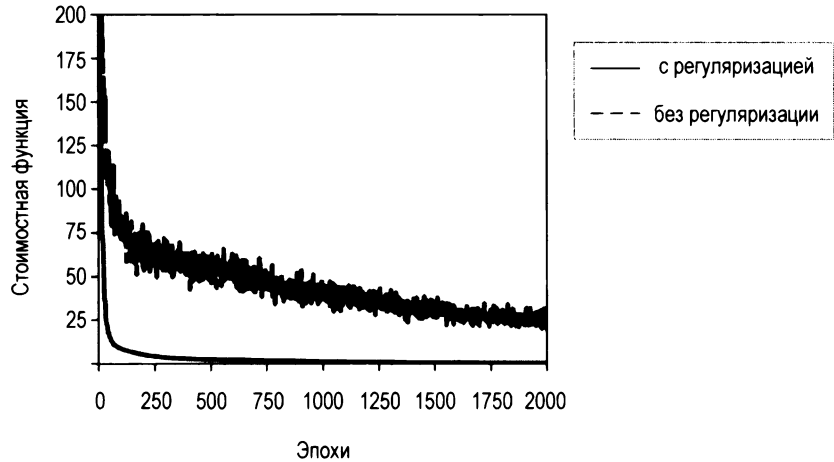


РИС. 5.11. Стоимостная функция для тренировочного набора данных для модели с двумя значениями переменной `keep_prob`: 1,0 (без отсева) и 0,5. Другие параметры: γ =0,01. Модели были натренированы в течение 5000 эпох. Мини-пакет не использовался. Осциллирующая линия вычислена с помощью регуляризации

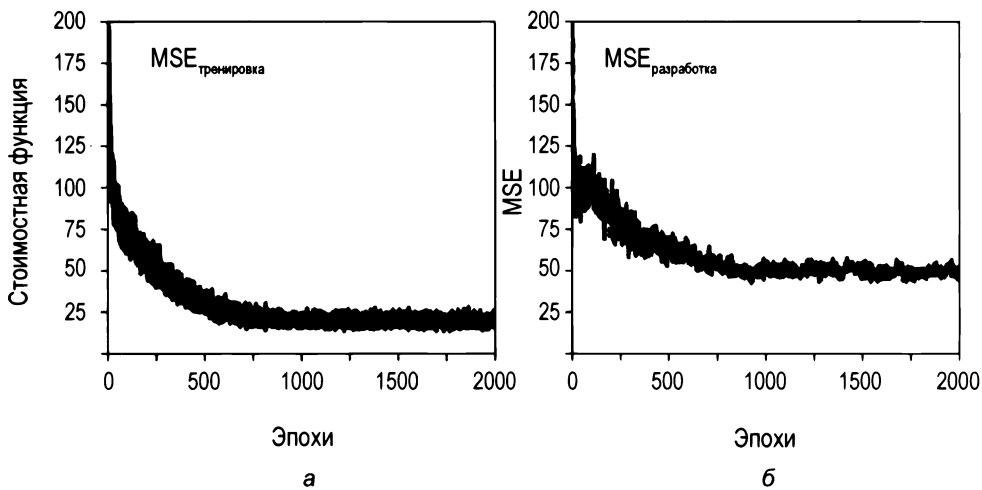


РИС. 5.12. MSE для тренировочного (*а*) и рабочего (*б*) наборов данных с отсевом (`keep_prob=0.4`)

На рис. 5.13 $MSE_{разработка}$ растет после падения в самом начале. Модель явно находится в режиме экстремальной переподгонки ($MSE_{тренировка} \ll MSE_{разработка}$), и она обобщает все хуже и хуже, когда применяется к новым данным. На рис. 5.12 видно, что $MSE_{тренировка}$ и $MSE_{разработка}$ имеют одинаковый порядок величины, и $MSE_{разработка}$ не продолжает расти. Таким образом, мы имеем модель, которая намного лучше обобщает, чем та, результаты которой показаны на рис. 5.13.

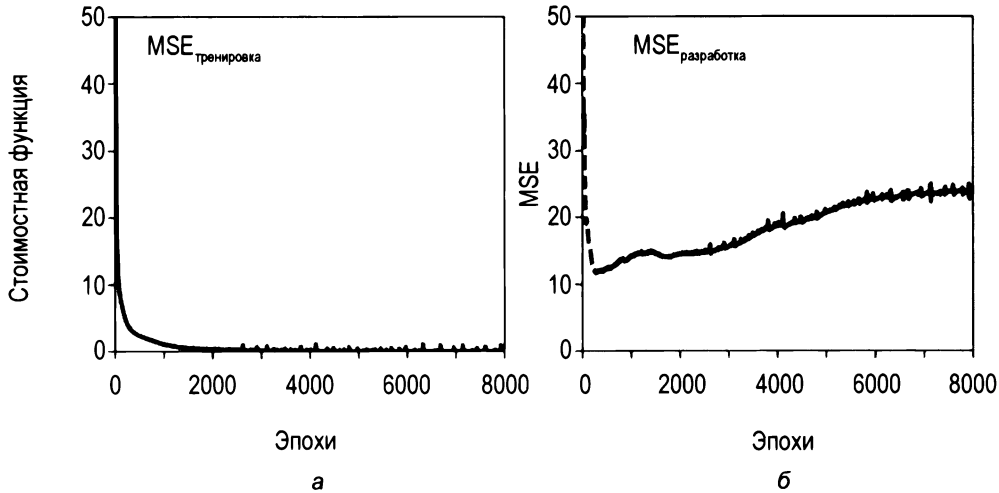


РИС. 5.13. MSE для тренировочного и рабочего наборов данных без отсева (`keep_prob=1.0`)

ПРИМЕЧАНИЕ. При применении отсева ваша метрика (в данном случае MSE) будет осциллировать, поэтому не удивляйтесь, когда будете пытаться отыскать наилучшие гиперпараметры и будете видеть, что оптимизационная метрика осциллирует.

Досрочная остановка

Для борьбы с переподгонкой иногда используется еще одно техническое решение. Строго говоря, это техническое решение не делает ничего, чтобы избежать переподгонки; оно просто прекращает процесс заучивания до того, как проблема с переподгонкой станет слишком серьезной. Рассмотрим пример в последнем разделе. На рис. 5.14 вы видите, что $MSE_{тренировка}$ и $MSE_{разработка}$ нанесены на одном и том же графике.

Досрочная остановка заключается в простой остановке процесса тренировки в точке, в которой $MSE_{разработка}$ имеет свой минимум (см. рис. 5.14, минимум обозначен вертикальной линией). Обратите внимание, что это не идеальный способ решения проблемы переподгонки. Ваша модель, скорее всего, будет очень плохо обобщать на новые данные. Обычно предпочтение отдается использованию других технических решений. Вдобавок этот ручной способ является времязатратным, а также очень подвержен ошибкам. Хороший обзор разных контекстов приложений можно получить, обратившись к странице Википедии, посвященной ранней остановке: **https://goo.gl/xnKo2s**.

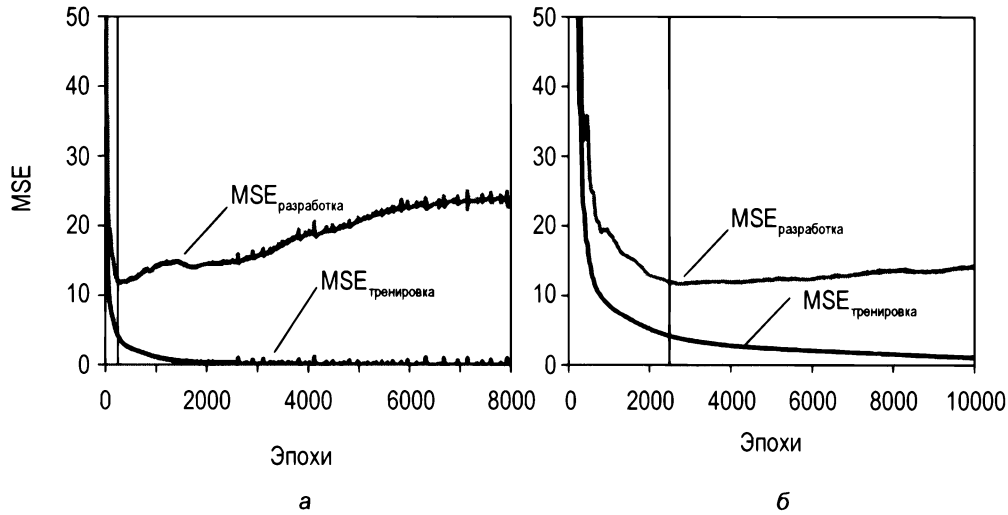


РИС. 5.14. MSE для тренировочного и рабочего наборов данных без отсева (`keep_prob=1.0`). Досрочная остановка заключается в том, чтобы остановить фазу самообучения на итерации, когда $MSE_{разработка}$ является минимальной (на графике обозначается вертикальной линией). Справа вы видите увеличенный участок левого графика для первых 1000 эпох

Дополнительные методы

Все обсуждавшиеся до сих пор методы в той или иной форме состоят в том, чтобы сделать модель менее сложной. Вы оставляете данные как есть и модифицируете модель. Но можно попробовать сделать наоборот: оставить модель как есть и поработать с данными. Приведем две общие стратегии, которые служат для борьбы с переподгонкой (но не очень легко применимые).

- *Получить больше данных.* Это самый простой способ борьбы с переподгонкой. К сожалению, очень часто в реальной жизни он невозможен. Имейте в виду, что этот сложный вопрос будет рассматриваться в следующей главе. Если вы классифицируете снимки кошек, сделанные с помощью смартфона, то можете подумать о получении дополнительных данных из Интернета. Хотя это может показаться удачной идеей, вы можете обнаружить, что изображения имеют разное качество, что, возможно, не на всех фотографиях действительно запечатлены кошки. (Как насчет игрушек в виде кошек?) Кроме того, вы можете найти изображения только молодых белых кошек и т. д. В сущности, ваши дополнительные наблюдения могут, по всей видимости, происходить из распределения, которое очень отличается от исходных данных, и, как вы увидите дальше, это станет проблемой. Таким образом, при получении дополнительных данных следует рассматривать потенциальные проблемы задолго до начала работы.

- *Усилить свои данные.* Например, если вы работаете с изображениями, то можно создать дополнительные данные, повернув, растянув, сдвинув свои изображения или выполнив иные виды обработки. Это очень распространенное техническое решение, которое может быть по-настоящему полезным.

Одной из главных целей машинного обучения является решение задачи улучшения способности модели обобщать на новых данных. Эта задача сложна, требует опыта и тестов. Много тестов. Во время работы над очень сложными задачами проводится целый ряд исследований в попытке решить эти виды дефектов. Дополнительные технические решения будут представлены в следующей главе.

ГЛАВА 6

Метрический анализ

Давайте рассмотрим задачу, детально проанализированную в *главе 3*, для которой мы выполнили классификацию на наборе данных Zalando. Выполняя всю нашу работу, мы приняли сильное допущение, не заявив о нем явным образом: мы допустили, что все наблюдения были помечены правильно. Однако мы не можем сказать этого с уверенностью. Для выполнения разметки требовалось некоторое ручное вмешательство, и, следовательно, определенное число изображений было, безусловно, классифицировано ошибочно, поскольку люди не совершенны. И это откровение является чрезвычайно важным. Рассмотрим следующий сценарий: в *главе 3* модель достигла примерно 90%-й точности. Можно было бы пытаться получать все более и более высокую точность, но когда будет разумным прекратить попытки? Если ваши метки являются ошибочными в 10% случаев, то ваша модель, какой бы сложной она ни была, никогда не сможет обобщать на новых данных с очень высокой точностью, потому что для многих изображений она заучит неправильные классы. Мы потратили довольно много времени на проверку и подготовку тренировочных данных, например на их нормализацию, но мы ни разу не тратили время на проверку самих меток. Мы также приняли допущение, что все классы имеют сходные характеристики. (Позже в этой главе будет показано, что именно это означает, но пока достаточно интуитивного понимания идеи.) Что делать, если качество изображений определенных классов хуже, чем других? Что делать, если число пикселов, чье значение оттенков серого отличается от нуля, резко различается у разных классов? Мы также не проверяли, являются ли некоторые изображения совершенно пустыми. Что происходит в этом случае? Как вы понимаете, мы не можем проверить все изображения вручную в попытке обнаружить такие проблемы. Предположим, у нас миллионы изображений. Ручной анализ, конечно же, не представляется возможным.

Здесь в нашем арсенале нужно новое оружие, которое позволило бы обнаруживать такие случаи и давало возможность говорить о результативности (performance) работы модели. Это новое оружие находится в центре внимания данной главы, и я

называю его "метрическим анализом". Очень часто специалисты в данной сфере именуют этот массив методов "анализом ошибок". По-моему, это название является довольно путанным, в особенности для начинающих. Термин "ошибка" может обозначать слишком многие вещи: дефекты Python-кода, ошибки в методах, в алгоритмах, ошибки в выборе оптимизаторов и т. д. В этой главе вы увидите способы получения фундаментальной информации о том, насколько хорошо ваша модель работает и насколько хороши ваши данные. Мы сделаем это, оценив вашу оптимизационную метрику на множестве разных наборов данных, которые можно получить из ваших данных.

Ранее вы уже видели простой пример. Как вы помните из обсуждения регрессии, мы говорили о том, каким образом в случае $MSE_{тренировка} \ll MSE_{разработка}$ мы попадаем в режим переподгонки. Нашей метрикой является MSE (среднеквадратическая ошибка), и ее оценивание на двух наборах данных, тренировочном и рабочем, и сравнение этих двух значений говорит нам о том, является ли модель переподогнанной. В этой главе данная методология будет расширена, что позволит вам извлекать гораздо больше информации из данных и модели.

Человеческая результативность и байесова ошибка

В большинстве наборов данных, которые мы используем для контролируемого обучения, кто-то уже разметил все наблюдения. Возьмем, к примеру, набор данных, в котором мы имеем изображения, подлежащие классифицированию. Если мы попросим людей классифицировать все изображения (представим, что это возможно, независимо от числа изображений), то полученная точность никогда не будет равняться 100%. Некоторые изображения могут быть настолько размытыми, что они будут классифицированы неправильно, а люди склонны совершать ошибки. Если, например, 5% изображений не поддаются правильной классификации, например, из-за того, что они очень размыты, то мы должны ожидать, что максимальная точность, которую могут достичь люди, всегда будет меньше 95%.

Рассмотрим классификационную задачу. Прежде всего, давайте определим, что мы подразумеваем под словом "ошибка". В этой главе слово "ошибка" будет использоваться для обозначения следующей величины, обозначаемой буквой ε:

$$\varepsilon = 1 - \text{точность}.$$

Например, если с помощью модели мы достигнем точности 95%, то получим $\varepsilon = 1 - 0,95 = 0,05$ или, в процентном выражении, $\varepsilon = 5\%$.

Очень полезно знать понятие "человеческая результативность", которое можно определить следующим образом.

Определение 1. Человеческая результативность (human-level performance) — это наименьшее значение ошибки ε, которое может быть достигнуто лицом, выполняющим классификационную задачу. Обозначим ее с помощью $\varepsilon_{чел}$.

Приведем конкретный пример. Будем считать, что у нас есть набор из 100 изображений. Теперь предположим, что мы просим трех человек расклассифицировать 100 изображений. Представим, что они получают точности 95, 93 и 94%. В этом случае человеческая точность работы будет равна $\varepsilon_{\text{чел}} = 5\%$. Обратите внимание, что кто-то другой может справиться с этой задачей намного лучше, и, следовательно, всегда важно учитывать, что значение $\varepsilon_{\text{чел}}$, которое мы получаем, всегда является оценочным и должно служить только в качестве ориентира.

Теперь давайте немного усложним ситуацию. Допустим, мы работаем над задачей, в которой врачи классифицируют магнитно-резонансные томограммы (МРТ) по двум классам: с признаками онкологического заболевания и без них. Теперь предположим, что мы вычисляем $\varepsilon_{\text{чел}}$, получив 15% по результатам неопытных студентов, 8% от врачей с несколькими годами стажа, 2% от опытных врачей и 0,5% от опытных групп врачей. Каким тогда будет $\varepsilon_{\text{чел}}$? По причинам, которые будут изложены позже, вы всегда должны выбирать максимально низкое значение, которое можно получить.

Теперь можно расширить понятие $\varepsilon_{\text{чел}}$ вторым определением.

Определение 2. Человеческая результативность — это наименьшее значение ошибки ε, которое может быть достигнуто лицами или группами лиц, выполняющими классификационную задачу.

ПРИМЕЧАНИЕ. Вам не обязательно решать, какое определение является правильным. Используйте то, которое дает наименьшее значение $\varepsilon_{\text{чел}}$.

Теперь немного о том, почему следует выбирать наименьшее значение, которое можно получить для $\varepsilon_{\text{чел}}$. Предположим, что из 100 изображений 9 слишком размыты для того, чтобы их можно было правильно классифицировать. Это означает, что наименьшая ошибка любого классификатора способна достичь 9%. Самая низкая ошибка, которая может быть достигнута любым классификатором, называется *байесовой ошибкой*[1]. Будем обозначать ее как $\varepsilon_{\text{Байес}}$. В данном примере $\varepsilon_{\text{Байес}} = 9\%$. Обычно $\varepsilon_{\text{чел}}$ очень близка к $\varepsilon_{\text{Байес}}$, по крайней мере, в задачах, в которых люди преуспевают, таких как распознавание изображений. Обычно говорят, что ошибка человеческой результативности является индикатором байесовой ошибки. Обычно невозможно или очень трудно узнать ошибку $\varepsilon_{\text{Байес}}$, и поэтому на практике специалисты используют ошибку $\varepsilon_{\text{чел}}$, исходя из того, что обе они находятся близко, потому что последнюю (относительно) легче оценить.

Учтите, что имеет смысл сравнивать эти два значения и исходить из того, что $\varepsilon_{\text{чел}}$ является индикатором $\varepsilon_{\text{Байес}}$ только в том случае, если лица (или группы лиц) вы-

[1] Байесова ошибка (bayes error) — это наименьшая возможная ошибка любого классификатора со случайным конечным результатом (например, на одну из двух категорий). Она аналогична неисправимой ошибке. См. **https://en.wikipedia.org/wiki/Bayes_error_rate**. — *Прим. пер.*

полняют классификацию таким же образом, как и классификатор. Например, совершенно нормально, если при классифицировании оба используют одни и те же изображения. Но если в нашем примере с раком для онкологической диагностики врачи используют дополнительные сканы и анализы, то сравнение больше не является справедливым, потому что человеческая результативность больше не будет индикатором байесовой ошибки. Врачи, которые в своем распоряжении имеют больше данных, очевидно, будут точнее, чем модель, которая на входе в своем распоряжении имеет только изображения.

ПРИМЕЧАНИЕ. $\varepsilon_{чел}$ и $\varepsilon_{Байес}$ находятся близко друг к другу только в тех случаях, когда классификация производится людьми и моделью одинаковым образом. Поэтому всегда проверяйте, так ли это, и только потом делайте допущение о том, что человеческая результативность является индикатором байесовой ошибки.

Во время работы вы также заметите, что при относительно небольших усилиях вы можете достичь довольно низкой частоты ошибок и часто (почти) достичь $\varepsilon_{чел}$. Как показано на рис. 6.1, после прохождения человеческой результативности (и, в некоторых случаях, это возможно) продвижение, как правило, становится очень и очень медленным.

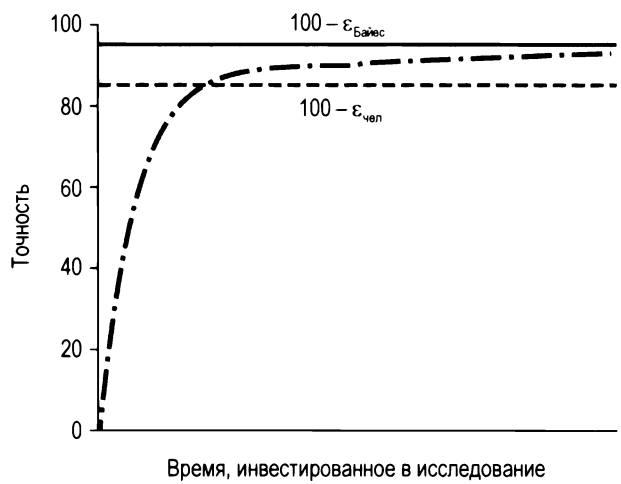


РИС. 6.1. Типичные значения точности, которые могут быть достигнуты в сопоставлении с инвестированным количеством времени. В самом начале с помощью машинного обучения очень легко достигается довольно хорошая точность, нередко $\varepsilon_{чел}$. На это интуитивно указывает линия на графике, после чего продвижение становится очень медленным

До тех пор, пока ошибка вашего алгоритма больше $\varepsilon_{чел}$, для получения более высоких результатов можно применять следующие технические решения:

- получить более качественные метки от отдельных лиц или групп лиц, например от групп врачей, как в случае с медицинскими данными в нашем примере;

- получить больше помеченных данных от отдельных лиц или групп лиц;
- провести хороший метрический анализ с целью определения наилучшей стратегии получения более качественных результатов. Вы узнаете, как это делается, в данной главе.

Как только ваш алгоритм превысит человеческую результативность, вы больше не сможете полагаться на эти технические решения. Поэтому важно иметь представление об этих цифрах, что позволит принять решение о дальнейших действиях для получения более высоких результатов. Взяв наш пример с МРТ-сканами, мы могли бы получить более качественные метки, опираясь на источники, которые не связаны с людьми, например, проверяя диагнозы через несколько лет после даты МРТ, когда обычно уже становится очевидным, развилось у пациента онкологическое заболевание или нет. Либо, например, в случае классифицирования изображений вы можете самостоятельно взять несколько тысяч изображений определенных классов. Обычно это невозможно, но здесь важно прояснить идею: вы можете получить метки, используя другие средства, а не просить людей выполнять тот же вид задач, который выполняет ваш алгоритм.

ПРИМЕЧАНИЕ. Человеческая результативность является хорошим индикатором байесовой ошибки в задачах, в которых люди справляются отлично, таких как распознавание образов. В задачах, в которых люди справляются очень плохо, результативность может быть весьма далека от байесовой ошибки.

Краткая история человеческой результативности

Показательна в этой связи история работы, которую проделал Андрей Карпати (Andrej Karpathy), пытаясь оценить человеческую результативность в конкретном случае. Вы можете прочитать всю историю в его блоге по адресу **https://goo.gl/iqCbC0** (этот пост, хоть и длинный, стоит того, чтобы его прочитать). Сперва следует подытожить то, что он сделал, поскольку это чрезвычайно поучительно для осмысления человеческой результативности. В 2014 году Карпати участвовал в конкурсе ILSVRC (ImageNet Large Scale Visual Recognition Challenge — конкурс по крупномасштабному распознаванию изображений ImageNet; **https://goo.gl/PCHWMT**). Задача состояла из 1,2 млн изображений (тренировочный набор), классифицированных по 1000 категориям, включая такие объекты, как животные, абстрактные объекты, в частности спираль, сцены и многое другое. Результаты оценивались на рабочем наборе данных. Разработанная Google модель GoogleLeNet достигла поразительной ошибки — всего 6,7%. Карпати было интересно, как этот результат соотносится с людьми.

Вопрос является гораздо более сложным, чем он выглядит на первый взгляд. Поскольку все изображения были классифицированы людьми, разве не должно $\varepsilon_{чел} = 0\%$? Вообще-то, нет. На самом деле, изображения сначала были получены

с помощью веб-поиска, затем они фильтровались и помечались с помощью заданных людям бинарных вопросов типа "это крючок или нет"? Как отмечает Карпати в своем блоге, изображения собирались в бинарном виде. Людей не просили назначать каждому изображению класс, выбирая из имеющихся 1000, как это делали алгоритмы. Вы можете подумать, что это техническая деталь, но разница в проведении разметки значительно усложняет правильное оценивание $\varepsilon_{чел}$. Итак, Карпати приступил к работе и разработал веб-интерфейс, который состоял из изображения, расположенного слева, и 1000 классов с примерами — справа. Пример интерфейса приведен на рис. 6.2. Вы можете поработать с ним (настоятельно советуем это сделать) на странице **https://goo.gl/Rh8S6g**, для того чтобы понять, насколько сложна такая задача. Люди, пробовавшие этот интерфейс, периодически пропускали классы и делали ошибки. Лучшая ошибка из достигнутых составила около 15%.

Таким образом, Карпати сделал то, что в какой-то момент своей карьеры должен сделать каждый ученый: ему стало до смерти скучно, и он проделал тщательную аннотацию снимков самостоятельно, иногда тратя по 20 минут на одно изображение. Карпати заявляет в своем блоге, что сделал это исключительно ради науки, #forscience. Он смог достичь потрясающего результата, $\varepsilon_{чел} = 5{,}1\%$, на 1,7% лучше, чем лучший алгоритм в то время. Он перечислил источники ошибок, которым модель GoogLeNet была подвержена больше, чем люди, такие как задачи с несколькими объектами на снимке, и источники ошибок, к которым люди были подвержены больше, чем модель GoogLeNet, такие как задачи с классами, имеющими огромную гранулярность (к примеру, собаки классифицируются на 120 разных подклассов).

РИС. 6.2. Веб-интерфейс, разработанный Карпати. Не каждый нашел бы забавным разглядывать 120 пород собак, пытаясь классифицировать собаку слева (которая, кстати, является тибетским мастифом)

Если у вас есть несколько свободных часов, то рекомендуется попробовать. И вы совершенно по-новому взглянете на трудности в оценивании человеческой результативности. Задача определения и оценивания понятия человеческой результативности является очень сложной. Важно понимать, что $\varepsilon_{\text{чел}}$ зависит от того, как люди подходят к классификационной задаче, а это зависит от инвестированного времени, терпения выполняющих задачу людей и от многих факторов, которые с трудом поддаются квантификации. Главная причина, почему это так важно, помимо философского аспекта осознания момента, когда машина становится лучше людей, заключается в том, что ее часто принимают за индикатор байесовой ошибки, которая дает нижний предел наших возможностей.

Человеческая результативность на наборе данных MNIST

Прежде чем перейти к следующей теме, стоило бы привести еще один пример человеческой результативности на наборе данных, который мы проанализировали вместе: наборе данных MNIST. Человеческая результативность на этом наборе данных была широко проанализирована, и было обнаружено, что $\varepsilon_{\text{чел}} = 0,2\%$. (Вы можете прочитать хороший обзор Дэна Сиреджана (Dan Cireşan) по этому вопросу: "Multi-column Deep Neural Networks for Image Classification" (Многостолбцовые глубокие нейронные сети для классификации снимков", технический отчет № IDSIA-04-12, Институт искусственного интеллекта им. Далле Молле, **https://goo.gl/pEHZVB**.)) Теперь вы можете задаться вопросом: почему человек не может достичь 100%-й точности при классифицировании простых цифр? Но взгляните на рис. 6.3 и попытайтесь определить, какие цифры там изображены. Я не смог. И следовательно, вы, может быть, лучше поймете, почему невозможно, чтобы $\varepsilon_{\text{чел}} = 0\%$, и почему человек не может достичь 100%-й точности. Другие причины могут быть связаны с принадлежностью к той или иной культуре. В некоторых странах цифра, представляющая семь, записывается очень похоже, например с единицами, и в некоторых случаях могут быть допущены ошибки. В других странах цифра семь имеет небольшую черточку вдоль вертикального штриха, которая помогает ее отличать от единицы.

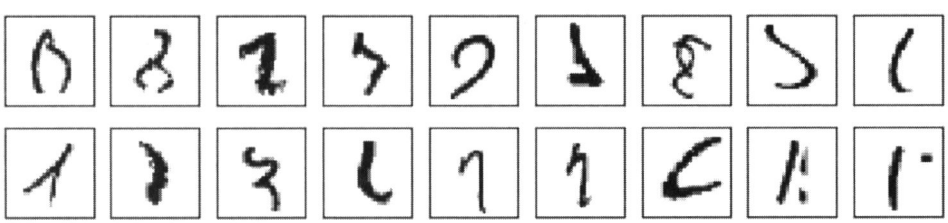


РИС. 6.3. Множество цифр из набора данных MNIST, которые практически невозможно распознать. Такие примеры являются одной из причин того, почему $\varepsilon_{\text{чел}}$ может быть нулевым

Смещение

Теперь давайте приступим к метрическому анализу: набору процедур, которые дадут вам информацию о результативности модели и качестве данных, хорошем или плохом, путем оценивания оптимизационной метрики на разных наборах данных.

ПРИМЕЧАНИЕ. Метрический анализ состоит из набора процедур, которые дают информацию о результативности модели и качестве данных, исходя из результатов оценивания оптимизационной метрики на разных наборах данных.

Для начала мы должны определить третью ошибку, т. е. ту, которая оценивается на тренировочном наборе данных, обозначаемую через $\varepsilon_{\text{тренировка}}$.

Нам предстоит ответить на первый вопрос: не является ли наша модель достаточно гибкой или сложной для того, чтобы быть в состоянии достичь человеческой результативности? Другими словами, мы хотим знать, имеет ли модель высокое смещение по отношению к человеческой результативности.

Для ответа на этот вопрос можно сделать следующее: вычислить ошибку из модели на тренировочном наборе данных $\varepsilon_{\text{тренировка}}$, а затем вычислить $\left|\varepsilon_{\text{тренировка}} - \varepsilon_{\text{чел}}\right|$. Если полученное число не будет малым (более нескольких процентов), то у нас имеется смещение (иногда именуемое *предотвратимым смещением*), т. е. модель слишком проста для того, чтобы улавливать реальные тонкости данных.

Давайте определим следующую величину:

$$\Delta\varepsilon_{\text{Смещение}} = \left|\varepsilon_{\text{тренировка}} - \varepsilon_{\text{чел}}\right|.$$

Чем больше $\Delta\varepsilon_{\text{Смещение}}$, тем больше в модели смещение. В этом случае вы хотите получить более высокие результаты на тренировочном наборе, потому что знаете, что на своих тренировочных данных вы можете добиться лучшего. (Мы рассмотрим проблему переподгонки через минуту.) Следующие ниже технические решения работают на сокращение смещения:

- более крупные сети (больше слоев или нейронов);
- более сложные архитектуры (например, сверточные нейронные сети);
- более продолжительная тренировка модели (в течение большего числа эпох);
- использование более качественных оптимизаторов (таких как Adam);
- выполнение более качественного гиперпараметрического поиска *(см. главу 7)*.

Необходимо понимать еще кое-что. Знать величину $\varepsilon_{\text{чел}}$ и сокращать смещение с целью ее достижения — это две совершенно разные вещи. Предположим, вы знаете $\varepsilon_{\text{чел}}$ своей задачи. Это не значит, что вы обязательно должны ее достичь. Вполне возможно, что вы используете неправильную архитектуру, но у вас может не быть навыков, необходимых для разработки более сложной сети. Возможно даже, что усилия, необходимые для достижения желаемого уровня ошибки, будут

чрезмерными (с точки зрения аппаратного обеспечения или инфраструктуры). Всегда учитывайте характер условий вашей задачи и старайтесь понять, что для нее подходит лучше всего. В случае приложения, которое распознает онкологическое заболевание, вы можете инвестировать как можно больше в достижение максимально возможной точности: вы же не хотите отправить кого-то домой только для того, чтобы обнаружить наличие рака через пару месяцев. С другой стороны, если вы создадите систему распознавания кошек из веб-снимков, то можете посчитать более высокую, чем $\varepsilon_{чел}$, ошибку совершенно приемлемой.

Диаграмма метрического анализа

В этой главе мы рассмотрим разные проблемы, с которыми вы столкнетесь при разработке моделей, и способы их выявления. Мы рассмотрели первую из них: смещение, иногда также именуемое предотвратимым смещением. Мы видели, как его можно обнаружить, вычислив $\Delta\varepsilon_{Смещение}$. В конце этой главы вы найдете несколько величин, которые можно вычислять для выявления проблем. Для того чтобы облегчить их понимание, здесь используется термин, который нередко называется *диаграммой метрического анализа* (metric analysis diagram, MAD). Она представляет собой простую гистограмму, в которой каждый столбик ассоциирован с проблемой. Давайте приступим к ее построению с единственной (на данный момент) обсуждаемой нами величиной: смещением. Это видно на рис. 6.4. На данный момент это довольно глупая диаграмма, но вы увидите, насколько полезно держать все под контролем, когда у вас несколько проблем одновременно.

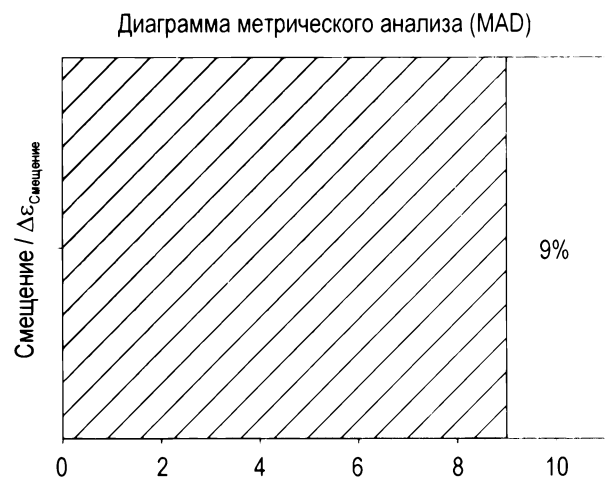


РИС. 6.4. Диаграмма метрического анализа (MAD) только с одной величиной из ряда других, с которыми мы столкнемся в этой главе: $\Delta\varepsilon_{Смещение}$

Переподгонка к тренировочному набору данных

Еще одна проблема, которую мы подробно обсуждали в предыдущих главах, — это чрезмерно плотная подгонка к тренировочным данным — переподгонка. Вы помните, что выполняя регрессию в *главе 5*, мы встретили экстремальный случай переподгонки, в котором $MSE_{тренировка} \ll MSE_{разработка}$. То же самое относится и к классификационной задаче. Давайте обозначим через $\varepsilon_{тренировка}$ ошибку модели на тренировочном наборе данных и через $\varepsilon_{разработка}$ ошибку на рабочем наборе данных. Тогда мы можем сказать, что мы слишком плотно аппроксимируем тренировочный набор, если $\varepsilon_{тренировка} \ll \varepsilon_{разработка}$. Давайте определим новую величину

$$\Delta\varepsilon_{переподгонка\ к\ трен\_набору} = \left|\varepsilon_{тренировка} - \varepsilon_{разработка}\right|.$$

Этой величиной мы говорим, что мы слишком плотно аппроксимируем тренировочный набор данных, если $\Delta\varepsilon_{переподгонка\ к\ трен\_набору}$ больше нескольких процентов.

Давайте подытожим то, что мы определили и рассмотрели до сих пор. У нас три ошибки:

- $\varepsilon_{тренировка}$ — ошибка классификатора на тренировочном наборе данных;
- $\varepsilon_{чел}$ — человеческая результативность (как обсуждалось в предыдущих разделах);
- $\varepsilon_{разработка}$ — ошибка классификатора на рабочем наборе данных.

С этими тремя величинами мы определили:

- $\Delta\varepsilon_{Смещение} = \left|\varepsilon_{тренировка} - \varepsilon_{чел}\right|$ — мера того, сколько "смещения" имеется между тренировочным набором данных и человеческой результативностью;
- $\Delta\varepsilon_{переподгонка\ к\ трен\_набору} = \left|\varepsilon_{тренировка} - \varepsilon_{разработка}\right|$ — мера количества переподгонки к тренировочному набору данных.

Вдобавок, до сих пор мы использовали два набора данных.

- Тренировочный набор данных: набор данных, который мы используем для тренировки модели (сейчас это должно быть уже понятно).
- Рабочий набор данных: второй набор данных, который мы используем для проверки на наличие переподгонки к тренировочному набору данных.

Теперь предположим, что наша модель имеет смещение и немного переподогнана к тренировочному набору данных, имея в виду, что мы имеем $\Delta\varepsilon_{Смещение} = 6\%$ и $\Delta\varepsilon_{переподгонка\ к\ трен\_набору} = 4\%$. Диаграмма MAD теперь выглядит так, как показано на рис. 6.5.

Как видно из рис. 6.5, вы получаете оперативный обзор относительной тяжести имеющихся проблем и можете решить, к какой из них обратиться в первую очередь.

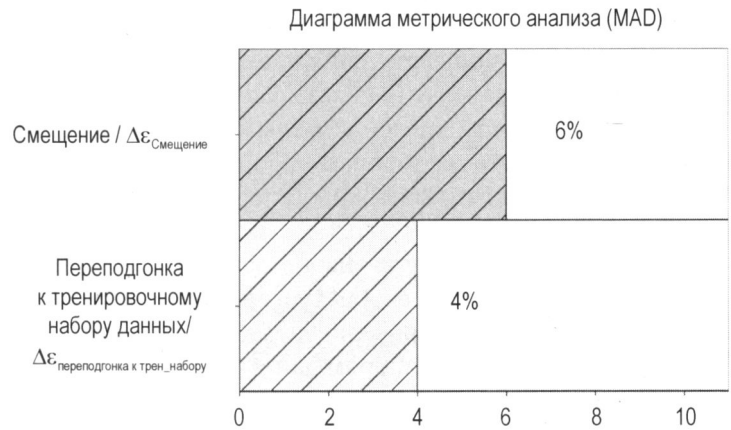


РИС. 6.5. Диаграмма MAD для двух проблем: смещение и переподгонка к тренировочному набору данных

Обычно, когда у вас переподгонка к тренировочному набору данных, такая проблема общеизвестна как проблема дисперсии. Когда это происходит, для минимизации этой проблемы вы можете попробовать следующие технические решения:

- получить для тренировочного набора больше данных;
- использовать регуляризацию (полное обсуждение данной темы *см. в главе 5*);
- попытаться усилить данные (например, если вы работаете с изображениями, то можно попробовать их повернуть, сдвинуть и т. д.);
- попробовать "более простые" сетевые архитектуры.

Как обычно, фиксированных правил нет, и вы должны проводить тестирование с целью нахождения таких технических решений, которые справляются с вашей проблемой лучше всего.

Тестовый набор

Следует бегло упомянуть еще одну проблему, с которой вы можете столкнуться. Мы рассмотрим ее подробно в *главе 7*, потому что она связана с гиперпараметрическим поиском. Вспомните процедуру отбора лучшей модели в проекте машинного обучения (это, кстати, не относится к глубокому обучению). Предположим, мы работаем над классификационной задачей. Прежде всего, мы решаем, какую оптимизационную метрику мы хотим задействовать. Допустим, мы решили использовать точность. Затем мы создаем первоначальную систему, вводим в нее тренировочные данные и смотрим, как она работает с рабочим набором данных с целью протестировать ее на наличие переподгонки к тренировочным данным. Как вы помните, в предыдущих главах мы часто говорили о гиперпараметрах — параметрах, на которые процесс заучивания не влияет. Примерами гиперпараметров являются темп заучивания, регуляризационный параметр и др. Многие из них мы встречали в пре-

дыдущих главах. Предположим, вы работаете с конкретной нейросетевой архитектурой. Вам нужно найти наилучшие значения гиперпараметров, которые позволят увидеть, насколько хорошо ваша модель показывает себя в работе. Для этого необходимо натренировать несколько моделей с разными значениями гиперпараметров и проверить их результативность на рабочем наборе данных. Нередко происходит так, что ваши модели будут показывать хорошие результаты на рабочем наборе данных, но при этом не будут обобщать вообще, потому что вы отбираете наилучшие значения, используя только рабочий набор данных. Выбирая специфические значения для своих гиперпараметров, вы рискуете вызвать переподгонку к рабочему набору данных. Для того чтобы проверить, так ли это, следует создать третий набор данных, именуемый тестовым набором данных, вырезав порцию наблюдений из исходного набора данных, которую вы затем можете использовать для проверки результативности моделей.

Мы должны определить новую величину:

$$\Delta\varepsilon_{\text{переподгонка к трен\_набору}} = \left|\varepsilon_{\text{разработка}} - \varepsilon_{\text{тест}}\right|,$$

где $\varepsilon_{\text{тест}}$ — ошибка, вычисленная на тестовом наборе. Мы можем добавить ее в диаграмму MAD (рис. 6.6).

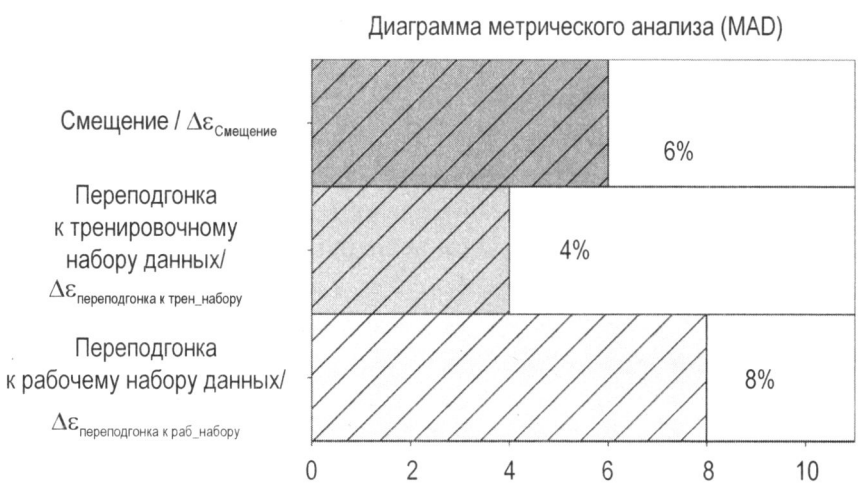


РИС. 6.6. Диаграмма MAD для трех проблем, с которыми мы можем столкнуться: смещение, переподгонка к тренировочному набору данных, переподгонка к рабочему набору данных

Обратите внимание, что если вы не проводите гиперпараметрического поиска, то тестовый набор данных вам не понадобится. Это полезно только тогда, когда вы проводите обширные поиски; в противном случае в большинстве случаев это бесполезно и отнимает наблюдения, которые могли бы использоваться для тренировки. Все предыдущее обсуждение базируется на допущении, что ваши наблюдения из рабочего и тестового наборов имеют одинаковые характеристики. Например, если вы работаете над задачей распознавания изображений и решили для трениро-

вочного и рабочего наборов данных использовать смартфонные снимки с высокой разрешающей способностью, а для тестового набора данных — изображения из Интернета с низкой разрешающей способностью, то вы можете увидеть большую величину $|\varepsilon_{\text{разработка}} - \varepsilon_{\text{тест}}|$, но это, вероятно, будет связано с различиями в изображениях, а не с проблемой переподгонки. Позже в этой главе будет показано, что может произойти, когда разные наборы поступают из различных распределений (еще один способ сказать, что наблюдения имеют разные характеристики), что именно это означает и что можно с этим сделать.

Как подразделить набор данных

Теперь кратко обсудим вопрос разбиения данных как в общем контексте, так и в контексте глубокого обучения.

Но что именно означает "разбиение"? Дело в том, что, как обсуждалось в предыдущем разделе, вам потребуется набор наблюдений, который заставляет модель обучаться. Этот набор данных вы называете тренировочным набором. Вам также понадобится набор наблюдений, который составит ваш рабочий набор, и заключительный набор, именуемый тестовым набором. Как правило, вы увидите пропорции разбиения, такие как 60% наблюдений для тренировочного набора, 20% наблюдений для рабочего набора и 20% наблюдений для тестового набора. Обычно эти виды разбиений указываются в следующей форме: 60/20/20, где первое число (60) обозначает процент от всего набора данных, который составляет тренировочный набор, второе (20) — процент от всего набора данных, который составляет рабочий набор, и последнее (20) — процент, который составляет тестовый набор. В книгах, блогах или статьях вы можете встретить такие высказывания, как "мы разделим набор данных в пропорции 80/10/10". Теперь вам будет понятно, что это значит.

В сфере глубокого обучения обычно вы будете работать с большими наборами данных. Например, если у нас $m = 10^6$, то мы могли бы использовать разбиение в пропорции 98/1/1. Учитывайте, что 1% от 10^6 равно 10^4, т. е. большому числу! Напомним, что рабочий/тестовый наборы должны быть достаточно большими с целью обеспечения высокой уверенности в результативности модели, но не излишне большими. Кроме того, вы наверняка захотите оставить как можно больше наблюдений для своего тренировочного набора.

ПРИМЕЧАНИЕ. Принимая решение о том, как разбить свой набор данных при наличии у вас большого числа наблюдений (например, 10^6 или даже больше), вы можете разбить набор данных в пропорции 98/1/1 либо 90/5/5. И после того как ваш рабочий и тестовый наборы данных достигнут разумного размера (в зависимости от вашей задачи), вы можете остановиться. При принятии решения о разбиении набора данных думайте о требуемой величине рабочего/тестового наборов.

Как вы уже знаете, размер это еще не всё. Рабочий и тестовый наборы данных должны быть репрезентативными для вашего тренировочного набора данных и за-

дачи в целом. Давайте создадим пример. Рассмотрим описанный ранее конкурс ImageNet. Там необходимо расклассифицировать изображения по 1000 разным классам. Для того чтобы узнать о качестве работы вашей модели на рабочем и тестовом наборах данных, вам потребуется достаточное число изображений по каждому классу в каждом наборе. Если вы решите взять для рабочего или тестового набора данных только 1000 наблюдений, то не получите никакого разумного результата, потому что если в рабочем наборе представлены все классы, то у вас будет только одно наблюдение на каждый класс. Вам следует принять другое решение: к примеру, создать свои рабочий и тестовый наборы данных, выбрав, по крайней мере, 100 изображений по каждому классу, построив два набора данных (рабочий и тестовый), по 10^5 наблюдений каждый (напомним, что у нас 1000 классов). В этом случае было бы неразумно ограничиваться числом ниже указанного. Эта тема актуальна не только в контексте глубокого обучения, но и в машинном обучении в целом. Вы всегда должны пытаться создавать рабочий/тестовый наборы данных, которые отражали бы то же самое распределение наблюдений, что и в вашем тренировочном наборе. Для лучшего понимания сути проблемы, в качестве примера возьмем набор данных MNIST. Давайте загрузим этот набор данных (как мы делали раньше) с помощью следующего фрагмента кода:

```
import numpy as np
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X,y = mnist["data"], mnist["target"]
total = 0
```

Затем можно проверить, как часто (в %) каждая цифра появляется в наборе данных.

```
for i in range(10):
    print ("цифра", i, "составляет",
           np.around(np.count_nonzero(y == i)/70000.0*100.0, decimals=1),
          "% из 70000 наблюдений")
```

В результате получим:

```
цифра 0 составляет 9.9 % из 70000 наблюдений
цифра 1 составляет 11.3 % из 70000 наблюдений
цифра 2 составляет 10.0 % из 70000 наблюдений
цифра 3 составляет 10.2 % из 70000 наблюдений
цифра 4 составляет 9.7 % из 70000 наблюдений
цифра 5 составляет 9.0 % из 70000 наблюдений
цифра 6 составляет 9.8 % из 70000 наблюдений
цифра 7 составляет 10.4 % из 70000 наблюдений
цифра 8 составляет 9.8 % из 70000 наблюдений
цифра 9 составляет 9.9 % из 70000 наблюдений
```

В этом наборе данных не каждая цифра появляетя одинаковое число раз. При создании рабочего и тестового наборов данных мы должны обеспечить, чтобы наши распределения отражали этот факт; в противном случае во время применения нашей модели к рабочему или тестовому набору данных можно получить результат,

который не имеет большого смысла, потому что модель обучилась на другом распределении классов. Как вы, возможно, помните, в *главе 5* мы создали рабочий набор данных с помощью следующих строк кода:

```
np.random.seed(42)
rnd = np.random.rand(len(y)) < 0.8

train_y = y[rnd]
dev_y = y[~rnd]
```

В данном случае для ясности разбиты только метки. Это сделано с целью увидеть работу алгоритма. В реальной жизни вам, разумеется, придется разбивать и признаки. Поскольку наше исходное распределение является почти однородным, вы должны ожидать результат, который очень похож на исходный. Давайте его проверим с помощью следующего фрагмента кода:

```
for i in range(10):
    print ("цифра", i, "составляет",
      np.around(np.count_nonzero(train_y == i)/56056.0*100.0, decimals=1),
           "% из 56056 наблюдений")
```

В результате получим:

```
цифра 0 составляет 9.9 % из 56056 наблюдений
цифра 1 составляет 11.3 % из 56056 наблюдений
цифра 2 составляет 9.9 % из 56056 наблюдений
цифра 3 составляет 10.1 % из 56056 наблюдений
цифра 4 составляет 9.8 % из 56056 наблюдений
цифра 5 составляет 9.0 % из 56056 наблюдений
цифра 6 составляет 9.8 % из 56056 наблюдений
цифра 7 составляет 10.4 % из 56056 наблюдений
цифра 8 составляет 9.8 % из 56056 наблюдений
цифра 9 составляет 9.9 % из 56056 наблюдений
```

Вы можете сравнить эти результаты с результатами всего набора данных и отметите, что они очень близкие, не одинаковые (сравните, например, цифру 2), но достаточно близкие. В этом случае можно без опасений продолжить работу. Но давайте приведем несколько иной пример. Предположим, что вместо случайного отбора наблюдений для создания тренировочного и рабочего наборов данных вы решаете взять первые 80% наблюдений и закрепить их за тренировочным набором, а последние 20% закрепить за рабочим набором, потому что вы исходите из допущения, что в исходных массивах NumPy ваши наблюдения имеют случайное распределение. Посмотрим, что получится. Прежде всего, давайте построим тренировочный и рабочий наборы данных, используя первые 56 000 (`0.8*70000`) наблюдений для тренировочного набора и остальные для рабочего набора.

```
srt = np.zeros_like(y, dtype=bool)

np.random.seed(42)
srt[0:56000] = True
```

```
train_y = y[srt]
dev_y = y[~srt]
```

Мы можем снова проверить число имеющихся цифр, используя следующий фрагмент кода:

```
total = 0
for i in range(10):
    print("класс", i, "составляет",
      np.around(np.count_nonzero(train_y == i)/56000.0*100.0, decimals=1),
      "% из 56000 наблюдений")
```

В результате получим:

```
класс 0 составляет 8.5 % из 56000 наблюдений
класс 1 составляет 9.6 % из 56000 наблюдений
класс 2 составляет 8.5 % из 56000 наблюдений
класс 3 составляет 8.8 % из 56000 наблюдений
класс 4 составляет 8.3 % из 56000 наблюдений
класс 5 составляет 7.7 % из 56000 наблюдений
класс 6 составляет 8.5 % из 56000 наблюдений
класс 7 составляет 9.0 % из 56000 наблюдений
класс 8 составляет 8.4 % из 56000 наблюдений
класс 9 составляет 2.8 % из 56000 наблюдений
```

Ничего не замечаете? Самая большая разница состоит в том, что теперь класс 9 появляется только в 2,8% случаев. Раньше он появлялся в 9,9% случаев. По-видимому, наша гипотеза о том, что классы распределены в соответствии со случайным равномерным распределением, оказалась ложной. Это может представлять довольно большую опасность во время проверки результативности модели, либо ваша модель может в конечном итоге обучиться на так называемом распределении несбалансированных классов.

