O'REILLY®

Основы глубокого обучения

СОЗДАНИЕ АЛГОРИТМОВ ДЛЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА СЛЕДУЮЩЕГО ПОКОЛЕНИЯ

Никхиль Будума
при участии Николаса Локашо

Нихиль Будума
при участии Николаса Локашо

Основы глубокого обучения

Создание алгоритмов для искусственного интеллекта следующего поколения

Перевод с английского Александра Коробейникова

Москва
«Манн, Иванов и Фербер»
2020

Nikhil Buduma
with contributions by Nicholas Locascio

Fundamentals of Deep Learning

Designing Next-Generation Machine Intelligence Algorithms

O'REILLY
Beijing • Boston • Farnham • Sebastopol • Tokyo

Эту книгу хорошо дополняют:

Искусственный интеллект
Ник Бостром

Верховный алгоритм
Педро Домингос

Машина, платформа, толпа
Эрик Бриньолфсон

Неизбежно
Кевин Келли

УДК 004.89
ББК 32.813
Б90

Научный редактор Андрей Созыкин

Издано с разрешения O'Reilly Media, Inc.

На русском языке публикуется впервые

Будума, Нихиль

Б90 Основы глубокого обучения. Создание алгоритмов для искусственного интеллекта следующего поколения / Нихиль Будума, Николас Локашо ; пер. с англ. А. Коробейникова ; [науч. ред. А. Созыкин]. — М. : Манн, Иванов и Фербер, 2020. — 304 с.

ISBN 978-5-00146-472-3

Глубокое обучение — это раздел машинного обучения, изучающий глубокие нейронные сети и выстраивающий процесс получения знаний на основе примеров. Такие крупные компании, как Google, Microsoft и Facebook, уделяют большое внимание глубокому обучению и расширяют свои подразделения в этой сфере. Для всех прочих глубокое обучение пока остается сложным, многогранным и малопонятным предметом.

Цель этой книги — заполнить этот пробел. Авторы разбирают основные принципы решения задач в глубоком обучении, исторический контекст современных подходов к нему и способы внедрения его алгоритмов.

Для всех, кто интересуется или занимается глубоким обучением.

УДК 004.89
ББК 32.813

ISBN 978-5-00146-472-3

Оглавление

Предисловие

С оживлением нейросетей в 2000-е годы глубокое обучение стало очень активно развивающейся областью исследований, основой передачи знаний с помощью машин. В этой книге приведены примеры и объяснения, которые помогут понять основные идеи этой сложной отрасли. Такие крупные компании, как Google, Microsoft и Facebook, уделяют внимание глубокому обучению и расширяют свои подразделения в этой области. Для всех прочих глубокое обучение остается сложным, многогранным и малопонятным предметом. В работах по этой теме много неясного жаргона, а разнообразные учебники, выложенные в сети, не дают четкого представления о том, как решаются задачи в данной области. Наша цель — восполнить этот пробел.

Требования и цели

Это книга для аудитории, на базовом уровне владеющей математическим анализом, матричным исчислением и программированием на Python. Понять материал без этих знаний можно, но, скорее всего, очень сложно. Для некоторых разделов пригодится эрудиция в области линейной алгебры. К концу книги, надеемся, читатели уже будут понимать принципы решения задач в глубоком обучении, исторический контекст современных подходов к нему и узнают, как внедрить его алгоритмы при помощи открытой библиотеки TensorFlow.

УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

В книге используются следующие виды выделений.

Курсив

Новые термины, ссылки, названия и расширения файлов.

`Моноширинный шрифт`

Им оформлены программные элементы — названия переменных или свойств, базы данных, переменные среды, операторы и ключевые слова.

`Моноширинный шрифт, полужирный`

Команды или иной текст, вводимые пользователем.

`Моноширинный шрифт, курсив`

То, что нужно заменить пользовательскими значениями или такими, которые определяются контекстом (например, переменные в формулах).

ОБРАЗЦЫ КОДА

Дополнительный материал (образцы кода, примеры и т. д.) вы найдете по адресу https://github.com/darksigma/Fundamentals-of-Deep-Learning-Book.

Эта книга поможет вам в работе. Если приведен пример кода, его можно использовать в программах и документах. Не нужно получать у нас разрешение, если только вы не воспроизводите значительную часть кода. Например, написание программы, где использовано несколько кусков кода из этой книги, согласования не требует. Но необходимо разрешение на продажу или распространение CD-ROM с примерами из книг O'Reilly. Цитирование текста и кода из этой книги при ответах на вопросы разрешения не требует. На включение крупных фрагментов образца кода из этой книги в документацию вашего продукта необходимо разрешение.

Мы не требуем ссылок, но будем благодарны за них. Ссылка обычно включает название, автора и издательство. Например: Fundamentals of Deep Learning by Nikhil Buduma and Nicholas Locascio (O'Reilly). Copyright 2017 Nikhil Buduma and Nicholas Locascio.

Если вам кажется, что использование вами образцов кода выходит за установленные выше рамки, свяжитесь с нами по электронной почте permissions@oreilly.com.

ГЛАВА 1

Нейросеть

Создание умных машин

Мозг — самый невероятный орган. Именно он определяет, как мы воспринимаем всё, что видим, слышим, обоняем, пробуем на вкус и осязаем. Он позволяет хранить воспоминания, испытывать эмоции и даже мечтать. Без мозга мы были бы примитивными организмами, способными лишь на простейшие рефлексы. В целом он делает человека разумным.

Мозг ребенка весит меньше полукилограмма, но как-то решает задачи, пока недоступные даже самым большим и мощным компьютерам. Всего через несколько месяцев после рождения дети способны распознавать лица родителей, отделять объекты от фона и даже различать голоса. За первый год у них развивается интуитивное понимание естественной физики, они учатся видеть, где находятся частично или полностью скрытые от них объекты, и ассоциировать звуки с их значениями. Уже в раннем возрасте они на высоком уровне овладевают грамматикой, а в их словаре появляются тысячи слов[1].

Десятилетиями мы мечтаем о создании разумных машин с таким же мозгом, как у нас: роботов-помощников для уборки в доме; машин, которые управляют собой сами; микроскопов, автоматически выявляющих болезни. Но создание машин с искусственным интеллектом требует решения сложнейших вычислительных задач в истории, которые, однако, наш мозг способен раскусить в доли секунды. Для этого нужно разработать иной способ программирования компьютеров при помощи методов, которые появились в основном в последние десять лет. Это очень активная отрасль в исследованиях искусственного интеллекта, которая получила название *глубокого обучения*.

Ограничения традиционных компьютерных программ

Почему некоторые задачи компьютерам решать тяжело? Стандартные программы доказали свою состоятельность в двух областях: 1) они очень быстро ведут вычисления; 2) они неукоснительно следуют инструкциям. Если

вы финансист и вам нужно провести сложные математические подсчеты, вам повезло. Типовые программы вам в помощь. Но представьте себе, что нам нужно сделать кое-что поинтереснее: например, написать программу для автоматического распознавания почерка. Возьмем за основу рис. 1.1.

Рис. 1.1. *Изображение из массива рукописных данных MNIST*[2]

Хотя каждая цифра на рисунке слегка отличается от предыдущей, мы легко опознаем в первом ряде нули, во втором — единицы и т. д. Теперь напишем компьютерную программу, которая решит ту же задачу. Какие правила нужно задать, чтобы различать цифры?

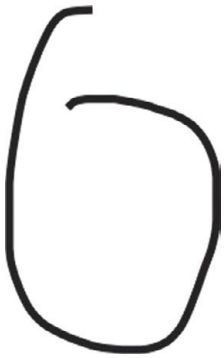


Рис. 1.2. *Ноль, алгоритмически трудноотличимый от шестерки*

Начнем с простого. Например, укажем, что нулю соответствует изображение округлого замкнутого контура. Все примеры с рис. 1.1, кажется,

удовлетворяют этому определению, но таких признаков недостаточно. Что, если у кого-то ноль — не всегда замкнутая фигура? И как отличить такой ноль (см. рис. 1.2) от шестерки?

Можно задать рамки расстояния между началом и концом петли, но не очень понятно какие. И это только начало проблем. Как различить тройки и пятерки? Четверки и девятки? Можно добавлять правила, или *признаки*, после тщательных наблюдений и месяцев проб и ошибок, но понятно одно: процесс будет нелегок.

Многие другие классы задач попадают в ту же категорию: распознавание объектов и речи, автоматический перевод и т. д. Мы не знаем, какие программы писать для них, потому что не понимаем, как с этим справляется наш мозг. А если бы и знали, такая программа была бы невероятно сложной.

Механика машинного обучения

Для решения таких задач нужен совсем иной подход. Многое из того, что мы усваиваем в школе, похоже на стандартные компьютерные программы. Мы учимся перемножать числа, решать уравнения и получать результаты, следуя инструкциям. Но навыки, которые мы получаем в самом юном возрасте и считаем самыми естественными, усваиваются не из формул, а на примерах.

Например, в двухлетнем возрасте родители не учат нас узнавать собаку, измеряя форму ее носа или контуры тела. Мы можем отличать ее от других существ, потому что нам показали много примеров собак и несколько раз исправили наши ошибки. Уже при рождении мозг дал нам модель, описывающую наше мировосприятие. С возрастом благодаря ей мы стали на основе получаемой сенсорной информации строить предположения о том, с чем сталкиваемся. Если предположение подтверждалось родителями, это способствовало укреплению модели. Если же они говорили, что мы ошиблись, мы меняли модель, дополняя ее новой информацией. С опытом она становится все точнее, поскольку включает больше примеров. И так происходит на подсознательном уровне, мы этого даже не понимаем, но можем с выгодой использовать.

Глубокое обучение — отрасль более широкой области исследований искусственного интеллекта: *машинного обучения*, подразумевающего получение знаний из примеров. Мы не задаем компьютеру огромный список правил решения задачи, а предоставляем *модель*, с помощью которой он может сравнивать примеры, и краткий набор инструкций для ее модификации

в случае ошибки. Со временем она должна улучшиться настолько, чтобы решать поставленные задачи очень точно.

Перейдем к более строгому изложению и сформулируем идею математически. Пусть наша модель — функция $h(\mathbf{x}, \theta)$. Входное значение $\mathbf{x}$ — пример в векторной форме. Допустим, если $\mathbf{x}$ — изображение в оттенках серого, компоненты вектора — интенсивность пикселей в каждой позиции, как показано на рис. 1.3.

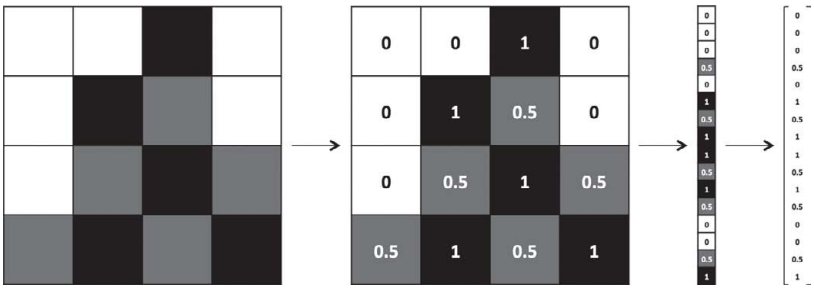


Рис. 1.3. *Векторизация изображения для алгоритма машинного обучения*

Входное значение θ — вектор параметров, используемых в нашей модели. Программа пытается усовершенствовать их значения на основе растущего числа примеров. Подробнее мы рассмотрим этот вопрос в главе 2.

Чтобы интуитивно понимать модели машинного обучения, рассмотрим пример. Допустим, мы решили узнать, как предсказывать результаты экзаменов, если известно количество часов сна и учебы в день перед испытанием. Мы собираем массив данных и при каждом замере $\mathbf{x} = [x_1 \ x_2]^T$ записываем количество часов сна (x_1), учебы (x_2) и отмечаем, выше или ниже они средних по классу. Наша цель — создать модель $h(x, \theta)$ с вектором параметров $\theta = [\theta_0 \ \theta_1 \ \theta_2]^T$, чтобы:

$$h(x,\theta) = \begin{cases} -1, \text{ если } x^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 < 0 \\ 1, \text{ если } x^T \cdot \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} + \theta_0 \geq 0 \end{cases}.$$

По нашему предположению, проект модели $h(x, \theta)$ будет таким, как описано выше (с геометрической точки зрения он описывает линейный классификатор, делящий плоскость координат надвое). Теперь мы хотим узнать вектор параметров θ, чтобы научить модель делать верные предсказания

(`-1`, если результаты ниже среднего уровня, и `1` — если выше) на основании примерного входного значения **x**. Такая модель называется линейным *персептроном* и используется с 1950-х[3]. Предположим, наши данные соответствуют тому, что показано на рис. 1.4.

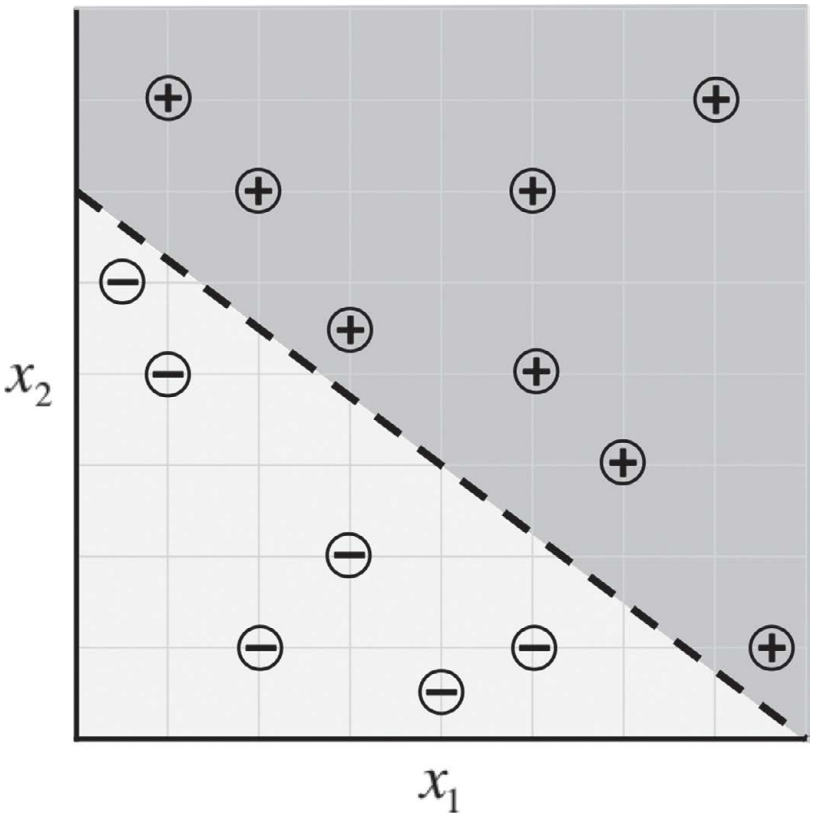


Рис. 1.4. *Образец данных для алгоритма предсказания экзаменов и потенциального классификатора*

Оказывается, при $\theta = [-24\ 3\ 4]^T$ модель машинного обучения способна сделать верное предсказание для каждого замера:

$$h(x,\theta)=\begin{cases}-1, \text{если } 3x_1+4x_2-24<0\\ 1, \text{если } 3x_1+4x_2-24\geq 0\end{cases}.$$

Оптимальный вектор параметров θ устанавливает классификатор так, чтобы можно было сделать как можно больше корректных предсказаний.

Обычно есть множество (иногда даже бесконечное) возможных оптимальных вариантов θ. К счастью, в большинстве случаев альтернативы настолько близки, что разницей между ними можно пренебречь. Если это не так, можно собрать больше данных, чтобы сузить выбор θ.

Звучит разумно, но есть много очень серьезных вопросов. Во-первых, откуда берется оптимальное значение вектора параметров θ? Решение этой задачи требует применения метода *оптимизации*. Оптимизаторы стремятся повысить производительность модели машинного обучения, последовательно изменяя ее параметры, пока погрешность не станет минимальной.

Мы подробнее расскажем об обучении векторов параметров в главе 2, описывая процесс *градиентного спуска*[4]. Позже мы постараемся найти способы еще больше увеличить эффективность этого процесса.

Во-вторых, очевидно, что эта модель (линейного персептрона) имеет ограниченный потенциал обучения. Например, случаи распределения данных на рис. 1.5 нельзя удобно описать с помощью линейного персептрона.

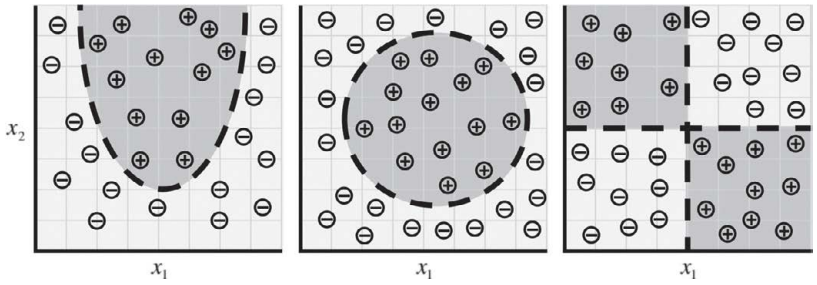


Рис. 1.5. *По мере того как данные принимают более комплексные формы, нам становятся необходимы более сложные модели для их описания*

Но эти ситуации — верхушка айсберга. Когда мы переходим к более комплексным проблемам — распознаванию объектов или анализу текста, — данные приобретают очень много измерений, а отношения, которые мы хотим описать, становятся крайне нелинейными. Чтобы отразить это, в последнее время специалисты по машинному обучению стали строить модели, напоминающие структуры нашего мозга. Именно в этой области, обычно называемой глубоким обучением, ученые добились впечатляющих успехов в решении проблем компьютерного зрения и обработки естественного языка. Их алгоритмы не только значительно превосходят все остальные, но даже соперничают по точности с достижениями человека, а то и превосходят их.

Нейрон

Нейрон — основная единица мозга. Небольшой его фрагмент, размером примерно с рисовое зернышко, содержит более 10 тысяч нейронов, каждый из которых в среднем формирует около 6000 связей с другими такими клетками[5]. Именно эта громоздкая биологическая сеть позволяет нам воспринимать мир вокруг. В этом разделе наша задача — воспользоваться естественной структурой для создания моделей машинного обучения, которые решают задачи аналогично. По сути, нейрон оптимизирован для получения информации от «коллег», ее уникальной обработки и пересылки результатов в другие клетки. Процесс отражен на рис. 1.6. Нейрон получает входную информацию по *дендритам* — структурам, напоминающим антенны. Каждая из входящих связей динамически усиливается или ослабляется на основании частоты использования (так мы учимся новому!), и сила соединений определяет вклад входящего элемента информации в то, что нейрон выдаст на выходе. Входные данные оцениваются на основе этой силы и объединяются в *клеточном теле*. Результат трансформируется в новый сигнал, который распространяется по клеточному *аксону* к другим нейронам.

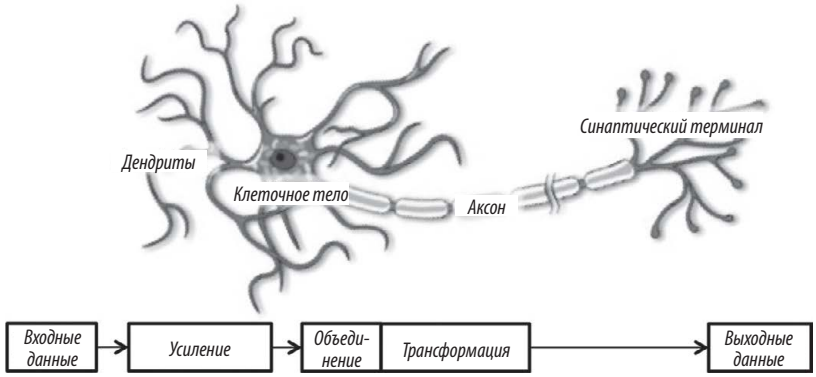


Рис. 1.6. *Функциональное описание биологической структуры нейрона*

Мы можем преобразовать функциональное понимание работы нейронов в нашем мозге в искусственную модель на компьютере. Последняя описана на рис. 1.7, где применен подход, впервые введенный в 1943 году Уорреном Маккаллоу и Уолтером Питтсом[6]. Как и биологические нейроны, искусственный получает некоторый объем входных данных — x_1, x_2, … , x_n, каждый элемент которых умножается на определенное значение веса — w_1, w_2, … , w_n. Эти значения, как и раньше, суммируются, давая *логит*

нейрона: $z = \sum_{i=0}^{n} w_i x_i$. Часто он включает также смещение (константа, здесь не показана). Логит проходит через функцию активации `f`, образуя выходное значение `y = f(z)`. Это значение может быть передано в другие нейроны.

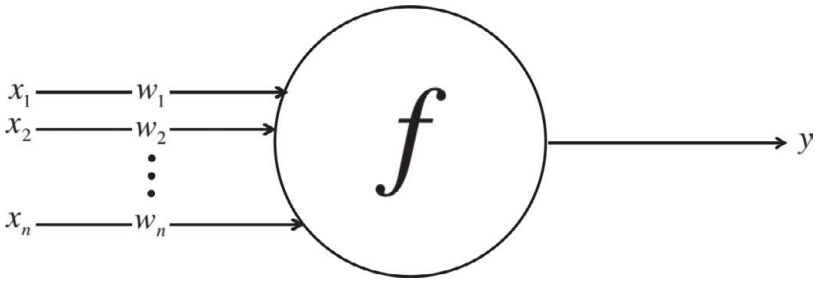


Рис. 1.7. *Схема работы нейрона в искусственной нейросети*

Математическое обсуждение искусственного нейрона мы закончим, выразив его функции в векторной форме. Представим входные данные нейрона как вектор $\mathbf{x} = [x_1\ x_2\ \ldots\ x_n]$, а веса нейрона как $\mathbf{w} = [w_1\ w_2\ \ldots\ w_n]$. Теперь выходные данные нейрона можно выразить как $y = f(\mathbf{x} \cdot \mathbf{w} + b)$, где b — смещение. Мы способны вычислить выходные данные из скалярного произведения входного вектора на вектор весов, добавив смещение и получив логит, а затем применив функцию активации. Это кажется тривиальным, но представление нейронов в виде ряда векторных операций очень важно: только в таком формате их используют в программировании.

Выражение линейных персептронов в виде нейронов

Выше мы говорили об использовании моделей машинного обучения для определения зависимости между результатом на экзаменах и временем, потраченным на обучение и сон. Для решения задачи мы создали линейный классификатор-персептрон, который делит плоскость декартовых координат надвое:

$$h(x,\theta) = \begin{cases} -1, \text{ если } 3x_1 + 4x_2 - 24 < 0 \\ 1, \text{ если } 3x_1 + 4x_2 - 24 \geq 0 \end{cases}.$$

Как показано на рис. 1.4, это оптимальный вариант для θ: он позволяет корректно классифицировать все примеры в нашем наборе данных. Здесь мы видим, что наша модель `h` работает по образцу нейрона. Посмотрите

на нейрон на рис. 1.8. У него два входных значения, смещение, и он использует функцию:

$$f(z)=\begin{cases}-1, \text{ если } z<0\\ 1, \text{ если } z\geq 0\end{cases}.$$

Легко показать, что линейный персептрон и нейронная модель полностью эквивалентны. И просто продемонстрировать, что одиночные нейроны более выразительны, чем линейные персептроны. Каждый из них может быть выражен в виде одиночного нейрона, но последние могут также отражать модели, которые нельзя выразить с помощью линейного персептрона.

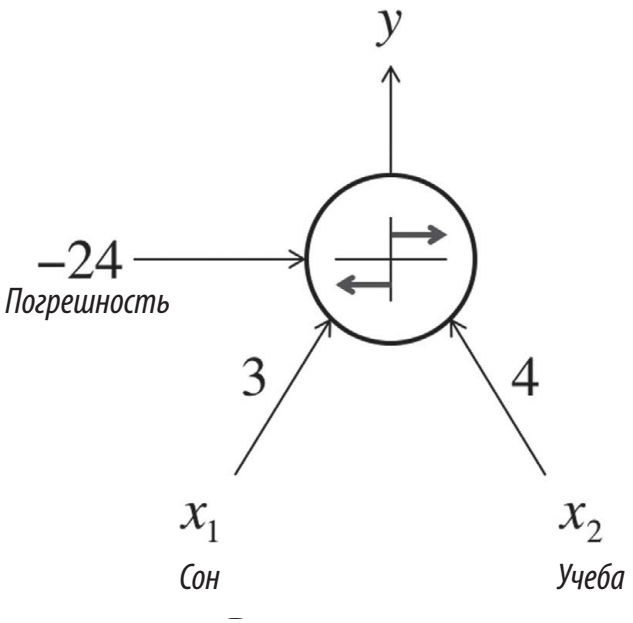


Рис. 1.8. *Выражение результатов экзамена в виде нейрона*

Нейросети с прямым распространением сигнала

Одиночные нейроны мощнее линейных персептронов, но не способны решить сложные проблемы обучения. Поэтому наш мозг состоит из множества нейронов. Например, при помощи одного из них невозможно различить написанные от руки цифры. И чтобы решать более сложные задачи, нам нужны модели машинного обучения.

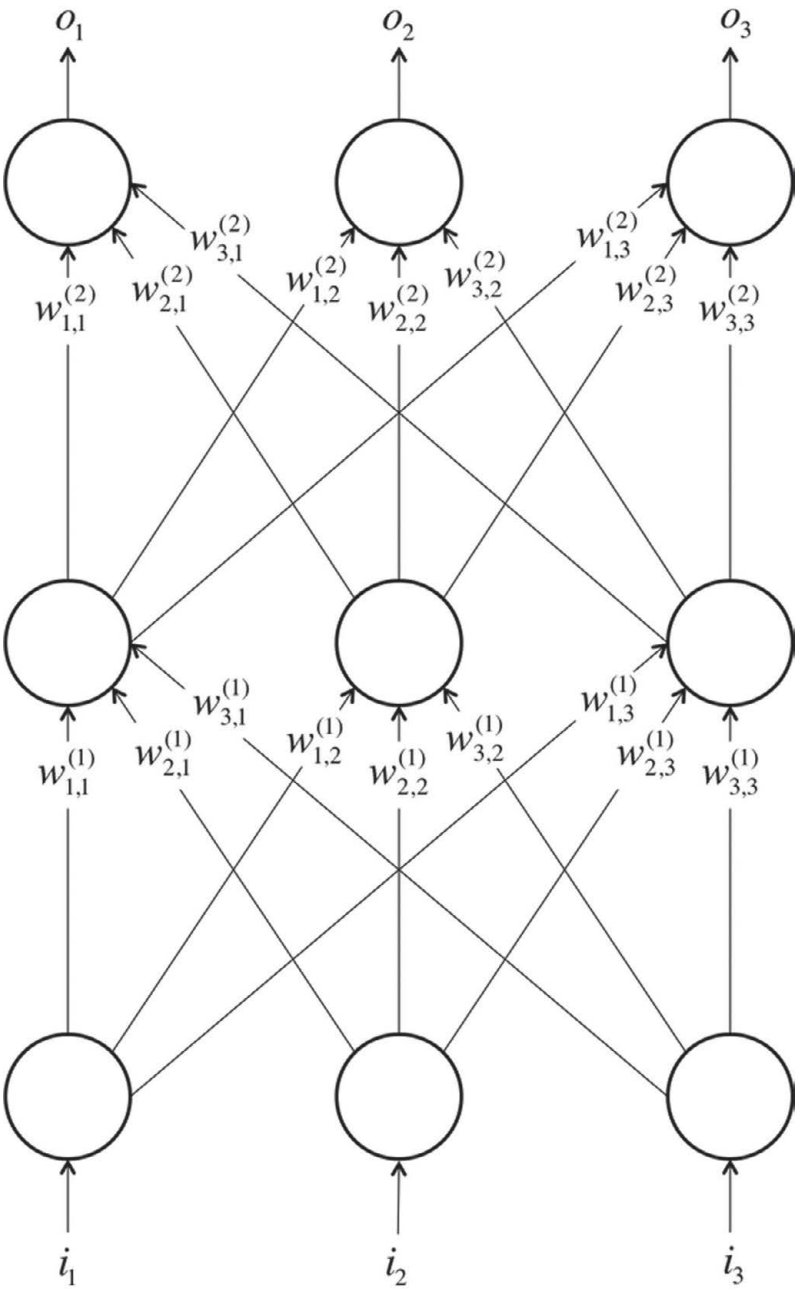


Рис. 1.9. *Простой пример нейросети с прямым распространением сигнала с тремя слоями (входной, скрытый, выходной) и тремя нейронами на каждый слой*

Нейроны в человеческом мозге расположены слоями. Его кора, по большей части отвечающая за интеллект, состоит из шести слоев. Информация перетекает по ним, пока сенсорные данные не преобразуются в концептуальное понимание[7]. Например, самый нижний слой визуальной зоны коры получает необработанные визуальные данные от глаз. Эта информация преобразуется в каждом следующем слое и передается далее, пока на шестом слое мы не заключаем, что видим кошку, банку газировки или самолет. На рис. 1.9 показан упрощенный вариант этих слоев.

На основе этих идей мы можем создать *искусственную нейросеть*. Она возникает, когда мы начинаем соединять нейроны друг с другом, со входными данными и выходными узлами, которые соответствуют ответам сети на изучаемую задачу. На рис. 1.9 показан простейший пример искусственной нейросети, схожей по архитектуре с той, что была описана в 1943 году в работе Маккаллоу и Питтса. В нижний слой поступают входные данные. Верхний (выходные узлы) вычисляет ответ. Средний слой (слои) нейронов именуется скрытым, и здесь $w_{i,j}^{(k)}$ — вес соединения *i-го* нейрона в *k-м* слое с *j-м* нейроном в *(k + 1)-м слое*. Эти веса образуют вектор параметров θ, и, как и ранее, наша способность решать задачи при помощи нейросетей зависит от нахождения оптимальных значений для θ.

В этом примере соединения устанавливаются только от нижних слоев к верхним. Отсутствуют связи между нейронами одного уровня, нет таких, которые передают данные от высшего слоя к низшему. Подобные нейросети называются *сетями с прямым распространением сигнала*, и мы начнем с них, потому что их анализировать проще всего. Такой разбор (процесс выбора оптимальных значений для весов) мы предложим в главе 2. Более сложные варианты связей будут рассмотрены в дальнейших главах.

Ниже мы рассмотрим основные типы слоев, используемые в нейросетях с прямым распространением сигнала. Но для начала несколько важных замечаний.

1. Как мы уже говорили, слои нейронов между первым (входным) и последним (выходным) слоями называются скрытыми. Здесь в основном и происходят волшебные процессы, нейросеть пытается решить поставленные задачи. Раньше (как при распознавании рукописных цифр) мы тратили много времени на определение полезных свойств; эти скрытые слои автоматизируют процесс. Рассмотрение процессов в них может многое сказать о свойствах, которые сеть научилась автоматически извлекать из данных.

2. В этом примере у каждого слоя один набор нейронов, но это не необходимое и не рекомендуемое условие. Чаще в скрытых слоях нейронов меньше, чем во входном: так сеть обучается сжатому представлению информации.

Например, когда глаза получают «сырые» пиксельные значения, мозг обрабатывает их в рамках границ и контуров. Скрытые слои биологических нейронов мозга заставляют нас искать более качественное представление всего, что мы воспринимаем.

3. Необязательно, чтобы выход каждого нейрона был связан с входами всех нейронов следующего уровня. Выбор связей здесь — искусство, которое приходит с опытом. Этот вопрос мы обсудим детально при изучении примеров нейросетей.

4. Входные и выходные данные — *векторные* представления. Например, можно изобразить нейросеть, в которой входные данные и конкретные пиксельные значения картинки в режиме RGB представлены в виде вектора (см. рис. 1.3). Последний слой может иметь два нейрона, которые соотносятся с ответом на задачу: [1, 0], если на картинке собака; [0, 1], если кошка; [1, 1], если есть оба животных; [0, 0], если нет ни одного из них.

Заметим, что, как и нейрон, можно математически выразить нейросеть как серию операций с векторами и матрицами. Пусть входные значение *i-го* слоя сети — вектор $\mathbf{x} = [x_1 \; x_2 \; \ldots \; x_n]$. Нам надо найти вектор $\mathbf{y} = [y_1 \; y_2 \; \ldots \; y_m]$, образованный распространением входных данных по нейронам. Мы можем выразить это как простое умножение матрицы, создав матрицу весов размера $n \times m$ и вектор смещения размера m. Каждый столбец будет соответствовать нейрону, причем j*-й* элемент сопоставлен весу соединения с j*-м* входящим элементом. Иными словами, $\mathbf{y} = f(\mathbf{W}^T\mathbf{x} + \mathbf{b})$, где функция активации применяется к вектору поэлементно. Эта новая формулировка очень пригодится, когда мы начнем реализовывать эти сети в программах.

Линейные нейроны и их ограничения

Большинство типов нейронов определяются функцией активации f, примененной к логиту `logit` z. Сначала рассмотрим слои нейронов, которые используют линейную функцию $f(z) = az + b$. Например, нейрон, который пытается подсчитать стоимость блюда в кафе быстрого обслуживания, будет линейным, $a = 1$ и $b = 0$. Используя $f(z) = z$ и веса, эквивалентные стоимости каждого блюда, программа присвоит линейному нейрону на рис. 1.10 определенную тройку из бургеров, картошки и газировки, и он выдаст цену их сочетания.

Вычисления с линейными нейронами просты, но имеют серьезные ограничения. Несложно доказать, что любая нейросеть с прямым распространением сигнала, состоящая только из таких нейронов, может быть представлена

как сеть без скрытых слоев. Это проблема: как мы уже говорили, именно скрытые слои позволяют узнавать важные свойства входных данных. Чтобы научиться понимать сложные отношения, нужно использовать нейроны с определенного рода нелинейностью.

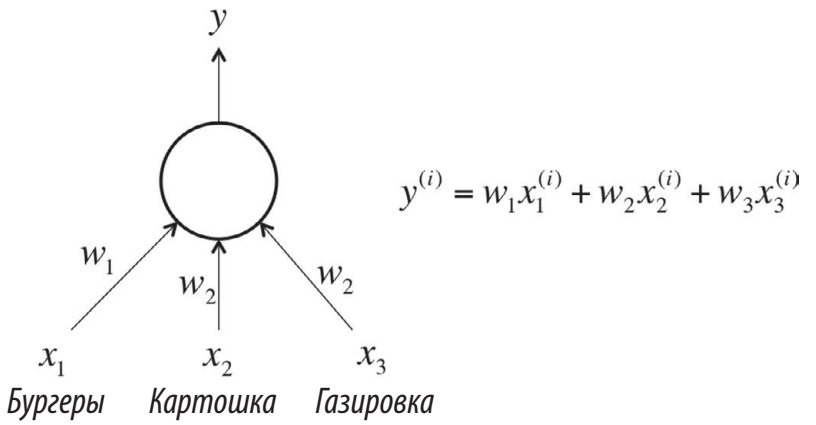


Рис. 1.10. *Пример линейного нейрона*

Нейроны с сигмоидой, гиперболическим тангенсом и усеченные линейные

На практике для вычислений применяются три типа нелинейных нейронов. Первый называется сигмоидным и использует функцию:

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

Интуитивно это означает, что, если логит очень мал, выходные данные логистического нейрона близки к 0. Если логит очень велик — то к 1. Между этими двумя экстремумами нейрон принимает форму буквы S, как на рис. 1.11.

Нейроны гиперболического тангенса (tanh-нейроны) используют похожую `S`-образную нелинейность, но исходящие значения варьируют не от 0 до 1, а от `-1` до `1`. Формула для них предсказуемая: `f(z) = tanh(z)`. Отношения между входным значением `y` и логитом `z` показаны на рис. 1.12. Когда используются `S`-образные нелинейности, часто предпочитают `tanh`-нейроны, а не сигмоидные, поскольку у `tanh`-нейронов центр находится в 0.

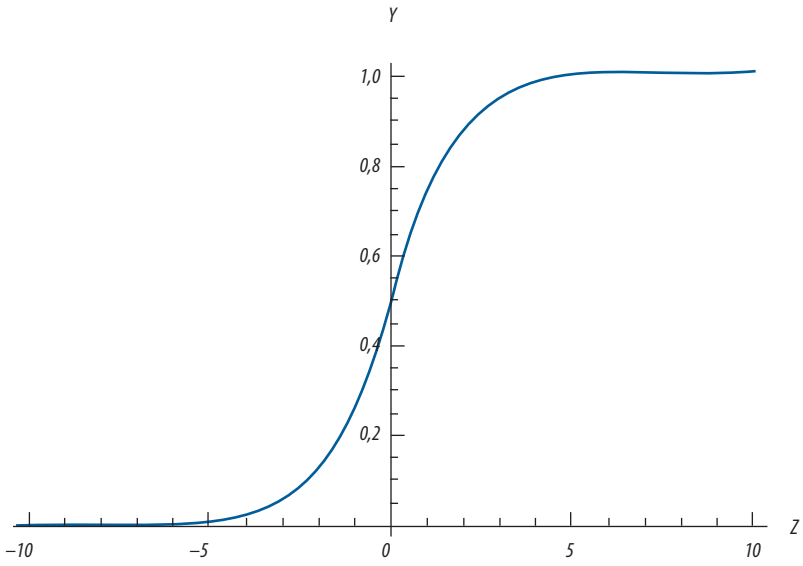


Рис. 1.11. *Выходные данные сигмоидного нейрона с переменной z*

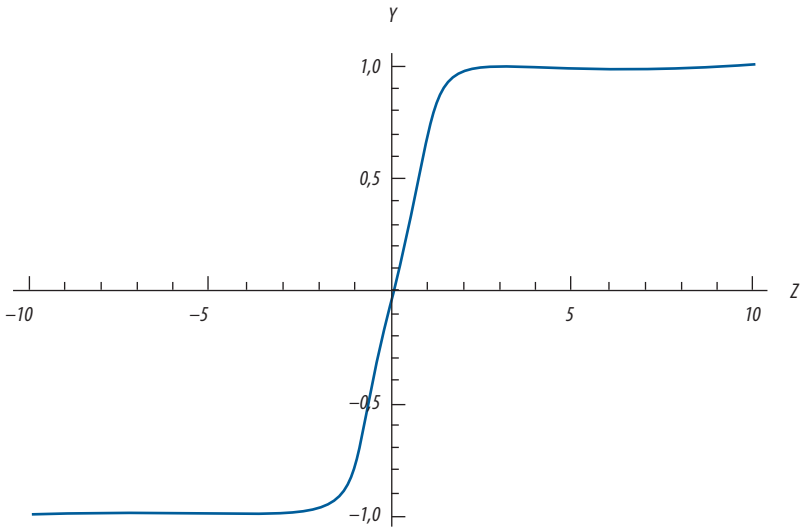


Рис. 1.12. *Выходные данные tanh-нейрона с переменной z*

Еще один тип нелинейности используется *нейроном с усеченным линейным преобразованием* (`ReLU`). Здесь задействована функция `f(z) = max(0, z)`, и ее график имеет форму хоккейной клюшки (рис. 1.13).

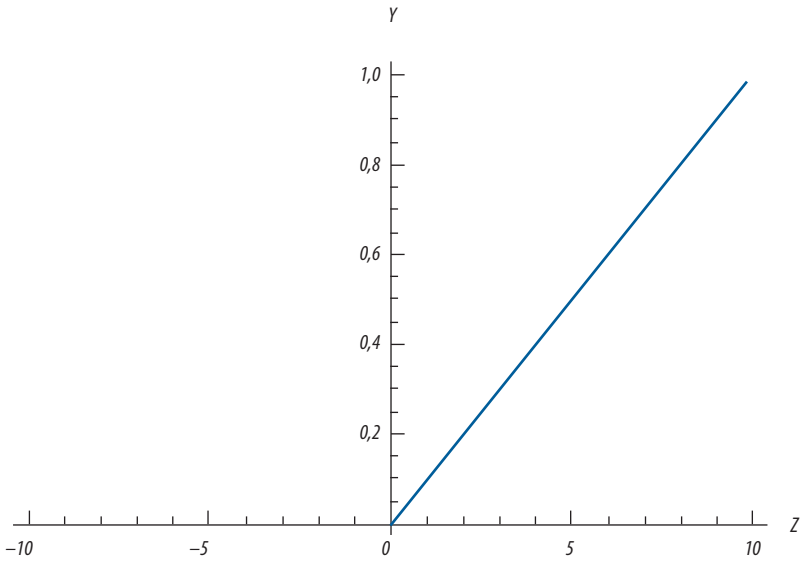


Рис. 1.13. *Выходные данные ReLU-нейрона с переменной z*

`ReLU` в последнее время часто выбирается для выполнения многих задач (особенно в системах компьютерного зрения) по ряду причин, несмотря на свои недостатки[8]. Этот вопрос мы рассмотрим в главе 5 вместе со стратегиями борьбы с потенциальными проблемами.

Выходные слои с функцией мягкого максимума

Часто нужно, чтобы выходной вектор был распределением вероятностей по набору взаимоисключающих значений. Допустим, нам нужно создать нейросеть для распознавания рукописных цифр из набора данных `MNIST`. Каждое значение (от 0 до 9) исключает остальные, но маловероятно, чтобы нам удалось распознать цифры со стопроцентной точностью. Распределение вероятностей поможет понять, насколько мы уверены в своих выводах. Желаемый выходной вектор приобретает такую форму, где $\sum_{i=0}^{9} p_i = 1$:

$$[p_0 \ p_1 \ p_2 \ p_3 \ \ldots \ p_9].$$

Для этого используется особый выходной слой, именуемый *слоем с мягким максимумом (softmax)*. В отличие от других типов, выходные данные нейрона в слое с мягким максимумом зависят от выходных данных всех остальных нейронов в нем. Нам нужно, чтобы сумма всех выходных значений

равнялась 1. Приняв z_i как логит i-го нейрона с мягким максимумом, мы можем достичь следующей нормализации, задав выходные значения:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_i}}.$$

При сильном предсказании одно из значений вектора будет близко к 1, остальные — к 0. При слабом останется несколько возможных значений, каждое из которых характеризуется своим уровнем вероятности.

Резюме

В этой главе мы дали базовые представления о машинном обучении и нейросетях. Мы рассказали о структуре нейрона, работе нейросетей с прямым распространением сигнала и важности нелинейности в решении сложных задач обучения. В следующей главе мы начнем создавать математический фундамент для обучения нейросети решению задач. Например, мы поговорим о нахождении оптимальных векторов параметров, лучших методов обучения нейросетей и основных проблемах. В последующих главах мы будем применять эти основополагающие идеи к более специализированным вариантам архитектуры нейросетей.

ГЛАВА 2

Обучение нейросетей с прямым распространением сигнала

Проблема фастфуда

Мы начинаем понимать, как решать некоторые интересные задачи с помощью глубокого обучения, но остается важный вопрос: как определить, какими должны быть векторы параметров (веса всех соединений нейросети)? Ответ прост: в ходе процесса, часто именуемого *обучением* (рис. 2.1). Мы демонстрируем нейросети множество обучающих примеров и последовательно модифицируем веса, чтобы минимизировать ошибки, которые уже были совершены. Продемонстрировав достаточное число примеров, мы ожидаем, что нейросеть будет эффективно решать поставленную задачу.

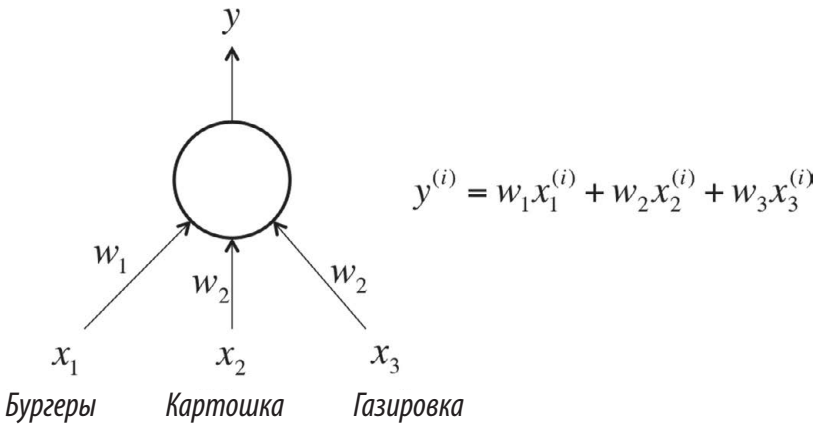


Рис. 2.1. *Нейрон, который мы хотим обучить решать проблему фастфуда*

Вернемся к примеру, который упоминали в предыдущей главе при обсуждении линейного нейрона. Итак: каждый день мы покупаем в ресторане быстрого обслуживания обед — бургеры, картошку и газировку, причем по несколько порций каждого наименования. Мы хотим предсказывать, сколько будет стоить обед, но ценников нет. Кассир сообщает только общую цену.

Мы хотим обучить один линейный нейрон решать эту задачу. Как?

Один из вариантов — разумно подбирать примеры для обучения. Для одного обеда купим один бургер, для второго — одну порцию картошки, для третьего — один стакан газировки. В целом разумный подбор примеров — хорошая идея. Многие исследования показывают, что, создав хорошую подборку данных для обучения, вы сможете заметно повысить эффективность нейросети. Но проблема использования только этого подхода в том, что в реальных ситуациях он редко приближает нас к решению. Например, при распознавании изображений аналога ему нет и решения мы не найдем.

Нам нужно найти вариант, который поможет решать задачу в общем случае. Допустим, у нас очень большой набор обучающих примеров. Это позволит нам вычислить, какие выходные значения выдаст нейросеть на i-м примере, при помощи простой формулы. Мы хотим обучить нейрон и подбираем оптимальные веса, чтобы свести к минимуму ошибки при распознавании примеров. Можно сказать, мы хотим свести к минимуму квадратичную ошибку во всех примерах, которые встретим. Формально, если мы знаем, что $t^{(i)}$ — верный ответ на i-й пример, а $y^{(i)}$ — значение, вычисленное нейросетью, мы хотим свести к минимуму значение функции потерь E:

$$E = \frac{1}{2}\sum_i \left(t^{(i)} - y^{(i)}\right)^2.$$

Квадратичная ошибка равна 0, когда модель дает корректные предсказания для каждого обучающего примера. Более того, чем ближе E к 0, тем лучше модель. Наша цель — выбрать такой вектор параметров θ (значения всех весов в этой модели), чтобы E было как можно ближе к 0.

Вы, возможно, недоумеваете: зачем утруждать себя функцией потерь, если проблему легко решить с помощью системы уравнений. В конце концов, у нас есть наборы неизвестных (весов) и уравнений (одно для каждого примера). Это автоматически даст нам ошибку, равную 0, если обучающие примеры подобраны удачно.

Хорошее замечание, но, к сожалению, актуальное не для всех случаев. Мы применяем здесь линейный нейрон, но на практике они используются редко,

ведь их способности к обучению ограничены. А когда мы начинаем использовать нелинейные нейроны — сигмоиду, tanh или усеченный линейный, о которых мы говорили в конце предыдущей главы, — мы не можем задать систему уравнений! Так что для обучения явно нужна стратегия получше*.

Градиентный спуск

Визуализируем для упрощенного случая то, как свести к минимуму квадратичную ошибку по всем обучающим примерам. Допустим, у линейного нейрона есть только два входа (и соответственно только два веса — w_1 и w_2). Мы можем представить себе трехмерное пространство, в котором горизонтальные измерения соответствуют w_1 и w_2, а вертикальное — значению функции потерь E. В нем точки на горизонтальной поверхности сопоставлены разным значениям весов, а высота в них — допущенной ошибке. Если рассмотреть все ошибки для всех возможных весов, мы получим в этом трехмерном пространстве фигуру, напоминающую миску (рис. 2.2).

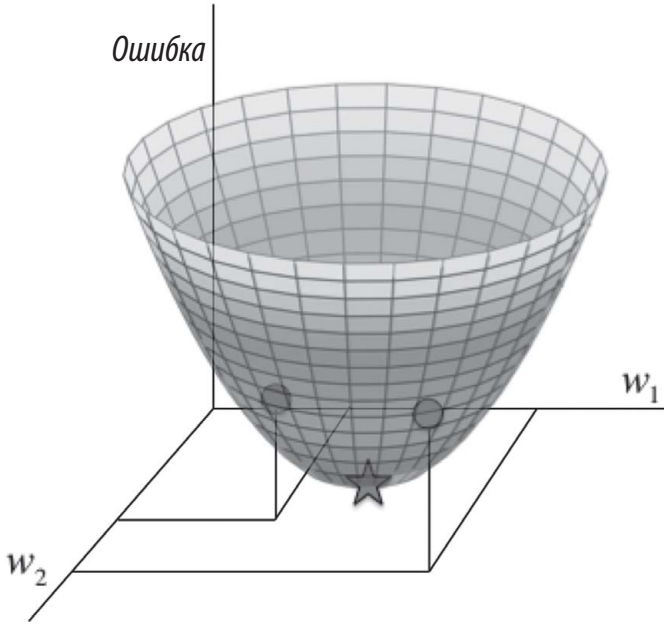


Рис. 2.2. *Квадратичная поверхность ошибки для линейного нейрона*

* Мы можем рассчитать значения неизвестных весов, решив систему линейных уравнений, и получим точное решение. Но такой подход возможен только для линейного нейрона. Для нелинейных составить систему уравнений и получить точное решение невозможно, поэтому необходимо обучение. *Прим. науч. ред.*

Эту поверхность удобно визуализировать как набор эллиптических контуров, где минимальная ошибка расположена в центре эллипсов. Тогда мы будем работать с двумерным пространством, где измерения соответствуют весам. Контуры сопоставлены значениям w_1 и w_2, которые дают одно и то же E. Чем ближе они друг к другу, тем круче уклон. Направление самого крутого уклона всегда перпендикулярно контурам. Его можно выразить в виде вектора, называемого *градиентом*.

Пора разработать высокоуровневую стратегию нахождения значений весов, которые сведут к минимуму функцию потерь. Допустим, мы случайным образом инициализируем веса сети, оказавшись где-то на горизонтальной поверхности. Оценив градиент в текущей позиции, мы можем найти направление самого крутого спуска и сделать шаг в нем. Теперь мы на новой позиции, которая ближе к минимуму, чем предыдущая. Мы проводим переоценку направления самого крутого спуска, взяв градиент, и делаем шаг в новом направлении. Как показано на рис. 2.3, следование этой стратегии со временем приведет нас к точке минимальной ошибки. Этот алгоритм известен как градиентный спуск, и мы будем использовать его для решения проблемы обучения отдельных нейронов и целых сетей[9].

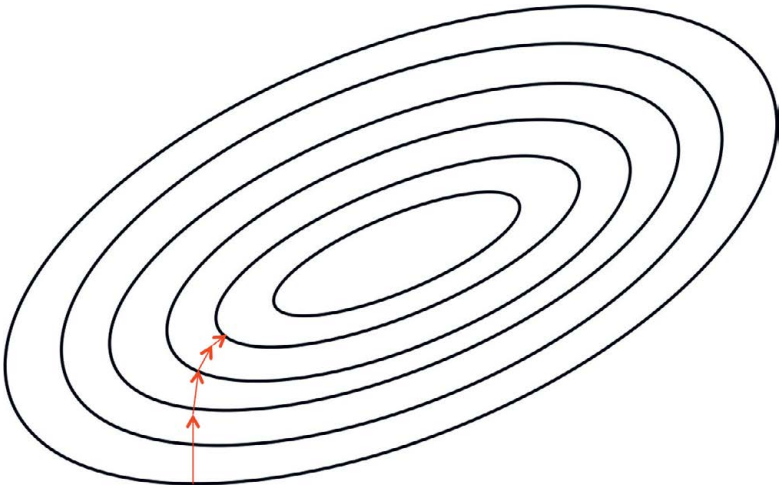


Рис. 2.3. *Визуализация поверхности ошибок как набора контуров*

Дельта-правило и темп обучения

Прежде чем вывести точный алгоритм обучения фастфудного нейрона, поговорим о *гиперпараметрах*. Помимо весов, определенных в нашей

нейросети, обучающим алгоритмам нужен ряд дополнительных параметров. Один из этих гиперпараметров — *темп обучения*.

На каждом шаге движения перпендикулярно контуру нам нужно решать, как далеко мы хотим зайти, прежде чем заново вычислять направление. Это расстояние зависит от крутизны поверхности. Почему? Чем ближе мы к минимуму, тем короче должны быть шаги. Мы понимаем, что близки к минимуму, поскольку поверхность намного более плоская и крутизну мы используем как индикатор степени близости к этому минимуму. Но если поверхность ошибки рыхлая, процесс может занять много времени. Поэтому часто стоит умножить градиент на масштабирующий коэффициент — темп обучения. Его выбор — сложная задача (рис. 2.4).

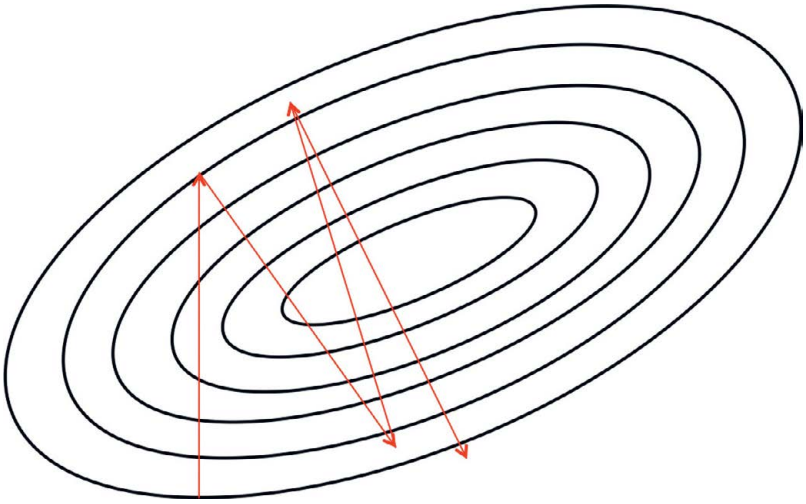


Рис. 2.4. *Если темп обучения слишком велик, возникают проблемы со сходимостью*

Как мы уже говорили, если он будет слишком мал, возможно, процесс займет слишком много времени. Но если темп будет слишком высоким, то кончится это, скорее всего, тем, что мы отклонимся от минимума. В главе 4 мы поговорим о методах оптимизации, в которых используются адаптивные темпы обучения для автоматизации выбора.

Теперь мы готовы вывести *дельта-правило* для обучения линейного нейрона. Чтобы вычислить, как изменять каждый вес, мы оцениваем градиент: по сути, частную производную функции потерь по каждому из весов. Иными словами, нам нужен такой результат:

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k} =$$
$$= -\epsilon \frac{\partial}{\partial w_k}\left(\frac{1}{2}\sum_i \left(t^{(i)} - y^{(i)}\right)^2\right) =$$
$$= \sum_i \epsilon \left(t^{(i)} - y^{(i)}\right)\frac{\partial y_i}{\partial w_k} =$$
$$= \sum_i \epsilon\, x_k^{(i)} \left(t^{(i)} - y^{[i]}\right)$$

Применяя этот метод изменения весов при каждой итерации, мы получаем возможность использовать градиентный спуск.

Градиентный спуск с сигмоидными нейронами

В этом и следующем разделах мы будем говорить об обучении нейронов и нейросетей, использующих нелинейности. В качестве образца возьмем сигмоидный нейрон, а расчеты для других нелинейных нейронов оставим читателям как упражнение. Для простоты предположим, что нейроны не используют смещение, хотя наш анализ вполне можно распространить и на такой случай. Допустим, смещение — вес входа, на который всегда подается 1.

Напомним механизм, с помощью которого логистические нейроны вычисляют выходные значения на основе входных:

$$z = \sum_k w_k x_k,$$
$$y = \frac{1}{1+e^{-z}}.$$

Нейрон определяет взвешенную сумму входящих значений — логит z. Затем он передает этот логит в нелинейную функцию для вычисления выходного значения y. К счастью для нас, эти функции имеют очень красивые производные, что значительно упрощает дело! Для обучения нужно вычислить градиент функции потерь по весам. Возьмем производную логита по входным значениям и весам:

$$\frac{\partial z}{\partial w_k} = x_k,$$
$$\frac{\partial z}{\partial x_k} = w_k.$$

Кроме того, как ни удивительно, производная выходного значения по логиту проста, если выразить ее через выходное значение:

$$\frac{dy}{dx} = \frac{e^{-z}}{\left(1+e^{-z}\right)^2} =$$
$$= \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}} =$$
$$= \frac{1}{1+e^{-z}} \left(1 - \frac{1}{1+e^{-z}}\right) =$$
$$= y\left(1-y\right).$$

Теперь можно использовать правило дифференцирования сложной функции, чтобы вычислить производную выходного значения по каждому из весов:

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz} \frac{\partial z}{\partial w_k} = x_k y\left(1-y\right).$$

Объединяя полученные результаты, мы можем вычислить производную функции потерь по каждому весу:

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial w_k} = -\sum_i x_k^{(i)} y^{(i)} \left(1-y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right).$$

Итоговое правило изменения весов будет выглядеть так:

$$\Delta w_k = \sum_i \in x_k^{(i)} y^{(i)} \left(1-y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right).$$

Как вы видите, новое правило очень похоже на дельта-правило, за исключением дополнительных множителей для учета логистического компонента сигмоидного нейрона.

Алгоритм обратного распространения ошибок

Теперь мы готовы приступить к проблеме обучения многослойных нейросетей, а не только одиночных нейронов. Обратимся к подходу *обратного распространения ошибок*, предложенному Дэвидом Румельхартом, Джеффри Хинтоном и Рональдом Уильямсом в 1986 году[10]. В чем основная идея? Мы не знаем, что делают скрытые нейроны, но можем вычислить, насколько быстро меняется ошибка, если мы вносим корректировки в эти

процессы. На основе этого мы способны определить, как быстро трансформируется ошибка, если изменить вес конкретного соединения. По сути, мы пытаемся найти наибольший уклон! Единственная сложность в том, что приходится работать в пространстве с очень большим числом измерений. Начнем с вычисления производных функции потерь по одному обучающему примеру.