ПРИМЕЧАНИЕ. Как правило, неравномерное распределение классов в наборе данных касается классификационной задачи, в которой число появлений одного или нескольких классов отличается от других. Обычно это становится проблемой в процессе заучивания, когда эта разница существенна. Разница в несколько процентов часто не является проблемой.

Если вы имеете набор данных с тремя классами, например такой, в котором у вас 1000 наблюдений в каждом классе, то этот набор данных имеет идеально сбалансированное распределение классов, но если в классе 1 только 100 наблюдений, в классе 2 — 10 000 наблюдений, а в классе 3 — 5000 наблюдений, то мы говорим о распределении несбалансированных классов. Не следует думать, что это редкое явление. Предположим, вам нужно построить модель, которая распознает мошеннические транзакции по кредитным картам. Можно с уверенностью предположить, что эти транзакции составляют очень маленький процент от всей совокупности транзакций, которые будут в вашем распоряжении.

ПРИМЕЧАНИЕ. Во время разбиения набора данных следует уделять большое внимание не только числу наблюдений в каждом наборе данных, но и тому, какие наблюдения поступают в каждый набор данных. Обратите внимание, что эта проблема не является специфичной лишь для глубокого обучения, но представляет важность для всего машинного обучения в целом.

Детальное рассмотрение вопроса способов работы с несбалансированными наборами данных выходит за рамки этой книги, но важно осознавать последствия, которые они могут иметь. В следующем разделе будет продемонстрировано, что происходит, если в нейронную сеть ввести несбалансированный набор данных. Это даст вам конкретное понимание последствий такой возможности. В конце раздела будет дано несколько советов по поводу того, что делать в таком случае.

Распределение несбалансированных классов: что может произойти

Поскольку мы говорим о том, как разбить набор данных для выполнения метрического анализа, важно разбираться в проблеме распределения несбалансированных классов и в том, как с ней бороться. В глубоком обучении очень часто возникают ситуации, когда приходится разбивать наборы данных на части, и вы обязаны знать проблемы, с которыми можете столкнуться, если сделаете это неправильно. Приведем конкретный пример, показывающий, как плохо все может пойти, если сделать это неправильно.

Мы будем использовать набор данных MNIST и выполним элементарную логистическую регрессию (как в *главе 2*) с одним-единственным нейроном. Давайте еще раз очень бегло просмотрим, как загружать и подготавливать данные. Мы сделаем это так же, как в *главе 2*, за исключением приведенных ниже нескольких изменений. Сначала мы загружаем данные:

```
import numpy as np
from sklearn.datasets import fetch_mldata
from sklearn.metrics import confusion_matrix
import tensorflow as tf

mnist = fetch_mldata('MNIST original')
Xinput,yinput = mnist["data"], mnist["target"]
```

А вот и важная часть. Мы создаем новую метку следующим образом: назначаем всем наблюдениям с цифрой ноль метку 0, а всем остальным цифрам (1, 2, 3 ,4, 5, 6, 7, 8 и 9) метку 1, используя вот этот фрагмент кода:

```
y_ = np.zeros_like(yinput)
y_[np.any([yinput == 0], axis = 0)] = 0
y_[np.any([yinput > 0], axis = 0)] = 1
```

Теперь массив y_ будет содержать новые метки. Обратите внимание, что теперь набор данных сильно несбалансирован. Метка 0 появляется примерно в 10% случаев, в то время как метка 1 появляется в 90% случаев. Давайте разобьем данные случайно на тренировочный и рабочий наборы данных.

```
np.random.seed(42)
rnd = np.random.rand(len(y_)) < 0.8

X_train = Xinput[rnd,:]
y_train = y_[rnd]
X_dev = Xinput[~rnd,:]
y_dev = y_[~rnd]
```

Далее выполним нормализацию тренировочных данных.

```
X_train_normalised = X_train/255.0
```

И затем транспонируем и подготовим тензоры.

```
X_train_tr = X_train_normalised.transpose()
y_train_tr = y_train.reshape(1,y_train.shape[0])
```

Назначим собственные имена переменным.

```
Xtrain = X_train_tr
ytrain = y_train_tr
```

И построим сеть с одним-единственным нейроном, точно так же, как в *главе 2*.

```
tf.reset_default_graph()
X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n_dim]))
b = tf.Variable(tf.zeros(1))

init = tf.global_variables_initializer()y_ = tf.sigmoid(tf.matmul(W,X)+b)
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Если вам не ясен этот фрагмент кода, то для получения более подробной информации пересмотрите *главу 2*. Надо надеяться, что теперь вы хорошо понимаете эту простую модель, т. к. мы с ней встречались уже несколько раз. Затем мы определяем функцию для выполнения модели (вы видели ее несколько раз в предыдущих главах).

```
def run_logistic_model(learning_r, training_epochs, train_obs,
                       train_labels, debug = False):
    sess = tf.Session()
    sess.run(init)
```

```
    cost_history = np.empty(shape=[0], dtype = float)

    for epoch in range(training_epochs+1):
        sess.run(training_step, feed_dict = {X:train_obs, Y:train_labels,
                                             learning_rate: learning_r})
        cost_ = sess.run(cost, feed_dict={X:train_obs, Y:train_labels,
                                          learning_rate: learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % 10 == 0) & debug:
            print("Достигнута эпоха",epoch,"стоимость J =",
                  str.format('{0:.6f}', cost_))

    return sess, cost_history
```

Давайте выполним модель с помощью следующего вызова функции:

```
sess, cost_history = run_logistic_model(learning_r=0.01,
                                        training_epochs=100,
                                        train_obs=Xtrain,
                                        train_labels=ytrain,
                                        debug=True)
```

и проверим ее точность с помощью следующего фрагмента кода (подробно объясненного в *главе 2*):

```
correct_prediction=tf.equal(tf.greater(y_, 0.5), tf.equal(Y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={X:Xtrain, Y:ytrain, learning_rate:0.05}))
```

Мы получим невероятную точность 91,2%. Неплохо, правда? Но уверены ли мы, что результат настолько хорош? Теперь давайте проверим матрицу несоответствий[2] для наших меток, используя для этого вот этот фрагмент кода:

```
ypred = sess.run(tf.greater(y_, 0.5),
                 feed_dict={X:Xtrain, Y:ytrain,
                            learning_rate: 0.05}).flatten().astype(int)
confusion_matrix(ytrain.flatten(), ypred)
```

После его выполнения вы получите:

```
array([[ 659,  4888],
       [   6, 50503]], dtype=int64)
```

Чуть более красиво отформатированная и с небольшой пояснительной информацией матрица несоответствий выглядит как табл. 6.1.

[2] По классификации машинного обучения матрица несоответствий (confusion matrix), или матрица ошибок, — это матрица, в которой каждый столбец представляет число экземпляров в предсказанном классе, а каждая строка — число экземпляров в фактическом классе.

Таблица 6.1. *Матрица несоответствий для модели, описанной в тексте*

	Предсказанный класс 0	**Предсказанный класс 1**
Реальный класс 0	659	4888
Реальный класс 1	6	50 503

Как читать эту таблицу? В столбце "Предсказанный класс 0" вы увидите число наблюдений, которое модель предсказывает как класс 0 по каждому реальному классу. Число 659 — это количество наблюдений, которые модель предсказывает как класс 0 и которые действительно принадлежат классу 0. Число 6 — это количество наблюдений, которые модель предсказывает как класс 0, но которые в действительности принадлежат классу 1.

Теперь несложно увидеть: наша модель фактически предсказывает, что почти все наблюдения принадлежат классу 1 (в общей сложности $4888 + 50\,503 = 55\,391$). Число правильно классифицированных наблюдений 659 (для класса 0) и 50 503 (для 1 класса), всего 51 162 наблюдений. Поскольку тренировочный набор всего насчитывает 56 056 наблюдений, мы получаем точность $51\,162 / 56\,056 = 0{,}912$, как и показал приведенный выше фрагмент кода TensorFlow. Этот показатель вовсе не означает высокую результативность модели; он просто означает, что она фактически отнесла все наблюдения к классу 1. В данном случае для достижения этой точности нам не нужна нейронная сеть. Произошло то, что модель крайне редко видит наблюдения, относящиеся к классу 0, и этот факт почти не влияет на ее самообучение, над которым доминируют наблюдения, принадлежащие классу 1.

То, что вначале казалось хорошим результатом, оказывается очень плохим. Данный пример наглядно показывает, как плохо все может пойти, если не обращать внимания на распределение классов. Разумеется, данная проблема возникает не только, когда вы разбиваете набор данных на части, но и в целом, когда вы приступаете к классификационной задаче, независимо от классификатора, который вы хотите натренировать (это касается не только нейронных сетей).

ПРИМЕЧАНИЕ. Во время разбиения набора данных на части в сложных задачах особое внимание следует обращать не только на число наблюдений в наборах данных, но и на то, какие наблюдения вы отбираете, а также на распределение классов.

В завершение этого раздела приведем несколько советов по работе с несбалансированными наборами данных.

- *Изменить метрику.* В предыдущем примере вместо точности вы можете применить нечто другое, потому что она может ввести в заблуждение. В частности, можно попробовать матрицу несоответствий либо другие метрики, такие как прецизионность, полнота или оценка F1. Еще один важный способ проверить результативность модели, и настоятельно рекомендуется его изучить, — это кривая ROC, которая вам очень поможет.

- *Работать с пониженной выборкой данных.* Если, например, у вас 1000 наблюдений из класса 1 и 100 — из класса 2, можно создать новый набор данных со 100 случайными наблюдениями из класса 1 и 100 наблюдениями из класса 2. Проблема этого метода, однако, в том, что обычно в вашу модель для ее тренировки будет подаваться намного меньше данных.
- *Работать с повышенной выборкой данных.* Можно попробовать сделать наоборот. Вы можете взять 100 наблюдений из упомянутого выше класса 2 и просто реплицировать их 10 раз подряд, в итоге получив 1000 наблюдений из класса 2 (эта процедура иногда называется отбором образцов с возвратом, или бутстрапом).
- *Попробовать получить больше данных из класса с меньшим числом наблюдений*: это не всегда возможно. В случае мошеннических операций с кредитными картами вы не сможете пойти по кругу и сгенерировать новые данные, если только не хотите попасть в тюрьму...

Метрики прецизионности, полноты и F1

Давайте обратимся к нескольким другим метрическим показателям, которые очень полезны при работе с несбалансированными наборами данных. Рассмотрим следующий пример. Предположим, мы проводим некоторые тесты с целью определить, есть ли у испытуемого конкретное заболевание или нет. Представим, что у нас имеется 250 результатов тестов. Рассмотрим следующую матрицу несоответствий (вы должны знать, что она представляет, из предыдущего раздела):

	Предсказание: нет	**Предсказание: да**
Истинное значение: нет	75	15
Истинное значение: да	10	150

Обозначим через N суммарное число результатов тестов, в данном случае $N = 250$. Мы будем использовать следующую терминологию[3]:

- истинноположительные (*tp*): тесты, которые предсказали "да", и у испытуемых действительно есть заболевание;
- истинноотрицательные (*tn*): тесты, которые предсказали "нет", и у испытуемых действительно нет заболевания;
- ложноположительные (*fp*): тесты, которые предсказали "да", но у испытуемых в действительности нет заболевания;

[3] Аббревиатура tp соответствует термину true positives, tn — true negatives, fp — false positives и fn — false negatives. — *Прим. пер.*

- ложноотрицательные (*fn*): тесты, которые предсказали "нет", но у испытуемых в действительности есть заболевание.

Это визуально транслируется в следующее:

	Предсказание: нет	**Предсказание: да**
Истинное значение: нет	Истинноотрицательные	Ложноположительные
Истинное значение: да	Ложноотрицательные	Истинноположительные

Давайте также обозначим через *ty* число пациентов, у которых действительно есть заболевание, в этом примере $ty = 10 + 150 = 160$, а через *tno* — число пациентов, у которых нет заболевания, в этом примере $tno = 75 + 15 = 90$. В этих примерах мы бы имели:

$$tp = 150;$$
$$tn = 75;$$
$$fp = 15;$$
$$fn = 10.$$

Несколько метрик можно выразить как функции от ранее рассмотренных терминов. Например:

- точность: $(tp + tn)/N$ — как часто тест является правильным;
- коэффициент неправильного классифицирования: $(fp + fn)/N$ — как часто тест является ошибочным. Обратите внимание, что этот показатель равен 1 – точность;
- чувствительность/полнота: tp/ty — как часто тест действительно предсказывает "да", когда у испытуемых есть заболевание;
- специфичность: tn/tno — как часто наш тест предсказывает "нет", когда у испытуемых нет заболевания;
- прецизионность[4]: $tp/(tp + fp)$ — порция тестов, правильно предсказывающих, что у испытуемого есть заболевание, по всем полученным утвердительным результатам.

В зависимости от вашей задачи все эти величины могут использоваться в качестве метрик. Давайте рассмотрим пример. Предположим, ваш тест должен предсказать,

---

[4] В машинном обучении прецизионность (precision, точность результатов измерений) является мерой статистической изменчивости и описывает случайные ошибки, а также описывает степень близости друг к другу независимых результатов измерений, полученных в конкретных установленных условиях. Точность модели/алгоритма (accuracy) — это мера статистического смещения и описывает систематические ошибки; иными словами, это близость результатов измерений истинному значению. См. **https://en.wikipedia.org/wiki/Accuracy_and_precision**. — *Прим. пер.*

есть ли у человека онкологическое заболевание или нет. В этом случае вы хотите иметь максимально высокую чувствительность, потому что крайне важно обнаружить заболевание. Но в то же время вы хотите, чтобы специфичность была высокой, потому что нет ничего хуже, чем отправить кого-то домой без лечения, когда оно необходимо.

Давайте взглянем поближе на прецизионность и полноту. Высокая прецизионность означает, что когда вы говорите, что кто-то болен, вы правы. Но вы не знаете, сколько людей действительно болеют, т. к. эта величина определяется лишь по результатам вашего теста. Прецизионность является мерой того, насколько качественно работает ваш тест. Наличие высокой полноты означает, что вы можете идентифицировать всех больных людей в своей выборке. Для того чтобы прояснить ситуацию, приведем еще один пример. Предположим, у нас 1000 человек. Только 10 из них больны, а все остальные 990 здоровы. Предположим, мы хотим определить здоровых людей (это важно), и мы строим тест, который возвращает "да", если кто-то здоров, и **всегда** предсказывает, что люди являются здоровыми. Матрица несоответствий будет выглядеть так:

	Предсказание: нет (болен)	**Предсказание: да (здоров)**
Истина: нет (болен)	0	10
Истина: да (здоров)	0	990

Мы бы имели:

$$tp = 990;$$
$$tn = 0;$$
$$fp = 10;$$
$$fn = 0.$$

Это означает, что:

- точность составит 99%;
- коэффициент неправильного классифицирования составит 10/1000, или, другими словами, 1%;
- полнота составит 990/990, или 100%;
- специфичность составит 0%;
- прецизионность составит 99%.

Выглядит неплохо, правда? Если вы хотите найти здоровых людей, то этот тест будет отличным. Единственная проблема в том, что гораздо важнее идентифицировать больных людей! Давайте пересчитаем предыдущие величины, но на этот раз учитывая, что утвердительный результат означает, что кто-то болен. В этом случае матрица несоответствий выглядела бы так:

	Предсказание: нет (здоров)	Предсказание: да (болен)
Истина: нет (здоров)	990	0
Истина: да (болен)	10	0

Потому что на этот раз результат "да" означает, что кто-то болен, а не, как раньше, что кто-то здоров. Давайте снова рассчитаем приведенные выше величины.

$$tp = 0;$$
$$tn = 990;$$
$$fp = 0;$$
$$fn = 10.$$

Следовательно,

- точность составит 99%;
- коэффициент неправильного классифицирования составит 10/1000, или, другими словами, 1%;
- полнота составит 0/10, или 0%;
- специфичность составит 990/990, или 100%;
- прецизионность составит $(0 + 0)/100$, или 0%.

Обратите внимание, что точность остается прежней! Если смотреть только на нее, то вы не сможете понять, насколько хорошо ваша модель работает. Мы просто изменили то, что хотим предсказать, и используем лишь точность. Мы ничего не можем сказать о результативности модели. Но посмотрите, как изменились полнота и прецизионность. Для сравнения взгляните на приведенную ниже матрицу:

	Предсказание, что человек здоров	Предсказание, что человек болен
Полнота	100%	0%
Прецизионность	99%	0%

Теперь у нас есть то, что меняется, давая нам возможность получить достаточную информацию в зависимости от поставленного нами вопроса. Обратите внимание, что изменив объект предсказания, мы меняем внешний вид матрицы несоответствий. Глядя на приведенную выше матрицу, можно сразу сказать, что модель, которая предсказывает, что все здоровы, имеет очень высокую результативность, когда она предсказывает здоровых людей (при этом ее полезность не очень высока), но оказывается безуспешной, когда пытается предсказать больных людей.

Существует еще одна метрика, о которой важно знать, и это оценка F1. Она определяется как

$$F1 = \frac{2}{\frac{1}{\text{прецизионность}} + \frac{1}{\text{полнота}}} = 2 \cdot \frac{\text{прецизионность} \cdot \text{полнота}}{\text{прецизионность} + \text{полнота}}.$$

Ее трудно понять интуитивно, но в сущности она представляет собой гармоническое среднее прецизионности и полноты. Созданный нами пример был немного экстремальным, и наличие 0%-й полноты или прецизионности не позволило бы вычислить F1. Давайте предположим, что наша модель плохо предсказывает больных людей, но не настолько плохо. Будем считать, что у нас имеется следующая матрица несоответствий:

	Предсказание: нет (здоров)	**Предсказание: да (болен)**
Истина: нет (здоров)	985	5
Истина: да (болен)	9	1

В этом случае мы имели бы (расчеты оставлены в качестве упражнения):

- прецизионность — 54,5%;
- полнота — 10%

и

$$F1 = 2 \cdot \frac{0,545 \cdot 0,1}{0,545 + 0,1} = 2 \cdot \frac{0,0545}{0,645} = 0,169 \rightarrow 16,9\%.$$

Эта величина даст вам информацию с учетом прецизионности (порция тестов, правильно предсказывающих, что у испытуемого есть заболевание по всем полученным утвердительным результатам) и полноты (как часто тест предсказывает "да", когда у обследуемых действительно есть болезнь). В некоторых задачах вы хотите максимизировать прецизионность, в других же — полноту. Если это так, то просто выберите нужную метрику. Оценка F1 будет одинаковой для двух случаев, где в одном у вас прецизионность равна 32% и полнота — 45%, а в другом прецизионность равна 45% и полнота равна 32%. Учитывайте это и используйте оценку F1, если хотите найти баланс между прецизионностью и полнотой.

ПРИМЕЧАНИЕ. Оценка F1 используется, когда требуется максимизировать гармоническое среднее прецизионности и полноты, или, другими словами, когда вы не хотите максимизировать только прецизионность либо только полноту, но желаете найти лучший баланс между ними.

Если бы мы рассчитали F1 во время предсказания здоровых людей, как в самом начале, то имели бы:

$$F1 = 2 \cdot \frac{1,0 \cdot 0,99}{1,0 + 0,99} = 2 \cdot \frac{0,99}{1,99} = 0,995 \rightarrow 99,5\%.$$

Эта оценка говорит о том, что модель довольно хорошо предсказывает здоровых людей.

Полезность оценки F1 состоит в том, что, как правило, в качестве метрики вы хотите одно-единственное число, и вам не нужно выбирать между прецизионностью или полнотой, поскольку обе полезны. Помните, что значения обсуждаемых метрик всегда будут зависеть от вопроса, который вы задаете (чем для вас являются ответ "да" и ответ "нет"). Интерпретация всегда зависит от вопроса, на который вы хотите дать ответ.

ПРИМЕЧАНИЕ. При вычислении метрики, какой бы она ни была, переформулировка вашего вопроса изменит результаты. С самого начала вы должны ясно понимать, что хотите предсказать, а затем выбрать правильную метрику. В случае сильно несбалансированных наборов данных всегда полезно использовать не точность, а другие метрики, такие как полнота, прецизионность или, даже более качественную оценку F1, т. е. среднее значение прецизионности и полноты.

Наборы данных с разными распределениями

Теперь следует обсудить еще один терминологический вопрос, который приведет вас к пониманию общей проблемы в сфере глубокого обучения. Очень часто вы будете слышать такие высказывания, как "множества происходят из разных распределений". Это высказывание не всегда легко понять. Возьмем, к примеру, два набора данных, сформированных из снимков, сделанных с помощью профессионального зеркального фотоаппарата класса DSLR[5], и второй набор, составленный из снимков, сделанных с помощью хитроумного смартфона. В сфере глубокого обучения мы бы охарактеризовали эти два набора как происходящие из разных распределений. Но каково истинное значение этого высказывания? Два набора данных различаются по разным причинам: разрешающая способность изображений, размытость из-за разнокачественных линз, число цветов в палитре изображения, качество фокуса и, возможно, больше. Все эти различия являются именно теми, что обычно подразумеваются под распределениями. Рассмотрим еще один пример. Мы могли бы взять два набора данных: один со снимками белых кошек и другой со снимками черных кошек. В этом случае мы тоже ведем речь о разных распределениях. Проблема распределения данных возникает, когда вы тренируете модель на одном наборе и хотите применить ее к другому. Например, если вы тренируете модель на наборе снимков белых кошек, то вы, вероятно, не очень хорошо справитесь с набором снимков черных кошек, потому что во время тренировки ваша модель не встречала ни одной черной кошки.

[5] Цифровой однообъективный зеркальный фотоаппарат (digital single-lens reflex, DSLR). — *Прим. пер.*

ПРИМЕЧАНИЕ. Когда речь идет о наборах данных, поступающих из разных распределений, обычно подразумевается, что в двух наборах данных наблюдения имеют разные характеристики: черные и белые кошки, изображения с высокой и низкой разрешающей способностью, речь, записанная на итальянском и немецком языках, и т. д.

Поскольку данные чрезвычайно ценны, люди часто пытаются создавать разные наборы данных (тренировочные, рабочие и т. д.) из разных источников. Например, вы можете решить натренировать свою модель на наборе изображений, взятых из Интернета, и проверить ее результативность на наборе снимков, сделанных с помощью смартфона. На первый взгляд, использовать как можно больше данных — идея и неплохая, но она вполне может стать причиной для головной боли. Давайте посмотрим, что происходит в реальности, чтобы вы прочувствовали последствия чего-то подобного.

Рассмотрим подмножество набора данных MNIST, которое мы использовали в *главе 2*, состоящее из двух цифр: 1 и 2. Мы построим рабочий набор данных из другого распределения, сдвинув подмножество изображений на 10 пикселов вправо. Мы натренируем модель на изображениях в том виде, какие они есть в исходном наборе данных, применим модель к изображениям, сдвинутым на 10 пикселов вправо, и посмотрим, что произойдет. Сначала загрузим данные (вы можете обратиться к более подробной информации в *главе 2*).

```
import numpy as np
from sklearn.datasets import fetch_mldata
%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt
from random import *

mnist = fetch_mldata('MNIST original')
Xinput,yinput = mnist["data"], mnist["target"]
```

Мы подготовим данные точно так же, как в *главе 2*. Прежде всего, давайте выберем только цифры 1 и 2.

```
X_ = Xinput[np.any([y == 1,y == 2], axis = 0)]
y_ = yinput[np.any([y == 1,y == 2], axis = 0)]
```

В наборе данных 14 867 наблюдений. Теперь создадим тренировочный и рабочий наборы данных, случайно отобрав образцы (как и раньше), т. к. в этом случае у нас получится примерно одинаковое число единиц и двоек.

```
np.random.seed(42)
rnd_train = np.random.rand(len(y_)) < 0.8
X_train = X_[rnd_train,:]
y_train = y_[rnd_train]
X_dev = X_[~rnd_train,:]
y_dev = y_[~rnd_train]
```

Далее мы выполним нормализацию признаков.

```
X_train_normalized = X_train/255.0
X_dev_normalized = X_dev/255.0
```

И затем преобразуем матрицы так, чтобы они имели правильные размерности.

```
X_train_tr = X_train_normalized.transpose()
y_train_tr = y_train.reshape(1,y_train.shape[0])
n_dim = X_train_tr.shape[0]
dim_train = X_train_tr.shape[1]
X_dev_tr = X_dev_normalized.transpose()
y_dev_tr = y_dev.reshape(1,y_dev.shape[0])
```

Наконец, мы сдвинем метки, чтобы они были равны 0 и 1. (Если вы не помните причины этой операции, то можете бегло просмотреть *главу 2*.)

```
y_train_shifted = y_train_tr - 1
y_dev_shifted = y_dev_tr - 1
```

Теперь дадим массивам осмысленные имена.

```
Xtrain = X_train_tr
ytrain = y_train_shifted

Xdev = X_dev_tr
ydev = y_dev_shifted
```

Мы можем проверить размерности массивов с помощью следующих инструкций:

```
print(Xtrain.shape)
print(Xdev.shape)
```

В результате получим:

```
(784, 11893)
(784, 2974)
```

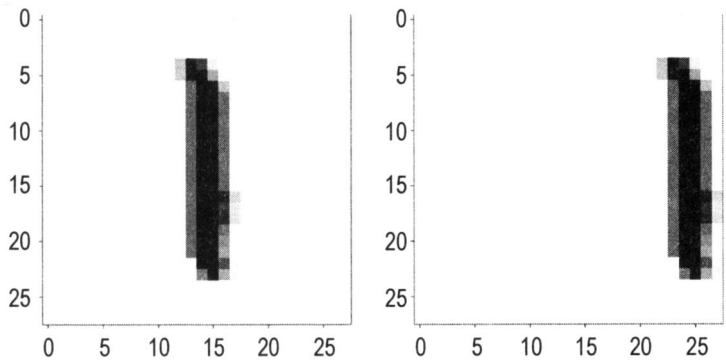
В тренировочном наборе у нас 11 893 наблюдения, тогда как в рабочем наборе — 2974 наблюдения. Теперь давайте продублируем рабочий набор данных и сдвинем каждое изображение вправо на 10 пикселов. Это делается довольно быстро с помощью следующего фрагмента кода:

```
Xtraindev = np.zeros_like(Xdev)
for i in range(Xdev.shape[1]):
    tmp = Xdev[:,i].reshape(28,28)
    tmp_shifted = np.zeros_like(tmp)
    tmp_shifted[:,10:28] = tmp[:,0:18]
    Xtraindev[:,i] = tmp_shifted.reshape(784)

ytraindev = ydev
```

Для облегчения сдвига сначала в матрице 28×28 были реформированы изображения, потом инструкцией `tmp_shifted[:,10:28] = tmp[:,0:18]` были сдвинуты столб-

цы, а затем изображения были реформированы в одномерный массив из 784 элементов. Метки остаются теми же. На рис. 6.7, *а* вы видите случайное изображение из рабочего набора данных, а на рис. 6.7, *б* — его смещенную версию.



а *б*

РИС. 6.7. Одно случайное изображение из набора данных (*а*) и его сдвинутая версия (*б*)

Теперь построим сеть с одним-единственным нейроном и посмотрим, что произойдет. Мы строим модель, как в *главе 2*.

```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n_dim]))
b = tf.Variable(tf.zeros(1))
init = tf.global_variables_initializer()
y_ = tf.sigmoid(tf.matmul(W,X)+b)
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Для тренировки модели мы будем использовать ту же функцию, которую вы уже встречали:

```
def run_logistic_model(learning_r, training_epochs, train_obs,
                       train_labels, debug = False):
    sess = tf.Session()
    sess.run(init)

    cost_history = np.empty(shape=[0], dtype = float)

    for epoch in range(training_epochs+1):
        sess.run(training_step, feed_dict = {X:train_obs, Y:train_labels,
                                             learning_rate: learning_r})
```

```
        cost_ = sess.run(cost, feed_dict={X:train_obs, Y:train_labels,
                                          learning_rate:learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % 10 == 0) & debug:
            print("Достигнута эпоха",epoch,"стоимость J =",
                  str.format('{0:.6f}',cost_))

    return sess, cost_history
```

и натренируем модель с помощью следующего фрагмента кода:

```
sess, cost_history = run_logistic_model(learning_r=0.01,
                                        training_epochs=100,
                                        train_obs=Xtrain,
                                        train_labels=ytrain,
                                        debug=True)
```

В результате получим:

```
Достигнута эпоха 0 стоимость J = 0.678501
Достигнута эпоха 10 стоимость J = 0.562412
Достигнута эпоха 20 стоимость J = 0.482372
Достигнута эпоха 30 стоимость J = 0.424058
Достигнута эпоха 40 стоимость J = 0.380005
Достигнута эпоха 50 стоимость J = 0.345703
Достигнута эпоха 60 стоимость J = 0.318287
Достигнута эпоха 70 стоимость J = 0.295878
Достигнута эпоха 80 стоимость J = 0.277208
Достигнута эпоха 90 стоимость J = 0.261400
Достигнута эпоха 100 стоимость J = 0.247827
```

Затем рассчитаем точность трех наборов данных: тренировочного `Xtrain`, рабочего `Xdev` и тренировочно-рабочего `Xtraindev` с помощью фрагмента кода

```
correct_prediction=tf.equal(tf.greater(y_, 0.5), tf.equal(Y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(sess.run(accuracy, feed_dict={X:Xtrain, Y:ytrain, learning_rate:0.05}))
```

используя соответствующий словарь `feed_dict` для этих трех наборов данных. После 100 итераций мы получили следующие результаты:

- для тренировочного набора данных мы получаем 96,8%;
- для рабочего набора данных мы получаем 96,7%;
- для тренировочно-рабочего (позже вы увидите, почему он так называется), т. е. со сдвинутыми изображениями, мы получаем 46,7%. Очень плохой результат.

Произошло то, что модель обучилась на наборе данных, в котором все изображения центрированы в прямоугольнике, и поэтому не смогла обобщить свои знания на смещенные и больше не центрированные изображения.

Во время тренировки модели на тренировочном наборе данных обычно получают хорошие результаты для наблюдений, которые похожи на наблюдения из этого набора данных. Но как узнать, что у нас проблема именно в этом? Для этого существует относительно простой способ: расширить диаграмму MAD. Посмотрим, как это сделать.

Предположим, у вас есть тренировочный и рабочий наборы данных, в которых наблюдения имеют разные характеристики (поступают из разных распределений). Вы создаете небольшое подмножество тренировочного набора данных, именуемое тренировочно-рабочим, в итоге получая три набора данных: тренировочный и тренировочно-рабочий из одного и того же распределения (наблюдения имеют одинаковые характеристики), а также рабочий набор, в котором наблюдения как-то различаются, как уже отмечалось ранее. Теперь вы тренируете свою модель на тренировочном наборе, а затем оцениваете свою ошибку ε на трех наборах данных: $\varepsilon_{\text{тренировка}}$, $\varepsilon_{\text{разработка}}$ и $\varepsilon_{\text{тренировка-разработка}}$. Если ваши тренировочный и рабочий наборы происходят из одних и тех же распределений, то и тренировочно-рабочий набор тоже. В этом случае следует ожидать $\varepsilon_{\text{разработка}} \approx \varepsilon_{\text{тренировка-разработка}}$. Если мы определим

$$\Delta\varepsilon_{\text{тренировка-разработка}} = \varepsilon_{\text{тренировка-разработка}},$$

то у нас ожидаемо должно быть $\Delta\varepsilon_{\text{тренировка-разработка}} \approx 0$. Если же тренировочный (и тренировочно-рабочий) и рабочий наборы происходят из разных распределений (наблюдения имеют разные характеристики), то у нас ожидаемо $\Delta\varepsilon_{\text{тренировка-разработка}}$ должна быть большой. Если взять пример MNIST, который мы создали раньше, то по факту у нас $\Delta\varepsilon_{\text{тренировка-разработка}} = 0{,}437$, или 43,7%, что является огромной разницей. Давайте вспомним, что следует сделать для того, чтобы определить, есть ли у ваших тренировочного и рабочего (или тестового) наборов данных наблюдения с разными характеристиками (из разных распределений).

1. Разбить тренировочный набор на две части — одну, которую вы будете использовать для тренировки и назовете ее тренировочным набором, и меньшую часть, которую вы назовете тренировочно-рабочим набором.
2. Натренировать модель на тренировочном наборе.
3. Оценить ошибку ε на трех наборах: тренировочном, рабочем и тренировочно-рабочем.
4. Рассчитать величину $\Delta\varepsilon_{\text{тренировка-разработка}}$. Если она является большой, то это даст убедительные доказательства того, что исходные тренировочный и рабочий наборы происходят из разных распределений.

На рис. 6.8 представлен пример диаграммы MAD с только что рассмотренной дополнительной проблемой. Не смотрите на цифры, они приведены только для иллюстрации (читай: просто чтобы было).

Диаграмма MAD на рис. 6.8 сообщит следующее. (В маркированном списке выделены лишь несколько пунктов. Для получения более полного списка просмотрите предыдущие разделы.)

- Смещение (между результативностью после тренировки и человеческой результативностью) является довольно малым, поэтому мы находимся не так далеко от лучшего, которое можем достичь (будем считать, что здесь человеческая результативность является индикатором байесовой ошибки). Здесь вы можете попробовать более крупные сети, более качественные оптимизаторы и т. д.
- Мы излишне плотно аппроксимируем наборы данных, поэтому можем попробовать регуляризацию либо получить больше данных.
- У нас явная проблема с несоответствием данных (наборы из разных распределений) между тренировочным и рабочим наборами. В конце этого раздела будут предложены меры, которые помогут решить данную проблему.
- У нас также имеется небольшая переподгонка к рабочему набору данных во время гиперпараметрического поиска.

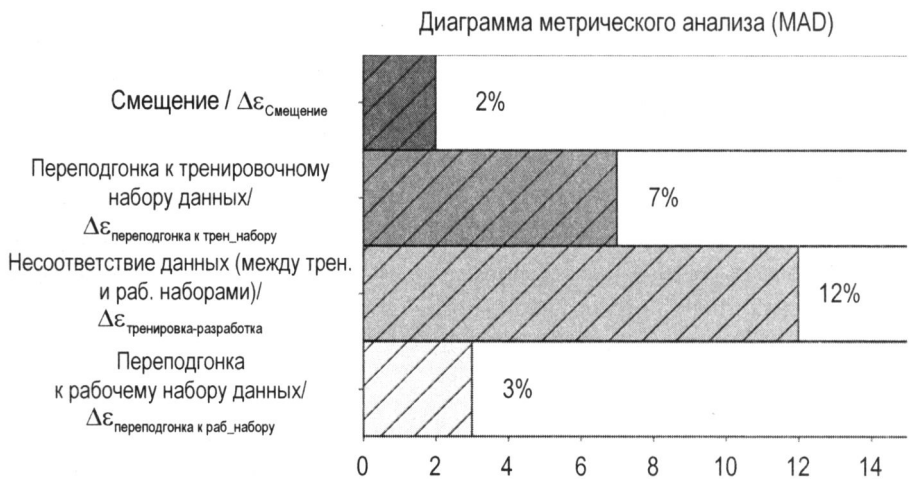


РИС. 6.8. Пример диаграммы MAD с добавленной проблемой несоответствия данных. Не смотрите на цифры, они приведены исключительно для иллюстрации

Обратите внимание, что вам не нужно создавать гистограмму, как было сделано здесь. Вообще, для того чтобы сделать те же самые выводы, вам нужны только четыре числа.

ПРИМЕЧАНИЕ. Интерпретация диаграммы MAD (либо просто чисел), которая имеется в вашем распоряжении, даст вам подсказки о том, что необходимо попытаться сделать, чтобы получить более высокие результаты, например более высокую точность.

Для устранения несоответствия между наборами данных можно попробовать следующие технические решения.

- Можно провести ручной анализ ошибок с целью понять разницу между наборами, а затем решить, что делать (в последнем разделе главы будет приведен при-

мер). Это техническое решение отнимает много времени и обычно довольно трудновыполнимо, потому что при известной разнице может оказаться очень трудно найти решение.

- Можно попытаться сделать тренировочный набор более похожим на рабочий/тестовый наборы. Например, если вы работаете с изображениями, и рабочий/тестовый наборы имеют более низкую разрешающую способность, то можно попробовать снизить разрешающую способность изображений в тренировочном наборе.

Как обычно, никаких фиксированных правил нет. Просто имейте в виду эту проблему и думайте о следующем: ваша модель будет обучаться характеристикам на тренировочных данных, поэтому при ее применении к совершенно другим данным показать хорошую результативность ей (обычно) не получится. Всегда старайтесь получать тренировочные данные, которые бы отражали именно те данные, с которыми ваша модель должна работать, а не наоборот.

K-блочная перекрестная проверка

Теперь в завершение этой главы рассмотрим еще одно техническое решение, которое является очень мощным и должно быть известно на практике любому специалисту в сфере машинного обучения (не только в сфере глубокого обучения): *k*-блочная перекрестная проверка. Данное техническое решение представляет собой способ нахождения решения следующих двух задач.

- Что делать, если набор данных является слишком малым для его разбиения на тренировочный и рабочий/тестовый наборы?
- Как получить информацию о дисперсии метрического показателя?

Давайте опишем ее идею с помощью псевдокода.

1. Разбить полный набор данных на k одинаковых по размеру подмножеств: f_1, f_2, ..., f_k. Эти подмножества также называются блоками. Обычно эти подмножества друг на друга не накладываются, иными словами, каждое наблюдение появляется в одном и только одном блоке.
2. Для i от 1 до k:
   - натренировать модель на всех блоках, кроме f_i;
   - оценить метрику на блоке f_i; блок f_i будет рабочим набором на итерации i.
3. Оценить среднее значение и дисперсию метрики на k результатах.

Типичное значение k равно 10, но это зависит от размера набора данных и характеристики задачи.

Стоит вспомнить, что обсуждавшиеся приемы разбивки набора данных также применимы и здесь.

ПРИМЕЧАНИЕ. При создании блоков вы должны заботиться о том, чтобы они отражали структуру исходного набора данных. Например, если исходный набор данных содержит 10 классов, то вы должны обеспечить, чтобы каждый блок содержал все 10 классов в одинаковых пропорциях.

Хотя это техническое решение может показаться весьма привлекательным для работы с наборами, чьи размеры не дотягивают до оптимального, его реализация может оказаться довольно сложной. Но, как вы вскоре увидите, проверка метрики на разных блоках даст вам важную информацию о возможной переподгонке модели к тренировочному набору данных.

Давайте попробуем это решение на реальном наборе данных и посмотрим, как его реализовать. Обратите внимание, что вы можете легко реализовать *k*-блочную перекрестную проверку с помощью библиотеки Scikit-learn, но здесь она будет разработана с нуля с целью показать, что происходит за кулисами. Любой (почти любой) может скопировать исходный код из Интернета и реализовать *k*-блочную перекрестную проверку с использованием Scikit-learn, но немногие могут объяснить, как она работает, либо в ней разбираются и, следовательно, способны выбрать правильный метод или параметры Scikit-learn. В качестве набора данных мы будем использовать тот же, что и в *главе 2*: сокращенный набор данных MNIST, содержащий только цифры 1 и 2. Мы выполним простую логистическую регрессию с одним-единственным нейроном для того, чтобы упростить понимание исходного кода и позволить нам сосредоточиться на той части, которая касается перекрестной проверки, а не на других деталях реализации, которые здесь не актуальны. Цель этого раздела — дать вам понять, как работает *k*-блочная перекрестная проверка и в чем ее польза, а не как реализовать ее с наименьшим числом строк кода.

Как обычно, давайте начнем с импортирования необходимых библиотек.

```
import numpy as np
from sklearn.datasets import fetch_mldata

%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt

from random import *
```

Затем импортируем набор данных MNIST.

```
mnist = fetch_mldata('MNIST original')
Xinput_,yinput_ = mnist["data"], mnist["target"]
```

Напомним, что этот набор данных имеет 70 000 наблюдений и состоит из изображений в оттенках серого, размером 28×28 пикселов каждое. Вы можете снова пересмотреть *главу 2*, где эта тема подробно рассматривается. Затем давайте выберем только цифры 1 и 2 и перешкалируем метки с целью обеспечения того, чтобы цифра 1 имела метку 0, а цифра 2 — метку 1. Из *главы 2* вы помните, что стоимостная

функция, которую мы будем использовать для логистической регрессии, ожидает, что эти две метки будут равны 0 и 1.

```
Xinput = Xinput_[np.any([yinput_ == 1,yinput_ == 2], axis = 0)]
yinput = yinput_[np.any([yinput_ == 1,yinput_ == 2], axis = 0)]
yinput = yinput - 1
```

Мы можем проверить число наблюдений с помощью инструкции

```
Xinput.shape[0]
```

У нас 14 867 наблюдений (изображений). Теперь проделаем небольшой трюк. Для упрощения исходного кода мы хотим, чтобы каждый блок имел одинаковое число наблюдений. Технически говоря, этого не требуется, и у вас часто последний блок в конце будет иметь число наблюдений меньше, чем в других. В этом случае, если мы хотим иметь 10 блоков, то не сможем иметь в каждом блоке одинаковое число наблюдений, потому что 14 867 не кратно 10. Для того чтобы упростить ситуацию еще больше, давайте просто удалим из набора данных последние семь изображений. (С эстетической точки зрения, это ужасно, но это сделает наш код намного легче для понимания и написания.)

```
Xinput = Xinput[:-7,:]
yinput = yinput[:-7]
```

Теперь давайте создадим 10 массивов, каждый из них содержит список индексов, которые мы будем использовать для отбора изображений.

```
foldnumber = 10
idx = np.arange(0,Xinput.shape[0])
np.random.shuffle(idx)
al = np.array_split(idx,foldnumber)
```

В каждом блоке будет, как и ожидалось, 1486 изображений. Теперь создадим массивы с изображениями.

```
Xinputfold = []
yinputfold = []
for i in range(foldnumber):
    tmp = Xinput[al[i],:]
    Xinputfold.append(tmp)
    ytmp = yinput[al[i]]
    yinputfold.append(ytmp)

Xinputfold = np.asarray(Xinputfold)
yinputfold = np.asarray(yinputfold)
```

Если вы думаете, что этот код запутан, то будете правы. С помощью библиотеки Scikit-learn это можно сделать быстрее, но очень поучительно посмотреть, как это делается вручную, шаг за шагом. Безусловно, приведенный выше фрагмент кода, в котором каждый шаг изолирован, облегчает его понимание. Сначала мы создаем пустые списки: `Xinputfold` и `yinputfold`. Каждый элемент списка будет блоком, т. е.