Каждый скрытый нейрон может влиять на многие выходные нейроны. Нам нужно учесть несколько эффектов ошибки, чтобы получить нужную информацию. В качестве стратегии выберем динамическое программирование. Получив производные функций потерь для одного слоя скрытых нейронов, мы применим их для вычисления производных функций потерь на выходе более низкого слоя. Когда мы найдем такие производные на выходе из скрытых нейронов, несложно будет получить производные функций потерь для весов входов в скрытый нейрон. Для упрощения введем дополнительные обозначения (рис. 2.5).

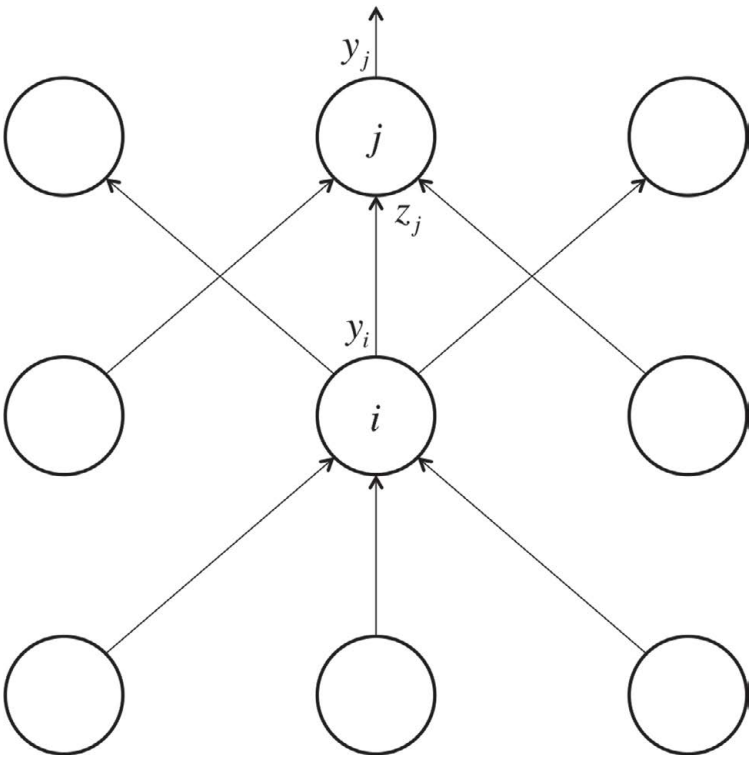


Рис. 2.5. *Справочная диаграмма для вывода алгоритма обратного распространения ошибок*

Нижний индекс будет обозначать слой нейронов; символ `y` — как обычно, выходное значение нейрона, а `z` — логит нейрона. Начнем с базового случая проблемы динамического программирования: вычислим производные функции потерь на выходном слое (output).

$$E = \frac{1}{2}\sum_{j \in output}\left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right).$$

Теперь сделаем индуктивный шаг. Предположим, у нас есть производные функции потерь для слоя `j`. Мы собираемся вычислить производные функции потерь для более низкого слоя `i`. Для этого необходима информация о том, как выходные данные нейрона в слое `i` воздействуют на логиты всех нейронов в слое `j`. Вот как это сделать, используя то, что частная производная логита по входящим значениям более низкого слоя — это вес соединения w_{ij}:

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial z_j}\frac{dz_j}{dy_i} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}.$$

Далее мы видим следующее:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{dy_j}{dz_j} = y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}.$$

Сведя эти факты воедино, мы можем выразить производные функций потерь слоя `i` через производные функций потерь слоя `j`:

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}.$$

Пройдя все стадии динамического программирования и заполнив таблицу всеми частными производными (функций потерь по выходным значениям скрытых нейронов), мы можем определить, как ошибка меняется по отношению к весам. Это даст нам представление о том, как корректировать веса после каждого обучающего примера:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}.$$

Наконец, чтобы завершить алгоритм, как и раньше, мы суммируем частные производные по всем примерам в нашем наборе данных (dataset). Это дает нам следующую формулу изменения:

$$\Delta w_{ij} = -\sum_{k \in dataset} \in y_i^{(k)} y_j^{(k)}\left(1 - y_j^{(k)}\right)\frac{\partial E^{(k)}}{\partial y_j^{(k)}}.$$

На этом описание алгоритма обратного распространения ошибок закончено!

Стохастический и мини-пакетный градиентный спуск

В алгоритмах, описанных в предыдущем разделе, мы использовали так называемый *пакетный градиентный спуск*. Идея в том, что мы при помощи всего набора данных вычисляем поверхность ошибки, а затем следуем градиенту, определяем самый крутой уклон и движемся в этом направлении. Для поверхности простой квадратичной ошибки это неплохой вариант. Но в большинстве случаев поверхность гораздо сложнее. Для примера рассмотрим рис. 2.6.

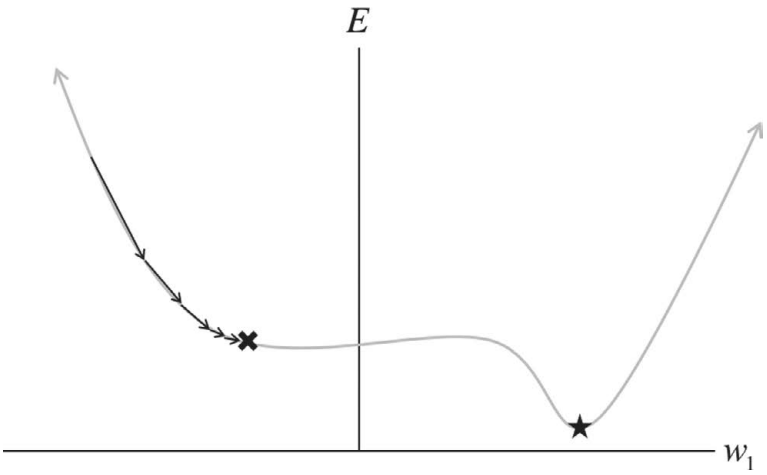


Рис. 2.6. *Пакетный градиентный спуск чувствителен к седловым точкам, что может привести к преждевременному схождению*

У нас только один вес, и мы используем случайную инициализацию и пакетный градиентный спуск для поиска его оптимального значения. Но поверхность ошибки имеет плоскую область (известную в пространствах с большим числом измерений как седловая точка). Если нам не повезет, то при пакетном градиентном спуске мы можем застрять в ней.

Другой возможный подход — стохастический градиентный спуск (СГС). При каждой итерации поверхность ошибки оценивается только для одного примера. Этот подход проиллюстрирован на рис. 2.7, где поверхность ошибки не единая статичная, а динамическая. Спуск по ней существенно улучшает нашу способность выходить из плоских областей.

Основной недостаток стохастического градиентного спуска в том, что рассмотрение ошибки для одного примера может оказаться недостаточным приближением поверхности ошибки.

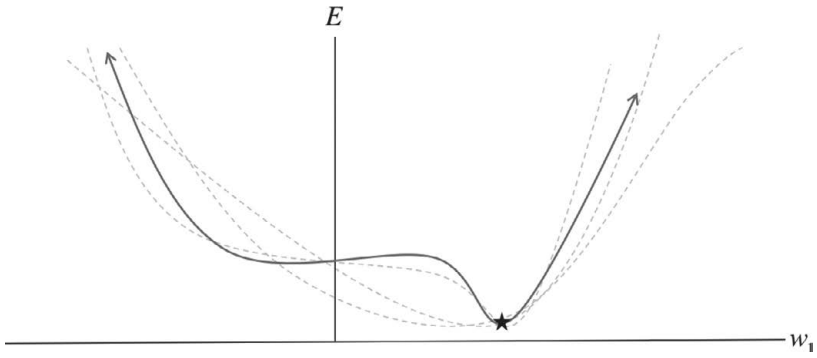


Рис. 2.7. *Стохастическая поверхность ошибки варьирует по отношению к пакетной, что позволяет решить проблему седловых точек*

Это, в свою очередь, приводит к тому, что спуск займет слишком много времени. Один из способов решения проблемы — использование *мини-пакетного градиентного спуска*. При каждой итерации мы вычисляем поверхность ошибки по некой выборке из общего набора данных (а не одному примеру). Это и есть мини-пакет (minibatch), и его размер, как и темп обучения, — гиперпараметр. Мини-пакеты уравновешивают эффективность пакетного градиентного спуска и способность избегать локальных минимумов, которую предоставляет стохастический градиентный спуск. В контексте обратного распространения ошибок изменение весов выглядит так:

$$\Delta w_{ij} = -\sum_{k \in minibatch} \in y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}.$$

Это идентично тому, что мы вывели в предыдущем разделе. Но вместо того чтобы суммировать все примеры в наборе данных, мы обобщаем все примеры из текущего мини-пакета.

Переобучение и наборы данных для тестирования и проверки

Одна из главных проблем искусственных нейросетей — чрезвычайная сложность моделей. Рассмотрим сеть, которая получает данные от изображения из базы данных `MNIST` (28×28 пикселов), передает их в два скрытых слоя по 30 нейронов, а затем в слой с мягким максимумом из 10 нейронов. Общее число ее параметров составляет около 25 тысяч. Это может привести к серьезным проблемам. Чтобы понять почему, рассмотрим еще один упрощенный пример (рис. 2.8).

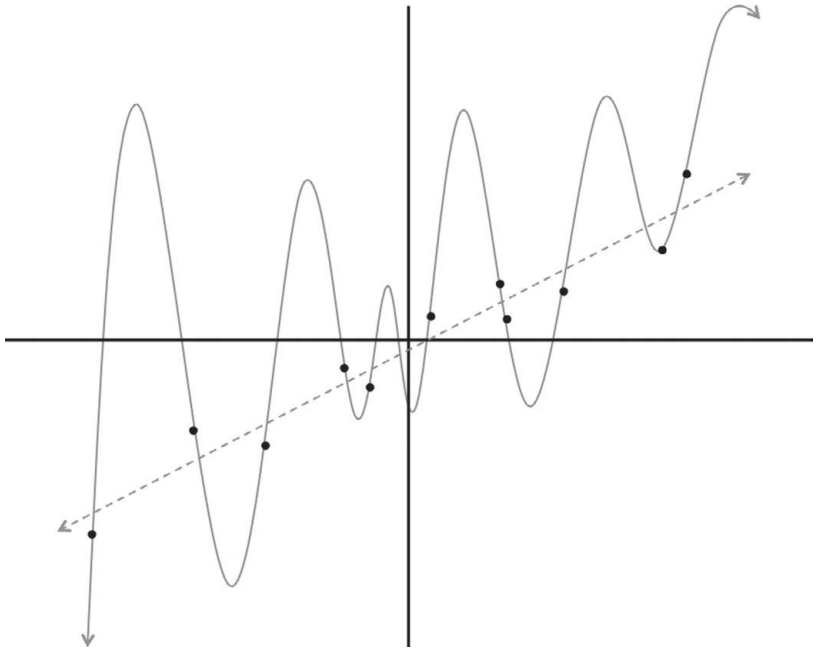


Рис. 2.8. *Две модели, которыми может быть описан наш набор данных: линейная и многочлен 12-й степени*

У нас есть ряд точек на плоской поверхности, задача — найти кривую, которая наилучшим образом опишет этот набор данных (то есть позволит предсказывать координату `y` новой точки, зная ее координату `x`). Используя эти данные, мы обучаем две модели: линейную и многочлен 12-й степени. Какой кривой стоит доверять? Той, которая не попадает почти ни в один обучающий пример? Или сложной, которая проходит через все точки из набора? Кажется, можно доверять линейному варианту, ведь он кажется более естественным. Но на всякий случай добавим данных в наш набор! Результат показан на рис. 2.9.

Вывод очевиден: линейная модель не только субъективно, но и количественно лучше (по показателю квадратичной ошибки). Но это ведет к очень интересному выводу по поводу усвоения информации и оценки моделей машинного обучения. Строя очень сложную модель, легко полностью подогнать ее к обучающему набору данных. Ведь мы даем ей достаточно степеней свободы для искажения, чтобы вписаться в имеющиеся значения. Но когда мы оцениваем такую модель на новых данных, она работает очень плохо, то есть слабо *обобщает*. Это явление называется переобучением. И это одна из главных сложностей, с которыми вынужден иметь дело

инженер по машинному обучению. Нейросети имеют множество слоев с большим числом нейронов, и в области глубокого обучения эта проблема еще значительнее. Количество соединений в моделях составляет миллионы. В результате переобучение — обычное дело (что неудивительно).

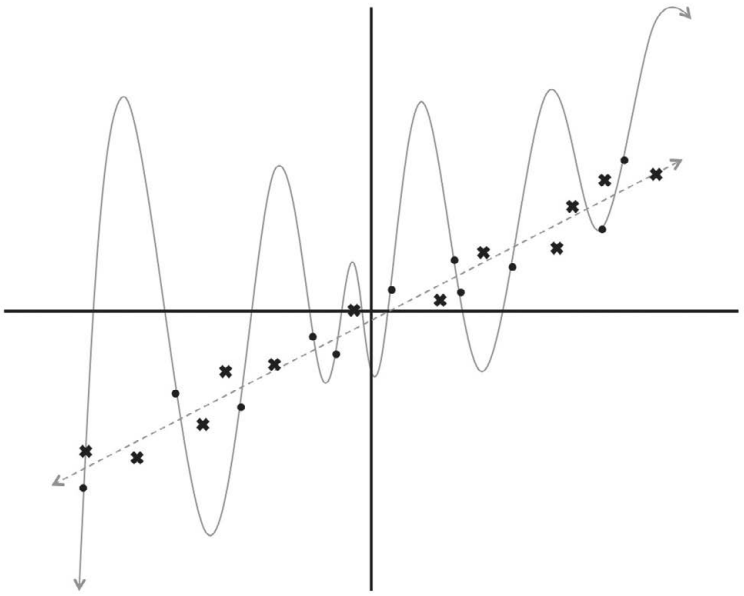


Рис. 2.9. *Оценка модели на основе новых данных показывает, что линейная модель работает гораздо лучше, чем многочлен 12-й степени*

Рассмотрим, как это работает в нейросети. Допустим, у нас есть сеть с двумя входными значениями, выходной слой с двумя нейронами с функцией мягкого максимума и скрытый слой с 3, 6 или 20 нейронами. Мы обучаем эти нейросети при помощи мини-пакетного градиентного спуска (размер мини-пакета 10); результаты, визуализированные в `ConvNetJS`, показаны на рис. 2.10[11].

Уже из этих изображений очевидно, что с увеличением числа соединений нейросети усиливается тенденция к переобучению. Усугубляется она и с углублением нейросетей. Результаты показаны на рис. 2.11, где используются сети с 1, 2 или 4 скрытыми слоями, в каждом из которых по 3 нейрона.

Отсюда следуют три основных вывода. Во-первых, инженер по машинному обучению всегда вынужден искать компромисс между переобучением

и сложностью модели. Если модель недостаточно сложна, она может оказаться недостаточно мощной для извлечения всей полезной информации, необходимой для решения задачи. Но если она слишком сложна (особенно когда у нас есть ограниченный набор данных), высока вероятность, что понадобится переобучение. Глубокое обучение связано с решением очень сложных задач при помощи сложных моделей, поэтому необходимо принимать дополнительные меры против возможного переобучения. О многих из них мы будем говорить в этой главе, а также в следующих.

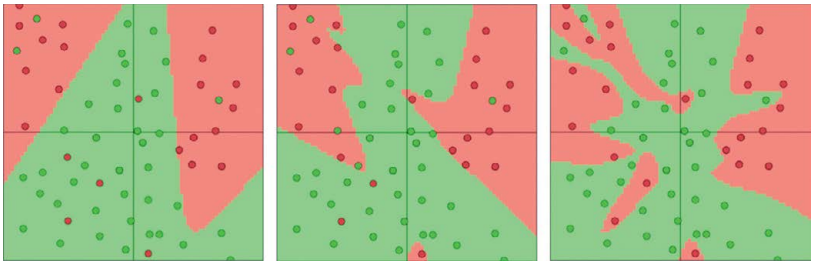


Рис. 2.10. *Визуализация нейросетей с 3, 6 и 20 нейронами (в таком порядке) в скрытом слое*

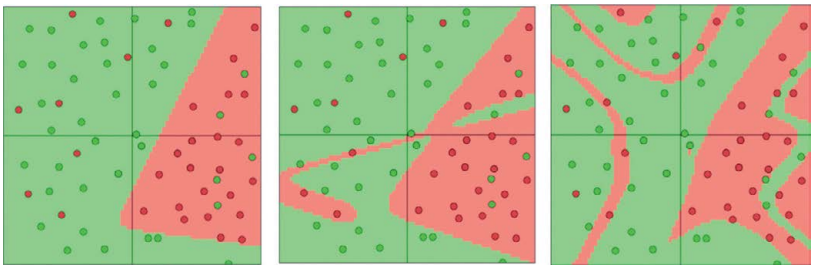


Рис. 2.11. *Визуализация нейросетей с 1, 2 и 4 скрытыми слоями (в таком порядке), по 3 нейрона в каждой*

Во-вторых, неприемлемо оценивать модель на основе данных, с помощью которых мы ее обучали. Так, пример на рис. 2.8 дает ошибочное представление о том, что модель многочлена 12-й степени лучше линейной. В результате мы почти никогда не обучаем модель на полном наборе данных. Как показано на рис. 2.12, мы делим данные на наборы для обучения и тестирования.

Это позволяет дать справедливую оценку модели, непосредственно измерив ее способность к обобщению на новых данных, с которыми она еще

не знакома. В реальном мире большие массивы данных встречаются редко, и можно подумать, что ошибкой было бы не использовать в обучающем процессе все данные, имеющиеся в нашем распоряжении. Порой очень хочется заново использовать обучающие данные для тестирования или срезать углы, собирая тестовый набор данных. Но будьте осторожны: если последний составлен недостаточно внимательно, мы не сможем сделать значимых выводов по поводу нашей модели.

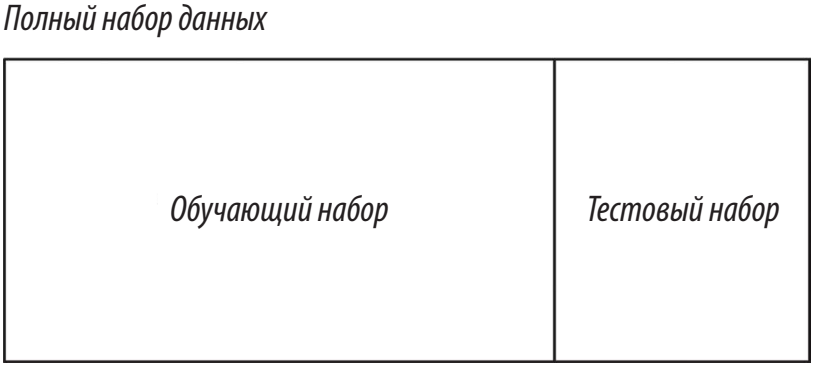


Рис. 2.12. *Мы часто делим данные на несовпадающие наборы для обучения и тестирования, чтобы дать справедливую оценку нашей модели*

В-третьих, вероятно, наступит момент, когда модель вместо исследования полезных признаков начнет переобучаться. Чтобы этого избежать, нужно предусмотреть немедленное завершение процесса при переобучении, что позволит избежать некорректных обобщений. Для этого тренировочный процесс делится на *эпохи*. Эпоха — одна итерация обучения на всем наборе. Если у нас есть набор размера d и мы проводим мини-пакетный градиентный спуск с размером пакета b, эпоха будет эквивалентна d/b обновлений. В конце каждой эпохи нужно измерить, насколько успешно наша модель обобщает. Для этого мы вводим дополнительный проверочный набор, показанный на рис. 2.13. В конце эпохи он покажет нам, как модель будет работать с еще не известными ей данными. Если точность на обучающем наборе будет возрастать, а для проверочного останется прежней или ухудшится, пора прекратить процесс: началось переобучение.

Проверочный набор данных полезен и как показатель точности при *оптимизации гиперпараметров*. Мы уже говорили о нескольких гиперпараметрах (темп обучения, размер мини-пакета и т. д.), но пока не разработали способов нахождения их оптимальных значений. Один из возможных вариантов — сеточный поиск, при котором мы выбираем значение для каждого

гиперпараметра из конечного набора вариантов (например, $\epsilon \in \{0,001, 0,01, 0,1\}$, размер мини-пакета $\in \{16, 64, 128, \ldots\}$) и обучаем модель на всех возможных вариантах. Мы выбираем сочетание гиперпараметров с лучшими результатами на проверочной выборке и получаем данные о точности модели, обученной с лучшим сочетанием, на тестовом наборе[12].

Рис. 2.13. *В глубоком обучении часто используется проверочный набор, препятствующий переобучению*

Прежде чем начать разговор о способах борьбы с переобучением, опишем рабочий процесс создания и обучения моделей глубокого обучения. Он подробно показан на рис. 2.14. Он сложен, но его понимание важно для правильного обучения нейросетей.

Сначала необходимо четко определить проблему. Мы рассматриваем входные данные, потенциальные выходные и векторное представление тех и других. Допустим, наша цель — обучение модели для выявления рака. Входные данные поступают в виде изображения в формате RGB, которое может быть представлено как вектор со значениями пикселов. Выходными данными будет распределение вероятностей по трем взаимоисключающим вариантам: 1) норма; 2) доброкачественная опухоль (без метастазов); 3) злокачественная опухоль (рак, давший метастазы в другие органы).

Далее нужно создать архитектуру нейросети для решения проблемы. Входной слой должен иметь достаточные размеры для приема данных изображения, а выходной должен быть размера 3 с мягким максимумом. Нам также следует определить внутреннюю архитектуру сети (количество скрытых слоев, связи и т. д.). В главе 5 мы поговорим об архитектуре моделей для компьютерного зрения, когда будем обсуждать сверточные нейросети. Еще нужно подобрать достаточно данных для обучения или моделирования. Они, возможно, будут представлены в виде фотографий патологий единообразного

размера, помеченных медицинским экспертом. Мы перемешиваем эти данные и разбиваем их на обучающий, проверочный и тестовый наборы.

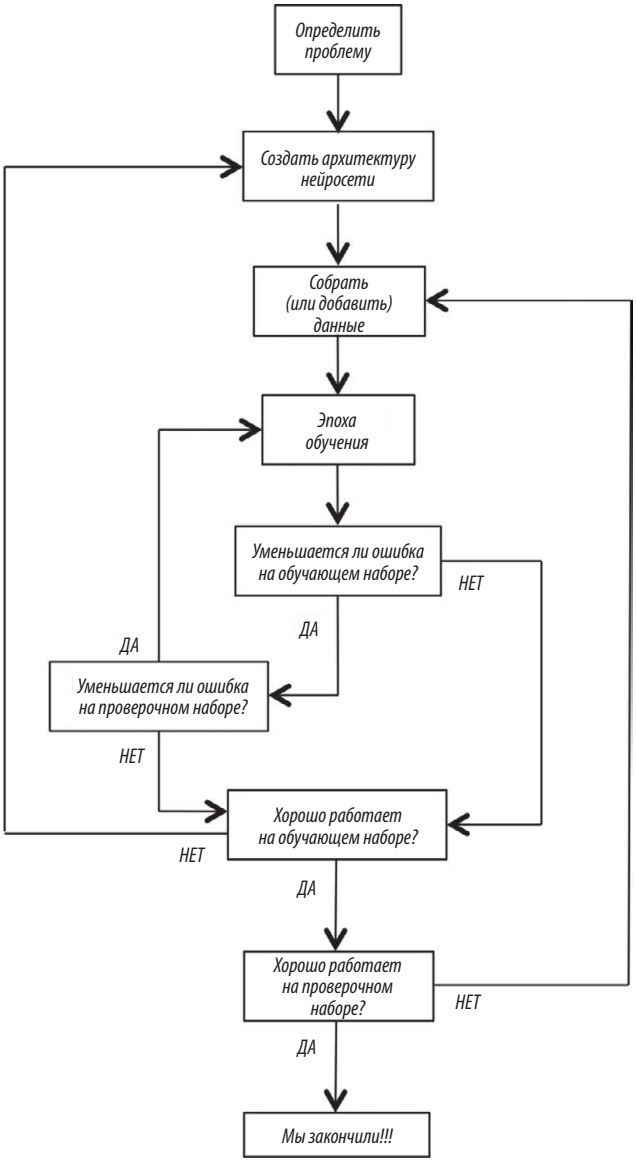


Рис. 2.14. *Подробный рабочий процесс и оценки модели глубокого обучения*

Мы готовы начать градиентный спуск. Мы тренируем модель на обучающем наборе в течение одной эпохи. В конце эпохи мы убеждаемся, что

ошибка на обучающем и проверочном наборах уменьшается. Когда улучшения прекращаются, мы останавливаемся и выясняем, устраивают ли нас результаты модели на тестовых данных. Если нет, следует пересмотреть архитектуру или подумать, действительно ли собранные данные содержат информацию, которая требуется для нужного нам предсказания. Если ошибка на обучающем наборе не уменьшается, возможно, стоит поработать над свойствами данных. Если не сокращается ошибка на проверочном наборе, пора принять меры против переобучения.

Если же нас устраивают результаты модели на обучающих данных, мы можем вычислить ее производительность на тестовых данных, с которыми она ранее не была знакома. Если результат неудовлетворителен, требуется добавить данных в обучающий набор, поскольку тестовый, вероятно, содержит примеры, которые были недостаточно представлены в обучающем. Если же все нормально, то мы закончили!

Борьба с переобучением в глубоких нейросетях

Есть несколько методов борьбы с переобучением. Ниже мы подробно их обсудим. Один из них носит название *регуляризации*. Он изменяет целевую функцию, которую мы минимизируем, добавляя условия, которые препятствуют появлениям больших весов. Иными словами, мы изменяем целевую функцию на $Error + \lambda f(\theta)$, где $f(\theta)$ увеличивается, когда компоненты θ растут, а λ — показатель регуляризации (еще один гиперпараметр). Значение λ определяет, в какой степени мы хотим защититься от переобучения. Если $\lambda = 0$, мы не принимаем никаких мер. Если λ слишком велико, приоритетом модели будет сохранение θ на низком уровне, а не нахождение значений параметров, которые дадут хорошие результаты на обучающем наборе. Выбор λ — очень важная задача, которая может потребовать ряда проб и ошибок.

Самый распространенный тип регуляризации в машинном обучении — так называемая *L2-регуляризация*[13]. Ее можно провести, дополнив функцию потерь квадратом величины всех весов в нейросети. Иными словами, для каждого веса w в нейросети мы добавляем $\frac{1}{2}\lambda w^2$ в функцию потерь. L2-регуляризация интуитивно интерпретируется как препятствующая появлению пиковых векторов весов и предпочитающая равномерные векторы весов.

Это полезное свойство, побуждающее сеть использовать в равной степени все входные данные, а не отдавать предпочтение одним входам в ущерб другим. К тому же в ходе градиентного спуска использование

L2-регуляризации в целом означает, что каждый вес линейно уменьшается до 0. Благодаря этому феномену L2-регуляризация получила второе название: *сокращение весов*.

Мы можем визуализировать эффекты L2-регуляризации с помощью ConvNetJS. Как на рис. 2.10 и 2.11, здесь используется нейросеть с двумя входами, двумя выходами с мягким максимумом и скрытый слой из 20 нейронов. Мы обучаем сети при помощи мини-пакетного градиентного спуска (размер пакета 10) и показателей регуляризации 0,01, 0,1 и 1. Результаты приведены на рис. 2.15.

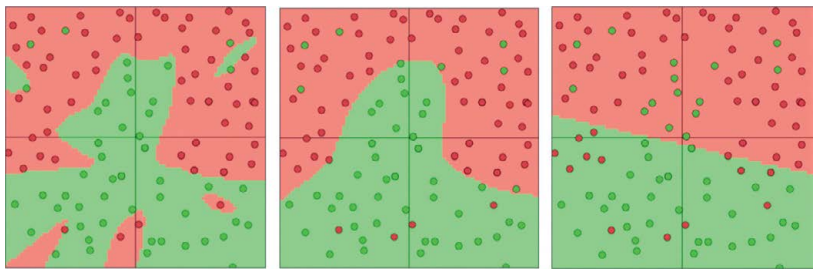


Рис. 2.15. *Визуализация нейросетей, обученных с показателями регуляризации 0,01, 0,1 и 1 (в таком порядке)*

Еще один распространенный вариант — *L1-регуляризация*. Здесь мы добавляем значение $\lambda|w|$ для каждого веса w в нейросети. L1-регуляризация обладает интригующим свойством: в ходе оптимизации векторы весов становятся очень разреженными (очень близкими к 0). Иными словами, нейроны начинают использовать небольшое количество самых важных входов и становятся устойчивыми к шуму на входе. А векторы весов, полученные при L2-регуляризации, обычно равномерны и невелики. L1-регуляризация очень полезна, когда вы хотите понять, какие именно свойства вносят вклад в принятие решения. Если такой уровень анализа свойств не нужен, мы используем L2-регуляризацию: она на практике работает лучше.

Максимальные ограничения нормы имеют схожую цель: это попытка предотвратить слишком большие значения θ, но более непосредственная[14]. Максимальные ограничения нормы задают абсолютную верхнюю границу для входного вектора весов каждого нейрона и при помощи метода проекции градиента устанавливают ограничение. Иными словами, каждый раз, когда шаг градиентного спуска изменяет входящий вектор весов, так что $||w||_2 > c$, мы проецируем вектор обратно на шар (центр которого

расположен в исходной точке) с радиусом c. Типичные значения c — 3 и 4. Примечательно, что вектор параметров не может выйти из-под контроля (даже если нормы обучения слишком высоки), поскольку обновления весов всегда ограничены.

Совсем иной метод борьбы с переобучением — *прореживание (Dropout)*, который особенно популярен у специалистов по глубоким нейросетям[15]. При обучении он используется так: нейрон становится активным только с некой вероятностью p (гиперпараметр), иначе его значение приравнивается к 0. На интуитивном уровне можно решить, что это заставляет нейросеть оставаться точной даже в условиях недостатка информации. Сеть перестает быть слишком зависимой от отдельного нейрона или их небольшого сочетания. С точки зрения математики прореживание препятствует переобучению, давая возможность приблизительно сочетать экспоненциально большое количество архитектур нейросетей, причем эффективно. Процесс прореживания показан на рис. 2.16.

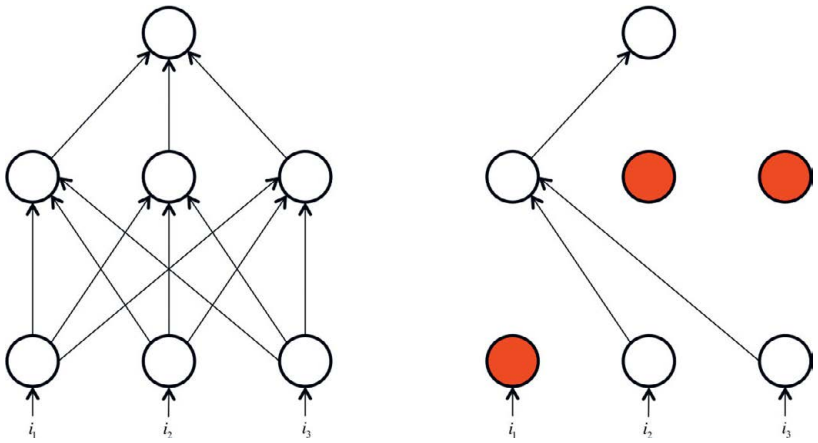


Рис. 2.16. *Прореживание помечает каждый нейрон сети как неактивный с некой случайной вероятностью на каждом этапе обучения*

Прореживание — понятный процесс, но стоит учесть несколько важных моментов. Во-первых, нужно, чтобы выходные значения нейронов во время тестирования были эквивалентны ожидаемым выходным значениям в процессе обучения. Мы можем добиться этого наивным способом, масштабировав параметры в тесте. Например, если $p = 0,5$, нейроны должны вдвое уменьшить выходные значения в тесте, чтобы обеспечить те же (ожидаемые) параметры в ходе обучения. Ведь выходное значение нейрона равно 0 с вероятностью $(1 - p)$. И если до прореживания оно равнялось x, после прореживания

ожидаемое значение будет $E[\text{output}] = px + (1 - p) \cdot 0 = px$. Но такое применение операции нежелательно, поскольку предполагает масштабирование выходных значений нейрона во время тестирования. Результаты тестов очень важны для оценки модели, и предпочтительнее использовать *обратное прореживание*, при котором масштабирование происходит в процессе обучения, а не тестирования. Выходное значение любого нейрона, активность которого не заглушена, делится на p перед передачей его на следующий уровень. Теперь

$$E\left[\text{output}\right] = p \cdot \frac{x}{p} + \left(1 - p\right) \cdot 0 = x,$$

что позволит не прибегать к произвольному масштабированию выходных значений нейрона во время тестирования.

Резюме

Мы познакомились с основами обучения нейронных сетей с прямым распространением сигнала, поговорили о градиентном спуске, алгоритме обратного распространения ошибки, а также методах борьбы с переобучением. В следующей главе мы применим полученные знания на практике, используя библиотеку `TensorFlow` для эффективного создания первых нейросетей. В главе 4 мы вернемся к проблеме оптимизации целевых функций для обучения нейросетей и разработки алгоритмов, значительно повышающих качество обучения. Эти улучшения позволят обрабатывать гораздо больше данных, а следовательно, и строить более сложные модели.

ГЛАВА 3

Нейросети в TensorFlow

Что такое TensorFlow?

Мы могли бы на протяжении всей книги описывать абстрактные модели глубокого обучения, но надеемся, что в итоге вы не только поймете, как они работают, но и получите навыки, необходимые для создания их с нуля, чтобы решать задачи в вашей области. Вы уже лучше понимаете теорию моделей глубокого обучения, так что в этой главе мы обсудим программную реализацию некоторых алгоритмов.

Основной инструмент, который нам нужен, называется `TensorFlow`[16]. Это открытая программная библиотека, выпущенная в 2015 году Google, чтобы облегчить создание, разработку и обучение моделей. `TensorFlow` изначально была внутренней библиотекой для разработчиков Google, и мы думаем, что в открытую версию будут добавляться новые функции по мере их тестирования и проверки в Google. `TensorFlow` — лишь один из вариантов, доступных разработчикам, и мы выбрали эту библиотеку за продуманный дизайн и простоту использования. Краткое ее сопоставление с альтернативами будет дано в следующем разделе.

`TensorFlow` — библиотека `Python`, которая дает пользователям возможность выражать произвольные вычисления в виде графа *потоков данных*. Узлы графа соответствуют математическим операциям, а ребра — данным, которые передаются из одного узла в другой. Данные в `TensorFlow` представлены в виде тензоров — многомерных массивов (векторы — одномерные тензоры, матрицы — двумерные и т. д.).

Такой способ представления полезен во многих областях, но `TensorFlow` в основном используется для глубокого обучения в практике и исследованиях. Представление нейросетей в виде тензоров, и наоборот, — не тривиальная задача, а скорее навык, который нужно развить при работе с этой книгой. Это позволит применить варианты ускорения, которые обеспечивают современные компьютеры (например, для параллельных тензорных

операций на графических процессорах) и даст четкий и выразительный способ внедрения моделей. Мы поговорим об основах использования TensorFlow и рассмотрим два простых примера (логистическую регрессию и многослойные сети с прямым распространением сигнала). Но, прежде чем погрузиться в предмет, сопоставим в общих чертах TensorFlow с альтернативами для моделей глубокого обучения.

Сравнение TensorFlow с альтернативами

Помимо TensorFlow, есть и ряд других библиотек для создания глубоких нейросетей. Это Theano, Torch, Caffe, Neon и Keras[17]. На основании двух простых критериев (выразительность и наличие активного сообщества разработчиков) мы в итоге сократили поле выбора до TensorFlow, Theano (создана в LISA Lab Монреальского университета) и Torch (в основном поддерживается командой Facebook AI Research).

Все три библиотеки могут похвастать солидным сообществом разработчиков, позволяют манипулировать тензорами с незначительными ограничениями и обеспечивают возможность автоматического дифференцирования (что позволяет пользователям обучать модели глубокого обучения без необходимости адаптировать алгоритм обратного распространения ошибок для различных архитектур нейросетей, как мы делали в предыдущей главе). Один из недостатков Torch, однако, в том, что эта среда написана на Lua. Это скриптовый язык, который напоминает Python, но мало используется за пределами собственного глубокого обучения. Мы решили не заставлять новичков осваивать новый язык ради создания моделей, так что вариантов теперь два: TensorFlow и Theano.

Из этих двух кандидатов выбрать было труднее (первый вариант главы написан с использованием Theano), но в конце концов мы остановились на TensorFlow по ряду незначительных причин. Во-первых, в Theano нужен дополнительный шаг — «компиляция графа», который занимает много времени при разработке определенных видов архитектур глубокого обучения. Хотя фаза компиляции невелика по сравнению со временем обучения, при написании и отладке нового кода она кажется неприятной. Во-вторых, у TensorFlow гораздо более понятный интерфейс. Многие классы моделей можно выразить значительно меньшим числом строк, не жертвуя общей выразительностью структуры. Наконец, TensorFlow создавалась для использования в продуктивных системах, а Theano разрабатывали ученые почти исключительно для исследовательских целей. Поэтому у TensorFlow много полезных функций и свойств, которые делают эту библиотеку лучшим вариантом для реальных систем (способность

работать в мобильной среде, легко создавать модели для запуска на нескольких графических процессорах на одной машине и обучать масштабные сети распределенным методом). Знакомство с `Theano` и `Torch` полезно при изучении примеров из открытых источников, но анализ этих библиотек выходит за рамки этой книги*.

Установка TensorFlow

Установка `TensorFlow` в вашей локальной среде разработки не представляет особой проблемы, если вы не планируете вносить изменения в исходный код `TensorFlow`. Воспользуйтесь менеджером установки на `Python` под названием `Pip`. Если он еще не установлен на вашем компьютере, используйте следующие команды ввода:

```
# Ubuntu/Linux 64-bit
$ sudo apt-get install python-pip python-dev
# Mac OS X
$ sudo easy_install pip
```

Установив `Pip` (версия 8.1 или новее), примените следующие команды для установки `TensorFlow`. Отметим разницу в именовании пакетов `Pip`, если мы хотим загрузить версию `TensorFlow` для графических процессоров (настоятельно советуем так и поступить):

```
$ pip install --upgrade tensorflow      # for Python 2.7
$ pip3 install --upgrade tensorflow     # for Python 3.n
$ pip install --upgrade tensorflow-gpu  # for Python 2.7
                                        # and GPU
$ pip3 install --upgrade tensorflow-gpu # for Python 3.n
                                        # and GPU
```

Дополненные обновленные инструкции и подробности по поводу установки приложения можно найти на сайте `TensorFlow`[18].

Создание переменных TensorFlow и работа с ними

Создавая модель глубокого обучения в `TensorFlow`, мы используем переменные для представления параметров модели. Переменные

* В сентябре 2017 года объявлено, что разработка Theano будет прекращена после выпуска версии 1.0 (см. https://groups.google.com/forum/#!msg/theano-users/7Poq8BZutbY/rNCIfvAEAwAJ). Для Torch создали реализацию на Python, названную PyTorch. Эта новая библиотека стремительно набирает популярность. *Прим. науч. ред.*

TensorFlow — буферы в оперативной памяти, содержащие тензоры. Однако, в отличие от нормальных тензоров, которые создаются только при запуске графа и затем тут же стираются из памяти, переменные переживают несколько выполнений графа. Поэтому они обладают следующими тремя свойствами.

— Переменные должны быть явно инициализированы до того, как граф будет использован впервые.

— Можно использовать градиентные методы модификации переменных после каждой итерации, ведь мы ищем оптимальные параметры модели.

— Значения, хранимые в переменных, можно скопировать на диск и восстанавливать для дальнейшего использования.

Эти три свойства делают TensorFlow особенно полезным инструментом в разработке моделей машинного обучения.

Создание переменных — процесс несложный, и в TensorFlow можно инициализировать их несколькими способами. Начнем с переменной, которая описывает веса, соединяющие нейроны двух слоев сети с прямым распространением сигнала:

```
weights = tf.Variable(tf.random_normal([300, 200], stddev=0.5),
                      name="weights")
```

Здесь мы сообщаем tf.Variable два аргумента[19]. Первый — tf.random_normal — операция, которая создает тензор, инициализированный при помощи нормального распределения со стандартным отклонением 0,5[20]. Мы указали размер тензора — 300×200, подразумевая, что веса соединяют слой из 300 нейронов со слоем из 200 нейронов. Мы задали имя обращения к tf.Variable. Это уникальный идентификатор, который позволяет обращаться к соответствующему узлу в графе вычислений. В этом случае веса считаются обучаемыми; мы будем автоматически вычислять градиенты и применять их к весам. Если они не должны быть обучаемыми, мы можем установить дополнительный флаг при обращении к tf.Variable:

```
weights = tf.Variable(tf.random_normal([300, 200], stddev=0.5),
                      name="weights", trainable=False)
```

Помимо tf.random_normal, есть еще несколько методов инициализации переменной TensorFlow:

```
# Common tensors from the TensorFlow API docs
tf.zeros(shape, dtype=tf.float32, name=None)
tf.ones(shape, dtype=tf.float32, name=None)
```

```
tf.random_normal(shape, mean=0.0, stddev=1.0,
                 dtype=tf.float32, seed=None,
                 name=None)
tf.truncated_normal(shape, mean=0.0, stddev=1.0,
                 dtype=tf.float32, seed=None,
                 name=None)
tf.random_uniform(shape, minval=0, maxval=None,
                 dtype=tf.float32, seed=None,
                 name=None)
```

При обращении к `tf.Variable` к графу вычислений добавляются три операции:

— операция, создающая тензор для инициализации переменной;

— операция `tf.assign`, которая наполняет переменную инициализирующим тензором до ее использования;

— операция переменной, которая содержит ее текущее значение.

Визуализация приведена на рис. 3.1.

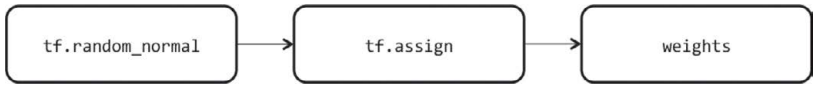


Рис. 3.1. *Три операции добавляются при реализации переменной TensorFlow. Здесь мы инициализируем переменные веса при помощи нормального распределения*

Как мы уже говорили, прежде чем использовать переменную `TensorFlow`, нужно запустить операцию `tf.assign`[21], чтобы переменной было присвоено желаемое начальное значение. Для этого можно запустить `tf.initialize_all_variables()`[22], что приведет к выполнению всех операций `tf.assign` в нашем графе. Мы можем также выборочно инициализировать некоторые переменные графа вычислений командой `tf.initialize_variables(var1, var2, ...)`[23]. Подробнее обо всем этом мы поговорим при обсуждении сессий `TensorFlow`.

Операции в TensorFlow

Мы уже упомянули о некоторых операциях в контексте инициализации переменных, но в `TensorFlow` доступны и многие другие. Они выражают абстрактные трансформации, которые применяются к тензорам в графе

вычислений. Операции могут иметь атрибуты, которые либо заданы изначально, либо вводятся в процессе работы. Например, атрибут может описывать ожидаемые типы входных данных (добавление тензоров типа `float32` или `int32`). Операциям, как и переменным, может присваиваться имя для простоты обращения на графе вычислений. Они состоят из одного или более ядер, которые содержат специфичные для устройств реализации.

Например, операция может иметь отдельные ядра для `CPU` и `GPU`, поскольку на `GPU` ее реализация более эффективна. Это справедливо для многих действий с матрицами в `TensorFlow`.

Общий обзор типов доступных операций приведен в таблице 3.1, взятой из оригинальной технической документации `TensorFlow`[24].

Таблица 3.1. *Общая таблица операций в TensorFlow*

Категория	**Примеры**
Поэлементные математические операции	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, …
Операции над массивами	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, …
Матричные операции	MatMul, MatrixInverse, MatrixDeterminant, …
Операции с отслеживанием состояния	Variable, Assign, AssignAdd, …
Стандартные блоки нейросетей	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, …
Операции контрольных точек	Save, Restore
Операции очередей и синхронизации	Enqueue, Dequeue, MutexAcquire, MutexRelease, …
Операции потока команд управления	Merge, Switch, Enter, Leave, NextIteration

Тензоры-заполнители

Теперь, обладая солидными познаниями в области переменных и операций в `TensorFlow`, мы имеем практически полное описание компонентов графа вычислений этой библиотеки. Не хватает представлений о том, как обеспечить ввод данных в глубокую модель при обучении и тестировании. Переменной недостаточно: она инициализируется лишь однажды. Нам нужен компонент, который мы будем загружать каждый раз при запуске графа вычислений.

TensorFlow решает эту проблему при помощи так называемого *заполнителя*[25]. Его можно использовать в операциях так же, как и все обычные переменные и тензоры TensorFlow, и он реализуется следующим образом:

```
x = tf.placeholder(tf.float32, name="x", shape=[None, 784])
W = tf.Variable(tf.random_uniform([784,10], -1, 1), name="W")
multiply = tf.matmul(x, W)
```

Здесь мы определяем заполнитель, где x — мини-пакет данных, сохраненных как float32. Можно отметить, что в x 784 столбца, то есть каждый образец данных имеет 784 измерения. Для x не определено число строк. Это значит, что он может быть инициализирован произвольным количеством образцов данных. Можно и отдельно умножать каждый из них на W, но представление всего мини-пакета в виде тензора позволяет вычислить результаты для всех образцов данных параллельно. В результате *i-я* строка тензора multiply соответствует W, умноженному на *i-й* образец данных.

Как и переменные, которые нужно инициализировать при первом построении графа вычислений, заполнители нужно наполнять каждый раз при запуске графа (или подграфа). Подробнее об этом мы поговорим в следующем разделе.

Сессии в TensorFlow

Программа TensorFlow взаимодействует с графом вычислений в рамках *сессии*[26]. В ходе сессии TensorFlow происходит создание изначального графа, а также инициализация всех переменных и запуск графа вычислений. Для анализа этих элементов рассмотрим простой скрипт на Python:

```
import tensorflow as tf
from read_data import get_minibatch()
x = tf.placeholder(tf.float32, name="x", shape=[None, 784])
W = tf.Variable(tf.random_uniform([784, 10], -1, 1), name="W")
b = tf.Variable(tf.zeros([10]), name="biases")
output = tf.matmul(x, W) + b
init_op = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init_op)
feed_dict = {"x" : get_minibatch()}
sess.run(output, feed_dict=feed_dict)
```

Первые четыре строчки после оператора импорта описывают граф вычислений, который строится в ходе сессии, когда она будет создана. Этот граф (без операций инициализации переменных) изображен на рис. 3.2. Затем мы инициализируем переменные, используя сессию для запуска соответствующей операции — `sess.run(init_op)`. Наконец, мы можем запустить подграф, вновь обратившись к `sess.run`, но уже передав на вход тензор или список тензоров, которые мы хотим вычислить, и `feed_dict`, который вводит необходимые данные в заполнители.

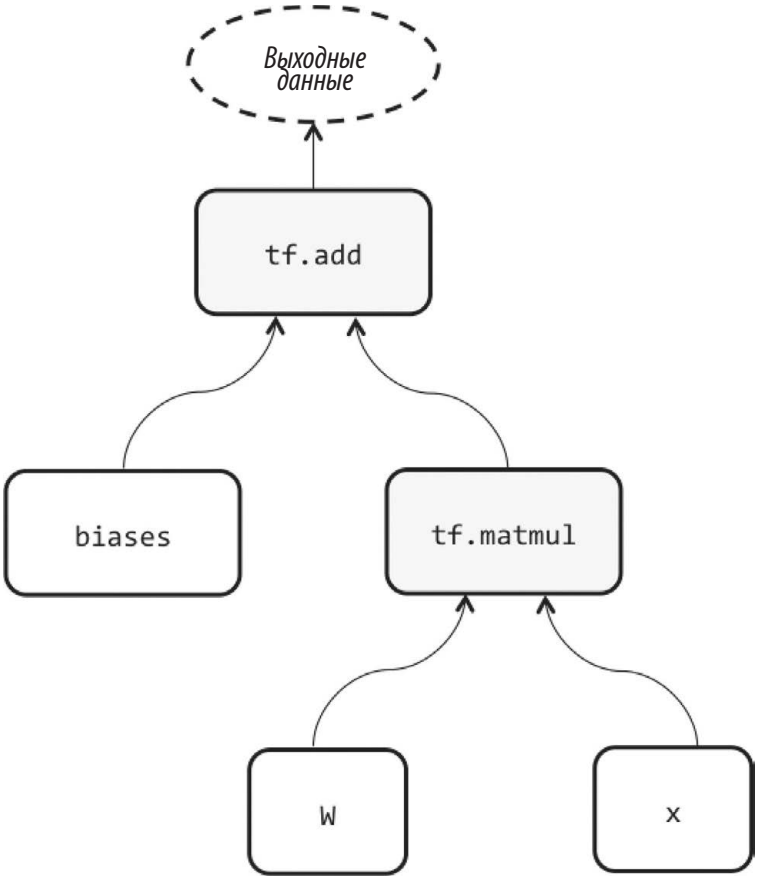


Рис. 3.2. *Пример простого графа вычислений в TensorFlow*

Наконец, интерфейс `sess.run` можно использовать для обучения сетей. Мы рассмотрим это подробнее, когда при помощи `TensorFlow` будем обучать нашу первую модель на `MNIST`. Но как именно единственная строка кода `(sess.run)` выполняет так много функций? Все дело в выразительности

лежащего в ее основе графа вычислений. Его функции представлены в виде операций `TensorFlow`, которые передаются в `sess.run` в качестве аргументов. Ей остается обратиться к графу вычислений и определить все зависимости, которые образуют соответствующий подграф; убедиться, что все переменные-заполнители, принадлежащие к выявленному подграфу, заданы при помощи `feed_dict`, и пройти по подграфу (выполнив промежуточные операции), чтобы вычислить исходные аргументы.

Теперь мы обратимся к еще двум важнейшим понятиям из области создания графов вычислений и управления ими.

Области видимости переменной и совместное использование переменных

Мы пока не встречались с этой задачей. Создание сложных моделей часто требует повторного и совместного использования больших наборов переменных, которые желательно создавать в одном месте. К сожалению, попытки обеспечить модульность и читабельность порой приводят к неожиданным проблемам, если мы неосторожны. Рассмотрим пример:

```
def my_network(input):
      W_1 = tf.Variable(tf.random_uniform([784, 100], -1, 1),
                                        name="W_1")
      b_1 = tf.Variable(tf.zeros([100]), name="biases_1")
      output_1 = tf.matmul(input, W_1) + b_1
      W_2 = tf.Variable(tf.random_uniform([100, 50], -1, 1),
                                        name="W_2")
      b_2 = tf.Variable(tf.zeros([50]), name="biases_2")
      output_2 = tf.matmul(output_1, W_2) + b_2
      W_3 = tf.Variable(tf.random_uniform([50, 10], -1, 1),
                                        name="W_3")
      b_3 = tf.Variable(tf.zeros([10]), name="biases_3")
      output_3 = tf.matmul(output_2, W_3) + b_3
      # printing names
      print "Printing names of weight parameters"
      print W_1.name, W_2.name, W_3.name
      print "Printing names of bias parameters"
      print b_1.name, b_2.name, b_3.name
      return output_3
```

Эта сеть включает шесть переменных, описывающих три слоя. Поэтому, чтобы использовать ее несколько раз, мы пробуем заключить ее в компактную функцию вроде `my_network`, к которой можно обращаться несколько раз. Но если мы попытаемся использовать эту сеть с двумя разными входными параметрами, получится нечто неожиданное:

```
In [1]: i_1 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_1")
In [2]: my_network(i_1)
Printing names of weight parameters
W_1:0 W_2:0 W_3:0
Printing names of bias parameters
biases_1:0 biases_2:0 biases_3:0
Out[2]: <tensorflow.python.framework.ops.Tensor …>
In [1]: i_2 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_2")
In [2]: my_network(i_2)
Printing names of weight parameters
W_1_1:0 W_2_1:0 W_3_1:0
Printing names of bias parameters
biases_1_1:0 biases_2_1:0 biases_3_1:0
Out[2]: <tensorflow.python.framework.ops.Tensor …>
```

Если приглядеться, во втором обращении к `my_network` используются не те переменные, что в первом (имена различны). Мы создали второй набор переменных! Чаще мы хотим не создавать копию, а повторно использовать модель и ее переменные. Оказывается, в этом случае не стоит использовать `tf.Variable`. Нужно применить более сложную схему именования, которая использует область видимости переменных `TensorFlow`.

Механизмы области видимости переменных `TensorFlow` по большей части контролируются двумя функциями.

```
tf.get_variable(<name>, <shape>, <initializer>)
```

Проверяет, существует ли переменная с этим именем, выбирает ее, если та существует, или создает ее при помощи формы и функции инициализации, если ее еще не существует[27].

```
tf.variable_scope(<scope_name>)
```

Управляет пространством имен и определяет область видимости `tf.get_variable`[28].

Попытаемся четче переписать `my_network` при помощи области видимости переменных `TensorFlow`.

Новые названия переменных включают пространство имен, например `layer1/W`, `layer2/b` и т. д.:

```
def layer(input, weight_shape, bias_shape):
      weight_init = tf.random_uniform_initializer(minval=-1,
           maxval=1)
      bias_init = tf.constant_initializer(value=0)
      W = tf.get_variable("W", weight_shape,
                          initializer=weight_init)
      b = tf.get_variable("b", bias_shape,
                          initializer=bias_init)
      return tf.matmul(input, W) + b
def my_network(input):
      with tf.variable_scope("layer_1"):
           output_1 = layer(input, [784, 100], [100])
      with tf.variable_scope("layer_2"):
           output_2 = layer(output_1, [100, 50], [50])
      with tf.variable_scope("layer_3"):
           output_3 = layer(output_2, [50, 10], [10])
      return output_3
```

Попробуем дважды обратиться к `my_network`, как мы сделали в предыдущем фрагменте кода:

```
In [1]: i_1 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_1")
In [2]: my_network(i_1)
Out[2]: <tensorflow.python.framework.ops.Tensor …>
In [1]: i_2 = tf.placeholder(tf.float32, [1000, 784],
                             name="i_2")
In [2]: my_network(i_2)
ValueError: Over-sharing: Variable layer_1/W already exists…
```

В отличие от `tf.Variable`, команда `tf.get_variable` проверяет, что переменная с соответствующим именем еще не создана. По умолчанию совместное использование запрещено (из соображений безопасности!), но если мы хотим его решить в области видимости, то должны прописать это прямо:

```
with tf.variable_scope("shared_variables") as scope:
      i_1 = tf.placeholder(tf.float32, [1000, 784], name="i_1")
      my_network(i_1)
      scope.reuse_variables()
      i_2 = tf.placeholder(tf.float32, [1000, 784], name="i_2")
      my_network(i_2)
```

Это позволяет сохранить модульность, не запрещая совместное использование переменных. К тому же схема именования становится проще и гораздо удобнее.

Управление моделями на CPU и GPU

`TensorFlow` позволяет при необходимости пользоваться несколькими вычислительными устройствами для обучения модели. Поддерживаемые устройства представлены строковыми идентификаторами, что обычно выглядит так:

```
"/cpu:0"
```

`CPU` нашей машины.

```
"/gpu:0"
```

Первый `GPU` нашей машины (если есть).

```
"/gpu:1"
```

Второй `GPU` нашей машины (если есть).

Если у операции есть ядра `CPU` и `GPU` и разрешено использование `GPU`, `TensorFlow` автоматически будет выбирать вариант для `GPU`. Чтобы посмотреть, какие устройства использует граф вычислений, мы можем инициализировать сессию `TensorFlow`, установив параметр `log_device_placement` в значение `True`:

```
sess = tf.Session(config=tf.ConfigProto(
      log_device_placement=True))
```

Если мы хотим использовать конкретное устройство, можно его выбрать с помощью конструкции `with tf.device`[29]. Но если оно недоступно, возникнет ошибка. Нужно, чтобы `TensorFlow` нашла другое доступное устройство, если выбранное не существует; можно передать флаг `allow_soft_placement` в переменную сессии таким образом[30]:

```
with tf.device('/gpu:2'):
      a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2], name='a')
```

```
        b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')
        c = tf.matmul(a, b)
sess = tf.Session(config=tf.ConfigProto(
            allow_soft_placement=True, log_device_placement=True))
sess.run(c)
```

TensorFlow также позволяет строить модели, которые используют несколько GPU. Они создаются в виде башни, как показано на рис. 3.3.

Ниже приведен пример кода для нескольких GPU:

```
c = []
for d in ['/gpu:0', '/gpu:1']:
        with tf.device(d):
        a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2],
             name='a')
        b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')
        c.append(tf.matmul(a, b))
with tf.device('/cpu:0'):
        sum = tf.add_n(c)
sess = tf.Session(config=tf.ConfigProto(
                    log_device_placement=True))
sess.run(sum)
```

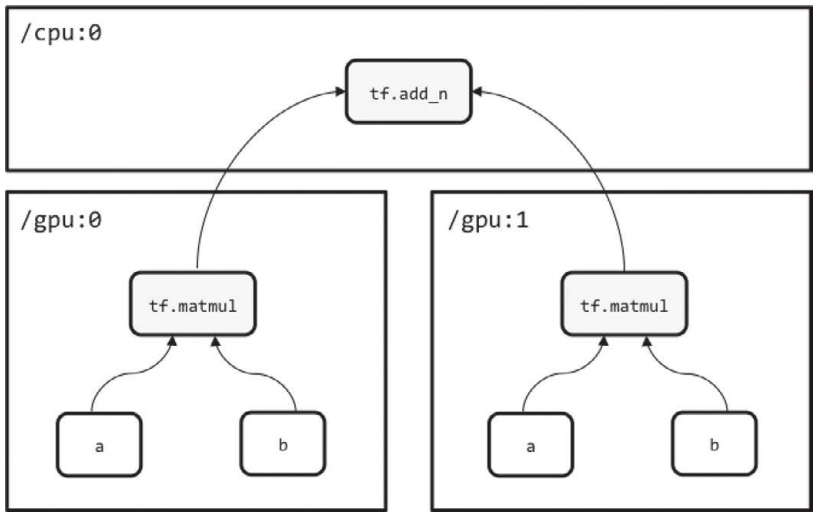


Рис. 3.3. *Создание моделей для нескольких GPU в виде башни*

Создание модели логистической регрессии в TensorFlow

Мы рассмотрели базовые понятия `TensorFlow` и можем построить простую модель для набора данных `MNIST`. Как вы, наверное, помните, наша цель — распознать рукописные цифры по черно-белым изображениям 28×28 единиц. Первая сеть, которую мы построим, реализует простой алгоритм машинного обучения — логистическую регрессию[31].

Логистическая регрессия — метод, с помощью которого мы можем вычислить вероятность того, что входные данные относятся к одному из целевых классов. Определим вероятность того, что данное изображение — 0, 1… или 9.

Наша модель использует матрицу W, которая представляет веса соединений в сети, и вектор b, соответствующий смещению, для вычисления того, принадлежит ли входящее значение x классу i, при помощи функции мягкого максимума (softmax), о которой мы уже говорили выше:

$$P\left(y=i\middle|x\right)=softmax_i\left(Wx+b\right)=\frac{e^{W_i x+b_i}}{\sum_j e^{W_j x+b_j}}.$$

Наша задача — определить значения W и b, которые будут наиболее эффективно и точно классифицировать входящие данные. Сеть логистической регрессии можно выразить в схеме (рис. 3.4). Для простоты мы опустили смещения и их соединения.

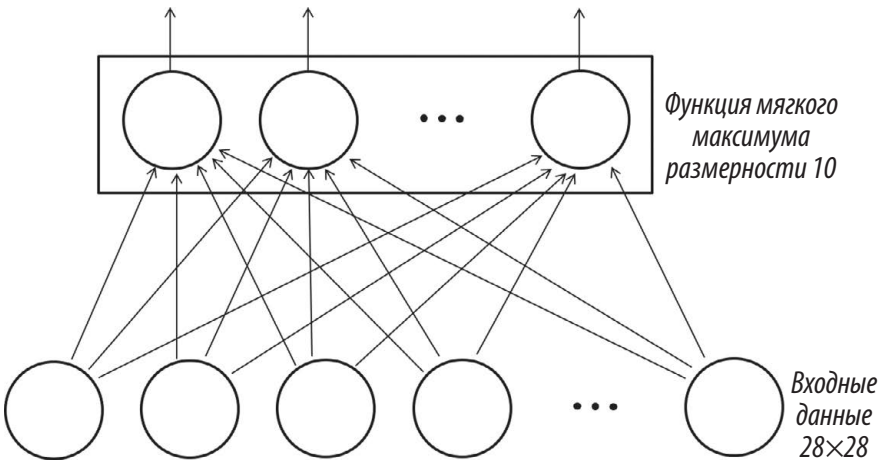


Рис. 3.4. *Интерпретация логистической регрессии как примитивной нейросети*

Легко заметить, что сеть для интерпретации логистической регрессии довольно примитивна. У нее нет скрытых слоев, а следовательно, ее способность усваивать сложные взаимоотношения ограничена. У нас есть выходная функция мягкого максимума размерности 10, поскольку у нас 10 возможных исходов для каждого входного значения. Более того, есть входной слой размера 784 — один входной нейрон для каждого пиксела изображения! Модель в целом способна корректно классифицировать наш набор данных, но еще есть куда расти. До конца этой главы и в главе 5 мы будем стараться повысить точность нашей работы. Но сначала посмотрим, как реализовать эту логистическую сеть в `TensorFlow`, чтобы обучить ее на нашем компьютере.

Модель логистической регрессии строится в четыре этапа:

1) `inference`: создается распределение вероятностей по выходным классам для мини-пакета*;

2) `loss`: вычисляется значение функции потерь (в нашем случае перекрестная энтропии);

3) `training`: отвечает за вычисление градиентов параметров модели и ее обновление;

4) `evaluate`: определяется эффективность модели.

Для мини-пакета из 784-мерных векторов, соответствующих изображениям `MNIST`, мы можем выразить логистическую регрессию через функцию мягкого максимума от входных данных, умноженных на матрицу, которая представляет веса соединений входного и выходного слоев.

Каждая строка выходного тензора содержит распределение вероятностей по классам для соответствующего образца данных в мини-выборке:

```
def inference(x):
tf.constant_initializer(value=0)
W = tf.get_variable("W", [784, 10],
                    initializer=init)
b = tf.get_variable("b", [10],
                    initializer=init)
output = tf.nn.softmax(tf.matmul(x, W) + b)
return output
```

* Для каждого экземпляра данных в мини-пакете нейронная сеть выдает вероятность принадлежности данных к каждому классу (то есть вероятность того, что на исходном изображении 0, 1, 2 и так далее до 9). *Прим. науч. ред.*

Теперь, с правильными метками для мини-пакета, мы можем вычислить среднюю ошибку на образец данных. При этом применяется следующий фрагмент кода, который вычисляет перекрестную энтропию по всему мини-пакету:

```
def loss(output, y):
      dot_product = y * tf.log(output)
      # Reduction along axis 0 collapses each column into a
      # single value, whereas reduction along axis 1 collapses
      # each row into a single value. In general, reduction along
      # axis i collapses the ith dimension of a tensor to size 1.
      xentropy = -tf.reduce_sum(dot_product, reduction_indices=1)
      loss = tf.reduce_mean(xentropy)
      return loss
```

Теперь, имея значение потерь, мы вычисляем градиенты и модифицируем наши параметры соответственно. `TensorFlow` облегчает процесс, обеспечивая доступ к встроенным оптимизаторам, которые выдают специальную операцию обучения. Ее можно запустить в сессии для минимизации ошибок. Отметим, что, создавая операцию обучения, мы передаем переменную, которая отражает количество обрабатываемых мини-выборок. Каждый раз, когда операция запускается, растет эта переменная, и мы можем отслеживать процесс:

```
def training(cost, global_step):
      optimizer = tf.train.GradientDescentOptimizer(
      learning_rate)
      train_op = optimizer.minimize(cost,
      global_step=global_step)
      return train_op
```

Наконец, мы можем создать простой вычислительный подграф для оценки модели на проверочных или тестовых данных:

```
def evaluate(output, y):
      correct_prediction = tf.equal(tf.argmax(output, 1),
                              tf.argmax(y, 1))
      accuracy = tf.reduce_mean(tf.cast(correct_prediction,
                                  tf.float32))
      return accuracy
```

На этом настройка графа в `TensorFlow` для модели логистической регрессии завершена.

Журналирование и обучение модели логистической регрессии

У нас есть все компоненты, можно сводить их воедино. Чтобы сохранять важную информацию в процессе обучения модели, мы записываем в журнал несколько сводок статистики. Например, мы используем команды `tf.scalar_summary`[32] и `tf.histogram_summary`[33] для записи ошибки в каждом мини-пакете, ошибки на проверочном множестве и распределения параметров. Для примера приведем скалярную сводку статистик для функции потерь:

```
def training(cost, global_step):
        tf.scalar_summary("cost", cost)
        optimizer = tf.train.GradientDescentOptimizer(
        learning_rate)
        train_op = optimizer.minimize(cost,
        global_step=global_step)
        return train_op
```

На каждой эпохе мы запускаем `tf.merge_all_summaries`[34], чтобы собрать все записанные сводки, и с помощью `tf.train.SummaryWriter` сохраняем журнал на диске. В следующем разделе мы расскажем, как визуализировать эти журналы при помощи встроенного инструмента `TensorBoard`.

Помимо сводок статистики, мы сохраняем параметры модели с помощью `tf.train.Saver`. По умолчанию это средство поддерживает пять последних контрольных точек, которые мы можем восстанавливать для дальнейшего использования. В результате получаем следующий скрипт на `Python`:

```
# Parameters
learning_rate = 0.01
training_epochs = 1000
batch_size = 100
display_step = 1
with tf.Graph().as_default():
        # mnist data image of shape 28*28=784
        x = tf.placeholder("float", [None, 784])
        # 0-9 digits recognition => 10 classes
        y = tf.placeholder("float", [None, 10])
```

```
        output = inference(x)
        cost = loss(output, y)
        global_step = tf.Variable(0, name='global_step',
                                  trainable=False)
        train_op = training(cost, global_step)
        eval_op = evaluate(output, y)
        summary_op = tf.merge_all_summaries()
        saver = tf.train.Saver()
        sess = tf.Session()
        summary_writer = tf.train.SummaryWriter("logistic_logs/",
                                        graph_def=sess.graph_def)
        init_op = tf.initialize_all_variables()
        sess.run(init_op)
# Training cycle
for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
             mbatch_x, mbatch_y = mnist.train.next_batch(
             batch_size)
             # Fit training using batch data
             feed_dict = {x : mbatch_x, y : mbatch_y}
             sess.run(train_op, feed_dict=feed_dict)
             # Compute average loss
             minibatch_cost = sess.run(cost,
                    feed_dict=feed_dict)
             avg_cost += minibatch_cost/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
             val_feed_dict = {
             x : mnist.validation.images,
             y : mnist.validation.labels
             }
             accuracy = sess.run(eval_op,
```

```
                feed_dict=val_feed_dict)
        print "Validation Error:", (1 - accuracy)
        summary_str = sess.run(summary_op,
                                    feed_dict=feed_dict)
        summary_writer.add_summary(summary_str,
                                    sess.run(global_step))
        saver.save(sess, "logistic_logs/model-checkpoint",
                           global_step=global_step)
print "Optimization Finished!"
test_feed_dict = {
        x : mnist.test.images,
        y : mnist.test.labels
}
accuracy = sess.run(eval_op, feed_dict=test_feed_dict)
print "Test Accuracy:", accuracy
```

Запуск этого скрипта обеспечивает нам аккуратность* в 91,9% по тестовому набору из 100 эпох обучения. Уже неплохо, но в последнем разделе главы мы постараемся улучшить этот результат при помощи нейросети с прямым распространением сигнала.

Применение TensorBoard для визуализации вычислительного графа и обучения

Настроив журналирование сводок статистики так, как показано в предыдущем разделе, мы можем визуализировать собранные данные. В `TensorFlow` предусмотрен инструмент, который обеспечивает простой в использовании интерфейс навигации по сводкам[35]. Запуск `TensorBoard` несложен:

```
tensorboard --logdir=<absolute_path_to_log_dir>
```

Параметр `logdir` должен быть установлен на каталог, в котором `tf.train.SummaryWriter` фиксировал сводки статистики. Нужно прописывать абсолютный, а не относительный путь, иначе `TensorBoard` может не найти журналы. Если мы успешно запустили `TensorBoard`, этот инструмент

* Аккуратность — одна из мер оценки качества работы нейронной сети (и других алгоритмов машинного обучения), показывающая, какая доля экземпляров данных была правильно классифицирована. *Прим. науч. ред.*

будет предоставлять доступ к данным через `http://localhost:6006/` — этот адрес можно открыть в браузере.

Как показано на рис. 3.5, первая вкладка содержит информацию о скалярных сводках, которые мы собирали. Как вы видите, потери и на мини-пакетах, и на проверочном множестве уменьшаются.

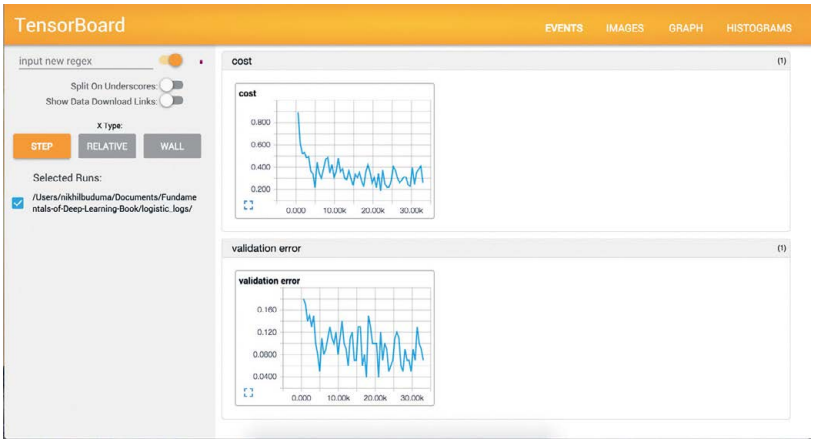


Рис. 3.5. *Представление событий в TensorBoard*

На рис. 3.6 показана другая вкладка, которая позволяет визуализировать весь построенный граф вычислений. Интерпретировать его не очень просто, но если мы столкнемся с непредсказуемым поведением, то представление графа будет полезным при отладке.

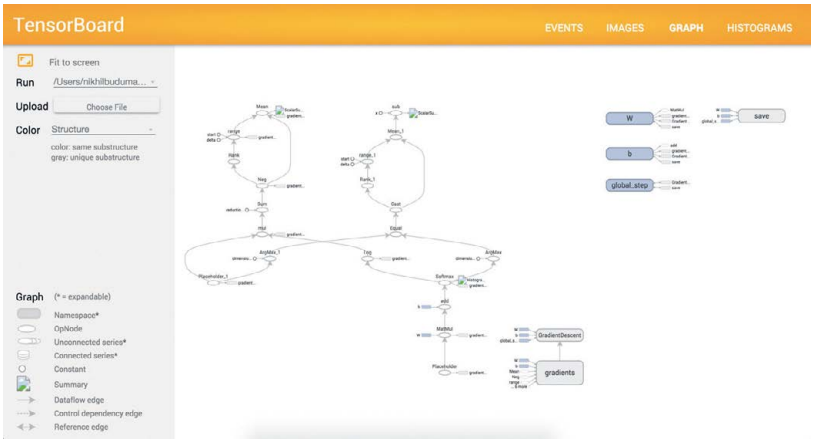


Рис. 3.6. *Графическое представление в TensorBoard*

Создание многослойной модели для MNIST в TensorFlow

Используя модель логистической регрессии, мы сократили частоту ошибок в наборе данных `MNIST` до 8,1%. Показатель впечатляющий, но для практического применения он не особо подходит.

Например, если система используется для проверки чеков на четырехзначные суммы (от 1000 до 9999 долларов), ошибки будут допускаться почти в 30% случаев! Чтобы создать более точную программу для чтения цифр из `MNIST`, построим нейросеть с прямым распространением сигнала.

Мы создаем такую модель с двумя скрытыми слоями, каждый из которых состоит из 256 нейронов `ReLU` (рис. 3.7).

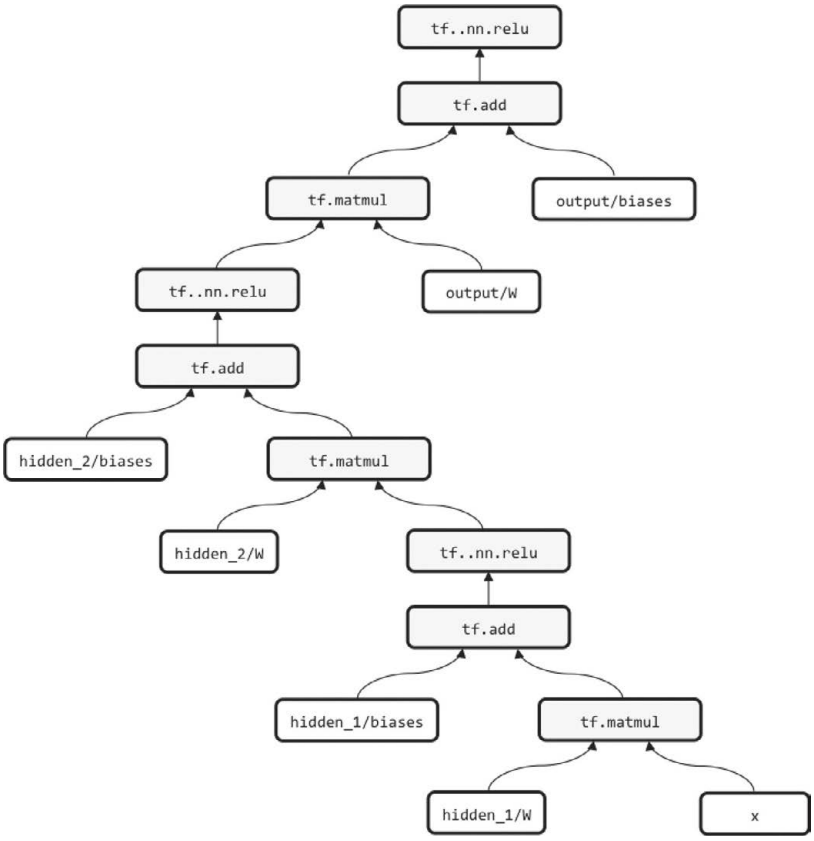


Рис. 3.7. *Сеть с прямым распространением сигнала на нейронах ReLU с двумя скрытыми слоями*

Мы можем использовать бóльшую часть кода из примера для логистической регрессии, внеся всего несколько изменений:

```
def layer(input, weight_shape, bias_shape):
       weight_stddev = (2.0/weight_shape[0])**0.5
       w_init = tf.random_normal_initializer(stddev=weight_stddev)
       bias_init = tf.constant_initializer(value=0)
       W = tf.get_variable("W", weight_shape,
                           initializer=w_init)
       b = tf.get_variable("b", bias_shape,
                           initializer=bias_init)
       return tf.nn.relu(tf.matmul(input, W) + b)
def inference(x):
       with tf.variable_scope("hidden_1"):
            hidden_1 = layer(x, [784, 256], [256])
       with tf.variable_scope("hidden_2"):
            hidden_2 = layer(hidden_1, [256, 256], [256])
       with tf.variable_scope("output"):
            output = layer(hidden_2, [256, 10], [10])
       return output
```

Новый код по большей части говорит сам за себя, но стратегия инициализации заслуживает дополнительного описания. Качество работы глубоких нейросетей во многом зависит от эффективности инициализации их параметров. Как мы расскажем в следующей главе, у поверхностей ошибок таких сетей много свойств, значительно усложняющих оптимизацию с помощью стохастического градиентного спуска.

Проблема усугубляется при росте числа слоев в модели, а следовательно, и сложности поверхности ошибок. Один из способов ее устранения — умная инициализация. Исследование 2015 года, опубликованное К. Хе и его коллегами, показывает, что для нейронов ReLU дисперсия весов в сети должна быть равна $2/n_{in}$, где n_{in} — число входов в нейрон[36]. Любопытному читателю стоит рассмотреть, что произойдет при изменении инициализации. Например, если снова заменить `tf.random_normal_initializer` на `tf.random_uniform_initializer`, который мы использовали в примере с логистической регрессией, результаты серьезно ухудшатся.

Наконец, чтобы еще немного улучшить качество работы, мы вычисляем функцию мягкого максимума при вычислении ошибки, а не на стадии предсказания. Отсюда новая модификация:

```
def loss(output, y):
        xentropy = tf.nn.softmax_cross_entropy_with_logits(output, y)
        loss = tf.reduce_mean(xentropy)
        return loss
```

Работа программы на протяжении 300 эпох выдает значительные улучшения по сравнению с моделью логистической регрессии. Она функционирует с аккуратностью 98,2%, почти на 78% снижая частоту ошибок на знак по сравнению с первым вариантом.

Резюме

В этой главе мы больше узнали о том, как использовать `TensorFlow` в качестве библиотеки для представления и обучения наших моделей. Мы поговорили о ряде ее важных свойств, в том числе управлении сессиями, переменными, операциями, графами вычислений и устройствами. В последних разделах мы на основе полученных знаний обучили и визуализировали модель логистической регрессии и нейросеть с прямым распространением сигнала при помощи стохастического градиентного спуска. И если модель логистической сети совершала много ошибок на наборе данных `MNIST`, то нейросеть с прямым распространением сигнала гораздо эффективнее: в среднем всего 1,8 ошибки на 100 символов. Мы улучшим этот показатель в главе 5.

В следующей главе мы начнем работу со множеством проблем, которые возникают, когда мы делаем наши нейросети глубже. Мы уже говорили о первом элементе пазла — нахождении умных способов инициализации параметров нашей сети. Скоро вы узнаете, что, когда модели усложняются, правильной инициализации недостаточно для хороших результатов. Чтобы преодолеть эти трудности, мы углубимся в современную теорию оптимизации и создадим более совершенные алгоритмы обучения глубоких сетей.

ГЛАВА 4

Не только градиентный спуск

Проблемы с градиентным спуском

Фундаментальные идеи в области нейросетей существуют уже десятилетия, но лишь в последнее время основанные на них модели обучения стали популярными. Наш интерес к нейросетям во многом вызван их выразительностью, которая обеспечивается многослойностью. Как мы уже говорили, глубокие нейросети способны решать проблемы, к которым раньше было невозможно даже подступиться.

Однако полное их обучение сопряжено с разными сложностями, которые требуют множества технологических инноваций, в том числе больших размеченных массивов данных (`ImageNet`, `CIFAR` и т. д.), более передового «железа» с ускорителями `GPU`, а также новинок в области алгоритмов.

Многие годы исследователи прибегали к поуровневому «жадному» предварительному обучению для обработки сложных поверхностей ошибок в моделях глубокого обучения[37]. Эти стратегии требовали больших затрат времени и были направлены на поиск более точных вариантов инициализации параметров модели по слою за раз перед тем, как использовать мини-пакетный градиентный спуск для поиска оптимальных параметров. Но недавние прорывы в методах оптимизации позволяют нам непосредственно обучать модели от начала и до конца. В этой главе речь пойдет именно о них.

Несколько следующих разделов будут в основном посвящены локальным минимумам и тому, как они препятствуют успешному обучению глубоких моделей. Далее мы поговорим о невыпуклых поверхностях ошибок, порожденных глубокими моделями, о том, почему обычный мини-пакетный градиентный спуск часто недостаточен и как современные невыпуклые оптимизаторы преодолевают эти трудности.

Локальные минимумы на поверхности ошибок глубоких сетей

Основные трудности при оптимизации моделей глубокого обучения связаны с тем, что мы вынуждены использовать информацию о локальных минимумах для выводов о глобальной структуре поверхности ошибок. Это серьезная проблема, ведь между локальной и глобальной структурами обычно мало связи. Рассмотрим такую аналогию.

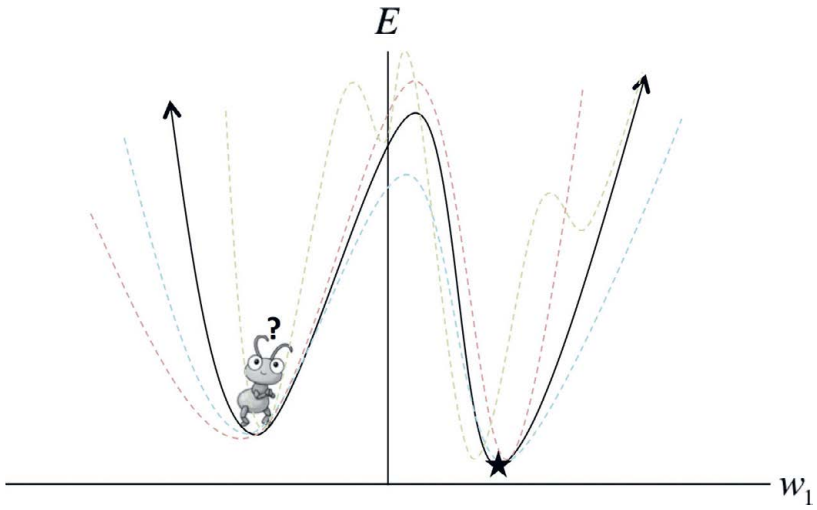


Рис. 4.1. *Мини-пакетный градиентный спуск помогает избежать мелкого локального минимума, но редко эффективен при наличии глубокого локального минимума*

Представьте себе, что вы — муравей, живущий в континентальной части США. Вас выбросили где-то в случайном месте, и ваша задача — найти самую низкую точку на этой поверхности. Как это сделать? Если вы можете видеть только то, что вас непосредственно окружает, задача кажется неразрешимой. Если бы поверхность США имела форму миски (была бы, говоря математически, выпуклой) и мы смогли бы удачно установить темп обучения, можно было бы воспользоваться алгоритмом градиентного спуска и в конце концов добраться до дна. Но рельеф США очень сложный. И даже если мы найдем какую-то долину (локальный минимум), мы не узнаем, действительно ли это самая низкая точка на карте (глобальный минимум). В главе 2 мы говорили о том, как мини-пакетный градиентный спуск помогает в продвижении по сложной поверхности ошибок, на которой есть

проблемные районы с нулевым градиентом. Но, как видно из рис. 4.1, даже стохастическая поверхность ошибок не спасает от глубокого локального минимума.

И тут встает важный вопрос. Теоретически локальные минимумы — серьезная проблема. Но как часто они встречаются на поверхности ошибок глубоких сетей на практике? И при каких сценариях они действительно затрудняют обучение? В двух следующих разделах мы рассмотрим распространенные заблуждения относительно локальных минимумов.

Определимость модели

Первый источник локальных минимумов связан с *определимостью модели*. Поверхности ошибок глубоких нейросетей гарантированно имеют значительное — иногда бесконечное — число локальных минимумов. И вот почему.

Внутри слоя полносвязной нейросети с прямым распространением сигнала любая перестановка нейронов не изменит данные на выходе. Проиллюстрируем это при помощи простого слоя из трех нейронов на рис. 4.2. Оказывается, что в слое из n нейронов существует $n!$ способов перестановки параметров. А для глубокой сети с l слоев, каждый из которых состоит из n нейронов, имеется $n!^l$ эквивалентных конфигураций.

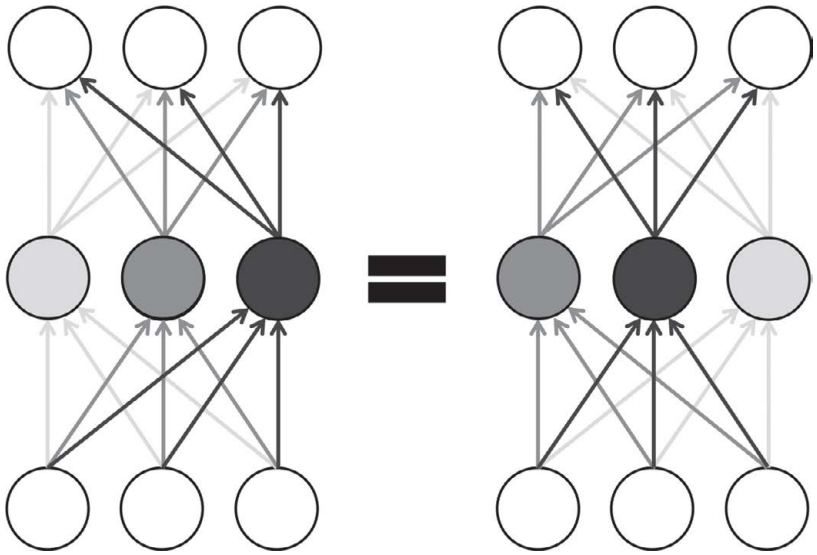


Рис. 4.2. *Перестройка нейронов в слое нейросети приводит к эквивалентным конфигурациям в силу симметрии*

Помимо симметрии перестроек нейронов, неопределимость присутствует в некоторых видах нейросетей и в других формах. Например, существует бесконечное число эквивалентных конфигураций, которые приводят к эквивалентным сетям для отдельного нейрона `ReLU`. Поскольку он использует кусочно-линейную функцию, мы можем умножить все веса входов на любую не равную 0 константу k, при этом умножая все веса выходов на $1/k$ без изменения поведения сети.

Пусть активные читатели сами обоснуют это утверждение. В целом локальные минимумы из-за неопределимости глубоких нейросетей по природе своей не создают проблем. Ведь все неопределимые конфигурации ведут себя примерно одинаково независимо от того, какие входные значения в них поступают. Они дадут одну ошибку на обучающем, проверочном и тестовом наборах данных. Все они достигнут одинаковых успехов на обучающих данных и будут вести себя идентично при обобщении до неизвестных примеров.

Локальные минимумы становятся проблемой, только если они *сомнительные*. Тогда они соответствуют конфигурации весов в нейросети, которая вызывает ошибку больше, чем конфигурация в глобальном минимуме. Если локальные минимумы встречаются часто, мы вскоре столкнемся с серьезными проблемами при использовании градиентных методов оптимизации, поскольку учитывать можем только локальную структуру.

Насколько неприятны сомнительные локальные минимумы в нейросетях?

Много лет специалисты во всех проблемах при обучении глубоких сетей винили сомнительные локальные минимумы, даже не имея достаточных доказательств. Сейчас открытым остается вопрос, действительно ли такие минимумы с высокой частотой ошибок по сравнению с глобальными часто встречаются в реальных глубоких сетях. Однако, судя по последним исследованиям, у большинства локальных минимумов частота ошибок и характеристики обобщения не очень отличаются от глобальных минимумов.

Можно попробовать решить эту проблему наивным путем: построить график функции потерь во время обучения глубокой нейросети. Но эта стратегия не даст достаточно информации о поверхности ошибок, ведь трудно судить о том, действительно ли она так «ухабиста» или мы никак не можем понять, куда двигаться.

Для более эффективного анализа проблемы И. Гудфеллоу и его коллеги (команда исследователей, объединившая специалистов из Google и Стэнфордского университета) опубликовали в 2014 году статью[38], в которой попытались разделить два этих фактора, способных вызвать затруднение. Вместо анализа функции ошибок во времени они исследовали, что происходит на поверхности ошибок между случайным образом инициализированным вектором параметров и успешным конечным решением, с помощью линейной интерполяции.

Итак, при случайным образом инициализированном векторе параметров θ_i и решении стохастического градиентного спуска θ_f мы намерены вычислить функцию ошибок в каждой точке методом линейной интерполяции $\theta_\alpha = \alpha \cdot \theta_f + (1 - \alpha) \cdot \theta_i$.

Ученые хотели понять, будут ли локальные минимумы помехой градиентному методу поиска даже в случае, если мы будем знать, куда двигаться. Оказалось, что для большого количества реальных сетей с разными типами нейронов прямой путь между случайным образом инициализированной точкой в пространстве параметров и решением стохастического градиентного спуска не осложняется сколь-нибудь проблемными локальными минимумами.

Мы можем даже показать это сами на примере сети с прямым распространением сигнала из нейронов `ReLU`, созданной в главе 3. Запустив контрольный файл, который мы сохранили при обучении исходной сети, мы можем пересоздать компоненты `inference` и `loss`, сохраняя список указателей на переменные в оригинальном графе для дальнейшего использования в `var_list_opt` (где `opt` — оптимальные настройки параметров):

```
# mnist data image of shape 28*28=784
x = tf.placeholder("float", [None, 784])
# 0-9 digits recognition => 10 classes
y = tf.placeholder("float", [None, 10])
sess = tf.Session()
with tf.variable_scope("mlp_model") as scope:
       output_opt = inference(x)
       cost_opt = loss(output_opt, y)
       saver = tf.train.Saver()
       scope.reuse_variables()
       var_list_opt = [
```

```
            "hidden_1/W",
            "hidden_1/b",
            "hidden_2/W",
            "hidden_2/b",
            "output/W",
            "output/b"
        ]
        var_list_opt = [tf.get_variable(v) for v in var_list_opt]
        saver.restore(sess, "mlp_logs/model-checkpoint-file")
```

Так же мы можем повторно использовать конструкторы компонентов для создания случайным образом инициализированной сети. Вот как мы сохраняем переменные в `var_list_rand` для следующего шага программы:

```
with tf.variable_scope("mlp_init") as scope:
        output_rand = inference(x)
        cost_rand = loss(output_rand, y)
        scope.reuse_variables()
        var_list_rand = [
            "hidden_1/W",
            "hidden_1/b",
            "hidden_2/W",
            "hidden_2/b",
            "output/W",
            "output/b"
        ]
        var_list_rand = [tf.get_variable(v) for v in var_list_rand]
        init_op = tf.initialize_variables(var_list_rand)
        sess.run(init_op)
```

Инициализировав две эти сети, мы можем вывести линейную интерполяцию при помощи параметров `alpha` и `beta`:

```
with tf.variable_scope("mlp_inter") as scope:
    alpha = tf.placeholder("float", [1, 1])
    beta = 1 - alpha
    h1_W_inter = var_list_opt[0] * beta + var_list_rand[0] * alpha
    h1_b_inter = var_list_opt[1] * beta + var_list_rand[1] * alpha
```

```
h2_W_inter = var_list_opt[2] * beta + var_list_rand[2] * alpha
h2_b_inter = var_list_opt[3] * beta + var_list_rand[3] * alpha
o_W_inter = var_list_opt[4] * beta + var_list_rand[4] * alpha
o_b_inter = var_list_opt[5] * beta + var_list_rand[5] * alpha
h1_inter = tf.nn.relu(tf.matmul(x, h1_W_inter) + h1_b_inter)
h2_inter = tf.nn.relu(tf.matmul(h1_inter, h2_W_inter) + h2_b_inter)
o_inter = tf.nn.relu(tf.matmul(h2_inter, o_W_inter) + o_b_inter)
cost_inter = loss(o_inter, y)
```

Наконец, мы можем варьировать значение `alpha`, чтобы определить, как меняется поверхность ошибок при переходе вдоль линии от случайным образом инициализированной точки к конечному решению стохастическим градиентным спуском:

```
import matplotlib.pyplot as plt
summary_writer = tf.train.SummaryWriter("linear_interp_logs/",
                                        graph_def=sess.graph_def)
summary_op = tf.merge_all_summaries()
results = []
for a in np.arange(-2, 2, 0.01):
      feed_dict = {
            x: mnist.test.images,
            y: mnist.test.labels,
            alpha: [[a]],
      }
      cost, summary_str = sess.run([cost_inter, summary_op],
                                        feed_dict=feed_dict)
      summary_writer.add_summary(summary_str, (a + 2)/0.01)
      results.append(cost)
plt.plot(np.arange(-2, 2, 0.01), results, 'ro')
plt.ylabel('Incurred Error')
plt.xlabel('Alpha')
plt.show()
```

Получаем рис. 4.3, который мы можем проанализировать сами. Если мы будем повторять этот эксперимент, окажется, что никаких локальных минимумов, которые всерьез поставили бы нас в тупик, и нет. Судя по всему, основная сложность с градиентным спуском — не они, а трудности

с выбором нужного направления. К этому вопросу мы вернемся чуть позже.

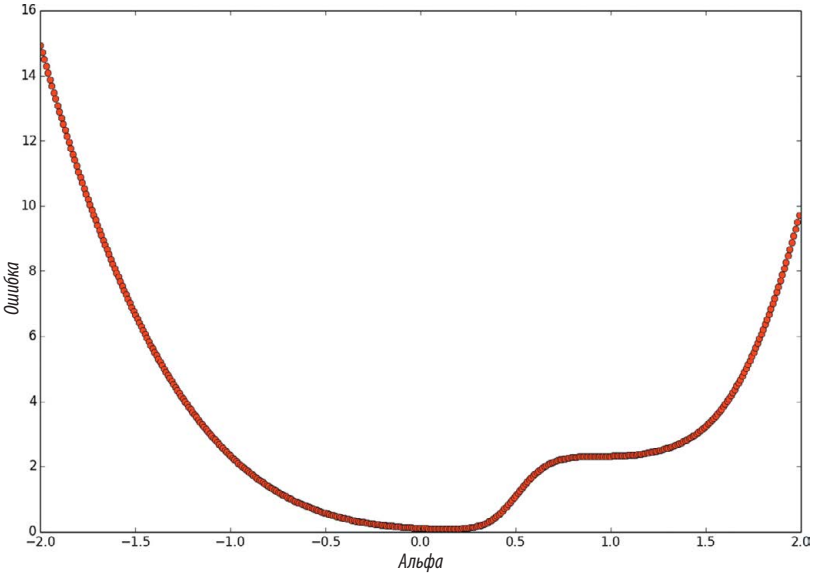


Рис. 4.3. *Функция потерь трехслойной нейросети с прямым распространением сигнала при линейной интерполяции на линию, соединяющую случайным образом инициализированный вектор параметров и решение стохастического градиентного спуска*

Плоские области на поверхности ошибок

Хотя наш анализ и показал отсутствие проблем с локальными минимумами, можно отметить особенно плоскую область, в которой градиент доходит до нуля, а `alpha = 1`. Это не локальный минимум, и вряд ли она поставит нас в затруднение, но кажется, что нулевой градиент может замедлить обучение.

В общем случае для произвольной функции точка, на которой градиент становится нулевым вектором, называется *критической*. Критические точки бывают разных типов. Мы уже говорили о локальных минимумах, несложно представить себе и их противоположности: *локальные максимумы*, которые, впрочем, не вызывают особых проблем при стохастическом градиентном спуске. Но есть и странные точки, лежащие где-то посередине. Эти «плоские» области — потенциально неприятные, но не обязательно

смертельные — называются *седловыми точками*. Оказывается, когда у нашей функции появляется все больше измерений (а у модели все больше параметров), седловые точки становятся экспоненциально более вероятными, чем локальные минимумы. Попробуем понять, почему так происходит.

Для одномерной функции ошибок критическая точка может принимать одну из трех форм, как показано на рис. 3.1. Представим себе, например, что все эти три конфигурации одинаково вероятны. Если взять случайную критическую точку случайной одномерной функции, то вероятность того, что это локальный минимум, будет равна 1/3. И если у нас есть k критических точек, то можно ожидать, что локальными минимумами окажутся $k/3$ из них.

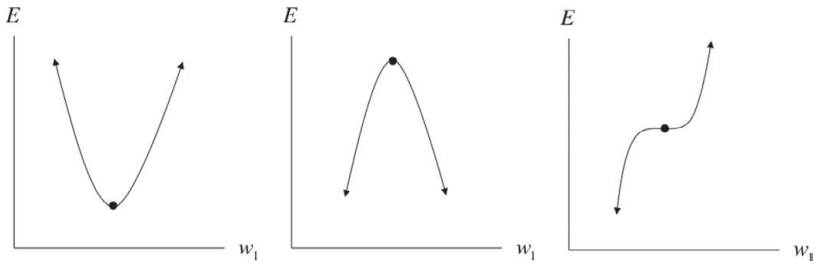


Рис. 4.4. *Анализ критической точки в одном измерении*

Можно распространить это утверждение на функции с большим числом измерений. Представим функцию потерь в d-мерном пространстве. Возьмем произвольную критическую точку. Оказывается, определить, будет ли она локальным минимумом, локальным максимумом или седловой точкой, сложнее, чем для одномерного случая. Рассмотрим поверхность ошибок на рис. 4.5. В зависимости от того, как выбирать направление (от A к B или от C к D), критическая точка будет казаться либо минимумом, либо максимумом. В реальности она ни то ни другое. Это более сложный тип седловой точки.

В целом в d-мерной области параметров мы можем провести через критическую точку d различные оси. Она может быть локальным минимумом, только если оказывается локальным минимумом в каждой из d одномерных подобластей. Зная, что критическая точка в одномерной подобласти может относиться к одному из трех типов, мы понимаем: вероятность того, что случайная критическая точка принадлежит случайной функции, равна $\frac{1}{3^d}$. Это значит, что случайная функция с k критических точек имеет ожидаемое количество $\frac{k}{3^d}$ локальных минимумов. С ростом размерности области

параметров число локальных минимумов экспоненциально уменьшается. Более строгое рассмотрение этой темы выходит за рамки настоящей книги; ознакомиться с вопросом можно в работе Яна Дофина и его коллег (2014)[39].

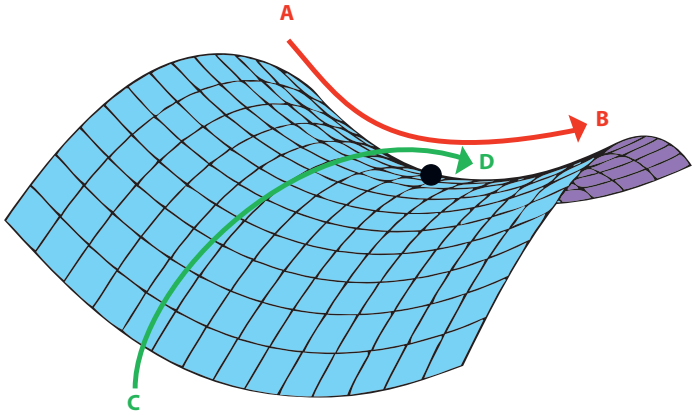


Рис. 4.5. *Седловая точка на двумерной поверхности ошибок*

Что все это значит в контексте оптимизации моделей глубокого обучения? Для стохастического градиентного спуска — пока непонятно. Судя по всему, плоские области на поверхности ошибок неприятны, но не мешают получить приемлемый ответ с использованием этого метода. Но они создают серьезные проблемы для методов, которые пытаются найти прямое решение в точке, где градиент равен 0. Именно это обстоятельство снижает пользу некоторых методов оптимизации второго порядка для моделей глубокого обучения, о чем мы поговорим ниже.

Когда градиент указывает в неверном направлении

При анализе поверхностей ошибок глубоких сетей, судя по всему, самое важное в оптимизации — поиск верной траектории. Неудивительно, что чаще всего эта проблема встает при рассмотрении поверхности ошибок вокруг локального минимума. В качестве примера рассмотрим поверхность ошибки, определенную двумерной областью параметров, на рис. 4.6.

Вновь обратившись к контурным диаграммам, о которых шла речь в главе 2, мы отмечаем, что градиент — обычно не лучший индикатор правильной траектории. Он указывает в направлении локального минимума, только если контуры идеально круглые. Если же они эллиптические (как обычно

и бывает с поверхностями ошибок глубоких сетей), градиент может указывать на 90° в сторону от верного направления!

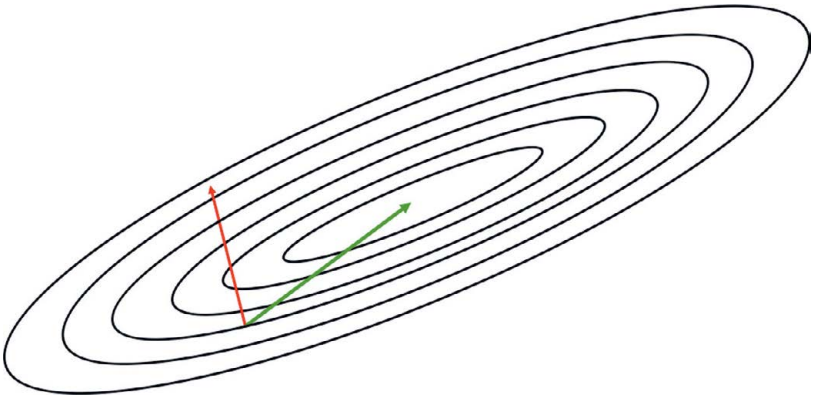


Рис. 4.6. *Локальная информация, представленная градиентом, обычно не соответствует глобальной структуре поверхности ошибок*

Распространим этот анализ на произвольное число измерений, введя несколько математических обозначений. Для каждого веса w_i в области параметров градиент вычисляет значение $\frac{\partial E}{\partial w_i}$, или то, как меняется значение ошибки при вариациях значения w_i. Определив и объединив все веса в области параметров, градиент дает направление кратчайшего спуска. Однако, прежде чем сделать уверенный шаг в этом направлении, нужно вспомнить о проблеме: градиент может меняться у нас на глазах в процессе движения! Этот простой факт показан на рис. 4.7. Возвращаясь к двумерному примеру, скажем: если наши контуры идеально круглые и мы делаем значительный шаг в направлении наискорейшего спуска, градиент больше направления не меняет. Но для сильно эллиптических контуров это неверно.

В общем случае мы можем определить изменения градиента во время спуска в определенном направлении, вычислив вторую производную. Мы подсчитываем

$$\frac{\partial\left(\partial E / \partial w_j\right)}{\partial w_i},$$

что дает возможность определить, как градиентный компонент w_j модифицируется с изменением значения w_i. Эту информацию мы можем свести в особую матрицу, известную как матрица Гессе, или гессиан (H). При описании поверхности ошибок, когда градиент меняется в ходе пути в направлении кратчайшего спуска, она называется *плохо обусловленной*.

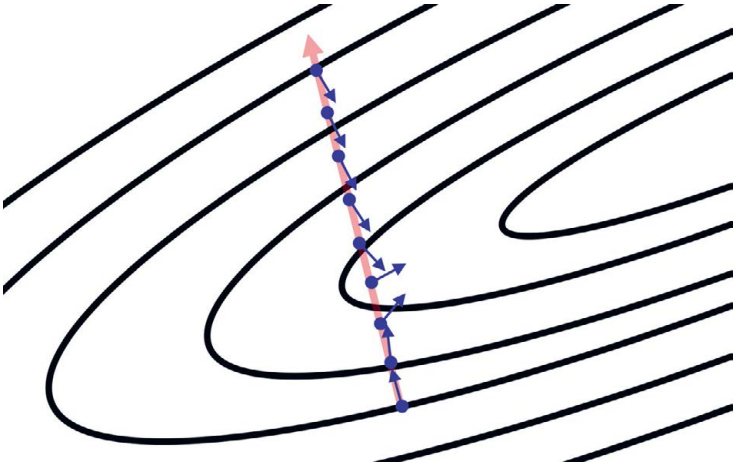


Рис. 4.7. *Мы показываем, как направление градиента меняется, когда мы следуем в направлении скорейшего спуска (определенного от начальной точки). Векторы нормализованы до идентичной длины, чтобы подчеркнуть смену их направления*

Для математически подкованных читателей подробнее расскажем о том, как гессиан ограничивает оптимизацию только с помощью градиентного спуска. Определенные свойства матрицы Гессе (реальность и симметричность) позволяют успешно определить вторую производную (которая аппроксимирует кривизну поверхности) при движении в определенном направлении. А именно: если есть единичный вектор $\boldsymbol{d}$, вторая производная в этом направлении задается $\boldsymbol{d}^{\mathrm{T}}\boldsymbol{H}\boldsymbol{d}$. Теперь можно использовать аппроксимацию второго порядка с помощью ряда Тейлора и понять, что происходит с функцией ошибок при переходе от текущего вектора параметров $\mathbf{x}^{(i)}$ к новому $\mathbf{x}$ по вектору градиента $\mathbf{g}^{\star}$, выраженному в $x^{(i)}$:

$$E(x) \approx E\left(x^{(i)}\right) + \left(x - x^{(i)}\right)^{\mathrm{T}} g + \frac{1}{2}\left(x - x^{(i)}\right)^{\mathrm{T}} H\left(x - x^{(i)}\right).$$

Если мы будем и далее двигаться в направлении градиента, можно упростить наше выражение:

$$E\left(x^{(i)} - \epsilon\, g\right) \approx E\left(x^{(i)}\right) - \epsilon\, g^{\mathrm{T}} g + \frac{1}{2}\epsilon^{2}\, g^{\mathrm{T}} H g.$$

Оно состоит из трех членов: 1) значение функции потерь для исходного вектора параметров; 2) улучшение функции потерь, обеспеченное величиной

* Более строго, мы движемся в направлении, противоположном градиенту, так как градиент указывает направления наиболее быстрого возрастания функции, а нам нужно направление убывания. *Прим. науч. ред.*

градиента; 3) корректирующий член, который выражает кривизну пространства, представленную матрицей Гессе.

Стоит воспользоваться этой информацией для разработки более эффективных алгоритмов оптимизации. Например, мы можем взять приближение второго порядка функции ошибок для определения скорости обучения на каждом этапе, больше всего снижающей функцию ошибки. Но оказывается, что точно вычислить матрицу Гессе — задача сложная. В следующих разделах мы рассмотрим открытия в оптимизации, которые позволяют справиться с плохой обусловленностью без непосредственного вычисления матрицы Гессе.

Импульсная оптимизация

Проблема плохо обусловленной матрицы Гессе проявляется в серьезных флуктуациях градиентов. Один из популярных вариантов решения — отказ от вычисления гессиана и сосредоточение на борьбе с этими флуктуациями во время обучения. Один из способов понять, как такой подход может быть полезен, — изучить, как мяч катится по холмистой поверхности. Под действием силы тяжести он в итоге оказывается в низшей точке, но почему-то не страдает от серьезных флуктуаций и отклонений, которые случаются при градиентном спуске. В чем же дело? В отличие от стохастического градиентного спуска (при котором используется только градиент), движение мяча по поверхности ошибки определяется еще двумя основными компонентами. Первый, который мы моделируем в виде градиента при стохастическом спуске, в обиходе называется ускорением. Но не только оно определяет движение мяча. Гораздо важнее его скорость. Ускорение влияет именно на нее и лишь опосредованно — на положение мяча.

Движение под действием скорости желательно, поскольку оно противодействует значительным флуктуациям градиента, постепенно спрямляя траекторию мяча. Скорость — форма памяти, позволяющая эффективно аккумулировать движение к минимуму, сглаживая колебания ускорения в ортогональных направлениях. Наша задача — создать аналог скорости в алгоритме оптимизации. Это можно сделать, отслеживая *экспоненциально взвешенное затухание* предыдущих градиентов. Каждое обновление вычисляется с помощью сочетания обновления предыдущей итерации с текущим градиентом. Изменения вектора параметров определяются так:

$$\nu_i = m\boldsymbol{\nu}_{i-1} - \epsilon g_i,$$
$$\theta_i = \theta_{i-1} + \boldsymbol{\nu}_i.$$

Иными словами, мы вводим гиперпараметр импульса m для определения того, какую долю предыдущей скорости нужно сохранить при обновлении, и добавляем «память» предыдущих градиентов к текущему. Этот метод обычно именуется *импульсным*. Поскольку введение импульса увеличивает размер шага, его использование порой требует сокращения скорости обучения по сравнению с обычным стохастическим градиентным спуском.

Чтобы нагляднее представить себе, как работает импульс, рассмотрим пример, как он влияет на обновления в ходе *случайных блужданий*, то есть совершения последовательности бессистемно выбранных шагов. Представим себе точку на линии, которая в каждый интервал времени случайно выбирает размер шага между –10 и 10 и перемещается в этом направлении. Это выражается просто:

```
step_range = 10
step_choices = range(-1 * step_range,step_range + 1)
rand_walk = [random.choice(step_choices) for x in xrange(100)]
```

Затем имитируем то, что произойдет при небольшой модификации импульса (с помощью стандартного алгоритма экспоненциально скользящего среднего), чтобы сгладить выбор шага на каждом интервале. Это можно кратко выразить так:

```
momentum_rand_walk = [random.choice(step_choices)]
for i in xrange(len(rand_walk) — 1):
       prev = momentum_rand_walk[-1]
       rand_choice = random.choice(step_choices)
       new_step = momentum * prev + (1 — momentum) * rand_choice
       momentum_rand_walk.append()
```

Результаты при варьировании импульса от 0 до 1 удивительны. Импульс существенно снижает волатильность обновлений. Чем он больше, тем мы менее чувствительны к новым обновлениям (например, серьезная неточность при первой оценке траектории сохраняется в течение значительного периода). Результаты эксперимента приведены на рис. 4.8.

Чтобы понять, как импульс влияет на обучение реальных нейросетей с прямым распространением сигнала, мы можем вернуться к `MNIST` и применить импульсный оптимизатор `TensorFlow`. Будем использовать тот же темп обучения (0,01) с типичным импульсом 0,9:

```
learning_rate = 0.01
momentum = 0.9
optimizer = tf.train.MomentumOptimizer(learning_rate, momentum)
train_op = optimizer.minimize(cost, global_step=global_step)
```

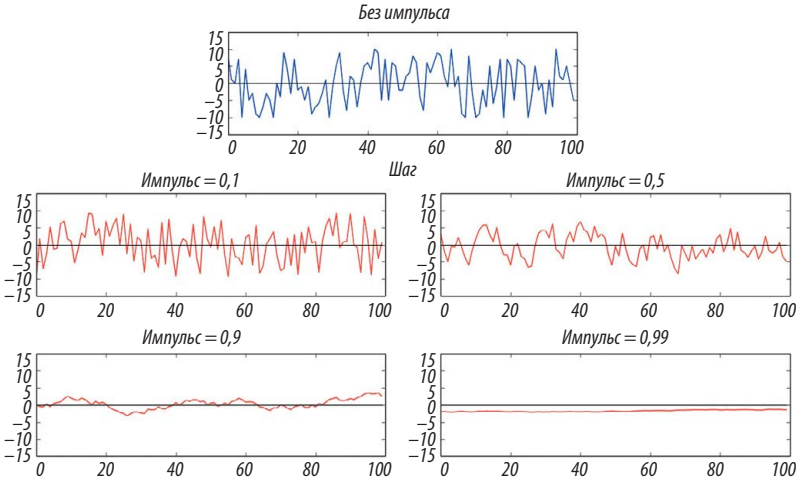


Рис. 4.8. *Импульс снижает волатильность в размерах шагов при случайных блужданиях благодаря экспоненциально скользящему среднему*

Ускорение вычислений поразительно. Изменение функции потерь со временем видно на примере сравнения визуализаций `TensorBoard` на рис. 4.9. Здесь показано, что для достижения потерь в 0,1 без импульса (слева) требуется почти 18 000 шагов (мини-пакетов), а с импульсом (справа) — чуть более 2000.

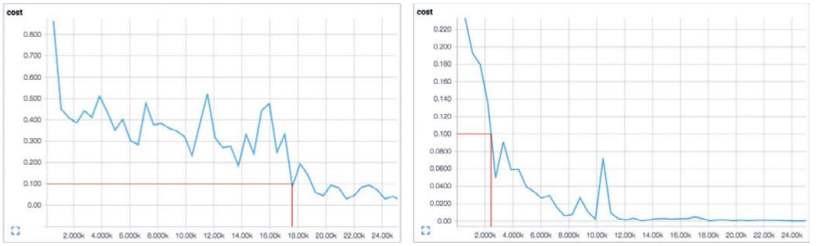


Рис. 4.9. *Сравнение для сети с прямым распространением сигнала с импульсом (справа) и без импульса (слева) показывает значительное сокращение времени обучения*

В последнее время появилось много исследований, направленных на улучшение классического импульсного метода. В работе Ильи Суцкевера и его коллег (2013) предложена альтернатива: импульсный метод Нестерова, который вычисляет градиент на поверхности ошибок во время обновления скорости при $\theta + \mathbf{v}_{i-1}$, а не θ[40]. Эта тонкая разница позволяет более

эффективно изменять скорость. Было доказано, что этот метод имеет явные преимущества при пакетном градиентном спуске (гарантирует сходимость и может использовать большее значение импульса для заданного темпа обучения по сравнению с классическим). Но не до конца понятно, есть ли выгоды при стохастическом мини-пакетном градиентном спуске, который используется в большинстве подходов оптимизации для глубокого обучения. На момент написания этой книги импульсный метод Нестерова в `TensorFlow` не поддерживался*.

Краткий обзор методов второго порядка

Как мы уже говорили в предыдущих разделах, вычисление гессиана — сложная задача, а импульс позволяет добиться значительного ускорения вычислений без его участия. Но за последние несколько лет разработано несколько методов второго порядка, которые направлены на аппроксимацию гессиана. Для более полной картины дадим краткий обзор этих методов, а их подробное описание выходит за рамки настоящей книги.

Один из них — метод сопряженных градиентов, который вырос из попыток улучшить наивный метод кратчайшего спуска. При кратчайшем спуске мы вычисляем направление градиента и проводим линейный поиск для нахождения минимума по нему. Мы переходим к минимуму и снова вычисляем градиент, чтобы определить направление следующего линейного поиска.

Этот метод приводит к разнообразным зигзагам (рис. 4.10), ведь каждый раз, когда мы движемся в сторону кратчайшего спуска, мы немного откатываемся в другом направлении. Решение — движение в *сопряженном направлении* относительно предыдущего, а не к кратчайшему спуску. Направление выбирается методом косвенного аппроксимирования гессиана для линейного сочетания градиента и предыдущего направления. При небольших модификациях метод обобщается до невыпуклых поверхностей ошибок, характерных для глубоких сетей[41].

Альтернативный алгоритм оптимизации называется алгоритмом *Бройдена — Флетчера — Гольдфарба — Шанно (`BFGS`)*[42] и заключается в итеративном вычислении обратной матрицы Гессе для более эффективной оптимизации вектора параметров. Изначально `BFGS` предъявлял значительные требования к памяти, но уже разработана более эффективная версия — *`L-BFGS`*[43].

---

* Сейчас импульсный метод Нестерова уже реализован в TensorFlow: https://www.tensorflow.org/api_docs/python/tf/train/MomentumOptimizer. *Прим. науч. ред.*

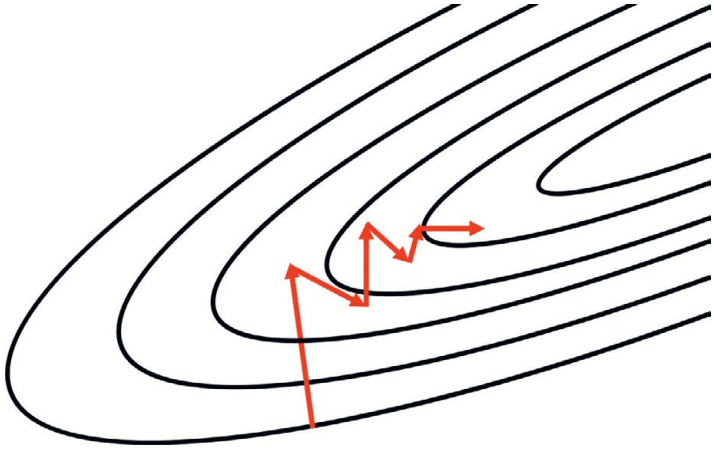


Рис. 4.10. *Метод кратчайшего спуска часто дает зигзаги; сопряженный спуск направлен на решение этой проблемы*

Эти подходы перспективны, но методы второго порядка по-прежнему остаются областью активных исследований и у практиков непопулярны. `TensorFlow` на момент написания этой книги не поддерживал ни метода сопряженных градиентов, ни `L-BFGS`.

Адаптация темпа обучения

Еще одна серьезная проблема при обучении глубоких сетей — выбор правильного темпа. Эта задача уже давно считается одним из самых проблематичных аспектов обучения глубоких сетей, поскольку темп серьезно влияет на эффективность. Слишком низкий не позволит обучаться быстро, а слишком высокий может привести к проблемам со сходимостью при достижении локального минимума или плохо обусловленного участка.