массивом изображений или меток. Поэтому если мы хотим получить все изображения в блоке 2, то используем `Xinputfold[1]`. (Напомним: в Python индексы начинаются с нуля.) В последних двух инструкциях эти два списка преобразовываются в массивы NumPy, которые будут иметь три размерности, как можно легко увидеть с помощью следующих инструкций:

```
print(Xinputfold.shape)
print(yinputfold.shape)
```

В результате получим:

```
(10, 1486, 784)
(10, 1486)
```

В `Xinputfold` первая размерность обозначает номер блока, вторая — наблюдение, третья — пиксельные значения оттенков серого. В `yinputfold` первая размерность обозначает номер блока, вторая — метку. Например, для того чтобы получить изображение с индексом 1234 из блока 0, вам нужно применить такую инструкцию:

```
Xinputfold[0][1234,:]
```

Напомним, что следует проверить, что набор данных по-прежнему сбалансирован в каждом блоке или, другими словами, что у вас столько же единиц, сколько двоек. Давайте проверим блок 0 (вы можете сделать ту же проверку и для других).

```
for i in range(0,2,1):
    print ("метка", i, "составляет",
           np.around(np.count_nonzero(yinputfold[0] == i)/1486.0*100.0,
                     decimals=1), "% из 1486 наблюдений")
```

В результате получим:

```
метка 0 составляет 51.2 % из 1486 наблюдений
метка 1 составляет 48.8 % из 1486 наблюдений
```

Для наших целей это выглядит достаточно сбалансированно. Теперь нужно нормализовать признаки (как это было в *главе 2*).

```
Xinputfold_normalized = np.zeros_like(Xinputfold, dtype = float)
for i in range (foldnumber):
    Xinputfold_normalized[i] = Xinputfold[i]/255.0
```

Вы могли бы нормализовать данные за один присест, но здесь нашей целью было показать, что мы имеем дело с блоками, и сделать исходный код понятным для читателя. Теперь реформируем массивы так, как нам нужно.

```
X_train = []
y_train = []
for i in range(foldnumber):
    tmp = Xinputfold_normalized[i].transpose()
    ytmp = yinputfold[i].reshape(1,yinputfold[i].shape[0])
    X_train.append(tmp)
    y_train.append(ytmp)
```

```
X_train = np.asarray(X_train)
y_train = np.asarray(y_train)
```

Данный исходный код написан самым простым и далеко не самым оптимальным способом для учебных целей. Теперь мы можем проверить размерности окончательных массивов с помощью инструкций:

```
print(X_train.shape)
print(y_train.shape)
```

В результате получим:

```
(10, 784, 1486)
(10, 1, 1486)
```

Именно то, что нужно. Теперь мы готовы приступить к построению сети. Мы будем использовать однонейронную сеть для логистической регрессии с сигмоидальной активационной функцией.

```
import tensorflow as tf
tf.reset_default_graph()
X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())

#W = tf.Variable(tf.zeros([1, n_dim]))
W = tf.Variable(tf.random_normal([1, n_dim],
                stddev= 2.0 / np.sqrt(2.0*n_dim)))
b = tf.Variable(tf.zeros(1))
y_ = tf.sigmoid(tf.matmul(W,X)+b)
cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
training_step = tf.train.AdamOptimizer(learning_rate=learning_rate,
                                       beta1=0.9, beta2=0.999,
                                       epsilon=1e-8).minimize(cost)
init = tf.global_variables_initializer()
```

Здесь мы использовали оптимизатор Adam, но градиентный спуск тоже будет работать. Этот случай очень простой. Для тренировки модели мы воспользуемся уже известной нам Python-функцией.

```
def run_logistic_model(learning_r, training_epochs, train_obs,
                       train_labels, debug = False):
    sess = tf.Session()
    sess.run(init)

    cost_history = np.empty(shape=[0], dtype = float)

    for epoch in range(training_epochs+1):

        sess.run(training_step, feed_dict = {X:train_obs, Y:train_labels,
                                             learning_rate:learning_r})
```

```
        cost_ = sess.run(cost, feed_dict={X:train_obs, Y:train_labels,
                                          learning_rate:learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % 200 == 0) & debug:
            print("Достигнута эпоха",epoch,"стоимость J =",
                  str.format('{0:.6f}', cost_))

    return sess, cost_history
```

В этом месте нам придется перебрать блоки в цикле. Вспомните наш псевдокод в самом начале. Надо выбрать один блок в качестве рабочего набора и натренировать модель на всех остальных блоках. Проделать то же самое для всех блоков. Исходный код может выглядеть следующим образом. (Он немного длинен, поэтому потратьте несколько минут на то, чтобы в нем разобраться.) В исходном коде имеются комментарии, которые сообщают о шагах.

```
train_acc = []
dev_acc = []

for i in range (foldnumber): # Шаг 1

    # Подготовить блоки - шаг 2
    lis = []
    ylis = []
    for k in np.delete(np.arange(foldnumber), i):
        lis.append(X_train[k])
        ylis.append(y_train[k])
        X_train_ = np.concatenate(lis, axis = 1)
        y_train_ = np.concatenate(ylis, axis = 1)

    X_train_ = np.asarray(X_train_)
    y_train_ = np.asarray(y_train_)

    X_dev_ = X_train[i]
    y_dev_ = y_train[i]

    # Шаг 3
    print('Рабочий блок:', i)

    sess, cost_history = run_logistic_model(learning_r=5e-4,
                                            training_epochs=600,
                                            train_obs=X_train_,
                                            train_labels=y_train_,
                                            debug=True)
    # Шаг 4
    correct_prediction=tf.equal(tf.greater(y_, 0.5), tf.equal(Y,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

```
print('Тренировочная точность:',sess.run(accuracy,
                                        feed_dict={X:X_train_,
                                                   Y:y_train_,
                                                learning_rate:5e-4}))
train_acc = np.append(train_acc, sess.run(accuracy,
                                         feed_dict={X:X_train_,
                                                    Y:y_train_,
                                                  learning_rate:5e-4}))
correct_prediction=tf.equal(tf.greater(y_, 0.5), tf.equal(Y,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print('Рабочая точность:',sess.run(accuracy,
                                   feed_dict={X:X_dev_,
                                              Y:y_dev_,
                                              learning_rate:5e-4}))
dev_acc = np.append(dev_acc, sess.run(accuracy,
                                      feed_dict={X:X_dev_,
                                                 Y:y_dev_,
                                                 learning_rate:5e-4}))
sess.close()
```

Исходный код выполняет следующие действия.

1. Перебрать блоки в цикле (в данном случае от 1 до 10), итеративно используя переменную `i` от 0 до 9.
2. Для каждого `i` использовать блок `i` в качестве рабочего набора, а все остальные блоки конкатенировать и использовать результирующий набор в качестве тренировочного.
3. Для каждого `i` натренировать модель.
4. Для каждого `i` оценить точность на двух наборах данных (тренировочном и рабочем) и сохранить значения в двух списках: `train_acc` и `dev_acc`.

Выполнение этого фрагмента кода даст результат, который будет выглядеть следующим образом для каждого блока (вы получите следующий ниже результат 10 раз подряд, один раз для каждого блока):

```
Рабочий блок: 0
Достигнута эпоха 0 стоимость J = 0.766134
Достигнута эпоха 200 стоимость J = 0.169536
Достигнута эпоха 400 стоимость J = 0.100431
Достигнута эпоха 600 стоимость J = 0.074989
Тренировочная точность 0.987289
Рабочая точность: 0.984522
```

Обратите внимание, что по каждому блоку вы получите несколько отличающиеся значения точности. Очень поучительно изучить распределения значений точности, всего 10 значений по числу блоков. На рис. 6.9 показано распределение значений для тренировочного набора (*а*) и рабочего набора (*б*).

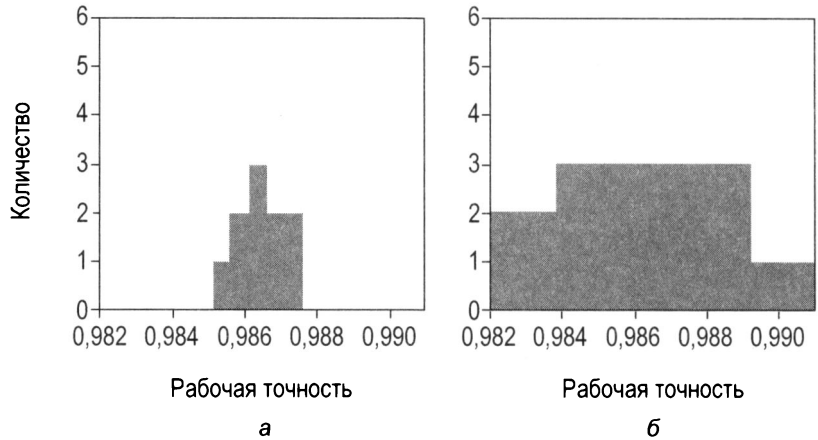


РИС. 6.9. Распределение значений точности для тренировочного (*а*) и рабочего (*б*) наборов. Обратите внимание, что на обеих осях двух графиков используется одинаковая шкала

Данный график весьма поучителен. Вы видите, что значения точности для тренировочного набора сильно сконцентрированы вокруг среднего значения, в то время как значения, которые оцениваются на рабочем наборе, гораздо более разбросаны. Это показывает, что модель на новых данных ведет себя хуже, чем на данных, которые она заучила. Стандартное отклонение для тренировочных данных составляет $5,4 \cdot 10^{-4}$, а для рабочего набора — $2,4 \cdot 10^{-3}$, т. е. в 4,5 раза больше, чем значение на тренировочном наборе. Благодаря этому вы также получаете дисперсию вашей метрики при применении к новым данным и можете оценить, насколько хорошо она обобщает. Если вам интересно узнать, как это сделать быстро с помощью Scikit-learn, можете обратиться к официальной документации по методу `KFold` по адресу: **https://goo.gl/GqlCe4**. При работе с наборами данных, в которых много классов (вспомните приемы разбиения наборов данных), возьмите на заметку и выполните стратифицированный отбор[6]. Библиотека Scikit-learn также предоставляет метод для этого вида отбора — `stratifiedKFold`, к которому можно обратиться здесь: **https://goo.gl/ZBKrdt**.

Теперь вы можете легко найти средние значения и стандартные отклонения. Для тренировочного набора мы имеем среднюю точность 98,7% и стандартное отклонение 0,054%, в то время как для рабочего набора у нас среднее значение 98,6% со стандартным отклонением 0,24%. Так что, теперь вы даже можете давать оценку дисперсии своей метрики. Довольно круто!

---

[6] Википедия, "Стратифицированный отбор образцов", **https://goo.gl/Wd8fuD**, 2018. "В статистических обследованиях, когда подпопуляции в пределах генеральной популяции различаются, целесообразно отбирать образцы из каждой подпопуляции (стратума) независимо. Стратификация — это процесс деления членов популяции на однородные подгруппы перед отбором образцов".

Ручной метрический анализ: пример

Ранее уже упоминалось, что иногда целесообразно выполнить ручной анализ данных с целью проверки правдоподобности получаемых вами результатов (или ошибок). Приведем простой пример, который даст конкретное представление о том, о чем идет речь и насколько это может быть сложным. Давайте рассмотрим следующее: наша очень простая модель (вспомните, что мы используем только один нейрон) может получать 98%-ю точность. Разве задача распознавания цифр настолько проста? Давайте посмотрим, так ли это. Прежде всего, обратите внимание, что тренировочный набор даже не имеет двумерной информации из изображений. Если вы помните, каждое изображение преобразовывается в одномерный массив значений: значения оттенков серого по каждому пикселу, начиная с левого верхнего и далее продвигаясь строка за строкой сверху вниз. Разве единицы и двойки так легко распознать? Давайте проверим, как может выглядеть реальный вход в нашу модель. Начнем с анализа цифры 1. Возьмем пример из блока 0. На рис. 6.10 представлены изображение единицы и гистограмма значений оттенков серого для 784 пикселов, как они видятся из нашей модели. Напомним, что в качестве наблюдений мы имеем одномерный массив из 784 значений пикселов изображения с оттенками серого.

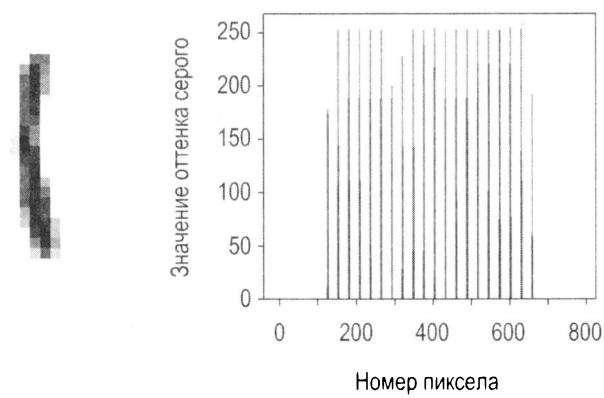


РИС. 6.10. Пример из блока 0 для цифры 1. Изображение слева представляет собой гистограмму значений 784 пикселов, содержащих оттенки серого, как они видятся в нашей модели справа. Напомним, что на входе мы имеем одномерный массив из 784 значений пикселов изображения с оттенками серого

Как вы помните, мы реформируем 28×28-пиксельное изображение в одномерный массив, поэтому при реформировании цифры 1 на рис. 6.10 мы будем находить черные точки примерно каждые 28 пикселов, т. к. единица представляет собой почти вертикальный столбик из черных точек. На рис. 6.11 показаны другие единицы, и вы заметите, что, когда они реформируются в одну размерность, все они выглядят одинаково: несколько полос примерно равноотстоят друг от друга. Теперь, когда вы знаете, что нужно искать, вы можете легко сказать, что все изображения на рис. 6.11 относятся к цифре 1.

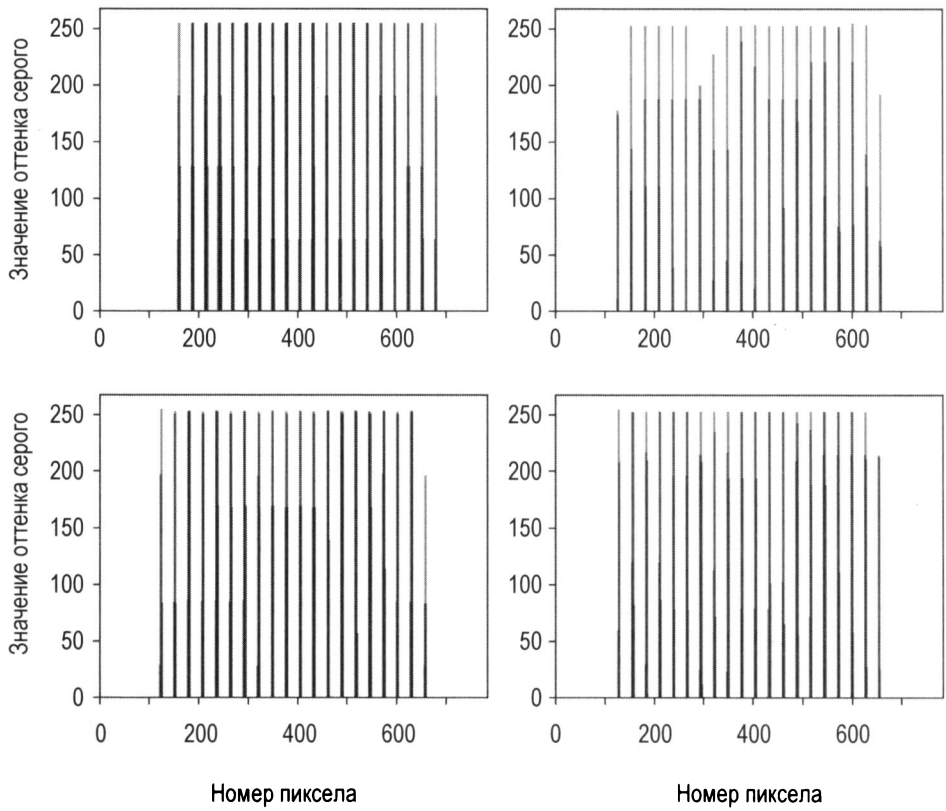


РИС. 6.11. Четыре примера цифры 1, реформированной в одномерные массивы. Все выглядят одинаково: несколько столбиков примерно равноотстоят друг от друга

Теперь давайте посмотрим на цифру 2. На рис. 6.12 приведен пример, подобный тому, что мы имели на рис. 6.10.

Теперь все выглядит иначе. У нас два участка, в которых столбики расположены намного плотнее, что видно на графике справа на рис. 6.12. Это происходит между пикселами 100 и 200 и в особенности после пиксела 500. Почему? Дело в том, что эти два участка соответствуют двум горизонтальным частям изображения. На рис. 6.13 показано, как выглядят разные части при реформировании в одномерные массивы.

При реформировании в одномерный массив горизонтальные части I и I явно отличаются от части III. Вертикальная часть III выглядит как цифра 1, со многими равноотстоящими столбиками, как видно в правой нижней гистограмме с надписью III, в то время как более горизонтальные части выглядят как множество столбиков, кластеризованных в группы, что можно увидеть в правой верхней и левой нижней гистограммах с надписями I и II. Поэтому если при изменении формы вы найдете эти группы столбиков, то вы смóтрите на 2. Если вы видите только равноотстоящие малые группы столбиков, как на графике III на рис. 6.13, то вы смóтрите на 1. Если вы знаете, что искать, то вам даже не нужно смотреть на двумерное изображение.

Обратите внимание, что эта закономерность очень постоянна. На рис. 6.14 представлены четыре примера цифры 2 и ясно видны более широкие кластеры столбиков.

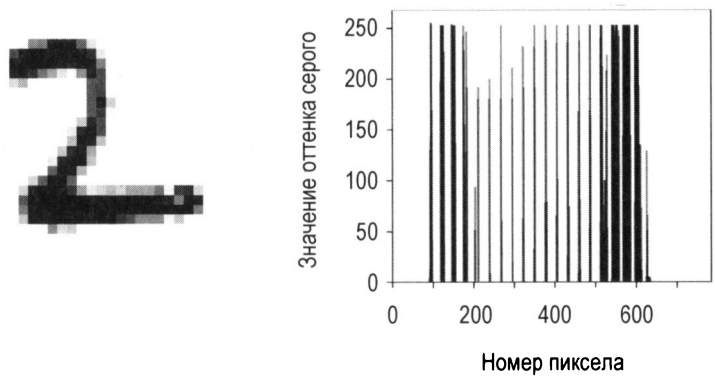


РИС. 6.12. Пример из блока 0 для цифры 2. Изображение слева представляет собой гистограмму значений 784 пикселов с оттенками серого, как они видятся в нашей модели. Напомним, что в качестве наблюдений мы имеем одномерный массив из 784 значений пикселов изображения с оттенками серого

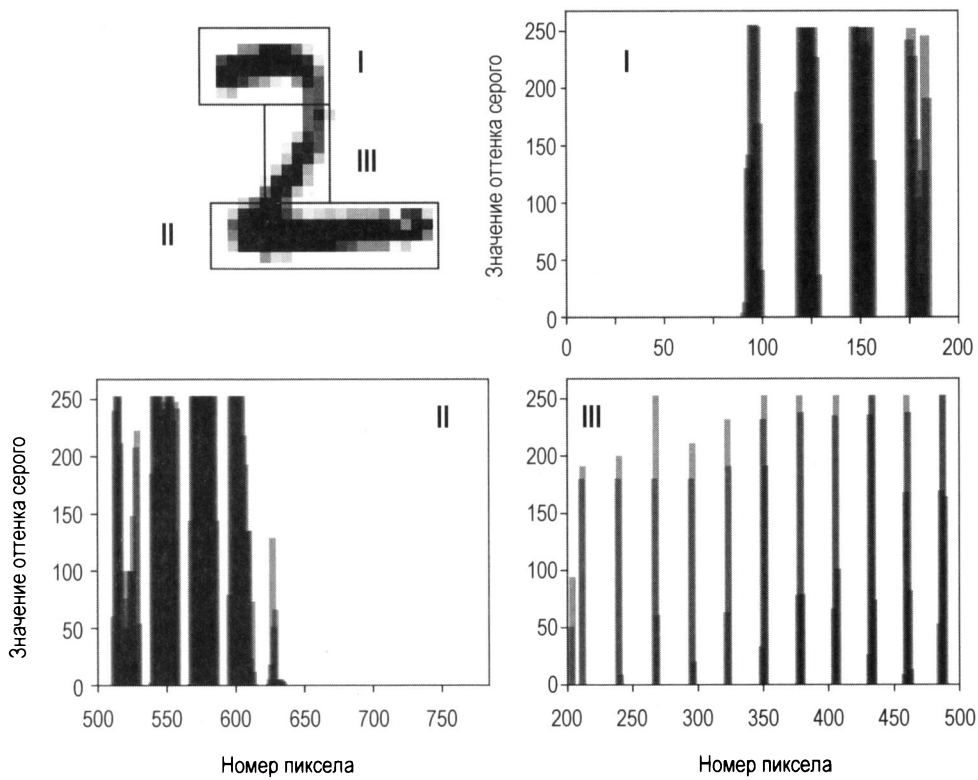


РИС. 6.13. Так выглядят разные части изображений при реформировании в одномерные массивы. Горизонтальные части обозначены, как I и II, и более вертикальная часть, как III

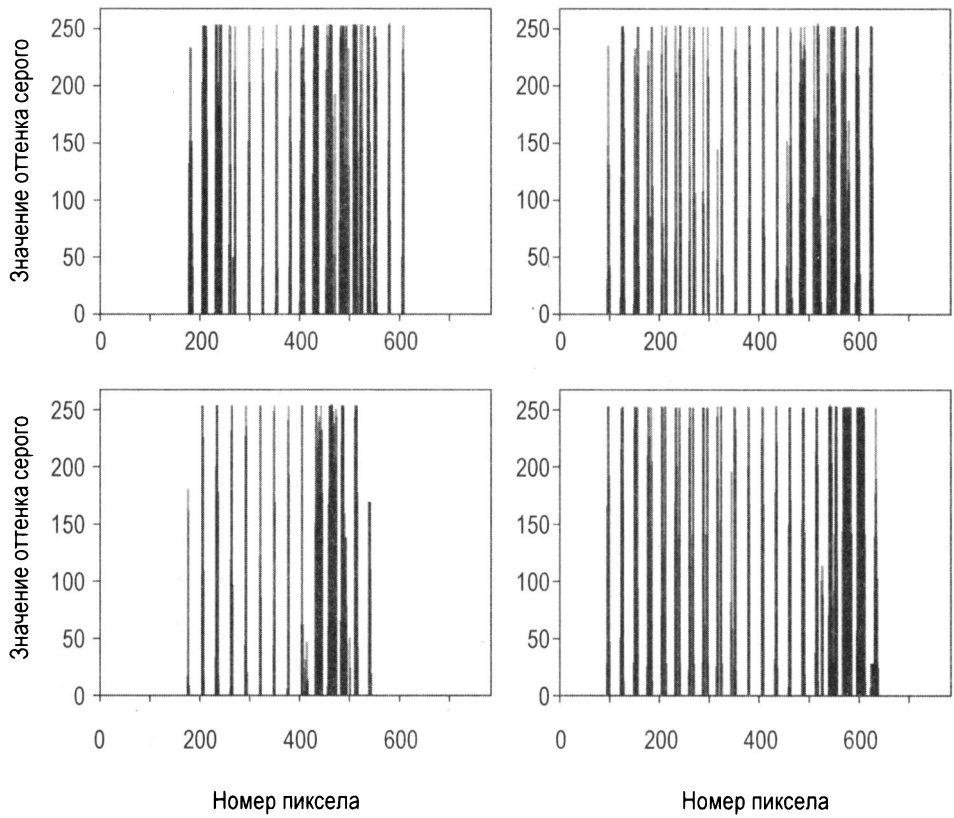


РИС. 6.14. Четыре примера цифры 2, реформированные в одномерные массивы. Четко видны более широкие группы столбиков

Как вы понимаете, эта закономерность легко улавливается алгоритмом, и поэтому следует ожидать, что наша модель будет работать хорошо. Даже человек без каких-либо усилий может распознать изображения после их реформирования. В реальном проекте нет необходимости в таком подробном анализе, но он весьма показателен и позволяет увидеть то, что вы можете узнать из своих данных. Понимание характеристик данных помогает в разработке модели либо помогает понять, почему она не работает. Продвинутые архитектуры, такие как сверточные сети, способны очень эффективно заучивать эти двумерные признаки.

Давайте же проверим, как сеть научилась распознавать цифры. Вы помните, что выход нейрона равен

$$\hat{y} = \sigma(z) = \sigma(w_1 x_1 + w_2 x_2 + ... + w_n x_n + b),$$

где σ — сигмоидальная функция; x_i для $i = 1, ..., 784$ — значения пиксела изображения с оттенками серого; w_i для $i = 1, ..., 784$ — веса; b — смещение. Напомним, что при $\hat{y} > 0,5$ мы относим изображение к классу 1 (цифра 2), а при $\hat{y} < 0,5$ мы относим изображение к классу 0 (цифра 1). Из обсуждения сигмоидальной

функции в *главе 2* вы помните, что $\sigma(z) \geq 0,5$ при $z \geq 0$ и $\sigma(z) < 0,5$ при $z < 0$. Это означает, что сеть должна заучивать веса так, чтобы для всех единиц у нас было $z < 0$, а для всех двоек $z \geq 0$. Посмотрим, так ли это на самом деле. На рис. 6.15 представлен график для цифры 1, из которого вы можете найти веса w_i (сплошная линия) натренированной сети после 600 эпох (и после достижения точности 98%) и значение пиксела x_i с оттенком серого, перешкалированное так, чтобы иметь максимум 0,5 (пунктирная линия). Обратите внимание, что всякий раз, когда x_i является большим, w_i — отрицательно. И когда $w_i > 0$, x_i почти равно нулю. Ясно, что результат $w_1x_1 + w_2x_2 + ... + w_nx_n + b$ будет отрицательным, и, следовательно, $\sigma(z) < 0,5$, и сеть будет идентифицировать изображение как 1. Масштаб изображения увеличен для того, чтобы сделать это поведение заметнее.

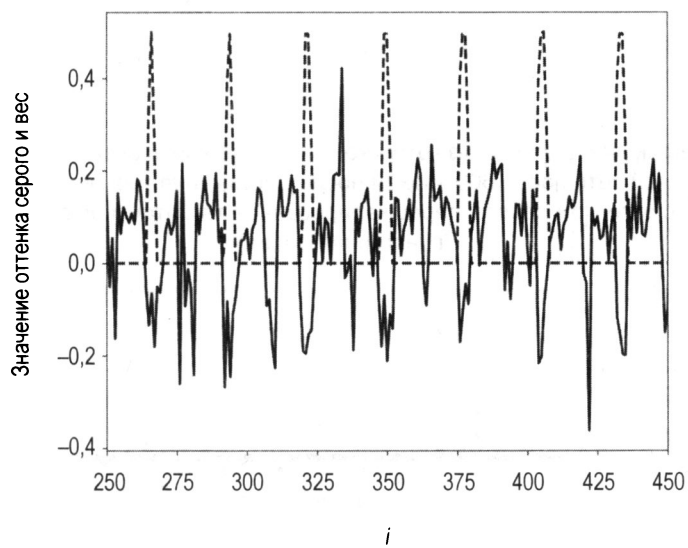


РИС. 6.15. График для цифры 1, из которого вы можете найти веса w_i (сплошная линия) натренированной сети после 600 эпох (и после достижения точности 98%) и значение пиксела x_i с оттенком серого, прошкалированное так, чтобы иметь максимум 0,5 (пунктирная линия)

На рис. 6.16 приведен тот же график для цифры 2. Как вы помните из предыдущего обсуждения, в случае двойки мы видим много столбиков, кластеризованных в группы вплоть до пиксела 250 (примерно). Давайте проверим веса в этом участке. Теперь вы увидите, что там, где значения пиксела с оттенками серого большие, веса — положительны, давая положительное значение z, и, следовательно, $\sigma(z) \geq 0,5$, и поэтому изображение будет классифицировано как 2.

В качестве дополнительной проверки построен график w_ix_i для всех значений i для цифры 1 (рис. 6.17). Видно, что почти все точки лежат ниже нуля. Также обратите внимание на то, что в этом случае $b = -0,16$.

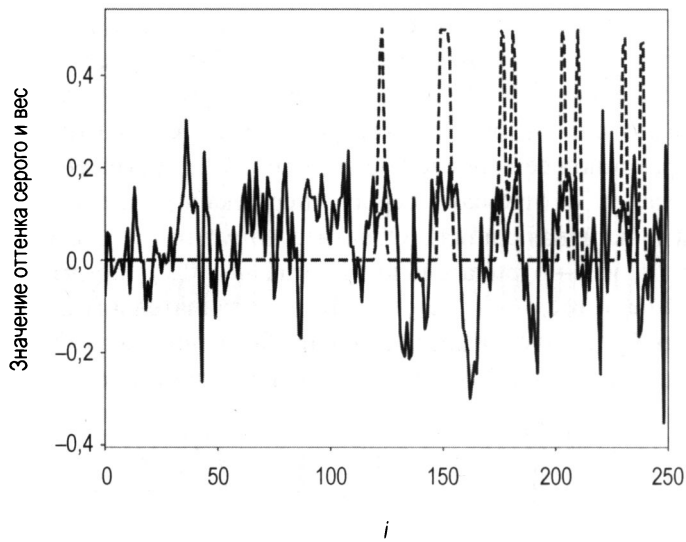


РИС. 6.16. График для цифры 2, из которого вы можете найти веса w_i (сплошная линия) натренированной сети после 600 эпох (и после достижения точности 98%) и значение пиксела x_i с оттенком серого, перешкалированное так, чтобы иметь максимум 0,5 (пунктирная линия)

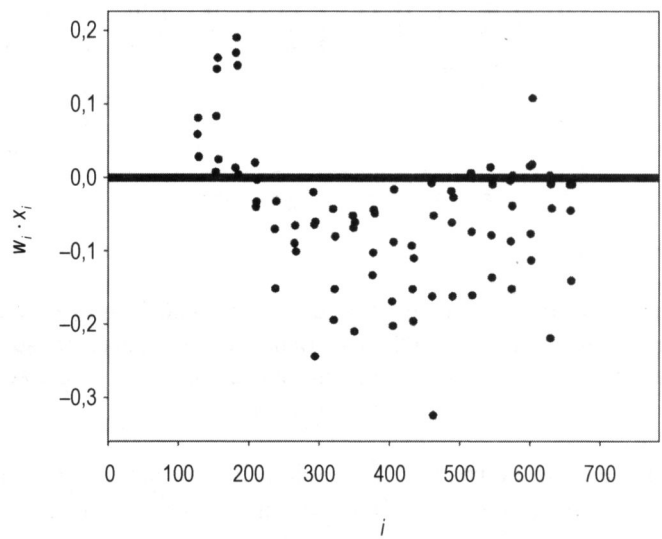


РИС. 6.17. $w_i x_i$ для $i = 1, \ldots, 784$ для цифры 1. Почти все значения лежат ниже нуля. Толстая линия в нуле состоит из всех точек i, таких, что $w_i x_i = 0$

Как видите, в очень простых случаях можно понять процесс заучивания в сети, и, следовательно, гораздо проще отлаживать случаи странного поведения. Но не ожидайте, что это будет возможным при работе с гораздо более сложными случаями. Анализ, который мы провели, был бы не так прост, например, если вместо цифр 1 и 2 вы попытались сделать то же самое с цифрами 3 и 8.

ГЛАВА 7

Гиперпараметрическая настройка

В этой главе мы рассмотрим задачу поиска наилучших гиперпараметров с целью обеспечения наилучших результатов от ваших моделей. В начале будет дано описание задачи черно-ящичной оптимизации и того, как этот класс задач связан с гиперпараметрической настройкой. Вы увидите три наиболее известных метода решения таких задач: решеточный поиск, случайный поиск и байесова оптимизация. Будет показано на примерах, какой из них и в каких условиях работает, и предоставлено несколько хитрых приемов, которые очень полезны для улучшения оптимизации и отбора на логарифмической шкале. В конце главы будут продемонстрированы приемы применения этих технических решений для регулировки глубокой модели с помощью набора данных Zalando.

Черно-ящичная оптимизация

Задача гиперпараметрической настройки является подклассом более общей задачи — черно-ящичной оптимизации. Черно-ящичная функция $f(x)$

$$f(x)\colon \mathbb{R}^n \to \mathbb{R}$$

представляет собой функцию, аналитическая форма которой неизвестна. Черно-ящичная функция может быть вычислена для получения ее значения по всем значениям x, которые определены, но при этом никакой другой информации (например, ее градиента) получить невозможно. Как правило, под глобальной оптимизацией черно-ящичной функции (иногда именуемой задачей черного ящика) подразумевается, что мы пытаемся найти максимум или минимум функции $f(x)$, иногда при определенных ограничениях. Вот несколько примеров такого рода задач.

- Отыскание гиперпараметра для той или иной машинно-обучающейся модели, который максимизирует выбранную оптимизационную метрику.

- Поиск максимума или минимума функции, которую можно вычислить только численно или с помощью программного кода, который мы не можем посмотреть. В некоторых отраслевых контекстах могут существовать очень сложный устаревший код и некоторые функции, которые должны быть максимизированы на основе его результатов.
- Отыскание лучшего места для бурения. В этом случае ваша функция будет состоять в том, сколько нефти вы можете найти и зафиксировать ваше местоположение.
- Нахождение оптимального сочетания параметров в ситуациях, которые слишком сложны для моделирования, например, как оптимизировать объем топлива, диаметр каждой ступени ракеты, точную траекторию и т. д. при запуске ракеты в космос.

Этот очень увлекательный класс задач породил целый ряд умных решений. Вы увидите три из них: решеточный поиск, случайный поиск и байесова оптимизация. Если вы заинтересовались и хотите узнать по этой теме больше, то можете посетить конкурс черно-ящичной оптимизации по адресу **https://goo.gl/LY7huY**. Правила данного конкурса отражают задачи из реальной жизни. Ставится задача, для которой вы должны оптимизировать функцию (найти максимум или минимум) через черно-ящичный интерфейс. Вы можете получить значение функции для всех значений x, но никакой другой информации, например ее градиенты, получить невозможно.

Почему отыскание наилучших гиперпараметров для нейронных сетей представляет собой черно-ящичную задачу? Потому что мы не можем вычислить информацию, такую как градиенты выхода сети, относительно гиперпараметров, в особенности при использовании сложных оптимизаторов или кластомизированных функций; нам нужны другие подходы, позволяющие отыскивать наилучшие гиперпараметры, которые максимизируют избранную оптимизационную метрику. Обратите внимание, что если бы у нас были градиенты, то для отыскания максимума или минимума мы могли бы использовать такой алгоритм, как градиентный спуск.

ПРИМЕЧАНИЕ. Наша черно-ящичная функция f будет нейросетевой моделью (включая такие вещи, как оптимизатор, форма стоимостной функции и т. д.), которая на выходе дает оптимизационную метрику с учетом полученных гиперпараметров на входе, и x будет массивом, содержащим гиперпараметры.

Проблема может показаться довольно тривиальной. Почему бы не попробовать все имеющиеся возможности? Дело в том, это могло бы быть возможным в примерах, которые вы рассматривали в предыдущих главах, но если вы работаете над залачей, и при этом тренировка модели занимает неделю, то такой подход может стать проблематичным. Поскольку, как правило, у вас будет не один и не два, а многочисленные гиперпараметры, опробовать все их варианты будет невозможно. Давайте рассмотрим пример, который даст нам более четкое понимание этой проблемы. Предположим, мы тренируем модель нейронной сети с несколькими слоями. Мы

можем принять решение рассмотреть следующие гиперпараметры с целью увидеть, какая комбинация работает лучше:

- темп заучивания: предположим, мы хотим попробовать значения $n \cdot 10^{-4}$ для $n = 1, ..., 10^2$ (100 значений);
- регуляризационный параметр: 0; 0,1; 0,2; 0,3; 0,4 и 0,5 (6 значений);
- выбор оптимизатора: градиентный спуск, RMSProp или Adam (3 значения);
- число скрытых слоев: 1, 2, 3, 5 и 10 (5 значений);
- число нейронов в скрытых слоях: 100, 200 и 300 (3 значения).

Имейте в виду, что если вы хотите проверить все возможные комбинации, то вам потребуется натренировать свою сеть

$$100 \times 6 \times 3 \times 5 \times 3 = 27\,000 \text{ раз.}$$

Если тренировка занимает 5 минут, то вам потребуется 13,4 недели вычислительного времени. Если тренировка займет часы или дни, то у вас не будет никаких шансов. Если, к примеру, тренировка занимает один день, то на то, чтобы испробовать все возможности, вам потребуется 73,9 лет. Большинство приемлемых вариантов гиперпараметров приходят с опытом. Например, вы всегда можете безопасно использовать Adam, потому что это лучший оптимизатор из имеющихся (почти во всех случаях). Но вы не сможете избежать попыток отрегулировать другие параметры, такие как число скрытых слоев или темп заучивания. Число необходимых комбинаций можно сократить, опираясь на накопленный опыт (как в случае с оптимизатором), либо с помощью некоторого умного алгоритма, как вы увидите позже в этой главе.

Замечания по черно-ящичным функциям

Черно-ящичные функции обычно подразделяются на два основных класса:

- *дешевые функции* — функции, которые могут вычисляться тысячи раз;
- *дорогостоящие функции* — функции, которые могут вычисляться только несколько раз, обычно менее 100 раз.

Если черно-ящичная функция является дешевой, то выбор оптимизационного метода не критичен. Например, мы можем оценить градиент относительно x численно либо попросту произведя поиск максимума, вычисляя функции на большом числе точек. Если функция является дорогостоящей, то нам нужны гораздо более умные подходы. Один из них — байесова оптимизация, о которой будет рассказано позже в этой главе для получения представления о том, как работают эти методы и насколько они сложны.

ПРИМЕЧАНИЕ. Это особенно касается сферы глубокого обучения, где нейронные сети почти всегда являются дорогостоящими функциями.

В случае дорогостоящих функций мы должны найти способы решить эту задачу с наименьшим числом возможных вычислений.

Задача гиперпараметрической настройки

Прежде чем обратиться к вопросу отыскания наилучших гиперпараметров, ненадолго вернемся к нейронным сетям и посмотрим, что требуется настраивать в глубоких моделях. Как правило, рассуждая о гиперпараметрах, новички имеют в виду только числовые параметры, такие как темп заучивания или, к примеру, регуляризационный параметр. Напомним, что с целью обеспечения более высоких результатов следующие параметры тоже могут варьироваться:

- *число эпох* — иногда просто более продолжительная тренировка даст вам более высокие результаты;
- *выбор оптимизатора* — вы можете попробовать выбрать другой оптимизатор. Если вы используете простой градиентный спуск, то можете попробовать Adam и посмотреть, будут ли результаты улучшены;
- *варьирование регуляризационного метода* — как обсуждалось ранее, существует несколько способов применения регуляризации. Возможно, стоит попробовать поварьировать регуляризационный метод;
- *выбор активационной функции* — хотя в предыдущих главах для нейронов в скрытых слоях всегда использовалась активационная функция ReLU, другие могут работать намного лучше. Например, сигмоидальная функция или функция Swish вполне способны помочь вам улучшить результаты;
- *число слоев и число нейронов в каждом слое* — попробовать разные конфигурации, к примеру, попробовать слои с разным числом нейронов;
- *методы ослабления темпа заучивания* — попробовать (если вы не используете оптимизаторы, которые уже это делают) другие методы ослабления темпа заучивания;
- *размер мини-пакета* — поварьировать размер мини-пакета. При малом объеме данных вы можете использовать пакетный градиентный спуск. При большом их объеме мини-пакеты эффективнее;
- *методы инициализации весов*.

Давайте сгруппируем параметры, которые можно регулировать в модели в следующие три категории.

- Параметры, которые являются непрерывными вещественными числами или, другими словами, которые могут принимать любое значение. Пример: темп заучивания, регуляризационный параметр.
- Параметры, которые являются дискретными, но теоретически могут принимать бесконечное число значений. Пример: число скрытых слоев, число нейронов в каждом слое или число эпох.

- Параметры, которые дискретны и могут принимать только конечное число возможностей. Пример: оптимизатор, активационная функция, метод ослабления темпа заучивания.

Что касается третьей категории, то тут не так уж много можно сделать, кроме как попробовать все возможности. Как правило, эти параметры полностью изменяют саму модель, а следовательно, невозможно моделировать их эффекты, делая тест единственной возможностью. Кроме того, эту категорию параметров можно отрегулировать прежде всего за счет опыта. Широко известно, что оптимизатор Adam почти всегда является лучшим вариантом выбора, поэтому в начале вы можете сконцентрировать свои усилия где-то еще. Что касается первой и второй категорий, то тут посложнее, и для нахождения наилучших значений параметров нам придется подыскать какие-то умные решения.

Образец черно-ящичной задачи

Для того чтобы попробовать свои силы в решении черно-ящичной задачи, давайте создадим "поддельную" черно-ящичную задачу. Данная задача заключается в следующем: найти максимум функции $f(x)$, заданной формулами

$$g(x) = \cos\frac{x}{4} - \sin\frac{x}{4} - \frac{5}{2}\cos\frac{x}{2} + \frac{1}{2}\sin\frac{x}{2};$$

$$h(x) = -\cos\frac{x}{3} - \sin\frac{x}{3} - \frac{5}{2}\cos\frac{2}{3}x + \frac{1}{2}\sin\frac{2}{3}x;$$

$$f(x) = 10 + g(x) + \frac{1}{2}h(x),$$

притворившись, что формулы неизвестны. Эти формулы позволят нам проверить результаты, но мы сделаем вид, что они неизвестны. Вы, возможно, задаетесь вопросом: почему мы используем такие сложные формулы? Дело в том, что просто требовалось иметь нечто с несколькими максимумами и минимумами; это позволит вам получить представление о работе методов на нетривиальном примере. Функция $f(x)$ может быть реализована на Python следующим образом:

```
def f(x):
    tmp1 = -np.cos(x/4.0)-np.sin(x/4.0)-
2.5*np.cos(2.0*x/4.0)+0.5*np.sin(2.0*x/4.0)
    tmp2 = -np.cos(x/3.0)-np.sin(x/3.0)-
2.5*np.cos(2.0*x/3.0)+0.5*np.sin(2.0*x/3.0)
    return 10.0+tmp1+0.5*tmp2
```

На рис. 7.1 видно, как выглядит $f(x)$.

Максимум находится в приблизительном значении $x = 69{,}18$ и имеет значение 15,027. Наша задача — найти этот максимум наиболее эффективным способом, не зная ничего о $f(x)$, кроме ее значения в любой точке. Когда мы говорим

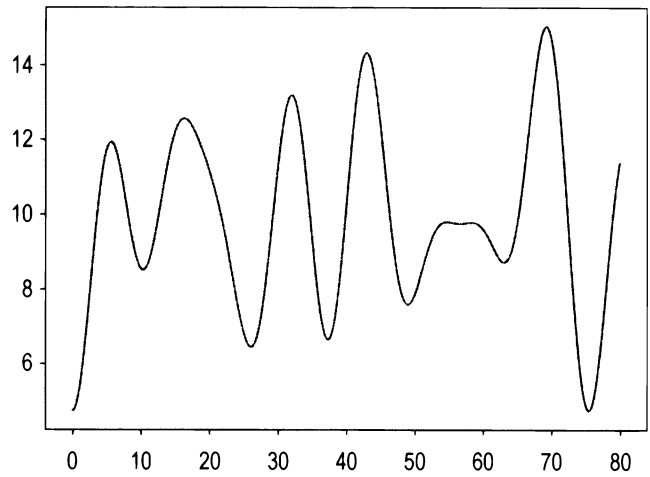


РИС. 7.1. График функции $f(x)$, как описано в тексте

"эффективно", мы, конечно же, имеем в виду наименьшее число возможных оцениваний функции.

Решеточный поиск

Первый метод, который мы рассмотрим, — решеточный поиск. Он является наименее "интеллектуальным". Решеточный поиск сводится к опробыванию функции через регулярные промежутки времени и проверке того, для какого x функция $f(x)$ принимает наибольшее значение. В этом примере мы хотим найти максимум функции $f(x)$ между двумя значениями $x_{\min}$ и $x_{\max}$. Мы просто возьмем n точек, равно расположенных между $x_{\min}$ и $x_{\max}$, и оценим функцию в этих точках. Определим вектор точек

$$\mathbf{x} = \left[x_{\min} \quad x_{\min} + \frac{\Delta x}{n} \quad \ldots \quad x_{\min} + (n-1)\frac{\Delta x}{n} \right],$$

где мы определили $\Delta x = x_{\max} - x_{\min}$. Затем мы оцениваем функцию $f(x)$ в этих точках, получая вектор значений

$$\mathbf{f} = \left[f(x_{\min}) \quad f\left(x_{\min} + \frac{\Delta x}{n} \right) \quad \ldots \quad f\left(x_{\min} + (n-1)\frac{\Delta x}{n} \right) \right].$$

Тогда оценка максимума ($\tilde{x}$, $\tilde{f}$) будет равна

$$\tilde{f} = \max_{0 \le i \le n-1} f,$$

и, допустив, что максимум найден в $i = \tilde{i}$, мы также будем иметь

$$\tilde{x} = x_{\min} + \frac{\tilde{i}\Delta x}{n}.$$

Теперь, как вы понимаете, чем больше используется точек, тем точнее будет оценка максимума. Проблема в том, что если оценивание $f(x)$ является дорогостоящим, то вы не сможете взять столько точек, сколько хотели бы. Вам нужно будет найти баланс между числом точек и точностью. Давайте рассмотрим пример с функцией $f(x)$, который был описан выше. Рассмотрим $x_{max} = 80$ и $x_{min} = 0$ и возьмем $n = 40$ точек. У нас будет $\Delta x/n = 2$. На языке Python можно легко создать вектор **x** с помощью следующей инструкции:

```
gridsearch = np.arange(0,80,2)
```

Массив `gridsearch` будет выглядеть следующим образом:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32,
       34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66,
       68, 70, 72, 74, 76, 78])
```

На рис. 7.2 график функции $f(x)$ изображен непрерывной линией, крестики отмечают точки, которые мы отбираем в решеточном поиске, черный квадрат отмечает точный максимум функции. Правый график показывает участок, увеличенный вокруг максимума.

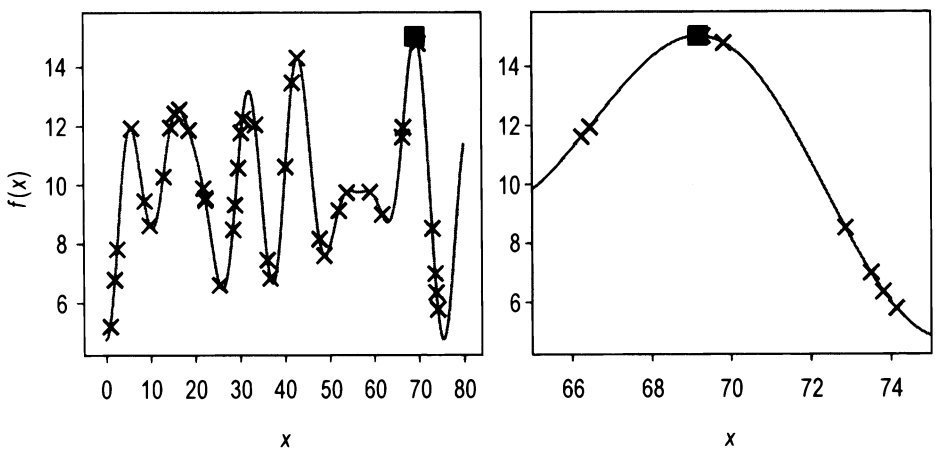


РИС. 7.2. Решетчатый поиск. Функция $f(x)$ на интервале [0; 80]. Крестики отмечают точки, которые мы отбираем в решеточном поиске, черный квадрат — максимум

Вы видите, как отбираемые нами на рис. 7.2 точки подбираются к максимуму, но не делают это точно. Конечно же, отбор большего числа точек приблизил бы к максимуму, но будет стоить большего числа оцениваний $f(x)$. Мы можем легко найти максимум ($\tilde{x}, \tilde{f}$), используя тривиальный фрагмент кода:

```
x = 0
m = 0.0
for i, val in np.ndenumerate(f(gridsearch)):
```

```
    if (val > m):
        m = val
        x = gridsearch[i]
print(x)
print(m)
```

В результате получим:

```
70
14.6335957578
```

Это близко к фактическому максимуму (69,18; 15,027), но не совсем верно. Давайте вернемся к предыдущему примеру, поварьировав число отбираемых точек, а затем посмотрим на полученные результаты. Мы будем варьировать число n точек, отбираемых на интервале от 4 до 160. Для каждого случая мы найдем максимум и его расположение, как описано ранее. Это можно сделать с помощью следующего фрагмента кода:

```
xlistg = []
flistg = []

for step in np.arange(1,20,0.5):
    gridsearch = np.arange(0,80,step)

    x = 0
    m = 0.0
    for i, val in np.ndenumerate(maxim(gridsearch)):
        if (val > m):
            m = val
            x = gridsearch[i]

    xlistg.append(x)
    flistg.append(m)
```

В списках `xlistg` и `flistg` мы найдем позицию найденного максимума и значение максимума для различных значений n.

На рис. 7.3 показано распределение результатов. Черная вертикальная линия является правильным значением максимума.

Как вы видите, результаты сильно различаются и могут быть очень далеки от правильного значения, вплоть до 10. Это говорит нам о том, что использование неправильного числа точек может дать очень неправильные результаты. Наилучшими результатами являются те, у которых наименьший шаг Δx, потому что вероятность приблизиться к максимуму больше. На рис. 7.4 видно, как значение найденного максимума варьирует вместе с шагом Δx.

На увеличенном правом графике рис. 7.4 видно, как меньшие значения Δx дают лучшие значения $\tilde{f}$. Обратите внимание, что шаг 1,0 означает отбор 80 значений $f(x)$. Если, например, оценивание занимает 1 день, то для получения всех необходимых результатов измерений вам придется прождать 80 дней.

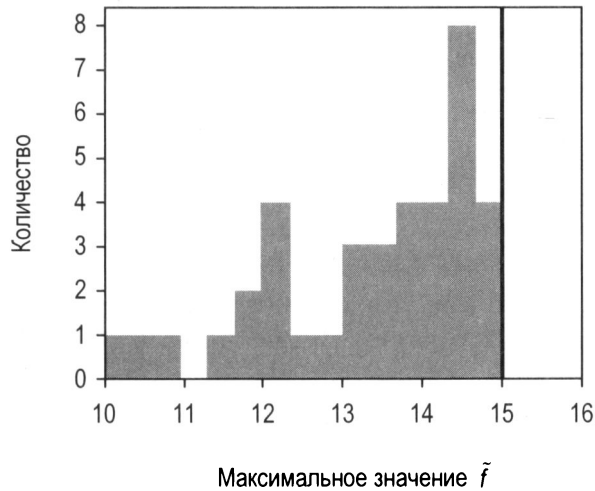


РИС. 7.3. Распределение результатов по $\tilde{f}$, полученных путем варьирования числа n точек, отобранных в решеточном поиске. Черная вертикальная линия указывает на реальный максимум $f(x)$

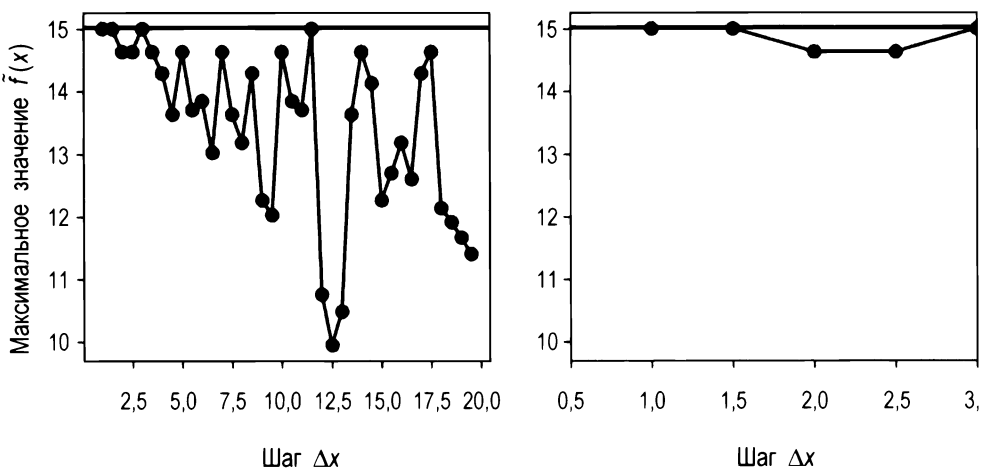


РИС. 7.4. Поведение найденного значения максимума в сопоставлении с шагом Δx

ПРИМЕЧАНИЕ. Метод решеточного поиска эффективен только тогда, когда черно-ящичная функция является дешевой. Для получения хороших результатов обычно требуется очень большое число образцов точек.

Для того чтобы убедиться, что вы действительно получаете максимум, вы должны уменьшать шаг Δx или увеличивать число образцов точек до тех пор, пока максимальное значение, которое вы найдете, не изменится заметно больше. В предыдущем примере, как видно из правого графика на рис. 7.4, мы уверенно приближаем-

ся к максимуму, когда шаг Δx становится меньше, чем примерно 2,0, или, другими словами, когда число отбираемых точек больше или примерно равно 40. Напомним, что на первый взгляд число 40 может показаться довольно небольшим, но если $f(x)$ оценивает метрику глубоко-обучающейся модели, а тренировка, к примеру, занимает 2 часа, то вам предстоит 3,3 дня компьютерного времени. Как правило, в сфере глубокого обучения 2 часа на тренировку модели — это не так много, поэтому перед тем, как начать продолжительный решеточный поиск, лучше сделать оперативную калькуляцию. Имейте также в виду, что при настройке гиперпараметров вы движетесь в многомерном пространстве (вы оптимизируете не один параметр, а много), поэтому число необходимых оцениваний очень быстро становится большим.

Давайте рассмотрим небольшой пример. Предположим, вы решили, что можете позволить себе 50 оцениваний черно-ящичной функции. Если вы решите, что хотите испытать следующие ниже гиперпараметры:

- оптимизатор (RMSProp, Adam или простой градиентный спуск) (3 значения);
- число эпох (1000, 5000 или 10 000) (3 значения),

то вам предстоит сделать девять оцениваний. Сколько вы можете позволить себе оцениваний значений темпа заучивания? Только пять! А с пятью значениями маловероятно приблизиться к оптимальному значению. Цель этого примера — помочь вам понять, что решеточный поиск жизнеспособен только для дешевых черно-ящичных функций. Нередко время является не единственной проблемой. Например, если для тренировки сети вы используете облачную платформу Google, то платите за используемое оборудование посекундно. Возможно, вы располагаете большим объемом времени, но расходы очень быстро могут превысить ваш бюджет.