Один из главных новых прорывов в области оптимизации глубоких сетей — возможность адаптации темпа обучения. Смысл в том, что темп модифицируется в процессе для достижения хорошей сходимости. В следующих разделах мы рассмотрим три самых популярных алгоритма адаптации темпа обучения: `AdaGrad`, `RMSProp` и `Adam`.

ADAGRAD — СУММИРОВАНИЕ ИСТОРИЧЕСКИХ ГРАДИЕНТОВ

Первым мы рассмотрим алгоритм `AdaGrad`, который стремится адаптировать общий темп обучения путем суммирования исторических градиентов.

Впервые алгоритм был предложен в работе Джона Дучи и его коллег (2011)[44]. Мы сохраняем изменение темпа обучения для каждого параметра. Темп обратно масштабируется с учетом величины квадратного корня суммы квадратов (среднеквадратичного значения) всех исторических градиентов параметра. Можно выразить это математически. Мы инициализируем вектор суммирования градиента $\boldsymbol{r}_0 = \mathbf{0}$. На каждом шаге мы суммируем квадраты градиентов всех параметров следующим образом (где операция $\odot$ — поэлементное умножение тензоров):

$$\boldsymbol{r}_i = \boldsymbol{r}_{i-1} + \boldsymbol{g} \odot \boldsymbol{g}_i.$$

Затем мы обычным путем вычисляем обновление, но теперь глобальный темп обучения делится на квадратный корень вектора сумм градиента:

$$\theta_i = \theta_{i-1} - \frac{\in}{\delta \oplus \sqrt{\boldsymbol{r}_i}} \odot \boldsymbol{g}.$$

Заметьте, что мы добавляем в знаменателе небольшую величину δ (~10–7), чтобы избежать деления на 0. Кроме того, операции деления и сложения связаны с размером вектора суммирования градиента и применяются поэлементно. Встроенный в `TensorFlow` оптимизатор позволяет легко использовать `AdaGrad` в качестве алгоритма обучения:

```
tf.train.AdagradOptimizer(learning_rate,
      initial_accumulator_value=0.1,
      use_locking=False,
      name='Adagrad')
```

Нужно только помнить, что в `TensorFlow` величина δ и исходный вектор суммирования градиента объединены в аргумент `initial_accumulator_value`.

На функциональном уровне такой механизм обновления означает, что параметры с наибольшими градиентами будут быстро снижать темп обучения, а с меньшими — незначительно. Можно констатировать, что `AdaGrad` обеспечивает больший прогресс на более пологих участках поверхности ошибок, помогая преодолеть плохо обусловленные поверхности. Теоретически это обеспечивает хорошие свойства, но на практике обучение глубоких моделей с помощью `AdaGrad` сопряжено с проблемами. По опыту можно сказать, что у него есть тенденция к преждевременному уменьшению темпа обучения, и он едва ли будет хорошо работать в некоторых глубоких моделях. Поговорим о `RMSProp` — алгоритме, призванном устранить этот недостаток.

RMSPROP — ЭКСПОНЕНЦИАЛЬНО ВЗВЕШЕННОЕ СКОЛЬЗЯЩЕЕ СРЕДНЕЕ ГРАДИЕНТОВ

`AdaGrad` хорошо работает с простыми выпуклыми функциями, но этот алгоритм не предназначен для сложных поверхностей ошибок глубоких сетей. Плоские области могут поставить `AdaGrad` в тупик, заставив снизить темп обучения еще до достижения минимума. Получается, простого сложения градиентов недостаточно.

Решением может стать идея, которую мы вводили при использовании импульса для снижения флуктуаций градиента. По сравнению с простым сложением экспоненциально взвешенные скользящие средние позволяют «отбрасывать» измерения, произведенные задолго до текущего момента. Обновление вектора суммирования градиента в этом случае выглядит так:

$$\boldsymbol{r}_i = \rho \boldsymbol{r}_{i-1} + (1-\rho)\boldsymbol{g}_i \odot \boldsymbol{g}_i.$$

Коэффициент затухания ρ определяет, как долго мы будем хранить старые градиенты. Чем он ниже, тем меньше рабочее окно. Внеся такую модификацию в `AdaGrad`, мы получим алгоритм обучения `RMSProp`, впервые предложенный Джеффри Хинтоном[45].

В `TensorFlow` создать оптимизатор `RMSProp` можно с помощью следующего кода. Отметим, что в этом случае, в отличие от `AdaGrad`, мы вводим δ отдельно как эпсилон-аргумент `конструктора`:

```
tf.train.RMSPropOptimizer(learning_rate, decay=0.9,
      momentum=0.0, epsilon=1e-10,
      use_locking=False, name='RMSProp')
```

Как предполагает этот шаблон, можно использовать `RMSProp` в сочетании с импульсом (особенно нестеровским). В целом он показал себя эффективным средством оптимизации глубоких нейросетей и стал выбором по умолчанию для многих опытных практиков.

ADAM — СОЧЕТАНИЕ ИМПУЛЬСНОГО МЕТОДА С RMSPROP

Прежде чем завершить разговор о современных средствах оптимизации, рассмотрим еще один алгоритм — `Adam`[46]. В принципе, можно считать его вариантом сочетания импульсного метода и `RMSProp`.

Основная идея такова. Мы хотим записывать экспоненциально взвешенное скользящее среднее градиента (скорость в классическом импульсном подходе), что можно выразить следующим образом:

$$\mathbf{m}_i = \beta_1 \mathbf{m}_{i-1} + (1 - \beta_1)\mathbf{g}_i.$$

Это аппроксимация так называемого *первого импульса* градиента, или $\mathbf{E}[g_i]$. Как и в `RMSProp`, можно сохранять экспоненциально взвешенное скользящее среднее исторических градиентов. Это оценка того, что называется *вторым импульсом* градиента, или $\mathbf{E}[\mathbf{g}_i \odot \mathbf{g}_i]$:

$$\mathbf{v}_i = \beta_2 \mathbf{v}_{i-1} + (1 - \beta_2)\mathbf{g}_i \odot \mathbf{g}_i.$$

Но оказывается, что эти приближения не соответствуют реальным импульсам, поскольку мы инициализируем оба вектора нулями. Чтобы устранить несоответствие, мы выводим поправочный коэффициент для обоих случаев. Здесь мы описываем вывод определения для второго импульса. Вывод первого импульса аналогичен, и мы оставим его в качестве упражнения для любителей математики.

Начнем с выражения определения второго импульса через все предыдущие градиенты. Для этого достаточно расширить рекуррентное соотношение:

$$\begin{gathered}
\boldsymbol{v}_i = \beta_2 \boldsymbol{v}_{i-1} + (1-\beta_2)\boldsymbol{g}_i \odot \boldsymbol{g}_i \\
\boldsymbol{v}_i = \beta_2^{i-1}(1-\beta_2)\boldsymbol{g}_1 \odot \boldsymbol{g}_1 + \beta_2^{i-2}(1-\beta_2)\boldsymbol{g}_2 \odot \boldsymbol{g}_2 + \ldots + (1-\beta_2)\boldsymbol{g}_i \odot \boldsymbol{g}_i \\
\boldsymbol{v}_i = (1-\beta_2)\sum_{k=1}^{i} \beta^{i-k} \boldsymbol{g}_k \odot \boldsymbol{g}_k
\end{gathered}$$

Возьмем ожидаемое значение обеих частей и определим, как наша оценка $\mathbf{E}[\mathbf{v}_i]$ соотносится с истинным значением $\mathbf{E}[\mathbf{g}_i \odot \mathbf{g}_i]$:

$$\mathbf{E}[\mathbf{v}_i] = \mathbf{E}\left[(1-\beta_2)\sum_{k=1}^{i} \beta^{i-k} \boldsymbol{g}_k \odot \boldsymbol{g}_k\right]$$

Можно принять, что $\mathbf{E}[\mathbf{g}_k \odot \mathbf{g}_k] \approx \mathbf{E}[\mathbf{g}_i \approx \mathbf{g}_i]$, поскольку, даже если второй импульс градиента поменял значение по сравнению с историческим, β_2 должно быть выбрано так, чтобы прежние вторые импульсы градиентов утратили релевантность. В результате становится возможным такое упрощение:

$$\begin{gathered}
\mathbf{E}[\mathbf{v}_i] \approx \mathbf{E}[\mathbf{g}_i \odot \mathbf{g}_i](1-\beta_2)\sum_{k=1}^{i} \beta^{i-k} \\
\mathbf{E}[\mathbf{v}_i] \approx \mathbf{E}[\mathbf{g}_i \odot \mathbf{g}_i](1-\beta_2^{\,i}).
\end{gathered}$$

Последнее упрощение сделано с помощью элементарного алгебраического тождества $1 - x^n = (1 - x)\ (1 + x + \ldots + x^{n-1})$. В результате вывода

второго импульса и аналогичного ему вывода первого мы получаем следующие поправочные схемы для борьбы с ошибкой инициализации:

$$\tilde{\mathbf{m}}_i = \frac{m_i}{1-\beta_1^i},$$

$$\tilde{\mathbf{v}}_i = \frac{\tilde{\mathbf{v}}_i}{1-\beta_2^i}.$$

Теперь можно с помощью скорректированных импульсов обновить вектор параметров, что приведет к окончательному обновлению Adam:

$$\theta_i = \theta_{i-1} - \frac{\in}{\delta \otimes \sqrt{\tilde{\mathbf{v}}_i}} \tilde{\mathbf{m}}_i.$$

В последнее время Adam набрал популярность благодаря возможности исправлять ошибку нулевой инициализации (в RMSProp это слабое место) и способности эффективно сочетать ключевые идеи RMSProp и импульсного метода. В TensorFlow оптимизатор Adam создается с помощью следующего конструктора:

```
tf.train.AdamOptimizer(learning_rate=0.001, beta1=0.9,
      beta2=0.999, epsilon=1e-08,
      use_locking=False, name='Adam')
```

Настройки по умолчанию гиперпараметров для Adam в TensorFlow обычно работают удовлетворительно, но Adam хорошо справляется и с изменениями в них. Единственное исключение: иногда может понадобиться изменить темп обучения (по умолчанию задана величина 0,001).

Философия при выборе метода оптимизации

В этой главе мы обсудили ряд стратегий для упрощения навигации по сложным поверхностям ошибок глубоких сетей. Они нашли свое высшее выражение в нескольких алгоритмах оптимизации, каждый из которых имеет свои достоинства и недостатки. Было бы замечательно заранее знать, когда какой алгоритм использовать, но практикующие эксперты пока не достигли здесь консенсуса. Сейчас самые популярные варианты — мини-пакетный градиентный спуск, мини-пакетный градиент в сочетании с импульсом, RMSProp, RMSProp в сочетании с импульсом, Adam и AdaDelta (об этом алгоритме мы здесь не говорили, а TensorFlow его пока не поддерживает). Мы включили в репозитории этой книги на Github скрипт TensorFlow, чтобы любознательный читатель мог поэкспериментировать

с алгоритмами оптимизации на примере построенной модели сети с прямым распространением сигнала:

```
$ python optimzer_mlp.py <sgd, momentum, adagrad, rmsprop,
      adam>
```

Важно отметить, однако, что для большинства практиков глубокого обучения лучший способ достичь вершин в своем деле состоит не в создании более совершенных оптимизаторов. Большинство прорывов в этой области за последние несколько десятилетий связаны с нахождением архитектур, которые проще обучать, а не попытками разобраться с неприятными поверхностями ошибок. Далее мы сосредоточимся на использовании архитектур для более эффективного обучения нейросетей.

Резюме

В этой главе мы поговорили о ряде проблем при попытках обучать глубокие сети со сложными поверхностями ошибок. Мы рассказали, что проблемы сомнительных локальных минимумов, возможно, преувеличены, а седловые точки и плохая обусловленность действительно могут серьезно мешать успеху обычного мини-пакетного градиентного спуска. Мы описали, как использовать импульс для борьбы с плохой обусловленностью, и дали краткий обзор последних исследований методов второго порядка, направленных на аппроксимацию матрицы Гессе. Мы рассказали об эволюции алгоритмов оптимизации с адаптацией темпа обучения, которые настраивают темп в процессе для улучшения сходимости.

В следующей главе мы начнем разговор о более широкой проблеме архитектуры и проектирования сетей. Начнем мы с анализа компьютерного зрения и того, как создавать глубокие сети, эффективно обучающиеся на сложных изображениях.

ГЛАВА 5

Сверточные нейросети

Нейроны и зрение человека

Человеческое зрение развито очень хорошо. За доли секунды мы можем распознавать видимые объекты без особых умственных усилий или задержек. Мы не только способны назвать то, на что смотрим, но и ощутить его глубину, четко определить контуры и отделить от фона. Каким-то образом наши глаза получают необработанные воксели* информации о цвете, но в мозге они перерабатываются в более значимые единицы — линии, кривые и формы, которые могут, например, подсказать, что мы смотрим на домашнюю кошку[47].

Человеческое зрение основано на нейронах. Последние отвечают за доставку световой информации в глаз[48]. Эта информация проходит здесь предварительную обработку, препровождается в зрительную кору мозга и только там анализируется. За все эти функции и отвечают нейроны. Поэтому интуитивно кажется, что стоило бы обобщить наши модели нейронных сетей для создания более эффективных систем компьютерного зрения. В этой главе мы воспользуемся нашим пониманием человеческого зрения для создания эффективных моделей глубокого обучения анализу изображений. Но для начала рассмотрим более традиционные подходы к вопросу и отметим их недостатки.

Недостатки выбора признаков

Начнем с простой проблемы компьютерного зрения. Я дам вам случайно выбранное изображение — например, с рис. 5.1. Ваша задача — ответить, есть ли здесь человеческое лицо. Именно эту проблему затрагивали Пол Виола и Майкл Джонс в своей основополагающей работе в 2001 году[49].

---

* Воксель (voxel) — элемент трехмерного изображения. Название образовано по аналогии с «пиксел» (picture element, элемент изображения), от англ. volume element — объемный элемент. *Прим. науч. ред.*

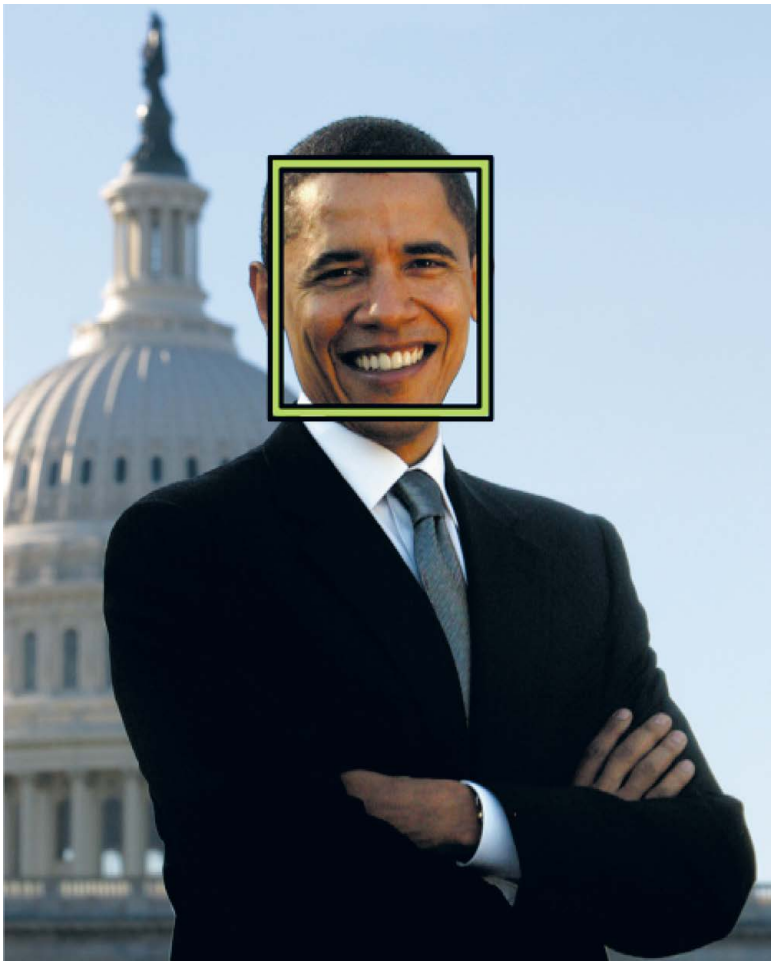


Рис. 5.1. *Гипотетический алгоритм распознавания лиц должен найти на этой фотографии лицо экс-президента Барака Обамы*

Для людей это тривиальная задача. А вот для компьютера она очень сложна. Как научить его тому, что на изображении есть лицо? Можно использовать традиционный алгоритм машинного обучения (вроде того, что был описан в главе 1), подавая на вход значения пикселов в надежде на то, что найдется подходящий классификатор. Но работает это плохо, поскольку отношение сигнала к шуму слишком низкое, чтобы удалось хоть чему-то обучиться. Нужна альтернатива.

Со временем появился компромиссный вариант — по сути, нечто среднее между традиционной компьютерной программой, логика которой

определяется человеком, и чистым подходом машинного обучения, где все бремя ложится на компьютер. Человек выбирает признаки (возможно, сотни или тысячи), которые, по его мнению, важны для принятия решений по классификации. Так он создает представление той же проблемы, но меньшей размерности. Алгоритм машинного обучения использует новые *векторы признаков* для принятия решений по классификации. Поскольку процесс извлечения признаков улучшает соотношение сигнала и шума (если выделены действительно важные аспекты), этот подход имел большой успех по сравнению с имевшимися на тот момент. Виола и Джонс отметили, что лица характеризуются определенным соотношением светлых и темных пятен, которое и можно здесь использовать. Например, есть разница в интенсивности света между областью глаз и скулами, переносицей и глазами по обе стороны от нее. Эти показатели приведены на рис. 5.2.

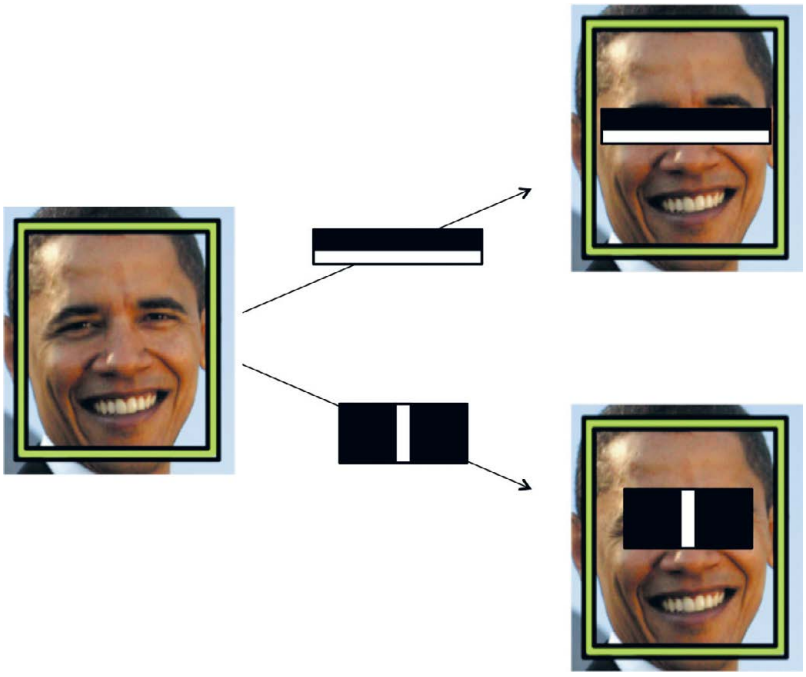


Рис. 5.2. *Показатели интенсивности Виолы — Джонса*

Каждый из этих признаков сам по себе не особо эффективен при идентификации лица. Но при совместном использовании (в рамках классического алгоритма машинного обучения — бустинга, описанного в оригинальной работе (`http://bit.ly/2qMguNT`)) их общая эффективность

значительно возрастает. На наборе данных из 130 изображений и 507 лиц алгоритм достигает эффективности 91,4% при 50 ложных срабатываниях. Тогда такие показатели были беспрецедентными, но у этого алгоритма есть фундаментальные ограничения. Если лицо частично затенено, сравнение интенсивности света не срабатывает. Более того, если алгоритм «смотрит» на морщинистое лицо или лицо персонажа мультфильма, неудача почти неизбежна.

Алгоритм на самом деле так и не научился «смотреть». Помимо различий в интенсивности света, наш мозг использует обширный спектр визуальных подсказок для определения того, что мы видим человеческое лицо. Это контуры, относительное расположение черт и цвет. И даже если одна из визуальных подсказок содержит расхождения (например, части лица скрыты или затенены), зрительная кора все равно с высокой надежностью способна вычленять лица.

Чтобы с помощью традиционных методов научить компьютер «видеть», нужно дать программе гораздо больше признаков. Только так можно добиться более точных решений. До эры глубокого обучения огромные коллективы исследователей компьютерного зрения многие годы спорили о пользе разных признаков. Поскольку задачи распознавания становились всё сложнее, ученым часто приходилось тяжело.

Чтобы проиллюстрировать возможности глубокого обучения, рассмотрим соревнование `ImageNet` — одно из самых престижных в области компьютерного зрения (иногда его даже называют Олимпийскими играми компьютерного зрения)[50]. Каждый год исследователи пытаются разбить изображения по 200 возможным классам на основе обучающего набора данных примерно из 450 тысяч образцов. Алгоритму дается пять попыток на ответ, прежде чем он переходит к следующему изображению из тестового набора. Цель — довести компьютерное зрение до уровня человеческого (примерно 95–96% точности). В 2011 году победитель соревнования `ImageNet` показал частоту ошибок 25,7%, в среднем одно изображение из четырех[51]. Конечно, это большой шаг по сравнению с гаданием, но для коммерческого применения такой результат не подходит.

В 2012 году Алекс Крижевский из лаборатории Джеффри Хинтона в Университете Торонто совершил невозможное. Он впервые применил архитектуру глубокого обучения, ныне известную как *сверточная нейросеть*, к проблемам особого масштаба и сложности, не оставив соперникам шансов. Второе место занял участник с похвальным результатом 26,1%. А вот `AlexNet` всего за несколько месяцев работы побила все рекорды 50-летних исследований в области традиционного компьютерного зрения с частотой

появления ошибок примерно 16%[52]. Не будет преувеличением сказать, что она единолично ввела глубокое обучение в работу над компьютерным зрением. Произошла настоящая революция в этой области.

Обычные глубокие нейросети не масштабируются

Фундаментальная цель применения глубокого обучения к компьютерному зрению — отказ от трудоемкого и имеющего значительные ограничения процесса выбора признаков. Как мы уже говорили в главе 1, глубокие нейросети идеальны для такого процесса, поскольку каждый их слой отвечает за усвоение и создание признаков, представляющих получаемые входные данные. Наивный подход мог бы состоять в использовании обычной глубокой нейросети вроде той, что мы разработали в главе 3, и применении ее к набору данных `MNIST` для решения проблемы классификации изображений. Но при таком методе мы вскоре столкнемся с очень неприятными трудностями, которые показаны на рис. 5.3.

В `MNIST` наши изображения имели масштаб всего 28×28 пикселов и к тому же были черно-белыми. В результате у нейрона в полносвязном скрытом слое было всего 784 входящих веса. Это нормально для работы с `MNIST`, и наша обычная нейросеть функционировала хорошо. Однако этот метод не масштабируется на более объемные изображения. Например, для полноцветного изображения 200×200 пикселов входной слой будет иметь 200×200×3 = 120 000 весов. А нам нужно будет разместить множество нейронов по разным слоям, и число параметров будет расти очень быстро! Полная связность не только окажется излишней, но и обусловит значительно большие шансы на переобучение.

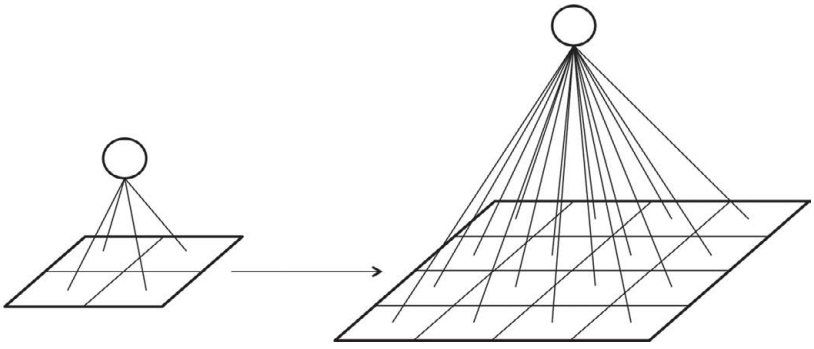


Рис. 5.3. *Плотность связей между слоями неприемлемо возрастает с увеличением размера изображения*

Сверточная сеть использует преимущество того, что мы анализируем именно изображения, и разумно ограничивает архитектуру глубокой сети. Поэтому мы можем значительно сократить число параметров в нашей модели. Слои сверточной сети, основанные на работе человеческого зрения, организованы в трех измерениях: у каждого слоя есть ширина, высота и глубина, как показано на рис. 5.4[53]. Как мы позже увидим, нейроны в сверточном слое соединены только с небольшой локальной областью предыдущего слоя. Это позволяет избежать избыточности полносвязных нейронов. Функцию сверточного слоя можно выразить просто: он обрабатывает трехмерный объем информации, порождая новый трехмерный объем информации. Подробнее о том, как это работает, мы поговорим в следующем разделе.

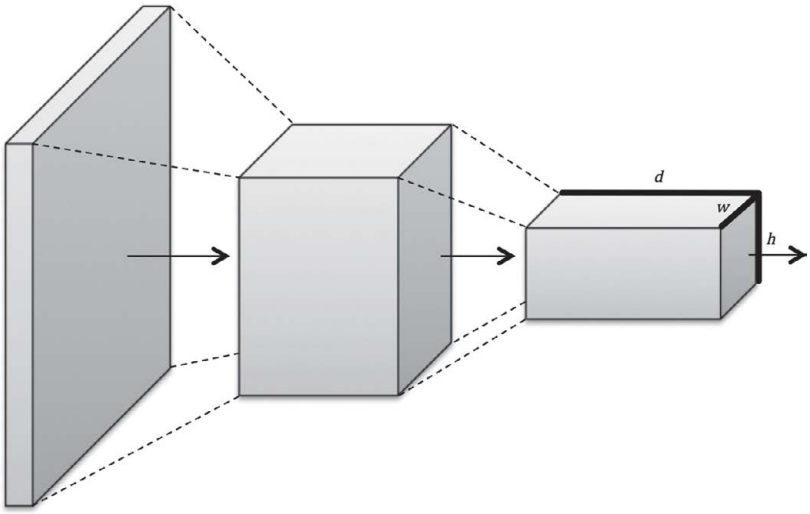


Рис. 5.4. *В сверточных слоях нейроны организуются в трех измерениях, эти слои имеют ширину, высоту и глубину*

Фильтры и карты признаков

Чтобы объяснить базовые элементы сверточного слоя, рассмотрим, как человеческий мозг сводит воедино необработанную визуальную информацию и понимает окружающий мир. Одно из самых знаковых исследований в этой области было проведено Дэвидом Хьюбелом и Торстеном Визелем. Они обнаружили, что зоны зрительной коры отвечают за определение границ. В 1959 году они вставили электроды в мозг кошки и стали

проецировать на экраны черно-белые изображения. Оказалось, некоторые нейроны активируются только при появлении вертикальных линий, другие — только горизонтальных, а третьи — линий под определенными углами[54].

В дальнейших работах было выяснено, что зрительная кора организована в слои. Каждый отвечает за дополнение признаков, обнаруженных на предыдущих слоях, «прописывая» линии, контуры, формы и, наконец, объекты целиком. Более того, в слое визуальной коры одни и те же детекторы расположены повсюду, что позволяет определять признаки во всех частях изображения. Эти идеи во многом повлияли на разработку сверточных нейронных сетей.

Первой из этого нового знания зародилась идея *фильтра*, к которой, как оказалось, были довольно близки Виола и Джонс. По сути, это детектор признака, и его работу мы рассмотрим на упрощенном примере на рис. 5.5.

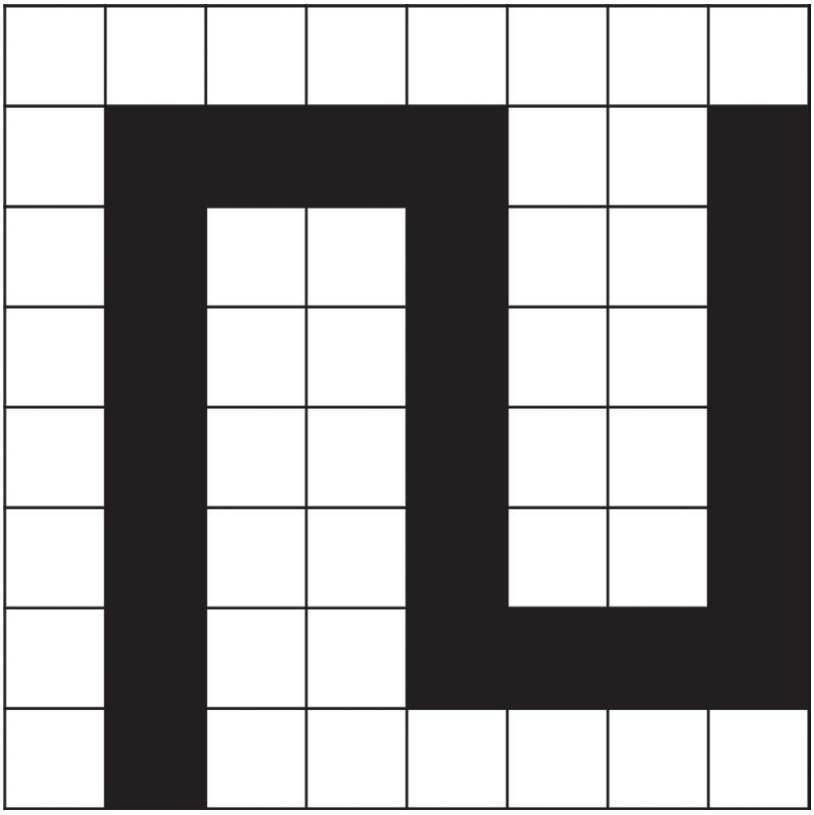


Рис. 5.5. *Пример простого черно-белого изображения*

Допустим, нам нужно определить вертикальные и горизонтальные линии на этом изображении. Один из вариантов решения — использовать подходящий детектор признаков, как на рис. 5.6. Например, чтобы определить вертикальные линии, воспользуемся детектором сверху, распространив его на все изображение, и на каждом шаге будем проверять совпадения. Запишем ответы в матрицу в правом верхнем углу. Если есть совпадение, мы красим соответствующую ячейку в черный цвет, если нет — оставляем ее белой. В результате получаем *карту признаков*, которая показывает, где мы нашли нужный признак в исходном изображении. То же можно сделать с детектором горизонтальных линий снизу — получится карта признаков, приведенная в правом нижнем углу.

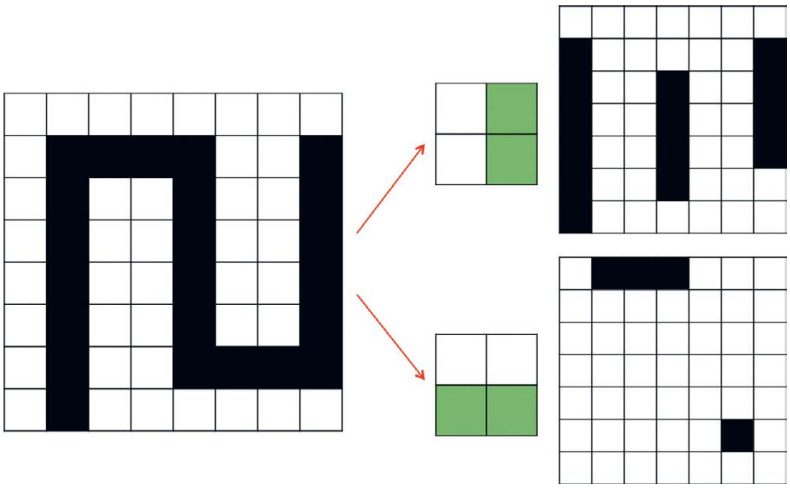


Рис. 5.6. *Применение фильтров для вертикальных и горизонтальных линий в примере*

Эта операция называется сверткой. Мы берем фильтр и распространяем его на всю область входящего изображения. Воспользуемся следующей схемой и постараемся выразить эту операцию в виде нейронов сети. Здесь слои в нейросети с прямым распространением сигналов представляют либо исходное изображение, либо карту признаков. Фильтры — сочетания связей (одно из них выделено на рис. 5.7), которые повторяются для всех входных данных. На рис. 5.7 связи одного цвета всегда будут иметь одинаковый вес. Добиться этого можно, инициализировав все связи в группе с идентичными весами и постоянно усредняя обновления весов группы, прежде чем применить их в конце каждой итерации обратного распространения ошибок. Выходной слой — карта признаков, созданная этим фильтром.

Нейрон на карте признаков активируется, если фильтр, отвечающий за эту операцию, обнаруживает подходящий признак в соответствующей позиции в предыдущем слое.

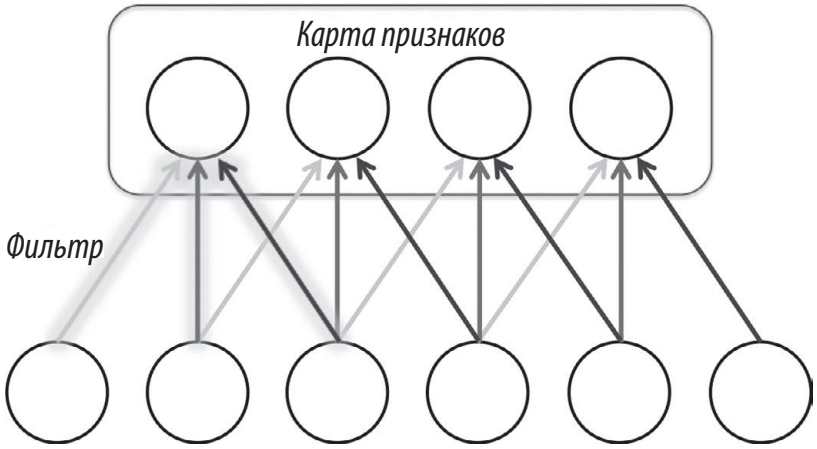


Рис. 5.7. *Выражение фильтров и карт признаков в виде нейронов в сверточной сети*

Обозначим k-ю карту признаков в слое m как m^k. Далее обозначим соответствующий фильтр по значениям его весов W. Предположим, что у нейронов на карте признаков может быть смещение b^k (оно идентично для всех нейронов на карте признаков), и теперь мы можем математически выразить карту признаков:

$$m_{ij}^k = f\left(\left(W * x\right)_{ij} + b^k\right).$$

Это математическое выражение, конечно, простое и сжатое, но не до конца описывает применение фильтров в сверточных нейронных сетях. Те работают не только на одной карте признаков. Они действуют по всему объему карт, созданных на конкретном слое. Рассмотрим, например, ситуацию, когда нам нужно определить лицо на конкретном уровне сверточной сети. Мы аккумулируем три карты признаков: одну для глаз, одну для носов, одну для ртов. Мы знаем, что на определенном участке изображения есть лицо, если соответствующие участки на базовых картах содержат подходящие признаки (два глаза, нос и рот). Иными словами, чтобы определить наличие лица на изображении, мы должны свести воедино свидетельства нескольких карт. Это столь же необходимо и для полноцветного

изображения. Пикселы в нем представлены в виде значений шкалы RGB, и во входном объеме нам потребуется три сектора (по одному для каждого цвета). Итак, карты признаков должны уметь работать на объемах, а не только плоскостях. Это показано на рис. 5.8. Каждая клетка входного объема — нейрон. Локальная область умножается на фильтр (соответствующий весам в сверточном слое), и образуется нейрон на карте фильтров в объемном слое нейронов.

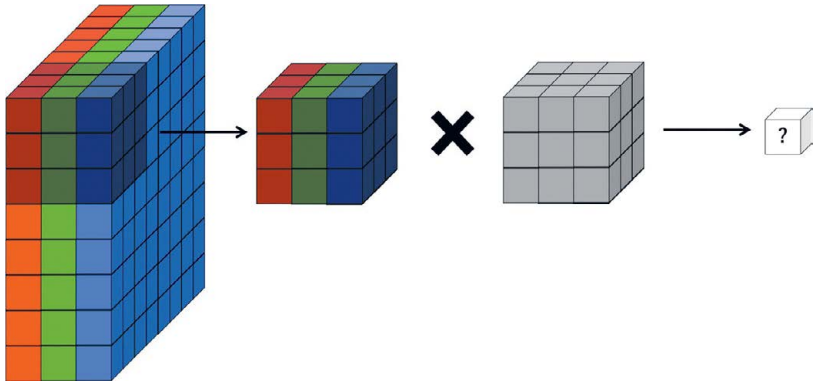


Рис. 5.8. *Представление полноцветного RGB-изображения в виде объема и наложение объемного сверточного фильтра*

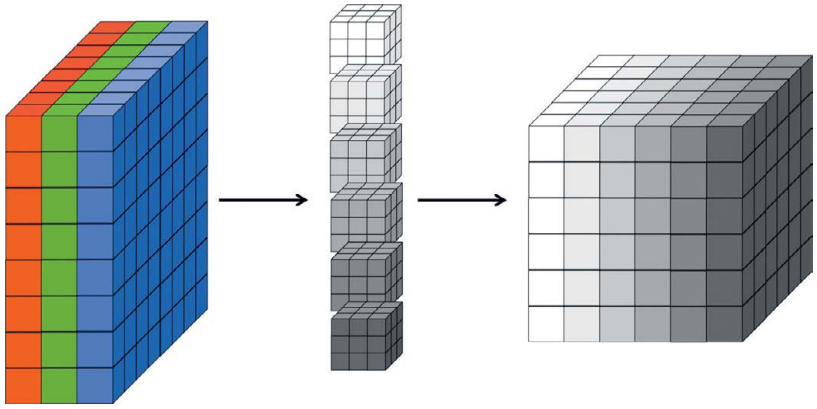


Рис. 5.9. *Трехмерная визуализация сверточного слоя, где каждый фильтр соответствует сектору в получившемся выходном объеме*

Как мы уже говорили в предыдущем разделе, сверточный слой (состоящий из набора фильтров) преобразует один набор значений в другой. Глубина

фильтра соответствует глубине входного объема. Фильтр может комбинировать информацию обо всех выученных признаках. Глубина исходящего объема сверточного слоя эквивалентна числу фильтров в нем, ведь каждый фильтр порождает свой сектор. Эти отношения показаны на рис. 5.9.

В следующем разделе мы применим эти концепции и заполним ряд пробелов, чтобы дать полное описание сверточного слоя.

Полное описание сверточного слоя

Воспользуемся выведенными понятиями и завершим описание сверточного слоя. Во-первых, он принимает следующие входящие характеристики.

— Ширина w_{in}.

— Высота h_{in}.

— Глубина d_{in}.

— Дополнение нулями p.

Входные значения обрабатываются фильтрами в количестве k, которые соответствуют весам и связям в сверточной сети. У фильтров есть ряд гиперпараметров, которые описываются так:

— *Пространственная протяженность* e, равная высоте и ширине фильтра.

— *Сдвиг* s, или дистанция между последовательными применениями фильтра к объему входных данных. Если использовать сдвиг 1, получится полная свертка, описанная в предыдущем разделе. Проиллюстрируем это на рис. 5.10.

— *Смещение* b (параметр, определяемый в процессе обучения, как и значения фильтра), добавляемый к каждому компоненту свертки.

Получаем выходные данные со следующими характеристиками.

— Функция активации f, применяемая ко входному логиту каждого нейрона в объеме входных данных для определения выходного значения.

— *Ширина* $w_{out} = \left[\frac{w_{in} - e + 2p}{s}\right] + 1.$

— *Высота* $h_{out} = \left[\frac{h_{in} - e + 2p}{s}\right] + 1.$

— *Глубина* $d_{out} = k.$

m-й «срез по глубине» объема выходных значений, где $1 \leq m \leq k$, соответствует функции активации f, примененной к сумме m-го фильтра, свернутого по объему входящих значений и смещения b^m. Более того, это значит, что на фильтр мы имеем $d_{in}e^2$ параметров. Всего в слое $kd_{in}e^2$ параметров и k смещений.

Чтобы продемонстрировать это в действии, мы приводим пример сверточного слоя на рис. 5.11 и 5.12 с объемом входных значений 5×5×3 и дополнением нулями $p = 1$. Возьмем два фильтра 3×3×3 (пространственная протяженность) со сдвигом $s = 2$. Мы применим линейную функцию и получим объем выходных данных размером 3×3×2.

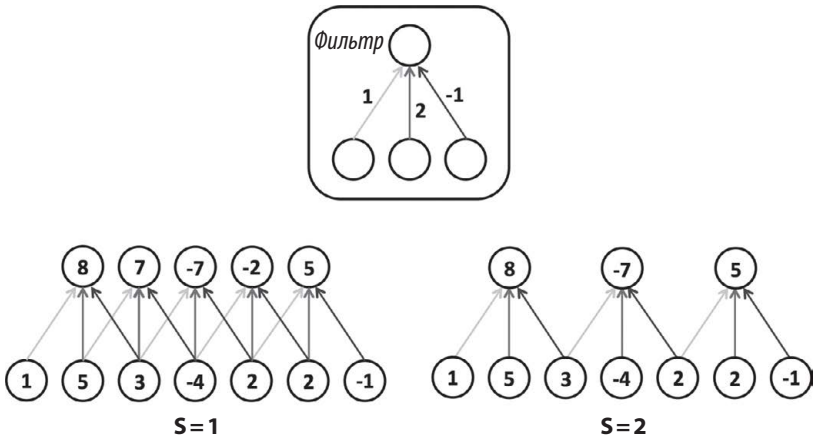


Рис. 5.10. *Иллюстрация гиперпараметра сдвига фильтра*

Разумно использовать фильтры небольших размеров (3×3 или 5×5). Реже применяются более крупные размеры (7×7), и то для первого сверточного слоя. Множество мелких фильтров повышает результативность и позволяет ограничиться меньшим числом параметров. Также предлагаем пользоваться сдвигом 1, чтобы ничего не упустить на карте признаков, и дополнение нулями, которое помогает достичь эквивалентности высоты и ширины объема входных и входных данных.

`TensorFlow` позволяет совершать удобные операции, легко сворачивая мини-пакет объемов входных данных (функцию активации f мы должны выбрать самостоятельно — операция не применит ее сама)[55]:

```
tf.nn.conv2d(input, filter, strides, padding,
use_cudnn_on_gpu=True,
       name=None)
```

Здесь `input` — четырехмерный тензор размера $N \times h_{in} \times w_{in} \times d_{in}$, где N — количество примеров в мини-пакете. Аргумент `filter` — тоже четырехмерный тензор, представляющий все фильтры, которые применены при свертке. Он имеет размер $e \times e \times d_{in} \times k$. Полученный при этой операции тензор имеет ту же структуру, что и `input`. Задав в `padding` значение `SAME`, мы выбираем дополнение нулями, так что высота и ширина сохраняются сверточным слоем.

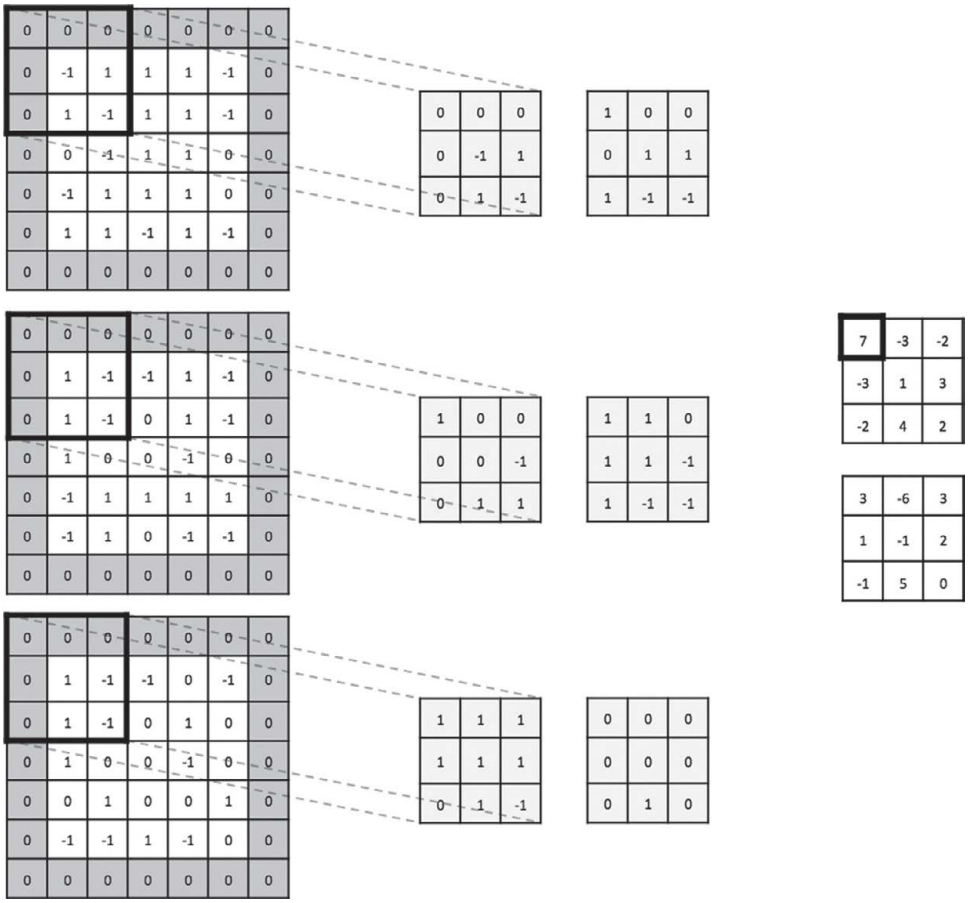


Рис. 5.11. *Сверточный слой с объемом входных значений шириной 5, высотой 5, глубиной 3 и дополнением нулями 1. Есть два фильтра пространственной протяженностью 3, применяемые со сдвигом 2. Получается объем выходных данных шириной 3, высотой 3 и глубиной 2. Первый сверточный фильтр мы применяем к верхнему левому сегменту объема входных данных 3×3 и получаем верхний левый элемент среза по глубине*

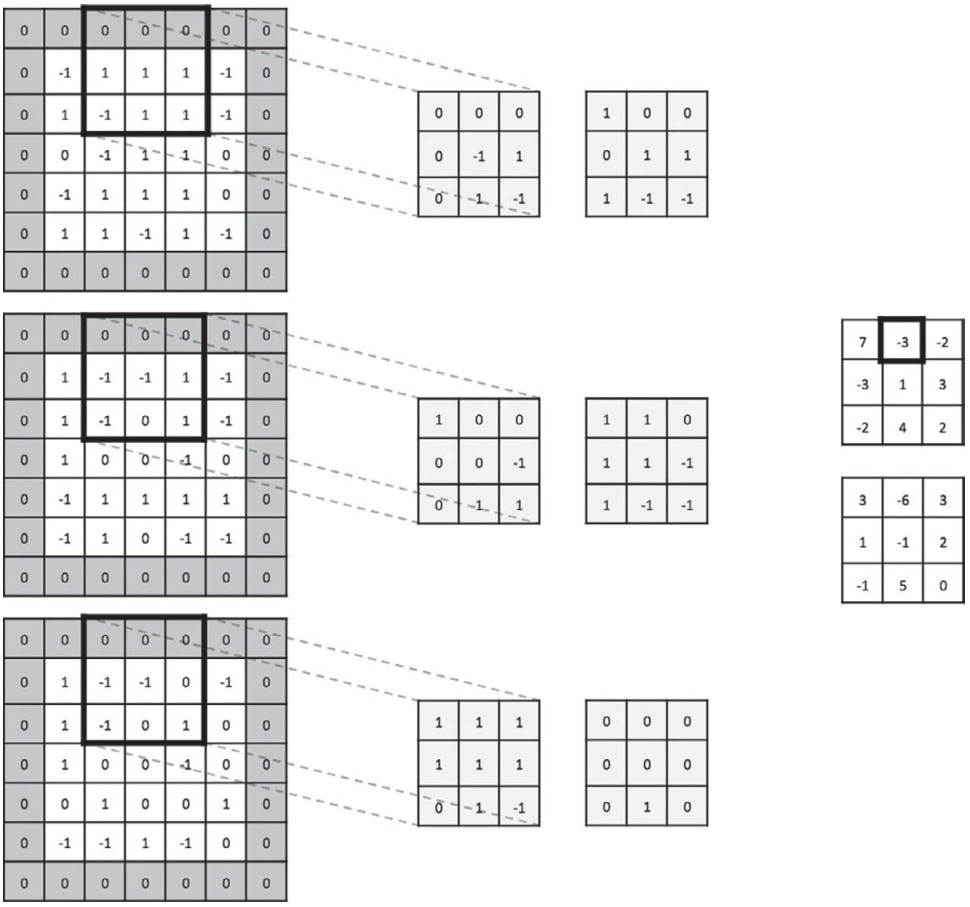


Рис. 5.12. *Так же, как на рис. 5.11, мы порождаем следующее значение в первом срезе по глубине объема выходных данных*

Max Pooling (операция подвыборки)

Чтобы резко уменьшить размерность карт признаков и заострить внимание на найденных признаках, мы иногда используем слой `max pooling` (слой операции подвыборки) после сверточного[56]. Смысл его — в разбиении карты на сектора равного размера. Так мы получаем сжатую карту признаков. Фактически мы создаем свою ячейку для каждого сектора, вычисляем максимальное значение по нему и записываем его в соответствующую ячейку сжатой карты признаков. Этот процесс показан на рис. 5.13.

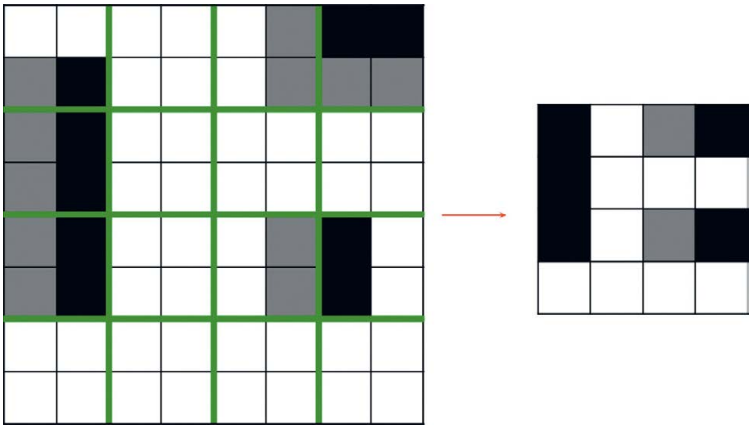


Рис. 5.13. *Иллюстрация того, как max pooling значительно сокращает параметры при продвижении по сети*

Более строго слой подвыборки можно описать при помощи двух параметров:

— пространственной протяженности `e`;

— сдвига `s`.

Важно отметить, что используются лишь два основных варианта слоя подвыборки. Первый — неперекрывающийся с параметрами `e = 2, s = 2`. Второй — перекрывающийся с `e = 3, s = 2`. Получаем следующие размеры каждой карты признаков.

— Ширина $w_{out} = \left[\frac{w_{in} - e}{s}\right] + 1.$

— Высота $h_{out} = \left[\frac{h_{in} - e}{s}\right] + 1.$

Интересное свойство `max pooling` — *локальная инвариантность*. Даже если входные значения немного варьируются, выходные остаются неизменными. Это имеет важные последствия для визуальных алгоритмов. Локальная инвариантность — очень полезное свойство, если нас больше интересует то, есть ли вообще данный признак, а не то, где именно он находится. Но в больших объемах она может повредить способности нашей сети переносить важную информацию. Поэтому пространственная протяженность слоев подвыборки должна оставаться небольшой.

В недавних исследованиях Бенджамин Грэм из Университета Уорвика[57] предложил идею *дробного `max pooling`*. При этом подходе используется генератор псевдослучайных чисел: он создает разбиение на области нецелой длины для дальнейшей подвыборки. Дробное `max pooling` работает

сильным регуляризатором, помогая предотвратить переобучение сверточных нейросетей.

Полное архитектурное описание сверточных нейросетей

Описав составляющие сверточных нейросетей, начнем сводить их вместе. На рис. 5.14 приведены несколько архитектур, которые могут быть полезны на практике.

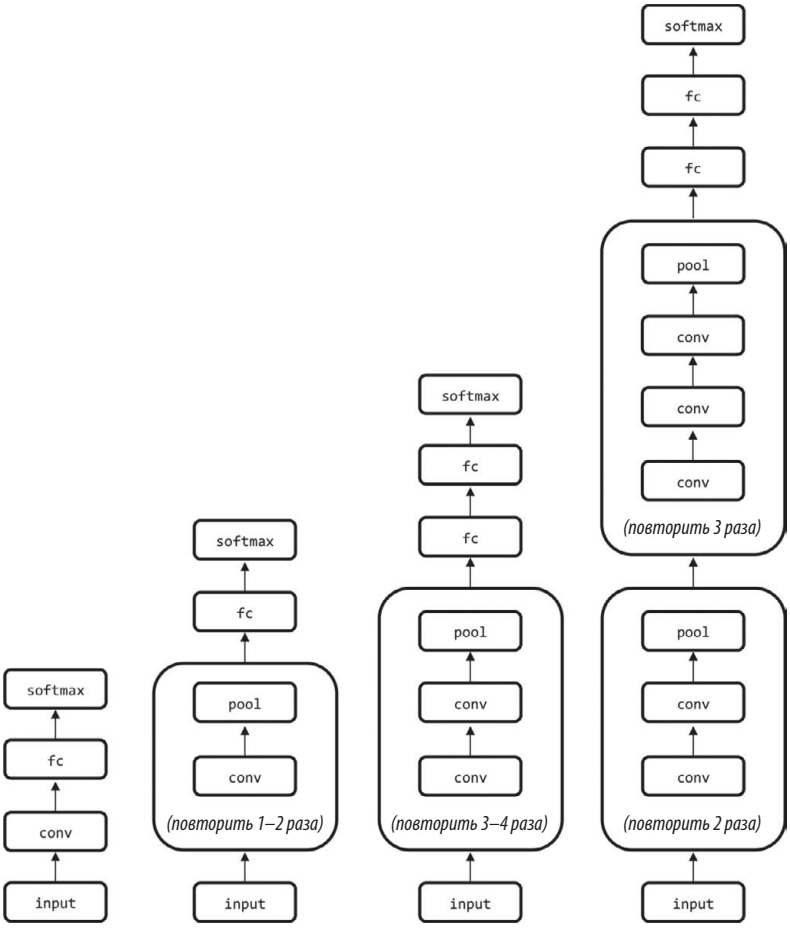


Рис. 5.14. *Различные архитектуры сверточных нейросетей разного уровня сложности. Архитектура VGGNet, глубокой сверточной сети, разработанной для ImageNet, показана на схеме справа*

При построении более глубоких сетей применяется следующий шаблон: мы сокращаем число уровней подвыборки и строим несколько сверточных слоев в тандемах. Обычно это эффективно, потому что операции подвыборки по природе своей деструктивны. Стекирование нескольких сверточных слоев перед каждым слоем подвыборки позволяет достичь большей репрезентативности.

На практике глубокие сверточные сети могут занимать значительное место, и большинство практиков сталкиваются с проблемой бутылочного горлышка: их ограничивает память графических процессоров. Архитектура `VGGNet`, например, занимает примерно 90 MB на изображение при прямом проходе и более 180 MB при обратном для обновления параметров[58]. Многие глубокие сети реализуют компромисс, используя в первом сверточном слое такие сдвиги и пространственные протяженности, которые сокращают объем информации, необходимый для распространения по сети.

Работа с MNIST с помощью сверточных сетей

Теперь мы лучше знаем, как строить сети для эффективного анализа изображений, и возвращаемся к проблеме `MNIST`, с которой разбираемся уже несколько глав. Здесь мы используем сверточную сеть для распознавания рукописных цифр. Наша сеть с прямым распространением сигнала сумела добиться точности 98,2%. Сейчас наша цель — превзойти этот результат.

Построим сверточную сеть с самой что ни на есть стандартной архитектурой (по модели второй сети с рис. 5.14): два слоя подвыборки и два сверточных друг за другом, а за ними полносвязный слой (с прореживанием, $p = 0,5$) и конечный слой с мягким максимумом. Чтобы упростить создание сети, допишем пару вспомогательных методов к генератору слоев из сети с прямым распространением сигнала:

```
def conv2d(input, weight_shape, bias_shape):
in = weight_shape[0] * weight_shape[1] * weight_shape[2]
weight_init = tf.random_normal_initializer(stddev=
       (2.0/in)**0.5)
W = tf.get_variable("W", weight_shape,
       initializer=weight_init)
bias_init = tf.constant_initializer(value=0)
b = tf.get_variable("b", bias_shape, initializer=bias_init)
conv_out = tf.nn.conv2d(input, W, strides=[1, 1, 1, 1],
       padding='SAME')
```

```
    return tf.nn.relu(tf.nn.bias_add(conv_out, b))
def max_pool(input, k=2):
    return tf.nn.max_pool(input, ksize=[1, k, k, 1],
           strides=[1, k, k, 1], padding='SAME')
```

Первый вспомогательный метод генерирует сверточный слой конкретной формы. Мы задаем значение 1 для сдвига и дополнение, чтобы ширина и высота входного и выходного тензоров совпадали. Мы также инициализируем веса при помощи той же эвристики, что применялась в сети с прямым распространением сигнала. Но здесь количество входящих в нейрон весов включает высоту и ширину фильтра и глубину входного тензора.

Второй вспомогательный метод генерирует слой `max pooling` с неперекрывающимися окнами размера `k`. Рекомендуемое значение по умолчанию `k=2`, им мы и воспользуемся для нашей сверточной сети `MNIST`.

С этими вспомогательными методами можно создать новый конструктор логического вывода:

```
def inference(x, keep_prob):
    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 1, 32], [32])
        pool_1 = max_pool(conv_1)
    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 32, 64], [64])
        pool_2 = max_pool(conv_2)
    with tf.variable_scope("fc"):
        pool_2_flat = tf.reshape(pool_2, [-1, 7 * 7 * 64])
        fc_1 = layer(pool_2_flat, [7*7*64, 1024], [1024])
        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)
    with tf.variable_scope("output"):
        output = layer(fc_1_drop, [1024, 10], [10])
    return output
```

Понять код несложно. Сначала берем входные значения пикселов в виде плоского вектора и преобразуем их в тензор $N \times 28 \times 28 \times 1$, где N — количество примеров в мини-выборке, 28 — ширина и высота каждого изображения, а 1 — глубина (изображения черно-белые; если бы они были в режиме `RGB`, глубина равнялась бы 3, по числу цветов). Затем строим сверточный слой с 32 фильтрами, имеющими пространственную протяженность 5.

В результате получаем объем входных значений глубиной 1 и выдаем выходной тензор глубиной 32. Он передается через слой `max pooling`, где информация сжимается. Затем мы строим второй сверточный слой с 64 фильтрами, снова пространственной протяженностью 5: берем входной тензор глубиной 32 и выдаем выходной глубиной 64. Он снова проходит через слой `max pooling`, где информация сжимается.

Мы готовы передать выходные данные из слоя `max pooling` на полносвязный слой. Для этого мы преобразуем тензор в плоский вектор. Это можно сделать, вычислив полный размер каждого «субтензора» в мини-выборке. У нас 64 фильтра, что соответствует глубине 64. Теперь нужно определить высоту и ширину после прохода через два слоя `max pooling`. Используя формулы из предыдущего раздела, легко убедиться, что каждая карта признаков имеет высоту и ширину 7. Подтверждение этого мы оставляем читателю в качестве упражнения.

После операции изменения размерности мы при помощи полносвязного слоя сжимаем сглаженное отображение до скрытого состояния размером `1024`. Вероятность прореживания в этом слое мы определяем как 0,5 при обучении и 1 во время оценки модели (стандартная процедура при использовании прореживания). Мы отправляем это скрытое состояние в исходящий слой мягкого максимума с 10 выходами (`softmax`, как обычно, вычисляется в конструкторе `loss` для повышения эффективности).

Наконец, мы обучаем сеть при помощи оптимизатора `Adam`. После нескольких эпох работы с набором данных мы получаем точность 99,4% — не шедевр (все-таки не 99,7–99,8%), но очень даже неплохо.

Предварительная обработка изображений улучшает работу моделей

Пока мы имели дело с простыми наборами данных. Почему `MNIST` считается таковым? В целом потому, что он уже прошел предварительную обработку и все изображения в нем похожи друг на друга. Рукописные цифры расположены примерно одинаково; нет отклонений цвета, поскольку `MNIST` включает черно-белые изображения, и т. д. Естественные же изображения — совсем другой уровень сложности. Они выглядят беспорядочно, и можно выполнить несколько операций по их предварительной обработке, чтобы облегчить обучение. Первая методика, которая поддерживается в `TensorFlow`, — приближенное выбеливание. Главная задача — сместить к 0 каждый пиксел в изображении, вычтя среднее значение и нормализовав дисперсию до 1. В `TensorFlow` это достигается так:

```
tf.image.per_image_whitening(image)
```

Можно искусственно расширить набор данных, случайным образом обрезав изображение, повернув его, изменив насыщенность или яркость и т. д.:

```
tf.random_crop(value, size, seed=None, name=None)
tf.image.random_flip_up_down(image, seed=None)
tf.image.random_flip_left_right(image, seed=None)
tf.image.transpose_image(image)
tf.image.random_brightness(image, max_delta, seed=None)
tf.image.random_contrast(image, lower, upper, seed=None)
tf.image.random_saturation(image, lower, upper, seed=None)
tf.image.random_hue(image, max_delta, seed=None)
```

Применение таких трансформаций позволяет создавать сети, эффективные при различных вариациях, которые встречаются в естественных изображениях, и делать предсказания с высокой точностью, несмотря на потенциальные искажения.

Ускорение обучения с помощью пакетной нормализации

В 2015 году исследователи из Google разработали замечательный способ ускорить обучение сетей с прямым распространением сигнала и сверточных с помощью *пакетной нормализации*[59]. Этот процесс можно визуализировать, представив себе башню из строительных кирпичиков (рис. 5.15).

Когда кирпичи в башне сложены аккуратно, структура стабильна. Но если мы случайным образом сдвигаем их, это может привести к нарастанию нестабильности башни: со временем она упадет и развалится.

Похожее явление наблюдается при обучении нейронных сетей. Представьте себе двухслойную нейронную сеть. В процессе обучения весов сети распределение на выходе нейронов в нижнем слое начинает смещаться. В результате верхний слой должен будет не только научиться делать соответствующие предсказания, но и измениться сам, чтобы нивелировать сдвиги во входном распределении. Это значительно замедляет процесс обучения, и проблемы нарастают с увеличением числа слоев.

Нормализация входных изображений улучшает процесс, делая его более устойчивым к вариациям. Пакетная нормализация делает шаг вперед, нормализуя данные на входе во все слои нейронной сети.

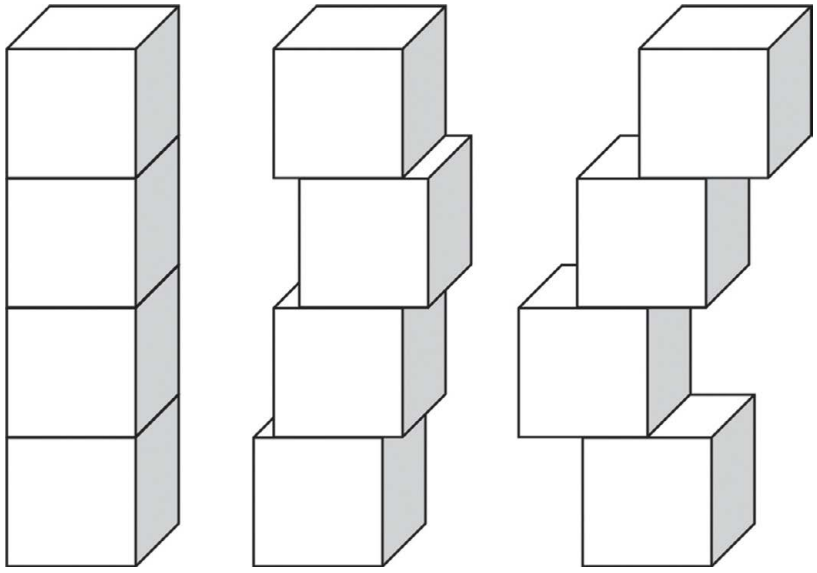


Рис. 5.15. *Когда кирпичики в башне смещаются слишком сильно и перестают быть единым целым, структура может стать очень нестабильной*

Мы модифицируем архитектуру сети, вводя операции, которые выполняют следующие функции.

1. Перехватывают векторы логитов, поступающие в слой, до того как те дойдут до нелинейности.

2. Нормализуют каждый компонент вектора логитов по всем примерам мини-выборки, вычитая среднее и деля на стандартное отклонение (мы записываем импульсы, используя экспоненциально взвешенное скользящее среднее).

3. Получив нормализованные входящие значения $\hat{\mathbf{x}}$, используют аффинные преобразования для восстановления репрезентативности с двумя векторами (обучаемых) параметров: $\gamma\hat{\mathbf{x}} + \beta$.

В `TensorFlow` пакетная нормализация для сверточного слоя может выглядеть так:

```
def conv_batch_norm(x, n_out, phase_train):
      beta_init = tf.constant_initializer(value=0.0,
                                          dtype=tf.float32)
      gamma_init = tf.constant_initializer(value=1.0,
                                          dtype=tf.float32)
```

```
    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                           initializer=gamma_init)
    batch_mean, batch_var = tf.nn.moments(x, [0,1,2],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
                        ema.average(batch_var)
    def mean_var_with_update():
         with tf.control_dependencies([ema_apply_op]):
                return tf.identity(batch_mean),
                       tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train,
         mean_var_with_update,
         lambda: (ema_mean, ema_var))
    normed = tf.nn.batch_norm_with_global_normalization(x,
         mean, var, beta, gamma, 1e-3, True)
    return normed
```

Так же можно выразить пакетную нормализацию и для несверточных слоев с прямым распространением сигнала — с небольшой модификацией вычисления моментов и преобразованием формы для совместимости с `tf.nn.batch_norm_with_global_normalization`.

```
def layer_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0,
                                        dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0,
                                        dtype=tf.float32)
    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                           initializer=gamma_init)
    batch_mean, batch_var = tf.nn.moments(x, [0],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
```

```
ema_apply_op = ema.apply([batch_mean, batch_var])
ema_mean, ema_var = ema.average(batch_mean),
                    ema.average(batch_var)
def mean_var_with_update():
     with tf.control_dependencies([ema_apply_op]):
            return tf.identity(batch_mean),
                    tf.identity(batch_var)
mean, var = control_flow_ops.cond(phase_train,
     mean_var_with_update,
     lambda: (ema_mean, ema_var))
x_r = tf.reshape(x, [-1, 1, 1, n_out])
normed = tf.nn.batch_norm_with_global_normalization(x_r,
     mean, var, beta, gamma, 1e-3, True)
return tf.reshape(normed, [-1, n_out])
```

Помимо ускорения обучения (предотвращаются значительные сдвиги в распределении входных значений в каждом слое), пакетная нормализация также позволяет существенно увеличить темп усвоения нового материала. Более того, она служит регуляризатором, устраняет необходимость в прореживании и `L2`-регуляризации. Мы не будем приводить пример, но авторы также заявляют, что пакетная регуляризация в основном устраняет необходимость фотометрических искажений и мы можем во время обучения поставлять в сеть более «реалистичные» изображения.

Сейчас, разработав усовершенствованное средство анализа естественных изображений при помощи сверточных сетей, мы создадим классификатор для работы с `CIFAR-10`.

Создание сверточной сети для CIFAR-10

Набор `CIFAR-10` состоит из цветных изображений размером 32×32 пиксела, принадлежащих к одному из возможных классов[60]. Это удивительно сложная задача, ведь даже человеку порой тяжело определить, что изображено на картинке. Пример приведен на рис. 5.16.

В этом разделе мы будем создавать сети с пакетной нормализацией и без нее — для сравнения. Мы увеличим норму обучения для сети с пакетной нормализацией в 10 раз, чтобы извлечь из нее все преимущества. Мы приводим здесь код только для такой сети, потому что обычная сверточная сеть создается очень похожим способом.

Рис. 5.16. *Собака из набора данных CIFAR-10*

Мы деформируем случайным образом обрезанные до размера 24×24 пиксела входящие изображения и загружаем их в нашу сеть для обучения. Применим образец кода, предоставляемый Google. Перейдем сразу к архитектуре сети. Для начала разберемся, как интегрировать пакетную нормализацию в сверточные и полносвязные слои. Как и ожидалось, она применяется к логитам до их поступления в нелинейность:

```
def conv2d(input, weight_shape, bias_shape, phase_train,
            visualize=False):
incoming = weight_shape[0] * weight_shape[1]
                           * weight_shape[2]
weight_init = tf.random_normal_initializer(stddev=
            (2.0/incoming)**0.5)
W = tf.get_variable("W", weight_shape,
                    initializer=weight_init)
```

```
if visualize:
      filter_summary(W, weight_shape)
bias_init = tf.constant_initializer(value=0)
b = tf.get_variable("b", bias_shape, initializer=bias_init)
logits = tf.nn.bias_add(tf.nn.conv2d(input, W,
           strides=[1, 1, 1, 1], padding='SAME'), b)
return tf.nn.relu(conv_batch_norm(logits, weight_shape[3],
      phase_train))
def layer(input, weight_shape, bias_shape, phase_train):
weight_init = tf.random_normal_initializer(stddev=
           (2.0/weight_shape[0])**0.5)
bias_init = tf.constant_initializer(value=0)
W = tf.get_variable("W", weight_shape,
                   initializer=weight_init)
b = tf.get_variable("b", bias_shape,
                   initializer=bias_init)
logits = tf.matmul(input, W) + b
return tf.nn.relu(layer_batch_norm(logits, weight_shape[1],
                   phase_train))
```

Дальнейшая архитектура несложная. Мы берем два сверточных слоя (за каждым из которых следует слой подвыборки). Затем идут два полносвязных слоя, а за ними — функция с мягким максимумом. На всякий случай мы включили прореживание, но для версии с пакетной нормализацией во время обучения значение `keep_prob` = 1:

```
def inference(x, keep_prob, phase_train):
with tf.variable_scope("conv_1"):
conv_1 = conv2d(x, [5, 5, 3, 64], [64], phase_train,
                   visualize=True)
pool_1 = max_pool(conv_1)
with tf.variable_scope("conv_2"):
      conv_2 = conv2d(pool_1, [5, 5, 64, 64], [64],
                   phase_train)
      pool_2 = max_pool(conv_2)
with tf.variable_scope("fc_1"):
      dim = 1
      for d in pool_2.get_shape()[1:].as_list():
```

```
dim *= d
pool_2_flat = tf.reshape(pool_2, [-1, dim])
fc_1 = layer(pool_2_flat, [dim, 384], [384],
             phase_train)
# apply dropout
fc_1_drop = tf.nn.dropout(fc_1, keep_prob)
with tf.variable_scope("fc_2"):
fc_2 = layer(fc_1_drop, [384, 192], [192], phase_train)
# apply dropout
fc_2_drop = tf.nn.dropout(fc_2, keep_prob)
with tf.variable_scope("output"):
output = layer(fc_2_drop, [192, 10], [10], phase_train)
return output
```

Наконец, при помощи оптимизатора `Adam` переходим к обучению сверточных сетей. Через какое-то время наши сети добиваются впечатляющей точности выполнения задачи по `CIFAR-10`: 92,3% без пакетной нормализации и 96,7% с ее использованием. Этот результат соответствует (а потенциально и превосходит) самому эффективному на данный момент варианту решении этой задачи! В следующем разделе мы подробнее рассмотрим визуализацию обучения и работы наших сетей.

Визуализация обучения в сверточных сетях

Проще всего визуализировать обучение, построив функцию потерь на данных и проверки по мере прохождения процесса. Четко показать выгоды пакетной нормализации можно, сравнив скорости схождения двух наших сетей. Графики середины процесса тренировки приведены на рис. 5.17.

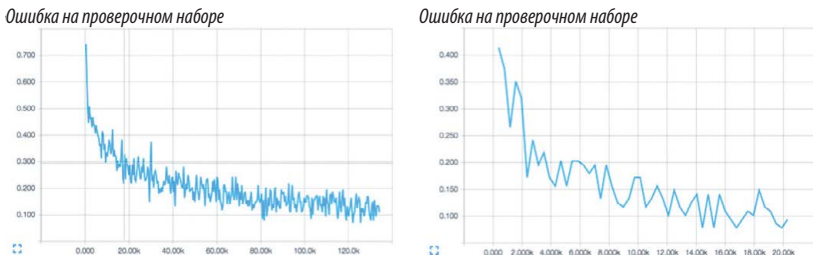


Рис. 5.17. *Обучение сверточной сети без пакетной нормализации (слева) и с пакетной нормализацией (справа). Последняя значительно ускоряет процесс*

Без пакетной нормализации достижение порога точности в 90% требует применения около 80 тысяч мини-пакетов. С пакетной же нормализацией этот порог достигается всего за 14 тысяч мини-пакетов.

Можно также рассмотреть фильтры, которым обучается наша сверточная сеть, и понять, что она считает важным для решений по классификации.

Сверточные слои обучаются иерархическим представлениям, и мы надеемся, что первый сверточный слой усвоит базовые признаки (края, простые кривые и т. д.), а второй — более сложные. К сожалению, интерпретация второго сверточного слоя сложна даже при визуализации, так что на рис. 5.18 представлены только фильтры первого слоя.

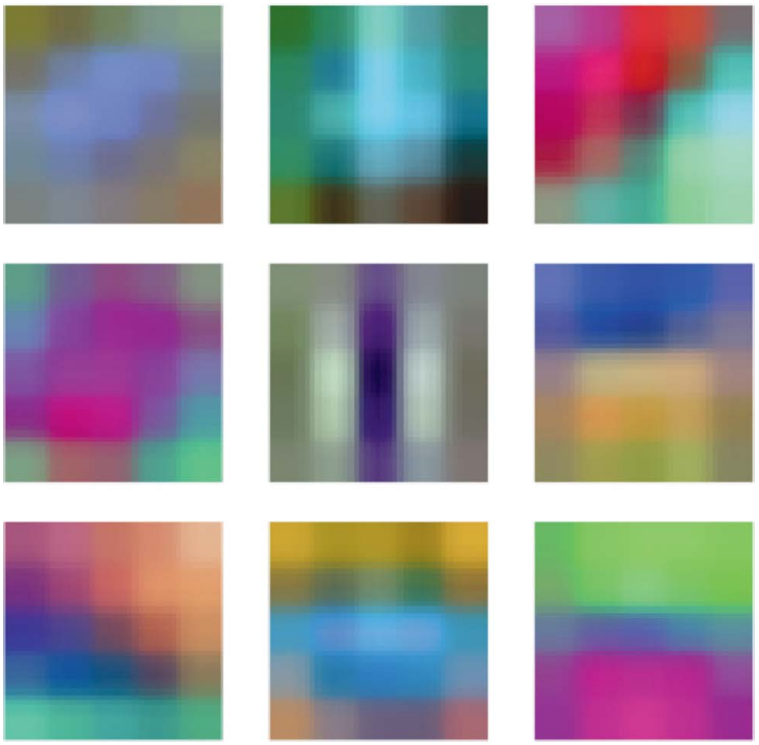


Рис. 5.18. *Фрагмент набора выученных фильтров в первом сверточном слое нашей сети*

У наших фильтров есть ряд интересных свойств: это вертикальные, горизонтальные и диагональные границы, а также небольшие точки или пятна одного цвета, окруженные другим. Безусловно, сеть обучается релевантным свойствам, поскольку фильтры точно нельзя считать обычным шумом.

Можно попытаться визуализировать то, как наша сеть научилась группировать разные виды изображений. Возьмем большую сеть, обученную на примере задач `ImageNet`, и рассмотрим скрытое состояние полносвязного слоя перед слоем подвыборки для каждого изображения.

Затем мы берем это многомерное представление каждого изображения и применяем алгоритм, известный как метод нелинейного снижения размерности и визуализации многомерных переменных (*`t-Distributed Stochastic Neighbor Embedding`*, или *`t-SNE`*), для сжатия до двумерного представления, которое уже можно визуализировать[61].

Подробно о `t-SNE` мы писать не будем, но есть много бесплатных доступных инструментов, в том числе код, которые нам помогут (*`https://lvdmaaten.github.io/tsne/code/tsne_python.zip`*). Мы визуализируем результаты на рис. 5.19, и они весьма впечатляющи.

Рис. 5.19. *Визуализация представлений алгоритма t-SNE (в центре), окруженная увеличенными подсегментами представлений (по краям).* *© Андрей Карпати*[62]

Сначала кажется, что изображения похожего цвета ближе друг к другу. Это интересно, но еще удивительнее другое: когда мы приближаем элементы визуализации, оказывается, что речь не только о цвете. Все изображения лодок находятся в одном месте, людей — в другом, а бабочек — в третьем сегменте визуализации. Очевидно, что возможности обучения сверточных сетей впечатляющи.

Применение сверточных фильтров для воссоздания художественных стилей

За последние пару лет мы разработали алгоритмы гораздо более сложного применения сверточных сетей. Один из них называется *нейронным стилем*[63]. Его цель — взять произвольную фотографию и переделать под стиль знаменитого художника. Задача кажется устрашающей, и не вполне понятно, как мы решили бы эту проблему, не будь сверточных сетей. Но оказывается, что умелое применение фильтров приводит к замечательным результатам.

Возьмем предварительно обученную сверточную сеть и три изображения. Первые два — источник содержания p (content) и источник стиля a (style). Третье — сгенерированное x. Наша задача — предложить функцию потерь для алгоритма обратного распространения ошибок так, чтобы при минимизации получилось идеальное сочетание содержания желаемой фотографии и стиля произведения искусства.

Начнем с содержания. Если слой в сети имеет k_l фильтров, он порождает k_l карт признаков. Обозначим размер каждой карты как m_l, высота на ширину карты признаков. Активация всех карт признаков в этом слое может быть записана в матрицу $\boldsymbol{F}^{(l)}$ размера $k_l \times m_l$. Также все активации фотографии можно внести в матрицу $\boldsymbol{P}^{(l)}$, а активации сгенерированного изображения — в матрицу $\boldsymbol{X}^{(l)}$. Воспользуемся `relu4_2` исходной `VGGNet`:

$$E_{content}(\mathbf{p}, \mathbf{x}) = \sum_{ij} \left(\mathbf{P}_{ij}^{(l)} - \mathbf{X}_{ij}^{(l)} \right)^2.$$

Попробуем перейти к стилю. Создадим *матрицу Грама*, в которой представлена корреляция между картами признаков в соответствующем слое. Корреляция представляет текстуру и ощущения, общие для всех признаков, которые мы рассматриваем. Создание матрицы Грама размером $k_l \times k_l$ для заданного изображения происходит следующим образом:

$$\mathrm{G}_{ij}^{(l)} = \sum_{c=0}^{ml} \mathbf{F}_{ic}^{(l)} \mathbf{F}_{jc}^{(l)}.$$

Мы можем вычислить матрицы Грама как для произведения искусства в матрице $\boldsymbol{A}^{(l)}$, так и для сгенерированного изображения в матрице $\boldsymbol{G}^{(l)}$. Теперь представим функцию ошибок:

$$E_{style}(\mathbf{a}, \mathbf{x}) = \frac{1}{4k_l^2 m_l^2} \sum_{l=1}^{L} \sum_{ij} \frac{1}{L} \left(A_{ij}^{(l)} - G_{ij}^{(l)} \right)^2.$$

Здесь мы равным образом взвешиваем каждый квадрат разности (разделив его на число слоев, которые мы хотим включить в работу по воссозданию стиля).

Мы используем слои `relu1_1`, `relu2_1`, `relu3_1`, `relu4_1` и `relu5_1` исходной `VGGNet`. Опустим для краткости обсуждение кода `TensorFlow` (*`http://bit.ly/2qAODnp`*), но отметим, что результаты, приведенные на рис. 5.20, снова впечатляют. Мы решили совместить фотографию знаменитого купола Массачусетского технологического университета и картину Леонида Афремова «Принцесса дождя».

Рис. 5.20. *Результат совмещения «Принцессы дождя» с фотографией купола Массачусетского технологического университета. © Аниш Атали*

Обучаем сверточные фильтры в других областях

Хотя примеры в этой главе касались только распознавания изображений, есть и другие области, в которых полезны сверточные сети. Естественным продолжением анализа изображений становится анализ видео. Использование пятимерных тензоров (где одно из измерений — время) и применение трехмерных сверток — простой способ распространить сверточную парадигму на видео[64]. Сверточные фильтры также успешно применяются для анализа аудиограмм[65]. В этом случае сеть скользит по входящей аудиограмме и предсказывает последующие фонемы.

Менее интуитивный вариант использования сверточных сетей — обработка естественного языка. Некоторые примеры мы рассмотрим в следующих главах. Более экзотические варианты применений сверточных сетей — обучение алгоритмов игре в настольные игры и анализ биомолекул для разработки лекарств. Оба примера будут описаны в следующих главах.

Резюме

В этой главе мы научились создавать нейронные сети для анализа изображений. Мы вывели понятие свертки и применили ее для создания удобных сетей, которые способны анализировать и простые, и более сложные естественные изображения. Мы построили несколько сверточных сетей в `TensorFlow`, применив разные варианты обработки изображений и пакетную нормализацию, что способствовало ускорению обучения наших сетей и увеличению их гибкости. Наконец, мы визуализировали процесс обучения сверточных сетей и рассмотрели другие интересные варианты применения этой технологии.

Анализировать изображения было несложно, ведь мы могли эффективно представить их в виде тензоров. В других ситуациях (например, при анализе естественного языка) не так очевидно, как представить входные данные в виде тензоров. Чтобы разобраться с этим и перейти к новым моделям глубокого обучения, рассмотрим в следующей главе несколько ключевых понятий векторного представления и изучения представлений.

ГЛАВА 6

Плотные векторные представления и обучение представлений

Обучение представлений в пространстве низкой размерности

В предыдущей главе мы простым аргументом мотивировали сверточную архитектуру. Чем больше входной вектор, тем больше наша модель. Крупные модели со множеством параметров выразительны, но требуют множества данных. Это значит, что без достаточного количества данных для обучения, скорее всего, неизбежно переобучение.

Сверточные архитектуры позволяют нам справиться с проклятием размерности, снижая число параметров в модели и не жертвуя выразительностью. Однако сверточным сетям все равно нужно много размеченных обучающих данных. И часто таких данных мало, а создавать их дорого. Наша цель в этой главе — разработать эффективные обучающиеся модели в таких ситуациях, когда размеченных данных мало, а «диких», неразмеченных очень много. Для этого мы будем обучать *плотные векторные представления (embedding)*, или представления с меньшей размерностью, без учителя. Поскольку модели без учителя позволяют выбирать признаки автоматически, мы можем при помощи сгенерированных плотных векторных представлений решать проблемы обучения для менее масштабных моделей, которым требуется меньше данных. Этот процесс представлен на рис. 6.1.

В процессе разработки алгоритмов, хорошо обучающих плотные векторные представления, мы рассмотрим и другие варианты применения обучения представлений с меньшей размерностью — например, визуализацию и семантическое хэширование. Начнем с ситуаций, когда вся важная

информация уже есть в самом входном векторе. В этом случае обучение плотного векторного представления эквивалентно разработке эффективного алгоритма сжатия.

Рис. 6.1. *Использование плотных векторных представлений позволяет автоматизировать выбор признаков при недостатке размеченных данных*

В следующем разделе мы введем понятие метода главных компонент (PCA — principal component analysis), классического способа уменьшения размерности. Далее мы рассмотрим более эффективные нейронные методы обучения плотных представлений.

Метод главных компонент

Основная идея метода главных компонент в том, что нам нужно найти такой набор осей, который будет передавать наибольшее количество информации о нашем наборе данных. А именно: если у нас есть d-размерные данные, нам нужно найти новый набор измерений $m < d$, который сохраняет максимально возможное количество ценной информации из исходного набора данных. Для удобства пусть $d = 2$, $m = 1$. Предположив, что вариативность связана с информацией, мы можем выполнить это преобразование с помощью итеративного процесса. Для начала находим единичный вектор, по которому набор данных наиболее вариативен. Поскольку это направление содержит больше всего информации, мы делаем его своей первой осью. Затем из набора векторов, ортогональных первому, мы выбираем новый единичный вектор, по которому набор данных опять же наиболее вариативен. Это вторая ось. Повторяем процесс, пока не находим d новых векторов, соответствующих новым осям. Проецируем наши данные

на эти новые оси. Затем определяем подходящее значение m и избавляемся от всех осей, кроме m первых (главных компонент, которые содержат больше всего информации). Результат показан на рис. 6.2.

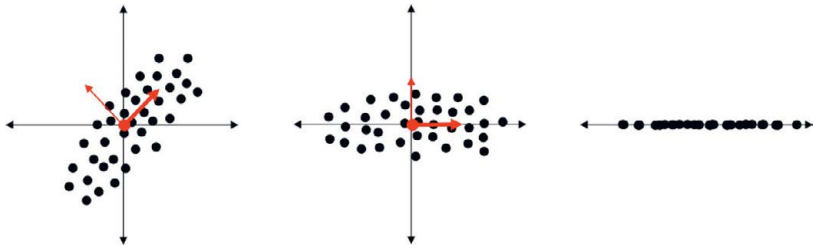


Рис. 6.2. *Иллюстрация метода главных компонент для снижения размерности, сохраняющего измерение с наибольшим количеством информации (выраженном в вариативности)*

Люди с математическим складом ума могут рассматривать эту операцию как проецирование на векторное пространство, порожденное верхними m собственными векторами ковариантной матрицы набора данных (при постоянном масштабе). Представим набор данных в виде матрицы $\mathbf{X}$ с размерами $n \times d$ (то есть n входов и d измерений). Мы хотим создать вложенную матрицу $\mathbf{T}$ с размерами $n \times m$. Эту матрицу можно вычислить, используя отношения $\mathbf{T} = \mathbf{X}$, где каждый столбец $\mathbf{W}$ соответствует собственному вектору матрицы $\mathbf{X}^T\mathbf{X}$.

Хотя метод главных компонент используется для снижения размерности уже несколько десятилетий, он почему-то не может сохранить важные отношения — как кусочно-линейные, так и кусочно-нелинейные. Возьмем пример, изображенный на рис. 6.3.

В этом примере показаны точки данных, случайно выбранные из двух концентрических кругов. Мы надеемся, что метод главных компонент преобразует этот набор данных так, чтобы мы смогли взять единую новую ось, которая позволит легко отделить красные точки от синих. К сожалению для нас, здесь нет такого линейного направления, которое содержит больше информации, чем любое другое (вариативность во всех направлениях равная). Мы, люди, отмечаем, что информация здесь кодируется нелинейно — тем, насколько точки удалены от центра. Имея в виду эту информацию, фиксируем, что преобразование к полярным координатам (выражение точек через их расстояние от центра в виде новой горизонтальной оси и через их угловое отклонение от первичной оси x в виде новой вертикальной оси) как раз справляется хорошо.

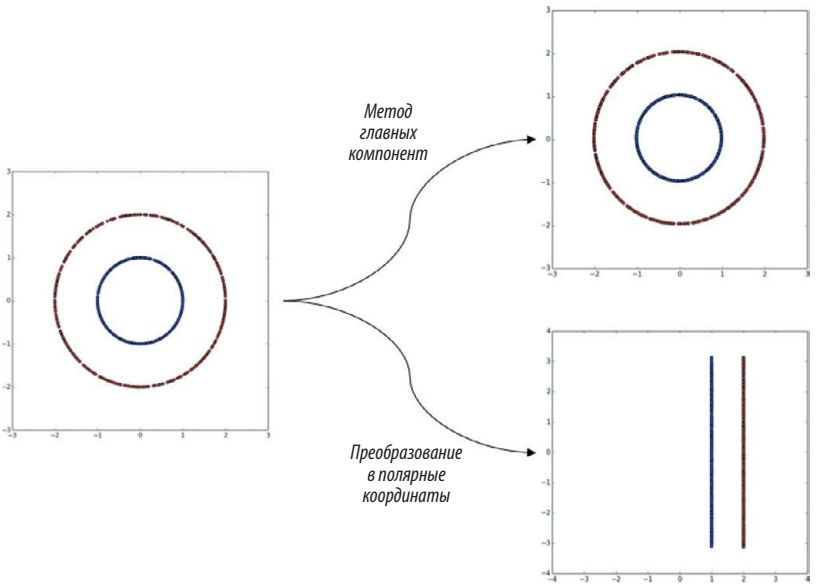


Рис. 6.3. *Ситуация, в которой метод главных компонент не может оптимально преобразовать данные для снижения размерности*

Рисунок 6.3 подчеркивает недостатки такого подхода, как метод главных компонент, при сохранении важных отношений в сложных наборах данных. Поскольку большинство наборов данных, которые встретятся нам на практике (изображения, тексты и т. д.), характеризуются нелинейными отношениями, нужно разработать теорию, способную обеспечить нелинейное снижение размерности. Практики глубокого обучения закрыли эту брешь с помощью нейронных моделей, к которым мы и перейдем в следующем разделе.

Мотивация для архитектуры автокодера

Когда мы говорили о сетях с прямым распространением сигнала, то показывали, как каждый слой последовательно обучался более релевантным представлениям входных данных. А в главе 5 мы взяли данные из последнего сверточного слоя и использовали их как представление входного изображения с пониженной размерностью. Но эти подходы связаны с фундаментальными проблемами, не говоря о том, что мы хотим создавать эти представления с меньшим числом размерностей без учителя. Хотя выбранный слой действительно содержит информацию о входных данных, сеть обучена так, что обращает внимание на те их аспекты входных данных,

которые необходимы для решения текущей задачи. В результате теряется много информации, которая могла бы понадобиться для других задач классификации, а для текущих целей не так важна.

Но и здесь помогает простая интуиция. Мы определяем новую архитектуру сети, которую называем *автокодером*. Сначала берем входящие данные и сжимаем их в вектор с пониженной размерностью. Эта часть сети называется *кодером*, поскольку она отвечает за создание плотного векторного представления, или *кода*. Вторая часть сети вместо того, чтобы соотнести плотное векторное представление с произвольной меткой, как в сети с прямым распространением сигнала, пытается инвертировать вычисления первой части сети и воссоздать исходные входные данные. Она называется *декодером*. Общая архитектура показана на рис. 6.4.

Рис. 6.4. *Архитектура автокодера пытается вместить входные данные высокой размерности в плотное векторное представление, а затем использует это представление для того, чтобы воссоздать входные данные*

Чтобы показать удивительную эффективность автокодера, построим и визуализируем на рис. 6.5 его архитектуру. Мы продемонстрируем его лучшие результаты в определении цифр `MNIST` по сравнению с анализом главных компонент.

Реализация автокодера в TensorFlow

Ключевая работа «Снижение размерности данных при помощи нейронных сетей», в которой описан автокодер, была создана Джефри Хинтоном и Русланом Салахутдиновым в 2006 году[66]. Их гипотеза состояла в том, что нелинейные сложности нейронной модели позволят сохранять структуру, которую нельзя зафиксировать линейными методами вроде метода главных компонент. Для этого они провели эксперимент с `MNIST`, используя как автокодер, так и `PCA` для сведения набора данных к двумерным точкам. В этом разделе мы воссоздадим их эксперимент, чтобы проверить эту гипотезу, и продолжим рассмотрение архитектуры и свойств автокодеров с прямым распространением сигнала.

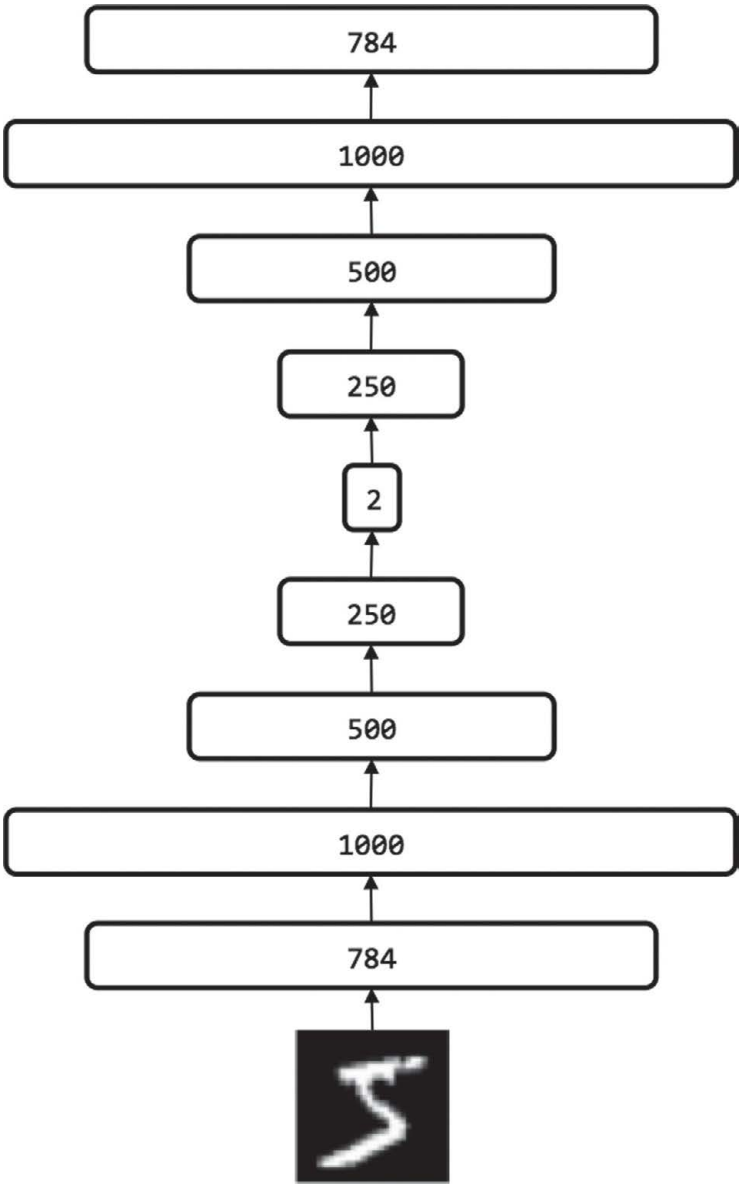


Рис. 6.5. *Эксперимент по снижению размерности набора данных MNIST, проведенный Хинтоном и Салахутдиновым в 2006 году*

Архитектура, показанная на рис. 6.5, основана на том же принципе, но сейчас двумерное представление считается входными данными, а сеть пытается воссоздать исходное изображение. Поскольку мы проводим

обратную операцию, мы строим декодирующую сеть, и автокодер будет иметь форму песочных часов. На выходе в ней получается вектор с 784 измерениями, который можно воссоздать в виде изображения размером 28×28 пикселов:

```
def decoder(code, n_code, phase_train):
   with tf.variable_scope("decoder"):
      with tf.variable_scope("hidden_1"):
         hidden_1 = layer(code, [n_code, n_decoder_hidden_1],
            [n_decoder_hidden_1], phase_train)
      with tf.variable_scope("hidden_2"):
         hidden_2 = layer(hidden_1, [n_decoder_hidden_1,
                     n_decoder_hidden_2], [n_decoder_hidden_2],
phase_train)
      with tf.variable_scope("hidden_3"):
         hidden_3 = layer(hidden_2, [n_decoder_hidden_2,
                     n_decoder_hidden_3], [n_decoder_hidden_3],
phase_train)
      with tf.variable_scope("output"):
         output = layer(hidden_3, [n_decoder_hidden_3, 784],
                  [784], phase_train)
return output
```

Для ускорения обучения мы используем ту же стратегию пакетной нормализации, что и в главе 5. Кроме того, поскольку мы хотим визуализировать результаты, мы избегаем резких переходов в нейронах. В этом примере мы используем сигмоидные нейроны вместо обычных `ReLU`-нейронов:

```
def layer(input, weight_shape, bias_shape, phase_train):
   weight_init = tf.random_normal_initializer(stddev=
                 (1.0/weight_shape[0])**0.5)
   bias_init = tf.constant_initializer(value=0)
   W = tf.get_variable("W", weight_shape,
                       initializer=weight_init)
   b = tf.get_variable("b", bias_shape,
                       initializer=bias_init)
   logits = tf.matmul(input, W) + b
   return tf.nn.sigmoid(layer_batch_norm(logits,
                       weight_shape[1],
            phase_train))
```

Теперь нужно выработать метрику (или целевую функцию), которая описывает, насколько хорошо функционирует наша модель. А именно: надо измерить, насколько близка реконструкция к исходному изображению. Это можно сделать, вычислив расстояние между исходными 784-мерными данными и воссозданными 784-мерными данными, полученными на выходе. Если входной вектор — I, а воссозданный — O, нужно минимизировать значение

$$\|I-O\|=\sqrt{\sum_i (I_i-O_i)^2},$$

известное также как L2-норма разницы между двумя векторами. Мы усредняем функцию по всему мини-пакету и получаем итоговую целевую функцию. После этого мы обучаем сеть с помощью оптимизатора Adam, записывая скалярную статистику ошибок в каждом мини-пакете при помощи tf.scalar_summary. В TensorFlow операции расчета функции потерь и обучения можно кратко выразить так:

```
def loss(output, x):
      with tf.variable_scope("training"):
            l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x)),
      1))
            train_loss = tf.reduce_mean(l2)
            train_summary_op = tf.scalar_summary("train_cost",
      train_loss)
            return train_loss, train_summary_op
def training(cost, global_step):
      optimizer = tf.train.AdamOptimizer(learning_rate=0.001,
            beta1=0.9, beta2=0.999, epsilon=1e-08,
            use_locking=False, name='Adam')
      train_op = optimizer.minimize(cost, global_step=global_step)
      return train_op
```

Наконец, нам нужен способ оценки обобщающей способности модели. Как обычно, мы берем проверочный набор данных и вычисляем тот же показатель L2-нормы. Вдобавок мы сохраняем статистические данные по картинкам для сравнения входных изображений с их реконструкциями:

```
def image_summary(summary_label, tensor):
      tensor_reshaped = tf.reshape(tensor, [-1, 28, 28, 1])
      return tf.image_summary(summary_label, tensor_reshaped)
def evaluate(output, x):
      with tf.variable_scope("validation"):
```

```
    in_im_op = image_summary("input_image", x)
    out_im_op = image_summary("output_image", output)
    l2 = tf.sqrt(tf.reduce_sum(tf.square(tf.sub(output, x,
           name="val_diff")), 1))
    val_loss = tf.reduce_mean(l2)
    val_summary_op = tf.scalar_summary("val_cost", val_loss)
    return val_loss, in_im_op, out_im_op, val_summary_op
```

Осталось построить из этих компонентов модель и начать ее обучать. По большей части код здесь знаком, но следует обратить внимание на пару дополнений. Во-первых, мы видоизменили обычный код, включив в него параметр командной строки для определения числа нейронов в слое плотного представления. Например, выполнение команды `$ python autoencoder_mnist.py 2` запустит модель с двумя нейронами в этом слое. Также мы изменяем конфигурацию сохранения модели, чтобы получить больше снимков ее мгновенного состояния. Затем мы загрузим самую эффективную модель, чтобы сравнить ее производительность с методом главных компонент, так что нам нужно больше мгновенных снимков состояния модели.

Воспользуемся `summary writer`, чтобы записывать данные по изображениям, полученные в конце каждой эпохи:

```
if __name__ == '__main__':
      parser = argparse.ArgumentParser(description='Test various
      optimization strategies')
      parser.add_argument('n_code', nargs=1, type=str)
      args = parser.parse_args()
      n_code = args.n_code[0]
      mnist = input_data.read_data_sets("data/", one_hot=True)
      with tf.Graph().as_default():
            with tf.variable_scope("autoencoder_model"):
                  x = tf.placeholder("float", [None, 784]) # mnist
                  data image of shape 28*28=784
                  phase_train = tf.placeholder(tf.bool)
                  code = encoder(x, int(n_code), phase_train)
                  output = decoder(code, int(n_code), phase_train)
                  cost, train_summary_op = loss(output, x)
                  global_step = tf.Variable(0, name='global_step',
                  trainable=False)
```

```
                    train_op = training(cost, global_step)
                    eval_op, in_im_op, out_im_op, val_summary_op =
                    evaluate(output, x)
                    summary_op = tf.merge_all_summaries()
                    saver = tf.train.Saver(max_to_keep=200)
                    sess = tf.Session()
                    train_writer = tf.train.SummaryWriter(
      "mnist_autoencoder_hidden=" + n_code +
                    "_logs/",graph=sess.graph)
            val_writer = tf.train.SummaryWriter(
"mnist_autoencoder_hidden=" + n_code +
                    "_logs/", graph=sess.graph)
            init_op = tf.initialize_all_variables()
            sess.run(init_op)
            # Training cycle
            for epoch in range(training_epochs):
            avg_cost = 0.
            total_batch = int(mnist.train.num_examples/
      batch_size)
            # Loop over all batches
            for i in range(total_batch):
            mbatch_x, mbatch_y =
mnist.train.next_batch(batch_size)
                    # Fit training using batch data
                    _, new_cost, train_summary = sess.run([
                            train_op, cost,
                            train_summary_op],
                    feed_dict={x: mbatch_x,
                            phase_train: True})
                    train_writer.add_summary(train_summary,
            sess.run(global_step))
# Compute average loss
            avg_cost += new_cost/total_batch
            # Display logs per epoch step
            if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1),
```

```
        "cost =", "{:.9f}".format(avg_cost)
                    train_writer.add_summary(train_summary,
                    sess.run(global_step))
                    val_images = mnist.validation.images
                    validation_loss, in_im, out_im,
        val_summary = sess.run([eval_op, in_im_op,
        out_im_op, val_summary_op],
        feed_dict={x: val_images,
        phase_train: False})
             val_writer.add_summary(in_im, sess.run
                    (global_step))
             val_writer.add_summary(out_im, sess.run
                    (global_step))
             val_writer.add_summary(val_summary, sess.run
                    (global_step))
             print "Validation Loss:", validation_loss
             saver.save(sess,
        "mnist_autoencoder_hidden=" + n_code +
        "_logs/model-checkpoint-"
             + '%04d' % (epoch+1),
        global_step=global_step)
        print "Optimization Finished!"
        test_loss = sess.run(eval_op, feed_dict={x:
  mnist.test.images, phase_train: False})
             print "Test Loss:", loss
```

При помощи `TensorBoard` можно визуализировать граф `TensorFlow`, ошибку на обучающем и проверочном наборах данных и информацию по изображениям. Выполните следующую команду:

```
$ tensorboard --logdir ~/path/to/mnist_autoencoder_hidden=2_logs
```

Затем наберите в браузере *http://localhost:6006/*. Результаты из вкладки `Graph` показаны на рис. 6.6.

Мы удачно организовали пространство имен компонентов нашего графа `TensorFlow`, что способствовало хорошей организации всей модели. Можно легко переходить от одного компонента к другому и углубляться дальше, отслеживая движение данных по разным слоям кодера и декодера, чтение оптимизатором выходных данных модуля обучения и влияние градиентов на все компоненты модели.

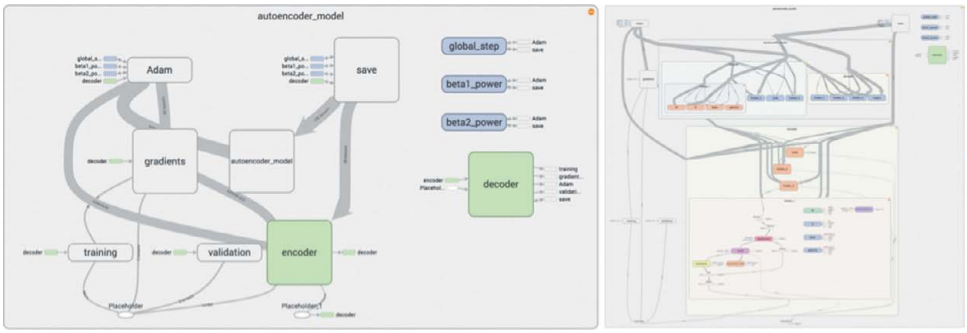


Рис. 6.6. *TensorFlow позволяет в сжатом виде наблюдать основные компоненты и потоки данных нашего графа вычислений (слева), а также более подробно изучить потоки данных индивидуальных подкомпонентов (справа)*

Мы также визуализируем ошибку на обучающем (после каждого мини-пакета) и проверочном (после каждой эпохи) наборах данных, внимательно следя за кривыми, чтобы выявить возможное переобучение. Визуализации ошибки в `TensorBoard` приведены на рис. 6.7. Как и можно ожидать от успешной модели, и обучающий, и проверочный графики ошибки убывают, пока не выпрямляются асимптотически. После примерно 200 эпох мы добиваемся ошибки на проверочном множестве 4,78. Хотя кривые выглядят многообещающе, сложно сразу понять, достигли ли мы плато и величина ошибки «хорошая» или наша модель пока плохо воссоздает входные данные.

Чтобы понять, что это значит, рассмотрим набор данных `MNIST`. Возьмем произвольное изображение `1` из набора и назовем его *X*. На рис. 6.8 мы сравниваем его со всеми остальными в наборе. Для каждого класса цифр мы вычисляем среднее значение `L2`, сравнивая *X* с каждым примером класса цифр. Для визуализации включаем также среднее всех примеров каждого класса цифр.

X отстоит в среднем на 5,75 пункта от остальных единиц в `MNIST`. Если говорить о расстоянии `L2`, то ближайшие к *X* цифры из не-единиц — семерки (8,94 пункта), а самые удаленные — нули (11,05 пункта). Эти показатели дают понять, что средняя ошибка автокодера 4,78 — высококачественная реконструкция.

Поскольку мы собираем результаты по изображениям, эту гипотезу можно подтвердить напрямую, рассмотрев сами входные изображения и их реконструкции. Реконструкции трех случайным образом выбранных образцов из тестового набора показаны на рис. 6.9.

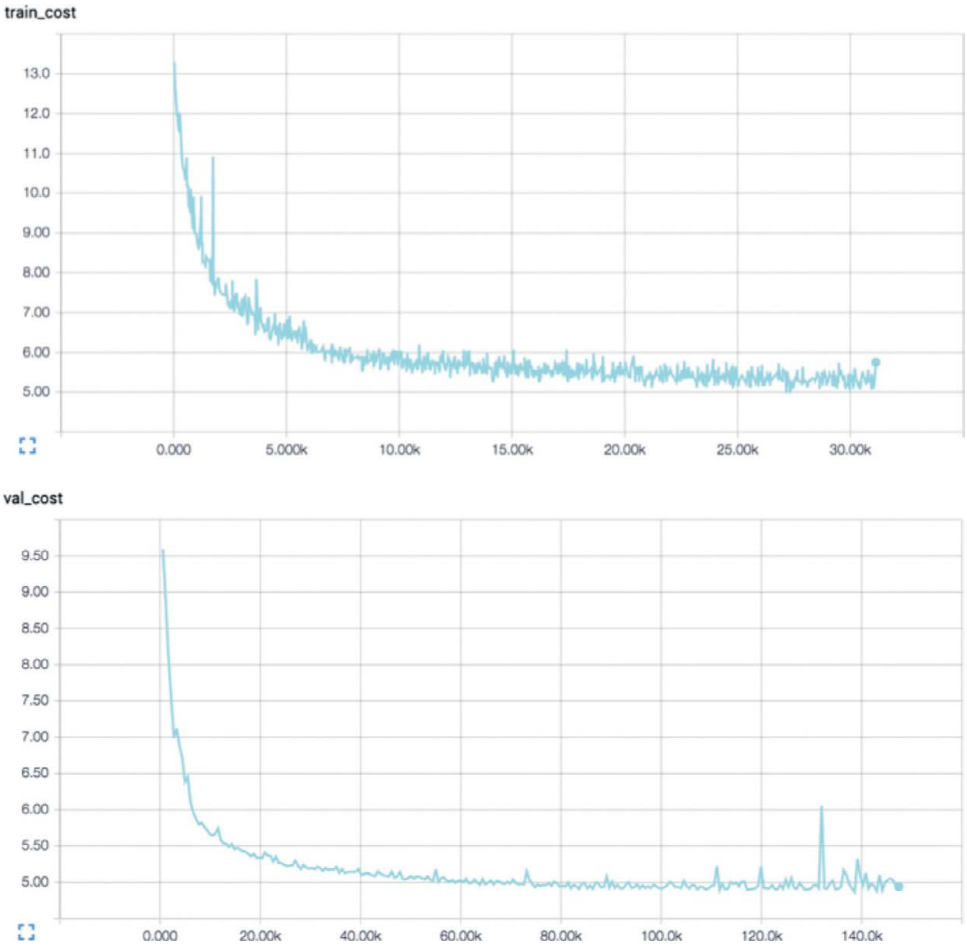


Рис. 6.7. *Ошибка на обучающем (записаны после каждого мини-пакета) и проверочном наборах (записаны после каждой эпохи)*

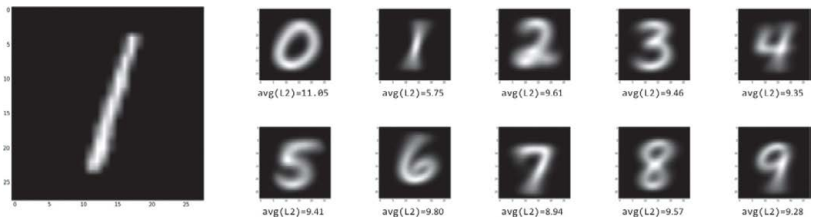


Рис. 6.8. *Изображение 1 слева сравнивается со всеми остальными цифрами в наборе данных MNIST; каждый класс цифр визуально представлен средним всех своих членов и помечен средним значением L2, единица слева сравнивается со всеми членами класса*

Рис. 6.9. *Сравнение входных данных из проверочного набора и их реконструкций после 5, 100 и 200 эпох обучения*

После пяти эпох автокодер уже начинает улавливать примечательные черты исходного изображения, но по большей части реконструкции — туманный набор похожих цифр. После 100 эпох уверенно опознаются 0 и 4, но у автокодера остаются проблемы выбора между 5, 3 и, возможно, 8. После 200 эпох уже видно, что и эта неоднозначность устранена и все цифры распознаются четко.

Закончим раздел сравнением двумерных кодов, выданных традиционным методом главных компонент и автокодерами. Мы хотим показать, что автокодеры обеспечивают лучшую визуализацию, особенно разграничение разных классов цифр, чем метод главных компонент. Сначала быстро рассмотрим код, который используем для получения двумерных кодов методом главных компонент:

```
from sklearn import decomposition
import input_data
mnist = input_data.read_data_sets("data/", one_hot=False)
pca = decomposition.PCA(n_components=2)
pca.fit(mnist.train.images)
pca_codes = pca.transform(mnist.test.images)
```

Загрузим набор данных MNIST. Мы установили флаг one_hot=False, поскольку хотим, чтобы метки были представлены как целые числа, а не прямые унитарные векторы (напомним, что прямой унитарный вектор, отображающий метку MNIST, — вектор размера 10 с i-м компонентом, равным 1, для отображения цифры i, а остальные компоненты равны 0). Возьмем привычную библиотеку машинного обучения scikit-learn и выполним PCA, установив n_components=2, чтобы библиотека генерировала двумерные коды. Можно воссоздать исходные изображения из двумерных кодов и визуализировать реконструкции:

```
from matplotlib import pyplot as plt
pca_recon = pca.inverse_transform(pca_codes[:1])
plt.imshow(pca_recon[0].reshape((28,28)), cmap=plt.cm.gray)
plt.show()
```

Этот фрагмент кода показывает, как визуализировать первое изображение в тестовом наборе данных, но легко модифицировать код, чтобы визуализировать любой произвольный их поднабор. Сравнение реконструкции метода главных компонент с реконструкциями автокодера на рис. 6.10 сразу дает понять, что автокодер значительно лучше PCA в работе с двумерными кодами. Результаты метода главных компонент хоть как-то приближены к результатам автокодера, который прошел только пять эпох обучения. У PCA большие проблемы с различением 5, 3 и 8, 0 и 8, 4 и 9. Повторение того же эксперимента с 30-мерными кодами дает значительное улучшение результатов PCA, но они все равно значительно хуже, чем результаты 30-мерного автокодера.

Рис. 6.10. *Сравнение реконструкций метода главных компонент и автокодера*

Чтобы завершить эксперимент, нужно загрузить сохраненную модель TensorFlow, получить двумерные коды и вывести коды как метода главных компонент, так и автокодера. Мы должны аккуратно перестроить граф TensorFlow именно так, как он выглядел во время обучения.

Мы задаем путь к контрольной точке модели, которую сохранили во время обучения как аргумент командной строки в скрипте. Наконец, используем собственную функцию, которая строит график с легендой и соответственно раскрашенными точками для данных разных классов цифр:

```
import tensorflow as tf
import autoencoder_mnist as ae
import argparse
def scatter(codes, labels):
      colors = [
            ('#27ae60', 'o'),
            ('#2980b9', 'o'),
            ('#8e44ad', 'o'),
            ('#f39c12', 'o'),
            ('#c0392b', 'o'),
            ('#27ae60', 'x'),
            ('#2980b9', 'x'),
            ('#8e44ad', 'x'),
            ('#c0392b', 'x'),
            ('#f39c12', 'x'),
      ]
      for num in xrange(10):
            plt.scatter([codes[:,0][i] for i in xrange(len
                  (labels)) if labels[i] == num],
            [codes[:,1][i] for i in xrange(len(labels)) if
                                    labels[i] == num], 7,
            label=str(num), color = colors[num][0],
                  marker=colors[num][1])
      plt.legend()
      plt.show()
with tf.Graph().as_default():
            with tf.variable_scope("autoencoder_model"):
            x = tf.placeholder("float", [None, 784])
            phase_train = tf.placeholder(tf.bool)
```

```
            code = ae.encoder(x, 2, phase_train)
            output = ae.decoder(code, 2, phase_train)
            cost, train_summary_op = ae.loss(output, x)
            global_step = tf.Variable(0, name='global_step',
      trainable=False)
            train_op = ae.training(cost, global_step)
eval_op, in_im_op, out_im_op, val_summary_op =
            ae.evaluate(output, x)
            saver = tf.train.Saver()
            sess = tf.Session()
            sess = tf.Session()
            saver = tf.train.Saver()
            saver.restore(sess, args.savepath[0])
            ae_codes= sess.run(code, feed_dict={x:
      mnist.test.images, phase_train: True})
            scatter(ae_codes,
            mnist.test.labels)
            scatter(pca_codes, mnist.test.labels)
```

На визуализации (рис. 6.11) найти четкие кластеры для двумерных кодов метода главных компонент сложно, а автокодер прекрасно с этим справился, собрав вместе коды разных классов цифр.