Случайный поиск

Случайный поиск представляет собой стратегию, которая так же "неинтеллектуальна", как и решеточный поиск, но на удивление работает гораздо лучше. Вместо регулярного отбора x точек на интервале ($x_{\min}$, $x_{\max}$) вы отбираете точки случайно. Это можно сделать с помощью фрагмента кода:

```
import numpy as np
randomsearch = np.random.random([40])*80.0
```

Массив `randomsearch` выглядит так:

```
array([ 0.84639256, 66.45122608, 74.12903502, 36.68827838, 61.71538757,
       69.29592273, 48.76918387, 69.81017465,  1.91224209, 21.72761762,
       22.17756662,  9.65059426, 72.85707634,  2.43514133, 53.80488236,
        5.70717498, 28.8624395 , 33.44796341, 14.51234312, 41.68112826,
       42.79934087, 25.36351055, 58.96704476, 12.81619285, 15.40065752,
       28.36088144, 30.27009067, 16.50286852, 73.49673641, 66.24748556,
```

```
8.55013954, 29.55887325, 18.61368765, 36.08628824, 22.1053749 ,
40.14455129, 73.80825225, 30.60089111, 52.01026629, 47.64968904])
```

В зависимости от используемого разброса получаемые по факту числа могут различаться. Как и в случае решеточного поиска, на рис. 7.5 показан график $f(x)$, где крестики отмечают отобранные точки, а черный квадрат — максимум. На правом графике вы видите участок, увеличенный вокруг максимума.

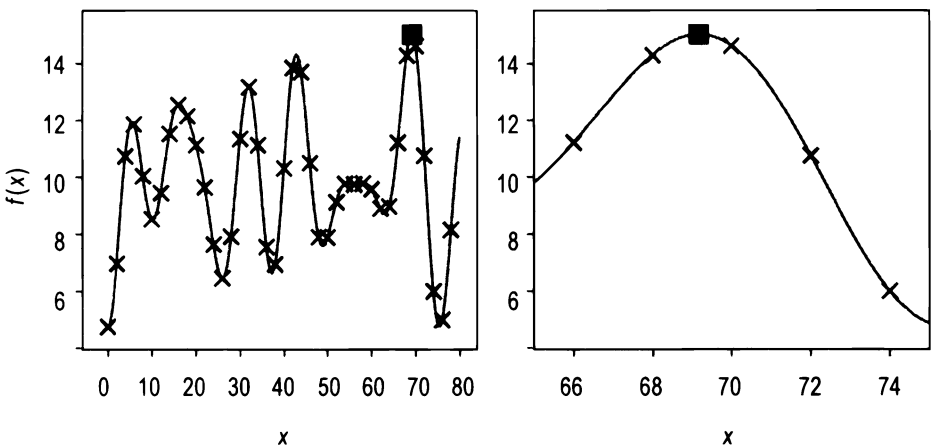


РИС. 7.5. Случайный поиск. Функция $f(x)$ на интервале [0; 80]. Крестики отмечают точки, отобранные с помощью случайного поиска, черный квадрат — максимум

Данный метод несет в себе риск того, что если вам очень не повезет, то ваши случайно отобранные точки ничуть не приблизятся к реальному максимуму. Однако вероятность этого весьма мала. Обратите внимание, что если для случайных точек вы берете постоянное распределение вероятностей, то у вас будет точно такая же вероятность получить точки повсюду. Интересно посмотреть этот метод в работе. Рассмотрим 200 разных случайных множеств из 40 точек, полученных путем варьирования случайного разброса, используемого в коде. Распределения максимальной найденной функции $\tilde{f}$ показаны на рис. 7.6.

Как видите, независимо от используемых случайных множеств, в большинстве случаев вы очень близко подбираетесь к реальному максимуму. На рис. 7.7 представлены распределения максимума, найденного с помощью случайного поиска путем варьирования числа отбираемых точек на интервале от 10 до 80.

Если случайный поиск сравнить с решеточным поиском, то вы увидите, что он приближает результаты к реальному максимуму неизменно лучше. На рис. 7.8 показано сравнение распределения, полученного для максимума $\tilde{f}$ при использовании разного числа n образцов точек с использованием случайного и решеточного поиска. В обоих случаях графики были сгенерированы с 38 разными множествами точек, благодаря чему суммарное количество одинаковое.

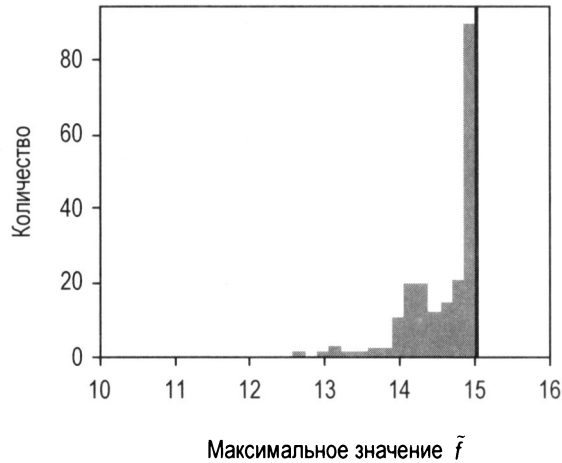


РИС. 7.6. Распределение результатов для $\tilde{f}$, полученных 200 разными случайными множествами из 40 точек, отобранных в случайном поиске. Черная вертикальная линия показывает реальный максимум функции $f(x)$

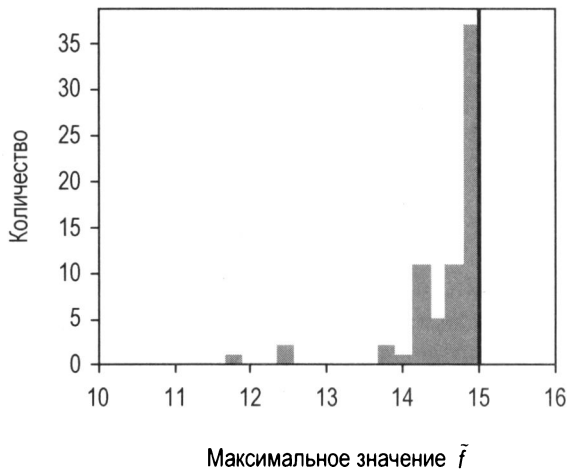


РИС. 7.7. Распределение результатов для $\tilde{f}$, полученных путем варьирования числа n точек, отобранных в случайном поиске на интервале от 10 до 80. Черная вертикальная линия указывает на реальный максимум $f(x)$

Легко увидеть, что случайный поиск в среднем оказывается лучше, чем решеточный поиск. Результирующие значения неизменно находятся ближе к правильному максимуму.

ПРИМЕЧАНИЕ. Случайный поиск неизменно оказывается лучше, чем решеточный поиск, и его следует использовать при любой возможности. Разница между случайным и решеточным поиском становится еще более заметной во время

работы с многомерным пространством для переменной x. Настройка гиперпараметров практически всегда является многомерной оптимизационной задачей.

Если вас интересует очень хорошая статья на тему масштабируемости случайного поиска для высокоразмерных задач, почитайте статью Джеймса Бергстры (James Bergstra) и Джошуа Бенджо (Yoshua Bengio) "Random Search for Hyper-Parameter Optimization" ("Случайный поиск для гиперпараметрической оптимизации"), доступную по адресу **https://goo.gl/efc8Qv**.

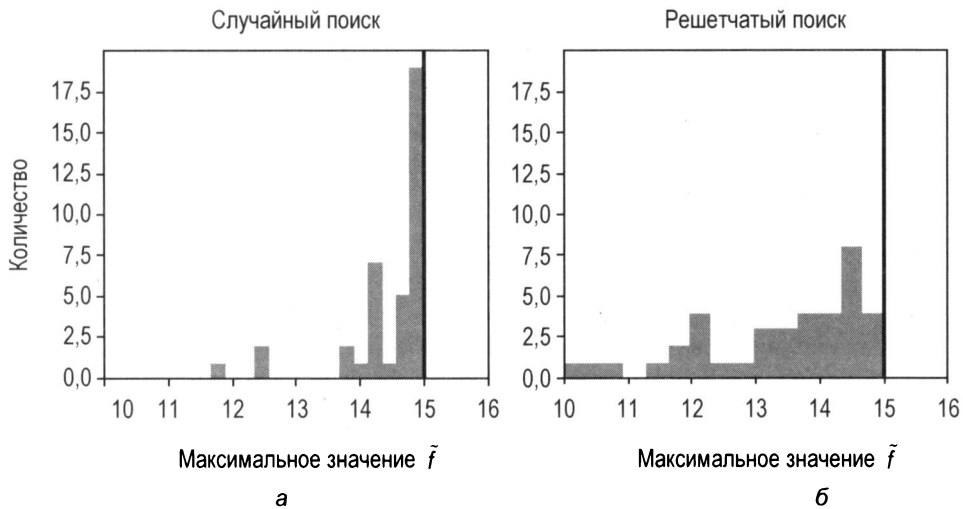


РИС. 7.8. Сравнение распределения $\tilde{f}$ между случайным (*а*) и решеточным (*б*) поиском с варьированием числа n образцов точек. Оба графика в сумме имеют 38 разных множеств образцов точек. Правильное значение максимума на обоих графиках отмечено вертикальной черной линией

Оптимизация с переходом "от крупнозернистости к мелкозернистости"

Тем не менее существует одна оптимизационная хитрость, которая помогает с решеточным или случайным поиском. Она называется оптимизацией "от крупнозернистости к мелкозернистости", или от грубого к утонченному. Предположим, мы хотим найти максимум функции $f(x)$ между $x_{\min}$ и $x_{\max}$. Хотя далее будет разъяснена идея, лежащая в основе случайного поиска, она работает так же хорошо и для решеточного поиска. Следующие шаги воспринимайте как алгоритм, который требуется выполнить для такого рода оптимизации.

1. Выполнить случайный поиск в участке $R_1 = (x_{\min}, x_{\max})$. Обозначим через (x_1, f_1) найденный максимум.

2. Взять меньший участок в окрестности точки x_1, $R_2 = (x_1 - \delta x_1, x_1 + \delta x_1)$ для некоторых δx_1 (это будет рассмотрено позже) и снова выполнить случайный поиск на этом участке. Наша гипотеза, разумеется, состоит в том, что реальный максимум лежит на этом участке. Обозначим максимум, который вы здесь найдете, через (x_2, f_2).
3. Повторить шаг 2 в окрестности точки x_2 на участке, который мы обозначим через R_3, где $\delta x_2 < \delta x_1$, и обозначим максимум, который вы найдете на этом шаге, через (x_3, f_3).
4. Теперь повторить шаг 2 в окрестности точки x_3 на участке, который мы обозначим через R_4, где $\delta x_3 < \delta x_2$.
5. Продолжать таким же образом с требуемой частотой до тех пор, пока максимум (x_i, f_i) на участке R_{i+1} не перестанет изменяться.

Обычно используется только одна или две итерации, но теоретически вы можете продолжить в течение большого числа итераций. Проблема с этим методом состоит в том, что нет никаких гарантий расположения реального максимума на ваших участках R_i. Но данная оптимизация имеет большое преимущество, если максимум все-таки существует. Давайте рассмотрим случай, в котором мы реализуем стандартный случайный поиск. Если мы хотим иметь расстояние между отобранными точками в среднем равное 1% от $x_{max} - x_{min}$, то нам понадобилось бы около 100 точек при выполнении только одного случайного поиска, и поэтому нам пришлось бы выполнить 100 оцениваний. Теперь давайте рассмотрим только что описанный алгоритм. Мы могли бы начать с 10 точек на участке $R_1 = (x_{min}, x_{max})$. Здесь мы обозначим найденный максимум через (x_1, f_1). Затем давайте возьмем $2\delta x = (x_{max} - x_{min})/10$ и снова рассмотрим 10 точек на участке $R_2 = (x_1 - \delta x, x_1 + \delta x)$. В интервале $(x_1 - \delta x, x_1 + \delta x)$ мы будем иметь расстояние между точками в среднем равное 1% от $x_{max} - x_{min}$, но при этом вместо 100 раз мы отбирали результаты функции всего 20 раз, т. е. в 5 раз меньше! Давайте с помощью следующего фрагмента кода выполним отбор результатов функции, которую мы использовали ранее, между $x_{min} = 0$ и $x_{max} = 80$ с 10 точками:

```
np.random.seed(5)
randomsearch = np.random.random([10])*80

x = 0
m = 0.0
for i, val in np.ndenumerate(f(randomsearch)):
    if (val > m):
        m = val
        x = randomsearch[i]
```

В результате мы получим максимальное расположение и значение $x_1 = 69{,}65$ и $f_1 = 14{,}89$, что неплохо, но еще не так точно, как реальные: 69,18 и 15,027. Теперь

давайте снова отберем 10 точек вокруг максимума, который мы нашли на участке $R_2 = (x_1 - 4, x_1 + 4)$.

```
randomsearch = x + (np.random.random([10])-0.5)*8

x = 0
m = 0.0
for i, val in np.ndenumerate(maxim(randomsearch)):
    if (val > m):
        m = val
        x = randomsearch[i]
```

В результате мы получим 69,189 и 15,027. Довольно точный результат всего за 20 оцениваний функции. Если выполнить простой случайный поиск с 500 (в 25 раз больше, чем мы только что проделали) образцами точек, то мы получим $x_1 = 69{,}08$ и $f_1 = 15{,}022$. Данный результат показывает, как этот хитрый прием может оказаться очень полезным. Но помните о риске: если ваш максимум отсутствует на участках R_i, то вы никогда не сможете его найти, т. к. вы все-таки имеете дело со случайными числами. Поэтому всегда полезно выбирать участки $(x_i - \delta x_i, x_i + \delta x_i)$, которые велики настолько, чтобы гарантированно содержать ваш максимум. Их размер зависит, как почти все остальное в сфере глубокого обучения, от набора данных и задачи, и это порой невозможно узнать заранее. К сожалению, требуется тестирование. На рис. 7.9 показаны отобранные точки на функции $f(x)$. На графике слева вы видите первые 10 точек, а справа — участок R_2 с дополнительными 10 точками. Небольшой прямоугольник на левом графике обозначает x на участке R_2.

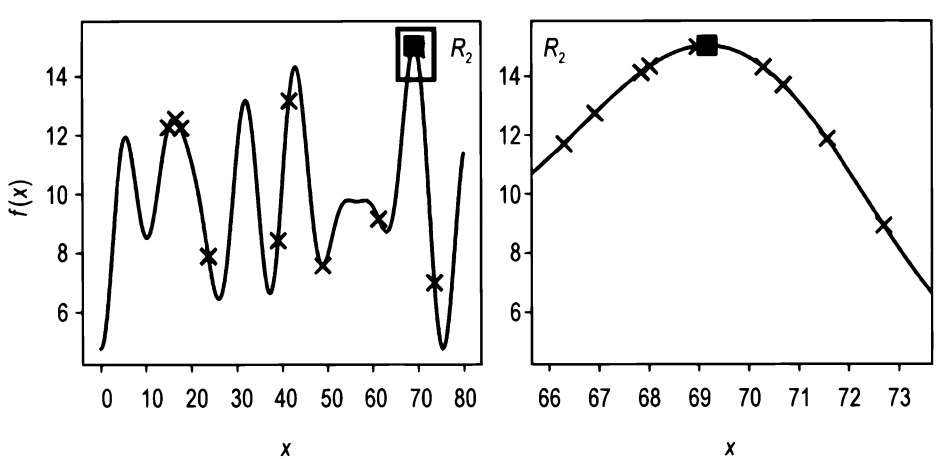


РИС. 7.9. Функция $f(x)$. Крестики отмечают отобранные точки: слева — 10 точек, отобранных на участках R_1 (весь интервал), справа — 10 точек, отобранных на R_2. Черный квадрат отмечает реальный максимум. Участок справа представляет собой увеличенный участок R_2

Выбор числа точек, отбираемых в самом начале, имеет решающее значение. Здесь нам повезло. Рассмотрим разные посевы при выборе первоначальных 10 случайных точек, а потом посмотрим, что произойдет (рис. 7.10). Выбор неправильных первоначальных точек приводит к катастрофе!

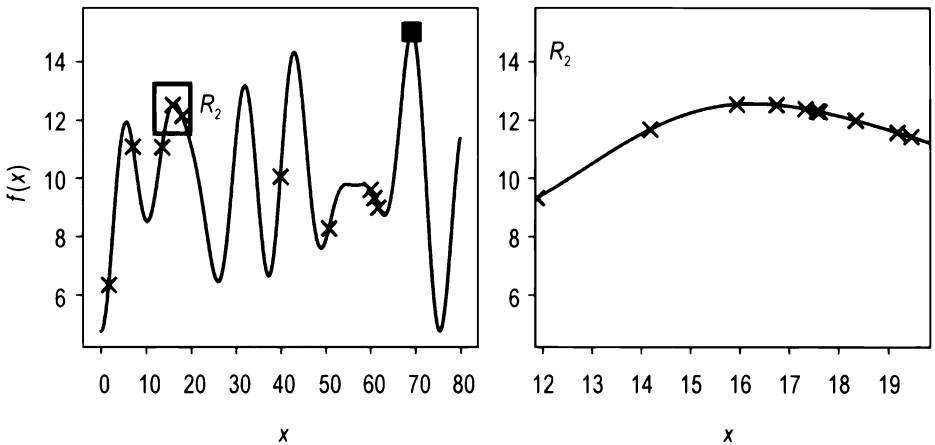


РИС. 7.10. Функция $f(x)$. Крестики отмечают отобранные точки: слева — 10 точек, отобранных на участках R_1 (весь интервал), справа — 10 точек, отобранных на R_2. Черный квадрат отмечает реальный максимум. Алгоритм очень хорошо находит максимум около отметки 16, просто не абсолютный максимум. Малый прямоугольник слева обозначает участок R_2. Участок справа представляет собой увеличенный участок R_2

Обратите внимание, что на рис. 7.10 алгоритм находит максимум вокруг отметки 16, потому что в первоначальных отобранных точках максимальное значение примерно равно $x = 16$, как вы можете видеть на графике слева на рис. 7.10. Ни одна точка не находится близко к реальному максимуму, вокруг $x = 69$. Данный алгоритм очень хорошо находит максимум, просто это не абсолютный максимум. В этом как раз и заключается опасность, с которой вы сталкнетесь при использовании этого приема. Все может быть даже хуже. Рассмотрим рис. 7.11, на котором в самом начале отбирается только одна точка. На рис. 7.11 на левом графике вы видите, что данный алгоритм ни разу не попадает ни в один максимум. В качестве своего результата он просто выдает самые высокие значения точек, отмеченные крестиками на правом графике: (58,4; 9,78).

Если вы решите использовать этот метод, то имейте в виду, что вам все равно в самом начале потребуется большое число точек на то, чтобы приблизиться к максимуму, и только потом вы сможете уточнить свой поиск. Это техническое решение можно использовать для уточнения поиска только после того, как вы относительно уверены в том, что имеете точки вокруг максимума.

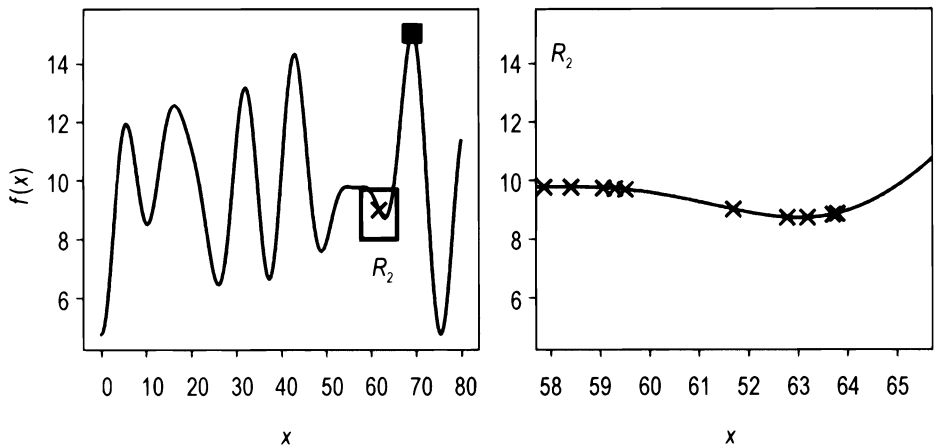


РИС. 7.11. Функция $f(x)$. Крестики отмечают отобранные точки: слева — 1, отобранная на участке R_1 (весь интервал), справа — 10 точек, отобранных на R_2. Черный квадрат отмечает реальный максимум. Данный алгоритм не находит максимума, т. к. на участке R_2 его нет. График справа представляет собой увеличение участка R_2

Байесова оптимизация

В этом разделе мы рассмотрим конкретное и эффективное техническое решение для поиска минимума (или максимума) черно-ящичной функции. Это довольно умный алгоритм, который в сущности заключается в гораздо более интеллектуальном выборе образцов точек, из которых будет оцениваться функция, чем случайный отбор или решеточный поиск. Для того чтобы понять особенности работы данного метода, вы должны сначала взглянуть на несколько новых математических терминов, потому что данный метод не тривиален и требует некоторого понимания более продвинутых понятий. Начнем с регрессии Надарая – Ватсона (Nadaraya – Watson).

Регрессия Надарая – Ватсона

Этот метод был разработан в 1964 году Элизбаром Акакиевичем Надараем в его исследовательской работе "On Estimating Regression" ("Об оценивании регрессии"), опубликованной в российском журнале "Теория вероятностей и ее приложения". Основная идея довольно проста. С учетом неизвестной функции $y(x)$ и N точек, где $x_i = 1, ..., N$, обозначим через $y_i = f(x_i)$, где $i = 1, ..., N$, значение функции, вычисленное в разных x_i. Идея регрессии Надарая – Ватсона заключается в том, что мы можем оценить неизвестную функцию в неизвестной точке x, используя формулу

$$y(x) = \sum_{i=1}^{N} w_i(x) y_i ,$$

где $w_i(x)$ — веса, вычисляемые по формуле

$$w_i(x) = \frac{K(x, x_i)}{\sum_{j=1}^{N} K(x, x_j)},$$

где $K(x, x_i)$ называется *ядром*. С учетом того, как мы определяем веса, мы имеем

$$\sum_{i=1}^{N} w_i(x) = 1.$$

В литературе можно найти несколько ядер, но нас интересует гауссово ядро, часто именуемое *радиальной базисной функцией* (radial basis function, RBF):

$$K(x, x_i) = \sigma^2 e^{-\frac{1}{2l^2}\|x - x_i\|^2}.$$

Параметр l делает гауссову форму шире или уже. Сигмой σ обычно обозначается дисперсия данных, но в этом случае она не играет никакой роли, потому что веса нормализуются до единицы. Как вы увидите позже, такая нормализация лежит в основе байесовой оптимизации.

Гауссов процесс

Прежде чем говорить о гауссовых процессах, сначала необходимо дать определение понятия случайного процесса. *Случайный процесс* относится к любой точке $x \in \mathbb{R}^d$, которой мы назначаем случайную величину $f(x) \in \mathbb{R}$. Случайный процесс является гауссовым, если для любого конечного числа точек их совместное распределение является нормальным. Это означает, что $\forall n \in N$ и для $\forall x_1, \ldots, x_n \in \mathbb{R}^d$ вектор равен

$$\mathbf{f} = \begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N},$$

где под этой формой записи мы подразумеваем, что векторные компоненты подчиняются нормальному распределению, обозначаемому буквой $\mathcal{N}$. Напомним, что случайная величина с гауссовым распределением считается нормально распределенной. Название "гауссов процесс" происходит именно от этого факта. Распределение вероятностей нормального распределения задается функцией

$$g(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}},$$

где μ — среднее значение или математическое ожидание распределения; σ — стандартное отклонение. С этого момента мы будем использовать следующие обозначения:

- m — среднее значение функции f;
- K — ковариация случайных величин:

$$\text{cov}\left[f(x_1), f(x_2)\right] = K(x_1, x_2).$$

Выбор буквы K имеет свою причину. В дальнейшем мы будем считать, что ковариация будет иметь гауссову форму, и для K мы будем использовать ранее определенную радиальную базисную функцию.

Стационарный процесс

Для простоты рассмотрим здесь только стационарные процессы. Случайный процесс является стационарным, если его совместное распределение вероятностей не изменяется со временем. Это означает также, что среднее значение и дисперсия не изменятся при сдвиге во времени. Рассмотрим также процесс, распределение которого зависит только от относительного положения точек. Это приводит к условиям:

$$K(x_1, x_2) = \tilde{K}(x_1 - x_2);$$
$$Var\left[f(x)\right] = \tilde{K}(0).$$

Обратите внимание, что для применения описываемого нами метода сначала необходимо преобразовать данные в стационарные, если это еще не так, например, исключив сезонность или тренды во времени.

Предсказание с помощью гауссовых процессов

Теперь мы достигли самой интересной части: с учетом вектора **f**, как оценить $f(x)$ в произвольной точке x? Поскольку мы имеем дело со случайными процессами, мы будем оценивать вероятность того, что неизвестная функция принимает заданное значение $f(x)$. В математическом плане мы будем предсказывать следующую величину:

$$p\left(f(x) \mid \mathbf{f}\right).$$

Или, другими словами, вероятность получения значения $f(x)$ при наличии вектора **f**, состоящего из всех точек $f(x_1), ..., f(x_n)$.

Исходя из того, что $f(x)$ является стационарным гауссовым процессом с $m = 0$, можно показать, что предсказание задается следующей формулой:

$$p\left(f(x) \mid \mathbf{f}\right) = \frac{p\left(f(x), f(x_1), ..., f(x_n)\right)}{p\left(f(x_1), ..., f(x_n)\right)} = \frac{\mathcal{N}\left(f(x), f(x_1), ..., f(x_n) \mid 0, \tilde{\mathbf{C}}\right)}{\mathcal{N}\left(f(x_1), ..., f(x_n) \mid 0, \mathbf{C}\right)},$$

где с помощью $\mathcal{N}\left(f(x), f(x_1), ..., f(x_n) \mid 0, \tilde{\mathbf{C}}\right)$ мы обозначили нормальное распределение, рассчитанное на точках с нулевым средним и ковариационной матри-

цей $\tilde{\mathbf{C}}$ размерности, потому что у нас в числителе $n+1$ точек. Математический вывод в некоторой степени запутан и основывается на нескольких теоремах, таких как теорема Байеса.

Для получения дополнительной информации вы можете обратиться к (расширенному) объяснению Чонг Б. До (Chuong B. Do) в работе "Gaussian Processes" ("Гауссовы процессы") (2007), доступной по адресу **https://goo.gl/cEPYwX**, в которой все подробно объясняется. Для понимания того, что такое байесова оптимизация, можно просто воспользоваться формулой без математического вывода. Матрица $\mathbf{C}$ будет иметь размерность $n \times n$, потому что у нас в знаменателе только n точек.

Мы имеем

$$\mathbf{C} = \begin{bmatrix} K(0) & K(x_1 - x_2) & \cdots \\ K(x_2 - x_1) & \ddots & \vdots \\ \vdots & \cdots & K(0) \end{bmatrix}$$

и ковариационную матрицу

$$\tilde{\mathbf{C}} = \begin{bmatrix} K(0) & \mathbf{k}^T \\ \mathbf{k} & \mathbf{C} \end{bmatrix},$$

где мы определили

$$\mathbf{k} = \begin{bmatrix} K(x - x_1) \\ \vdots \\ K(x - x_n) \end{bmatrix}.$$

Можно показать[1], что отношение двух нормальных распределений также является нормальным распределением, поэтому мы можем написать:

$$p(f(x)|\mathbf{f}) = \mathcal{N}(f(x)|\mu, \sigma^2),$$

где

$$\mu = \mathbf{k}^T \mathbf{C}^{-1} \mathbf{f};$$
$$\sigma^2 = K(0) - \mathbf{k}^T \mathbf{C}^{-1} \mathbf{k}.$$

Математический вывод точной формы для μ и σ довольно длинный и выходит за рамки данной книги. В сущности, теперь мы знаем, что в среднем неизвестная функция примет значение μ в x с дисперсией σ. Давайте теперь посмотрим, как этот метод действительно работает на практике на языке Python. Сначала определим ядро $K(x)$.

---

[1] Напомним, что нормальное распределение имеет экспоненциальную форму, а отношение двух экспонент по-прежнему является экспоненциальным.

```
def K(x, l, sigm = 1):
    '''Радиальная базисная функция'''
    return sigm**2*np.exp(-1.0/2.0/l**2*(x**2))
```

Смоделируем неизвестную функцию, взяв простую функцию:

$$f(x) = x^2 - x^3 + 3 + 10x + 0,07x^4,$$

которая реализуется как:

```
def f(x):
    '''Неизвестная функция'''
    return x**2-x**3+3+10*x+0.07*x**4
```

Рассмотрим функцию на интервале (0, 12). На рис. 7.12 показано, как эта функция выглядит.

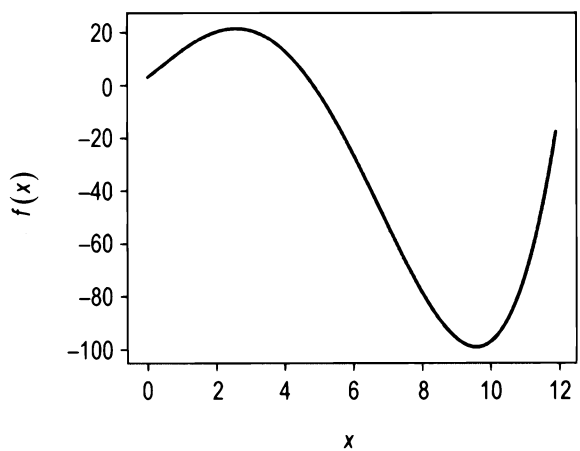


РИС. 7.12. График неизвестной тестовой функции, как описано в тексте

Давайте сначала построим вектор **f** с пятью точками, как показано ниже:

```
randompoints = np.random.random([5])*12.0
f_ = f(randompoints)
```

где мы использовали посев 42 для случайных чисел: `np.random.seed(42)`. На рис. 7.13 показаны случайные точки, отмеченные на графике крестиками.

Описанный выше метод можно применить с помощью следующего фрагмента кода:

```
xsampling = np.arange(0,14,0.2)
ybayes_ = []
sigmabayes_ = []

for x in xsampling:
    f1 = f(randompoints)
    sigm_ = np.std(f1)**2
    f_ = (f1-np.average(f1))
```

```
    k = K(x-randompoints, 2 , sigm_)

    C = np.zeros([randompoints.shape[0], randompoints.shape[0]])

    # Ковариационная матрица
    Ctilde = np.zeros([randompoints.shape[0]+1, randompoints.shape[0]+1])

    for i1,x1_ in np.ndenumerate(randompoints):
        for i2,x2_ in np.ndenumerate(randompoints):
            C[i1,i2] = K(x1_-x2_, 2.0, sigm_)

    Ctilde[0,0] = K(0, 2.0, sigm_)
    Ctilde[0,1:randompoints.shape[0]+1] = k.T
    Ctilde[1:,1:] = C
    Ctilde[1:randompoints.shape[0]+1,0] = k

    mu = np.dot(np.dot(np.transpose(k), np.linalg.inv(C)), f_)
    sigma2 = K(0, 2.0,sigm_)- np.dot(np.dot(np.transpose(k),
                                    np.linalg.inv(C)), k)
    ybayes.append(mu)
    sigmabayes_.append(np.abs(sigma2))

ybayes = np.asarray(ybayes_)+np.average(f1)
sigmabayes = np.asarray(sigmabayes_)
```

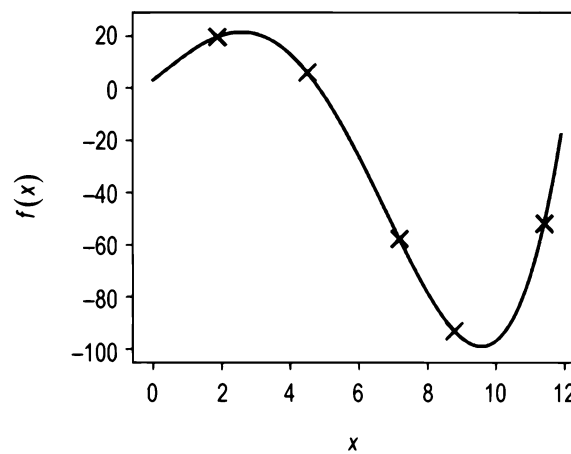


РИС. 7.13. График неизвестной функции. Крестики отмечают случайные точки, выбранные в тексте

Теперь потратьте некоторое время на то, чтобы в нем разобраться. В списке `ybayes` мы найдем значения $\mu(x)$, вычисленные на значениях, содержащихся в массиве `xsampling`. Вот несколько советов, которые помогут разобраться в этом фрагменте кода.

- Мы обходим интервал точек `x` в цикле, в котором мы хотим оценить нашу функцию, используя циклическую конструкцию `for x in xsampling:`.
- Мы строим векторы **k** и **f** с помощью инструкций для каждого элемента векторов `k = K(x-randompoints, 2 , sigm_)` и `f1 = f(randompoints)`. Для ядра мы выбрали заданное в функции значение параметра `l`, равное 2. Из вектора `f1` мы вычли среднее и получили $m(x) = 0$ для того, чтобы иметь возможность применить формулы в том виде, в каком они выведены.
- Мы строим матрицы **C** и $\tilde{\mathbf{C}}$.
- Мы вычисляем μ с помощью `mu = np.dot(np.dot(np.transpose(k), np.linalg.inv(C)), f_)` и стандартное отклонение.
- В конце мы повторно применяем все выполненные преобразования для того, чтобы сделать процесс стационарным, но в обратном порядке, снова добавив среднее от $f(x)$ к оцениваемой суррогатной функции

```
ybayes = np.asarray(ybayes_)+np.average(f1)
```

На рис. 7.14 показано, как этот метод работает. Пунктирная линия — это предсказанная функция, полученная путем построения $\mu(x)$, как вычислено в исходном коде, когда в нашем распоряжении имеется пять точек ($n = 5$). Серый участок — это область между оцениваемой функцией и $\pm\sigma$.

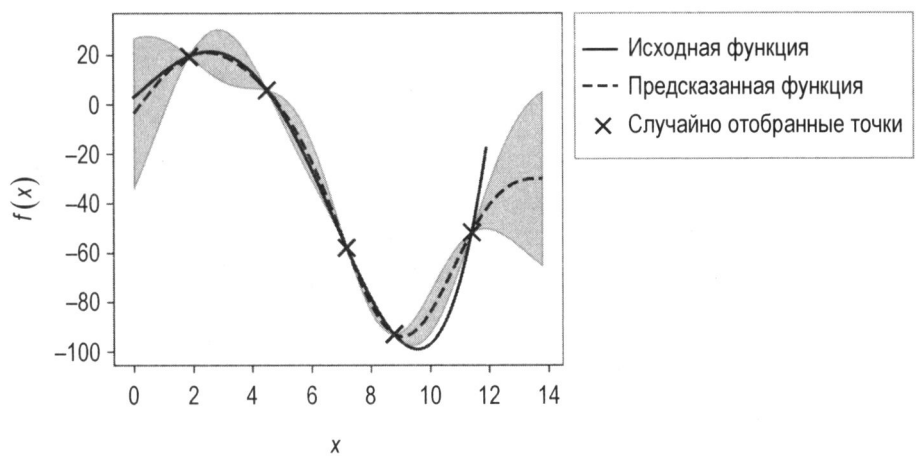


РИС. 7.14. Предсказанная функция (пунктирная линия), рассчитанная путем вычисления $\mu(x)$. Серый участок — это область между оцениваемой функцией и $\pm\sigma$

С учетом тех немногих точек, которые у нас есть, это неплохой результат. Теперь имейте в виду, что для получения разумной аппроксимации вам все же понадобится несколько точек. На рис. 7.15 показан результат для случая, если в нашем распоряжении только две точки. Результат не так хорош. Серый участок — это область между оцениваемой функцией и $\pm\sigma$. Видно, что чем дальше мы находимся от

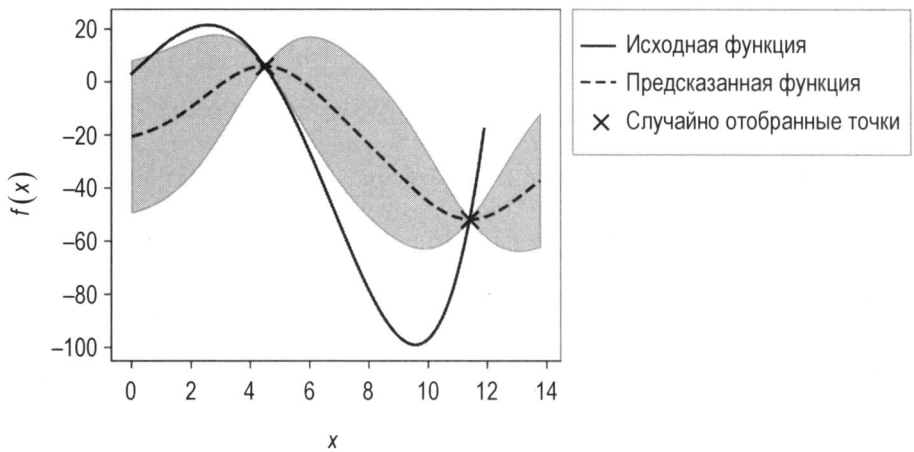


РИС. 7.15. Предсказанная функция (пунктирная линия), когда в нашем распоряжении только две точки. Серый участок — это область между оцениваемой функцией и $\pm\sigma$

имеющихся точек, тем выше неопределенность или дисперсия предсказанной функции.

Давайте остановимся на секунду и подумаем, зачем мы все это делаем. Идея состоит в том, чтобы найти так называемую суррогатную функцию $\tilde{f}$, которая аппроксимирует нашу функцию f и имеет свойство

$$\max \tilde{f} \approx \max f.$$

Эта суррогатная функция должна иметь еще одно очень важное свойство: ее оценивание должно обходиться дешево. Благодаря этому мы можем легко найти максимум функции $\tilde{f}$, и, используя указанное выше свойство, мы будем иметь максимум функции f, которая, по гипотезе, является дорогостоящей.

Но, как вы видели, бывает очень трудно узнать, имеем ли мы достаточное число точек для того, чтобы найти правильное значение максимума. В конце концов, по определению мы понятия не имеем, как выглядит наша черно-ящичная функция. Так как же тогда решить эту задачу?

Главная идея метода может быть описана следующим алгоритмом:

1. Мы начинаем с малого числа образцов точек, отобранных случайно (насколько их должно быть мало, зависит от вашей задачи).
2. Мы используем это множество точек для получения суррогатной функции, как описано выше.
3. Мы добавляем во множество дополнительную точку с помощью специального метода, который будет рассмотрем позже, и переоцениваем суррогатную функцию.
4. Если максимум, который мы находим с помощью суррогатной функции, продолжает меняться, мы будем продолжать добавлять точки, как на шаге 3, до тех

пор, пока максимум больше не будет изменяться, либо у нас не закончится время или бюджет и мы не сможем выполнять какое-либо дальнейшее оценивание.

Если метод, который был упомянут в шаге 3, будет достаточно умен, то мы сможем найти максимум относительно быстро и точно.

Функция обнаружения

Но как выбрать дополнительные точки, которые были упомянуты в шаге 3 в предыдущем разделе? Идея состоит в том, чтобы использовать так называемую функцию обнаружения (acquisition function). Ее алгоритм работает таким образом:

1. Мы выбираем функцию (и вскоре мы увидим несколько возможностей), называемую функцией обнаружения.
2. Затем в качестве дополнительной точки x мы выбираем ту, в которой функция обнаружения имеет максимум.

Можно использовать несколько функций обнаружения. Здесь будет описана только одна, которую мы будем использовать для того, чтобы увидеть, как работает этот метод, но есть несколько других, которые вы, возможно, захотите проверить, такие как энтропийный поиск, вероятность улучшения, ожидаемое улучшение и верхняя доверительная граница (upper confidence bound, UCB).

Верхняя доверительная граница

В литературе можно найти два варианта этой функции обнаружения. Данную функцию можно написать как

$$a_{\text{UCB}}(x) = \mathbb{E}\tilde{f}(x) + \eta\sigma(x),$$

где через $\mathbb{E}\tilde{f}(x)$ мы обозначили "ожидаемое" значение (математическое ожидание) суррогатной функции над интервалом x, который мы имеем в задаче. Ожидаемое значение — это не что иное, как среднее значение функции в заданном интервале x. $\sigma(x)$ — это дисперсия суррогатной функции, которую мы вычисляем с помощью нашего метода в точке x. Новая отбираемая нами точка находится там, где $a_{\text{UCB}}(x)$ является максимальной. $\eta > 0$ является компромиссным параметром. Данная функция обнаружения в сущности отбирает точки, где дисперсия самая высокая. Пересмотрите рис. 7.15. Данный метод отбирает точки, в которых дисперсия выше, поэтому точки лежат как можно дальше от имеющихся у нас точек. Благодаря этому, аппроксимация демонстрирует тенденцию становиться все лучше и лучше.

Приведем еще один вариант функции обнаружения $\tilde{a}_{\text{UCB}}(x)$:

$$\tilde{a}_{\text{UCB}}(x) = \tilde{f}(x) + \eta\sigma(x).$$

На этот раз функция обнаружения будет находить оптимальное соотношение между выбором точек вокруг максимума суррогатной функции и точками, где ее дис-

персия является наибольшей. Этот второй метод лучше всего работает в быстром отыскании хорошей аппроксимации максимума, в то время как первый, как правило, дает хорошую аппроксимацию по всему интервалу x. В следующем разделе мы увидим, как эти методы работают.

Пример

Начнем с тригонометрической функции, описанной в начале главы, и рассмотрим интервал x между [0; 40]. Наша цель — найти его максимум и аппроксимировать функцию. Для того чтобы облегчить программирование, давайте определим две функции: одну для оценивания суррогатной функции и другую для оценивания новой точки. Для оценивания суррогатной функции можно применить следующую Python-функцию:

```
def get_surrogate(randompoints):
    ybayes_ = []
    sigmabayes_ = []

    for x in xsampling:
        f1 = f(randompoints)
        sigm_ = np.std(f_)**2
        f_ = (f1-np.average(f1))
        k = K(x-randompoints, 2.0, sigm_)

        C = np.zeros([randompoints.shape[0], randompoints.shape[0]])
        Ctilde = np.zeros([randompoints.shape[0]+1,
                           randompoints.shape[0]+1])
        for i1,x1_ in np.ndenumerate(randompoints):
            for i2,x2_ in np.ndenumerate(randompoints):
                C[i1,i2] = K(x1_-x2_, 2.0, sigm_)

        Ctilde[0,0] = K(0, 2.0)
        Ctilde[0,1:randompoints.shape[0]+1] = k.T
        Ctilde[1:,1:] = C
        Ctilde[1:randompoints.shape[0]+1,0] = k

        mu = np.dot(np.dot(np.transpose(k), np.linalg.inv(C)), f_)
        sigma2 = K(0, 2.0, sigm_) - np.dot(np.dot(np.transpose(k),
                                           np.linalg.inv(C)), k)
        ybayes_.append(mu)
        sigmabayes_.append(np.abs(sigma2))

    ybayes = np.asarray(ybayes_)+np.average(f1)
    sigmabayes = np.asarray(sigmabayes_)

    return ybayes, sigmabayes
```

Эта функция имеет тот же исходный код, который уже рассматривался в примере в предыдущих разделах, но она упакована в Python-функцию, которая возвращает суррогатную функцию, содержащуюся в массиве `ybayes`, и σ^2, содержащееся в массиве `sigmabayes`. Кроме того, нам требуется функция, которая оценивает новые точки, используя функцию обнаружения $a_{\text{UCB}}(x)$. Она легко реализуется с помощью Python-функции:

```
def get_new_point(ybayes, sigmabayes, eta):
    idxmax = np.argmax(np.average(ybayes)+eta*np.sqrt(sigmabayes))
    newpoint = xsampling[idxmax]

    return newpoint
```

Для того чтобы сделать всё еще проще, определим массив, содержащий все значения x, которые мы хотим иметь в самом начале вне функций. Начнем с шести случайно отобранных точек. Давайте проверим работу метода, начав с некоторых определений.

```
xmax = 40.0
numpoints = 6
xsampling = np.arange(0,xmax,0.2)
eta = 1.0

np.random.seed(8)
randompoints1 = np.random.random([numpoints])*xmax
```

В массиве `randompoints1` будут первые шесть случайно отобранных точек. Мы можем легко получить суррогатную функцию нашей функции с помощью инструкции:

```
ybayes1, sigmabayes1 = get_surrogate(randompoints1)
```

На рис. 7.16 приведен результат. Пунктирная линия — это функция обнаружения $a_{\text{UCB}}(x)$, нормализованная для вписывания в график.

Данная суррогатная функция еще не очень хороша, потому что у нас недостаточно точек, и высокая дисперсия (серый участок) делает это заметным. Единственный участок, который хорошо аппроксимирован, — это участок $x \gtrsim 35$. Видно, что функция обнаружения является большой тогда, когда суррогатная функция не аппроксимирует черно-ящичную функцию хорошо, и малой тогда, когда она это делает, как например, для $x \gtrsim 35$. Поэтому интуитивный отбор точек там, где $a_{\text{UCB}}(x)$ является максимальной, эквивалентен отбору точек там, где функция менее хорошо аппроксимирована, или, в математических терминах, где дисперсия выше. Для сравнения на рис. 7.17 представлен тот же график, что и на рис. 7.16, но с функцией обнаружения $\tilde{a}_{\text{UCB}}(x)$ и с $\eta = 3{,}0$.

Как вы видите, $\tilde{a}_{\text{UCB}}(x)$ демонстрирует тенденцию иметь максимум вокруг максимума суррогатной функции. Имейте в виду, что если η является большим, то мак-

симум функции обнаружения будет сдвигаться в сторону участков с высокой дисперсией. Но эта функция обнаружения, как правило, находит "некий" максимум немного быстрее. Слово "некий" используется намеренно, потому что все зависит от того, где находится максимум суррогатной функции, а не максимум черно-ящичной функции.

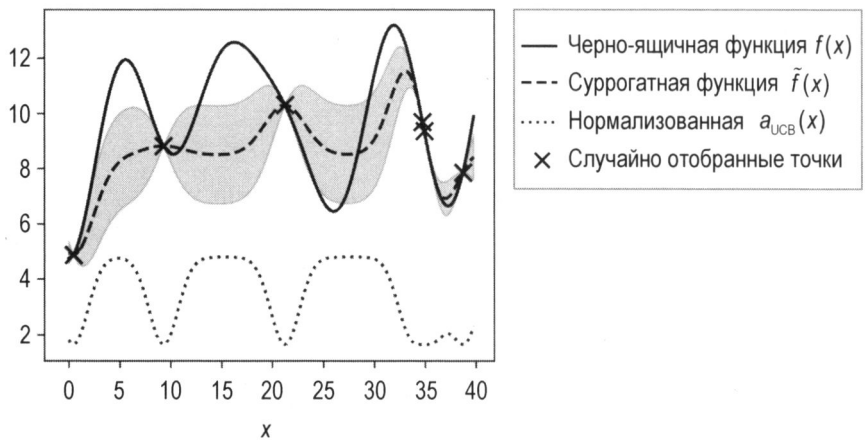


РИС. 7.16. Вид черно-ящичной функции $f(x)$ (сплошная линия), произвольно отобранных точек (отмеченных крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $a_{UCB}(x)$ (пунктирная линия), сдвинутых для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$

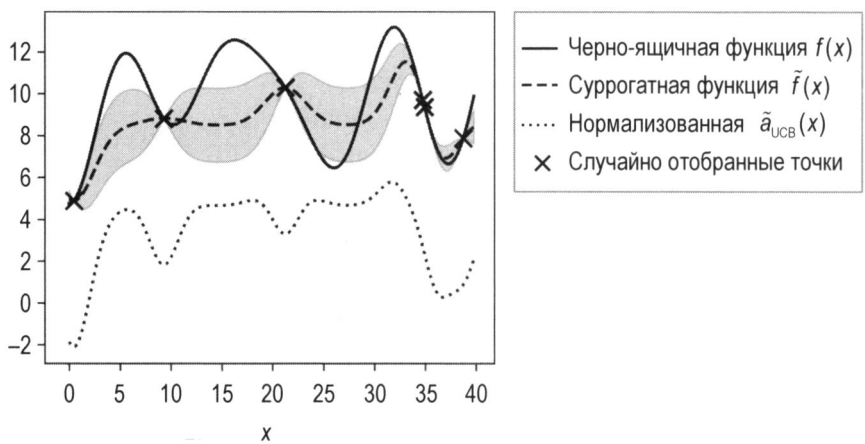


РИС. 7.17. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек (отмеченных крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутых для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Добавлена функция обнаружения $\tilde{a}_{UCB}(x)$ с $\eta = 3,0$

Теперь посмотрим, что происходит при использовании $a_{\text{UCB}}(x)$ с $\eta = 1,0$. Для первой дополнительной точки мы должны лишь выполнить следующие три строки кода:

```
newpoint = get_new_point(ybayes1, sigmabayes1, eta)
randompoints2 = np.append(randompoints1, newpoint)
ybayes2, sigmabayes2 = get_surrogate(randompoints2)
```

Для простоты вместо того, чтобы создавать список, каждый массив для каждого шага назван по-разному. Но, как правило, вы должны делать эти итерации автоматически. На рис. 7.18 представлен результат с дополнительной точкой, отмеченной черным кружком.