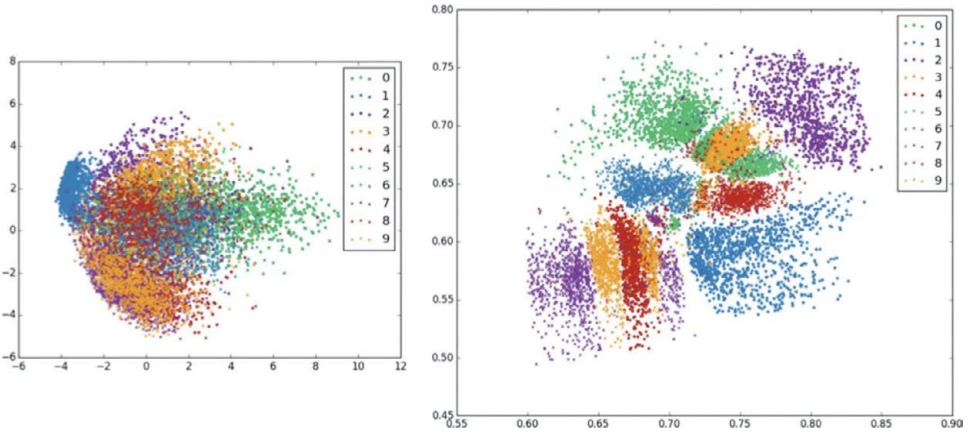


Рис. 6.11. *Визуализируем двумерные представления метода главных компонент (слева) и автокодера (справа). Заметьте, что автокодер гораздо лучше группирует коды разных классов цифр*

Простая модель машинного обучения способна классифицировать точки данных, состоящие из плотных представлений автокодера, гораздо эффективнее, чем из плотных представлений метода главных компонент.

Мы успешно создали и обучили автокодер с прямым распространением сигнала и показали, что полученные плотные векторные представления работают лучше, чем в методе главных компонент, классическом подходе к уменьшению размерности. В следующем разделе мы рассмотрим понятие шумопонижения, которое служит формой регуляризации, делая наши плотные векторные представления более эффективными.

Шумопонижение для повышения эффективности плотных векторных представлений

Рассмотрим дополнительный механизм *шумопонижения*, повышающий способность автокодера порождать устойчивые к шуму плотные векторные представления. Человеческие органы восприятия удивительно устойчивы к шуму.

Возьмем, например, рис. 6.12. Хотя я исказил половину пикселов в каждом изображении, у вас едва ли возникли проблемы с распознаванием цифр. Даже те, которые легко перепутать (например, 2 и 7), вполне различимы.

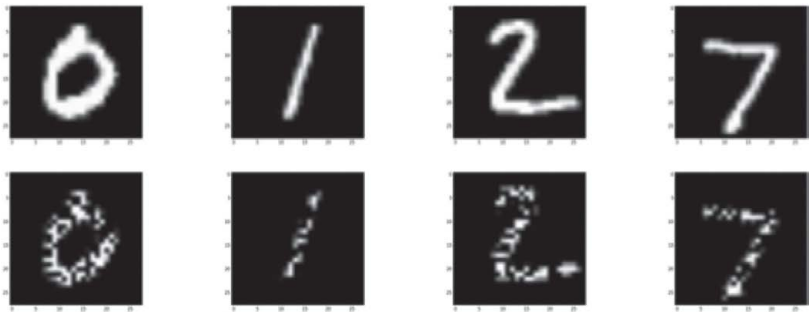


Рис. 6.12. *В верхнем ряду — исходные изображения из набора данных MNIST. В нижнем мы случайным образом затемнили половину пикселов. Однако человек способен различить цифры в этом ряду*

Один из подходов к этому явлению — вероятностный. Даже если мы сталкиваемся со случайным удалением фрагментов изображения, при достаточном количестве информации мозг все равно обычно способен уяснить смысл. Он буквально заполняет пустоты и делает вывод.

Даже если на сетчатку поступает искаженный вариант цифры, мозг все равно способен воссоздать серию активаций (например, код или плотное векторное представление), которую мы обычно используем для представления изображения этого символа. Это свойство можно попробовать воссоздать в алгоритме для плотного векторного представления. Оно впервые было исследовано Паскалем Винсентом и его коллегами в 2008 году, когда они представили *шумопонижающий автокодер*[67].

Основные принципы шумопонижения просты. Мы искажаем определенный процент пикселов входного изображения, назначая им нулевое значение. Если исходное изображение — X, искаженный вариант назовем $C(X)$. Шумопонижающий автокодер идентичен обычному, за исключением одной детали: в его сеть поступает искаженное изображение $C(X)$, а не X. Автокодер вынужден обучаться для каждого входящего изображения коду, устойчивому к механизму искажения и способному строить интерполяции отсутствующей информации, воссоздавая неповрежденное изображение.

Этот процесс можно представить и геометрически. Допустим, у нас есть двумерный набор данных с разными метками. Возьмем все точки данных в определенной категории (например, с фиксированной меткой) и назовем их S. Любое произвольное семплирование точек может вылиться в любую форму визуализации, но мы предполагаем, что для реальных категорий есть системообразующие структуры, объединяющие все точки S. И они называются *многообразием*.

Многообразие — форма, которую мы хотим сохранить при уменьшении размерности данных; как писали в 2013 году Ален и Бенджо[68], автокодер неявно усваивает его, обучаясь воссоздавать данные после их прохождения через узкое место нейронной сети (слой кода). Автокодер может понять, к какому из многообразий принадлежит точка, проводя реконструкцию для случая с потенциально разными метками.

Рассмотрим сценарий с рис. 6.13, где точки S — простое малоразмерное многообразие (здесь окружность, выделенная черным). На рис. А мы видим точки данных в S, помеченные черными крестиками, и многообразие, которое оптимально их описывает. Также мы видим аппроксимацию нашей операции искажения.

Красная стрелка и сплошная красная окружность показывают все пути, которыми искажение могло изменить точку данных. Поскольку мы применяем его к каждой точке (на всем многообразии), она искусственно расширяет набор данных, который теперь включает не только многообразие, но и все точки в пространстве вокруг него вплоть до максимального предела ошибки. Последний показан пунктирными красными окружностями

на рис. A, а расширение набора данных — красными крестиками на рис. B. Автокодеру надо учиться сжимать все точки данных в этом пространстве до многообразия. Иными словами, обучаясь тому, какие аспекты точки данных обобщаемы и важны, а какие представляют собой «шум», шумопонижающий автокодер обучается аппроксимировать многообразие S.

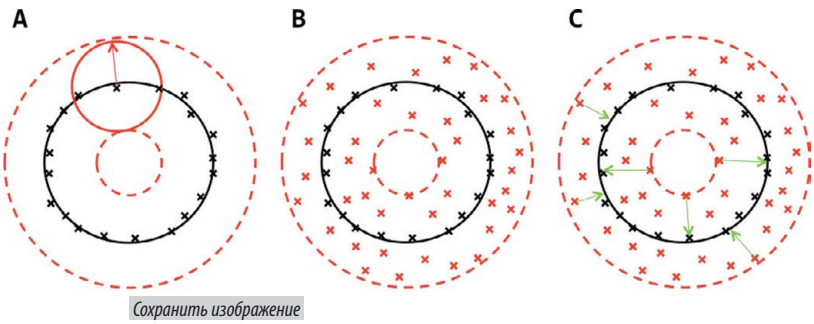


Рис. 6.13. *Шумопонижение позволяет модели обучиться многообразию (черная окружность), поняв, как соотнести поврежденные данные (красные крестики) с неповрежденными (черные крестики) при минимизации ошибок (зеленые стрелки) между их представлениями*

Сформировав философскую мотивацию для шумопонижения, мы можем внести небольшие изменения в скрипт и создать шумопонижающий автокодер:

```
def corrupt_input(x):
      corrupting_matrix = tf.random_uniform(shape=tf.shape(x),
                          minval=0,maxval=2,dtype=tf.int32)
      return x * tf.cast(corrupting_matrix, tf.float32)
x = tf.placeholder("float", [None, 784])   # mnist data image of
                                           # shape 28*28=784
corrupt = tf.placeholder(tf.float32)
phase_train = tf.placeholder(tf.bool)
c_x = (corrupt_input(x) * corrupt) + (x * (1 — corrupt))
```

Этот фрагмент кода искажает входные данные, если заполнитель `corrupt` равен 1, и воздерживается от искажения, если он равен 0. Внеся это изменение, можно снова запустить автокодер и получить реконструкции, показанные на рис. 6.14. Очевидно, что шумопонижающий автокодер достоверно воспроизвел невероятную человеческую способность к дополнению недостающих фрагментов.

Рис. 6.14. *Мы применяем операцию искажения к набору данных и обучаем шумопонижающий автокодер воссоздавать исходные, неискаженные изображения*

Разреженность в автокодерах

Один из самых сложных аспектов глубокого обучения — проблема *интерпретируемости*. Это свойство модели машинного обучения, которое измеряет, насколько легко отслеживать и объяснять ее процессы и/или результаты. Интерпретировать глубокие модели обычно сложно из-за нелинейностей и большого числа параметров. Такие модели обычно точнее, но недостаточная интерпретируемость часто затрудняет их применение в очень важных, но рискованных областях. Например, если модель машинного обучения выдает, есть ли у пациента рак, врачу, вероятно, потребуется объяснение, чтобы подтвердить заключение.

Интерпретируемость можно повысить, исследуя характеристики выходных данных автокодера. В целом его представления достаточно плотные, и это влияет на то, как они изменяются при внесении поправок во входные данные. Рассмотрим ситуацию на рис. 6.15.

Автокодер выдает *плотное* векторное представление, то есть такое, в котором исходное изображение сильно сжато.

Поскольку мы можем работать только с измерениями, содержащимися в представлении, его активации содержат информацию из многих источников, и распутать результат крайне сложно. При добавлении или удалении

компонентов представление меняется непредсказуемо. Почти невозможно интерпретировать то, как и почему оно получается именно таким.

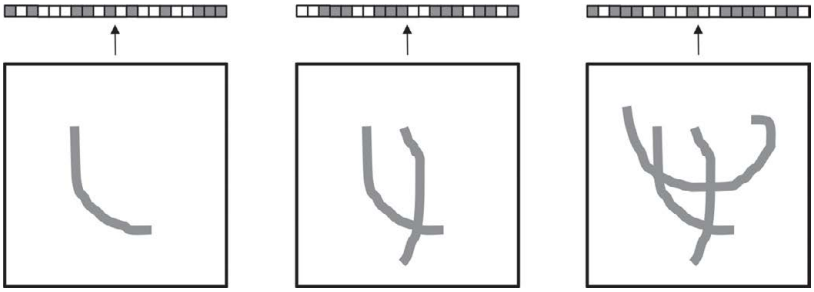


Рис. 6.15. *Активации плотного векторного представления сочетаются и перекрываются, информацию о разных признаках трудно интерпретировать*

Идеальным исходом для нас была бы возможность создания представления с четким соответствием 1 к 1 (или близко к этому) между высокоуровневыми признаками и индивидуальными компонентами кода. Сумев этого добиться, мы близко подойдем к системе, описанной на рис. 6.16. На рис. А показано, как меняется представление при добавлении и вычитании компонентов, а на В цветом выделена связь между штрихами и компонентами кода. Здесь понятно, как и почему меняется представление: фактически это сумма отдельных штрихов в изображении.

Это идеальный вариант, но нужно тщательно продумать, какие механизмы использовать, чтобы добиться такой интерпретируемости изображения. Проблема, очевидно, в том, что слой кода — узкое место сети. К сожалению, простое повышение количества нейронов в слое не поможет. Мы можем увеличить размер слоя кода, но обычно нет механизма, который не позволяет каждому признаку, выбранному автокодером, незначительно влиять на множество компонентов. В крайнем случае, когда выбранные признаки сложнее и, следовательно, полезнее, размер слоя кода может быть даже больше, чем входных данных. Тогда модель способна в буквальном смысле выполнять операции «копирования», при которых слой кода не учится никаким полезным представлениям.

Нам нужно заставить автокодер использовать как можно меньше компонентов вектора плотного представления без ущерба для эффективности воссоздания входных данных. Это очень похоже на регуляризацию во избежание переобучения в простых нейронных сетях (см. главу 2), но на этот раз нужно, чтобы как можно больше компонентов векторного представления имели нулевое значение или близкое к нему.

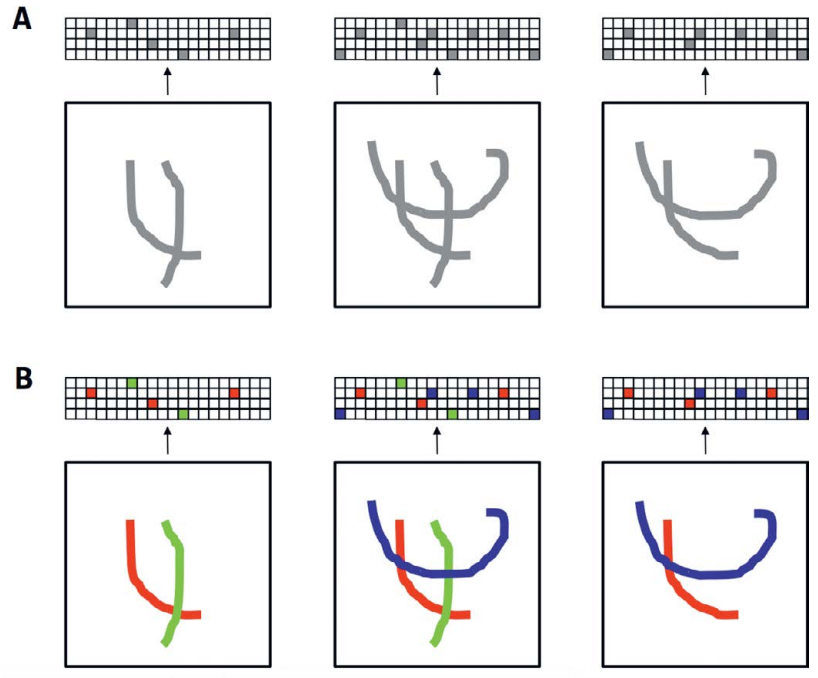


Рис. 6.16. *При грамотном сочетании пространства и разреженности представление лучше поддается интерпретации. На рис. А мы показываем, как активации представления меняются с добавлением и вычитанием штрихов. На рис. В мы кодируем цветом активации, которые соответствуют каждому штриху, чтобы показать возможность интерпретации воздействия конкретного штриха на представление*

Как и в главе 2, достичь этого можно, введя в целевую функцию модификатор разреженности (SparsityPenalty), который повышает стоимость любого представления, где многие компоненты не равны 0:

$$E_{\text{Sparse}} = E + \beta \cdot \text{SparsityPenalty}.$$

Значение β определяет, насколько мы стремимся к разреженности в ущерб качеству реконструкций. Математически подкованные читатели могут рассматривать значения каждого компонента всех отображений как результат случайной переменной с неизвестным средним. Затем мы задействуем меру расстояния, сравнивая распределение наблюдений этой случайной переменной (значений каждого компонента) и случайной переменной, среднее значение которой известно и равно 0. В данном случае

часто используется расстояние Кулльбака — Лейблера (KL). Дальнейшее рассмотрение разреженности в автокодере выходит за рамки этой книги, но приводится в работах Маркаурелио Ранзато (2007 и 2008)[69]. Теоретические свойства и эмпирическая эффективность введения функции-посредника до слоя кода, которая обнуляет все активации в представлении, кроме `k`, были исследованы в работе Алирезы Макзани и Брендана Фрея (2014)[70]. *k-Разреженные автокодеры* оказались столь же эффективны, как и остальные механизмы разрежения, несмотря на поразительную простоту реализации и понимания (а в плане вычислений их эффективность даже выше).

Мы выяснили, как использовать автокодеры для нахождения хороших представлений точек данных, выбирая основные характеристики их содержания. Такой механизм сокращения размерности эффективен, когда независимые точки данных содержательны — обладают всей релевантной информацией, относящейся к их структуре во входном представлении. Далее мы рассмотрим стратегии, которые можно использовать, когда основной источник информации — контекст точек данных, а не они сами.

Когда контекст информативнее, чем входной вектор данных

В предыдущих разделах мы в основном занимались вопросом сокращения размерности. В этом процессе обычно имеются богатые входные данные, содержащие много шума, который и нужно отделить от ключевой, структурной информации. Нам требуется выделить эту информацию, игнорируя вариации и шум, которые затрудняют фундаментальное понимание данных.

В других ситуациях представления входных данных очень мало говорят о содержании, которое мы пытаемся вычленить. Тогда наша цель — не извлечь информацию, а выбрать ее из контекста для создания полезных представлений.

Возможно, пока эти рассуждения кажутся слишком абстрактными, так что перейдем к конкретике и реальным примерам.

Построение моделей для естественного языка — задача очень сложная. Первая проблема, которую предстоит решить при построении языковых моделей, — как найти хороший способ представления отдельных слов. На первый взгляд не вполне понятно, как же создать хорошее представление. Начнем с наивного подхода, пример которого приведен на рис. 6.17.

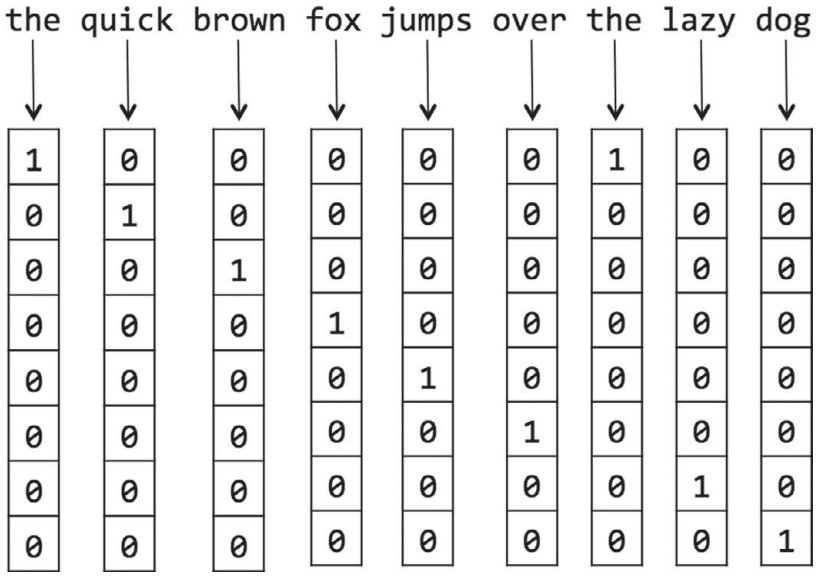


Рис. 6.17. *Пример генерации прямых унитарных векторных представлений слов для простого документа*

Если словарь документа равен V и насчитывает $|V|$ слов, мы можем представить их в виде прямых унитарных векторов. У нас будут $|V|$-мерные векторы представления, и каждое слово будет ассоциироваться с индексом в этом векторе. Для представления уникального слова w_i мы задаем для i-го компонента вектора значение 1, а для всех остальных — 0.

Но такая схема представления выглядит почти случайной. Векторизация не преобразует похожие слова в похожие векторы. И это проблема, ведь мы хотим, чтобы наши модели понимали, что слова `jump` и `leap` («прыгать» и «скакать») очень похожи по значению, и умели различать часть речи: например, глагол, существительное и предлог.

Примитивная прямая унитарная векторная кодировка слов не сохраняет ни одной из этих характеристик. Нужно найти какие-то способы выявления таких отношений и векторного кодирования этой информации.

Оказывается, отношения между словами можно определить с помощью анализа контекста. Например, такие синонимы, как `jump` и `leap`, часто взаимозаменяемы в одном и том же контексте. И оба обычно применяются, когда субъект совершает действие над прямым объектом. Этим принципом мы будем руководствоваться всегда, сталкиваясь с новым словом. Например, прочтя предложение «Милитарист спорил с толпой», мы можем

сразу сделать кое-какие выводы по поводу слова «милитарист», даже не зная его значения. В этом контексте оно предшествует глаголу; это позволяет предположить, что «милитарист» — существительное и подлежащее в предложении.

Идем дальше. «Милитарист» «спорил», отсюда логично заключить, что это некто агрессивный или скандальный. И мало-помалу, как показано на рис. 6.18, анализ контекста (фиксированного окна слов, окружающего искомое) позволяет быстро установить значение слова.

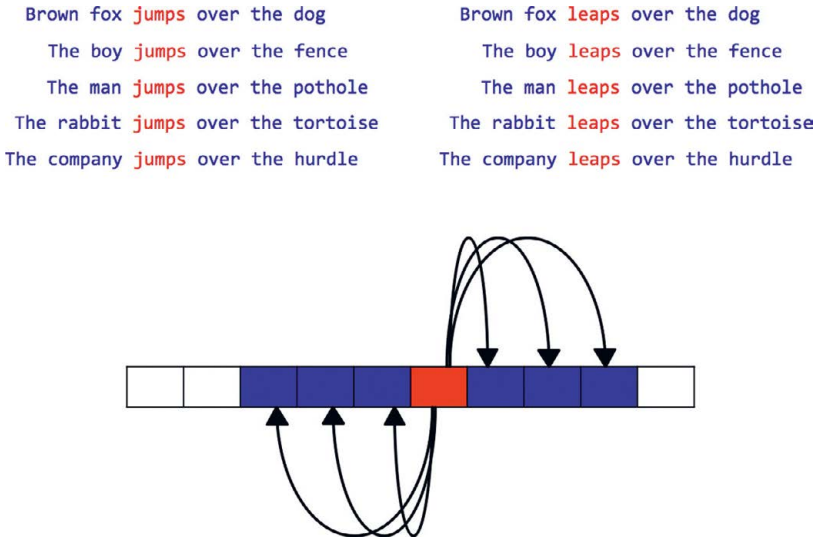


Рис. 6.18. *На основе контекста можно определить слова со схожими значениями. Например, jumps и leaps должны иметь сходное векторное представление, поскольку почти взаимозаменяемы. Можно даже сделать выводы об их значениях, просто посмотрев на окружающие их слова*

Оказывается, основания, которыми мы руководствовались при создании автокодера, можно применить и здесь, построив сеть, которая создает мощные, распределенные представления. Две основные стратегии показаны на рис. 6.19. Один из возможных методов (рис. A) передает целевое слово по кодирующей сети, которая создает плотное векторное представление. Затем последнее принимается декодирующей сетью, но она не пытается воссоздать исходное значение, как в автокодере, а стремится подобрать слово по контексту. Второй возможный метод (рис. B) строго обратный: кодер берет слово из контекста как входные данные, порождая целевое слово.

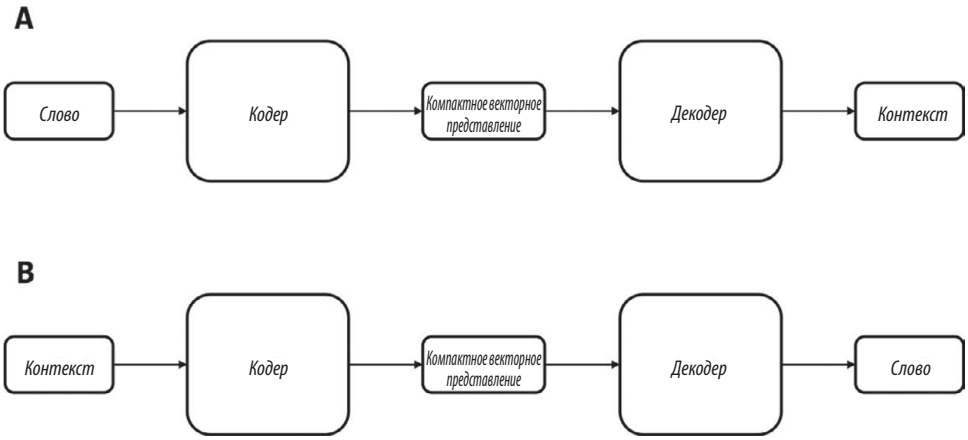


Рис. 6.19. *Общие архитектуры разработки кодеров и декодеров, которые генерируют компактные векторные представления, связывая слова с их контекстами (A) или наоборот (B)*

В следующем разделе мы расскажем, как применить эту стратегию (с небольшими улучшениями), чтобы получить компактные векторные представления слов на практике.

Технология Word2Vec

`Word2Vec`, технология генерации компактных векторных представлений слов, разработана Томашем Миколовым и его коллегами. В их работе приводились две стратегии генерации представлений, очень похожие на стратегии кодирования контекста, о которых мы говорили в предыдущем разделе.

Первый алгоритм `Word2Vec`, который ввели Миколов и коллеги, назывался моделью «непрерывного мешка со словами» (*`Continuous Bag of Words`* (`CBOW`))[71]. Эта модель во многом напоминает стратегию B из предыдущего раздела.

`CBOW` при помощи кодера создает плотное векторное представление из полного контекста (который рассматривается как единица входных данных) и предсказывает целевое слово. Как выяснилось, она лучше всего работает на небольших наборах данных, о чем говорилось в исходной статье.

Второй алгоритм `Word2Vec` — модель *`Skip-Gram`*, тоже предложенная Миколовым и коллегами[72]. Это инвертированный вариант `CBOW`: исполь-

зуя текущее слово, она пытается предсказать одно из слов контекста. Рассмотрим на примере, как выглядит набор данных для Skip-Gram.

Возьмем предложение The boy went to the bank. Разобьем его на пары (контекст, цель) и получим [([the, went], boy), ([boy, to], went), ([went, the], to), ([to, bank], the)]. Теперь каждую пару (контекст, цель) нужно разбить на пары (вход, выход), где вход — цель, а выход — одно из слов из контекста. Из первой пары ([the, went], boy) получаем (boy, the) и (boy, went). Продолжая применять эти операции к каждой паре (контекст, цель), мы получим набор данных. Наконец, мы меняем каждое слово его уникальным индексом $i \in \{0, 1, \ldots, |V| - 1\}$, соответствующим индексу в словаре.

Структура кодера удивительно проста. По сути, это таблица соответствия с V рядов, из них i-й — компактное векторное представление, соответствующее i-му слову словаря. Кодеру достаточно взять индекс входящего слова и вернуть соответствующий ряд из таблицы. Это эффективная операция, на GPU она может быть представлена в виде умножения транспонированной таблицы и прямого унитарного вектора, представляющего входное слово. Это легко реализуется в TensorFlow при помощи следующей функции:

```
tf.nn.embedding_lookup(params, ids, partition_strategy='mod',
                       name=None, validate_indices=True),
```

где params — матрица плотных векторных представлений, а ids — тензор индексов, которые мы ищем. Подробную информацию о дополнительных параметрах можно найти в документации TensorFlow API[73].

Декодер сложнее, поскольку мы внесем несколько изменений для повышения производительности. Наивный вариант — попытаться воссоздать прямой унитарный кодирующий вектор для выходного значения, который можно реализовать в обычном слое с прямым распространением сигнала, дополненном слоем с функцией мягкого максимума. Но это неэффективно, поскольку придется рассчитывать распределение вероятности по всему пространству словаря.

Чтобы сократить число параметров, Миколов и коллеги воспользовались иной стратегией реализации декодера с названием «шумно-контрастная оценка» (noise-contrastive estimation, NCE). Эта стратегия показана на рис. 6.20.

В стратегии NCE таблица соответствия используется для того, чтобы найти плотное векторное представление для выходного слова, а также представлений для случайно выбранных из словаря слов, отсутствующих в контексте

входного. Применяем модель бинарной логистической регрессии, которая поочередно берет представление входного слова и выходного либо случайно выбранного и выдает значение от 0 до 1, показывающее вероятность того, что сравниваемое представление относится к слову, присутствующему в контексте входного. Берем сумму вероятностей, соответствующих неконтекстным сравнениям, и вычитаем вероятность, сопоставленную контекстному сравнению.

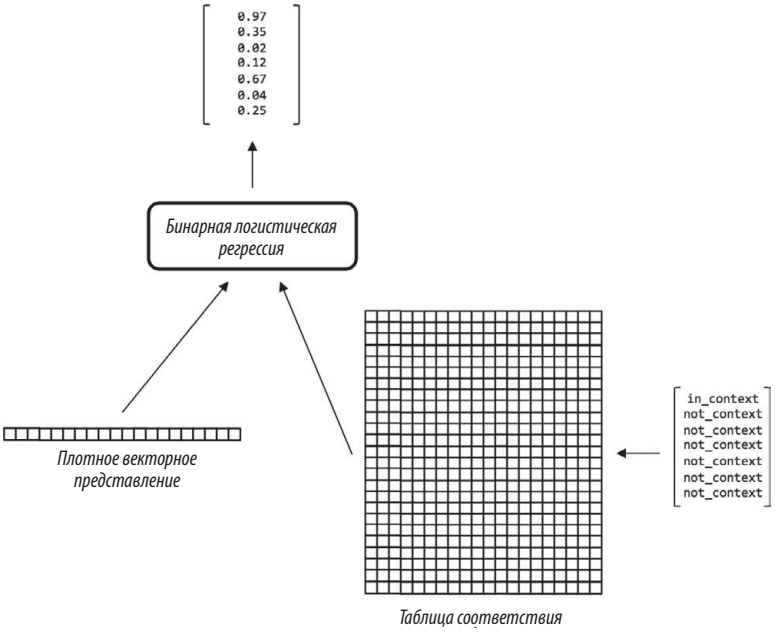


Рис. 6.20. *Иллюстрация работы шумно-контрастной оценки. Бинарная логистическая регрессия сравнивает представление целевого слова с представлением слова из контекста и случайным образом выбранных неконтекстных слов. Строим функцию потерь, которая описывает, насколько эффективно векторное представление позволяет распознать слова в контексте целевого по отношению к словам вне контекста*

Это целевая функция, которую мы желаем минимизировать (при оптимальном сценарии, в котором производительность модели идеальна, значение будет равно –1). `NCE` в `TensorFlow` реализуется следующим фрагментом кода:

```
tf.nn.nce_loss(weights, biases, inputs, labels, num_sampled,
       num_classes, num_true=1, sampled_values=None,
```

```
        remove_accidental_hits=False, partition_strategy=
              'mod',
        name='nce_loss')
```

Веса (weights) должны иметь ту же размерность, что и матрица плотных векторных представлений, а смещения (biases) — быть тензорами с размером, эквивалентным размеру словаря. Входные значения (inputs) — результаты из таблицы представлений, `num_sampled`, число отрицательных образцов, использованных при вычислении `NCE`, а `num_classes` — размер словаря.

Хотя `Word2Vec` не относится к моделям глубокого обучения, мы обсуждаем этот инструмент по ряду причин. Во-первых, он представляет стратегию (нахождение плотных векторных представлений по контексту), которая обобщается до многих моделей глубокого обучения. Когда в главе 7 мы будем рассматривать модели для анализа последовательностей, мы увидим реализацию этой стратегии для порождения векторов с компактным представлением предложений. Более того, когда мы начнем строить все больше языковых моделей, окажется, что плотные векторные представления `Word2Vec` для слов обеспечивают гораздо лучшие результаты, чем прямые унитарные векторы.

Теперь понятно, как строить модель `Skip-Gram`, мы оценили ее важность. Пора реализовать ее в `TensorFlow`.

Реализация архитектуры Skip-Gram

Чтобы построить набор данных для модели `Skip-Gram`, воспользуемся модифицированной версией программы чтения данных в `TensorFlow Word2Vec` — `input_word_data.py`. Для начала зададим пару важных параметров для обучения и регулярной проверки модели. Мы берем мини-пакет из 32 примеров и обучаем в течение пяти эпох (полных проходов по всему набору данных). Будем пользоваться плотными векторными представлениями размера 128. Зададим контекстное окно в пять слов слева и справа от каждого текущего и семплируем четыре контекстных слова из этого окна. Наконец, возьмем 64 случайно выбранных неконтекстных слова для `NCE`.

Реализация слоя, строящего плотные векторные представления, не особенно сложна. Достаточно инициализировать таблицу матрицей значений:

```
def embedding_layer(x, embedding_shape):
      with tf.variable_scope("embedding"):
            embedding_init = tf.random_uniform(embedding_shape,
```

```
        -1.0, 1.0)
            embedding_matrix = tf.get_variable("E",
                    initializer=embedding_init)
            return tf.nn.embedding_lookup(embedding_matrix, x),
                    embedding_matrix
```

Воспользуемся встроенным в `TensorFlow` вариантом `tf.nn.nce_loss` для вычисления потерь `NCE` на каждом обучающем примере, а затем соберем все результаты по мини-пакету в единый показатель:

```
def noise_contrastive_loss(embedding_lookup, weight_shape,
                           bias_shape, y):
    with tf.variable_scope("nce"):
    nce_weight_init = tf.truncated_normal(weight_shape,
                        stddev=1.0/(
        weight_shape[1])**0.5)
    nce_bias_init = tf.zeros(bias_shape)
    nce_W = tf.get_variable("W",
        initializer=nce_weight_init)
    nce_b = tf.get_variable("b", initializer=nce_bias_init)
    total_loss = tf.nn.nce_loss(nce_W, nce_b,
                embedding_lookup,
                        y, neg_size,
                data.vocabulary_size)
    return tf.reduce_mean(total_loss)
```

Наша целевая функция выражена как среднее потерь `NCE`, и мы запускаем обучение обычным путем. Здесь мы следуем по стопам Миколова и коллег и задействуем стохастический градиентный спуск с темпом обучения 0,1:

```
def training(cost, global_step):
    with tf.variable_scope("training"):
        summary_op = tf.scalar_summary("cost", cost)
        optimizer = tf.train.GradientDescentOptimizer(
                learning_rate)
        train_op = optimizer.minimize(
                cost, global_step=global_step)
        return train_op, summary_op
```

Также мы регулярно проверяем модель с помощью функции, которая нормализует плотные векторные представления в таблице и использует

косинусную меру близости, чтобы вычислить расстояния для набора проверочных слов до всех остальных в словаре:

```
def validation(embedding_matrix, x_val):
      norm = tf.reduce_sum(embedding_matrix**2, 1,
            keep_dims=True)**0.5
      normalized = embedding_matrix/norm
      val_embeddings = tf.nn.embedding_lookup(normalized, x_val)
      cosine_similarity = tf.matmul(val_embeddings, normalized,
            transpose_b=True)
      return normalized, cosine_similarity
```

Сведя вместе все компоненты, мы готовы запустить модель `Skip-Gram`. Этот фрагмент кода даем без комментариев, ведь он очень похож на прежние модели. Единственная разница — дополнительный код на шаге проверки. Мы случайным образом выбираем 20 слов из 500 самых распространенных в словаре из 10 000 единиц. Для каждого мы используем построенную ранее функцию косинусной меры близости, чтобы найти ближайших соседей:

```
if __name__ == '__main__':
      with tf.Graph().as_default():
            with tf.variable_scope("skipgram_model"):
                  x = tf.placeholder(tf.int32, shape=[batch_size])
                  y = tf.placeholder(tf.int32, [batch_size, 1])
                  val = tf.constant(val_examples, dtype=tf.int32)
                  global_step = tf.Variable(0, name='global_step',
                        trainable=False)
                  e_lookup, e_matrix =
                        embedding_layer(x,
                  [data.vocabulary_size, embedding_size])
                  cost = noise_contrastive_loss(e_lookup,
                  [data.vocabulary_size,
                        embedding_size],
                  [data.vocabulary_size], y)
                  train_op, summary_op = training(cost, global_step)
                  val_op = validation(e_matrix, val)
                  sess = tf.Session()
                  train_writer = tf.train.SummaryWriter(
                  "skipgram_logs/", graph=sess.graph)
```

```
init_op = tf.initialize_all_variables()
sess.run(init_op)
step = 0
avg_cost = 0
for epoch in xrange(training_epochs):
        for minibatch in xrange(batches_per_epoch):
        step +=1
        mbatch_x, mbatch_y = data.generate_batch(
        batch_size,
        num_skips, skip_window)
        feed_dict = {x : mbatch_x, y : mbatch_y}
        _, new_cost, train_summary = sess.run([
                train_op, cost,
                summary_op],
                feed_dict=feed_dict)
        train_writer.add_summary(train_summary,
                sess.run(global_step))
        # Compute average loss
        avg_cost += new_cost/display_step
        if step % display_step == 0:
        print "Elapsed:", str(step), "batches.
                Cost =",
        "{:.9f}".format(avg_cost)
        avg_cost = 0
        if step % val_step == 0:
        _, similarity = sess.run(val_op)
        for i in xrange(val_size):
        val_word = data.reverse_dictionary
        [val_examples[i]]
        neighbors = (-similarity[
        i, :]).argsort()
        [1:top_match+1]
        print_str = "Nearest neighbor of
        %s:"
        % val_word
        for k in xrange(top_match):
        print_str += " %s," %
        data.reverse_dictionary[
```

```
                    neighbors[k]]
                    print print_str[:-1]
                    final_embeddings, _ = sess.run(val_op)
```

Код начинает работу, а мы наблюдаем за развитием модели во времени. Сначала она плохо создает плотные векторные представления (что очевидно на шаге проверки). Однако к окончанию обучения модель явно находит представления, которые верно отражают значения отдельных слов:

```
ancient: egyptian, cultures, mythology, civilization, etruscan,
greek, classical, preserved
however: but, argued, necessarily, suggest, certainly, nor,
believe, believed
type: typical, kind, subset, form, combination, single,
description, meant
white: yellow, black, red, blue, colors, grey, bright, dark
system: operating, systems, unix, component, variant, versions,
version, essentially
energy: kinetic, amount, heat, gravitational, nucleus,
radiation, particles, transfer
world: ii, tournament, match, greatest, war, ever, championship,
cold
y: z, x, n, p, f, variable, mathrm, sum,
line: lines, ball, straight, circle, facing, edge, goal, yards,
among: amongst, prominent, most, while, famous, particularly,
argue, many
image: png, jpg, width, images, gallery, aloe, gif, angel
kingdom: states, turkey, britain, nations, islands, namely,
ireland, rest
long: short, narrow, thousand, just, extended, span, length,
shorter
through: into, passing, behind, capture, across, when, apart,
goal
i: you, t, know, really, me, want, myself, we
source: essential, implementation, important, software, content,
genetic, alcohol, application
because: thus, while, possibility, consequently, furthermore,
but, certainly, moral
eight: six, seven, five, nine, one, four, three, b
```

```
french: spanish, jacques, pierre, dutch, italian, du, english,
belgian
written: translated, inspired, poetry, alphabet, hebrew,
letters, words, read
```

Хотя результаты не идеальны, некоторые кластеры очень удачны. В одном оказались числа, страны и культурные явления. Местоимение `I` («я») попало к другим местоимениям. Слово `world` («мир») забавным образом оказалось рядом с `championship` («чемпионат») и `war` («война»). А `written` («письменный») попало по соседству с `translated` («переведенный»), `poetry` («поэзия»), `alphabet` («алфавит»), `letters` («буквы») и `words` («слова»). Закончить раздел мы хотим визуализацией плотных векторных представлений слов на рис. 6.21.

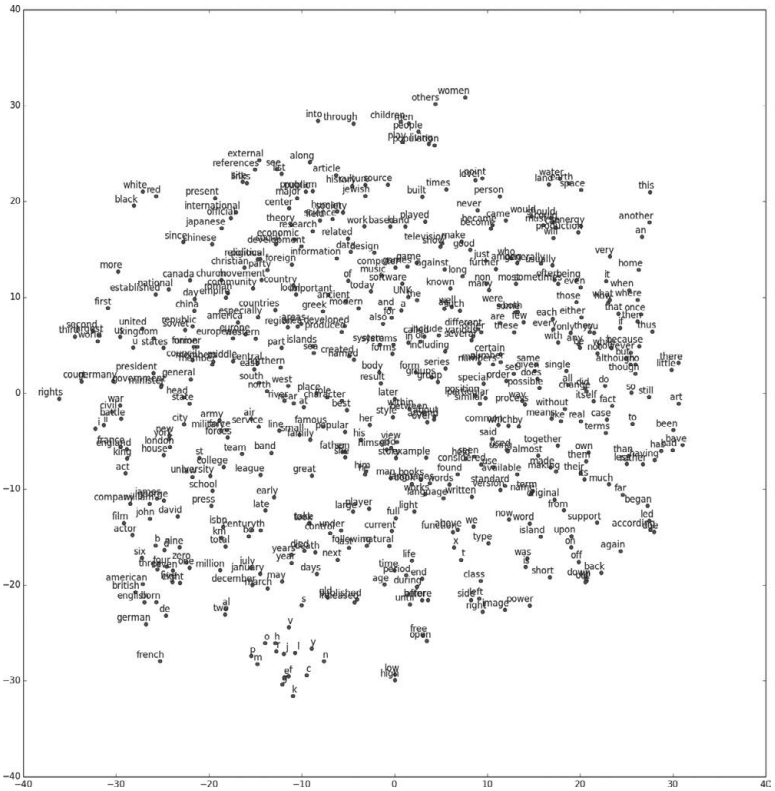


Рис. 6.21. *Визуализация наших представлений Skip-Gram при помощи t-SNE. Отмечаем, что близкие понятия действительно расположены ближе, чем разрозненные. Это показывает, что наши представления содержат значимую информацию о функциях и определениях отдельных слов*

Чтобы отобразить 128-мерные представления в двумерном пространстве, воспользуемся методом визуализации t-SNE. Если помните, мы использовали t-SNE и в главе 5 для визуализации отношений между изображениями в ImageNet. Пользоваться им несложно, этот метод имеет встроенную функцию в общепринятой библиотеке машинного обучения scikitlearn. Визуализацию можно создать при помощи следующего кода:

```
tsne = TSNE(perplexity=30, n_components=2, init='pca',
      n_iter=5000)
plot_embeddings = np.asfarray(final_embeddings[:plot_num,:],
      dtype='float')
low_dim_embs = tsne.fit_transform(plot_embeddings)
labels = [reverse_dictionary[i] for i in xrange(plot_only)]
data.plot_with_labels(low_dim_embs, labels)
```

Более подробное объяснение свойств плотных векторных представлений слов и интересных шаблонов (времена глаголов, страны и столицы, завершение аналогии и т. д.) любознательный читатель сможет найти в оригинальной работе Миколова и его коллег.

Резюме

В этой главе мы рассмотрели разные методы обучения представлений. Мы узнали, как эффективно снижать размерность при помощи автокодера. Мы познакомились с понятиями шумопонижения и разреженности, которые придают дополнительные полезные свойства автокодерам. Потом мы переключились на обучение представлений, при котором контекст входных данных информативнее, чем они сами. Мы научились порождать плотные векторные представления для английских слов при помощи модели Skip-Gram, что будет полезно при работе с моделями глубокого обучения для понимания языка. В следующей главе мы продолжим анализировать язык и другие последовательности методами глубокого обучения.

ГЛАВА 7

Модели анализа последовательностей

Сурья Бхупатираджу

Анализ данных переменной длины

До сих пор мы работали только с данными фиксированного размера — изображениями из `MNIST`, `CIFAR-10` и `ImageNet`. Эти модели очень хороши, но возникает немало ситуаций, в которых они недостаточны. Подавляющее большинство видов взаимодействия в повседневной жизни требует хорошего понимания последовательностей: это и чтение утренней газеты, и приготовление овсяной каши, и слушание радио, и просмотр презентации, и решение совершить операцию на фондовом рынке. Чтобы адаптироваться к данным переменной длины, нужно проявить чуть больше смекалки при создании моделей глубокого обучения.

На рис. 7.1 мы демонстрируем неудачи нейронной сети с прямым распространением сигнала при анализе последовательностей. Если она имеет тот же размер, что и входной слой, модель будет работать как ожидается. Можно даже работать с входными данными меньшего размера, дополнив их нулями до нужной длины. Но когда входные данные начинают превышать размер входного слоя, примитивное использование сети с прямым распространением сигнала становится невозможным.

Не все потеряно. В следующих двух разделах мы расскажем о нескольких стратегиях, которые можно применить, чтобы научить сети с прямым распространением сигнала обрабатывать последовательности. Далее мы рассмотрим ограничения этих методов и расскажем о новых архитектурах, призванных их устранить. Наконец, закончим повествованием о самых совершенных на данный момент архитектурах, позволяющих решать наиболее сложные проблемы в воссоздании человеческого логического мышления при работе с последовательностями.

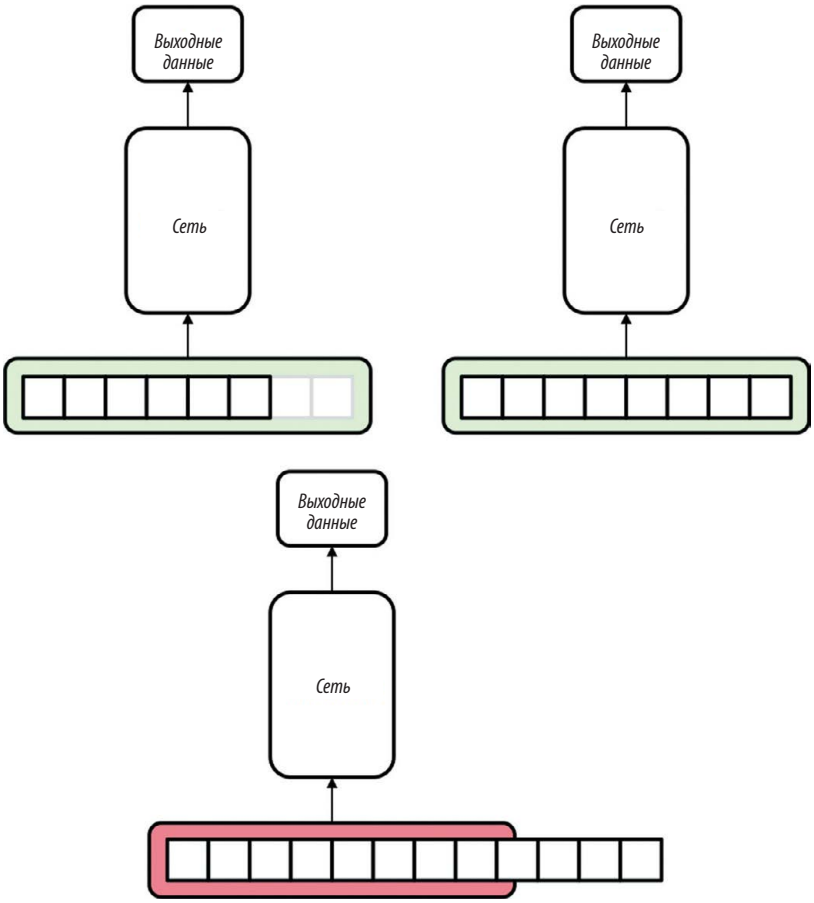


Рис. 7.1. *Сети с прямым распространением сигнала прекрасно решают проблемы с входными данными фиксированного размера. Дополнение нулями позволяет работать с данными меньшего размера, но при прямом использовании эти модели отказывают, когда входные данные начинают превышать фиксированный размер*

seq2seq и нейронные N-граммные модели

Рассмотрим архитектуру сети с прямым распространением сигнала, которая может обрабатывать текст и выдавать последовательность меток частей речи (`part-of-speech, POS`). Мы хотим присвоить ярлык каждому слову исходного текста, пометив его как существительное, глагол, предлог и т. д. Пример приведен на рис. 7.2. Это не так сложно, как создать искусственный интеллект, который сможет, изучив текст, ответить на вопросы о нем, но это серьезный первый шаг к разработке алгоритма, который

способен понять значение использованных в предложении слов. Задача интересна еще и тем, что представляет собой частный случай класса, именуемого *seq2seq*, где основная цель — трансформация входной последовательности в соответствующую выходную. Среди других знаменитых задач `seq2seq` — перевод текста с одного языка на другой (о чем мы поговорим позже), составление аннотации текста и распознавание устной речи.

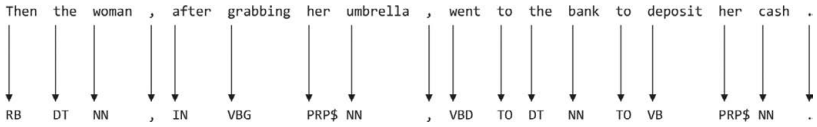


Рис. 7.2. *Пример точного определения частей речи для английского предложения*

Как мы уже говорили, не вполне понятно, как взять текст и сразу получить полную последовательность меток частей речи. Здесь используется хитрость, напоминающая способ разработки распределенных векторных представлений слов (см. главу 6).

Ключевое наблюдение: *чтобы определить часть речи любого слова, не обязательно учитывать долгосрочные зависимости*.

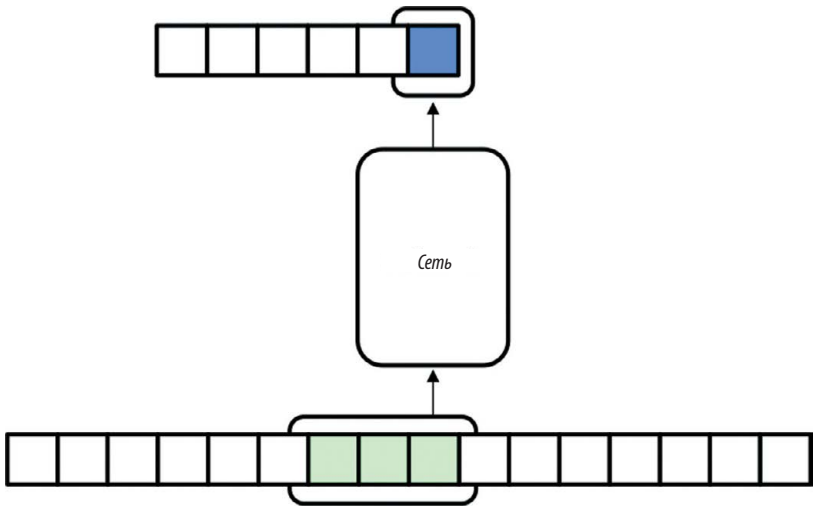


Рис. 7.3. *Использование сети с прямым распространением сигнала для seq2seq, когда можно игнорировать долгосрочные зависимости*

Вывод таков: вместо того чтобы брать все предложение для определения всех меток частей речи одновременно, можно присваивать метки поочередно, используя последовательности фиксированной длины. Так, мы

берем подпоследовательность, начинающуюся с интересующего нас слова и распространяющуюся на *n* предыдущих. Эта *нейронная N-грамм стратегия* изображена на рис. 7.3.

Так, когда мы определяем метку части речи для *i*-го слова во входных данных, мы используем в качестве этих данных слова *i* – *n* + 1-*e*, *i* – *n* + 2-*e*, … , *i*-*e*. Эту подпоследовательность мы называем *контекстным окном*. Для обработки всего текста мы сначала устанавливаем сеть в самом его начале. Затем мы перемещаем контекстное окно по слову за раз, определяя метку части речи для самого правого слова, пока не достигаем конца текста.

Применяя стратегию плотных векторных представлений слов из предыдущей главы, мы используем также плотные представления вместо прямых унитарных векторов. Это позволит нам сократить число параметров модели и ускорить процесс обучения.

Реализация разметки частей речи

Теперь, владея знаниями об архитектуре POS-сети, можно перейти к реализации. Сеть состоит из входного слоя, который использует контекстное окно 3-грамм. Мы применим 300-мерные векторные представления слов, что даст нам контекстное окно размерностью 900. Сеть с прямым распространением сигнала будет иметь два скрытых слоя, из 512 и 256 нейронов соответственно.

Выходной слой — функция мягкого максимума, подсчитывающая распределение выходных меток по 44 возможным вариантам.

Как обычно, мы воспользуемся оптимизатором Adam с настройками гиперпараметров по умолчанию, будем обучать сеть в течение 1000 эпох и применим пакетную нормализацию для регуляризации.

Сеть будет очень похожа на те, которые мы уже реализовали раньше. Самое сложное в создании разметчика частей речи — подготовка набора данных. Мы применим предварительно обученные векторные представления слов из Google News[74]. Набор содержит векторы для 3 миллионов слов и обучен примерно на 100 млрд слов. Для его чтения можно воспользоваться пакетом gensim Python. Устанавливаем его при помощи pip:

```
$ pip install gensim
```

Теперь можно загрузить векторы в память при помощи следующей команды:

```
from gensim.models import Word2Vec
model = Word2Vec.load_word2vec_format('/path/to/googlenews.bin',
                                      binary=True)
```

Проблема в том, что эта операция невероятно медленна: может занимать до часа времени в зависимости от быстродействия вашего компьютера. Чтобы не загружать в память полный набор данных при каждом запуске программы, особенно при отладке кода или экспериментах с гиперпараметрами, мы кэшируем нужный поднабор векторов на диске при помощи быстрой базы данных `LevelDB`[75]. Чтобы создать необходимые привязки в `Python` (что позволит обеспечить взаимодействие `LevelDB` и `Python`), введем следующую команду:

```
$ pip install leveldb
```

Как мы уже говорили, модель `gensim` содержит 3 миллионов слов, что превышает размер нашего набора данных. Для эффективности мы кэшируем векторы слов из нашего набора и игнорируем всё остальное. Чтобы понять, какие слова кэшировать, загрузим набор данных POS из задачи CoNLL-2000[76].

```
$ wget http://www.cnts.ua.ac.be/conll2000/chunking/train.txt.gz
      -O – | gunzip |
      cut -f1,2 -d" " > pos.train.txt
$ wget http://www.cnts.ua.ac.be/conll2000/chunking/test.txt.gz
      -O – | gunzip |
      cut -f1,2 -d " " > pos.test.txt
```

Набор данных состоит из текста, отформатированного как последовательность строк, где первый элемент — слово, а второй — соответствующая ему часть речи. Вот первые несколько строк обучающего набора данных:

```
Confidence NN
in IN
the DT
pound NN
is VBZ
widely RB
expected VBN
to TO
take VB
another DT
sharp JJ
dive NN
if IN
trade NN
figures NNS
for IN
```

```
September NNP
, ,
due JJ
for IN
release NN
tomorrow NN
…
```

Чтобы сопоставить форматирование набора данных с моделью `gensim`, нужна предварительная обработка. Например, модель заменяет цифры знаками #, объединяет при необходимости отдельные слова в словосочетания (например, рассматривает `New_York` как одну единицу, а не два отдельных слова) и заменяет дефисы в исходных данных подчеркиваниями. Мы подвергаем набор данных предварительной обработке с помощью следующего кода (аналогичный используется и при обработке данных для обучения):

```
with open("/path/to/pos.train.txt") as f:
            train_dataset_raw = f.readlines()
            train_dataset_raw = [e.split() for e in
            train_dataset_raw if len(e.split()) > 0]
      counter = 0
      while counter < len(train_dataset_raw):
            pair = train_dataset_raw[counter]
            if counter < len(train_dataset_raw) – 1:
                  next_pair = train_dataset_raw[counter + 1]
                  if (pair[0] + "_" + next_pair[0] in model) and
                        (pair[1] == next_pair[1]):
                              train_dataset.append([pair[0] + "_" +
                        next_pair[0], pair[1]])
                        counter += 2
                        continue
            word = re.sub("\d", "#", pair[0])
            word = re.sub("-", "_", word)
            if word in model:
                  train_dataset.append([word, pair[1]])
                  counter += 1
                  continue
            if "_" in word:
                  subwords = word.split("_")
```

```
                for subword in subwords:
            if not (subword.isspace() or len(subword) == 0):
                train_dataset.append([subword, pair[1]])
            counter += 1
            continue
        train_dataset.append([word, pair[1]])
        counter += 1
with open('/path/to/pos.train.processed.txt', 'w')
as train_file: for item in train_dataset:
            train_file.write("%s\n" % (item[0] + " " +
                                       item[1]))
```

Теперь, подготовив данные к использованию, можно загрузить слова в `LevelDB`. Если слово или фраза присутствуют в модели `gensim`, мы кэшируем их в `LevelDB`. Если же нет, мы случайным образом выбираем вектор, который будет представлять слово или фразу, и кэшируем так, чтобы использовать тот же вектор при дальнейших вхождениях:

```
db = leveldb.LevelDB("data/word2vecdb")
counter = 0
        for pair in train_dataset + test_dataset:
            dataset_vocab[pair[0]] = 1
            if pair[1] not in tags_to_index:
                tags_to_index[pair[1]] = counter
                index_to_tags[counter] = pair[1]
                counter += 1
            nonmodel_cache = {}
            counter = 1
            total = len(dataset_vocab.keys())
            for word in dataset_vocab:
                if counter % 100 == 0:
                print "Inserted %d words out of %d total" % (
                        counter, total)
            if word in model:
                db.Put(word, model[word])
            elif word in nonmodel_cache:
                db.Put(word, nonmodel_cache[word])
            else:
                print word
                nonmodel_cache[word] = np.random.uniform(-0.25,
```

```
                    0.25, 300).
                            astype(np.float32)
                db.Put(word, nonmodel_cache[word])
        counter += 1
```

После запуска скрипта в первый раз мы можем просто загружать данные из базы, если она уже существует:

```
db = leveldb.LevelDB("data/word2vecdb")
    with open("data/pos_data/pos.train.processed.txt") as f:
        train_dataset = f.readlines()
        train_dataset = [element.split() for element in
            train_dataset if
            len(element.split()) > 0]
    with open("data/pos_data/pos.train.processed.txt") as f:
        test_dataset = f.readlines()
        test_dataset = [element.split() for element in test_dataset
            if len(element.split()) > 0]
                counter = 0
        for pair in train_dataset + test_dataset:
        dataset_vocab[pair[0]] = 1
    if pair[1] not in tags_to_index:
        tags_to_index[pair[1]] = counter
        index_to_tags[counter] = pair[1]
    counter += 1
```

Наконец, мы создаем объекты как для обучающего, так и для тестового наборов данных, чтобы на их основе разработать мини-пакеты для обучения и тестирования. Создание объекта набора данных требует доступа к дескриптору базы `LevelDB db`, самому набору данных `dataset`, словарю `tags_to_index` для сопоставления меток частей речи с индексами входного вектора и логическому флагу `get_all`, который определяет, будет ли вызов метода для создания мини-пакета возвращать полный набор по умолчанию:

```
class POSDataset():
    def __init__(self, db, dataset, tags_to_index,
        get_all=False):
        self.db = db
        self.inputs = []
        self.tags = []
        self.ptr = 0
        self.n = 0
```

```
            self.get_all = get_all
        for pair in dataset:
            self.inputs.append(np.fromstring(db.Get(pair[0]),
                                                dtype=np.float32))
        self.tags.append(tags_to_index[pair[1]])
        self.inputs = np.array(self.inputs, dtype=np.float32)
        self.tags = np.eye(len(tags_to_index.keys()))
                                [self.tags]
        def prepare_n_gram(self, n):
                            self.n = n
        def minibatch(self, size):
            batch_inputs = []
        batch_tags = []
if self.get_all:
        counter = 0
            while counter < len(self.inputs) - self.n + 1:
        batch_inputs.append(self.inputs[
        counter:counter+self.n].flatten())
        batch_tags.append(self.tags[counter +
                    self.n - 1])
        counter += 1
            elif self.ptr + size < len(self.inputs) - self.n:
                    counter = self.ptr
        while counter < self.ptr + size:
            batch_inputs.append(self.inputs
            [counter:counter+self.n].flatten())
            batch_tags.append(self.tags[counter +
                            self.n - 1])
        counter += 1
            else:
                    counter = self.ptr
                    while counter < len(self.inputs) - self.n + 1:
                    batch_inputs.append(self.inputs[
                            counter:counter+self.n].flatten())
        batch_tags.append(self.tags[counter +
                            self.n - 1])
        counter += 1
        counter2 = 0
```

```
        while counter2 < size – counter + self.ptr:
            batch_inputs.append(self.inputs[
                counter2:counter2+self.n].flatten())
            batch_tags.append(self.tags[
                counter2 + self.n – 1])
                counter2 += 1
        self.ptr = (self.ptr + size) % (len(self.inputs) -
                self.n)
    return np.array(batch_inputs, dtype=np.float32),
            np.array
        (batch_tags)
train = POSDataset(db, train_dataset, tags_to_index)
test = POSDataset(db, test_dataset, tags_to_index,
get_all=True)
```

Наконец, мы строим сеть с прямым распространением сигнала так же, как и в других главах. Обсуждение кода опустим, детали можно узнать из прилагаемого к книге репозитория, файл `feedforward_pos.py`. Чтобы запустить модель с входными векторами `3-грамм`, выполняем следующую команду:

```
$ python feedforward_pos.py 3
LOADING PRETRAINED WORD2VEC MODEL…
Using a 3-gram model
Epoch: 0001 cost = 3.149141798
Validation Error: 0.336273431778
Then ``
the DT
woman NN
, RP
after UH
grabbing VBG
her PRP
umbrella NN
, RP
went UH
to TO
the PDT
bank NN
to TO
```

```
deposit PDT
her PRP
cash NN
. SYM
Epoch: 0002 cost = 2.971566474
Validation Error: 0.300647974014
Then ``
the DT
woman NN
, RP
after UH
grabbing RBS
her PRP$
umbrella NN
, RP
went UH
to TO
the PDT
bank NN
to TO
deposit )
her PRP$
cash NN
. SYM
…
```

Во время каждой эпохи мы вручную проверяем модель, запуская разбор предложения The woman, after grabbing her umbrella, went to the bank to deposit her cash («Женщина, взяв свой зонт, пошла в банк внести свои наличные»). За 100 эпох обучения алгоритм достигает точности более 96% и почти идеально разбирает проверочное предложение (делая только одну объяснимую ошибку: путая метки притяжательного и личного местоимений при первом появлении слова `her` («ее»)). Визуализация работы нашей модели с помощью `TensorBoard` показана на рис. 7.4.

Модель разметки частей речи была отличным упражнением, но в основном на повторение и закрепление идей из предыдущих глав. Сейчас мы начнем работу с гораздо более сложными задачами, связанными с последовательностями. Для этого потребуются новые идеи и архитектуры. Мы выхо-

дим на передний край современных работ в области глубокого обучения. Начнем с проблемы определения зависимостей.

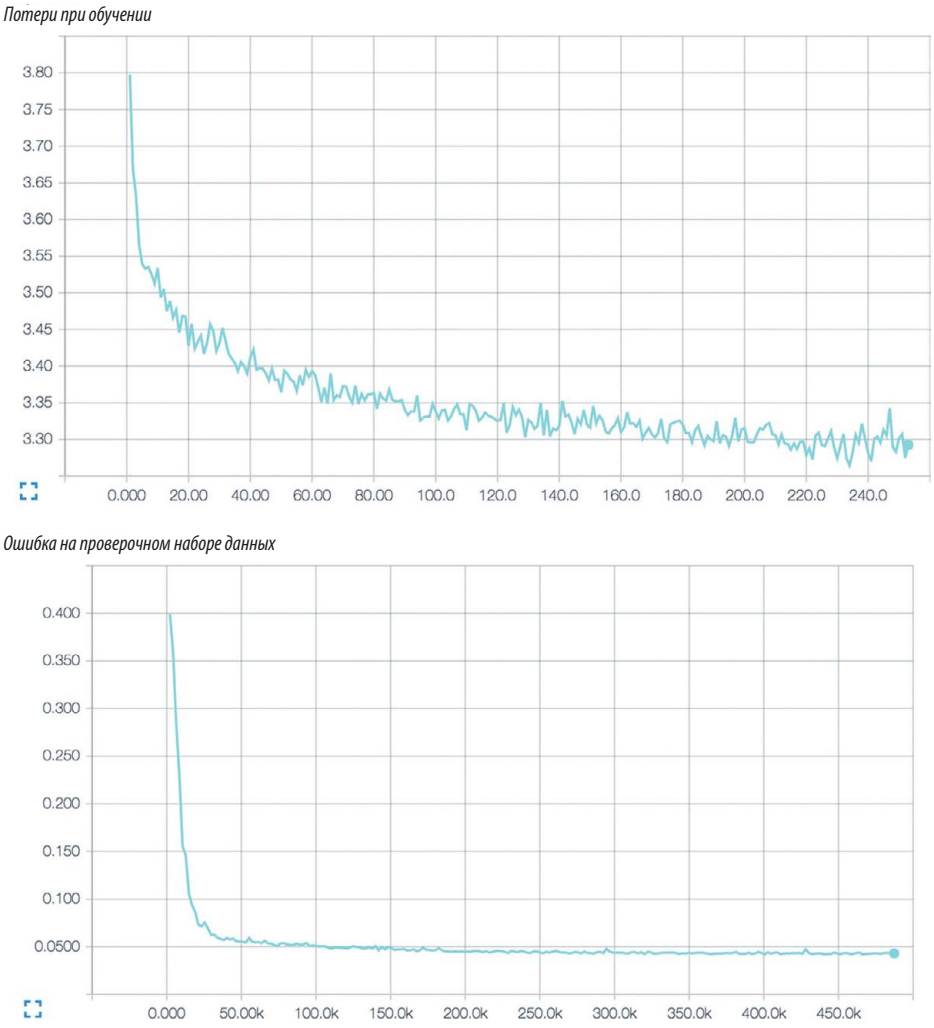


Рис. 7.4. *Визуализация в TensorBoard модели с прямым распространением сигнала для определения частей речи*

Определение зависимостей и SyntaxNet

Подход, который мы использовали для разметки частей речи, был простым. Но часто при решении задач `seq2seq` нужно применять куда

более творческие решения, особенно если сложность проблемы возрастает. Ниже мы рассмотрим стратегии, которые задействуют изощренные структуры данных для решения сложных задач `seq2seq`. Для иллюстрации возьмем проблему определения зависимостей.

Идея построения дерева определения зависимостей в том, чтобы установить связи между словами в предложении. Возьмем, например, зависимость на рис. 7.5. Слова `I` («я») и `taxi` («такси») — дочерние для `took` («взял»), поскольку это подлежащее и прямое дополнение при глаголе.

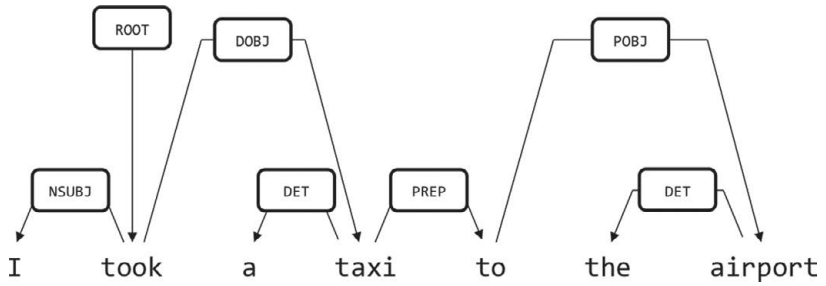


Рис. 7.5. *Пример поиска зависимостей, который строит дерево связей между словами в предложении*

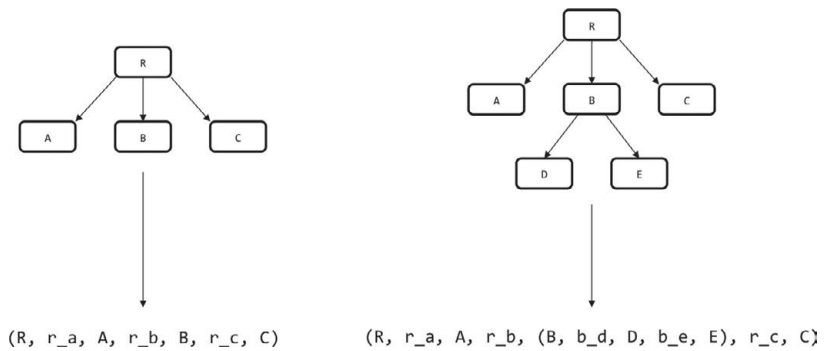


Рис. 7.6. *Пример линейного преобразования двух деревьев. Для удобства чтения на диаграммах не подписаны ребра*

Один из вариантов выражения дерева в виде последовательности — линеаризация. Рассмотрим примеры на рис. 7.6. Если у вас есть граф с корнем R и дочерними элементами A (связаны ребром `r_a`), `B` (связаны ребром `r_b`) и `C` (связаны ребром `r_c`), можно выполнить линейное преобразование, получив (`R, r_a, A, r_b, B, r_c, C`). Так легко представить и более сложные графы. Допустим, у узла `B` есть еще два дочерних элемента:

D (ребро b_d) и E (ребро b_e). Этот новый граф можно представить как (R, r_a, A, r_b, [B, b_d, D, b_e, E], r_c, C).

В той же парадигме мы можем выполнить линейное преобразование нашего примера разбора зависимости, как показано на рис. 7.7.

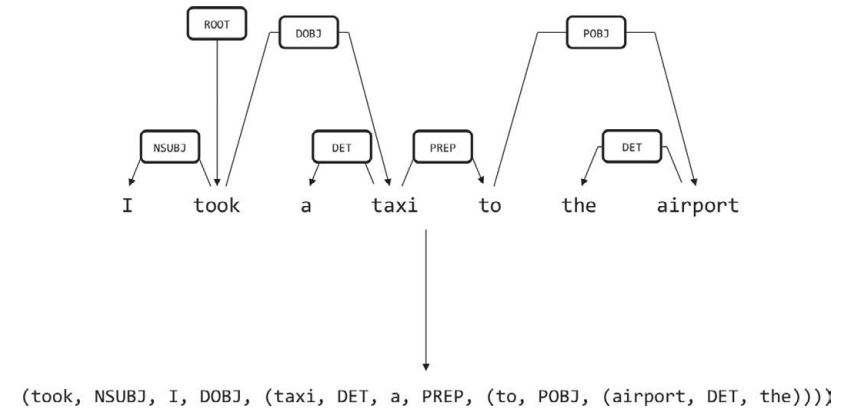


Рис. 7.7. *Линейное преобразование примера дерева разбора зависимостей*

Одна из интерпретаций этой проблемы seq2seq — чтение предложения ввода и получение на выходе последовательности знаков, представляющей собой линейное преобразование разбора зависимостей входящих данных. Но не до конца понятно, как реализовать здесь ту же стратегию, что и в предыдущем разделе, если там существовала четкая привязка слов к меткам частей речи. Более того, там мы могли легко принимать решения о метке для части речи на основе ближайшего контекста. При разборе зависимостей нет четкой связи между расположением слов в предложении и символов в его линейном представлении. И, судя по всему, при разборе зависимостей придется определять ребра, которые могут относиться ко множеству слов. На первый взгляд кажется, будто схема прямо противоречит изначальному заявлению о том, что нам не нужно учитывать долгосрочные связи.

Чтобы упростить решение, можно переосмыслить задачу разбора зависимостей как попытку поиска последовательности верных «действий», которая генерирует корректный разбор. Эта техника получила название *системы стандартных дуг*, была впервые описана в 2004 году Иоакимом Нивром[77], а затем применена в контексте нейросетей Данки Ченом и Кристофером Мэннингом в 2014 году[78].

В системе стандартных дуг мы сперва помещаем в стек первые два слова предложения, а оставшиеся сохраняем в буфере, как показано на рис. 7.8.

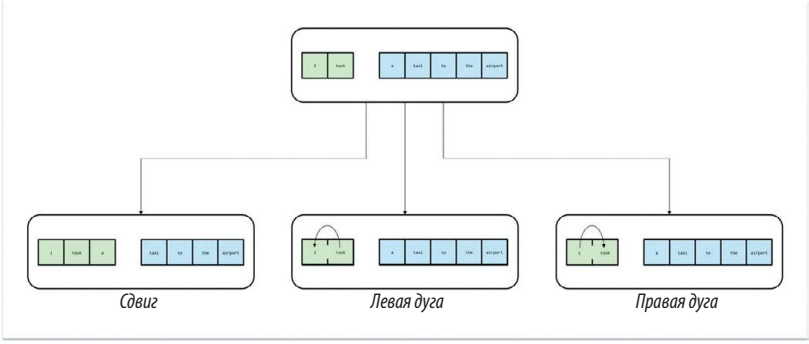


Рис. 7.8. *На каждом шаге у нас три варианта: передвинуть слово из буфера (голубого) в стек (зеленый), провести дугу от правого элемента к левому (левая дуга) или от левого к правому (правая дуга)*

На каждом шаге мы можем выполнить одно из трех действий.

СДВИГ

Передвинуть слово из буфера в переднюю часть стека.

ЛЕВАЯ ДУГА

Объединить два элемента в передней части стека в общую единицу, где корень правого элемента будет родительским узлом, а левого — дочерним.

ПРАВАЯ ДУГА

Объединить два элемента в передней части стека в общую единицу, где корень левого будет родительским узлом, а правого — дочерним.

И если `СДВИГ` можно выполнить только одним способом, то `ДУГИ` могут осуществляться разными методами в соответствии с ярлыками зависимости, назначаемыми результату. Но мы упростим изложение и иллюстрации в этом разделе и будем считать, что каждое решение — выбор из трех действий (а не нескольких десятков).

Мы заканчиваем процесс, когда буфер пустеет, а в стеке остается один элемент, представляющий полный разбор зависимостей. Чтобы проиллюстрировать процесс, рассмотрим на рис. 7.9 пример последовательности действий, которые создают разбор зависимостей для нашего входного предложения.

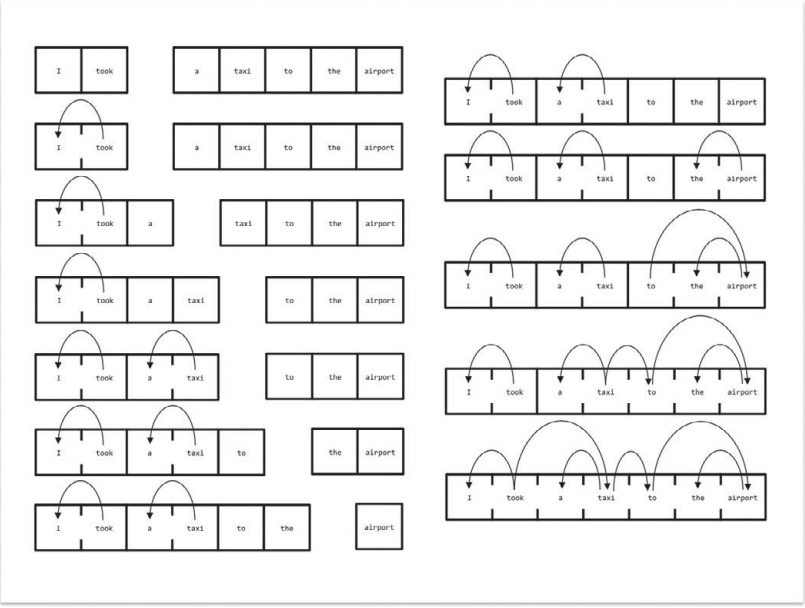


Рис. 7.9. *Последовательность действий, которые ведут к корректному разбору зависимостей (ярлыки опускаем)*

Не так сложно переформулировать эту принимающую решения схему в виде проблемы обучения. На каждом шаге мы берем текущую конфигурацию и преобразуем ее в векторный вид, извлекая множество описывающих ее признаков (слов в определенных местах стека/буфера, дочерних элементов слов в этих местах, меток частей речи и т. д.).

Во время обучения мы можем ввести этот вектор в сеть с прямым распространением сигнала и сравнить ее предсказания последующих действий с «золотым стандартом» решений, принимаемых лингвистом-человеком.

Для использования модели на практике можно выполнить рекомендованное сетью действие, применить его к конфигурации и взять новую конфигурацию за основу для следующего шага (извлечение признаков, предсказание действия и его выполнение). Этот процесс показан на рис. 7.10.

Все эти идеи лежат в основе `SyntaxNet` от `Google`, выдающейся открытой реализации разбора зависимостей. Вдаваться в подробности мы не будем, детали есть в открытом репозитории[79], который содержит реализацию `Parsey McParseface` — самого точного из публично описанных парсеров английского языка на момент публикации этой книги.

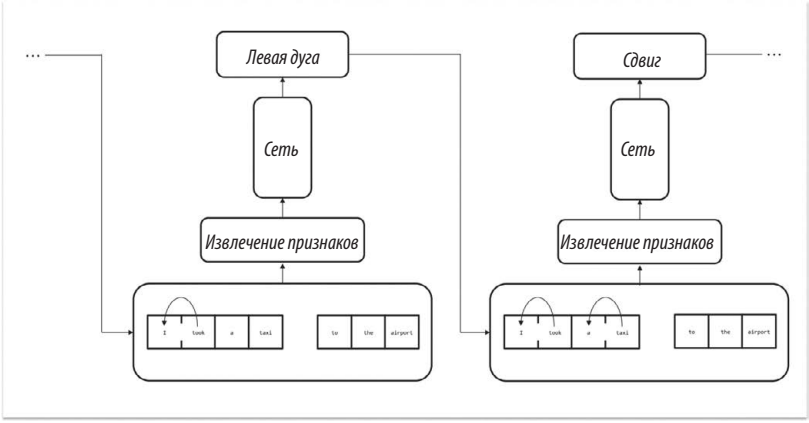


Рис. 7.10. *Нейронная схема разбора зависимостей с помощью метода стандартных дуг*

Лучевой поиск и глобальная нормализация

В предыдущем разделе мы описали наивную стратегию практического применения `SyntaxNet`. Она была *жадной*: мы выбирали самые вероятные предсказания, не заботясь о том, что можем загнать себя в угол, сделав ошибку на раннем этапе. В примере с частями речи неверное предсказание по сути не влекло никаких последствий. В этом случае его можно считать независимой проблемой, поскольку его результаты не влияют на входные данные на следующем шаге. Но это предположение в `SyntaxNet` не подтверждается, поскольку наше предсказание на шаге `n` влияет на входные данные, которые используются на шаге `n + 1`. Отсюда следует, что любая ошибка повлияет на все дальнейшие решения. Более того, нет простого способа «вернуться» и исправить огрехи, когда они станут очевидными. Крайний случай — *предложения с подвохом*. Рассмотрим такое предложение: `The complex houses married and single soldiers and their families` [«Комплекс обеспечивает жильем женатых и одиноких солдат и их семьи»]. На первый взгляд выглядит странно: большинство людей решат, что `complex` — прилагательное, `houses` — существительное, а `married` — глагол прошедшего времени. Смысла мало [дословно «сложные дома женились»], и мы в недоумении, когда добираемся до конца предложения. Тут нам становится ясно, что `complex` — существительное (военный комплекс), а `houses` — глагол («обеспечивает жильем»). Иными словами, предложение сообщает, что военный комплекс дает жилье солдатам (как одиноким, так и женатым) и их семьям. *Жадная* версия `SyntaxNet` не сможет исправить первичную ошибку восприятия `complex`

houses как «сложных домов» и создать верную трактовку предложения. Для устранения этого недостатка используется стратегия лучевого поиска, показанная на рис. 7.11.

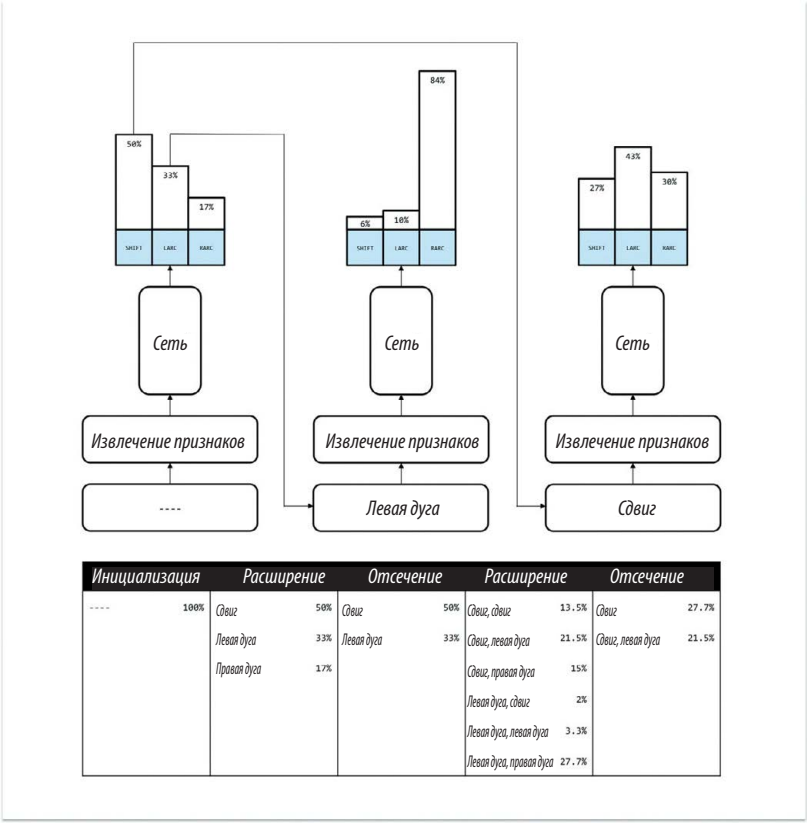


Рис. 7.11. *Пример использования лучевого поиска (с размером луча 2) при разворачивании обученной модели SyntaxNet*

Обычно она применяется как раз в ситуациях вроде SyntaxNet, когда выводы сети на определенном шаге влияют на входные данные, используемые на следующих шагах. Основная идея — не быстро выбирать самое вероятное предсказание на каждом шаге, а определять *луч* наиболее вероятной гипотезы (*фиксированного размера b*) для последовательности первых k действий и связанных с ними возможностей. Лучевой поиск можно разбить на две основные стадии: расширение и отсечение.

Во время расширения мы анализируем каждую гипотезу, рассматривая ее как возможный ввод в SyntaxNet. Допустим, SyntaxNet выдает

распределение вероятностей по количеству действий $|A|$. Затем мы вычисляем вероятность $b|A|$ возможных гипотез для последовательности первых $(k + 1)$ действий. А во время отсечения мы оставляем только b гипотез с наибольшей вероятностью из $b|A|$ вариантов.

Как показывает рис. 7.11, лучевой поиск позволяет `SyntaxNet` постфактум исправлять неверные предположения, задействуя с самого начала и менее вероятные гипотезы, которые, однако, могут впоследствии оказаться более плодотворными. Если углубляться в пример на рисунке, можно отметить, что жадный алгоритм предположил бы, будто верная последовательность шагов — СДВИГ плюс ЛЕВАЯ ДУГА. На самом деле лучшим (наиболее вероятным) вариантом будет использовать ЛЕВУЮ ДУГУ и за ней ПРАВУЮ ДУГУ.

Лучевой поиск с размером луча 2 отражает этот результат.

Полная версия с открытыми исходными кодами делает шаг вперед и пытается внедрить лучевой поиск в процесс обучения сети. В 2016 году Дэниел Андор и его коллеги[80] описали этот процесс *глобальной нормализации*. Он дает как хорошие теоретические результаты, так и очевидные практические преимущества перед *локальной нормализацией*. Во втором случае сеть получает задачу выбора лучшего действия в зависимости от конфигурации. Она выдает результат, который нормализуется слоем функции мягкого максимума. Так моделируется распределение вероятностей по всем возможным действиям, если они уже совершены. Функция потерь пытается довести распределение вероятностей до идеального выходного результата (например, вероятности 1 для правильного действия и 0 для всех остальных). И функция потерь — перекрестная энтропия — прекрасно с этим справляется.

В глобально нормализованной сети интерпретация результатов будет немного другой. Вместо того чтобы прогонять их через функцию мягкого максимума для получения распределения вероятностей каждого действия, мы суммируем все результаты последовательности действий гипотезы. Один из способов выбрать верную последовательность — рассчитать такую сумму по всем гипотезам и наложить на результаты слой функции мягкого максимума, получив распределение вероятностей. Теоретически можно использовать ту же функцию потерь перекрестной энтропии, что и в локально нормализованной сети. Но эта стратегия сопряжена с проблемой: число возможных гипотез последовательности невероятно велико. Даже если взять предложение со средней длиной 10 и консервативной оценкой общего числа действий в 15, один сдвиг и по семь меток для левой и правой дуг, гипотез получится 1 000 000 000 000 000.

Чтобы разобраться с этой проблемой, мы, как показано на рис. 7.12, применяем лучевой поиск с фиксированным размером луча, пока либо не достигнем конца предложения, либо верная последовательность действий не появится

на луче. После этого мы строим функцию потерь, которая будет поддерживать «золотой стандарт» последовательности действий (выделенный голубым цветом) как можно выше на луче, максимизируя его оценку по сравнению с другими гипотезами. Не будем вдаваться в подробности создания этой функции, детали описаны в работе Андора и коллег[81]. В ней также показан более изощренный разметчик частей речи, который использует глобальную нормализацию и лучевой поиск и тем самым значительно увеличивает точность по сравнению с разметчиком, который мы создали в этой главе.

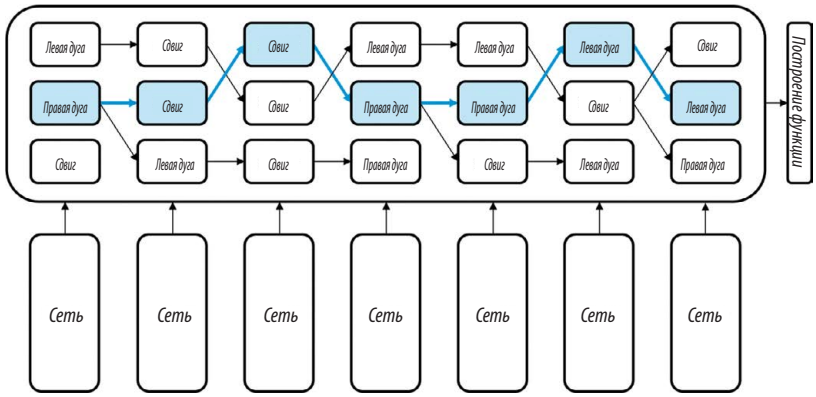


Рис. 7.12. *Реализовать глобальную нормализацию в SyntaxNet можно, если сочетать обучение и лучевой поиск*

Когда нужна модель глубокого обучения с сохранением состояния

Мы уже рассмотрели несколько хитростей, позволяющих приспособить сети с прямым распространением сигнала к анализу последовательностей, но нам еще предстоит найти изящное решение. В примере с разметкой частей речи мы прямо предположили, что можем игнорировать долгосрочные зависимости. Нам удалось преодолеть ряд ограничений, введя понятия лучевого поиска и глобальной нормализации, но все равно поле действия было ограничено ситуациями, в которых возможно однозначное соответствие между элементами входной и выходной последовательностей. Даже в модели разбора зависимостей пришлось переформулировать проблему, чтобы найти такое соответствие между рядом конфигураций входных данных при создании дерева разбора и действиями над стандартными дугами.

Иногда задача куда сложнее нахождения однозначных соответствий между входной и выходной последовательностями. Например, мы хотим построить модель, которая может сразу принимать все предложение и делать вывод,

положительна или отрицательна его эмоциональная окраска. В этой главе мы построим простую модель, способную решать эту задачу. Или нам может понадобиться алгоритм, который будет получать комплексные входные данные (например, изображение) и порождать предложение (слово за словом), его описывающее. Можно даже попробовать перевести предложения с одного языка на другой (например, с английского на французский). Во всех этих случаях нет однозначной очевидной связи между символами на входе и выходе. Процесс больше напоминает ситуацию, приведенную на рис. 7.13.

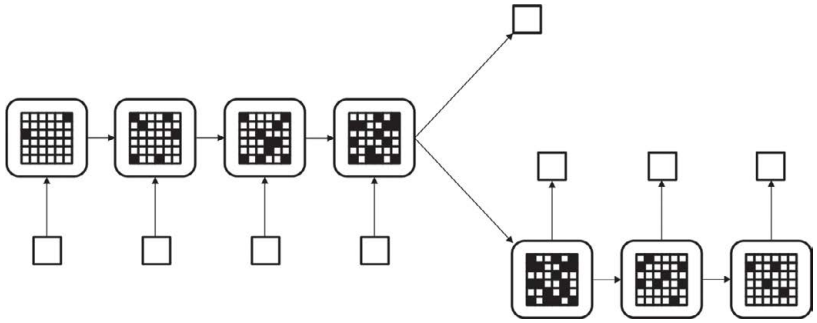


Рис. 7.13. *Идеальная модель анализа последовательностей может хранить информацию в памяти долгое время, порождая устойчивый «мыслительный» вектор, который используется для нахождения ответа*

Идея проста. Мы хотим, чтобы наша модель сохраняла какую-то память при считывании входной последовательности. В процессе чтения она должна быть способна изменять банк памяти, учитывая получаемую информацию. Когда она достигнет конца входной последовательности, внутренняя память должна содержать «мысль», представляющую ключевые элементы информации, то есть смысл входных данных. Затем мы, как показано на рис. 7.13, можем с помощью этого вектора мысли либо создать метку для исходной последовательности, либо породить соответствующую выходную последовательность (перевод, описание, резюме и т. д.).

В предыдущих главах эта идея не рассматривалась. Сети с прямым распространением сигнала по природе своей не имеют «состояний». После обучения любая из них становится статичной. Она не может ни переключать память между разными входными данными, ни изменять способы их обработки на основе входных данных, с которыми имела дело в прошлом. Чтобы реализовать эту стратегию, придется пересмотреть архитектуру нейронных сетей и начать создавать модели глубокого обучения с фиксацией состояния. Вернемся к рассмотрению сетей на уровне нейронов. В следующем разделе поговорим о том, как рекуррентные связи (в отличие

от прямых, которые мы рассматривали ранее) позволяют моделям фиксировать состояние, и опишем класс моделей, известных как *рекуррентные нейронные сети (РНС)*.

Рекуррентные нейронные сети

Первые РНС были предложены в 1980-е годы, но популярность обрели лишь недавно благодаря нескольким интеллектуальным и техническим прорывам, которые помогли повысить их обучаемость. РНС отличаются от сетей с прямым распространением сигнала, потому что в них используется особый тип нейронного слоя, именуемый рекуррентным и позволяющий сети сохранять свое состояние в промежутках между сеансами ее использования.

На рис. 7.14 показана нейронная архитектура рекуррентного слоя. У всех нейронов есть входные соединения, приходящие от всех нейронов предыдущего слоя, и выходные, ведущие ко всем нейронам последующего. Однако эти типы соединений для рекуррентного слоя не единственные. В отличие от слоя с прямым распространением сигнала, он обладает рекуррентными соединениями, которые распространяют информацию между нейронами одного слоя. В полносвязном подобном слое информационный поток идет от каждого нейрона к каждому нейрону того же слоя (в том числе и к себе). У такого слоя с числом нейронов r есть $r2$ рекуррентных соединений.

Чтобы лучше понять, как работает РНС, рассмотрим ее функционирование после соответствующего обучения. Каждый раз, когда нам нужно обработать очередную последовательность, мы создаем новый экземпляр нашей модели.

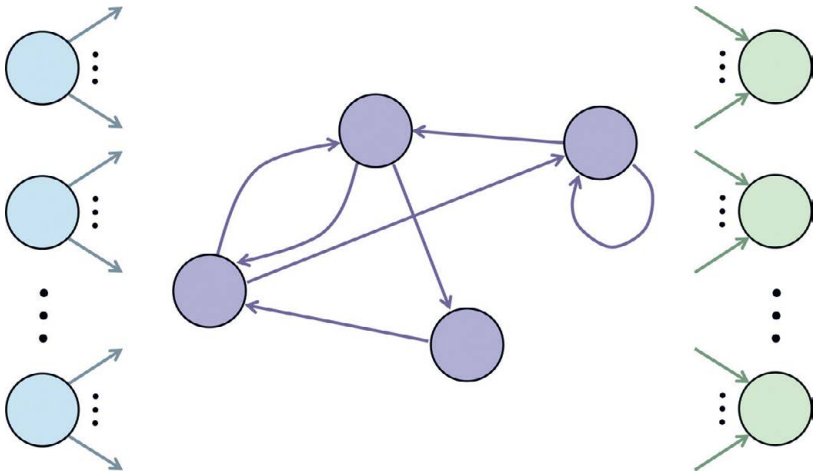


Рис. 7.14. *Рекуррентный слой содержит рекуррентные соединения между нейронами, расположенными на одном уровне*

Можно работать с сетями, содержащими рекуррентные слои, разделив срок жизни экземпляра сети на дискретные временные шаги. На каждом шаге мы вводим в модель следующий элемент данных. Прямые соединения отражают информационный поток от одного нейрона к другому, в котором передаваемые данные — подсчитанная нейронная активация на текущем шаге. В случае же рекуррентных связей данные — сохраненная нейронная активация на *предыдущем* шаге. Таким образом, активации нейронов здесь отражают накапливаемое состояние экземпляра сети. Изначальные активации нейронов в рекуррентном слое — параметры нашей модели, и их оптимальные значения мы определяем точно так же, как лучшие значения для весов каждого соединения в процессе обучения. Оказывается, при фиксированном времени жизни (например, t шагов) экземпляра РНС мы можем выразить его в виде сети с прямым распространением сигнала, хоть и с нерегулярной структурой.

Это хитрое преобразование, показанное на рис. 7.15, часто именуется «разворачиванием» РНС во времени. Рассмотрим экземпляр РНС на рисунке. Нам нужно связать последовательность двух входов (каждый — размера 1) с одним выходом (тоже размера 1). Это преобразование мы осуществляем, взяв нейроны одного рекуррентного слоя и скопировав их t раз — по разу для каждого шага. Точно так же мы воспроизводим нейроны входного и выходного слоев. С каждым шагом мы копируем и прямые связи — такими, какими они были в исходной сети. Затем мы воссоздаем рекуррентные связи в виде прямых от каждой временной копии к следующей (поскольку такие связи отражают нейронную активацию предыдущего временного шага).

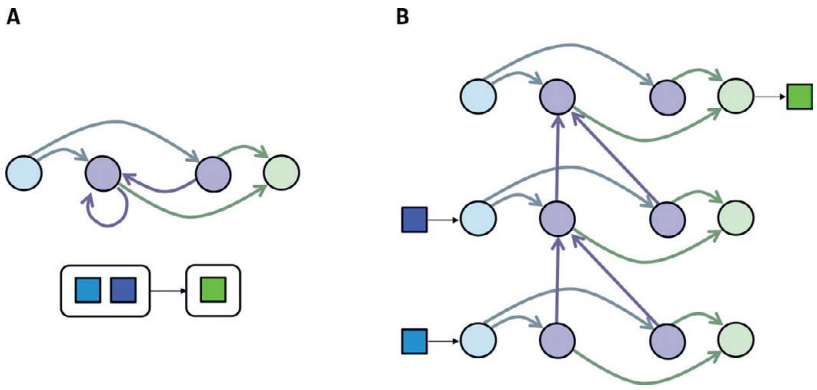


Рис. 7.15. *Мы можем развернуть РНС во времени, чтобы выразить ее в виде сети с прямым распространением сигнала, которую можно обучить с помощью алгоритма обратного распространения ошибок*

Теперь мы можем обучить РНС, вычислив градиент для развернутой версии. Это значит, что все алгоритмы обратного распространения ошибок, которые использовались для сетей с прямым распространением сигнала, подойдут и для РНС. Но есть одна проблема. После каждого пакета обучающих примеров необходимо изменить веса на основании вычисленных производных ошибок. В развернутой сети у нас есть наборы связей, и все они соответствуют одной связи из исходной РНС. А вот производные ошибок для этих развернутых связей не обязательно будут равны (на практике они чаще всего и не равны). Можно обойти эту проблему, усреднив или суммировав производные ошибок для всех связей одного набора. Это позволит использовать производную ошибок, учитывающую всю действующую на веса соединения динамику, при попытке заставить сеть выдать точный результат.

Проблема исчезающего градиента

Мотивы использования модели сети с фиксацией состояния связаны с идеей отражения долгосрочных зависимостей во входной последовательности. Вроде бы логично предположить, что РНС с большим банком памяти (рекуррентным слоем значительного размера) сможет запомнить все эти зависимости. И действительно, теоретически еще в 1996 году Джо Килиан и Хава Сигельманн показали, что РНС — универсальное функциональное представление[82]. Иными словами, при достаточном количестве нейронов и правильной настройке параметров РНС можно использовать для представления любых функциональных зависимостей между входными и выходными последовательностями. Эта теория много обещает, но на практике реализуется не всегда. Хоть и полезно знать, что РНС *может* выразить любую произвольную функцию, лучше понимать, насколько *практически полезно* с нуля обучать ее реалистическим функциональным связям путем применения алгоритмов градиентного спуска. Если это непрактично, мы окажемся в затруднительном положении, так что рассмотреть вопрос нужно максимально строго. Начнем с анализа самой простой РНС из возможных (рис. 7.16): один входной нейрон, один выходной, полносвязный рекуррентный слой тоже с одним нейроном.

Начнем с простого. При нелинейности f мы можем выразить активацию $h^{(t)}$ скрытого нейрона рекуррентного слоя на шаге t следующим образом, где $i^{(t)}$ — входной логит входного нейрона на временном шаге t:

$$h^{(t)} = f\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right).$$

Попытаемся вычислить, как активация скрытого нейрона будет меняться в ответ на корректировки входящего логита в течение k шагов в прошлом.

Анализ этого компонента градиентных выражений обратного распространения ошибок начнем с оценки того, сколько «памяти» сохраняется от предшествующих входных данных.

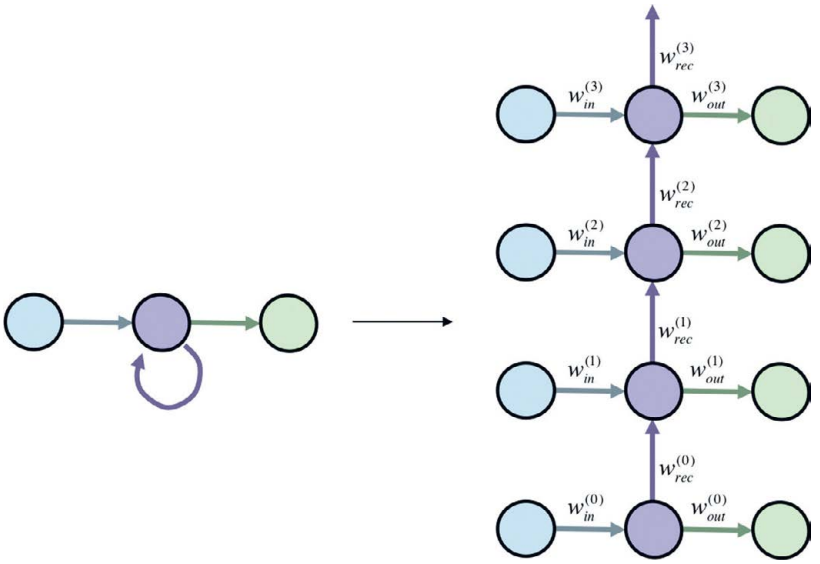


Рис. 7.16. *Один нейрон в полносвязном рекуррентном слое (в сжатом и развернутом во времени видах) — пример анализа обучающих алгоритмов на основе градиента*

Сначала возьмем частную производную и применим правило дифференцирования сложной функции:

$$\frac{\partial h^{(t)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right) \frac{\partial}{\partial i^{(t-k)}} \left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right).$$

Поскольку значения весов входного и рекуррентного соединений не зависят от входного логита на временном шаге $(t - k)$, можно упростить выражение:

$$\frac{\partial h^{(t)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t)} i^{(t)} + w_{rec}^{(t-1)} h^{(t-1)}\right) w_{rec}^{(t-1)} \frac{\partial h^{(t-1)}}{\partial i^{(t-k)}}.$$

Поскольку нам интересна величина этой производной, можно взять абсолютное значение обеих сторон. Мы также знаем, что для всех обычных

нелинейностей (гиперболического тангенса — `tanh`, логистической и `ReLU`) максимальное значение `f'` равно 1. Отсюда выводим следующее рекурсивное неравенство:

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \left|\frac{\partial h^{(t-1)}}{\partial i^{(t-k)}}\right|.$$

Можно продолжить рекурсивно расширять его, пока не дойдем до основного случая на шаге `(t – k)`:

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \ldots \cdot \left|w_{rec}^{(t-k)}\right| \cdot \left|\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}}\right|.$$

Оценить эту частную производную можно уже рассмотренным путем:

$$h^{(t-k)} = f\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right).$$

$$\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right) \frac{\partial}{\partial i^{(t-k)}}\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right).$$

В этом выражении активация скрытого слоя на шаге `(t – k – 1)` не зависит от значения входных данных на шаге `(t – k)`.

Поэтому мы можем переписать его:

$$\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}} = f'\left(w_{in}^{(t-k)} i^{(t-k)} + w_{rec}^{(t-k-1)} h^{(t-k-1)}\right) w_{in}^{(t-k)}.$$

Наконец, взяв абсолютное значение для обеих сторон и вновь применив наблюдение относительно максимального значения `f'`, можно записать:

$$\left|\frac{\partial h^{(t-k)}}{\partial i^{(t-k)}}\right| \leq \left|w_{in}^{(t-k)}\right|.$$

В результате получаем итоговое неравенство (которое можно упростить, поскольку мы хотим, чтобы связи на разных шагах имели одинаковые значения):

$$\left|\frac{\partial h^{(t)}}{\partial i^{(t-k)}}\right| \leq \left|w_{rec}^{(t-1)}\right| \cdot \ldots \cdot \left|w_{rec}^{(t-k)}\right| \cdot \left|w_{in}^{(t-k)}\right| = \left|w_{rec}\right|^{k} \cdot w_{in}.$$

Это отношение устанавливает жесткую верхнюю границу того, как изменения во входных данных на шаге ($t - k$) влияют на скрытое состояние на шаге t. Поскольку веса нашей модели в начале обучения невелики, значение этой производной с возрастанием k стремится к 0. Иными словами, градиент быстро уменьшается, когда он вычисляется по входным данным на несколько шагов назад, что существенно ограничивает способность модели к изучению долгосрочных зависимостей. Эта проблема обычно называется проблемой *исчезающего градиента*. Она серьезно влияет на способности обычных рекуррентных нейронных сетей к обучению. Наша задача — устранить эти ограничения, и в следующем разделе мы поговорим о чрезвычайно эффективном подходе к рекуррентным слоям, который именуется долгой краткосрочной памятью.

Нейроны долгой краткосрочной памяти (long short-term memory, LSTM)

Для борьбы с проблемой исчезающего градиента Зепп Хохрайтер и Юрген Шмидхубер ввели архитектуру `долгой краткосрочной памяти (LSTM)`. Основной ее принцип таков: сеть создается для надежного переноса важной информации на много шагов в будущее. Эти соображения привели к созданию архитектуры, показанной на рис. 7.17.

Для простоты обсуждения отойдем от уровня отдельных нейронов и будем говорить о сети как о наборе тензоров и операций над ними.

Как ясно из рисунка, нейрон `LSTM` состоит из нескольких ключевых компонентов. Один из них — *`ячейка памяти`*, тензор, выделенный жирным в центре рисунка. Она содержит важную информацию, которую усвоила со временем, а сеть призвана эффективно сохранять в ней эту полезную информацию на протяжении нескольких шагов. На каждом шаге нейрон `LSTM` изменяет ячейку памяти, снабжая ее новой информацией в три этапа. Сначала он должен определить, какую часть предшествующей информации следует хранить, при помощи *вентиля забвения* (рис. 7.18).

Основная идея проста. Тензор состояния памяти с предыдущего шага насыщен информацией, но часть ее может быть устаревшей, и ее следует стереть. Мы выясняем, какие элементы тензора релевантны, а какие уже нет, вычисляя двоичный тензор (состоящий из нулей и единиц), который мы умножаем на предыдущее состояние. Если соответствующее место в двоичном тензоре содержит 1, это значит, что место ячейки памяти по-прежнему значимо и его нужно сохранить. Если же на этом месте 0, оно утратило значимость и его следует забыть.

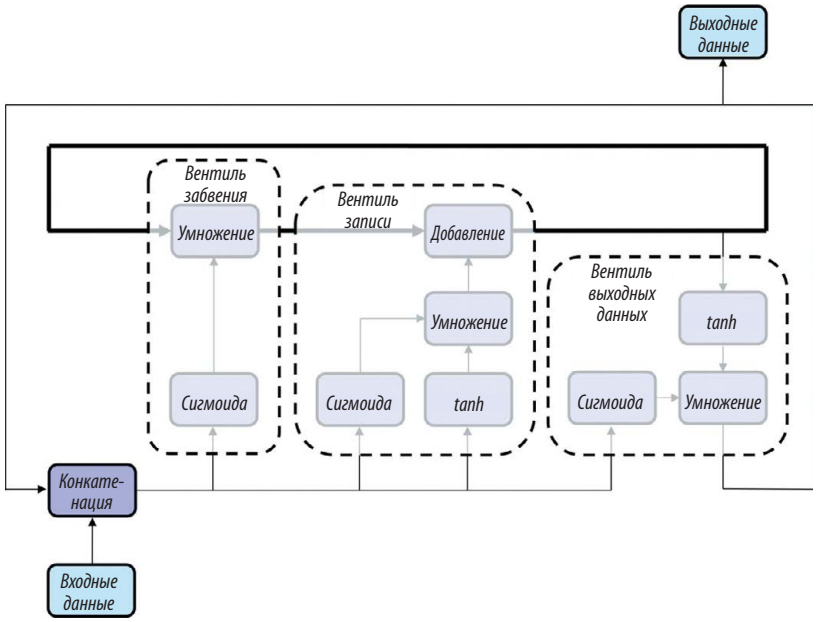


Рис. 7.17. *Архитектура нейрона LSTM на уровне тензоров (стрелки) и операций (сиреневые блоки)*

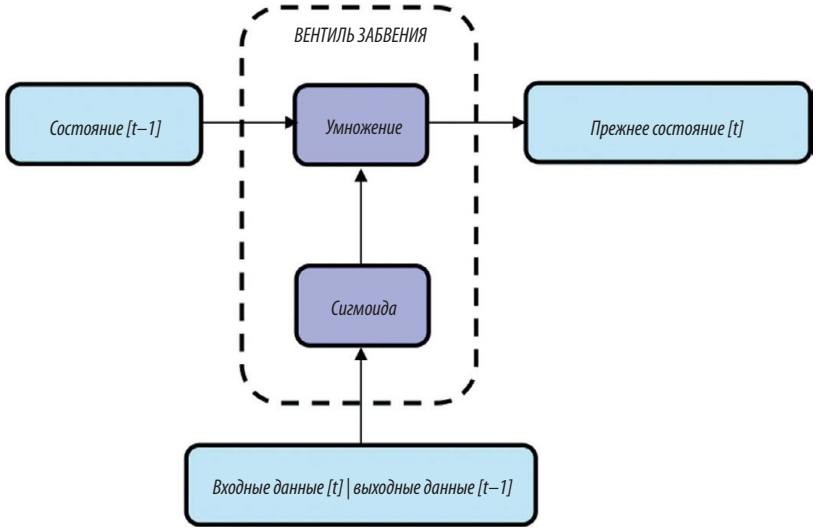


Рис. 7.18. *Архитектура вентиля забвения нейрона LSTM*

Мы аппроксимируем этот двоичный тензор, соединив входные данные этого шага и выходные данные нейрона LSTM с предыдущего и наложив на полученный тензор сигмоидный слой (sigmoid). Как вы наверняка помните, последний на выходе дает значение, которое обычно очень близко к 0 или 1 (единственное исключение — если входное значение само близко к 0). Выходные данные сигмоидного слоя — хорошее приближение двоичного тензора, чем можно воспользоваться при построении вентиля забвения. Поняв, какую информацию от прежних состояний следует сохранить, а какую забыть, мы переходим к той, которую нужно добавить в память. Эта часть нейрона LSTM называется *вентилем записи*, и она показана на рис. 7.19. Она делится на две основные части. Первая определяет, какую информацию мы хотим добавить в состояние. Это вычисляется в слое tanh путем создания промежуточного тензора. Второй компонент определяет, какие части этого тензора мы хотим ввести в новое состояние, а какие выбросить и не записывать. Для этого мы аппроксимируем двоичный вектор из нулей и единиц с помощью той же стратегии (сигмоидного слоя), что и для вентиля забвения. Затем мы умножаем двоичный вектор на промежуточный тензор и добавляем полученный результат, создавая новый вектор состояния для LSTM.

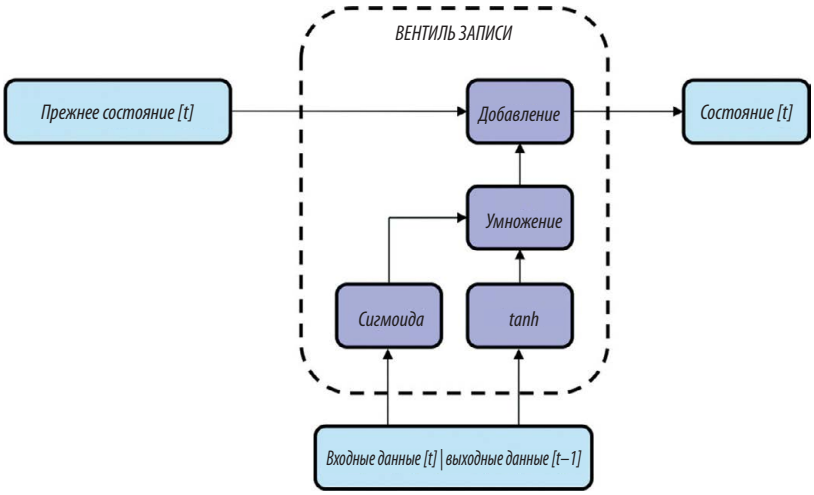


Рис. 7.19. *Архитектура вентиля записи в нейроне LSTM*

Наконец, на каждом временном шаге нейрон LSTM должен выдавать данные. Можно воспринимать вектор состояния как выходные данные, но нейрон LSTM призван обеспечить большую гибкость, передавая на выход тензор — «интерпретацию» или внешнюю «коммуникацию» того, что содержит вектор состояния. Архитектура выходного вентиля показана на рис. 7.20. Мы используем структуру, почти идентичную

реализованной для вентиля записи: слой `tanh` порождает промежуточный тензор от вектора состояния; сигмоидный слой создает маску двоичного тензора на основе текущего ввода и предыдущего вывода; промежуточный тензор умножается на двоичный тензор, что дает нам конечные выходные данные.

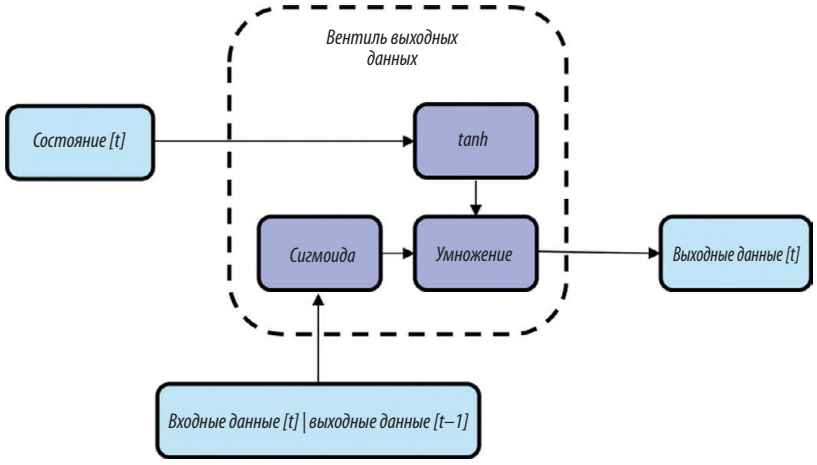


Рис. 7.20. *Архитектура выходного вентиля в нейроне LSTM*

Почему этот вариант лучше, чем обычный нейрон РНС? Главное здесь то, как информация распространяется по сети, когда мы разворачиваем нейрон `LSTM` во времени. Развернутая архитектура показана на рис. 7.21.

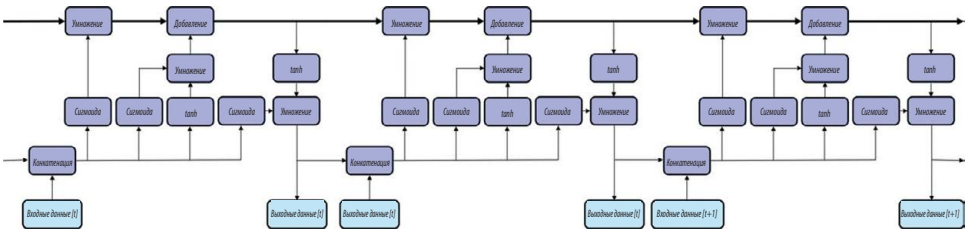


Рис. 7.21. *Разворачивание нейрона LSTM во времени*

На самом верху видно распространение вектора состояния, взаимодействия которого во времени в основном линейны. В результате градиент, который связывает входные данные, поступившие за несколько временных шагов до этого, с текущими выходными данными, не затухает так резко, как в обычной архитектуре РНС. А значит, `LSTM` может обучаться долгосрочным связям гораздо эффективнее, чем в исходной формулировке РНС.

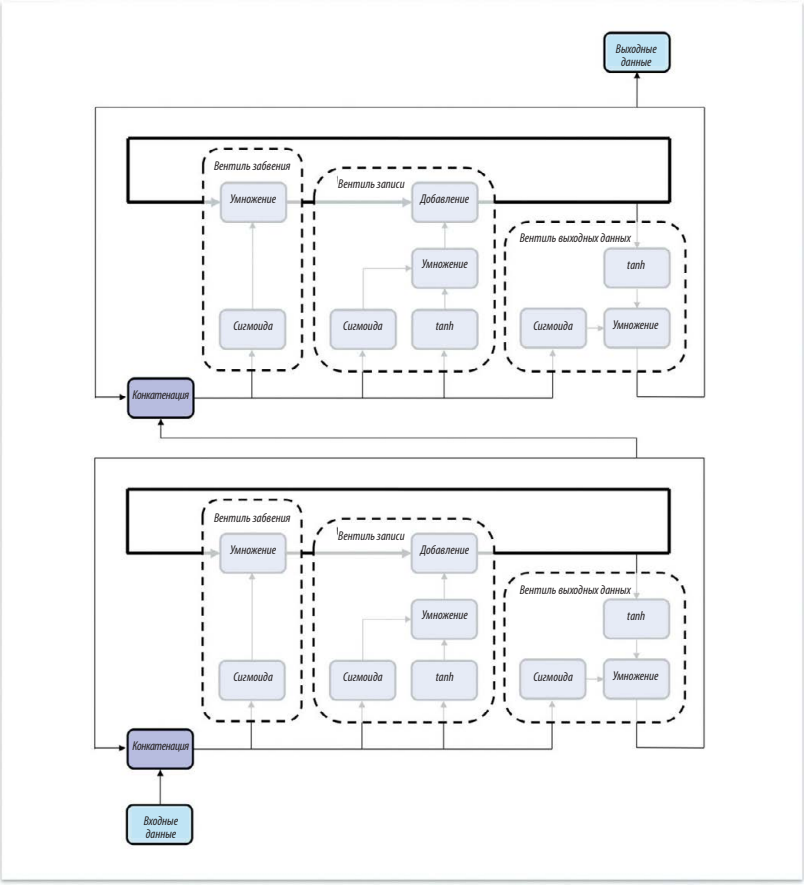


Рис. 7.22. *Соединение нейронов LSTM ничем не отличается от соединения рекуррентных слоев нейронной сети*

Наконец, нужно понять, насколько легко порождать произвольные архитектуры при помощи нейрона `LSTM`. Насколько они «компонуемы»? Не придется ли при использовании нейронов `LSTM` вместо обычных РНС пожертвовать гибкостью? Мы можем соединять их для большей выразительности так же, как соединяли обычные нейроны РНС, где вход второго нейрона будет выходом первого, вход третьего — выходом второго и т. д. Иллюстрация того, как это работает, показана на рис. 7.22, где из двух нейронов `LSTM` составлено целое. Это значит, что обычные слои РНС всегда можно заменить нейронами `LSTM`.

Итак, мы решили проблему исчезающего градиента и познакомились с работой нейронов `LSTM`. Теперь мы готовы заняться реализацией наших первых моделей РНС.

Примитивы TensorFlow для моделей РНС

`TensorFlow` предлагает несколько примитивов, которые можно использовать для создания моделей РНС. Во-первых, есть объекты `tf.RNNCell`, которые представляют либо слой РНС, либо `LSTM`:

```
cell_1 = tf.nn.rnn_cell.BasicRNNCell(num_units, input_size=None,
                                     activation=tanh)
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(num_units,
                                      forget_bias=1.0,
                                      input_size=None,
                                      state_is_tuple=True,
                                      activation=tanh)
cell_3 = tf.nn.rnn_cell.LSTMCell(num_units, input_size=None,
                                 use_peepholes=False,
                                 cell_clip=None,
                                 initializer=None,
                                 num_proj=None,
                                 proj_clip=None,
                                 num_unit_shards=1,
                                 num_proj_shards=1,
                                 forget_bias=1.0,
                                 state_is_tuple=True,
                                 activation=tanh)
cell_4 = tf.nn.rnn_cell.GRUCell(num_units, input_size=None,
                                 activation=tanh)
```

Абстракция `BasicRNNCell` представляет обычный рекуррентный нейронный слой. `BasicLSTMCell` соответствует простейшей реализации `LSTM`, а `LSTMCell` — реализации с большим количеством возможностей конфигурации (замочные скважины, клипирование переменных состояния и т. д.). Библиотека `TensorFlow` также содержит разновидность нейрона `LSTM`, известную как *вентильная рекуррентная единица* (Gated Recurrent Unit, GRU) и предложенную в 2014 году группой Джошуа Бенджо. Важнейший параметр для всех этих нейронов — размер вектора скрытого состояния, или `num_units`.