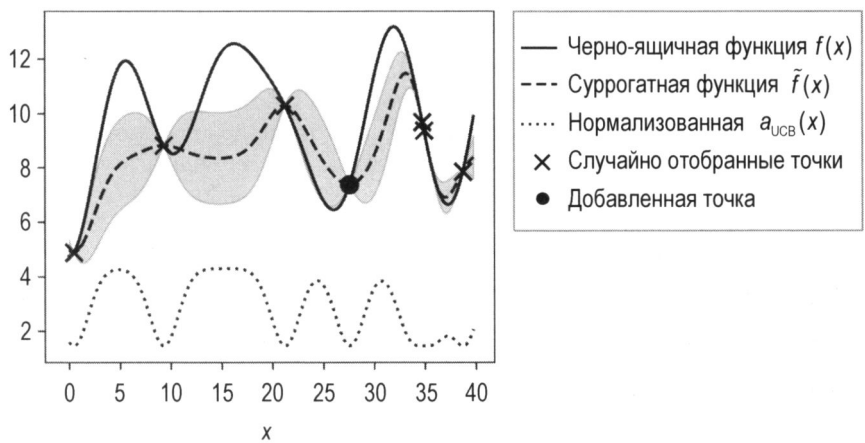


РИС. 7.18. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек (отмеченных крестиками) и с новой отобранной точкой вокруг $x \approx 27$ (отмеченной кружком), суррогатной функции (пунктирная линия) и функции обнаружения $a_{\text{UCB}}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x) + \sigma(x)$ и $\tilde{f}(x) - \sigma(x)$. То же что и на предыдущем рисунке, но с дополнительной точкой, отмеченной черным кружком

Новая точка находится вблизи $x \approx 27$. Давайте продолжим добавлять точки. На рис. 7.19 представлены результаты после добавления пяти точек.

Обратите внимание на пунктирную линию. Теперь суррогатная функция довольно хорошо аппроксимирует черно-ящичную функцию, в особенности вокруг реального максимума. Используя эту суррогатную функцию, мы можем найти очень хорошую аппроксимацию нашей исходной функции всего за 11 оцениваний! Учтите, что у нас нет никакой дополнительной информации, кроме 11 оцениваний.

Теперь давайте посмотрим, что происходит с функцией обнаружения $\tilde{a}_{\text{UCB}}(x)$, и проверим, как быстро мы можем найти максимум. В этом случае мы будем использовать $\eta = 3,0$ с целью получить более оптимальный баланс между максимумом суррогатной функции и ее дисперсией. На рис. 7.20 приведен результат уже

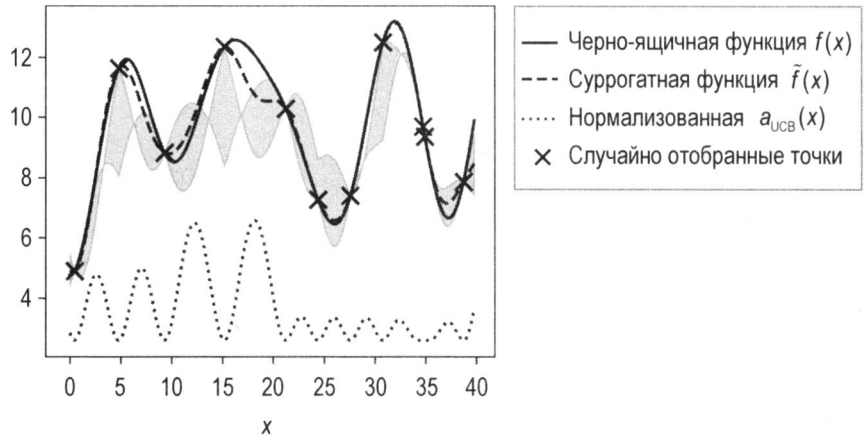


РИС. 7.19. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек с шестью только что отобранными точками (отмеченными крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Результаты после добавления пяти точек

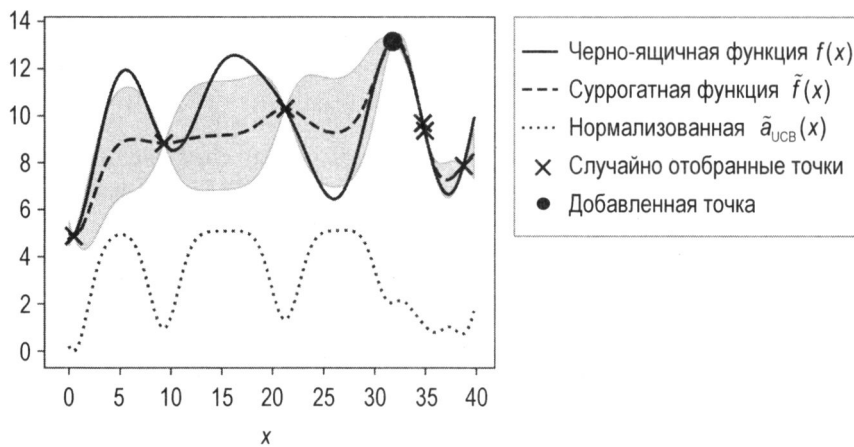


РИС. 7.20. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек с дополнительно отобранными точками (отмеченными крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Результат после добавления одной дополнительной точки, отмеченной черным кружком

после добавления одной дополнительной точки, отмеченной черным кружком. У нас уже есть достаточно хорошая аппроксимация реального максимума!

Теперь добавим еще одну точку. На рис. 7.21 видно, что дополнительная точка теперь по-прежнему находится близко к максимуму, но смещена в направлении участка с высокой дисперсией в районе 30.

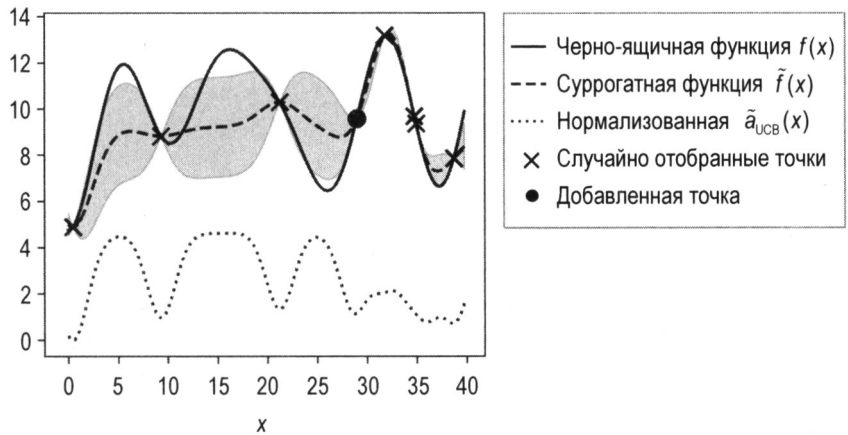


РИС. 7.21. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек с дополнительно отобранными точками (отмеченными крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Показано смещение точки в направлении участка с высокой дисперсией в районе 30

Если мы решим сделать η меньше, то точка будет ближе к максимуму, а если решим сделать этот параметр больше, то точка будет ближе к точке с наивысшей дисперсией, между 25 и примерно 32. Теперь давайте добавим еще одну точку и посмотрим, что произойдет. На рис. 7.22 показано, как этот метод теперь выбирает точку, близкую к еще одному участку с высокой дисперсией, между 10 и примерно 22, снова отмеченную черным кругом.

И наконец, данный метод уточняет максимальный участок вокруг 15, как видно на рис. 7.23, добавляя точку вокруг 14, отмеченную черным кружком.

Приведенное выше обсуждение и сравнение поведения двух типов функций обнаружения должно было прояснить, как выбирать правильную функцию обнаружения, в зависимости от того, какую стратегию вы хотите применить для аппроксимирования вашей черно-ящичной функции.

ПРИМЕЧАНИЕ. Различные типы функций обнаружения будут давать разные стратегии в аппроксимировании черно-ящичной функции. Например, $a_{UCB}(x)$ добавит точки в участках с наивысшей дисперсией, а $\tilde{a}_{UCB}(x)$ добавит точки, отыскивая баланс, регулируемый параметром η, между максимумом суррогатной функции и участками с высокой дисперсией.

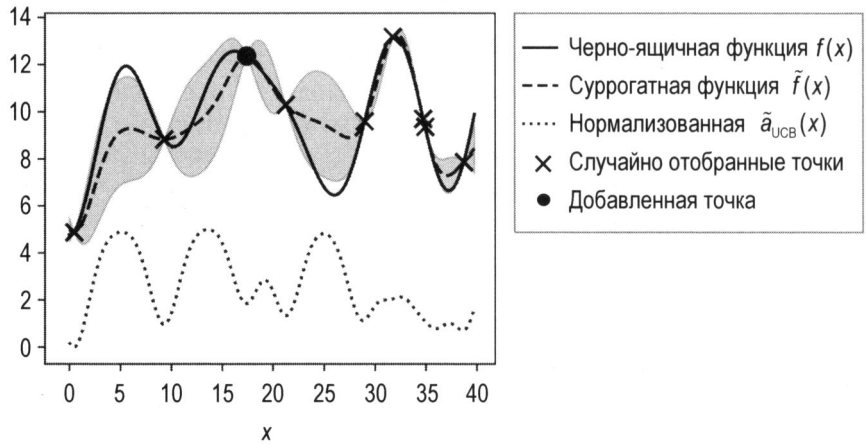


РИС. 7.22. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек (отмеченных крестиками) и дополнительно отобранной точки (отмеченной черным кружком), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Добавление другой точки между 10 и примерно 22, снова отмеченной черным кругом

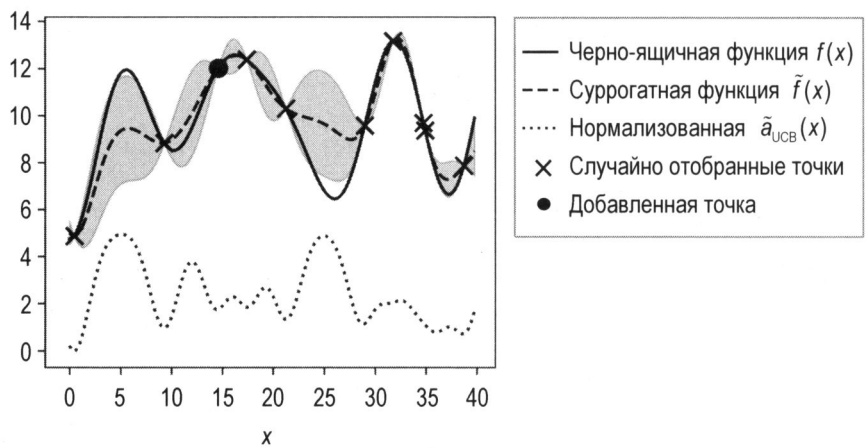


РИС. 7.23. Вид черно-ящичной функции $f(x)$ (сплошная линия), случайно отобранных точек с дополнительно отобранными точками (отмеченными крестиками), суррогатной функции (пунктирная линия) и функции обнаружения $\tilde{a}_{UCB}(x)$ (пунктирная линия), сдвинутой для вписывания в график. Серый участок — это область, содержащаяся между линиями $\tilde{f}(x)+\sigma(x)$ и $\tilde{f}(x)-\sigma(x)$. Результат после добавления точки вокруг 14, отмеченной черным кружком

Анализ всех типов функций обнаружения выходит за рамки данной книги. Для получения достаточного опыта и понимания того, как работают и ведут себя разные функции обнаружения, потребуется много исследований и чтения опубликованных работ.

Если вы хотите использовать байесову оптимизацию со своей моделью TensorFlow, то вам не нужно разрабатывать этот метод полностью с нуля. Вы можете попробовать библиотеку GPflowOpt, описание которой дается в статье Николаса Кнадда (Nicolas Knudde) и соавт. "GPflowOpt: A Bayesian Optimization Library using TensorFlow" ("GPflowOpt: библиотека байесовой оптимизации с использованием TensorFlow") на **https://goo.gl/um4LSy** или **arXiv.org**.

Отбор образцов на логарифмической шкале

Существует еще одна маленькая тонкость, которую следовало бы обсудить. Иногда вы оказываетесь в ситуации, когда хотите опробовать большой диапазон возможных значений параметра, но по опыту знаете, что лучшее его значение, вероятно, находится в определенном поддиапазоне. Предположим, для вашей модели вы хотите найти наилучшее значение темпа заучивания и решаете протестировать значения от 10^{-4} до 1, но вы знаете или, по крайней мере, подозреваете, что ваше лучшее значение, вероятно, лежит между 10^{-3} и 10^{-4}. Теперь предположим, что вы работаете с решеточным поиском и отбираете 1000 точек. Вы можете подумать, что у вас точек достаточно, но вы получите:

- 0 точек между 10^{-4} и 10^{-3};
- 8 точек между 10^{-3} и 10^{-2};
- 89 точек между 10^{-1} и 10^{-2};
- 899 точек между 1 и 10^{-1}.

Вы получите намного больше точек в менее интересных поддиапазонах и ноль точек в наиболее предпочтительном поддиапазоне. На рис. 7.24 показано распределение точек. Обратите внимание, что по оси *x* использована логарифмическая шкала. Из рисунка ясно видно, что гораздо больше точек получено для более крупных значений темпа заучивания.

Вы, вероятно, хотите отбирать образцы гораздо более тонким способом с меньшими значениями темпа заучивания, чем с бóльшими. Для этого всего лишь нужно отбирать их на логарифмической шкале. Поясним. Основная идея состоит в том, что требуется отобрать одинаковое число точек между 10^{-4} и 10^{-3}, 10^{-3} и 10^{-2}, 10^{-2} и 10^{-1}, 10^{-1} и 1. Для этого можно использовать приведенную далее строку кода. Сначала отобрать случайные числа между 0 и –4, т. е. минусом абсолютного значения наибольшего числа в степени 10, которое у вас есть.

```
r = - np.arange(0,1,0.001)*4.0
```

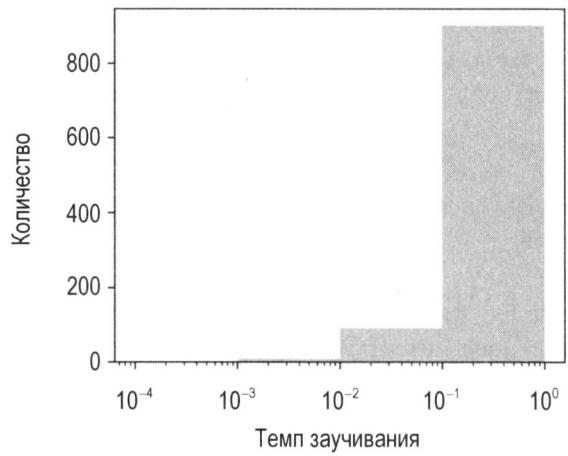


РИС. 7.24. Распределение точек, отобранных с помощью решеточного поиска на логарифмической шкале *x*

Затем массив с отобранными точками можно создать с помощью

```
alpha = 10**r
```

На рис. 7.25 видно, что распределения точек, содержащихся в массиве точек `alpha`, являются совершенно плоскими, как мы и хотели.

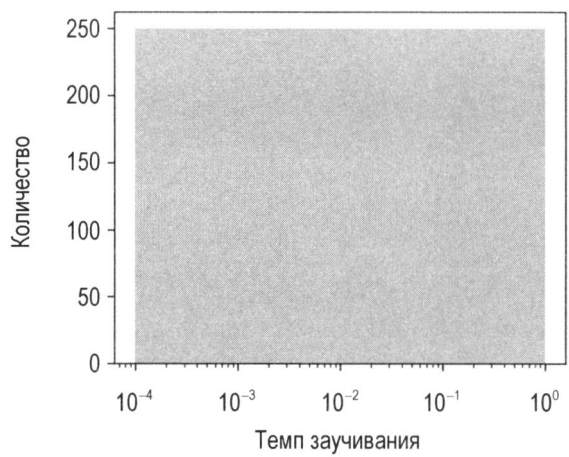


РИС. 7.25. Распределение 1000 точек, отобранных с помощью решеточного поиска на логарифмической шкале *x*, с помощью модифицированного метода отбора

С помощью следующего ниже Python-кода вы получаете 250 точек в каждом участке, что можно легко проверить, для интервала от 10^{-3} до 10^{-4} (для других интервалов просто измените числа в коде):

```
print(np.sum((alpha <= 1e-3) & (alpha > 1e-4)))
```

Теперь вы видите, что у вас одинаковое число точек между разными степенями 10. С помощью этой хитрости вы также можете обеспечить достаточное число точек в участке выбранного вами интервала, где, в противном случае, вы не получите почти никаких точек. Напомним, что в этом примере с 1000 точками стандартным методом мы получаем ноль точек между 10^{-3} и 10^{-4}. Этот интервал для темпа заучивания наиболее интересен, поэтому в целях оптимизации модели вы хотели бы иметь достаточное число точек в этом интервале. Обратите внимание, что то же самое относится и к случайному поиску. Он работает точно так же.

Гиперпараметрическая настройка с набором данных Zalando

В качестве конкретного примера того, как работает настройка гиперпараметров, давайте применим наши знания, полученные в простом случае. Как обычно, начнем с данных. Мы будем использовать набор данных Zalando из *главы 3*. Давайте быстро загрузим и подготовим данные, а затем рассмотрим настройку.

Сначала загрузите необходимые библиотеки.

```
import pandas as pd
import numpy as np
import tensorflow as tf

%matplotlib inline

import matplotlib
import matplotlib.pyplot as plt

from random import *
```

Вам потребуются необходимые CSV-файлы в папке, в которой находится ваш блокнот Jupyter. О том, как их получить, *см. в главе 3*. Если файлы находятся в той же папке, где и блокнот, то вы можете загрузить данные с помощью инструкций:

```
data_train = pd.read_csv('fashion-mnist_train.csv', header = 0)
data_dev = pd.read_csv('fashion-mnist_test.csv', header = 0)
```

Напомним, что у нас 60 000 наблюдений в тренировочном наборе данных и 10 000 в рабочем наборе данных. Например, печать формы массива `data_train` с помощью инструкции

```
print(data_train.shape)
```

даст вам (60 000, 785). Напомним, что один из столбцов в массиве `data_train` содержит метки, а 784 — это значения пикселов изображения с оттенками серого (изображения имеют размер 28×28 пикселов). Необходимо отделить метки от признаков (пиксельные значения с оттенками серого), затем нам нужно реформировать массивы.

```
labels = data_train['label'].values.reshape(1, 60000)
labels_ = np.zeros((60000, 10))
labels_[np.arange(60000), labels] = 1
labels_ = labels_.transpose()
train = data_train.drop('label', axis=1).transpose()
```

Проверка размерностей:

```
print(labels_.shape)
print(train.shape)
```

даст нам

```
(10, 60000)
(784, 60000)
```

как того и хотелось. (Подробное обсуждение вопроса подготовки этого набора данных *см. в главе 3*.) Разумеется, мы должны проделать то же самое для рабочего набора данных.

```
labels_dev = data_test['label'].values.reshape(1, 10000)
labels_dev_ = np.zeros((10000, 10))
labels_dev_[np.arange(10000), labels_test] = 1
labels_dev_ = labels_test_.transpose()
dev = data_dev.drop('label', axis=1).transpose()
```

Теперь давайте нормализуем признаки и преобразуем все в массив NumPy.

```
train = np.array(train / 255.0)
dev = np.array(dev / 255.0)
labels_ = np.array(labels_)
labels_dev_ = np.array(labels_dev_)
```

Мы подготовили необходимые данные. Теперь перейдем к модели. Начнем с чего-нибудь попроще. В качестве метрики в этом примере мы будем использовать точность, поскольку набор данных сбалансирован. Давайте рассмотрим сеть только с одним слоем и посмотрим, какое число нейронов дает нам наилучшую точность. Нашим гиперпараметром в этом примере будет число нейронов в скрытом слое. В сущности, нам придется строить новую сеть для каждого значения гиперпараметра (число нейронов в скрытом слое) и тренировать ее. Нам потребуются две функции: одна для построения сети и одна для ее тренировки. Для построения модели можно определить следующую Python-функцию:

```
def build_model(number_neurons):
  n_dim = 784
  tf.reset_default_graph()

  # Число нейронов в слоях
  n1 = number_neurons  # Число нейронов в скрытом слое
  n2 = 10              # Число нейронов в выходном слое
```

```
  cost_history = np.empty(shape=[1], dtype = float)
   learning_rate = tf.placeholder(tf.float32, shape=())

  X = tf.placeholder(tf.float32, [n_dim, None])
  Y = tf.placeholder(tf.float32, [10, None])
  W1 = tf.Variable(tf.truncated_normal([n1, n_dim], stddev=.1))
  b1 = tf.Variable(tf.constant(0.1, shape = [n1,1]) )
  W2 = tf.Variable(tf.truncated_normal([n2, n1], stddev=.1))
  b2 = tf.Variable(tf.constant(0.1, shape = [n2,1]))

  # Построим нашу сеть...
  Z1 = tf.nn.relu(tf.matmul(W1, X) + b1) # n1 x n_dim * n_dim x n_obs = n1 x
n_obs
  Z2 = tf.matmul(W2, Z1) + b2 # n2 x n1 * n1 * n_obs = n2 x n_obs
  y_ = tf.nn.softmax(Z2,0) # n2 x n_obs (10 x None)

  cost = - tf.reduce_mean(Y * tf.log(y_)+(1-Y) * tf.log(1-y_))
  optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

  init = tf.global_variables_initializer()

  return optimizer, cost, y_, X, Y, learning_rate
```

Вы должны уже разбираться в этой функции, т. к. этот исходный код уже использовался несколько раз в книге. Эта функция имеет входной параметр: `number_neurons`, который будет содержать, как указывает его имя, число нейронов в скрытом слое. Но есть небольшая разница: функции возвращают тензоры, на которые мы должны ссылаться во время тренировки, например, когда во время тренировки мы хотим оценить стоимостной тензор `cost`. Если мы не вернем их вызывающему коду, то они будут видны только внутри этой функции, и мы не сможем натренировать эту модель. Функция, которая будет тренировать модель, выглядит следующим образом:

```
def model(minibatch_size, training_epochs, features, classes,
          logging_step=100, learning_r=0.001, number_neurons=15):
  opt, c, y_, X, Y, learning_rate = build_model(number_neurons)

  sess = tf.Session()
  sess.run(tf.global_variables_initializer())

  cost_history = []

  for epoch in range(training_epochs+1):
    for i in range(0, features.shape[1], minibatch_size):
      X_train_mini = features[:,i:i + minibatch_size]
      y_train_mini = classes[:,i:i + minibatch_size]
```

```
        sess.run(opt, feed_dict = {X:X_train_mini, Y:y_train_mini,
                                   learning_rate: learning_r})
        cost_ = sess.run(c, feed_dict={X:features, Y:classes,
                                       learning_rate: learning_r})
        cost_history = np.append(cost_history, cost_)

        if (epoch % logging_step == 0):
          print("Достигнута эпоха",epoch,"стоимость J =", cost_)

    correct_predictions = tf.equal(tf.argmax(y_,0), tf.argmax(Y,0))
          accuracy = tf.reduce_mean(tf.cast(correct_predictions, "float"))
    accuracy_train = accuracy.eval({X:train, Y:labels_,
                                    learning_rate:learning_r}, session = sess)
    accuracy_dev = accuracy.eval({X:dev, Y:labels_dev_,
                                  learning_rate:learning_r}, session = sess)

    return accuracy_train, accuracy_dev, sess, cost_history
```

Вы уже несколько раз видели функцию, очень похожую на эту. Главные ее части должны быть понятными. Вы найдете в ней несколько новых вещей. Прежде всего, мы строим модель в самой функции

```
opt, c, y_, X, Y, learning_rate = build_model(number_neurons)
```

а также оцениваем точность на тренировочном и рабочем наборах данных и возвращаем значения вызывающему коду. Благодаря этому мы можем выполнять цикл для нескольких значений числа нейронов в скрытом слое и получать показатели точности. Обратите внимание, что на этот раз функция имеет дополнительный входной параметр: `number_neurons`. Необходимо передать это число функции, которая и построит модель.

Предположим, мы выбираем следующие параметры: размер мини-пакета равен 50 наблюдениям, мы выполняем тренировку в течение 100 эпох, темп заучивания равен 0,01, строим нашу модель с 15 нейронами в скрытом слое.

Затем мы выполняем модель

```
acc_train, acc_test, sess, cost_history = model(minibatch_size=50,
                                                training_epochs=100,
                                                features=train,
                                                classes=labels_,
                                                logging_step=10,
                                                learning_r=0.001,
                                                number_neurons=15)
print(acc_train)
print(acc_test)
```

Для тренировочного набора данных мы получаем точность, равную 0,75755, а для рабочего набора данных — точность, равную 0,754. Сможем ли мы добиться лучшего? Попробуем. Мы можем начать с решеточного поиска.

```
nn = [1,5,10,15,25,30, 50, 150, 300, 1000, 3000]
for nn_ in nn:
acc_train, acc_test, sess, cost_history = model(minibatch_size=50,
                                                training_epochs=50,
                                                features=train,
                                                classes=labels_,
                                                logging_step=50,
                                                learning_r=0.001,
                                                number_neurons=nn_)
print('Число нейронов:',nn_,'Тренировочная точность:', acc_train,
                            'Тестовая точность', acc_test)
```

Имейте в виду, что это займет довольно много времени. Три тысячи нейронов — довольно большое число, поэтому, в случае если вы хотите попробовать этот пример, будьте осторожны. Мы получим результаты, как показано в табл. 7.1.

Таблица 7.1. *Сводная таблица точности на тренировочном и тестовом наборах данных для разного числа нейронов*

Число нейронов	Точность на тренировочном наборе данных	Точность на тестовом наборе данных
1	0,201383	0,2042
5	0,639417	0,6377
10	0,639183	0,6348
15	0,687183	0,6815
25	0,690917	0,6917
30	0,6965	0,6887
50	0,73665	0,7369
150	0,78545	0,7848
300	0,806267	0,8067
1000	0,828117	0,8316
3000	0,8468	0,8416

Неудивительно, что большее число нейронов обеспечивают более высокую точность, без симптомов переподгонки к тренировочному набору данных, потому что точность на рабочем наборе данных почти равна точности на тренировочном наборе данных. На рис. 7.26 показан график точности на тестовом наборе данных в сопоставлении с числом нейронов в скрытом слое. Обратите внимание, что для того чтобы сделать изменения заметнее, в оси x используется логарифмическая шкала.

РИС. 7.26. Точность тестового набора данных в сопоставлении с числом нейронов в скрытом слое

Если ваша цель — достичь точности 80%, вы вполне можете остановиться здесь. Но есть несколько вещей, о которых стоит подумать. Во-первых, мы можем добиться лучшего, и во-вторых, тренировка сети с 3000 нейронами занимает довольно много времени — на моем ноутбуке примерно 35 минут. Мы должны посмотреть, сможем ли мы получить тот же результат за более короткий промежуток времени. Нам нужна модель, которая будет тренироваться как можно быстрее. Давайте попробуем немного другой подход. Поскольку мы хотим повысить быстродействие, рассмотрим модель с четырьмя слоями. Мы могли бы также настроить число слоев, но давайте в этом примере придерживаться четырех слоев и отрегулируем другие параметры. Мы постараемся найти оптимальное значение темпа заучивания, размера мини-пакета, числа нейронов в каждом слое и числа эпох. Мы будем использовать случайный поиск: случайно отберем 10 значений каждого параметра.

- Число нейронов — от 35 до 60.
- Темп заучивания — мы будем использовать поиск в логарифмической шкале между 10^{-1} и 10^{-3}.
- Размер мини-пакета — между 20 и 80.
- Число эпох — от 40 до 100.

С помощью следующего фрагмента кода мы создадим массивы с возможными значениями:

```
neurons_ = np.random.randint(low=35, high=60.0, size=(10))
r = -np.random.random([10])*3.0-1
learning_ = 10**r
mb_size_ = np.random.randint(low=20, high=80, size = 10)
epochs_ = np.random.randint(low = 40, high = 100, size = (10))
```

Обратите внимание, что мы не будем испытывать все вероятные комбинации, а рассмотрим только десять возможных комбинаций: первое значение каждого массива, второе значение каждого массива и т. д. Цель состоит в том, чтобы показать вам, насколько эффективным может быть случайный поиск всего с десятью оцениваниями! Протестировать модель можно с помощью следующего цикла:

```
for i in range(len(neurons_)):
  acc_train, acc_test, sess, cost_history =
    model_layers(minibatch_size=mb_size_[i],
                 training_epochs=epochs_[i], features=train,
                 classes=labels_, logging_step=50,
                 learning_r=learning_[i],
                 number_neurons=neurons_[i],
                 debug=False)
    print('Эпохи:', epochs_[i],
          'Число нейронов:',neurons_[i],
          'Темп заучивания:', learning_[i],
          'Размер мини-пакета',mb_size_[i],
          'Тренировочная точность:', acc_train,
          'Тестовая точность', acc_test)
```

Если выполнить этот фрагмент кода, то получится несколько комбинаций, которые заканчиваются на `nan`, и, следовательно, код дает вам точность 0,1 (в сущности, случайную, потому что у нас десять классов) и несколько хороших комбинаций. Вы обнаружите, что комбинации с 41 эпохой, 41 нейроном в каждом слое, темпом заучивания 0,0286 и мини-пакетом размером 61 наблюдение дают точность на рабочем наборе данных, равную 0,86. Неплохо, учитывая, что этот прогон занял всего 2,5 минуты, и значит, в 14 раз быстрее модели с одним слоем и 3000 нейронами и на 6% лучше. Первоначальные наивные испытания дали точность 0,75, так что с гиперпараметрической настройкой мы получили 11%-е улучшение по сравнению с первоначальной догадкой. Одиннадцатипроцентное повышение точности в глубоком обучении — это вообще-то невероятный результат. Как правило, даже 1%-е или 2%-е улучшение считается отличным результатом. Наши действия должны дать представление о том, насколько мощной может быть гиперпараметрическая настройка, если все делать правильно. Имейте в виду, что вы должны тратить на нее довольно много времени, в особенности на размышления о том, как ее проводить.

ПРИМЕЧАНИЕ. Всегда думайте о том, как вы хотите выполнить настройку гиперпараметров, используйте свой опыт или попросите помощи у кого-то, обладающего опытом. Бесполезно тратить время и ресурсы на то, чтобы испытывать комбинации параметров, о которых вы знаете, что они не будут работать. Например, лучше потратить время на тестирование очень малых темпов заучивания, чем тестировать темпы заучивания около единицы. Помните, что каждый тренировочный раунд вашей сети будет стоить времени, даже если результаты не несут никакой пользы!

Смысл этого раздела не в том, чтобы получить наилучшую модель, а в том, чтобы дать вам представление о возможной работе процесса настройки. Вы можете продолжить, пробуя разные оптимизаторы (например, Adam), рассматривая более широкие диапазоны параметров, больше комбинаций параметров и т. д.

Краткое замечание о радиальной базисной функции

Прежде чем завершить эту главу, стоит внести небольшое замечание относительно радиальной базисной функции

$$K(x, x_i) = \sigma^2 e^{-\frac{1}{2l^2}\|x - x_i\|^2}.$$

Важно понимать роль параметра l. В примерах было выбрано значение $l = 2$, но не была разъяснена причина. А причина заключается в следующем. Выбор слишком

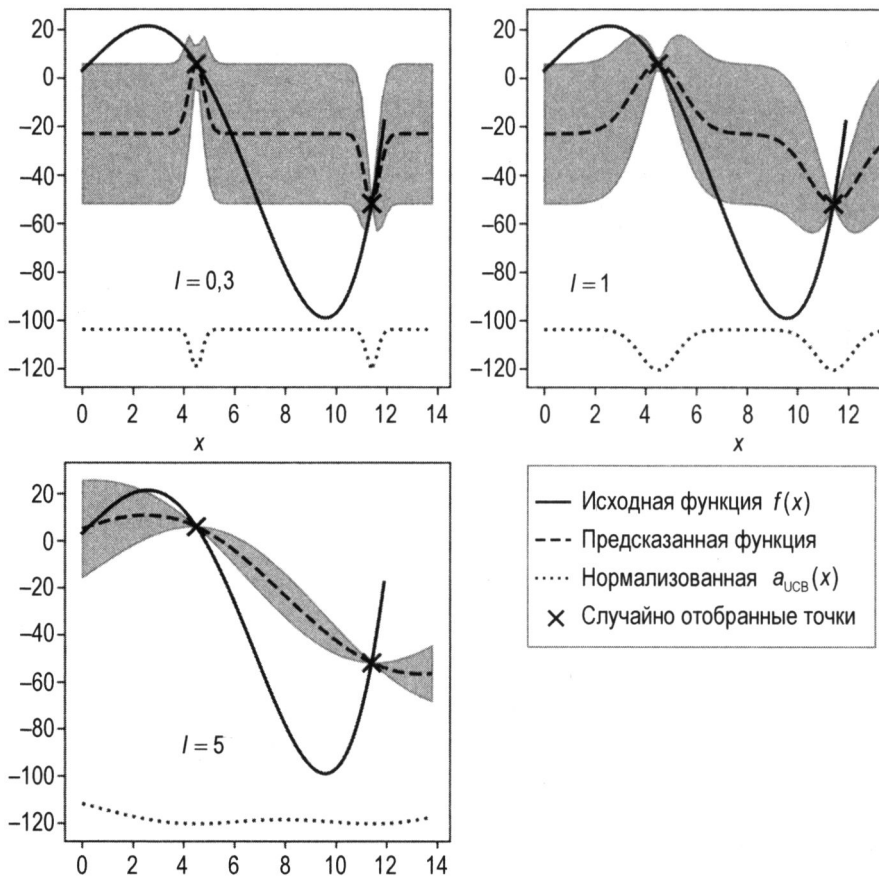


РИС. 7.27. Эффект изменения параметра *l* в радиальной базисной функции

малого значения l побуждает функцию обнаружения развивать очень узкие пики вокруг точек, которые у нас уже есть. Это можно увидеть на левом графике на рис. 7.27. Большие значения l окажут сглаживающее влияние на функцию обнаружения, как видно в центре и справа на рис. 7.27.

Обычно рекомендуется избегать слишком малых либо слишком больших значений l, что позволяет иметь дисперсию, которая плавно варьирует между известными точками, как на рис. 7.27 для $l = 1$. Наличие очень малого l делает дисперсию между точками почти постоянной и, следовательно, побуждает алгоритм почти всегда выбирать срединную точку между точками, как это можно видеть из функции обнаружения. Выбор большого l сделает дисперсию малой и, следовательно, затруднит использование некоторых функций обнаружения. Как видно из рис. 7.27, при $l = 5$ функция обнаружения является почти постоянной. Типичными значениями, как правило, являются те, которые лежат вокруг 1 или 2.

ГЛАВА 8

Сверточные и рекуррентные нейронные сети

В предыдущих главах мы рассмотрели полносвязные сети и все проблемы, возникающие во время их тренировки. Используемая нами сетевая архитектура, в которой каждый нейрон в слое соединен со всеми нейронами в предыдущем и следующем слоях, не очень хороша во многих фундаментальных задачах, таких как распознавание изображений, распознавание речи, предсказание временных рядов и многих других. Сверточные нейронные сети (convolutional neural network, CNN) и рекуррентные нейронные сети (recurrent neural network, RNN) являются передовыми архитектурами, которые используются сегодня наиболее часто. В этой главе вы рассмотрите свертку и сведение — основные строительные блоки CNN-сетей. Затем вы обратитесь к RNN-сетям, посмотрите на их работу на высоком уровне и увидите ограниченное число показательных примеров их использования. Здесь также будут представлены полные, хотя и базовые, реализации CNN- и RNN-сетей в TensorFlow. Тема CNN- и RNN-сетей слишком обширна для того, чтобы можно было ее охватить в одной главе. Поэтому мы остановимся здесь только на фундаментальных концепциях для того, чтобы вы могли увидеть особенности работы этих архитектур, но для их полного рассмотрения потребуется отдельная книга.

Ядра и фильтры

Одной из главных составляющих CNN-сетей являются фильтры — квадратные матрицы, имеющие размерность $n_{\text{к}} \times n_{\text{к}}$, где, как правило, $n_{\text{к}}$ — это малое число, например 3 или 5. Иногда фильтры также именуются ядрами. Давайте определим четыре разных фильтра и проверим их эффективность позже в этой главе, когда они будут использоваться в операциях свертки. В этих примерах мы будем рабо-

тать с фильтрами 3×3. А пока для справки рассмотрим некоторые определения; чуть позже в этой главе вы увидите, как ими пользоваться.

- Следующее ядро позволит обнаруживать горизонтальные края:

$$\mathbf{J}_H = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}.$$

- Следующее ядро позволит обнаруживать вертикальные края:

$$\mathbf{J}_V = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}.$$

- Следующее ядро позволит обнаруживать края при резком изменении яркости:

$$\mathbf{J}_L = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}.$$

- Следующее ядро размывает края изображения:

$$\mathbf{J}_B = -\frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Далее в разделах мы применим свертку к тестовому изображению с помощью фильтров, и вы увидите их эффект.

Свертка

Первым шагом к пониманию CNN-сетей является понимание операции свертки. Самым простым способом увидеть ее в действии — взять нескольких простых случаев. В первую очередь в контексте нейронных сетей свертка выполняется между тензорами. Эта операция получает два тензора на входе и производит тензор на выходе. Данная операция обычно обозначается оператором $*$.

Посмотрим, как она работает. Давайте возьмем два тензора, оба с размерностями 3×3. Операция свертки выполняется по следующей формуле:

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} * \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix} = \sum_{i=1}^{9} a_i k_i .$$

В этом случае результатом является просто сумма произведений каждого элемента a_i на соответствующий элемент k_i. В более типичном матричном формализме эта формула может быть записана двойной суммой как

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{26} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ij}k_{ij} .$$

Но первый вариант записи имеет преимущество в том, что делает фундаментальную идею очень ясной: каждый элемент из одного тензора умножается на соответствующий элемент (элемент в той же позиции) второго тензора, а затем все произведения суммируются и дают результат.

В предыдущем разделе были упомянуты ядра, и неспроста: свертка обычно выполняется между тензором, который мы обозначим здесь через **A**, и ядром. Как правило, ядра являются малыми, 3×3 или 5×5, в то время как входные тензоры **A** обычно больше. Например, в распознавании изображений входные тензоры **A** являются изображениями, размерности которых могут достигать $1024\times1024\times3$, где 1024×1024 — это разрешающая способность, и последняя размерность (3) — число цветовых каналов, значения RGB (красный, зеленый, синий). В расширенных приложениях изображения могут иметь еще более высокую разрешающую способность. Как применять свертку при наличии матриц с разными размерностями? Для понимания того, как это делать, возьмем матрицу **A** с размерностью 4×4.

$$\mathbf{A} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}.$$

И давайте посмотрим, как выполнить свертку с ядром **K**, которое в этом примере мы возьмем равным 3×3.

$$\mathbf{K} = \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix}.$$

Идея состоит в том, чтобы начать с левого верхнего угла матрицы **A** и выбрать участок 3×3. В нашем примере это будет

$$\mathbf{A}_1 = \begin{bmatrix} a_1 & a_2 & a_3 \\ a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \end{bmatrix}.$$

Эти элементы отмечены жирным шрифтом:

$$\mathbf{A} = \begin{bmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} & \boldsymbol{a_3} & a_4 \\ \boldsymbol{a_5} & \boldsymbol{a_6} & \boldsymbol{a_7} & a_8 \\ \boldsymbol{a_9} & \boldsymbol{a_{10}} & \boldsymbol{a_{11}} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}.$$

Затем мы выполняем свертку, как объяснено в самом начале, между этой меньшей матрицей $\mathbf{A}_1$ и $\mathbf{K}$, получив (обозначим результат через B_1)

$$B_1 = \mathbf{A}_1 * \mathbf{K} = a_1k_1 + a_2k_2 + a_3k_3 + k_4a_5 + k_5a_5 + k_6a_7 + k_7a_9 + k_8a_{10} + k_9a_{11}.$$

Затем мы должны сдвинуть выбранный участок 3×3 в матрице $\mathbf{A}$ на один столбец вправо и выбрать следующие элементы, отмеченные жирным шрифтом:

$$\mathbf{A} = \begin{bmatrix} a_1 & \boldsymbol{a_2} & \boldsymbol{a_3} & \boldsymbol{a_4} \\ a_5 & \boldsymbol{a_6} & \boldsymbol{a_7} & \boldsymbol{a_8} \\ a_9 & \boldsymbol{a_{10}} & \boldsymbol{a_{11}} & \boldsymbol{a_{12}} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}.$$

В результате получим вторую подматрицу — $\mathbf{A}_2$:

$$\mathbf{A}_2 = \begin{bmatrix} a_2 & a_3 & a_4 \\ a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \end{bmatrix},$$

и мы снова выполняем свертку между этой меньшей матрицей $\mathbf{A}_2$ и $\mathbf{K}$:

$$B_2 = \mathbf{A}_2 * \mathbf{K} = a_2k_1 + a_3k_2 + a_4k_3 + k_4a_6 + k_5a_7 + k_6a_8 + k_7a_{10} + k_8a_{11} + k_9a_{12}.$$

Теперь мы не можем сдвинуть участок 3×3 вправо, потому что уже достигли конца матрицы $\mathbf{A}$, поэтому мы сдвигаем его на одну строку вниз и начинаем снова с левой стороны. Следующим выбранным участком будет

$$\mathbf{A}_3 = \begin{bmatrix} a_5 & a_6 & a_7 \\ a_9 & a_{10} & a_{11} \\ a_{13} & a_{14} & a_{15} \end{bmatrix}.$$

И снова мы выполняем свертку $\mathbf{A}_3$ с $\mathbf{K}$:

$$B_3 = \mathbf{A}_3 * \mathbf{K} = a_5k_1 + a_6k_2 + a_7k_3 + k_4a_9 + k_5a_{10} + k_6a_{11} + k_7a_{13} + k_8a_{14} + k_9a_{15}.$$

Как вы, возможно, уже догадались, последний шаг состоит в сдвиге выбранного участка 3×3 вправо на один столбец и выполнении свертки снова. Наш выбранный участок теперь будет

$$\mathbf{A}_4 = \begin{bmatrix} a_6 & a_7 & a_8 \\ a_{10} & a_{11} & a_{12} \\ a_{14} & a_{15} & a_{16} \end{bmatrix},$$

и свертка даст результат

$$B_4 = \mathbf{A}_4 * \mathbf{K} = a_6 k_1 + a_7 k_2 + a_8 k_3 + k_4 a_{10} + k_5 a_{11} + k_6 a_{12} + k_7 a_{14} + k_8 a_{15} + k_9 a_{16}.$$

Теперь мы не можем больше сдвинуть участок 3×3 ни вправо, ни вниз. Мы рассчитали 4 значения: B_1, B_2, B_3 и B_4. Эти элементы образуют результирующий тензор операции свертки, давая нам тензор **B**.

$$\mathbf{B} = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}.$$

Тот же самый процесс можно применять, когда тензор **A** больше. Вы просто получите больший результирующий тензор **B**, но алгоритм получения элементов B_i будет таким же. Прежде чем идти дальше, следует обсудить еще одну маленькую деталь — понятие сдвига, или страйда. В предыдущем процессе мы всегда перемещали участок 3×3 на один столбец вправо и одну строку вниз. Число строк и столбцов в примере 1 называется *сдвигом* и часто обозначается через s. Сдвиг $s = 2$ просто означает, что мы будем сдвигать участок 3×3 на два столбца вправо и две строки вниз.

Еще следует обсудить размеры выбранного участка во входной матрице **A**. Размерность выбранного участка, который мы сдвигали в этом процессе, должна быть такой же, как и у используемого ядра. Если вы используете ядро 5×5, то должны выбрать участок 5×5 в **A**. В общем случае с учетом ядра $n_\mathbf{K} \times n_\mathbf{K}$ вы будете выбирать участок $n_\mathbf{K} \times n_\mathbf{K}$ в **A**.

Более формальное определение состоит в том, что свертка со сдвигом s в нейросетевом контексте — это процесс, который на входе принимает тензор **A** размерности $n_\mathbf{A} \times n_\mathbf{A}$ и ядро **K** размерности $n_\mathbf{K} \times n_\mathbf{K}$ и дает на выходе матрицу **B** размерности $n_\mathbf{B} \times n_\mathbf{B}$ с

$$n_\mathbf{B} = \left[\frac{n_\mathbf{A} - n_\mathbf{K}}{s} + 1 \right],$$

где через $[x]$ мы обозначили целую часть x (в программировании она часто именуется округлением x вниз, или операцией `floor`). Обсуждение доказательства этой формулы заняло бы слишком много времени, но легко понять причину, почему она является верной (попробуйте выполнить ее математический вывод). В целях еще большего упрощения мы предположим, что $n_\mathbf{K}$ является нечетным. Скоро вы поймете, почему это важно (хотя и не существенно).

Начнем с формального объяснения случая со сдвигом $s=1$. Алгоритм генерирует новый тензор **B** из входного тензора **A** и ядра **K** согласно формуле

$$B_{ij} = (\mathbf{A} * \mathbf{K})_{ij} = \sum_{f=0}^{n_K-1} \sum_{h=0}^{n_K-1} a_{i+f,j+h} k_{i+f,j+h} .$$

Данная формула выглядит загадочно и очень трудна для понимания. Для лучшего понимания ее смысла рассмотрим еще несколько примеров. На рис. 8.1 показано визуальное объяснение работы свертки. Предположим, у вас есть фильтр 3×3. На рисунке видно, что девять левых верхних элементов матрицы **A**, очерченных черной непрерывной линией, используются для генерирования первого элемента B_1 матрицы, согласно приведенной выше формуле. Элементы, очерченные пунктирной линией, используются для генерирования второго элемента B_2 и т. д. Повторим то, что обсуждалось в примере в самом начале: основная идея заключается в том, что каждый элемент в квадрате 3×3 из матрицы **A** умножается на соответствующий элемент ядра **K**, и все числа суммируются. Затем сумма становится элементом новой матрицы **B**. После вычисления значения для B_1 вы сдвигаете участок, который рассматриваете в исходной матрице, на один столбец вправо (квадрат, обозначенный на рис. 8.1 пунктирной линией) и повторяете операцию. Вы продолжаете сдвигать участок вправо до тех пор, пока не достигнете границы, а затем перемещаете ограничительный квадрат один элемент вниз, начинаете вновь слева и продолжаете таким образом до тех пор, пока не достигнете правого нижнего угла матрицы. Одно и то же ядро используется для всех участков исходной матрицы.

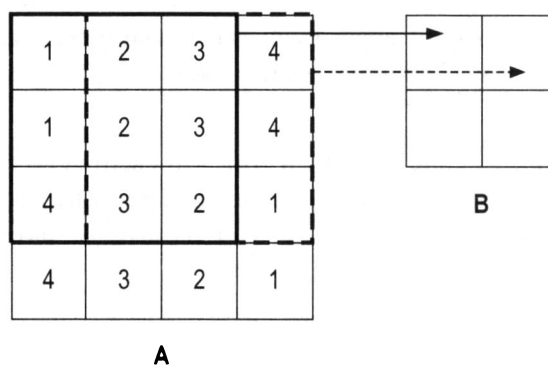


РИС. 8.1. Визуальное объяснение свертки

С учетом ядра $\mathbf{J}_H$, например, на рис. 8.2 показано, какой элемент в матрице **A** умножается на какой элемент в $\mathbf{J}_H$ и как получается результирующий элемент в B_1, т. е. не что иное, как сумму всех произведений:

$$B_1 = 1\cdot1+2\cdot1+3\cdot1+1\cdot0+2\cdot0+3\cdot0+4\cdot(-1)+3\cdot(-1)+3\cdot(-1)=-3.$$

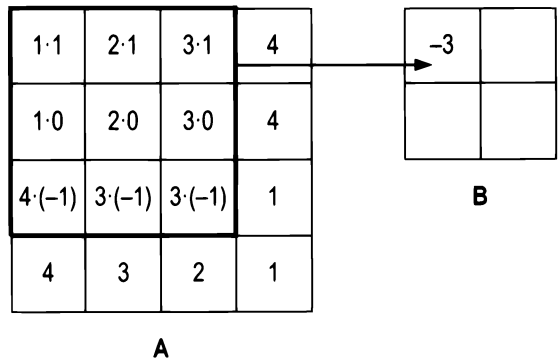


РИС. 8.2. Визуализация свертки с ядром $\mathbf{J}_H$

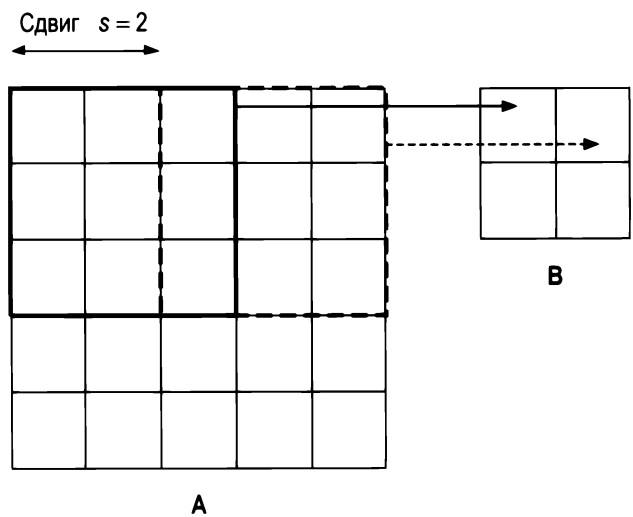


РИС. 8.3. Визуальное объяснение свертки со сдвигом $s = 2$

На рис. 8.3 приведен наглядный пример свертки со сдвигом $s = 2$.

Объяснение, почему размерность выходной матрицы принимает только округленную вниз (целочисленную часть) выражения $((n_{\mathbf{A}} - n_{\mathbf{K}})/s) + 1$, можно увидеть на рис. 8.4. Если $s > 1$, то в зависимости от размерности **A** может случиться так, что в какой-то момент вы не сможете сдвинуть окно на матрице **A** (например, черный квадрат на рис. 8.3) и не сможете полностью покрыть всю матрицу **A**. На рис. 8.4 видно, что для выполнения операции свертки понадобится дополнительный столбец справа от матрицы **A** (отмеченный многочисленными X). На рис. 8.4 $s = 3$, и поскольку мы имеем $n_{\mathbf{A}} = 5$ и $n_{\mathbf{K}} = 3$, **B** в результате будет скаляром.

$$n_{\mathbf{B}} = \left[\frac{n_{\mathbf{A}} - n_{\mathbf{K}}}{s} + 1\right] = \left[\frac{5-3}{3} + 1\right] = \left[\frac{5}{3}\right] = 1.$$

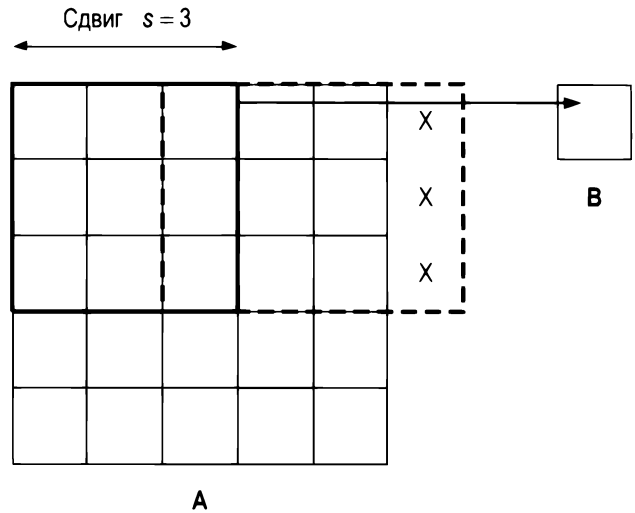


РИС. 8.4. Визуальное объяснение того, почему во время оценивания результирующих размеров матрицы **B** требуется функция округления вниз

Из рис. 8.4 четко видно, как с участком 3×3 можно охватить только левый верхний участок **A**, потому что при сдвиге $s = 3$ вы окажетесь вне **A** и, следовательно, в операции свертки сможете рассмотреть только один участок, тем самым получив скаляр для результирующего тензора **B**.

Давайте теперь рассмотрим несколько дополнительных примеров, которые сделают эту формулу еще яснее. Начнем с небольшой матрицы 3×3:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

Возьмем ядро

$$\mathbf{K} = \begin{bmatrix} k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 \\ k_7 & k_8 & k_9 \end{bmatrix}$$

и сдвиг $s = 1$. Свертка будет задана выражением

$$B = \mathbf{A} * \mathbf{K} = 1 \cdot k_1 + 2 \cdot k_2 + 3 \cdot k_3 + 4 \cdot k_4 + 5 \cdot k_5 + 6 \cdot k_6 + 7 \cdot k_7 + 8 \cdot k_8 + 9 \cdot k_9,$$

и результат B будет скаляром, потому что $n_{\mathbf{A}} = 3$, $n_{\mathbf{K}} = 3$, следовательно

$$n_{\mathbf{B}} = \left[\frac{n_{\mathbf{A}} - n_{\mathbf{K}}}{s} + 1 \right] = \left[\frac{3-3}{1} + 1 \right] = 1.$$

Если теперь взять матрицу **A** размерностью 4×4 или $n_{\mathbf{A}}=4$, $n_{\mathbf{K}}=3$ и $s=1$, то на выходе вы получите матрицу **B** размерностью 2×2, потому что

$$n_{\mathbf{B}}=\left[\frac{n_{\mathbf{A}}-n_{\mathbf{K}}}{s}+1\right]=\left[\frac{4-3}{1}+1\right]=2.$$

Например, вы можете убедиться, что при наличии

$$\mathbf{A}=\begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix}$$

и

$$\mathbf{K}=\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix}$$

мы имеем со сдвигом $s=1$

$$\mathbf{B}=\mathbf{A}*\mathbf{K}=\begin{bmatrix}348 & 393\\ 528 & 573\end{bmatrix}.$$

Давайте проверим один из элементов, B_{11}, с помощью приведенной выше формулы. Мы имеем

$$\begin{aligned}B_{11}&=\sum_{f=0}^{2}\sum_{h=0}^{2}a_{1+f,1+h}k_{1+f,1+h}=\sum_{f=0}^{2}\left(a_{1+f,1}k_{1+f,1}+a_{1+f,2}k_{1+f,2}+a_{1+f,3}k_{1+f,3}\right)=\\&=\left(a_{11}k_{11}+a_{12}k_{12}+a_{13}k_{13}\right)+\left(a_{21}k_{21}+a_{22}k_{22}+a_{23}k_{23}\right)+\\&+\left(a_{31}k_{31}+a_{32}k_{32}+a_{33}k_{33}\right)=\left(1\cdot1+2\cdot2+3\cdot3\right)+\left(5\cdot4+6\cdot5+7\cdot6\right)+\\&+\left(9\cdot7+10\cdot8+11\cdot9\right)=14+92+242=348.\end{aligned}$$

Отметим, что предоставленная формула свертки работает только для сдвига $s=1$, но может быть легко обобщена для других значений *s*.

Этот расчет очень легко реализовать на Python. Следующая ниже функция достаточно легко вычисляет свертку двух матриц для $s=1$. (Это можно реализовать на Python с помощью уже существующих функций, но, вне всяких сомнений, поучительно посмотреть, как это можно сделать с нуля.)

```
import numpy as np
def conv_2d(A, kernel):
    output = np.zeros([A.shape[0]-(kernel.shape[0]-1),
                       A.shape[1]-(kernel.shape[0]-1)])
```

```
    for row in range(1,A.shape[0]-1):
        for column in range(1, A.shape[1]-1):
            output[row-1, column-1] = np.tensordot(A[row-1:row+2,
                                                     column-1:column+2], kernel)
    return output
```

Обратите внимание, что входная матрица **A** даже не обязана быть квадратной, однако предполагается, что ядро таковым является и его размерность n_K нечетна. Предыдущий пример можно вычислить с помощью следующего фрагмента кода:

```
A = np.array([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]])
K = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(conv_2d(A,K))
```

В результате получим:

```
[[ 348. 393.]
 [ 528. 573.]]
```

Примеры свертки

Теперь давайте попробуем применить ядра, которые мы определили в начале, к тестовому изображению и посмотрим результаты. В качестве тестового изображения создадим шахматную доску размером 160×160 пикселов с помощью следующего ниже фрагмента кода:

```
chessboard = np.zeros([8*20, 8*20])
for row in range(0, 8):
    for column in range (0, 8):
        if ((column+8*row) % 2 == 1) and (row % 2 == 0):
            chessboard[row*20:row*20+20, column*20:column*20+20] = 1
        elif ((column+8*row) % 2 == 0) and (row % 2 == 1):
            chessboard[row*20:row*20+20, column*20:column*20+20] = 1
```

На рис. 8.5 показано, как выглядит шахматная доска.

Теперь попробуем применить свертку к этому изображению с разными ядрами со сдвигом $s = 1$.

Использование ядра $\mathbf{J}_H$ позволит обнаружить горизонтальные края. Это можно реализовать с помощью двух следующих инструкций:

```
edgeh = np.matrix('1 1 1; 0 0 0; -1 -1 -1')
outputh = conv_2d(chessboard, edgeh)
```

На рис. 8.6 показан полученный результат.

Теперь вы понимаете, почему утверждалось, что это ядро обнаруживает горизонтальные края. Кроме того, это ядро обнаруживает переход от светлого к темному, и наоборот. Обратите внимание, что это изображение составляет всего 158×158 пикселов, потому что

$$n_{\text{В}} = \left[\frac{n_{\text{А}} - n_{\text{К}}}{s} + 1\right] = \left[\frac{160 - 3}{1} + 1\right] = \left[\frac{157}{1} + 1\right] = [158] = 158.$$

Теперь давайте применим $\mathbf{J}_V$ с помощью пары строк кода

```
edgev = np.matrix('1 0 -1; 1 0 -1; 1 0 -1')
outputv = conv_2d(chessboard, edgev)
```

На выходе получим результат, показанный на рис. 8.7.

Теперь можно применить ядро $\mathbf{J}_I$

```
edgel = np.matrix ('-1 -1 -1; -1 8 -1; -1 -1 -1')
outputl = conv_2d(chessboard, edgel)
```

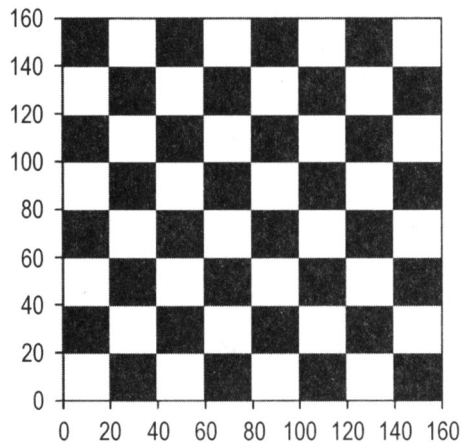


РИС. 8.5. Программно сгенерированное шахматное изображение

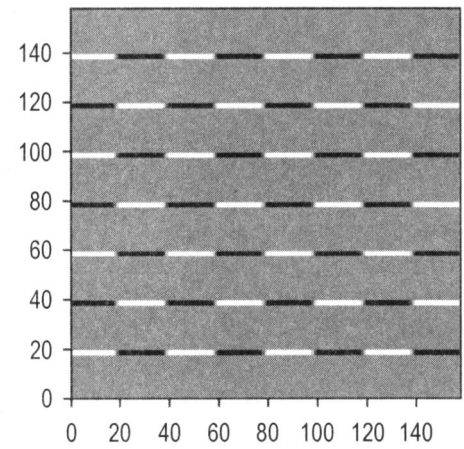


РИС. 8.6. Результат выполнения свертки между ядром $\mathbf{J}_H$ и изображением шахматной доски

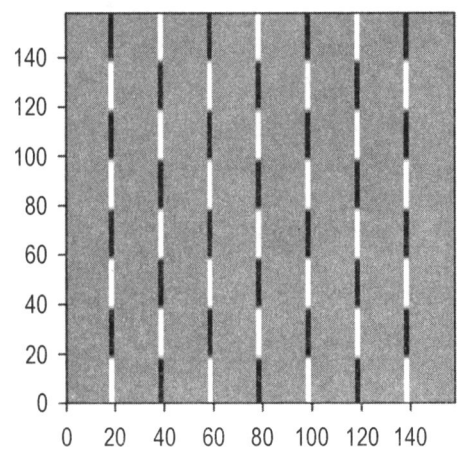


РИС. 8.7. Результат выполнения свертки между ядром $\mathbf{J}_V$ и изображением шахматной доски

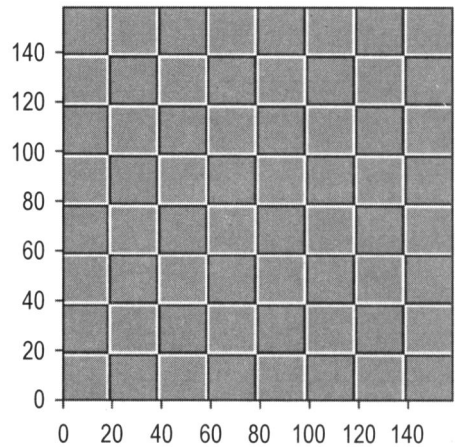


РИС. 8.8. Результат выполнения свертки между ядром J_L и изображением шахматной доски

На выходе получим результат, показанный на рис. 8.8.

И, наконец, можно применить размывающее ядро J_B.

```
edge_blur = -1.0/9.0*np.matrix('1 1 1; 1 1 1; 1 1 1')
output_blur = conv_2d(chessboard, edge_blur)
```

На рис. 8.9 видны два участка: слева — размытое изображение, справа — исходное. Эти изображения показывают только небольшой участок исходной шахматной доски с целью сделать размытие четче.

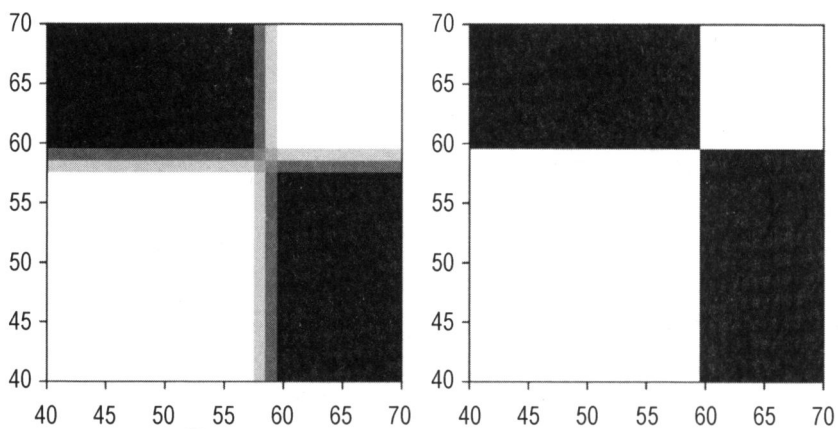


РИС. 8.9. Эффект размытия ядра J_B. Слева — размытое изображение, справа — исходное

В заключение этого раздела давайте попробуем получше разобраться в том, как обнаруживаются края. Рассмотрим следующую матрицу с резким вертикальным переходом, т. к. левая часть заполнена десятками, а правая — нулями.

```
ex_mat = np.matrix('10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0
0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10 10 10 0 0 0 0; 10 10
10 10 0 0 0 0; 10 10 10 10 0 0 0 0')
```

Результат выглядит следующим образом:

```
matrix([[10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0],
        [10, 10, 10, 10, 0, 0, 0, 0]])
```

Теперь рассмотрим ядро $\mathbf{J}_V$. Свертка выполняется с помощью одной строки кода

```
ex_out = conv_2d(ex_mat, edgev)
```

В результате получим:

```
array([[ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.],
       [ 0., 0., 30., 30., 0., 0.]])
```

На рис. 8.10 показаны исходная матрица (слева) и вывод свертки (справа). Свертка с ядром $\mathbf{J}_V$ четко обнаружила резкий переход в исходной матрице, отметив черной вертикальной линией место, где происходит переход от черного к белому. Например, рассмотрим $B_{11} = 0$.

$$B_{11} = \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix} * \mathbf{J}_V = \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} =$$

$$= 10 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) + 10 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) + 10 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) = 0.$$

Обратите внимание, что во входной матрице

$$\begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

нет никакого перехода, т. к. все значения одинаковы. Напротив, если взять B_{13}, то вы должны рассмотреть вот этот участок входной матрицы:

$$\begin{bmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{bmatrix},$$

где есть четкий переход, потому что самый правый столбец состоит из нулей, а остальные — из десятков. Теперь вы получите другой результат.

$$B_{11} = \begin{bmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{bmatrix} * \mathbf{J}_{l\cdot} = \begin{bmatrix} 10 & 10 & 0 \\ 10 & 10 & 0 \\ 10 & 10 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} =$$

$$= 10 \cdot 1 + 10 \cdot 0 + 0 \cdot (-1) + 10 \cdot 1 + 10 \cdot 0 + 0 \cdot (-1) + 10 \cdot 1 + 10 \cdot 0 + 0 \cdot (-1) = 30.$$

Благодаря именно этому, свертка будет возвращать высокое значение, как только происходит большое изменение значений вдоль горизонтального направления, потому что значения, умноженные на столбец с 1 в ядре, будут больше. Когда же, наоборот, вдоль горизонтальной оси происходит переход от малых значений к высоким, элементы, умноженные на -1, дадут результат, который будет больше по абсолютному значению, и, следовательно, окончательный результат будет отрицательным и больше по абсолютному значению. Как раз по этой причине данное ядро также может обнаруживать переход от светлого цвета к более темному, и наоборот. По сути дела, если бы вы рассматривали противоположный переход (от 0 до 10) в другой, гипотетической, матрице **A**, то вы бы имели

$$B_{11} = \begin{bmatrix} 0 & 10 & 10 \\ 0 & 10 & 10 \\ 0 & 10 & 10 \end{bmatrix} * \mathbf{J}_{l\cdot} = \begin{bmatrix} 0 & 10 & 10 \\ 0 & 10 & 10 \\ 0 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} =$$

$$= 0 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) + 0 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) + 0 \cdot 1 + 10 \cdot 0 + 10 \cdot (-1) = -30,$$

потому что на этот раз мы перейдем от 0 до 10 по горизонтали.

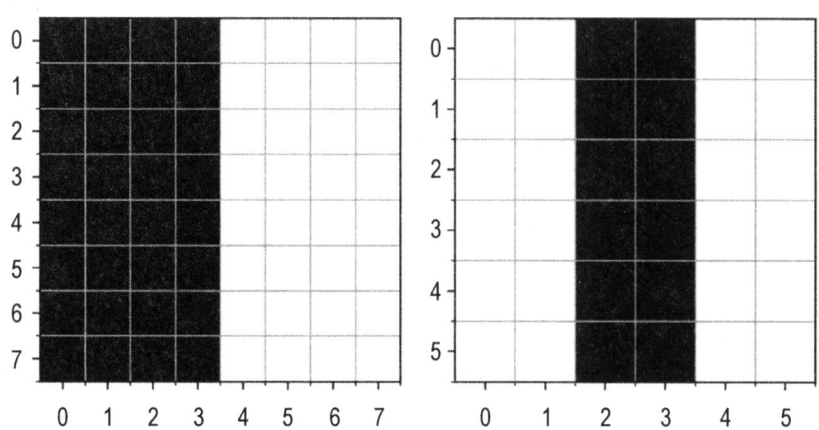


РИС. 8.10. Результат свертки матрицы `ex_mat` с ядром $\mathbf{J}_V$, как описано в тексте

Обратите внимание, что, как и ожидалось, выходная матрица имеет размерность 5×5, поскольку исходная матрица имеет размерность 7×7 и ядро 3×3.

Сведение

Сведение, или объединение (pooling) — это вторая фундаментальная для CNN-сетей операция. Она гораздо легче для понимания, чем свертка. Давайте опять рассмотрим конкретный пример и max pooling, т. е. сведение на основе максимума (функции max). Снова воспользуемся матрицей 4×4 из нашего анализа свертки.

$$\mathbf{A}=\begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}.$$

Для того чтобы выполнить сведение на основе максимума, мы должны определить участок размерности $n_\mathbf{K}\times n_\mathbf{K}$, аналогичный тому, что мы сделали для свертки. Рассмотрим $n_\mathbf{K}=2$. Мы должны начать с левого верхнего угла матрицы **A** и выбрать участок $n_\mathbf{K}\times n_\mathbf{K}$, в нашем случае 2×2, из **A**. Здесь мы выбираем

$$\begin{bmatrix} a_1 & a_2 \\ a_5 & a_6 \end{bmatrix}$$

или элементы, выделенные жирным шрифтом в матрице **A**, следующим образом:

$$\mathbf{A}=\begin{bmatrix} \boldsymbol{a_1} & \boldsymbol{a_2} & a_3 & a_4 \\ \boldsymbol{a_5} & \boldsymbol{a_6} & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}.$$

Из элементов a_1, a_2, a_5 и a_6 операция сведения на основе максимума выбирает максимальное значение, дав результат, который мы обозначим через B_1.

$$B_1=\max_{i=1,2,5,6} a_i.$$

Теперь мы должны переместить окно 2×2 вправо на два столбца, как правило, на то же самое число столбцов выбранного участка, и выбрать элементы, отмеченные жирным шрифтом

$$\mathbf{A}=\begin{bmatrix} a_1 & a_2 & \boldsymbol{a_3} & \boldsymbol{a_4} \\ a_5 & a_6 & \boldsymbol{a_7} & \boldsymbol{a_8} \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{bmatrix}$$

или, другими словами, меньшую матрицу

$$\begin{bmatrix} a_3 & a_4 \\ a_7 & a_8 \end{bmatrix}.$$

Затем алгоритм сведения на основе максимума выберет максимум значений, дав результат, который мы обозначим через B_2.

$$B_2 = \max_{i=3,\,4,\,7,\,8} a_i.$$

В этом месте мы больше не можем сдвигать участок 2×2 вправо, поэтому сдвигаем его на две строки вниз и начинаем процесс снова, с левой стороны **A**, выбрав элементы, отмеченные жирным шрифтом:

$$\mathbf{A} = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ \boldsymbol{a_9} & \boldsymbol{a_{10}} & a_{11} & a_{12} \\ \boldsymbol{a_{13}} & \boldsymbol{a_{14}} & a_{15} & a_{16} \end{bmatrix}$$

и получив максимум, назвав его B_3.

Сдвиг s в этом контексте имеет то же значение, что и при обсуждении свертки. Это просто число строк или столбцов, которые вы перемещаете свой участок во время выбора элементов. Наконец, мы выбираем последний участок, 2×2 в правой нижней части **A**, состоящий из элементов a_{11}, a_{12}, a_{15} и a_{16}. Затем мы получаем максимум и называем его B_4. С помощью значений, которые мы получим в результате этого процесса (в нашем примере четырех значений — B_1, B_2, B_3 и B_4), мы строим выходной тензор.

$$\mathbf{B} = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix}.$$

В этом примере мы имеем $s = 2$. В сущности, эта операция на входе принимает матрицу **A**, сдвиг s и размер ядра $n_{\mathbf{K}}$ (размерность участка, который мы выбрали в предыдущем примере) и на выходе возвращает новую матрицу **B** с размерностью, заданной той же самой формулой, которую мы применили при обсуждении свертки.

$$n_{\mathbf{B}} = \left[\frac{n_{\mathbf{A}} - n_{\mathbf{K}}}{s} + 1 \right].$$

Следует повторить, что идея состоит в том, чтобы начать с левого верхнего угла матрицы **A**, взять участок размерностью $n_{\mathbf{K}} \times n_{\mathbf{K}}$, применить функцию max к выбранным элементам, затем сдвинуть участок на s элементов вправо, выбрать новый участок — опять же с размерностью $n_{\mathbf{K}} \times n_{\mathbf{K}}$, применить функцию к ее значениям и т. д. На рис. 8.11 показано, как выбираются элементы из матрицы **A** со сдвигом $s = 2$.

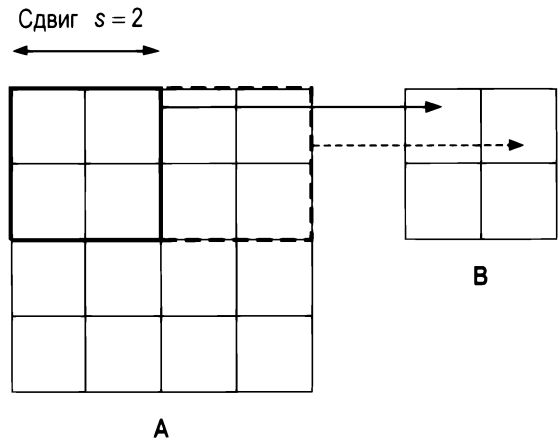


РИС. 8.11. Визуализация сведения со сдвигом $s = 2$

Например, применив сведение на основе максимума к входной матрице **A**

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 4 & 5 & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{bmatrix},$$

вы получите результат (это очень легко проверить)

$$\mathbf{B} = \begin{bmatrix} 5 & 11 \\ 15 & 21 \end{bmatrix},$$

потому что 5 является максимумом значений, выделенных жирным шрифтом

$$\mathbf{A} = \begin{bmatrix} \mathbf{1} & \mathbf{3} & 5 & 7 \\ \mathbf{4} & \mathbf{5} & 11 & 3 \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{bmatrix},$$

11 является максимумом следующих значений, выделенных жирным шрифтом:

$$\mathbf{A} = \begin{bmatrix} 1 & 3 & \mathbf{5} & \mathbf{7} \\ 4 & 5 & \mathbf{11} & \mathbf{3} \\ 4 & 1 & 21 & 6 \\ 13 & 15 & 1 & 2 \end{bmatrix},$$

и т. д. Стоит упомянуть еще один способ сведения, хотя и не столь широко используемый, как сведение на основе максимума: сведение на основе среднего значения (average pooling). Вместо того чтобы возвращать максимум выбранных значений, она возвращает среднее значение.

ПРИМЕЧАНИЕ. Наиболее часто используемой операцией сведения является сведение на основе максимума. Сведение на основе среднего значения используется не так широко, но его можно встретить в специфических сетевых архитектурах.

Заполнение

Заслуживает упоминания понятие заполнения (padding). Иногда во время работы с изображениями возникает не лучшая ситуация, когда получается результат операции свертки, размерность которого отличается от размерности исходного изображения. Поэтому иногда делают то, что называется *заполнением*. Его идея очень проста: заполнение состоит из добавления строк пикселов вверху, внизу и столбцов пикселов справа и слева от окончательных изображений, заполненных некими значениями для того, чтобы придать результирующим матрицам тот же размер, что и исходные. Некоторые стратегии заключаются в заполнении добавленных пикселов нулями, значениями ближайших пикселов и т. д. В нашем примере матрица `ex_out` с заполнением нулями будет выглядеть следующим образом:

```
array([[ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0., 30., 30.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]])
```

Использование и рассмотрение причин, лежащих в основе заполнения, выходит за рамки этой книги, но важно знать, что оно существует. Для справки, если вы используете заполнение p (ширину строк и столбцов, которые вы взяли в качестве заполнения), то итоговая размерность матрицы **B** в случае свертки и сведения задается формулой:

$$n_{\mathbf{B}} = \left[\frac{n_{\mathbf{A}} - 2p - n_{\mathbf{K}}}{s} + 1 \right].$$

ПРИМЕЧАНИЕ. Во время работы с реальными изображениями цветные изображения всегда кодируются в трех каналах: RGB. Это означает, что свертку и сведение необходимо исполнять в трех размерностях: ширина, высота и цветовой канал, в результате чего в алгоритмы добавляется еще один уровень сложности.

Строительные блоки CNN-сети

Операции свертки и сведения применяются для построения слоев, используемых в CNN-сетях. Обычно в CNN-сетях можно найти следующие слои:

- сверточные;
- сводящие;
- полносвязные.

Во всех предыдущих главах вы встречали именно полносвязные слои: слой, в котором нейроны связаны со всеми нейронами предыдущего и следующего слоя. Вы уже знакомы с перечисленными слоями, но первые два требуют некоторых дополнительных пояснений.

Сверточные слои

Сверточный слой на входе принимает тензор (благодаря трем цветовым каналам, он может быть трехмерным), например изображение определенной размерности; применяет определенное число ядер, обычно 10, 16 или даже больше; добавляет смещение; применяет активационные функции ReLU (например) для того, чтобы ввести нелинейность в результат свертки, и производит матрицу **B** на выходе. Если вы помните математические обозначения, которые мы использовали в предыдущих главах, то результат свертки будет играть роль члена $\mathbf{W}^{[l]}\mathbf{Z}^{[l-1]}$, который обсуждался в *главе 3*.

В предыдущих разделах было представлено несколько примеров использования свертки только с одним ядром. Как применить несколько ядер одновременно? Ответ очень прост. Окончательтный тензор **B** (будем говорить "тензор", потому что он больше не будет простой матрицей) теперь будет иметь не 2 размерности, а 3. Обозначим число, которое вы хотите применить, через n_c (использован подстрочный индекс c, потому что иногда эту размерность называют каналами, channels). Вы применяете к входу каждый фильтр независимо и складываете результаты ярусами в стек. Поэтому вместо одной матрицы **B** с размерностью $n_{\mathbf{B}} \times n_{\mathbf{B}}$ вы получаете окончательный тензор $\tilde{\mathbf{B}}$ размерности $n_{\mathbf{B}} \times n_{\mathbf{B}} \times n_c$. Это значит, что

$$\tilde{\mathbf{B}}_{i,j,1} \quad \forall i,\ j \in \left[1; n_{\mathbf{B}}\right]$$

будет выходом свертки входного изображения с первым ядром,

$$\tilde{\mathbf{B}}_{i,j,2} \quad \forall i,\ j \in \left[1; n_{\mathbf{B}}\right]$$

будет выходом свертки со вторым ядром и т. д. Сверточный слой — это не что иное, как фрагмент сети, который преобразовывает вход в выходной тензор. Но чем являются веса в этом слое? Веса, или параметры, которые сеть заучивает во время учебной фазы, сами являются элементами ядра. Мы уже касались того, что мы имеем n_c ядер с размерностью $n_{\mathbf{K}} \times n_{\mathbf{K}}$ каждое. Это означает, что в сверточном слое у нас $n_{\mathbf{K}}^2\ n_c$ параметров (или весов).

ПРИМЕЧАНИЕ. Число параметров $n_K^2 \, n_c$ в сверточном слое не зависит от размера входного изображения. Этот факт помогает в сокращении переподгонки, в особенности во время работы с входными изображениями большого размера.

Иногда этот слой обозначается словом Conv и числом. В нашем случае мы могли бы обозначить этот слой как Conv1. На рис. 8.12 показан внешний вид сверточного слоя. Входное изображение преобразовывается путем применения свертки с n_c ядрами в тензор размерности $n_A \times n_A \times n_c$.

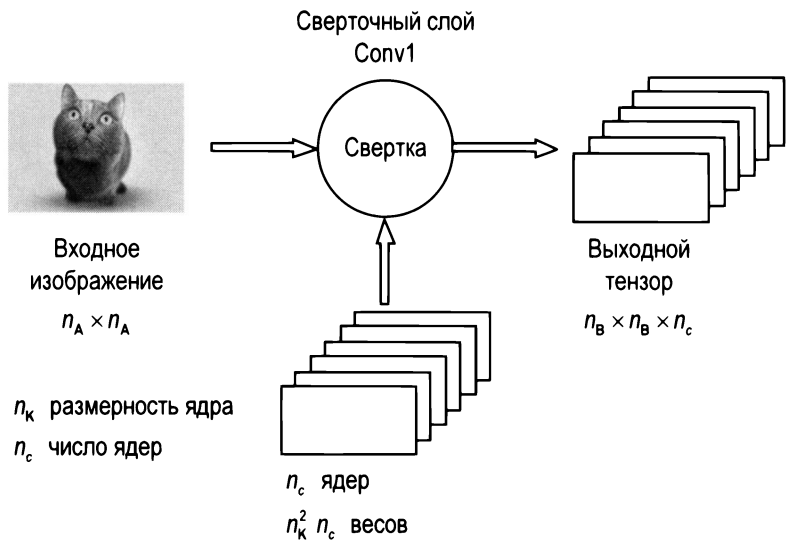


РИС. 8.12. Внешний вид сверточного слоя[1]

Сверточный слой не обязательно должен располагаться сразу после входа. Конечно же, сверточный слой может получать на входе выход любого другого слоя. Имейте в виду, что обычно входное изображение будет иметь размерность $n_A \times n_A \times 3$, потому что изображение в цвете имеет три канала: RGB. Полный анализ тензоров, задействуемых в CNN-сети при рассмотрении цветных изображений, выходит за рамки этой книги. Очень часто на диаграммах слой просто обозначается как куб или квадрат.

Сводящие слои

Сводящий слой обычно обозначается как Pool и числом, например Pool1. Он принимает тензор на входе и выдает другой тензор на выходе, после применения ко входу операции сведения.

[1] Источник фотоснимка с котом: **www.shutterstock.com/**.

ПРИМЕЧАНИЕ. Сводящий слой не имеет заучиваемых параметров, но он вводит дополнительные гиперпараметры: $n_к$ и сдвиг s. Как правило, в сводящих слоях операция заполнения не применяется, потому что одна из причин использования сведения часто заключается в сокращении размерности тензоров.

Стековая укладка слоев

В CNN-сетях обычно сверточные и сводящие слои укладываются в стек, ярусно один над другим. На рис. 8.13 представлен стек из сверточного и сводящего слоев. За сверточным слоем всегда следует сводящий слой. Иногда они вместе называются одним слоем. Это связано с тем, что сводящий слой не имеет заучиваемых весов, и поэтому он рассматривается как простая операция, связанная со сверточным слоем. При чтении исследовательских работ или блогов учитывайте данный факт и сверяйтесь с тем, что там имеется в виду.

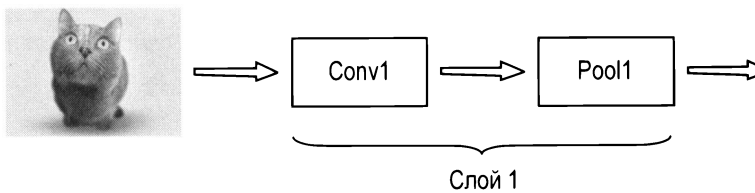


РИС. 8.13. Внешний вид стека, состоящего из сверточного и сводящего слоев

В завершение анализа CNN-сетей на рис. 8.14 приведен пример CNN-сети. Он похож на очень известную сеть LeNet-5, о которой вы можете прочитать здесь: **https://goo.gl/hMlkAL**. Он состоит из входа, затем двух пар сверточно-сводящих слоев, трех полносвязных слоев и выходного слоя, в котором вы можете иметь функцию `softmax`, в случае, например, выполнения мультиклассовой классификации. На рисунке помещено несколько произвольных чисел для того, чтобы дать вам представление о размере разных слоев.

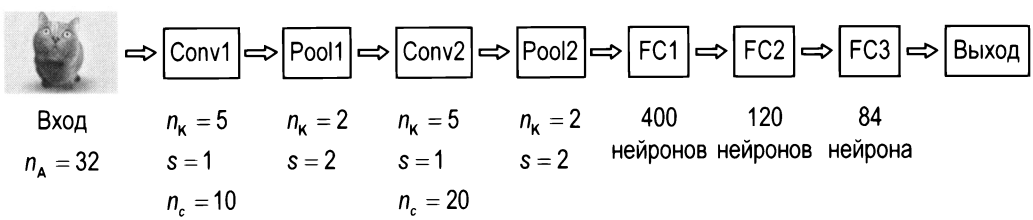


РИС. 8.14. Внешний вид CNN-сети, аналогичной знаменитой сети LeNet-5

Пример CNN-сети

Давайте попробуем построить одну такую сеть, чтобы прочувствовать весь процесс в работе и посмотреть, как будет выглядеть исходный код. Для того чтобы данный

раздел оказался для вас понятным, мы не будем прибегать к настройке и оптимизации гиперпараметров. Мы построим следующую архитектуру со слоями в указанном порядке:

1. Сверточный слой 1 с шестью фильтрами 5×5, сдвиг $s = 1$.
2. Сводящий слой 1 на основе максимума с окном 2×2, сдвиг $s = 2$.
3. Затем мы применяем ReLU к выходу из предыдущего слоя.
4. Сверточный слой 2 с 16 фильтрами 5×5, сдвиг $s = 1$.
5. Сводящий слой 2 на основе максимума с окном, 2×2, сдвиг $s = 2$.
6. Затем мы применяем ReLU к выходу из предыдущего слоя.
7. Полносвязный слой с 128 нейронами и активационной функцией ReLU.
8. Полносвязный слой с 10 нейронами для классификации набора данных Zalando.
9. Выходной нейрон Softmax.

Мы импортируем набор данных Zalando, как это было в *главе 3*, следующим образом:

```
data_train = pd.read_csv('fashion-mnist_train.csv', header = 0)
data_test = pd.read_csv('fashion-mnist_test.csv', header = 0)
```

Подробное объяснение того, как получить файлы, *см. в главе 3*. Далее давайте подготовим данные.

```
labels = data_train['label'].values.reshape(1, 60000)
labels_ = np.zeros((60000, 10))
labels_[np.arange(60000), labels] = 1
labels_ = labels_.transpose()
train = data_train.drop('label', axis=1)
```

и

```
labels_dev = data_test['label'].values.reshape(1, 10000)
labels_dev_ = np.zeros((10000, 10))
labels_dev_[np.arange(10000), labels_dev] = 1
test = data_dev.drop('label', axis=1)
```

Обратите внимание, что в этом случае, в отличие от *главы 3*, мы будем использовать транспонированную версию всех тензоров, т. е. в каждой строке у нас будет наблюдение. В *главе 3* каждое наблюдение было в столбце. Если проверить размерность с помощью этих строк кода

```
print(labels_.shape)
print(labels_dev_.shape)
```

то вы получите следующие результаты:

```
(60000, 10)
(10000, 10)
```

В *главе 3* размерности были в обратном порядке. Причина в том, что для разработки сверточных и сводящих слоев мы будем использовать функции, предоставляемые библиотекой TensorFlow, потому что разработка их с чистого листа потребует слишком много времени. Кроме того, в случае некоторых функций библиотеки TensorFlow легче, если тензоры имеют разные наблюдения в строках. Как и в *главе 3*, мы должны нормализовать данные.

```
train = np.array(train / 255.0)
dev = np.array(dev / 255.0)
labels_ = np.array(labels_)
labels_test_ = np.array(labels_test_)
```

Теперь можно начать строить сеть.

```
x = tf.placeholder(tf.float32, shape=[None, 28*28])
x_image = tf.reshape(x, [-1, 28, 28, 1])
y_true = tf.placeholder(tf.float32, shape=[None, 10])
y_true_scalar = tf.argmax(y_true, axis=1)
```

Вторая строка, `x_image = tf.reshape(x, [-1, 28, 28, 1])`, нуждается в разъяснениях. Напомним, что сверточный слой требует двумерного изображения, а не сглаженного списка пиксельных значений с оттенками серого, как это было в *главе 3*, где нашим входом был вектор с 784 (28×28) элементами.

ПРИМЕЧАНИЕ. Одним из самых больших преимуществ CNN-сетей является то, что они используют двумерную информацию, содержащуюся во входном изображении. Именно по этой причине входом в сверточные слои являются двумерные изображения, а не сглаженные векторы.

При построении CNN-сетей, как правило, для построения разных слоев определяют специальные функции. Благодаря этому, позже будет проще заниматься настройкой гиперпараметров, как мы видели ранее. Еще одна причина заключается в том, что когда мы сложим все части воедино с помощью функций, исходный код будет гораздо читабельнее. Имена функций должны быть понятными. Начнем с функции построения сверточного слоя. Обратите внимание, что в документации TensorFlow применяется термин "фильтр", поэтому мы будем его использовать в исходном коде.

```
def new_conv_layer(input, num_input_channels, filter_size, num_filters):
  shape = [filter_size, filter_size, num_input_channels, num_filters]
  weights = tf.Variable(tf.truncated_normal(shape, stddev=0.05))
  biases = tf.Variable(tf.constant(0.05, shape=[num_filters]))
  layer = tf.nn.conv2d(input=input, filter=weights, strides=[1, 1, 1, 1],
                       padding='SAME')
  layer += biases
  return layer, weights
```

В этом месте мы инициализируем веса из усеченного нормального распределения, смещения в виде константы, и затем будем использовать сдвиг $s = 1$. Сдвиг являет-

ся списком, который выдает сдвиг в разных размерностях. В наших примерах использованы изображения в оттенках серого, но у нас, например, также могут быть изображения RGB, вследствие чего размерностей будет больше: три цветовых канала.

Сводящий слой будет проще, т. к. он не имеет весов.

```
def new_pool_layer(input):
    layer = tf.nn.max_pool(value=input, ksize=[1, 2, 2, 1],
                           strides=[1, 2, 2, 1], padding='SAME')
    return layer
```

Теперь давайте определим Python-овскую функцию, которая применяет активационную функцию, в нашем случае ReLU, к предыдущему слою.

```
def new_relu_layer(input_layer):
    layer = tf.nn.relu(input_layer)
    return layer
```

Наконец, нам нужна функция для построения полносвязного слоя.

```
def new_fc_layer(input, num_inputs, num_outputs):
    weights = tf.Variable(tf.truncated_normal([num_inputs, num_outputs],
                                              stddev=0.05))
    biases = tf.Variable(tf.constant(0.05, shape=[num_outputs]))
    layer = tf.matmul(input, weights) + biases
    return layer
```

Здесь мы применили новые функции TensorFlow и, как можно догадаться по их именам, `tf.nn.conv2d` строит сверточный слой, `tf.nn.max_pool` строит сводящий слой на основе максимума. У нас нет возможности для подробного изложения того, что делает каждая из этих функций, но вы можете найти достаточно информации в официальной документации. Теперь давайте сложим все воедино и построим сеть, описанную в самом начале.

```
layer_conv1, weights_conv1 = new_conv_layer(input=x_image,
num_input_channels=1, filter_size=5, num_filters=6)
layer_pool1 = new_pool_layer(layer_conv1)
layer_relu1 = new_relu_layer(layer_pool1)
layer_conv2, weights_conv2 = new_conv_layer(input=layer_relu1,
num_input_channels=6, filter_size=5, num_filters=16)
layer_pool2 = new_pool_layer(layer_conv2)
layer_relu2 = new_relu_layer(layer_pool2)
```

Мы должны создать полносвязный слой, но для того чтобы использовать `layer_relu2` как вход, мы сначала должны его сгладить, потому что он по-прежнему двумерный.

```
num_features = layer_relu2.get_shape()[1:4].num_elements()
layer_flat = tf.reshape(layer_relu2, [-1, num_features])
```

Затем мы можем создать окончательные слои:

```
layer_fc1 = new_fc_layer(layer_flat, num_inputs=num_features, num_outputs=128)
layer_relu3 = new_relu_layer(layer_fc1)
layer_fc2 = new_fc_layer(input=layer_relu3, num_inputs=128, num_outputs=10)
```

Теперь вычислим предсказания, чтобы позже иметь возможность оценить точность.

```
y_pred = tf.nn.softmax(layer_fc2)
y_pred_scalar = tf.argmax(y_pred, axis=1)
```

Массив `y_pred_scalar` будет содержать номер класса в виде скаляра. Теперь необходимо определить стоимостную функцию, и, опять же, для того чтобы облегчить нашу жизнь и сохранить длину этой главы разумной, мы воспользуемся существующей функцией TensorFlow.

```
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=layer_
fc2, labels=y_true))
```

Как обычно, нам нужен оптимизатор.

```
optimizer = tf.train.AdamOptimizer(learning_rate=1e-4).minimize(cost)
```

Теперь можно, наконец, определить операции для оценивания точности.

```
correct_prediction = tf.equal(y_pred_scalar, y_true_scalar)
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Самое время натренировать нашу сеть. Мы будем использовать мини-пакетный градиентный спуск с пакетом размером 100 наблюдений и тренировать сеть всего в течение десяти эпох. Мы определим переменные следующим образом:

```
num_epochs = 10
batch_size = 100
```

Тренировка будет выполняться с помощью этого фрагмента кода:

```
with tf.Session() as sess:
  sess.run(tf.global_variables_initializer())
  for epoch in range(num_epochs):
    train_accuracy = 0
    for i in range(0, train.shape[0], batch_size):
      x_batch = train[i:i + batch_size,:]
      y_true_batch = labels_[i:i + batch_size,:]

      sess.run(optimizer, feed_dict={x:x_batch, y_true:y_true_batch})
      train_accuracy += sess.run(accuracy, feed_dict={x:x_batch,
                                          y_true:y_true_batch})
      train_accuracy /= int(len(labels_)/batch_size)
      dev_accuracy = sess.run(accuracy, feed_dict={x:dev,
                                                   y_true:labels_dev_})
```

Если выполнить этот фрагмент кода (на моем ноутбуке на это ушло примерно 10 минут), то всего через одну эпоху он начнет с тренировочной точности 63,7% и

через 10 эпох достигнет тренировочной точности 86% (то же самое на рабочем наборе). Напомним, что с первой довольно-таки "прямолинейной" сетью, которую мы разработали в *главе 3* с пятью нейронами в одном слое, мы достигли 66% с минипакетным градиентным спуском. Здесь мы тренировали нашу сеть только 10 эпох. Вы можете получить гораздо более высокую точность, если потренируете ее дольше. Кроме того, обратите внимание, что мы не делали никакой гиперпараметрической настройки, поэтому вы получите гораздо более высокие результаты, если потратите время на регулировку параметров.

Как вы могли заметить, всякий раз, когда вы вводите сверточный слой, вы будете вводить в каждый слой новые гиперпараметры:

- размер ядра;
- сдвиг;
- заполнение.

Они должны быть отрегулированы с целью получения оптимальных результатов. Как правило, разработчики используют существующие архитектуры для конкретных задач, которые уже оптимизированы другими специалистами и хорошо задокументированы в исследовательских работах.

Введение в RNN-сети

RNN-сети сильно отличаются от CNN-сетей и, как правило, используются для работы с последовательной информацией, другими словами, с данными, для которых порядок следования имеет значение. Типичным примером является набор слов в предложении. Вы можете легко понять, насколько важен порядок слов в предложении. Например, предложение "the rabbit ate the man" (человек съел кролика) имеет совершенно другое значение, чем предложение "the rabbit ate the man" (кролик съел человека), причем единственная разница заключается в порядке слов и в том, кто кем съеден. RNN-сети могут использоваться для предсказания, например, следующего слова в предложении. Возьмем, к примеру, предложение "Париж является столицей". Его легко завершить словом "Франции". Это означает, что существует информация о последнем слове предложения, закодированном в предыдущих словах, и эта информация является как раз тем, что используется RNN-сетями для предсказания последующих членов последовательности. Термин "рекуррентный" происходит от того, как работают эти сети: они применяют одну и ту же операцию к каждому элементу последовательности, накапливая информацию о предыдущих членах. Подытоживая, можно сказать, что:

- RNN-сети используют последовательные данные и информацию, закодированную в порядке следования членов последовательности;
- RNN-сети применяют один и тот же вид операции ко всем членам последовательности и запоминают предыдущие члены последовательности для предсказания следующего члена.

Прежде чем попытаться немного лучше понять принцип работы RNN-сетей, давайте рассмотрим несколько важных случаев использования, в которых они могут применяться. Это даст вам представление о потенциальном диапазоне приложений.

- *Генерирование текста*: предсказание вероятности слов с учетом предыдущего множества слов. Например, с помощью RNN-сетей можно легко генерировать текст, который выглядит как принадлежащий перу Шекспира, как это сделал А. Карпати в своем блоге, доступном по адресу **https://goo.gl/FodLp5**.
- *Машинный перевод*: с учетом множества слов на одном языке можно получить слова на другом языке.
- *Распознавание речи*: с учетом ряда звуковых сигналов (слов) можно предсказывать последовательность букв, образующих слова, как в речи.
- *Генерирование аннотаций на изображениях*: с помощью CNN-сетей RNN-сети можно использовать для создания меток на изображениях. См. статью А. Карпати на тему "Deep Visual-Semantic Alignments for Generating Image Descriptions" ("Глубокие визуально-семантические выравнивания для генерирования описаний изображений"), доступную по адресу **https://goo.gl/8Ja3n2**. Имейте в виду, что это довольно продвинутая исследовательская работа, которая требует обширной математической подготовки.
- *Виртуальные собеседники, или чатботы*: с учетом последовательности слов на входе RNN-сети пытаются генерировать ответы на эти входные данные.

Как вы понимаете, для того чтобы реализовать все приведенное выше, вам потребуются сложные архитектуры, которые нелегко описать в нескольких предложениях и которые требуют более глубокого (каламбур) понимания того, как работают RNN-сети, что выходит за рамки этой главы и книги.

Обозначения

Давайте рассмотрим последовательность "Paris is the capital of France" (Париж является столицей Франции). Это предложение будет подаваться в RNN-сеть по одному слову за раз: сначала "Paris", потом "is", затем "the" и т. д. В нашем примере,

- "Paris" будет первым словом последовательности: `w1 = 'Paris'`;
- "is" будет вторым словом последовательности: `w2 = 'is'`;
- "the" будет третьим словом последовательности: `w3 = 'the'`;
- "capital" будет четвертым словом последовательности: `w4 = 'capital'`;
- "of" будет пятым словом последовательности: `w5 = 'of'`;
- "France" будет шестым словом последовательности: `w6 = 'France'`.

Слова будут подаваться в RNN-сеть в следующем порядке: `w1`, `w2`, `w3`, `w4`, `w5` и `w6`. Разные слова будут обрабатываться сетью одно за другим, или, как некоторые любят говорить, в разные моменты времени. Обычно говорят, что если слово `w1` обрабатывается в момент t, то `w2` обрабатывается в момент $t+1$, `w3` в момент $t+2$ и т. д.

Время t не связано с реальным временем, но предполагает, что каждый элемент в последовательности обрабатывается последовательно, а не параллельно. Время t также не связано с вычислительным временем или чем-либо, связанным с ним. И приращение 1 в $t+1$ не имеет никакого значения. Это просто означает, что мы говорим о следующем элементе в нашей последовательности. При чтении статей, блогов или книг можно найти следующие обозначения:

- x_t — вход в момент времени t. Например, `w1` может быть входом в момент времени 1 (x_1), `w2` в момент времени 2 (x_2) и т. д.;
- s_t — это обозначение используется для внутренней памяти, которую мы еще не определили, в момент t. Величина s_t содержит накопленную информацию о предыдущих членах последовательности, рассмотренных ранее. Ее интуитивного понимания будет достаточно, потому что более математическое определение потребует слишком подробного объяснения;
- o_t — выход сети в момент времени t или, другими словами, после того, как все элементы последовательности t, включая элемент x_t, были поданы в сеть.

Основная идея RNN-сетей

Как правило, RNN-сеть в литературе обозначается в качестве левой части того, что проиллюстрировано на рис. 8.15. Используемое обозначение является индикативным — оно попросту показывает разные элементы сети: x обозначает входы, s — внутреннюю память, W — один набор весов и U — другой набор весов. На самом деле это схематическое описание представляет собой просто способ изобразить реальную структуру сети, которую вы видите справа от рис. 8.15. Иногда она называется развернутой версией сети.

Правая часть рис. 8.15 должна читаться слева направо. Первый нейрон на рисунке выполняет свое вычисление в указанный момент времени t, производит o_t и создает состояние внутренней памяти s_t. Второй нейрон, который вычисляет в период

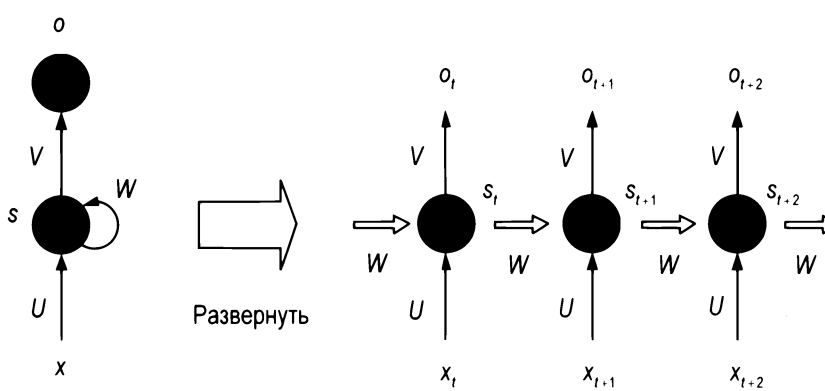


РИС. 8.15. Схематическое представление RNN-сети

времени $t+1$ после первого нейрона, получает на входе следующий элемент последовательности x_{t+1} и предыдущее состояние памяти s_t. Далее второй нейрон генерирует выход o_{t+1} и новое внутреннее состояние памяти s_{t+1}. Затем третий нейрон (крайний правый на рис. 8.15) на входе получает новый элемент последовательности x_{t+2} и предыдущее состояние внутренней памяти s_{t+1}, и процесс протекает таким образом для конечного числа нейронов. На рис. 8.15 вы видите, что существует два набора весов: W и U. Один набор (обозначен буквой W) используется для состояний внутренней памяти, другой, U — для элемента последовательности. Как правило, каждый нейрон будет генерировать новое состояние внутренней памяти с помощью формулы, которая выглядить следующим образом:

$$s_t = f\left(Ux_t + Ws_{t-1}\right),$$

где через $f()$ мы обозначили одну из активационных функций, которые мы уже встречали: ReLU или tanh. Кроме того, приведенная выше формула, конечно же, будет многомерной. s_t может пониматься как память сети в момент времени t. Число нейронов (или временных шагов, квантов времени), которые можно использовать, является новым гиперпараметром, и его необходимо регулировать в зависимости от задачи. Исследования показали, что когда это число слишком велико, у сети возникают большие проблемы во время тренировки.

Очень важно отметить, что на каждом временном шаге веса не изменяются. На каждом шаге выполняется одна и та же операция, просто изменяя входы всякий раз, когда выполняется вычисление. Вдобавок на рис. 8.15 показан выход (o_t, o_{t+1} и o_{t+2}) для каждого шага, но, как правило, этого не требуется. В нашем примере, в котором мы хотим предсказать последнее слово в предложении, может потребоваться только окончательный результат.

Почему именно "рекуррентная" сеть?

Здесь следует очень кратко обсудить вопрос, почему эти сети называются рекуррентными. Как было показано ранее, состояние внутренней памяти в момент времени t задается формулой:

$$s_t = f\left(Ux_t + Ws_{t-1}\right).$$

Состояние внутренней памяти в момент времени t оценивается с использованием того же самого состояния памяти в момент времени $t-1$, состояния в момент времени $t-1$ с учетом значения в момент времени $t-2$ и т. д. Именно это лежит в основе рекуррентности.

Учимся считать

Для того чтобы дать вам представление об мощи RNN-сетей, приведем очень простой пример того, в чем они очень хороши, а стандартные полносвязные сети, как та, которую вы встречали в предыдущих главах, действительно плохи. Попробуем

научить сеть считать. Задача, которую мы хотим решить, заключается в следующем: с учетом определенного вектора, который, предположим, состоит из 15 элементов и содержит только 0 и 1, мы хотим построить нейронную сеть, которая способна подсчитывать число единиц, содержащихся в векторе. Для стандартной сети эта задача является сложной, но почему? Для того чтобы интуитивно понять причину, давайте рассмотрим проанализированную нами задачу различения цифр 1 и 2 в наборе данных MNIST.

Как вы помните из обсуждения темы метрического анализа, заучивание происходит потому, что единицы и двойки имеют черные пикселы в принципиально разных позициях. Цифра 1 всегда будет отличаться (по крайней мере, в наборе данных MNIST) одинаковым образом от цифры 2, поэтому сеть идентифицирует эти различия, и как только они будут обнаружены, может осуществляться четкая идентификация. В нашем случае, это уже невозможно. Рассмотрим, например, более простой случай вектора с пятью элементами.

В этом случае единица появляется ровно один раз. У нас пять возможных случаев: [1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0] и [0, 0, 0, 0, 1]. Здесь нет никакой видимой закономерности. Нет легкой весовой конфигурации, которая могла бы охватывать эти случаи одновременно. В случае изображения эта задача аналогична задаче определения позиции черного квадрата на белом изображении. Мы можем построить сеть в TensorFlow и проверить качество работы таких сетей. Однако ввиду вводного характера этой главы мы не будем тратить время на обсуждение гиперпараметров, метрического анализа и т. д., а просто построим элементарную сеть, которая может считать.

Начнем с создания векторов. Мы создадим 105 векторов, которые разделим на тренировочный и рабочий наборы. Сперва, как обычно, давайте импортируем необходимые библиотеки.

```
import numpy as np
import tensorflow as tf
from random import shuffle
```

Затем создадим список векторов. Исходный код немного сложнее, и мы рассмотрим его подробнее.

```
nn = 15
ll = 2**15
train_input = ['{0:015b}'.format(i) for i in range(ll)]
shuffle(train_input)
train_input = [map(int,i) for i in train_input]
temp = []
for i in train_input:
    temp_list = []
    for j in i:
        temp_list.append([j])
    temp.append(np.array(temp_list))
train_input = temp
```

Мы хотим иметь все возможные комбинации 1 и 0 в векторах из 15 элементов. Поэтому простой способ сделать это — взять все числа вплоть до 2^{15} в двоичном формате. Для понимания причины, предположим, что мы хотим проделать это только с четырьмя элементами: мы хотим иметь все возможные комбинации 0 и 1 из четырех элементов. Рассмотрим все числа вплоть до 2^4 в двоичном формате, которые вы можете получить с помощью этой строки кода:

```
['{0:04b}'.format(i) for i in range(2**4)]
```

Данный код просто форматирует все числа, которые вы получаете с помощью функции `range(2**4)`, в интервале от 0 до `2**4`, в двоичном формате с помощью спецификатора `{0:04b}`, ограничивая число цифр до 4 позиций. Результатом является следующий список:

```
['0000',
 '0001',
 '0010',
 '0011',
 '0100',
 '0101',
 '0110',
 '0111',
 '1000',
 '1001',
 '1010',
 '1011',
 '1100',
 '1101',
 '1110',
 '1111']
```

Как вы можете легко убедиться, здесь перечислены все возможные комбинации. Мы имеем все возможные комбинации появления единицы ровно один раз (`[0001]`, `[0010]`, `[0100]` и `[1000]`), появления единицы два раза и т. д. В нашем примере мы сделаем это с 15 цифрами, т. е. с числами вплоть до 2^{15}. Остальная часть предыдущего кода состоит в том, чтобы просто преобразовать строку, к примеру "0100", в список `[0,1,0,0]`, а затем объединить все списки со всеми возможными комбинациями. Если проверить размерность выходного массива, то можно отметить, что получится (32 768, 15, 1). Каждое наблюдение представляет собой массив размерности (15, 1). Затем мы готовим целевую переменную, версию счетчиков, кодированную с одним активным состоянием. Это означает, что если у нас имеется вход с четырьмя единицами в векторе, то целевой вектор будет выглядеть как `[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]`. Ожидаемо, выходной массив `train_` будет иметь размерность (32 768, 16). Теперь давайте построим наши целевые переменные.

```
train_output = []

for i in train_input:
    count = 0
```

```
    for j in i:
        if j[0] == 1:
            count+=1
    temp_list = ([0]*(nn+1))
    temp_list[count]=1
    train_output.append(temp_list)
```

Разделим набор данных на тренировочный и рабочие наборы, как мы делали это уже несколько раз. Мы сделаем это здесь "прямолинейным" способом.

```
train_obs = 11-2000
dev_input = train_input[train_obs:]
dev_output = train_output[train_obs:]
train_input = train_input[:train_obs]
train_output = train_output[:train_obs]
```

Напомним, что это будет работать, потому что в самом начале векторы были перетасованы, и поэтому мы будем иметь случайное распределение вариантов. Мы будем использовать 2000 случаев для рабочего набора и остальные (примерно 30 000) для тренировки. Тренировочный набор `train_input` будет иметь размерность (30 768, 15, 1), а рабочий набор `dev_input` — размерность (2000, 15, 1).

Теперь вы можете построить сеть с помощью приведенного ниже фрагмента кода, который для вас должен быть понятным.

```
tf.reset_default_graph()

data = tf.placeholder(tf.float32, [None, nn,1])
target = tf.placeholder(tf.float32, [None, (nn+1)])

num_hidden_el = 24
RNN_cell = tf.nn.rnn_cell.LSTMCell(num_hidden_el, state_is_tuple=True)
val, state = tf.nn.dynamic_rnn(RNN_cell, data, dtype=tf.float32)
val = tf.transpose(val, [1, 0, 2])
last = tf.gather(val, int(val.get_shape()[0]) - 1)

W = tf.Variable(tf.truncated_normal([num_hidden, int(target.get_shape()[1])]))
b = tf.Variable(tf.constant(0.1, shape=[target.get_shape()[1]]))
prediction = tf.nn.softmax(tf.matmul(last, W) + b)
cross_entropy = -tf.reduce_sum(target * tf.log( \
                        tf.clip_by_value(prediction,1e-10,1.0)))
optimizer = tf.train.AdamOptimizer()
minimize = optimizer.minimize(cross_entropy)
errors = tf.not_equal(tf.argmax(target, 1), tf.argmax(prediction, 1))
error = tf.reduce_mean(tf.cast(errors, tf.float32))
```

Затем натренируем эту сеть.

```
init_op = tf.global_variables_initializer()
sess = tf.Session()
```

```
sess.run(init_op)
mb_size = 1000
no_of_batches = int(len(train_input)/mb_size)
epoch = 50
for i in range(epoch):
    ptr = 0
    for j in range(no_of_batches):
        train, output = train_input[ptr:ptr+mb_size],
train_output[ptr:ptr+mb_size]
        ptr+=mb_size
        sess.run(minimize,{data: train, target: output})
incorrect = sess.run(error,{data: test_input, target: test_output})
print('Эпоха {:2d} ошибка {:3.1f}%'.format(i + 1, 100 * incorrect))
```

Следующий фрагмент кода является новым, который вы, вероятно, не узнаете:

```
num_hidden_el = 24
RNN_cell = tf.nn.rnn_cell.LSTMCell(num_hidden_el,state_is_tuple=True)
val, state = tf.nn.dynamic_rnn(RNN_cell, data, dtype=tf.float32)
val = tf.transpose(val, [1, 0, 2])
last = tf.gather(val, int(val.get_shape()[0]) - 1)
```

По соображениям результативности и для того, чтобы вы поняли, насколько эффективными являются RNN-сети, здесь использован особый вид нейрона с долгой краткосрочной памятью (long short-term memory, LSTM). Этот нейрон представляет собой специальный механизм вычисления внутреннего состояния. Обсуждение LSTM-нейронов выходит далеко за рамки этой книги. На данный момент вы должны сосредоточиться не на исходном коде, а на результатах. Если его выполнить, то вы получите следующий результат:

```
Эпоха 0 ошибка 80.1%
Эпоха 10 ошибка 27.5%
Эпоха 20 ошибка 8.2%
Эпоха 30 ошибка 3.8%
Эпоха 40 ошибка 3.1%
Эпоха 50 ошибка 2.0%
```

После всего лишь 50 эпох сеть будет права в 98% случаев. Потренируйте ее побольше, в течение большего числа эпох, и вы сможете достичь невероятной точности. После 100 эпох вы сможете достичь ошибки 0,5%. Поучительным упражнением будет попытаться натренировать полносвязную сеть считать (одну из тех, о которых мы говорили до сих пор). Вы увидите, что это невозможно.

Теперь вы должны иметь базовое представление о том, как и на каких принципах работают CNN- и RNN-сети. Исследования в области этих сетей огромны, поскольку они по-настоящему гибки, однако материал, предоставленный в предыдущих разделах, должен был дать вам достаточно информации для понимания особенности работы этих архитектур.

ГЛАВА 9

Исследовательский проект

Обычно, говоря о глубоком обучении, люди думают о распознавании образов, распознавании речи, обнаружении образов и т. д. Это самые известные приложения, но возможности глубоких нейронных сетей безграничны. В этой главе будет показано, как глубокие нейронные сети могут успешно применяться в решении менее традиционной задачи: извлечении параметра в сенсорных приложениях. Для этой конкретной задачи мы разработаем алгоритмы для датчика, которые будут описаны позже, с целью определения концентрации кислорода в среде, например газе.

Данная глава организована следующим образом: вначале будет поставлена исследовательская задача, которую предстоит решить, затем будут даны пояснения по некоторым вводным материалам, необходимым для ее решения, и, наконец, будут представлены первые результаты этого продолжающегося исследовательского проекта.

Описание задачи

Принцип функционирования многих сенсорных устройств основан на измерении физической величины, такой как напряжение, объем или интенсивность света, которую обычно легко измерить. Эта величина должна быть тесно связано с другой физической величиной, которую предстоит определить и, как правило, которую трудно измерить непосредственно, например температура или, как в этом примере, концентрация газа. Если мы знаем, как две величины соотносятся (как правило, с помощью математической модели), то из первой величины можно вывести вторую, ту, которая нас действительно интересует. Таким образом, в упрощенном виде мы можем представить себе датчик как черный ящик, который при заданном входе (температура, концентрация газа и т. д.) выдает выход (напряжение, объем или интенсивность света). Зависимость выхода от входа характерна для конкретного типа сенсорного детектирования и может быть весьма сложной. Это делает реализацию необходимых алгоритмов на реальном оборудовании очень сложной или даже не-

возможной. Здесь вместо набора формул мы будем использовать подход к определению выхода из входа, используя нейронные сети.

Данный исследовательский проект касается измерения концентрации кислорода на основе принципа "люминесцентного гашения": чувствительный элемент, красящее вещество, находится в контакте с газом, в котором мы хотим измерить содержание кислорода. Краситель освещается так называемым возбуждающим светом (как правило, в синей части светового спектра), и, поглотив его часть, он переизлучает свет в другой части спектра (как правило, в красной части). Интенсивность и продолжительность испускаемого света сильно зависят от концентрации кислорода в газе, контактирующем с красителем. Если в газе есть кислород, то часть испускаемого красителем света подавляется или "гасится" (отсюда и название принципа измерения), причем этот эффект тем сильнее, чем выше количество кислорода в газе. Целью проекта является разработка новых алгоритмов определения концентрации кислорода (вход) по измеренному сигналу, так называемому фазовому сдвигу (выходу) между возбуждающим и излучаемым светом. Если вы не понимаете, что это значит, не переживайте. Для того чтобы понять содержание данной главы, этого вовсе не требуется. Достаточно интуитивного понимания того, что фазовый сдвиг измеряет изменение между светом, который приводит к возбуждению красителя, и светом, который был эмитирован после эффекта "гашения", и что на это "изменение" сильно влияет кислород, содержащийся в газе.

Сложность реализации датчика заключается в том, что отклик системы зависит (нелинейно) от нескольких параметров. Эта зависимость для большинства молекул красителя настолько сложна, что практически невозможно написать уравнения концентрации кислорода как функцию от всех этих влияющих параметров. Следовательно, типичным подходом является разработка очень сложной эмпирической модели с целым рядом параметров, настраиваемых вручную.

Типичная установка для измерения люминесценции схематически показана на рис. 9.1. Эта установка использовалась для получения контрольного набора данных. Образец, содержащий люминесцентное красящее вещество, освещается возбуждающим светом (на рисунке синий свет), излучаемым светодиодом или лазером, сфокусированным линзой. Излучаемая люминесценция (красный свет справа от рисунка) собирается детектором с помощью другой линзы. Держатель образца содержит краситель и газ, обозначенный на рисунке надписью "Образец", для которого мы хотим измерить концентрацию кислорода.

Интенсивность люминесценции, собираемая детектором, не постоянна во времени, и, скорее, уменьшается. Ответ на вопрос "Насколько быстро она уменьшается?" зависит от количества кислорода, обычно квантифицируемого временем затухания, обозначенным как τ. Простейшее описание этого затухания задается одной экспоненциальной функцией затухания, $e^{-t/\tau}$, характеризующейся временем затухания, τ. Распространенным техническим решением, используемым на практике для определения такого времени затухания, является модуляция интенсивности возбуждающего света или, другими словами, периодическое варьирование интенсивности с частотой $f = 2\pi\omega$, где ω — это так называемая угловая частота. Переизлученный

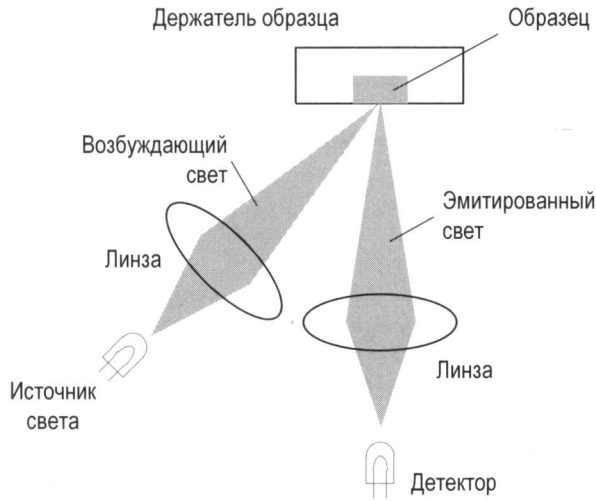


РИС. 9.1. Схема системы измерения люминесценции

люминесцентный свет имеет интенсивность, которая также модулируется, или, другими словами, периодически варьируется, но характеризуется фазовым сдвигом θ. Этот фазовый сдвиг связан со временем затухания τ как тангенс фазы $\tan\theta = \omega\tau$. Для интуитивного понимания того, что такое этот фазовый сдвиг, рассмотрим свет (да простит меня читатель, если он является физиком) в его простейшей форме, как волну с амплитудой, варьирующейся как тригонометрическая функция.

$$\sin(\omega t + \theta).$$

Величина θ называется фазовой константой волны. Далее, свет, который приводит к возбуждению краситель, имеет фазовую константу $\theta_{возб}$, а излучаемый свет имеет другую фазовую константу $\theta_{эмит}$. Принцип, который положен в основу измерений, измеряет как раз это фазовое изменение: $\theta \equiv \theta_{возб} - \theta_{эмит}$, потому что оно сильно влияет на содержание кислорода в газе. Пожалуйста, имейте в виду, что это объяснение очень интуитивное и с точки зрения физики не совсем правильное, но оно должно дать вам приблизительное представление о том, что именно мы измеряем.

Подводя итог, можно сказать, что измеряемым сигналом является фазовый сдвиг θ, далее просто именуемый фазой, в то время как искомой величиной (той, которую мы хотим предсказать) является концентрация кислорода в газе, контактирующем с красителем.

В реальной жизни ситуация, к сожалению, еще сложнее. Световой фазовый сдвиг зависит не только от частоты модуляции ω и концентрации кислорода O_2 в газе, но и нелинейно от температуры и химического состава окружения молекулы красителя. Кроме того, лишь в редких случаях затухание интенсивности света можно описать одним-единственным временем затухания. Чаще всего требуется не менее двух времен затухания, что еще больше увеличивает число параметров, необходи-

мых для описания системы. С учетом частоты лазерной модуляции ω, температуры T в градусах Цельсия и концентрации кислорода O_2 (выраженной в процентнах кислорода, содержащегося в воздухе), система возвращает фазу θ. На рис. 9.2 представлен график типично измеренного тангенса фазы $\tan\theta$ для $T = 45\,°C$ и $O_2 = 4\%$.

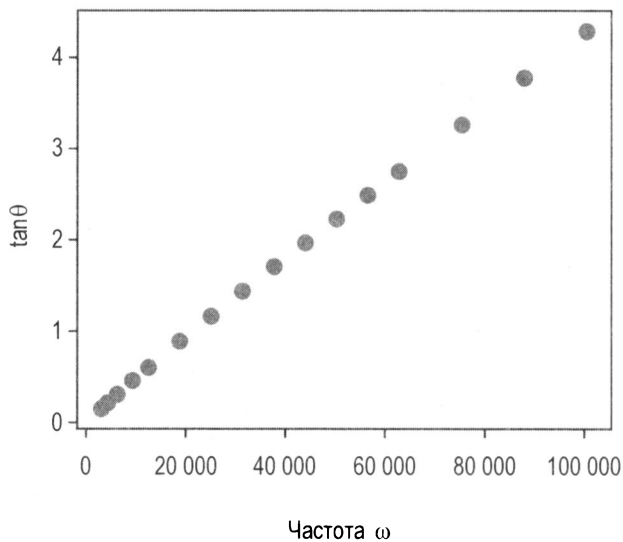


РИС. 9.2. График измеренного тангенса фазы $\tan\theta$ для $T = 45\,°C$ и $O_2 = 4\%$

Идея этого исследовательского проекта состоит в том, чтобы получить концентрацию кислорода из данных без необходимости разработки какой-либо теоретической модели поведения датчика. Для этого мы попытаемся использовать глубокие нейронные сети и из искусственно созданных данных дадим ей обучиться распознавать уровень концентрации кислорода в газе для любой данной фазы, а затем применим нашу модель к реальным экспериментальным данным.

Математическая модель

Рассмотрим одну из математических моделей, которая может быть использована для определения концентрации кислорода. Во-первых, модель дает представление о том, насколько она является сложной, а во-вторых, она будет использоваться в этой главе для создания тренировочного набора данных. Не вдаваясь в физику, связанную с измерительной методологией, которая выходит за рамки этой книги, простая модель, описывающая, как фаза θ связана с концентрацией кислорода O_2, может быть описана следующей формулой:

$$\frac{\tan\theta(\omega, T, O_2)}{\tan\theta(\omega, T, O_2 = 0)} = \frac{f(\omega, T)}{1 + \mathrm{KSV}_1(\omega, T) \cdot O_2} + \frac{1 - f(\omega, T)}{1 + \mathrm{KSV}_2(\omega, T) \cdot O_2}.$$

Величины $f(\omega, T)$, $\mathrm{KSV}_1(\omega, T)$ и $\mathrm{KSV}_2(\omega, T)$ — это параметры, аналитическая форма которых неизвестна и которые специфичны для используемой молекулы красителя, зависят от того, каков ее возраст, как построен датчик, и других факторов. Наша цель — натренировать нейронную сеть в лаборатории, а позже развернуть ее на датчике, который можно использовать в полевых условиях. Главная проблема здесь заключается в определении частотно- и температурно-зависимой формы функций f, KSV_1 и KSV_2. По этой причине коммерческие датчики обычно полагаются на многочленные или экспоненциальные аппроксимации с достаточным числом параметров и на процедуры подгонки для определения достаточно хорошей аппроксимации величин.

В этой главе мы создадим тренировочный экспериментальный набор данных с помощью только что описанной математической модели, затем применим ее к экспериментальным данным для того, чтобы увидеть, насколько хорошо мы сможем предсказывать концентрацию кислорода. Цель — исследовать работоспособность данного метода на практике.

Подготовка тренировочного набора данных в этом случае немного сложна и запутанна, поэтому сперва рассмотрим аналогичную, но гораздо более простую задачу с целью дать вам понять, что мы хотим сделать в более сложном случае.

Регрессионная задача

Рассмотрим сначала следующую задачу. При заданной функции $L(x)$ с параметром A мы хотим натренировать нейронную сеть извлекать значение параметра A из множества значений функции. Другими словами, при заданном множестве значений входной переменной x для $i = 1, ..., N$ мы вычислим массив из N значений $L_i = L(x_i)$ для $i = 1, ..., N$ и будем использовать их в качестве входа в нейронную сеть. Мы натренируем сеть и на выходе получим A. В качестве конкретного примера рассмотрим следующую функцию:

$$L(x) = \frac{A^2}{A^2 + x^2}.$$

Это так называемая лоренцева функция с максимумом в $x = 0$ и с $L(0) = 1$. Задача, которую мы хотим здесь решить, состоит в том, чтобы определить A при заданном конкретном числе точек данных этой функции. В нашем случае это довольно просто, потому что это можно сделать, например, с помощью классической нелинейной подгонки или даже путем решения простого квадратичного уравнения, но предположим, что мы хотим натренировать нейронную сеть, которая будет это делать. Мы хотим, чтобы нейронная сеть обучилась выполнять нелинейную подгонку этой функции. С учетом всего того, что вы узнали из этой книги, это не окажется слишком сложным. Начнем с создания тренировочного набора данных. Сначала определим функцию для $L(x)$:

```
def L(x,A):
    y = A**2/(A**2+x**2)
    return y
```

Теперь возьмем 100 точек и сгенерируем массив из всех x точек, которые мы хотим использовать.

```
number_of_x_points = 100
min_x = 0.0
max_x = 5.0
x = np.arange(min_x, max_x, (max_x-min_x)/number_of_x_points )
```

Наконец, сгенерируем 1000 наблюдений, которые мы будем использовать в качестве входа в сеть.

```
number_of_samples = 1000
np.random.seed(20)
A_v = np.random.normal(1.0, 0.4, number_of_samples)

for i in range(len(A_v)):
    if A_v[i] <= 0:
        A_v[i] = np.random.random_sample([1])
data = np.zeros((number_of_samples, number_of_x_points))
targets = np.reshape(A_v, [1000,1])

for i in range(number_of_samples):
    data[i,:] = L(x, A_v[i])
```

Данные массива теперь будут построчно содержать все наблюдения. Обратите внимание, что в исходный код встроена проверка.

```
if A_v[i] <= 0:
    A_v[i] = np.random.random_sample([1])
```

Это сделано для того, чтобы избежать отрицательных значений для A. Благодаря этому, если выбранное для A случайное значение является отрицательным, назначается новое случайное значение. Возможно, вы заметили, что в уравнении для $L(x)$ величина A всегда возводится в квадрат, поэтому на первый взгляд отрицательное значение не будет проблемой. Но напомним, что это отрицательное значение будет именно той целевой переменной, которую мы хотим предсказать. Когда данная модель разрабатывалась впервые, имелось несколько отрицательных значений для A, и сети не удавалось проводить различие между положительными и отрицательными значениями, тем самым выдавая неправильные результаты.

Если проверить форму массива `A_v`, то вы получите

```
(1000, 100)
```

Это переводится так: 1000 наблюдений по 100 значений каждое, которые представляют собой разные значения L, вычисляются в значениях x, которые мы сгенерировали. Конечно же, нам также нужен рабочий набор данных.

```
number_of_dev_samples = 1000

np.random.seed(42)
A_v_dev = np.random.normal(1.0, 0.4, number_of_samples)

for i in range(len(A_v_dev)):
    if A_v_dev[i] <= 0:
        A_v_dev[i] = np.random.random_sample([1])

data_dev = np.zeros((number_of_samples, number_of_x_points))
targets_dev = np.reshape(A_v_dev, [1000,1])

for i in range(number_of_samples):
    data_dev[i,:] = L(x, A_v_dev[i])
```

На рис. 9.3 представлены четыре произвольных примера функций, которые мы будем использовать в качестве входа в сеть.

Теперь давайте построим простую сеть с одним слоем и десятью нейронами для того, чтобы попытаться извлечь это значение.

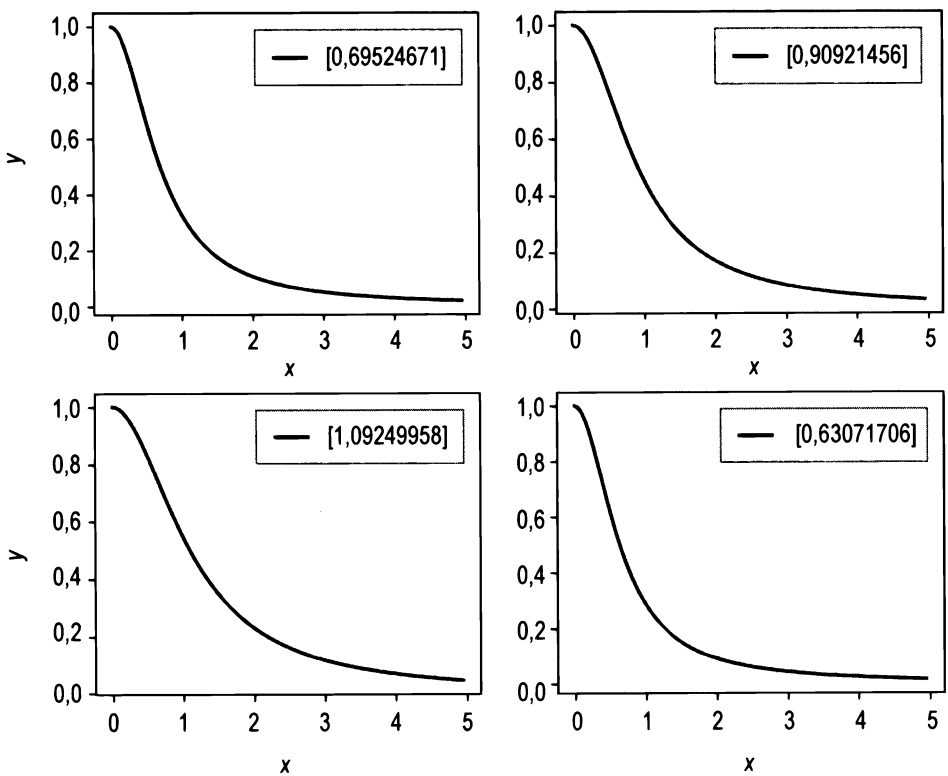


РИС. 9.3. Четыре случайных примера функции $L(x)$. В легенде указаны значения, использованные для A на графике

```
tf.reset_default_graph()

n1 = 10
nx = number_of_x_points
n2 = 1

W1 = tf.Variable(tf.random_normal([n1,nx]))/500.0
b1 = tf.Variable(tf.ones((n1,1)))/500.0
W2 = tf.Variable(tf.random_normal([n2,n1]))/500.0
b2 = tf.Variable(tf.ones((n2,1)))/500.0

X = tf.placeholder(tf.float32, [nx, None]) # Inputs
Y = tf.placeholder(tf.float32, [1, None]) # Labels

Z1 = tf.matmul(W1,X)+b1
A1 = tf.nn.sigmoid(Z1)
Z2 = tf.matmul(W2,A1)+b2
y_ = Z2
cost = tf.reduce_mean(tf.square(y_-Y))
learning_rate = 0.1
training_step = tf.train.AdamOptimizer(learning_rate).minimize(cost)

init = tf.global_variables_initializer()
```

Обратите внимание, что мы инициализировали веса случайно и не будем использовать мини-пакетный градиентный спуск. Натренируем сеть в течение 20 000 эпох.

```
sess = tf.Session()
sess.run(init)

training_epochs = 20000
cost_history = np.empty(shape=[1], dtype = float)

train_x = np.transpose(data)
train_y = np.transpose(targets)

cost_history = []
for epoch in range(training_epochs+1):
    sess.run(training_step, feed_dict = {X: train_x, Y: train_y})
    cost_ = sess.run(cost, feed_dict={ X:train_x, Y: train_y})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 1000 == 0):
        print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Модель сходится очень быстро. MSE (стоимостная функция) начинается в 1,1 и проходит до примерно $2{,}5 \cdot 10^{-4}$ после 10 000 эпох. После 20 000 эпох MSE дости-

гает 10^{-6}. Можно построить график предсказанных и реальных значений с целью получить визуальную проверку результативности системы. На рис. 9.4 вы видите результативность системы.

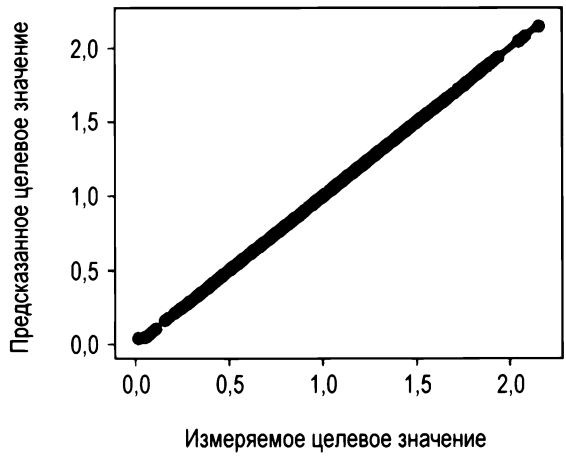


РИС. 9.4. Предсказанные значения *A* в сопоставлении с реальными

Кстати, MSE на рабочем наборе равна $3 \cdot 10^{-5}$, поэтому у нас, вероятно, имеется небольшая переподгонка к тренировочному набору данных. Одна из причин заключается в том, что мы рассматриваем относительно узкий интервал *x*: только от 0 до 5. Поэтому, когда вы имеете дело с очень большими значениями *A* (порядка 2,5, например), система склонна работать не так хорошо. Если вы проверите тот же график, что и на рис. 9.4, но для рабочего набора данных (рис. 9.5), то увидите, что модель имеет проблемы с большими значениями *A*.

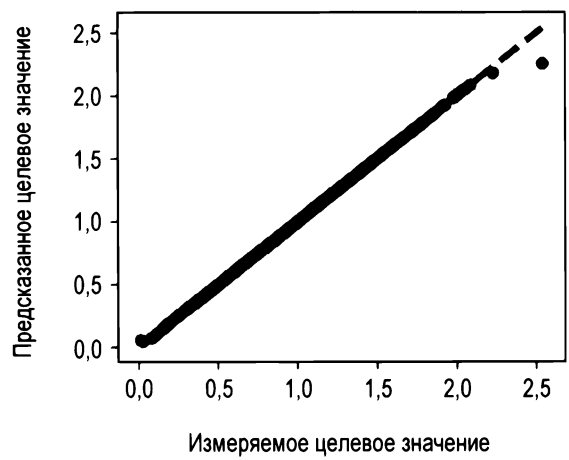


РИС. 9.5. Предсказанные значения *A*
в сопоставлении с реальными для рабочего набора данных

Существует еще одна причина наличия таких плохих предсказаний при более высоких значениях A. Когда мы генерировали тренировочные данные, для значений A мы использовали следующую строку кода:

```
A_v = np.random.normal(1.0, 0.4, number_of_samples)
```

Она означает, что выбираемые для A значения распределены в соответствии с нормальным распределением со средним значением 1,0 и стандартным отклонением 0,4. Будет очень мало наблюдений с A больше 2,0. Вы можете повторить все проделанное нами упражнение, но на этот раз выбрав значения A из равномерного распределения с помощью строки кода

```
A_v = np.random.random_sample([number_of_dev_samples])*3.0
```

Эта строка кода даст вам случайные числа между 0 и 3,0. После 20 000 эпох теперь мы получим $MSE_{тренировка} = 3,8 \cdot 10^{-6}$ и $MSE_{разработка} = 1,7 \cdot 10^{-6}$. На этот раз наши предсказания на рабочем наборе данных будут намного лучше, и, похоже, переподгонка отсутствует.

ПРИМЕЧАНИЕ. Когда вы генерируете тренировочные данные синтетически, следует всегда проводить проверку на экстремальные значения. Ваши тренировочные данные должны охватывать все возможные случаи, которые вы ожидаете увидеть в реальной жизни; в противном случае предсказания будут безуспешными.

Подготовка набора данных

Теперь приступим к генерированию набора данных, который нам понадобится для проекта. Генерирование данных будет немного сложнее, и, как вы увидите, нам придется потратить на него больше времени, чем на разработку и регулировку сети. Цель этой главы — показать, как нейронные сети можно использовать для исследовательских проектов, не относящихся к "классическому" варианту применения, например распознаванию образов. Экспериментальные данные состоят из 50 мерных показателей θ; 5 температур: 5, 15, 25, 35 и 45 °C; и 10 разных значений концентрации кислорода: 0, 4, 8, 15, 20, 30, 40, 60, 80 и 100%. Каждый мерный показатель состоит из 22 частотных замеров для следующих значений ω:

```
0 62.831853
1 282.743339
2 628.318531
3 1256.637061
4 3141.592654
5 4398.229715
6 6283.185307
7 9424.777961
8 12566.370614
9 18849.555922
```

```
10 25132.741229
11 31415.926536
12 37699.111843
13 43982.297150
14 50265.482457
15 56548.667765
16 62831.853072
17 75398.223686
18 87964.594301
19 100530.964915
20 113097.335529
21 125663.706144
```

Для тренировочного набора данных мы будем использовать только частоты между 3 и 100 кГц. Из-за ограничений экспериментальной установки артефакты и ошибки начинают появляться ниже 3 кГц и выше 10 кГц. При попытке использовать все данные целиком сеть работает намного хуже. И это не станет ограничением.

Даже если у вас нет готовых к работе файлов данных, далее будут даны пояснения по поводу того, как их подготавливать для многократного их использования в своем случае. Прежде всего, файлы были сохранены в одной папке под названием data. Мы создаем список с именами всех файлов, которые хотим загрузить:

```
files = os.listdir('./data')
```

Такая информация, как температура и концентрация кислорода, закодирована в имени файла, поэтому мы должны ее оттуда извлечь. Имя файла выглядит так: 20180515_PST3-1_45C_Mix8_00.txt, где 45C — это температура, а 8_00 — концентрация кислорода. Для извлечения информации предназначена специальная функция.

```
def get_T_O2(filename):
    T_ = float(filename[17:19])
    O2_ = float(filename[24:-4].replace('_','.'))
    return T_, O2_
```

Она будет возвращать два значения, одно из которых содержит значение температуры (`T_`), а другое — значение концентрации кислорода (`O2_`). Затем содержимое файла конвертируется в кадр данных библиотеки pandas, который позволит с ним работать.

```
def get_df(filename):
    frame = pd.read_csv('./data/'+filename, header = 10, sep = '\t')
    frame = frame.drop(frame.columns[6], axis=1)
    frame.columns=['f', 'ref_r', 'ref_phi', 'raw_r', 'raw_phi', 'sample_phi']
    return frame
```

Разумеется, эта функция была написана в соответствии с тем, как файлы были структурированы. Так выглядят первые строки файла:

```
StereO2

Probe: PST3-1
Medium: N2+Mix, Mix 0 %
Temperatur: 5 °C
Detektionsfilter: LP594 + SP682
HW Config Ref:
D:\Projekt\20180515_Quarzglas_Reference_00.ini
HW Config Sample: D:\Projekt\20180515_PST3_Sample_00.ini
Date, Time: 15.05.2018, 10:37
Filename: D:\Projekt\20180515_ PST3-1_05C_Mix0_00.txt
$Data$
Frequency (Hz)  Reference R (V)  Reference Phi (deg)  Sample Raw R (V)
Sample Raw Phi (deg)  Sample Phi (deg)
10.00E+0        247.3E-3         18.00E-3             371.0E-3
258.0E-3              240.0E-3
45.00E+0        247.4E-3         72.00E-3             371.0E-3
1.164E+0              1.092E+0
100.0E+0        248.4E-3         108.0E-3             370.9E-3
2.592E+0              2.484E+0
200.0E+0        247.5E-3         396.0E-3             369.8E-3
5.232E+0              4.836E+0
```

Если вы хотите сделать что-то подобное, то, естественно, вам придется модифицировать функции. Теперь давайте переберем все файлы в цикле и создадим списки со значениями T, O_2 и кадры данных. В папке data находятся 50 файлов.

```
frame = pd.DataFrame()
df_list = []
T_list = []
O2_list = []

for file_ in files:
    df = get_df(file_)
    T_, O2_ = get_T_O2(file_)

    df_list.append(df)
    T_list.append(T_)
    O2_list.append(O2_)
```

Можно проверить содержимое одного из файлов. Давайте наберем, например,

```
get_df(files[2]).head()
```

На рис. 9.6 показаны первые пять файловых записей с индексом 2.

Этот файл содержит больше информации, чем требуется.

У нас есть частота f, которую мы должны конвертировать в угловую частоту ω (между ними существует коэффициент, равный 2π), и мы должны вычислить тангенс фазы θ. Мы делаем это с помощью следующего фрагмента кода:

```
for df_ in df_list:
    df_['w'] = df_['f']*2*np.pi
    df_['tantheta'] = np.tan(df_['sample_phi']*np.pi/180.0)
```

добавляя, таким образом, два новых столбца в каждый кадр данных. В этом месте мы должны найти хорошую аппроксимацию для f, KSV_1 и KSV_2, что дает нам возможность создать набор данных. В качестве примера и для того, чтобы сделать эту главу более краткой, возьмем только одну температуру: $T = 45$ °C. Отфильтруем все наши данные, оставив только те, которые были измерены при этой температуре. Для этого используется следующий фрагмент кода:

```
T = 45

Tdf = pd.DataFrame(T_list, columns = ['T'])
Odf = pd.DataFrame(O2_list, columns = ['O2'])
filesdf = pd.DataFrame(files, columns = ['filename'])

files45 = filesdf[Tdf['T'] == T]
filesref = filesdf[(Tdf['T']==T) & (Odf['O2']==0)]
fileref_idx = filesref.index[0]
O5 = Odf[Tdf['T'] == T]
dfref = df_list[fileref_idx]
```

	f	ref_r	ref phi	raw_r	raw_phi	sample_phi
0	10.0	0.2473	0.018	0.2501	0.192	0.174
1	45.0	0.2474	0.072	0.2502	0.846	0.774
2	100.0	0.2484	0.108	0.2501	1.902	1.794
3	200.0	0.2475	0.396	0.2498	3.798	3.402
4	500.0	0.2474	1.008	0.2471	9.456	8.448

РИС. 9.6. Первые пять записей файла с индексом 2

Сначала мы конвертируем списки `T_list` и `O2_list` в кадры данных `Tdf` и `Odf`, потому что в этом формате легче отбирать правильные данные. Затем, как вы, возможно заметили, мы отбираем из кадра данных `filesdf` все файлы с $T = 45$ °C в кадр данных `files45`. Вдобавок мы отбираем из `filesdf` $T = 45$ °C и $O_2 = 0\%$ и называем новый кадр данных `dfref`. Всё это делается из-за того, что в самом начале была предоставлена формула θ, которая включала $\tan\theta(\omega, T, O_2 = 0)$. Кадр данных `dfref` будет содержать как раз измеренные данные для $\tan\theta(\omega, T, O_2 = 0)$. Напомним, что нам предстоит моделировать величину

$$\frac{\tan\theta(\omega, T = 45\ °C)}{\tan\theta(\omega, T = 45\ °C, O_2 = 0)}.$$

Вполне понятно, что все становится сложнее, но стоит подождать, т. к. мы почти закончили. Выбрать правильный кадр данных из списка кадров данных немного сложнее, но это можно сделать следующим образом:

```
from itertools import compress
A = Tdf['T'] == T
data = list(compress(df_list, A))

B = (Tdf['T']==T) & (Odf['O2']==0)
dataref_ = list(compress(df_list, B))
```

Функцию `compress` легко понять. Более подробную информацию вы можете найти на странице официальной документации, доступной по адресу **https://goo.gl/xNZEHH**. В сущности, идея заключается в том, что при наличии двух списков `d` и `s` на выходе из `compress(d, s)` задается новый список, равный `[(d[0] if s[0]), (d[1] if s[1]), ...]`. В нашем случае `A` и `B` представляют собой списки, состоящие из булевых значений, поэтому приведенный код возвращает только те значения `df_list` для позиции в списке `A`, которые равны `True`.

Применив нелинейную подгонку, мы найдем значения f, KSV_1 и KSV_2 для каждого значения ω, которое имеется в нашем распоряжении. Мы должны перебрать все значения ω в цикле и выполнить подгонку функции

$$\frac{\tan\theta(\omega, T = 45\ ^\circ\mathrm{C}, O_2)}{\tan\theta(\omega, T = 45\ ^\circ\mathrm{C}, O_2 = 0)}$$

относительно O_2 с использованием функции

```
def fitfunc(x, f, KSV, KSV2):
    return (f/(1.0+KSV*x)+ (1.0-f)/(1+KSV2*x))
```

для извлечения f, KSV_1 и KSV_2 для каждого значения O_2. Это делается с помощью приведенного ниже фрагмента кода:

```
f = []
KSV = []
KSV2 = []

for w_ in wred:
    # Подготовить файлы
    O2x = []
    tantheta = []
    #tantheta0 = float(dfref[dfref['w']==w_]['tantheta'])
    tantheta0 = float(dataref_[0][dataref_[0]['w']==w_]['tantheta'])

    # Перебрать файлы в цикле
    for idx, df_ in enumerate(data_train):
        O2xvalue = float(Odf.loc[idx])
        O2x.append(O2xvalue)
```

```
        tanthetavalue = float(df_[df_['w'] == w_]['tantheta'])
        tantheta.append(tanthetavalue)

    popt, pcov = curve_fit(fitfunc_2, O2x, np.array(tantheta)/tantheta0,
                                                    p0 = [0.4,0.06, 0.003])
    f.append(popt[0])
    KSV.append(popt[1])
    KSV2.append(popt[2])
```

Потратьте некоторое время на то, чтобы изучить этот фрагмент кода. Он несколько замысловат, потому что каждый файл содержит данные с фиксированным значением O_2. Для каждого значения частоты мы хотим построить массив, содержащий значения, подлежащие подгонке как функции от O_2. Именно по этой причине нам пришлось заниматься упорядочением данных. В списках f, KSV_1 и KSV_2 теперь у нас находятся значения, которые мы нашли относительно частоты. Давайте сначала отберем только те значения угловых частот, которые лежат между 3000 и 100 000.

```
w_ = w[4:20]
f_ = f[4:20]
KSV_ = KSV[4:20]
```

На рис. 9.7 вы видите, как f, KSV_1 и KSV_2 зависят от угловой частоты.

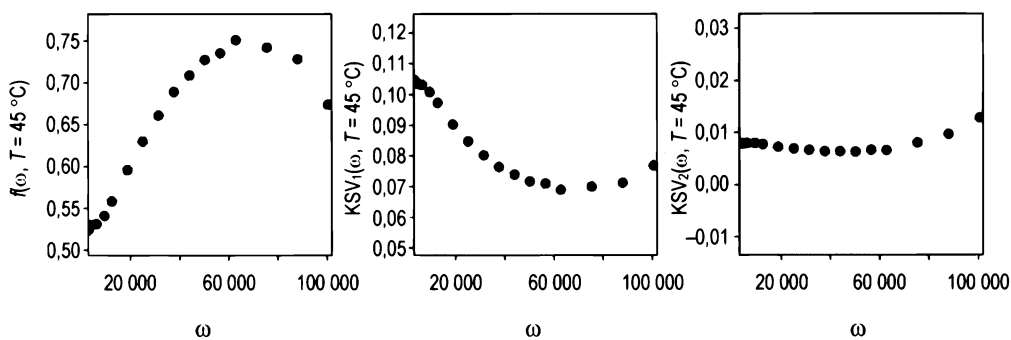


РИС. 9.7. Зависимость f, KSV_1 и KSV_2 от угловой частоты

Есть еще одна небольшая проблема, которую мы должны преодолеть. Мы должны быть в состоянии вычислять f, KSV_1 и KSV_2 для любого значения ω, а не только для тех, которые получили. Для этого мы должны применить интерполяцию. Ради экономии времени не будем разрабатывать интерполяционную функцию с нуля. Вместо этого воспользуемся функцией `interp1d` из программного пакета SciPy.

```
from scipy.interpolate import interp1d
```

И сделаем это следующим образом:

```
finter = interp1d(wred, f, kind='cubic')
KSVinter = interp1d(wred, KSV, kind = 'cubic')
KSV2inter = interp1d(wred, KSV2, kind = 'cubic')
```

Обратите внимание, что `finter`, `KSVinter` и `KSV2inter` представляют собой функции, которые на входе принимают значение ω в виде массива NumPy и возвращают соответственно значение f, KSV_1 и KSV_2. Непрерывная кривая на рис. 9.8 показывает интерполированные функции, полученные точками на рис. 9.7.

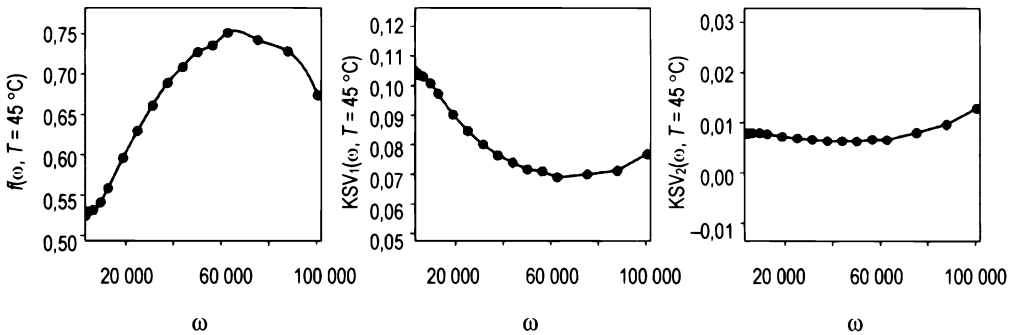


РИС. 9.8. Зависимость f, KSV_1 и KSV_2 от угловой частоты. Непрерывная кривая является интерполированной функцией

Итак, у нас есть все необходимые ингредиенты. И мы, наконец, можем создать тренировочный набор данных с помощью математической формулы

$$\frac{\tan\theta(\omega, T, O_2)}{\tan\theta(\omega, T, O_2 = 0)} = \frac{f(\omega, T)}{1 + KSV_1(\omega, T)\cdot O_2} + \frac{1 - f(\omega, T)}{1 + KSV_2(\omega, T)\cdot O_2}.$$

Теперь давайте создадим 5000 наблюдений для случайных значений O_2.

```
number_of_samples = 5000
number_of_x_points = len(w_)
np.random.seed(20)
O2_v = np.random.random_sample([number_of_samples])*100.0
```

Нам нужна приведенная выше математическая функция

```
def fitfunc2(x, O2, ffunc, KSVfunc, KSV2func):
    output = []
    for x_ in x:
        KSV_ = KSVfunc(x_)
        KSV2_ = KSV2func(x_)
        f_ = ffunc(x_)
        output_ = f_/(1.0+KSV_*O2)+(1.0-f_)/(1.0+KSV2_*O2)
        output.append(output_)
    return output
```

для того чтобы вычислить

$$\frac{\tan\theta(\omega, T, O_2)}{\tan\theta(\omega, T, O_2 = 0)}$$

для значения угловой частоты и O_2. Данные могут быть сгенерированы с помощью следующих строк кода:

```
data = np.zeros((number_of_samples, number_of_x_points))
targets = np.reshape(O2_v, [number_of_samples,1])

for i in range(number_of_samples):
    data[i,:] = fitfunc2(w_, float(targets[i]), finter, KSVinter, KSV2inter)
```

На рис. 9.9 представлено несколько произвольных примеров сгенерированных нами данных.

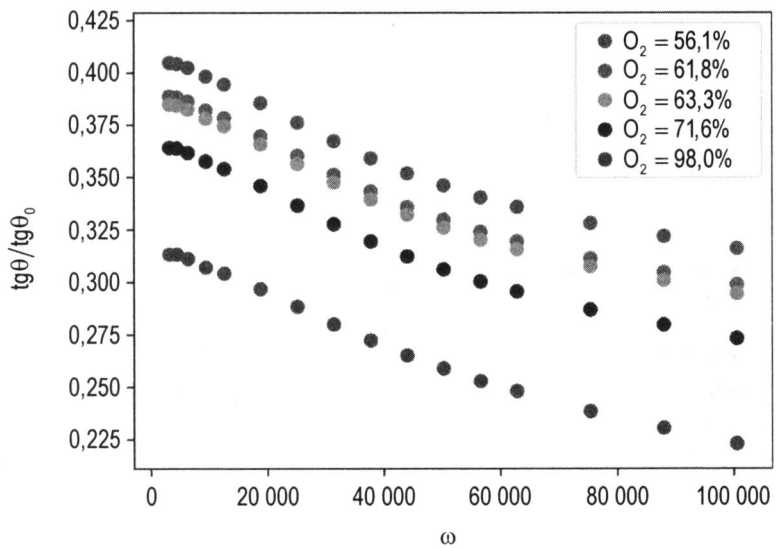


РИС. 9.9. Случайные образцы сгенерированных нами данных

Тренировка модели

Давайте начнем строить сеть. По причинам экономии пространства мы здесь ограничимся простой трехслойной сетью с пятью нейронами в каждом слое.

```
tf.reset_default_graph()

n1 = 5 # Число нейронов в слое 1
n2 = 5 # Число нейронов в слое 2
n3 = 5 # Число нейронов в слое 3
nx = number_of_x_points
n_dim = nx
n4 = 1

stddev_f = 2.0
```

```
tf.set_random_seed(5)

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [10, None])
W1 = tf.Variable(tf.random_normal([n1, n_dim], stddev=stddev_f))
b1 = tf.Variable(tf.constant(0.0, shape = [n1,1]) )
W2 = tf.Variable(tf.random_normal([n2, n1], stddev=stddev_f))
b2 = tf.Variable(tf.constant(0.0, shape = [n2,1]))
W3 = tf.Variable(tf.random_normal([n3,n2], stddev = stddev_f))
b3 = tf.Variable(tf.constant(0.0, shape = [n3,1]))
W4 = tf.Variable(tf.random_normal([n4,n3], stddev = stddev_f))
b4 = tf.Variable(tf.constant(0.0, shape = [n4,1]))
X = tf.placeholder(tf.float32, [nx, None]) # Входы
Y = tf.placeholder(tf.float32, [1, None])  # Метки

# Построим сеть
Z1 = tf.nn.sigmoid(tf.matmul(W1, X) + b1) # n1 x n_dim * n_dim x n_obs =
                                                                  n1 x n_obs
Z2 = tf.nn.sigmoid(tf.matmul(W2, Z1) + b2) # n2 x n1 * n1 * n_obs = n2 x n_obs
Z3 = tf.nn.sigmoid(tf.matmul(W3, Z2) + b3)
Z4 = tf.matmul(W4, Z3) + b4
y_ = Z2
```

По крайней мере, так было в самом начале разработки. В качестве выхода сети был выбран нейрон с активационной функцией в виде тождественного отображения `y_= Z2`. К сожалению, тренировка не работала и была очень нестабильной. Поскольку требовалось предсказать процентное соотношение, нужен был выход между 0 и 100. Поэтому было решено попробовать сигмоидальную активационную функцию, умноженную на 100.

```
y_ = tf.sigmoid(Z2)*100.0
```

И тренировка сразу же заработала великолепно. При этом был применен оптимизатор Adam.

```
cost = tf.reduce_mean(tf.square(y_-Y))
learning_rate = 1e-3
training_step = tf.train.AdamOptimizer(learning_rate).minimize(cost)
init = tf.global_variables_initializer()
```

На этот раз был использован мини-пакет размером 100 наблюдений.

```
batch_size = 100
sess = tf.Session()
sess.run(init)

training_epochs = 25000
cost_history = np.empty(shape=[1], dtype = float)
train_x = np.transpose(data)
train_y = np.transpose(targets)
```

```
cost_history = []

for epoch in range(training_epochs+1):
    for i in range(0, train_x.shape[0], batch_size):
        x_batch = train_x[i:i + batch_size,:]
        y_batch = train_y[i:i + batch_size,:]

        sess.run(training_step, feed_dict = {X: x_batch, Y: y_batch})

    cost_ = sess.run(cost, feed_dict={ X:train_x, Y: train_y})
    cost_history = np.append(cost_history, cost_)

    if (epoch % 1000 == 0):
        print("Достигнута эпоха",epoch,"стоимость J =", cost_)
```

Вы уже должны понимать этот код без лишних объяснений. Он представляет собой то, что мы использовали уже неоднократно. Возможно, вы заметили, что веса были инициализированы случайно. При этом было испробовано несколько стратегий, но оказалось, что эта сработала лучше всего. Очень поучительно проверить ход тренировки. На рис. 9.10 показана стоимостная функция, оцениваемая на тренировочном наборе данных и на экспериментальном рабочем наборе данных. Вы можете видеть, что она осциллирует. В основном это объясняется двумя причинами.

- Во-первых, мы используем мини-пакеты, и, следовательно, стоимостная функция осциллирует.
- Во-вторых, экспериментальные данные зашумлены, потому что измерительный аппарат не идеален. Например, газовый смеситель имеет абсолютную ошибку, равную примерно 1–2%, а значит, что если мы имеем экспериментальное

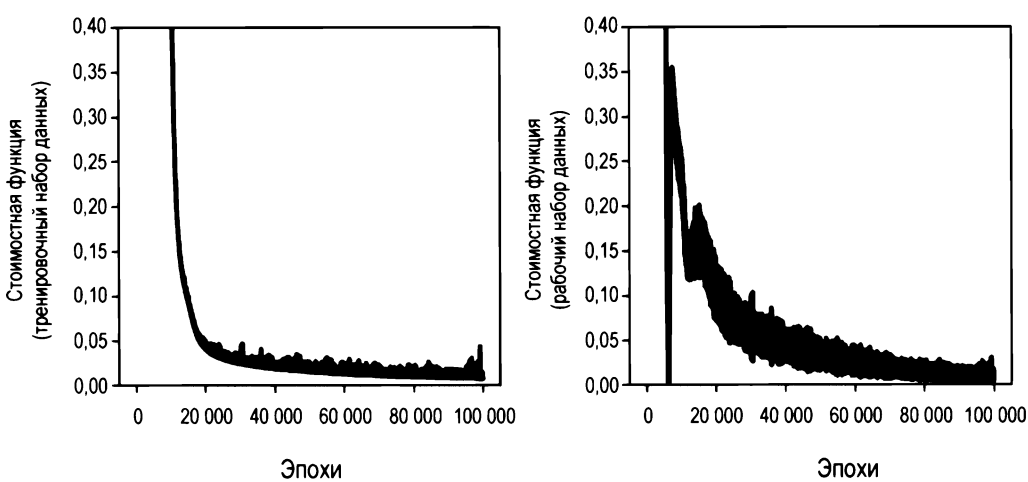


РИС. 9.10. Стоимостная функция в сопоставлении с эпохами, вычисленными на тренировочном и на рабочем наборах данных

наблюдение для $O_2 = 60\%$, то оно может падать до 58 или 59% и повышаться до 61 или 62%.

С учетом этой ошибки следует ожидать, что в предсказании абсолютная ошибка в среднем будет примерно равна 1%.

ПРИМЕЧАНИЕ. Грубо говоря, выход хорошо натренированной сети никогда не сможет превысить точность используемых целевых переменных. При оценивании точности целевых переменных не забывайте всегда проверять наличие ошибок, которые могут в них быть. Поскольку в предыдущем примере наши целевые значения O_2 имеют максимальную абсолютную ошибку ±1% (т. е. экспериментальную ошибку), ожидаемая ошибка результатов сети будет такого порядка величины.

ВНИМАНИЕ! Сеть заучит функцию, которая производит определенный выход с учетом конкретного входа. Если выход является неправильным, то заученная функция тоже будет неправильной.

Наконец, давайте проверим результативность данной сети.

Вы должны иметь в виду, что наш рабочий набор данных — совсем крошечный, что делает осцилляции еще более заметными. На рис. 9.11 вы видите предсказываемое для O_2 значение в сопоставлении с измеренным значением. Как видите, они красиво укладываются по диагонали.

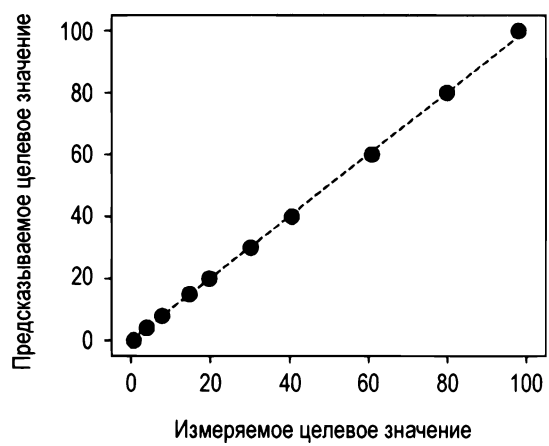


РИС. 9.11. Предсказываемое для O_2 значение в сопоставлении с измеренным значением

На рис. 9.12 показана абсолютная ошибка, вычисленная на рабочем наборе данных для $O_2 \in [0; 100]$.

Полученные результаты потрясающе хороши. Для всех значений O_2, за исключением 100%, ошибка лежит ниже 1%. Напомним, что наша сеть училась на искусст-

венно созданном наборе данных. Данная сеть теперь может использоваться в этом типе датчиков без необходимости реализации сложных математических уравнений для оценивания O_2. Следующий этап этого проекта будет автоматически получать примерно 10 000 мерных величин при различных значениях температуры T и концентрации кислорода O_2 и использовать эти величины в качестве тренировочного набора для предсказания T и O_2 одновременно.

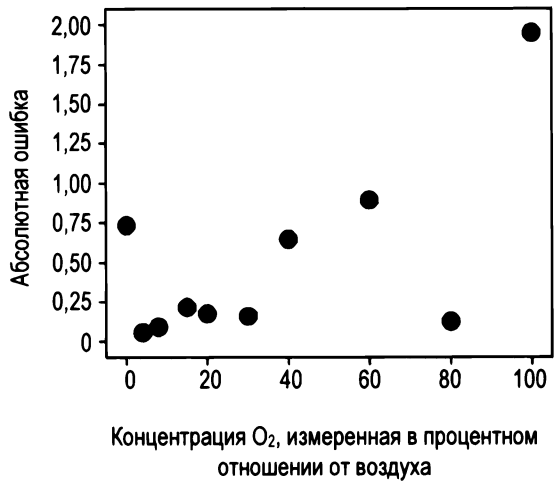


РИС. 9.12. Абсолютная ошибка, вычисленная на рабочем наборе данных для $O_2 \in [0; 100]$

ГЛАВА 10

Логистическая регрессия с нуля

В *главе 2* мы разработали логистическую регрессионную модель для бинарной классификации с одним нейроном и применили ее к двум цифрам набора данных MNIST. Фактический Python-код для построения вычислительного графа состоял всего из десяти строк (за исключением той части, которая выполняет тренировку модели; просмотрите *главу 2*, если вы не помните, что мы там делали).

```
tf.reset_default_graph()

X = tf.placeholder(tf.float32, [n_dim, None])
Y = tf.placeholder(tf.float32, [1, None])
learning_rate = tf.placeholder(tf.float32, shape=())

W = tf.Variable(tf.zeros([1, n_dim]))
b = tf.Variable(tf.zeros(1))

init = tf.global_variables_initializer()
y_ = tf.matmul(tf.transpose(W),X)+b
cost = tf.reduce_mean(tf.square(y_-Y))
training_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Этот код компактен и пишется очень быстро. На самом деле за кулисами происходит много разных вещей, которые не видны из этого Python-кода. TensorFlow на заднем плане проделывает большой объем работы, о которой вы можете не знать. И было бы очень поучительно попытаться разработать эту точную модель совершенно с нуля, математически и на Python, без использования TensorFlow, для того чтобы понаблюдать за тем, что происходит на самом деле. В следующих разделах излагается весь требуемый математический каркас и его реализация на Python (с использованием только библиотеки NumPy). Наша цель состоит в том, чтобы построить модель, которую можно было бы натренировать делать бинарную классификацию.

Мы не будем слишком долго останавливаться на математических обозначениях или идеях, лежащих в основе математического каркаса, потому что вы видели их несколько раз в предыдущих главах. Обсуждение вопроса подготовки набора данных также будет очень кратким, т. к. он уже был рассмотрен в *главе 2*.

Мы не будем построчно анализировать весь Python-код, потому что если вы прочли предыдущие главы, то должны иметь довольно хорошее представление об используемых здесь практических решениях и идеях. Каких-то новых понятий здесь нет; почти все из них вы уже встречали. Рассмотрим эту главу как справочный материал по реализации логистической модели абсолютно с нуля. Настоятельно рекомендуется попробовать разобраться во всем математическом каркасе и реализовать его на Python. Этот материал является очень показательным и многому научит относительно отладки и того, как важно писать хороший код и насколько удобны такие библиотеки, как TensorFlow.

Математический каркас логистической регрессии

Давайте начнем с некоторых обозначений и напоминаний о том, что именно мы собираемся делать. Нашим предсказанием будет переменная $\hat{y}$, которая может быть равна только 0 или 1. (Мы будем обозначать 0 и 1 два класса, которые мы пытаемся предсказать.)

$$\text{Предсказание} \rightarrow \hat{y} \in \{0, 1\}.$$

В качестве выхода, или предсказания, наш метод будет выдавать вероятность того, что предсказание $\hat{y} = 1$ при наличии случая x на входе. Или, в более математической форме,

$$\hat{y} = P(y = 1 \mid x).$$

Затем мы определим входное наблюдение как принадлежащее классу 1, если $\hat{y} > 0{,}5$, и классу 0, если $\hat{y} < 0{,}5$. Как и в *главе 2* (см. рис. 2.2), мы рассмотрим n_x входов и один нейрон с сигмоидальной (обозначаемой буквой σ) активационной функцией. Выход $\hat{y}$ нейрона для наблюдения i можно легко записать как

$$\hat{y}^{[i]} = \sigma\left(z^{[i]}\right) = \sigma\left(\mathbf{w}^T \mathbf{x}^{[i]} + b\right) = \sigma\left(w_1 x_1^{[i]} + w_2 x_2^{[i]} + \ldots + w_{n_x} x_{n_x}^{[i]} + b\right).$$

Для того чтобы найти наилучшие веса и смещения, мы минимизируем стоимостную функцию, которая написана здесь для одного наблюдения

$$\mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right) = -\left(y^{[i]} \log \hat{y}^{[i]} + \left(1 - y^{[i]}\right) \log\left(1 - \hat{y}^{[i]}\right)\right),$$

где через y мы обозначили метки. Мы будем использовать алгоритм градиентного спуска, как описано в *главе 2*, поэтому нам понадобятся частные производные стоимостной функции относительно весов и смещения. Вы помните, что на итера-

ции $n+1$ (будем обозначать итерацию нижним индексом в квадратных скобках) мы будем обновлять веса из итерации n, используя уравнения

$$w_{j,[n+1]} = w_{j,[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j}$$

и для смещения

$$b_{[n+1]} = b_{j,[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b}$$

где γ — темп заучивания. Производные не особо сложны и могут быть легко вычислены по цепному правилу

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial w_j}$$

так же, как и b

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} \frac{d\hat{y}^{[i]}}{dz^{[i]}} \frac{\partial z^{[i]}}{\partial b}.$$

Теперь, вычисляя производные, вы можете проверить, что

$$\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}} = \frac{y^{[i]}}{\hat{y}^{[i]}} + \frac{1-y^{[i]}}{1-\hat{y}^{[i]}};$$

$$\frac{d\hat{y}^{[i]}}{dz^{[i]}} = \hat{y}^{[i]}\left(1-\hat{y}^{[i]}\right);$$

$$\frac{\partial z^{[i]}}{\partial w_j} = x_j^{[i]};$$

$$\frac{\partial z^{[i]}}{\partial b} = 1.$$

Когда мы соберем все это вместе, то получим:

$$w_{j,[n+1]} = w_{j,[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = w_{j,[n]} - \gamma\left(1-\hat{y}^{[i]}\right)x_j^{[i]};$$

$$b_{[n+1]} = b_{[n]} - \gamma \frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = b_{j,[n]} - \gamma\left(1-\hat{y}^{[i]}\right).$$

Эти уравнения справедливы только для одного тренировочного случая; следовательно, как мы уже делали, давайте обобщим их на многочисленные тренировочные случаи, помня, что мы определяем стоимостную функцию J для многочисленных наблюдений как

$$J(\mathbf{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right),$$

где, как обычно, мы обозначили число наблюдений через m. Выделенная жирным буква $\mathbf{w}$ — это вектор всех весов $\mathbf{w} = (w_1, w_2, ..., w_n)$. Нам здесь также понадобится всеми любимый матричный формализм (который вы несколько раз встречали в предыдущих главах):

$$\mathbf{Z} = \mathbf{W}^T\mathbf{X} + \mathbf{B},$$

где через $\mathbf{B}$ мы обозначили матрицу размерности $(1, n_x)$ (с тем, чтобы соблюсти совместимость с используемыми здесь обозначениями), в которой все элементы равны b (нам не придется ее определять на Python, потому что механизм транслирования об этом позаботится за нас). $\mathbf{X}$ будет содержать наблюдения и признаки и иметь размерность (n_x, m) (наблюдения в столбцах, признаки в строках), а $\mathbf{W}^T$ будет содержать результат транспонирования весовой матрицы, которая в наших случаях имеет размерность $(1, n_x)$, потому что она транспонирована. Выход нейрона в матричной форме будет равняться

$$\hat{\mathbf{Y}} = \sigma(\mathbf{Z}),$$

где сигмоидальная функция действует поэлементно. Уравнения для частных производных теперь становятся

$$\frac{\partial J(\mathbf{w}, b)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{y}^{[i]}}\frac{d\hat{y}^{[i]}}{dz^{[i]}}\frac{\partial z^{[i]}}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}\left(1-\hat{y}^{[i]}\right)x_j^{[i]}$$

и для b

$$\frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial \mathcal{L}\left(\hat{y}^{[i]}, y^{[i]}\right)}{\partial \hat{\mathbf{y}}^{[i]}}\frac{d\hat{y}^{[i]}}{dz^{[i]}}\frac{\partial z^{[i]}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(1-\hat{y}^{[i]}\right)$$

Эти уравнения могут быть записаны в матричном виде (где $\nabla_{\mathbf{w}}$ обозначает градиент по $\mathbf{w}$) как

$$\nabla_{\mathbf{w}} J(\mathbf{w}, b) = \frac{1}{m}\mathbf{X}\left(\hat{\mathbf{Y}} - \mathbf{Y}\right)^T$$

и для b

$$\frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{\mathbf{Y}}_i - \mathbf{Y}_i\right)^T.$$

Наконец, уравнение, которое нам нужно реализовать для алгоритма градиентного спуска, равно

$$\mathbf{w}_{[n+1]} = \mathbf{w}_{[n]} - \gamma\frac{1}{m}\mathbf{X}\left(\hat{\mathbf{Y}} - \mathbf{Y}\right)^T$$

и для b

$$b_{[n+1]} = b_{[n]} - \gamma\frac{1}{m}\sum_{i=1}^{n}\left(\hat{\mathbf{Y}}_i - \mathbf{Y}_i\right)^T.$$

К этому моменту вы уже должны были обрести совершенно иной взгляд на библиотеку TensorFlow. Данная библиотека проделывает все это за вас на заднем плане и, что более важно, абсолютно автоматически. Напомним: здесь мы имеем дело только с одним нейроном. Вы можете легко убедиться сами, насколько сложно это окажется, когда вы захотите вычислить те же самые уравнения для сетей с многочисленными слоями и нейронами или для чего-то вроде сверточной или рекуррентной нейронной сети.

Теперь у нас имеется весь математический каркас, необходимый для реализации с нуля логистической регрессии. Давайте перейдем к программированию на Python.

Реализация на Python

Начнем с импорта необходимых библиотек.

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

Обратите внимание, что мы не импортируем TensorFlow. Здесь эта библиотека нам не понадобится. Давайте напишем Python-функцию для сигмоидальной активационной функции `sigmoid(z)`.

```
def sigmoid(z):
    s = 1.0 / (1.0 + np.exp(-z))
    return s
```

Нам также потребуется функция для инициализации весов. В данном случае мы можем просто проинициализировать все нулями. Логистическая регрессия все равно будет работать.

```
def initialize(dim):
    w = np.zeros((dim,1))
    b = 0
    return w,b
```

Затем мы должны реализовать следующие уравнения, которые мы рассчитали в предыдущем разделе:

$$\nabla_{\mathbf{w}} J(\mathbf{w}, b) = \frac{1}{m}\mathbf{X}\left(\hat{\mathbf{Y}} - \mathbf{Y}\right)^T$$

и

$$\frac{\partial J(\mathbf{w}, b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\hat{\mathbf{Y}}_i - \mathbf{Y}_i\right).$$

```
def derivatives_calculation(w, b, X, Y):
    m = X.shape[1]
    z = np.dot(w.T,X)+b
    y_ = sigmoid(z)
```

```
    cost = -1.0/m*np.sum(Y*np.log(y_)+(1.0-Y)*np.log(1.0-y_))

    dw = 1.0/m*np.dot(X, (y_-Y).T)
    db = 1.0/m*np.sum(y_-Y)

    derivatives = {"dw": dw, "db":db}
    return derivatives, cost
```

Теперь нам нужна функция, которая будет обновлять веса.

```
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost=False):
    costs = []
    for i in range(num_iterations):
        derivatives, cost = derivatives_calculation(w, b, X, Y)
        dw = derivatives ["dw"]
        db = derivatives ["db"]

        w = w - learning_rate*dw
        b = b - learning_rate*db

        if i % 100 == 0:
            costs.append(cost)

        if print_cost and i % 100 == 0:
            print ("Стоимость (итерация %i) = %f" %(i, cost))

    derivatives = {"dw": dw, "db": db}
    params = {"w": w, "b": b}

    return params, derivatives, costs
```

Следующая далее функция `predict()` создает матрицу размерности (1, *m*), содержащую предсказания модели с учетом входов **w** и *b*.

```
def predict (w, b, X):
    m = X.shape[1]

    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0],1)
    A = sigmoid (np.dot(w.T, X)+b)

    for i in range(A.shape[1]):
        if (A[:,i] > 0.5):
            Y_prediction[:, i] = 1
        elif (A[:,i] <= 0.5):
            Y_prediction[:, i] = 0
    return Y_prediction
```

Наконец, давайте соберем все вместе в функции `model()`.

```
def model (X_train, Y_train, X_test, Y_test, num_iterations=1000,
           learning_rate=0.5, print_cost=False):
    w, b = initialize(X_train.shape[0])
    parameters, derivatives, costs = optimize(w, b, X_train, Y_train,
                                              num_iterations,
                                              learning_rate, print_cost)
    w = parameters["w"]
    b = parameters["b"]

    Y_prediction_test = predict (w, b, X_test)
    Y_prediction_train = predict (w, b, X_train)

    train_accuracy = 100.0 - np.mean(np.abs(Y_prediction_train-Y_train)*100.0)
    test_accuracy = 100.0 - np.mean(np.abs(Y_prediction_test-Y_test)*100.0)

    d = {"costs":costs, "Y_prediction_test":Y_prediction_test,
         "Y_prediction_train":Y_prediction_train,
         "w":w, "b":b, "learning_rate":learning_rate,
         "num_iterations":num_iterations}

    print ("Тренировочная точность: ", train_accuracy)
    print ("Тестовая точность: ", test_accuracy)

    return d
```

Тестирование модели

После построения модели мы должны проверить, каких результатов она может достигать при подаче в нее данных. В следующем разделе мы сначала подготовим набор данных, который уже использовали в *главе 2* (две цифры — единицу и двойку) из набора данных MNIST, а затем натренируем наш нейрон на этом наборе данных и посмотрим, какие результаты получим.

Подготовка набора данных

В качестве оптимизационной метрики мы выбрали точность, поэтому посмотрим, какое ее значение мы можем достичь с нашей моделью. Мы будем использовать тот же набор данных, что и в *главе 2*: подмножество набора данных MNIST, состоящее из цифр 1 и 2. Здесь вы найдете исходный код получения данных без объяснения причин, потому что мы уже подробно рассмотрели его в *главе 2*.

Нам потребуется следующий исходный код:

```
from sklearn.datasets import fetch_mldata
# mnist = fetch_mldata('MNIST original') # Устарело
# X,y = mnist["data"], mnist["target"]
```

```
X_mnist, y_mnist = fetch_openml('mnist_784', version=1 , return_X_y=True)
X, y = X_mnist, y_mnist.astype(float)

X_12 = X[np.any([y == 1,y == 2], axis = 0)]
y_12 = y[np.any([y == 1,y == 2], axis = 0)]
```

Поскольку мы загрузили все изображения одним блоком, мы должны создать тренировочный и рабочий наборы данных (разбив в пропорции 80% для тренировки и 20% для разработки) следующим образом:

```
shuffle_index = np.random.permutation(X_12.shape[0])
X_12_shuffled, y_12_shuffled = X_12[shuffle_index], y_12[shuffle_index]

train_proportion = 0.8
train_dev_cut = int(len(X_12)*train_proportion)
X_train, X_dev, y_train, y_dev = (
    X_12_shuffled[:train_dev_cut],
    X_12_shuffled[train_dev_cut:],
    y_12_shuffled[:train_dev_cut],
    y_12_shuffled[train_dev_cut:] )
```

Как обычно, мы нормализуем входы:

```
X_train_normalised = X_train/255.0
X_dev_normalised = X_dev/255.0
```

приводим матрицы в правильный формат:

```
X_train_tr = X_train_normalised.transpose()
y_train_tr = y_train.reshape(1,y_train.shape[0])
X_dev_tr = X_dev_normalised.transpose()
y_dev_tr = y_dev.reshape(1,y_dev.shape[0])
```

и определяем пару констант:

```
dim_train = X_train_tr.shape[1]
dim_dev = X_dev_tr.shape[1]
```

Теперь давайте сдвинем наши метки (причина сдвига объяснялась в *главе 2*). Мы имеем 1 и 2, и нам нужны 0 и 1.

```
y_train_shifted = y_train_tr - 1
y_test_shifted = y_dev_tr - 1
```

Выполнение теста

Наконец, мы можем протестировать модель, вызвав ее следующим образом:

```
d = model (X_train_tr, y_train_shifted, X_dev_tr, y_test_shifted,
           num_iterations=4000, learning_rate=0.05, print_cost=True)
```

Хотя ваши числа могут отличаться, вы должны получить результат, подобный приведенному ниже, в котором опущено несколько итераций по причинам экономии пространства:

```
Стоимость (итерация 0) = 0.693147
Стоимость (итерация 100) = 0.109078
Стоимость (итерация 200) = 0.079466
Стоимость (итерация 300) = 0.067267
Стоимость (итерация 400) = 0.060286
.........
Стоимость (итерация 3600) = 0.031350
Стоимость (итерация 3700) = 0.031148
Стоимость (итерация 3800) = 0.030955
Стоимость (итерация 3900) = 0.030769
Тренировочная точность: 99.1003111074
Тестовая точность: 99.092131809
```

Совсем неплохой результат[1].

Заключение

Вам следует попытаться по-настоящему разобраться во всех шагах, рассмотренных в этой главе для того, чтобы понять весь тот объем работы, который делается за вас библиотекой TensorFlow. Напомним, что перед нами невероятно простая модель с одним-единственным нейроном. Теоретически вы могли бы написать все уравнения для более сложных сетевых архитектур, но это было бы очень трудоемко и чрезвычайно подвержено ошибкам. TensorFlow вычисляет все производные за вас независимо от сложности сети. Если вам интересно узнать, что вообще TensorFlow может делать, то рекомендуется почитать официальную документацию, доступную по адресу **https://goo.gl/E5DpHK**.

ПРИМЕЧАНИЕ. Теперь вы должны оценить библиотеку TensorFlow по достоинству и отдавать себе отчет в том, что при ее использовании происходит огромная работа на заднем плане. Вы также теперь должны осознавать сложность вычислений и важность понимания деталей алгоритмов и того, как они реализованы, для того, чтобы уметь оптимизировать и отлаживать свои модели.

[1] Если вы задаетесь вопросом "Почему мы получаем результат, который отличается от результата в *главе 2*?", то напомним, что для заучивания параметров мы используем несколько иной тренировочный набор, что и приводит к этой разнице.

Предметный указатель

K

K-блочная перекрестная проверка
◊ Scikit-Learn 240, 241, 246
◊ Xinputfold и yinputfold 241
◊ библиотеки 240
◊ логистическая регрессия 241, 243
◊ массивы 241
◊ наблюдения 240
◊ набор данных MNIST 240
◊ нормализация данных 242
◊ оптимизатор Adam 243
◊ псевдокод 239, 244, 245
◊ сбалансированный набор данных 242
◊ стандартное отклонение 246
◊ тренировочный набор и рабочий набор, значения точности 245, 246

L

LeNet-5, нейросеть 317

T

TensorFlow:
◊ вычислительные графы:
  ▫ tf.constant 40
  ▫ tf.placeholder 42–44
  ▫ tf.variable 40, 41
  ▫ входные величины 36
  ▫ выполнение и вычисление 44, 45
  ▫ закрепление значений 37
  ▫ нейронная сеть 37
  ▫ переменные 35
  ▫ построение и вычисление узлов 45
  ▫ создание и закрытие сеанса 46, 47
  ▫ суммирование двух переменных 36
  ▫ суммирование двух тензоров 39
◊ кодирование с одним активным состоянием 114
◊ линейная регрессия *См. Регрессия линейная*
◊ построение модели 116–118
◊ сетевая архитектура:
  ▫ tf.nn.softmax() 113
  ▫ скрытый слой 94, 111
  ▫ функция softmax 112
◊ тензоры 38–39
◊ установка 31, 32

А

Анализ метрический:
◊ анализ ошибок 208
◊ диаграмма MAD 215, 216
◊ метрики прецизионности, полноты и F1 227–232
◊ наборы данных
  ▫ MNIST 233
  ▫ Xtrain, Xdev и Xtraindev 236
  ▫ диаграмма MAD 237
  ▫ единственный нейрон 235
  ▫ источники 233
  ▫ массивы 234
  ▫ матрицы 234
  ▫ наблюдения 233, 234
  ▫ построение модели 235
  ▫ профессиональная зеркальная фотокамера DSLR и смартфон 232
  ▫ случайное изображение и его сдвинутая версия 235
  ▫ технические решения, несоответствие данных 238
  ▫ тренировка модели 221
  ▫ тренировочный и рабочий 237
◊ описание 207
◊ разбиение набора данных
  ▫ наблюдения 219, 221
  ▫ набор данных MNIST 220
  ▫ рабочий и тестовый наборы данных 219, 220
  ▫ тренировочный и рабочий наборы данных 221
◊ ручной:
  ▫ массив одномерный, значения оттенков серого 247–250
  ▫ сеть натренированная 250
  ▫ точность 247
  ▫ характеристики данных 250
◊ смещение 214, 215
◊ тестовый набор 217–219
  ▫ переподгонка к тренировочному набору данных 216, 217
◊ человеческая результативность
  См. Человеческая результативность
Архитектура сетевая:
◊ выход нейронов 96, 97
◊ гиперпараметры 98
◊ матрица весовая 95
◊ матрица смещений 96
◊ представление графическое 94
◊ размерности матриц 97
◊ сеть обобщенная 95
◊ слои входные и выходные 111
◊ функция softmax 94, 99, 100

Б

Байесова оптимизация:
◊ верхняя доверительная граница (UCB) 277
◊ гауссов процесс 270
◊ предсказание с помощью гауссовых процессов 271–277
◊ процесс стационарный 271
◊ регрессия Надарая – Ватсона 269
◊ функция:
  ▫ обнаружения 277, 279
  ▫ суррогатная 279
  ▫ тригонометрическая 278
  ▫ черно-ящичная 279, 280
Байесова ошибка 213
Блокнот Jupyter:
◊ документация 33, 35
◊ кнопка New 34
◊ описание 33
◊ открытие 33
◊ пустая страница 34

В

Вариации градиентного спуска:
◊ 100 эпох 123
◊ гиперпараметры 125
◊ мини-пакетный 123, 127
  ▫ градиентный спуск 121, 122
◊ пакетный 119, 120
◊ стоимостная функция:
  ▫ время работы 123–125
  ▫ размер мини-пакета 123
◊ стохастический градиентный спуск 120, 121
◊ функция model() и ее параметры 126, 127
Верхняя доверительная граница (UCB) 277

Г

Гауссовы процессы 270
◊ предсказание 271–277
Гиперболический тангенс (tanh) 58
Гиперпараметр 65, 98

Гиперпараметрическая настройка:
◊ байесова оптимизация *См. Байесова оптимизация*
◊ варианты оптимизаторов 256
◊ категории 256
◊ логарифмическая шкала 285–287
◊ метод:
  ▫ инициализации весов 256
  ▫ ослабления темпа заучивания 256
  ▫ регуляризационный 256
◊ набор данных Zalando *См. Набор данных Zalando*
◊ оптимизация с переходом от крупнозернистости к мелкозернистости 265–268
◊ поиск:
  ▫ решеточный 258–262
  ▫ случайный 262–265
◊ размер мини-пакета 256
◊ слои и нейроны 256
◊ функция:
  ▫ радиально-базисная 294
  ▫ активационная 256
◊ черно-ящичная оптимизация *См. Черно-ящичная оптимизация*
◊ число эпох 256
Граф вычислительный 51

Д

Диаграмма метрического анализа (MAD) 215, 216, 237
Дисперсия 105
Долгая краткосрочная память (LSTM) 329

З

Заполнение 314

И

Исследовательский проект:
◊ концентрация газа 331
◊ люминесцентное гашение 332, 333
◊ математические модели 334
◊ подготовка набора данных:
  ▫ записи файловые 342
  ▫ кадры данных 341, 343
  ▫ набор данных тренировочный 346
  ▫ подгонка нелинейная 344
  ▫ примеры произвольные 347
  ▫ сети нейронные 340
  ▫ страница официальной документации 344
  ▫ температура и концентрация кислорода 340, 341, 351
  ▫ функция интерполяционная 345
  ▫ функция математическая 346
  ▫ частоты угловые 332, 342, 345
◊ регрессионная задача:
  ▫ значение случайное 336
  ▫ наблюдения 336
  ▫ набор данных рабочий 336, 339
  ▫ набор данных тренировочный 335
  ▫ предсказанные значения в сопоставлении с реальными 339
  ▫ произвольные примеры функций 337
  ▫ сеть нейронная 335
  ▫ сеть простая 337
  ▫ спуск мини-пакетный градиентный 338
  ▫ функция лоренцева 335
  ▫ функция стоимостная 338
◊ сенсорные устройства 331
◊ тренировка модели:
  ▫ абсолютная ошибка, кислород 349
  ▫ концентрация 350
  ▫ нейроны 347
  ▫ оптимизатор Adam 348
  ▫ предсказанное значение для O_2 в сопоставлении с измеренным значением 350
  ▫ размер мини-пакетов 348
  ▫ сигмоидальная активационная функция 348
  ▫ стоимостная функция в сопоставлении с эпохами 350

М

Массив, транслирование 57
Метрика:
◊ разумно достаточная 80
◊ оптимизационная 80

Н

Набор данных MNIST 82
Набор данных Zalando 108–111, 159–160, 172
◊ CSV-файлы 108

◊ data_train.head() 110
◊ data_train['label'] 110
◊ веб-сайт и конкурс kaggle 108
◊ гиперпараметрическая настройка:
  ▫ build_model(number_neurons) 288
  ▫ CSV-файлы 287
  ▫ библиотеки 287
  ▫ выполнение модели 290
  ▫ массив:
    • data_train 287
    • NumPy 288
  ▫ набор данных:
    • рабочий 288
    • тестовый в сопоставлении с числом нейронов 291
  ▫ поиск:
    • решеточный 290
    • случайный 292
  ▫ тензор стоимостной 289
  ▫ точность на тренировочном и тестовом наборах данных 291
  ▫ функции 288
◊ классы 108
◊ лицензия MIT 108
◊ набор данных MNIST 108
◊ образцы тренировочный и тестовый 108
◊ тензор с метками 111
◊ функции NumPy 109
Навигатор Anaconda:
◊ NumPy 29, 30
◊ Python-пакеты 26
◊ TensorFlow *См. TensorFlow*
◊ блокнот Jupyter 33–35
◊ кнопка Create 27
◊ команда conda 27
◊ панель навигационная:
  ▫ левая 27
  ▫ средняя 27
◊ раскрывающееся меню Not Installed (Не установлены) 29
◊ скачивание и установка 25
◊ установка программных пакетов 29
◊ экран 26
Нейрон:
◊ граф вычислительный 51
◊ обозначение матричное 52
◊ оптимизация на основе градиентного спуска 49
◊ представление 49–52
◊ реализация в TensorFlow 86–90
◊ регрессия:
  ▫ логистическая *См. Регрессия логистическая*
  ▫ линейная *См. Регрессия линейная*
◊ стоимостная функция и градиентный спуск 63–65
◊ структура 49
◊ темп заучивания:
  ▫ алгоритм градиентного спуска 66, 69–70
  ▫ функция стоимостная 65, 68
    • в сопоставлении с числом итераций 69
◊ функция:
  ▫ активационная 50
    • ArcTan 62
    • ELU 62
    • ReLU 58
    • ReLU с утечкой 60
    • Softplus 62
    • Swish 61
    • tanh 58
    • сигмоидальная 56–58
    • тождественное отображение 55
  ▫ передаточная 50
◊ циклы и библиотека NumPy 53–55
Нейронные сети прямого распространения:
◊ TensorFlow *См. TensorFlow*
◊ архитектура *См. Сетевая архитектура*
◊ добавление слоев 130–132
◊ инициализация весов 128–130
◊ набор данных Zalando *См. Набор данных Zalando*
◊ неправильные предсказания 127, 128
◊ описание 93
◊ переподгонка *См. Переподгонка*
◊ практический пример 97
◊ скрытые слои 132, 133
◊ сравнений сетей 133–137
Нейросеть LeNet-5 317
Норма ℓ_p 185

О

Оптимизатор 76
◊ Adam 170–172
◊ momentum 164–167
  ▫ TensorFlow 165
  ▫ градиентный спуск 164
  ▫ график трехмерной поверхности, стоимостная функция 167
  ▫ стоимостная функция vs, число эпох 165

Оптимизатор *(прод.)*
◊ momentum *(прод.)*
▫ траектория 167
▫ экспоненциально взвешенные средние 160
◊ RMSProp 167–170
◊ набор данных Zalando 172
◊ собственной разработки 173–177
◊ экспоненциально взвешенные средние 160–163
Оптимизация на основе градиентного спуска 49
Ослабление:
◊ обратно-временное 146–149
◊ пошаговое 143–146
◊ ступенчатое 142, 143
◊ экспоненциальное 149, 150
▫ естественное 150–155
◊ динамическое темпа заучивания:
▫ алгоритм градиентного спуска 139
▫ естественное экспоненциальное 150–155
▫ итерации/эпохи 141
▫ набор данных Zalando 159, 160
▫ обратно-временное 146–149
▫ пошаговое 143–146
▫ реализация в TensorFlow 155–158
▫ ступенчатое 142, 143
▫ экспоненциальное 149, 150
Ошибка байесова 209

П

Переменная целевая 73
Переподгонка:
◊ анализ ошибок 106, 107
◊ данные 101
◊ к тренировочному набору данных 216–217
◊ массив NumPy 101
◊ многочлен:
▫ 21-й степени 103
▫ второй степени 101, 103
◊ модель линейная 102, 104
◊ ошибка среднеквадратическая 100
◊ параметры 100
◊ смещение и дисперсия 105
◊ точки двумерные 100
◊ функция curve_fit 101
Показатель одиночный оценочный метрический 79
Процесс стационарный 271

Р

Распределение несбалансированных класов 223–227
◊ выборка данных:
▫ повышенная 227
▫ пониженная 227
◊ выполнение модели 224
◊ изменение метрики 226
◊ матрица для меток 225
◊ наблюдения 227
◊ набор данных MNIST 223
◊ наборы данных тренировочный и рабочий 224
◊ нейрон единственный 224
◊ получение большего объема данных из класса с меньшим числом наблюдений 227
◊ регрессия логистическая 223
◊ точность 225
Регрессия:
◊ линейная:
▫ NumPy 71
▫ векторы и матрицы 71
▫ метрика оптимизационная 80
▫ метрика разумно достаточная 80
▫ набор данных 72, 74
▫ наблюдения 70
▫ нейрон и стоимостная функция:
• MSE (среднеквадратическая ошибка) 75
• бостонский набор данных 72
• выход инструкции 78
• исходный код TensorFlow 75
• предсказанное целевое значение в сопоставлении с измеренным целевым значением 78, 79
• темп заучивания 76
• тождественное отображение как активационная функция 75–79
• число наблюдений 75
▫ показатель одиночный оценочный метрический 79
▫ признаки и наблюдения 71
▫ функция стоимостная 80
◊ логистическая 81
▫ MNIST 82
▫ TensorFlow 357

- алгоритм градиентного спуска 356
- веса и смещение, стоимостная функция 354–357
- итерации 360
- конструирование вычислительного графа, Python-код 353
- набор данных 82–86
- набор данных MNIST 353, 359
- подготовка набора данных 359, 360
- предсказание 354
- реализация на Python 357–359
- функция:
  - активационная 82
  - сигмоидальная активационная 354
  - стоимостная 81

◊ Надарая – Ватсона 269
Регуляризация:
◊ дополнительные методы 204
◊ наборы данных тренировочный и рабочий, MSE 202
◊ норма (p 185
◊ определение 183–185
◊ отсев:
- код конструкционный исходный 200
- наборы данных тренировочный и рабочий, MSE 201
- параметр keep_prob 200
- предсказания, рабочий набор данных 200
- фаза тренировочная 200
- функция стоимостная 201

◊ переподгонка 182, 204
◊ сети сложные:
- MSE, тренировочный и рабочий наборы данных 182
- анализ ошибок 182
- массив NumPy целевой 180
- набор данных бостонский с ценами на жилую недвижимость 179
- наборы данных тренировочный и рабочий 180
- оптимизатор Adam 181
- пакеты программные 179
- функции активационные ReLU 180

◊ сложность сетевая 185
Регуляризация (1
◊ веса в сопоставлении с эпохами 197–199
◊ процентное соотношение весов меньше 1e-3 196
◊ реализация в TensorFlow 196
◊ функция стоимостная 196
Регуляризация (2
◊ алгоритм градиентного спуска 186
◊ граница принятия решения 193–195
◊ лямбда 187
◊ наборы данных тренировочный и рабочий 188, 191
◊ процентное соотношение весов меньше 1e-3 191
◊ распределение весов 189
◊ реализация в TensorFlow 187
◊ режим переподгонки 189, 191
◊ функция стоимостная 185, 187, 193
◊ число заучиваемых параметров 189
◊ эффекты 192
Рекуррентные нейронные сети (RNN-сети)
◊ LSTM-память 329
◊ TensorFlow 326
◊ анализ метрический 326
◊ генерирование аннотаций на изображениях 323
◊ генерирование текста 323
◊ набор данных MNIST 326
◊ наборы тренировочный и рабочий 326
◊ обозначения 324, 325
◊ описание 322
◊ перевод машинный 323
◊ переменные целевые 327
◊ представление схематическое 324
◊ распознавание речи 323
◊ сети полносвязные 325, 329
◊ собеседники виртуальные (чатботы) 323
◊ состояние внутренней памяти 324
◊ функция активационная:
- ReLU 325
- tanh 325

C

Сведение 311–314
◊ на основе максимума 311
◊ на основе среднего значения 314
Свертка 298–302
Сверточные нейронные сети (CNN-сети)
◊ RGB 320
◊ TensorFlow 319–322
◊ активационная функция ReLU 320
◊ гиперпараметрическая настройка 318
◊ заполнение 314
◊ мини-пакетный градиентный спуск 321

Сверточные нейронные сети (CNN-сети) *(прод.)*
◊ набор данных Zalando 318
◊ операция свертки:
▫ Python 305
▫ визуальное объяснение 302, 303
▫ матричный формализм 298–302
▫ примеры 304
▫ распознавание образов 299
▫ сдвиги 301, 303
▫ тензоры 298
▫ формальное определение 301
▫ шахматная доска 306–311
◊ полносвязный слой 320
◊ сведение 311–314
◊ стоимостная функция 321
◊ строительные блоки
▫ сверточные слои 315, 316
▫ сводящие слои 316
▫ стековая (ярусная) укладка слоев 317
◊ ядра и фильтры 297, 298
Смещение 105
Спуск градиентный 63
◊ мини-пакетный 121–123
◊ стохастический 120, 123, 141
◊ пакетный 119, 120, 123
Стандартная агломерация г. Бостон (Standard Metropolitan Statistical Area, SMSA) 72

Т

Темп заучивания 64
Тензор разреженный 199
Тождественное отображение 55–56

Ф

Функция:
◊ активационная ELU 62
◊ лоренцева 335
◊ нейрона:
▫ активационная 50
• ArcTan 62
• ReLU 58–60
• ReLU с утечкой 60
• Softplus 62
• Swish 61
• tanh 58
• сигмоидальная 56–58
• тождественного отображения 55–56
▫ передаточная 50
◊ обнаружения 277, 279
◊ радиально-базисная (RBF) 270, 294
◊ черно-ящичная:
▫ гиперпараметры 255
▫ классы 255
▫ модель нейросетевая 254
▫ образец задачи 257
▫ оптимизация глобальная 253
▫ функция обнаружения 279–285

Ч

Человеческая результативность:
◊ байесова ошибка 208
◊ блог-пост Карпати А. 211–213
◊ набор данных MNIST 213
◊ определение 208–211
◊ технические решения 210
◊ точность 208, 210

Э

Элемент экспоненциальный линейный (ELU) 62
Эпоха 119

Для разработчиков от разработчиков

Прикладное глубокое обучение

Подход к пониманию глубоких нейронных сетей на основе метода кейсов

Затронуты расширенные темы глубокого обучения: оптимизационные алгоритмы, настройка гиперпараметров, отсев и анализ ошибок, стратегии решения типичных задач во время тренировки глубоких нейронных сетей.

Описаны простые активационные функции с единственным нейроном (ReLu, сигмоида и Swish), линейная и логистическая регрессии с использованием библиотеки TensorFlow, выбор стоимостной функции, а также более сложные нейросетевые архитектуры с многочисленными слоями и нейронами и случайной инициализацией весов. Проведен анализ ошибок нейронной сети с примерами решения проблем, возникающих из-за дисперсии, смещения, переподгонки, а также наборов данных, поступающих из разных распределений. Показано, как реализовывать логистическую регрессию средствами Python и NumPy.

Книга содержит примеры использования по каждому техническому решению с целью реализации на практике всей теоретической информации. Вы также найдете советы и рекомендации по написанию оптимизированного кода Python.

На практике вы освоите :

- Передовые технические решения на Python и TensorFlow
- Отладку и оптимизацию расширенных методов (таких как отсев и регуляризация)
- Проведение анализа ошибок (с целью выяснить наличие проблем со смещением, дисперсией, смещением данных и т. д.)
- Настройку проектов машинного обучения, ориентированных на глубокое обучение с использованием сложных наборов данных

Apress®

bhv®

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-4118-3