Кроме примитивов, в нашем арсенале есть несколько надстроек. Если мы хотим соединить рекуррентные нейроны или слои, можно сделать следующее:

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
cell_2 = tf.nn.rnn_cell.BasicLSTMCell(10)
full_cell = tf.nn.rnn_cell.MultiRNNCell([cell_1, cell_2])
```

Также можно при помощи надстройки применить прореживание ко входам и выходам LSTM с указанными вероятностями остаточной памяти входа и выхода:

```
cell_1 = tf.nn.rnn_cell.BasicLSTMCell(10)
       tf.nn.rnn_cell.DropoutWrapper(cell_1, input_keep_prob=1.0,
       output_keep_prob=1.0,
       seed=None)
```

Завершаем мы создание РНС, упаковывая всё в соответствующий примитив Tensor Flow:

```
outputs, state = tf.nn.dynamic_rnn(cell, inputs,
       sequence_length=None,
       initial_state=None,
       dtype=None,
       parallel_iterations=None,
       swap_memory=False,
       time_major=False,
       scope=None)
```

Ячейка (cell) — объект RNNCell, который мы уже создали. Если time_major == False (значение по умолчанию), входные данные должны быть тензором вида [batch_size, max_time, …]. Если же time_major == True, входной тензор должен принять форму [max_time, batch_size, …]. Подробности есть в документации TensorFlow, где можно узнать больше и о других параметрах конфигурации.

Результат обращения к tf.nn.dynamic_rnn — тензор, представляющий выход РНС вместе с вектором конечного состояния. Если time_major == False, вывод будет иметь форму [batch_size, max_time, cell.output_size]. Иначе тензор должен иметь вид [max_time, batch_size, cell.output_size]. Состояние, как можно ожидать, будет иметь форму [batch_size, cell.state_size].

Познакомившись с инструментами для создания рекуррентных нейронных сетей, которые есть в TensorFlow, мы построим первую LSTM, решающую задачу анализа эмоциональной окраски.

Реализация модели анализа эмоциональной окраски

Здесь мы попробуем проанализировать эмоциональную окраску рецензий на кинофильмы из базы данных Large Movie Review Dataset. Она состоит из 50 тысяч рецензий с сайта IMDB, каждая из которых

помечена как положительная или отрицательная с эмоциональной точки зрения. Мы возьмем простую модель LSTM с прореживанием, чтобы понять, как классифицировать эмоциональную окраску. Рецензии будут поступать в модель по слову за раз. По окончании процесса мы возьмем выходное значение модели за основу двоичной классификации, которая пометит эмоциональную окраску как «положительную» или «отрицательную». Начнем с загрузки базы данных. Воспользуемся вспомогательной библиотекой tflearn. Установить ее можно с помощью следующей команды:

```
$ pip install tflearn
```

Теперь можно загрузить базу данных, выбрать из словаря 30 тысяч самых распространенных слов, ограничить длину входных последовательностей 500 словами* и обработать метки:

```
from tflearn.data_utils import to_categorical, pad_sequences
from tflearn.datasets import imdb
train, test, _ = imdb.load_data(path='data/imdb.pkl',
      n_words=30000,
           valid_portion=0.1)
trainX, trainY = train
testX, testY = test
trainX = pad_sequences(trainX, maxlen=500, value=0.)
testX = pad_sequences(testX, maxlen=500, value=0.)
trainY = to_categorical(trainY, nb_classes=2)
testY = to_categorical(testY, nb_classes=2)
```

Теперь наши входные данные — 500-мерные векторы. Каждый соответствует кинорецензии, причем *i*-й компонент вектора сопоставлен индексу *i*-го слова в рецензии в нашем общем словаре из 30 тысяч единиц. Чтобы закончить подготовку, создадим особый класс Python для подачи минипакетов желаемого размера из основной базы данных:

```
class IMDBDataset():
      def __init__(self, X, Y):
           self.num_examples = len(X)
           self.inputs = X
           self.tags = Y
           self.ptr = 0
      def minibatch(self, size):
```

* Если длина рецензии меньше 500 слов, то она дополняется символами-заполнителями, как делалось для сетей с прямым распространением сигнала. *Прим. науч. ред.*

```
            ret = None
            if self.ptr + size < len(self.inputs):
            ret = self.inputs[self.ptr:self.ptr+size],
                   self.tags[self.ptr:self.ptr+size]
            else:
                   ret = np.concatenate((self.inputs[self.ptr:],
                          self.inputs[:size-len(
                          self.inputs[self.ptr:])])),
                   np.concatenate((self.tags[self.ptr:],
                          self.tags[:size-len(
                          self.tags[self.ptr:])]))
            self.ptr = (self.ptr + size) % len(self.inputs)
            return ret
train = IMDBDataset(trainX, trainY)
val = IMDBDataset(testX, testY)
```

Класс Python `IMDBDataset` используется для подачи как обучающих, так и проверочных данных, которые будут использоваться в обучении нашей модели анализа эмоциональной окраски.

Итак, данные готовы, приступим к пошаговому созданию модели анализа эмоциональной окраски. Для начала нужно привязать каждое слово из входной рецензии к вектору слов. Для этого используется слой плотного векторного представления, который, как вы наверняка помните из предыдущей главы, представляет собой простую таблицу соответствия. Она хранит вектор плотного представления, сопоставленный каждому слову. В отличие от предыдущих примеров, когда обучение векторных представлений слов мы рассматривали как отдельную проблему (и создавали, например, модель `Skip-Gram`), мы будем обучать векторные представления слов вместе с проблемой анализа эмоциональной окраски и считать матрицу векторных представлений матрицей параметров общей проблемы. Воспользуемся примитивами `TensorFlow` для управления плотными векторными представлениями (помните, что входные данные — это один полный мини-пакет, а не просто вектор кинорецензии):

```
def embedding_layer(input, weight_shape):
       weight_init = tf.random_normal_initializer(stddev=(
            1.0/weight_shape[0])**0.5)
       E = tf.get_variable("E", weight_shape,
            initializer=weight_init)
       incoming = tf.cast(input, tf.int32)
       embeddings = tf.nn.embedding_lookup(E, incoming)
       return embeddings
```

Затем берем результат из слоя векторных представлений слов и строим LSTM с прореживанием при помощи тех же примитивов, что и в предыдущем разделе. Проделаем небольшую дополнительную работу, чтобы сохранить последнее значение, выданное LSTM, при помощи операторов tf.slice и tf.squeeze, которые находят фрагмент, где содержится последний вывод LSTM, и устраняют ненужные измерения. Смена измерений выглядит так:

```
[batch_size, max_time, cell.output_size] to [batch_size, 1, cell.out
put_size] to [batch_size, cell.output_size].
```

Реализовать LSTM можно следующим образом:

```
def lstm(input, hidden_dim, keep_prob, phase_train):
      lstm = tf.nn.rnn_cell.BasicLSTMCell(hidden_dim)
      dropout_lstm = tf.nn.rnn_cell.DropoutWrapper(lstm,
            input_keep_prob=keep_prob,
            output_keep_prob=keep_prob)
      # stacked_lstm = tf.nn.rnn_cell.MultiRNNCell(
            [dropout_lstm] * 2,
            state_is_tuple=True)
      lstm_outputs, state = tf.nn.dynamic_rnn(dropout_lstm,
            input, dtype=tf.float32)
      return tf.squeeze(tf.slice(lstm_outputs,
            [0, tf.shape(
            lstm_outputs)[1]-1, 0],
            [tf.shape(lstm_outputs)[0],
            1, tf.shape(
            lstm_outputs)[2]])
```

В завершение мы добавляем скрытый слой пакетной нормализации, идентичный тем, которые мы использовали в предыдущих примерах. Собрав все компоненты вместе, мы можем перейти к построению графа логического вывода:

```
def inference(input, phase_train):
      embedding = embedding_layer(input, [30000, 512])
      lstm_output = lstm(embedding, 512, 0.5, phase_train)
      output = layer(lstm_output, [512, 2], [2], phase_train)
      return output
```

Опустим остальной шаблонный код, который сохраняет сводную статистику, промежуточные состояния и создает сессию: он такой же, как и остальные модели, уже созданные в этой книге; вы можете посмотреть исходный код в репозитории GitHub. Теперь можно запустить и визуализировать работу модели при помощи TensorBoard (рис. 7.23).

В начале обучения модель несколько нестабильна, а в конце явно происходит переобучение, поскольку ошибки на обучающем и проверочном наборах начинают сильно различаться. В период же оптимальной работы модель демонстрирует эффективные результаты и показывает на тестовом наборе данных аккуратность примерно 86%. Поздравляем! Вы создали свою первую рекуррентную нейронную сеть.

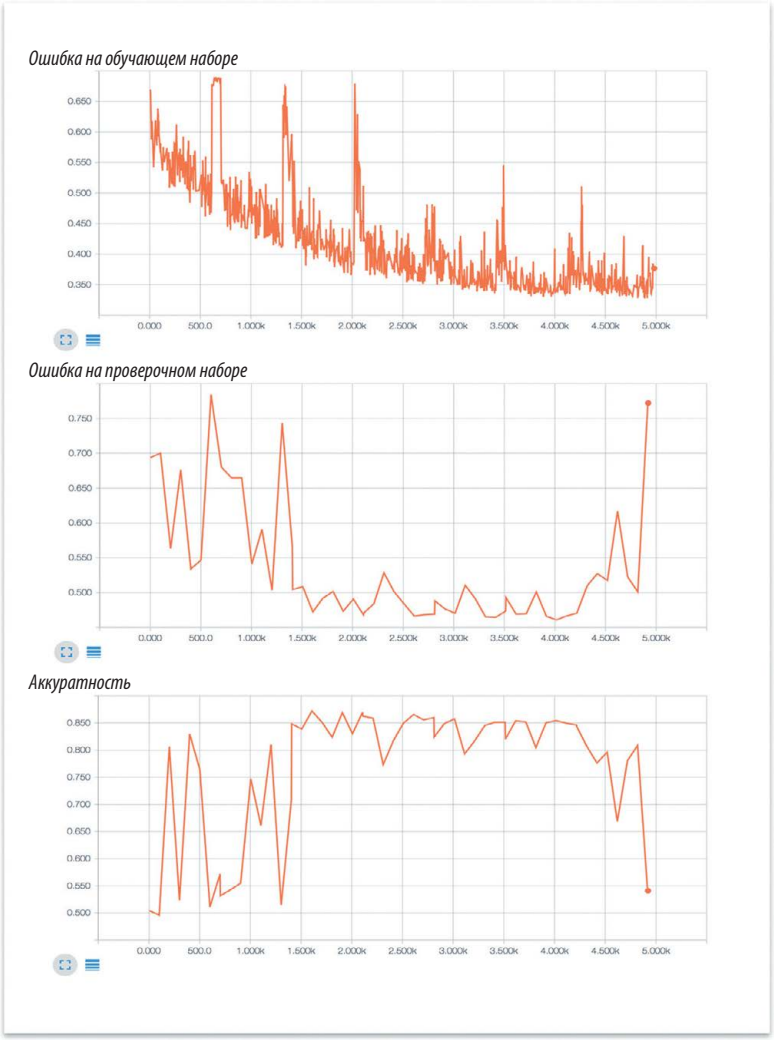


Рис. 7.23. *Ошибка на обучающем и проверочном наборах данных, а также аккуратность модели анализа эмоциональной окраски кинорецензий*

Решение задач класса seq2seq при помощи рекуррентных нейронных сетей

Теперь мы хорошо понимаем работу рекуррентных нейронных сетей и можем вернуться к проблеме `seq2seq`. Мы начали эту главу примером задачи класса `seq2seq` — привязкой последовательности слов в предложении к последовательности меток частей речи.

Работать с этой проблемой было можно, поскольку для создания соответствующих меток не требовалось учитывать долгосрочные зависимости. Но некоторые проблемы `seq2seq`, например перевод с одного языка на другой или создание резюме к видеофайлу, таковы, что долгосрочные зависимости в них крайне важны для успеха. И тут в дело вступают РНС.

Работа РНС с `seq2seq` во многом напоминает работу автокодера, о которой шла речь в предыдущей главе. Модель `seq2seq` состоит из двух отдельных сетей. Первая называется *`кодер (encoder)`*. Это рекуррентная сеть (обычно использующая нейроны LSTM), которая читает всю входную последовательность. Цель ее — создать сжатое понимание входных данных и выразить его в единой «мысли», представленной конечным состоянием сети. Затем используется *`декодер (decoder)`*, начальное состояние которой инициализируется конечным состоянием кодера. Символ за символом генерируется выходная последовательность. На каждом шаге декодер использует собственные выходные данные с предыдущего шага в качестве входных данных текущего шага. Весь процесс представлен на рис. 7.24.

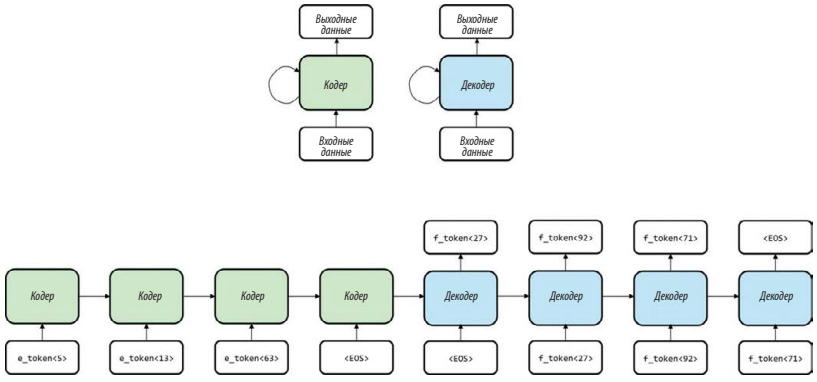


Рис. 7.24. *Иллюстрация использования схемы рекуррентной сети кодера-декодера для решения задач класса seq2seq*

В этом примере мы попробуем перевести предложение с английского на французский. Мы токенизируем входное предложение и используем плотное векторное представление (так же, как для модели анализа эмоциональных состояний, созданной в предыдущем разделе), загружая по слову в кодер. В конце предложения мы ставим специальную метку (`end of sentence`, `EOS`), которая отмечает конец входного предложения для кодера.

Теперь берем скрытое состояние последней для инициализации декодера. Первый элемент на входе в нее — метка `EOS`, а выходное значение интерпретируется как первое слово предсказанного французского перевода.

С этого момента мы используем выходные данные декодера как входные данные в нее же на следующем шаге. Продолжаем, пока кодер не выдаст на выходе метку `EOS`, что будет свидетельствовать об окончании работы над переводом исходного английского предложения. Мы проанализируем открытую рабочую реализацию этой сети (с парой улучшений и хитростей для повышения точности) ниже.

Архитектура RNC для `seq2seq` может быть использована и для обучения хорошим плотным векторным представлениям последовательностей. Например, Райан Кирос и коллеги в 2015 году ввели понятие вектора *`skip-thought`*[83], который по архитектурным характеристикам напоминал как структуру автокодера, так и модель `Skip-Gram` (см. главу 6). Вектор `skip-thought` был создан путем разбиения фрагмента на серию триплетов, состоящих из последовательных предложений. Авторы воспользовались одним кодером и двумя декодерами (рис. 7.25).

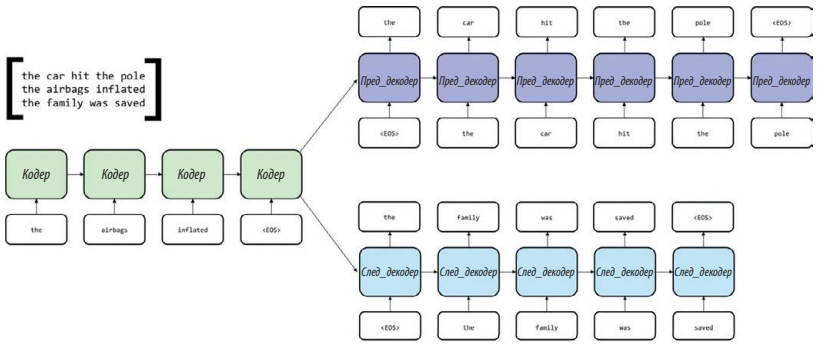


Рис. 7.25. *Архитектура skip-thought для seq2seq порождает компактные векторные представления для целых предложений*

Кодер прочел предложение, плотное векторное представление которого (хранимое в последнем конечном состоянии кодера) мы и намеревались

создать. Затем начался этап декодирования. Первая из сетей использовала это представление для инициализации собственного скрытого состояния и попыталась воссоздать предложение, предшествовавшее введенному. Вторая старалась воссоздать предложение, непосредственно следующее за введенным. Вся система была полностью обучена на этих триплетах и после завершения могла выдавать связные фрагменты текста; кроме того, повысилась эффективность решения ключевых задач на уровне предложений. Вот пример порождения рассказа из исходной статьи:

```
she grabbed my hand .
"come on . "
she fluttered her back in the air .
"i think we're at your place . I ca n't come get you . "
he locked himself back up
" no . she will . "
kyrian shook his head*
```

Теперь, поняв, как применять рекуррентные нейронные сети к задачам класса `seq2seq`, мы почти готовы создать свой вариант. Но сначала нужно разобраться еще с одной проблемой, к которой мы и перейдем в следующем разделе: идеей внимания в РНС для `seq2seq`.

Дополнение рекуррентных сетей вниманием

Продолжим разговор о проблемах перевода. Если вы когда-то пытались выучить иностранный язык, то знаете, что именно помогает успешному переводу. Во-первых, стоит полностью прочесть предложение, чтобы понять, какую идею нужно передать. Затем вы слово за словом записываете перевод, и каждое слово вытекает из предыдущего. При этом, составляя новое предложение, вы часто обращаетесь к исходному тексту, сосредоточиваясь на определенных фрагментах, которые важны для текущего перевода. На каждом шаге вы обращаете внимание на самые важные в данный момент части «входных данных», чтобы принять наилучшее решение по поводу следующего слова, которое должно будет появиться на бумаге.

---

* она взяла меня за руку
"давай…"
она потрясла спиной в воздухе
"я думаю, мы у тебя… я не могу тебя заставить…"
он снова закрылся
"нет, она будет…"
кириан покачал головой

Вернемся к нашему подходу к `seq2seq`. Прочтя полностью входные данные и резюмируя их в виде «мысли» в скрытом состоянии, сеть кодера фактически копирует первую часть процесса перевода. Используя предшествующие выходные данные как текущие входные, сеть декодера реализует вторую часть процесса. Это явление внимания еще не отражено в нашем подходе к `seq2seq`, так что сейчас мы этим и займемся.

Единственные входные данные декодера на шаге t — его выходные на шаге $(t - 1)$. Один из способов дать сети декодера представление об исходном предложении — предоставить доступ ко всему выводу кодера (который мы пока игнорировали). Эти выходные данные интересуют нас, поскольку отражают изменение внутреннего состояния кодера после поступления каждого нового токена. Предлагаемая реализация этой стратегии показана на рис. 7.26.

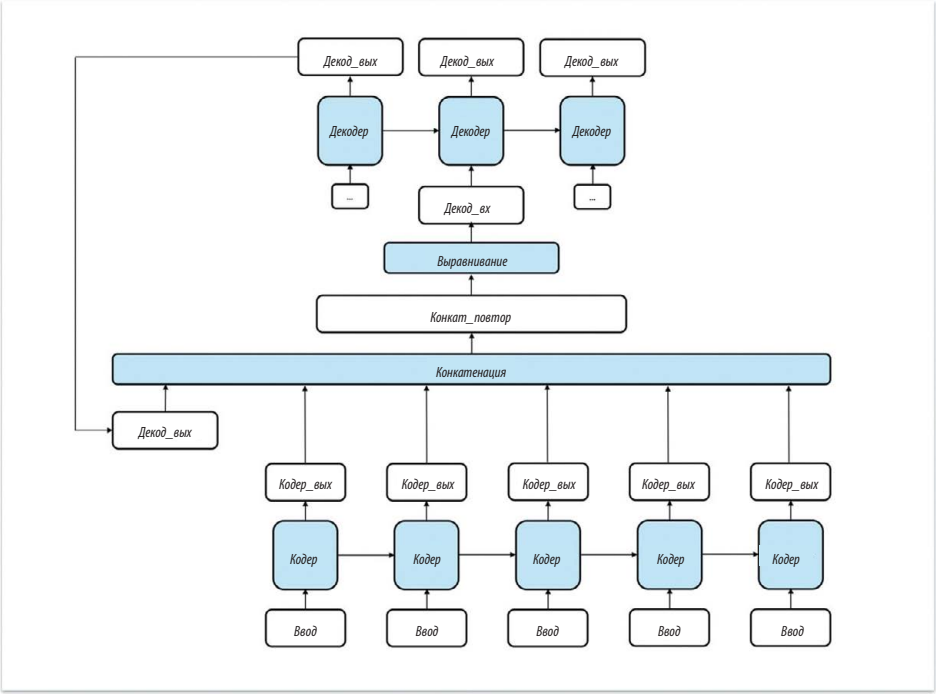


Рис. 7.26. *Попытка ввести внимание в архитектуру seq2seq. Нас ждет неудача, поскольку сеть не может динамически выбирать самые важные части входных данных и сосредоточиваться на них*

Но у этого подхода обнаруживается важный недостаток. На каждом шаге декодер рассматривает все выходные данные кодера одинаково. Человек

же в процессе перевода действует не так. Работая над разными фрагментами, мы сосредоточиваемся на разных аспектах оригинала. Важно понять, что дать декодеру доступ ко всем выходным данным недостаточно. Нужно придумать механизм, с помощью которого он мог бы динамически обращать внимание на конкретную подвыборку выходных данных кодера.

Проблему можно решить, подвергнув входные данные конкатенации. Поможет в этом предложение, внесенное в 2015 году Дмитрием Баданау и коллегами[84]. Вместо того чтобы непосредственно работать с сырыми выходными данными из кодера, мы присваиваем им веса. Для этого используем состояние сети декодера в момент $(t-1)$ как основу.

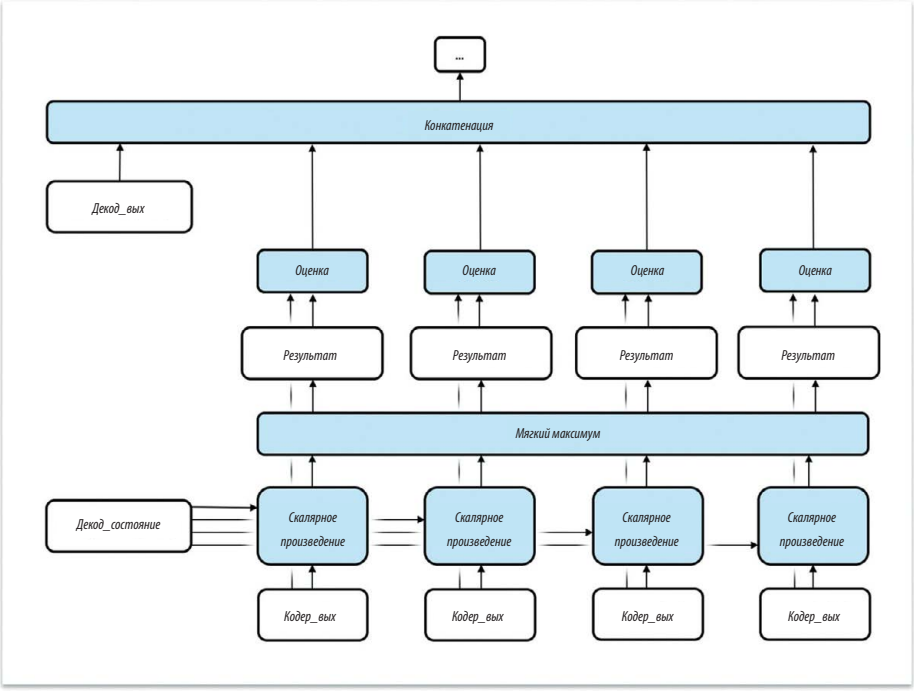


Рис. 7.27. *Внесение изменений в первичный вариант позволяет создать динамический механизм внимания на основе скрытого состояния сети декодера на предыдущем шаге*

Операция присвоения весов показана на рис. 7.27. Сначала назначаем скалярный (одно число, а не тензор) коэффициент релевантности для каждого выходного значения кодера. Для этого вычисляем скалярное произведение каждого вывода кодера и состояния декодера на шаге $(t-1)$. Затем нормализуем эти результаты с помощью операции мягкого максимума. Наконец,

с помощью нормализованных результатов индивидуально оцениваем все выходные значения кодера, прежде чем начать конкатенацию. Важно, что относительные показатели для каждого выходного значения кодера отражают степень его важности для решения декодера на шаге `t`. Позже мы покажем, как визуализировать то, какие элементы выходных данных наиболее важны для перевода на каждом шаге, с помощью анализа выходных данных операции мягкого максимума.

Вооружившись пониманием стратегии введения внимания в архитектуру `seq2seq`, мы готовы заняться построением модели PHC для перевода английских предложений на французский. Но сначала стоит отметить, что внимание очень важно и для других проблем, не связанных с переводом. Оно может сыграть свою роль в задаче распознавания языка, когда алгоритм обучается динамически обращать внимание на соответствующие части аудиофайла при его переводе в текст. Оно применимо для улучшения алгоритма описания изображений и дает ему возможность фокусироваться на конкретных фрагментах входного изображения при создании описания. Если существуют отдельные элементы входных данных, которые тесно связаны с верным воспроизводством соответствующих сегментов выходных данных, внимание может существенно повысить производительность.

Разбор нейронной сети для перевода

Современные нейронные сети для перевода используют ряд методов и достижений, основанных всё на той же архитектуре кодера-декодера `seq2seq`. Внимание, рассмотренное в предыдущем разделе, — жизненно важное новшество. Ниже мы разберем полностью реализованную нейронную систему машинного перевода, дополним ее обработкой данных, построим модель, обучим ее и используем в качестве системы перевода английских фраз на французский! Работать мы будем с упрощенной версией официального учебного кода для машинного перевода `TensorFlow`[85].

Обучение и последующее использование нейронной системы машинного перевода очень похожи на большинство других процессов в области машинного обучения: сбор данных, их подготовка, построение модели, ее обучение, оценка ее качества и, наконец, использование обученной модели для получения полезных предсказаний или выводов. Все эти этапы мы и рассмотрим.

Сначала мы получаем данные из репозитория `WMT'15`, который содержит большие массивы, используемые для обучения систем перевода.

Мы воспользуемся англо-французскими данными. Заметим, что если нужно будет переводить с нескольких или на несколько языков, то придется с нуля обучать модель на новых данных. Предварительно обрабатываем данные, преобразуя их в формат, который модели смогут использовать при обучении и логическом выводе. Для этого надо провести чистку и разбиение на токены предложений в каждой английской и французской фразе. Мы приведем методы, используемые в подготовке данных, и поговорим об их реализации.

Первый шаг — преобразовать предложения и фразы в совместимый с моделью формат путем *разбиения на токены*, или составляющие (и для английских, и для французских фраз). Например, простой пословный токенизатор, получив предложение `I read`. («я читаю»), выдаст набор `["I", "read","."]`, а из французского предложения `Je lis.` сделает набор `["Je", "lis","."]`. Посимвольный токенизатор разобьет предложение на отдельные символы или пары символов, получится `["I", " ", "r", "e", "a", "d", "."]` и `["I ","" "re", "ad", "."]` соответственно.

В каждом случае нужно решить, какое разбиение лучше: у обоих есть свои достоинства и недостатки. Например, пословный токенизатор обеспечит выдачу слов из словаря, но размер последнего может быть слишком велик для эффективного выбора во время декодирования. Эта проблема известна, мы обратимся к ней позже. Токенизатор на основе посимвольного разбиения не всегда может выдать читабельные данные, но словарь, из которого декодер выбирает варианты, гораздо меньше: это набор всех печатаемых символов `ASCII`. Мы используем пословное разбиение, но читателю предлагаем поэкспериментировать с разными вариантами и посмотреть на результаты. Нам придется добавить `EOS` — особый символ конца последовательности — по окончании всех входных последовательностей, поскольку нам нужен способ указать декодеру, что он завершил работу. Обычную пунктуацию использовать нельзя, ведь мы не можем по умолчанию считать, что переводим полные предложения. В наших входных последовательностях символы `EOS` не нужны, поскольку они уже отформатированы и нам не надо специально отмечать конец каждой исходной последовательности.

Следующий шаг — новые изменения представления каждого входного и выходного предложения. Здесь мы вводим идею *группирования*. Этот метод используется в основном в задачах вроде `seq2seq`, особенно при машинном переводе, и помогает модели эффективно обрабатывать предложения или фразы разной длины. Сначала рассмотрим простейший метод ввода обучающих данных и покажем его недостатки. При введении токенов в кодер и декодер длина исходной и целевой последовательностей

в парах данных для обучения не всегда идентична. Например, исходная последовательность может иметь длину `X`, а целевая — `Y`. Кажется, что нам нужны разные сети `seq2seq` для каждой пары `(X, Y)`, что сразу можно расценить как неэффективную трату сил и времени. Чуть лучше пойдут дела, если мы дополним каждую последовательность до определенной длины, как показано на рис. 7.28 (считаем, что мы используем пословное разбиение и уже добавили в целевые последовательности метки `EOS`).

I	read	.	<PAD>	<PAD>	<PAD>	<PAD>
Je	lis	.	<EOS>	<PAD>	<PAD>	<PAD>
See	you	in	a	little	while	.
A	tout	a	l'heure	<EOS>	<PAD>	<PAD>
			...			

Рис. 7.28. *Наивная стратегия дополнения последовательностей*

Этот шаг дает возможность не создавать разные модели `seq2seq` для каждой пары исходной и целевой длины. Но при этом возникает проблема иного рода: если найдется очень длинная последовательность, все остальные придется дополнять *до этой длины*. И дополненная короткая последовательность будет требовать столько же вычислительных ресурсов, что и длинная с небольшим количеством заполнителей, а это приведет к трате сил и может крайне негативно сказаться на производительности нашей модели. Допустимо разбить каждую последовательность в корпусе на фразы, чтобы длина каждой из них не превышала определенного максимального значения, но не вполне понятно, как разбивать переводы. Здесь-то и поможет группирование.

Группирование основано на том, что пары кодера и декодера можно поместить в группы сходного размера и дополнять до максимальной длины последовательности в каждой соответствующей группе. Например, допустимо обозначить ряд групп `[(5, 10), (10, 15), (20, 25), (30,40)]`, где каждая запись — максимальная длина исходной и целевой последовательностей соответственно. Если вернуться к предыдущему примеру, то можно поместить пару последовательностей `(["I", "read", "."], ["Je", "lis", ".", "EOS"])` в первую группу, поскольку исходное предложение меньше пяти токенов, а целевое — меньше 10. Во вторую группу помещаем набор `(["See", "you", "in", "a", "little", "while"], ["A", "tout", "a","l'heure", "EOS"])` и т. д. Этот метод позволяет найти компромисс между двумя экстремумами: теперь дополнять можно ровно столько, сколько необходимо, что и показано на рис. 7.29.

	I	read	.	<PAD>			
Bucket i	Je	lis	.	<EOS>			
			...				
...							
	See	you	in	a	little	while	.
Bucket j	A	tout	a	l'heure	<EOS>	<PAD>	<PAD>
			...				

Рис. 7.29. *Дополнение последовательностей в группах*

Группирование позволяет существенно ускорить время обучения и тестирования, а разработчикам дает возможность писать оптимизированный код, используя то обстоятельство, что все последовательности из одной группы имеют один размер, и упаковывать данные для повышения эффективности работы GPU.

Как только все дополнения последовательностей выполнены, добавляем в целевые последовательности еще один токен — *GO*. Он говорит декодеру, что пора начать декодирование. Тот действует соответственно.

Последнее, что нужно делать во время подготовки данных, — переворачивать исходные предложения. Исследователи установили, что это повышает эффективность, и это действие стало стандартным при обучении нейронных моделей машинного перевода. Это своего рода трюк, но вспомните, что наше нейронное состояние фиксированного размера и, соответственно, не может содержать информации больше определенного предела. Получается, информация, закодированная при обработке начала предложения, может подвергнуться перезаписи при кодировании последующих фрагментов.

Во многих языковых парах начало предложений перевести сложнее, чем конец, и переворачивание повышает точность: последнее слово на финальной стадии кодирования остается за началом последовательности. После реализации всех этих идей конечные последовательности должны выглядеть так, как на рис. 7.30.

	<PAD>	<PAD>	.	read	I		
Bucket i	<GO>	Je	lis	.	<EOS>		
			...				
...							
	.	while	little	in	a	you	See
Bucket j	<GO>	A	tout	a	l'heure	<EOS>	<PAD>
			...				

Рис. 7.30. *Итоговая схема дополнения в группах после переворачивания данных и добавления метки GO*

Описав эти техники, мы можем перейти к подробностям реализации. Идеи реализованы в методе `get_batch()` кода. Он получает отдельный пакет обучающих данных, учитывая `bucket_id`, который берется из цикла обучения. В результате создаются токены для исходного и целевого предложений, метод задействует все описанные выше техники, включая дополнение в группах и переворачивание ввода:

```
def get_batch(self, data, bucket_id):
      encoder_size, decoder_size = self.buckets[bucket_id]
      encoder_inputs, decoder_inputs = [], []
```

Сначала мы задаем заполнители для каждого входного значения, поступающего в кодер и декодер:

```
for _ in xrange(self.batch_size):
      encoder_input, decoder_input = random.choice(data[
      bucket_id])
      # Encoder inputs are padded and then reversed. (Входные данные
           кодера дополняются и переворачиваются)
      encoder_pad = [data_utils.PAD_ID] * (encoder_size - len(
           encoder_input))
      encoder_inputs.append(list(reversed(encoder_input +
           encoder_pad)))
      # Decoder inputs get an extra "GO" symbol,
      # and are then padded. (Входные данные декодера получают
      дополнительный символ "GO" и затем переворачиваются)
      decoder_pad_size = decoder_size - len(decoder_input) - 1
      decoder_inputs.append([data_utils.GO_ID] + decoder_input +
           [data_utils.PAD_ID] *
           decoder_pad_size)
```

В соответствии с размером пакета мы берем соответствующее количество кодирующих и декодирующих последовательностей:

```
# Now we create batch-major vectors from the data selected
# above. (Теперь создаем пакетные векторы на основании выбранных выше
данных)
batch_encoder_inputs, batch_decoder_inputs, batch_weights =
[], [], []
# Batch encoder inputs are just re-indexed encoder_inputs. (Пакетные
      вводы кодера – переформатированные encoder_inputs)
for length_idx in xrange(encoder_size):
batch_encoder_inputs.append(
np.array([encoder_inputs[batch_idx][length_idx]
```

```
for batch_idx in xrange(self.batch_size)],
dtype=np.int32))
# Batch decoder inputs are re-indexed decoder_inputs,
# we create weights. (Пакетные вводы декодера — переформатированные
      decoder_inputs, мы назначаем веса)
for length_idx in xrange(decoder_size):
      batch_decoder_inputs.append(
            np.array([decoder_inputs[batch_idx][length_idx]
                  for batch_idx in xrange(self.batch_size)],
                        dtype=np.int32))
```

Убеждаемся, что первое измерение тензора — размер мини-пакета, и преобразуем определенные ранее заполнители в нужный формат:

```
# Create target_weights to be 0 for targets that
# are padding. (Назначаем target_weights равными 0 для дополнений
      в целевой последовательности)
      batch_weight = np.ones(self.batch_size, dtype=np.float32)
      for batch_idx in xrange(self.batch_size):
# We set weight to 0 if the corresponding target is
# a PAD symbol. (Назначаем вес 0, если соответствующая цель — символ PAD)
# The corresponding target is decoder_input shifted
# by 1 forward. (Соответствующая цель — decoder_input со сдвигом вперед на 1)
      if length_idx < decoder_size — 1:
            target = decoder_inputs[batch_idx][length_idx + 1]
      if length_idx == decoder_size — 1 or
            target == data_utils.PAD_ID:
      batch_weight[batch_idx] = 0.0
      batch_weights.append(batch_weight)
            return batch_encoder_inputs, batch_decoder_inputs,
      batch_weights
```

Наконец, назначаем нулевые целевые веса для токенов-заполнителей. Подготовка данных завершена, и мы можем начать строить и обучать модель! Сначала определяем код, используемый при обучении и тестировании, на время абстрагируясь от модели. Так можно убедиться, что мы понимаем процесс в общих чертах. Потом можно перейти к более глубокому анализу модели `seq2seq`. Как всегда, первый шаг в обучении — загрузка данных:

```
def train():
"""Train a en->fr translation model using WMT data."""
# Prepare WMT data. (Готовим данные WMT)
```

```
print("Preparing WMT data in %s" % FLAGS.data_dir)
en_train, fr_train, en_dev, fr_dev, _, _ =
data_utils.prepare_wmt_data(
FLAGS.data_dir, FLAGS.en_vocab_size, FLAGS.fr_vocab_size)
```

Запустив сессию `TensorFlow`, первым делом создаем модель. Отметим, что этот метод применим к ряду архитектур, если они удовлетворяют требованиям к формату входных и выходных данных, описанным в методе `train()`:

```
with tf.Session() as sess:
# Create model. (Создаем модель)
print("Creating %d layers of %d units." % (FLAGS.num_layers,
FLAGS.size))
model = create_model(sess, False)
```

Теперь при помощи разных функций группируем данные; группы позже будут использованы `get_batch()` для извлечения информации. Создаем набор действительных чисел от 0 до 1, который будет приблизительно определять вероятность выбора группы, нормализованную на размер последней. Метод `get_batch()` выбирает группы с учетом этих вероятностей:

```
# Read data into buckets and compute their sizes. (Группируем данные
      и вычисляем размеры групп)
print ("Reading development and training data (limit: %d)."
% FLAGS.max_train_data_size)
dev_set = read_data(en_dev, fr_dev)
train_set = read_data(en_train, fr_train,
FLAGS.max_train_data_size)
train_bucket_sizes = [len(train_set[b]) for b in xrange(
len(_buckets))]
train_total_size = float(sum(train_bucket_sizes))
# A bucket scale is a list of increasing numbers
# from 0 to 1 that we'll use to select a bucket. (Масштаб группы — список
      чисел от 0 до 1 по возрастающей, используемых для выбора группы)
# Length of [scale[i], scale[i+1]] is proportional to
# the size if i-th training bucket, as used later. (Длина [scale[i],
      scale[i+1]] пропорциональна размеру при i-й обучающей группе,
      как показано ниже)
train_buckets_scale = [sum(train_bucket_sizes[:i + 1]) /
train_total_size
```

```
for i in xrange(len(
train_bucket_sizes))]
```

Теперь, когда данные готовы, запускаем основной цикл обучения. Мы инициализируем различные его переменные — например, `current_step` и `previous_losses`, делая их пустыми или равными 0. Важно отметить, что каждая итерация цикла соотносится с одной эпохой — полной обработкой пакета обучающих данных. Для эпохи мы выбираем `bucket_id`, получаем пакет при помощи `get_batch`, а затем делаем *шаг вперед* в рамках модели:

```
# This is the training loop. (Это цикл обучения)
step_time, loss = 0.0, 0.0
current_step = 0
previous_losses = []
while True:
# Choose a bucket according to data distribution. (Выбираем группу
      на основании распределения данных)
# We pick a random number
# in [0, 1] and use the corresponding interval
# in train_buckets_scale. (Берем случайное число из [0, 1] и используем
      соответствующий интервал в train_buckets_scale)
random_number_01 = np.random.random_sample()
bucket_id = min([i for i in xrange(len(
train_buckets_scale))
if train_buckets_scale[i] >
random_number_01])
# Get a batch and make a step. (Получаем пакет и делаем шаг)
start_time = time.time()
encoder_inputs, decoder_inputs, target_weights =
model.get_batch(
train_set, bucket_id)
_, step_loss, _ = model.step(sess, encoder_inputs,
decoder_inputs,
target_weights, bucket_id,
False)
```

Оцениваем потери при предсказании и запоминаем все остальные метрики:

```
step_time += (time.time() - start_time) /
FLAGS.steps_per_checkpoint
loss += step_loss / FLAGS.steps_per_checkpoint
current_step += 1
```

Наконец, после некоторого числа итераций, которое предписывается глобальной переменной, нам нужно выполнить ряд дополнительных операций. Сначала выводим статистику для предыдущего пакета: это размер функции потери, темп обучения и перплексия*. Если потери не уменьшаются, возможно, модель оказалась в локальном минимуме. Чтобы помочь ей этого избежать, снижаем темп обучения: тогда не будет больших скачков ни в одном направлении. В это же время мы сохраняем копию модели с весами и активациями на диск:

```
# Once in a while, we save checkpoint, print statistics,
# and run evals. (В определенный момент сохраняем контрольную точку,
      выводим статистику и проводим оценку)
if current_step % FLAGS.steps_per_checkpoint == 0:
# Print statistics for the previous epoch. (Выводим статистику для
      предыдущей эпохи)
perplexity = math.exp(float(loss)) if loss <
300 else float("inf")
print ("global step %d learning rate %.4f
step-time %.2f perplexity "
"%.2f" % (model.global_step.eval(),
model.learning_rate.eval(),
step_time, perplexity))
# Decrease learning rate if no improvement was seen over
# last 3 times. (Снижаем темп обучения, если за последние 3 раза не
      было улучшений)
if len(previous_losses) > 2 and loss > max(
previous_losses[-3:]):
sess.run(model.learning_rate_decay_op)
previous_losses.append(loss)
# Save checkpoint and zero timer and loss. (Сохраняем контрольную
      точку, нулевой таймер и величину потерь)
checkpoint_path = os.path.join(FLAGS.train_dir,
"translate.ckpt")
model.saver.save(sess, checkpoint_path,
global_step=model.global_step)
step_time, loss = 0.0, 0.0
```

* Одна из популярных мер оценки качества языковой модели. Перплексия языковой модели на наборе данных — обратная вероятность этого набора, нормализованная по числу слов. Ее можно понимать как коэффициент «ветвления»: сколько в среднем разных токенов может быть после каждого токена в последовательности. *Прим. науч. ред.*

Теперь измеряем производительность нашей модели на тестовом наборе данных. Тем самым мы определяем обобщающую способность модели, чтобы узнать, повышается ли она, и если да, то насколько. Мы снова получаем данные с помощью `get_batch`, но на этот раз используем `bucket_id` из тестового набора. Снова делаем шаг вперед в рамках модели, но не обновляем веса, поскольку последний аргумент в методе `step()` — `True` — противоположен `False` во время основного цикла обучения; семантику `step()` мы рассмотрим позже. Измеряем потери на тестовом наборе данных и выводим их пользователю:

```
# Run evals on development set and print
# their perplexity. (Запускаем оценки качества на наборе данных для
      тестирования, печатаем перплексию)
for bucket_id in xrange(len(_buckets)):
if len(dev_set[bucket_id]) == 0:
print(" eval: empty bucket %d" % (bucket_id))
continue
encoder_inputs, decoder_inputs,
target_weights = model.get_batch(
dev_set, bucket_id)
# attns, _, eval_loss, _ = model.step(sess,
encoder_inputs, decoder_inputs,
_, eval_loss, _ = model.step(sess, encoder_inputs,
decoder_inputs,
target_weights,
bucket_id,
True)
eval_ppx = math.exp(float(eval_loss)) if eval_loss <
300 else float(
"inf")
print(" eval: bucket %d perplexity %.2f" % (
bucket_id, eval_ppx))
sys.stdout.flush()
```

Есть еще один подходящий сценарий использования нашей модели — однократное предсказание. Мы хотим иметь возможность использовать обученную модель для перевода новых предложений — наших или других пользователей. Для этого пользуемся методом `decode()`. Он включает примерно те же функции, что были реализованы в цикле оценки на тестовом наборе данных. Основное различие в том, что при обучении и оценке нам не требовалось переводить выходные векторные представления в понятные людям токены, а этим мы сейчас и займемся. Изложим метод подробно.

Поскольку это отдельный режим вычислений, необходимо снова запустить сессию TensorFlow и либо создать модель, либо загрузить уже сохраненную в предыдущей контрольной точке:

```
def decode():
with tf.Session() as sess:
# Create model and load parameters. (Создаем модель и загружаем
      параметры)
model = create_model(sess, True)
```

Мы назначаем размер пакета 1, поскольку параллельно не обрабатываем новых предложений, а загружаем только входные и выходные словари, но не сами данные:

```
model.batch_size = 1
# We decode one sentence at a time. (Декодируем по предложению за раз)
# Load vocabularies. (Загружаем словари)
en_vocab_path = os.path.join(FLAGS.data_dir,
"vocab%d.en" %
FLAGS.en_vocab_size)
fr_vocab_path = os.path.join(FLAGS.data_dir,
"vocab%d.fr" %
FLAGS.fr_vocab_size)
en_vocab, _ = data_utils.initialize_vocabulary(
en_vocab_path)
_, rev_fr_vocab = data_utils.initialize_vocabulary(
fr_vocab_path)
```

Читаем входные данные со стандартного ввода, чтобы запрашивать предложения у пользователя:

```
# Decode from standard input. (Декодировать стандартный ввод)
sys.stdout.write("> ")
sys.stdout.flush()
sentence = sys.stdin.readline()
```

Если полученное предложение не пустое, оно разбивается на токены и обрезается при превышении определенной длины:

```
while sentence:
# Get token-ids for the input sentence. (Получаем токены для входящего
      предложения)
token_ids = data_utils.sentence_to_token_ids(
tf.compat.as_bytes(sentence), en_vocab)
# Which bucket does it belong to? (К какой группе оно принадлежит?)
```

```
bucket_id = len(_buckets) - 1
for i, bucket in enumerate(_buckets):
if bucket[0] >= len(token_ids):
bucket_id = i
break
else:
logging.warning("Sentence truncated: %s", sentence)
```

Хотя мы не загружаем никаких данных, `get_batch()` преобразует введенные данные в нужный формат и готовит к использованию в `step()`:

```
# Get a 1-element batch to feed the sentence to
# the model. (Получаем одноэлементный пакет для ввода предложения
      в модель)
encoder_inputs, decoder_inputs, target_weights =
model.get_batch(
{bucket_id: [(token_ids, [])]}, bucket_id)
```

Мы делаем шаг вперед в рамках модели. На этот раз нам нужны `output_logits`, то есть ненормализованные вероятности выходных токенов, а не величина функции потерь. Декодируем их при помощи выходного словаря и останавливаем декодирование при первом появлении метки `EOS`.

Теперь выводим французское предложение или фразу пользователю и ожидаем следующую фразу:

```
# Get output logits for the sentence. (Получаем логиты вывода для
      предложения)
_, _, output_logits = model.step(sess, encoder_inputs,
decoder_inputs,
target_weights,
bucket_id, True)
# This is a greedy decoder - outputs are just argmaxes
# of output_logits. (Это жадный декодер: выводы - просто номера
      максимумов в output_logits)
outputs = [int(np.argmax(logit, axis=1))
for logit in output_logits]
# If there is an EOS symbol in outputs, cut them
# at that point. (Если в выводах есть символ EOS, обрезаем их
      в этом месте)
if data_utils.EOS_ID in outputs:
outputs = outputs[:outputs.index(data_utils.EOS_ID)]
# Print out French sentence corresponding to outputs. (Выдаем
      французское предложение, соответствующее выводам)
```

```
print(" ".join([tf.compat.as_str(rev_fr_vocab[output])
for output in outputs]))
print("> ", end="")
sys.stdout.flush()
sentence = sys.stdin.readline()
```

На этом мы заканчиваем высокоуровневое рассмотрение обучения и использования моделей. Мы сильно абстрагировались от самой модели, и некоторым пользователям достаточно и этого. Но следует рассказать и об особенностях функции `step()`. Она отвечает за расчет целевой функции модели, соответствующее обновление весов и задание графа вычислений для модели. Начнем с первого.

Функция `step()` получает несколько аргументов: сессия `TensorFlow`, список векторов для ввода в кодер, входные данные для декодера, целевые веса, выбранное во время обучения значение `bucket_id` и логический флаг `forward_only`, который определяет, использовать градиентную оптимизацию для обновления весов или заморозить их. Отметим, что изменение этого флага с `False` на `True` позволило нам декодировать произвольное предложение и оценить работу модели на тестовом наборе данных:

```
def step(self, session, encoder_inputs, decoder_inputs,
target_weights, bucket_id, forward_only):
```

После нескольких защитных проверок того, что все векторы имеют совместимые размеры, мы готовим функции для прямого и обратного прохода. Метод прямого прохода получает всю информацию, изначально переданную функции `step()`, то есть необходимую при вычислении величины ошибки для одного примера:

```
# Check if the sizes match. (Проверяем, сходятся ли размеры)
encoder_size, decoder_size = self.buckets[bucket_id]
if len(encoder_inputs) != encoder_size:
raise ValueError("Encoder length must be equal to the one
in bucket,"
" %d != %d." % (len(
encoder_inputs), encoder_size))
if len(decoder_inputs) != decoder_size:
raise ValueError("Decoder length must be equal to the one
in bucket,"
" %d != %d." % (len(decoder_inputs),
decoder_size))
if len(target_weights) != decoder_size:
raise ValueError("Weights length must be equal to the one
```

```
in bucket,"
" %d != %d." % (len(target_weights),
decoder_size))
# Input feed: encoder inputs, decoder inputs, target_weights,
# as provided. (Ввод: вводы кодера, вводы декодера, target_weights
      в данном порядке)
input_feed = {}
for l in xrange(encoder_size):
input_feed[self.encoder_inputs[l].name] = encoder_inputs[l]
for l in xrange(decoder_size):
input_feed[self.decoder_inputs[l].name] = decoder_inputs[l]
input_feed[self.target_weights[l].name] = target_weights[l]
# Since our targets are decoder inputs shifted by one,
# we need one more. (Поскольку наши цели — вводы декодера со сдвигом 1,
нужна еще одна цель)
last_target = self.decoder_inputs[decoder_size].name
input_feed[last_target] = np.zeros([self.batch_size],
dtype=np.int32)
```

Метод обратного прохода, если ошибка рассчитана и необходимо ее распространить обратно по сети, содержит операцию обновления, которая выполняет стохастический градиентный спуск, вычисляет норму градиента и потери на пакет:

```
# Output feed: depends on whether we do a backward step or
# not. (Вывод: зависит от того, делается шаг назад или нет)
if not forward_only:
output_feed = [self.updates[bucket_id], # Update Op that
# does SGD. (Операция обновления запускает обратный градиентный спуск)
self.gradient_norms[bucket_id], # Gradient
# norm. (норма градиента)
self.losses[bucket_id]] # Loss for this
# batch. (Потери на пакет)
else:
output_feed = [self.losses[bucket_id]] # Loss for this
# batch. (потери на пакет)
for l in xrange(decoder_size): # Output logits. (Исходящие логиты)
output_feed.append(self.outputs[bucket_id][l])
```

Два этих метода передаются в `session.run()`. В зависимости от флага `forward_only` возвращаются либо норма градиента и ошибка для сбора статистики, либо выходные данные для декодирования:

```
outputs = session.run(output_feed, input_feed)
if not forward_only:
return outputs[1], outputs[2], None #, attns
# Gradient norm, loss, no outputs. (Норма градиента, потери,
      без исходящих)
else:
return None, outputs[0], outputs[1:] #, attns
# No gradient norm, loss, outputs. (Без нормы градиента, потерь,
      исходящие данные)
```

Теперь можно изучить саму модель. Ее конструктор задает граф вычислений на основе созданных высокоуровневых компонентов. Сначала дадим краткий обзор метода `create_model()`, вызывающего этот конструктор, а затем подробно рассмотрим сам конструктор.

Метод `create_model()` сам по себе несложен: он задействует ряд определенных пользователем или заданных по умолчанию флагов, таких как размеры английского и французского словарей и размер пакета для создания модели с помощью конструктора `seq2seq_model.Seq2SeqModel`. Очень интересен здесь флаг `use_fp16 flag`. Используется тип данных в массивах `NumPy` с пониженной точностью, что приводит к ускорению работы ценой некоторых потерь в точности. Но часто 16-битных представлений потерь и обновлений градиента достаточно: они нередко близки по величине к 32-битным. Создать модель можно при помощи следующего кода:

```
def create_model(session, forward_only):
"""Create translation model and initialize or
load parameters in session."""  (Создаем модель перевода
      и инициализируем или загружаем параметры в сеансе)
dtype = tf.float16 if FLAGS.use_fp16 else tf.float32
model = seq2seq_model.Seq2SeqModel(
FLAGS.en_vocab_size,
FLAGS.fr_vocab_size,
_buckets,
FLAGS.size,
FLAGS.num_layers,
FLAGS.max_gradient_norm,
FLAGS.batch_size,
FLAGS.learning_rate,
FLAGS.learning_rate_decay_factor,
forward_only=forward_only,
dtype=dtype)
```

Прежде чем вернуть эту модель, проверяем, нет ли других, сохраненных в контрольных точках и оставшихся с предыдущих обучающих запусков. Такая модель и ее параметры читаются в переменную и используются. Это позволяет прекратить обучение в контрольной точке и возобновить его не с нуля. Иначе в качестве основного объекта возвращается вновь созданная модель:

```
ckpt = tf.train.get_checkpoint_state(FLAGS.train_dir)
if ckpt and tf.train.checkpoint_exists(
ckpt.model_checkpoint_path):
print("Reading model parameters from %s" (Чтение параметров модели)
% ckpt.model_checkpoint_path)
model.saver.restore(session, ckpt.model_checkpoint_path)
else:
print("Created model with fresh parameters.") (Созданная модель
      с новыми параметрами)
session.run(tf.global_variables_initializer())
return model
```

Теперь рассмотрим конструктор `seq2seq_model.Seq2SeqModel`. Он создает весь граф вычислений и иногда вызывает определенные низкоуровневые конструкции. Прежде чем перейти к деталям, изучим код сверху вниз и детали общего графа вычислений.

Те же аргументы, сообщенные `create_model()`, передаются и этому конструктору, создаются несколько полей уровня класса:

```
class Seq2SeqModel(object):
def __init__(self,
source_vocab_size,
target_vocab_size,
buckets,
size,
num_layers,
max_gradient_norm,
batch_size,
learning_rate,
learning_rate_decay_factor,
use_lstm=False,
num_samples=512,
forward_only=False,
dtype=tf.float32):
self.source_vocab_size = source_vocab_size
```

```
self.target_vocab_size = target_vocab_size
self.buckets = buckets
self.batch_size = batch_size
self.learning_rate = tf.Variable(
float(learning_rate), trainable=False, dtype=dtype)
self.learning_rate_decay_op = self.learning_rate.assign(
self.learning_rate * learning_rate_decay_factor)
self.global_step = tf.Variable(0, trainable=False)
```

Следующая часть создает семплированную функцию мягкого максимума и проекцию вывода. Это улучшение по сравнению с базовыми моделями `seq2seq`, которое позволяет эффективно декодировать большие выходные словари и проецировать выходные логиты в нужное пространство:

```
# If we use sampled softmax, we need an output projection. (Если мы используем
      семплированную функцию мягкого максимума, нужна проекция вывода)
output_projection = None
softmax_loss_function = None
# Sampled softmax only makes sense if we sample less than
# vocabulary size. (Семплированная функция мягкого максимума нужна,
      только если мы делаем выборку меньше размера словаря)
if num_samples > 0 and num_samples <
self.target_vocab_size:
w_t = tf.get_variable("proj_w", [self.target_vocab_size,
size], dtype=dtype)
w = tf.transpose(w_t)
b = tf.get_variable("proj_b", [self.target_vocab_size],
dtype=dtype)
output_projection = (w, b)
def sampled_loss(inputs, labels):
labels = tf.reshape(labels, [-1, 1])
# We need to compute the sampled_softmax_loss using
# 32bit floats to avoid numerical instabilities. (Нужно вычислить значение
      семплированной функции мягкого максимума при помощи 32-битных
      плавающих запятых во избежание числовых нестабильностей)
local_w_t = tf.cast(w_t, tf.float32)
local_b = tf.cast(b, tf.float32)
local_inputs = tf.cast(inputs, tf.float32)
return tf.cast(
tf.nn.sampled_softmax_loss(local_w_t, local_b,
```

```
local_inputs, labels,
num_samples,
self.target_vocab_size),
dtype)
softmax_loss_function = sampled_loss
```

На основании флагов выбираем соответствующий нейрон РНС. Это может быть `GRU`, обычный или многослойный нейрон `LSTM`. На практике однослойные нейроны `LSTM` используются редко, но их гораздо быстрее обучать и они могут ускорить цикл отладки:

```
# Create the internal multi-layer cell for our RNN. (Создаем внутреннюю
      многослойную ячейку для РНС)
single_cell = tf.nn.rnn_cell.GRUCell(size)
if use_lstm:
single_cell = tf.nn.rnn_cell.BasicLSTMCell(size)
cell = single_cell
if num_layers > 1:
cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] *
num_layers)
```

Рекуррентная функция `seq2seq_f()` определяется с `seq2seq.embedding_attention_seq2seq()`, о которой мы поговорим позже:

```
# The seq2seq function: we use embedding for the
# input and attention. (Функция seq2seq: для ввода и внимания
      используем вложение)
def seq2seq_f(encoder_inputs, decoder_inputs, do_decode):
return seq2seq.embedding_attention_seq2seq(
encoder_inputs,
decoder_inputs,
cell,
num_encoder_symbols=source_vocab_size,
num_decoder_symbols=target_vocab_size,
embedding_size=size,
output_projection=output_projection,
feed_previous=do_decode,
dtype=dtype)
```

Определяем заполнители для входных и выходных данных:

```
# Feeds for inputs. (Значения ввода)
self.encoder_inputs = []
self.decoder_inputs = []
```

```
self.target_weights = []
for i in xrange(buckets[-1][0]): # Last bucket is
# the biggest one. (Последняя группа — самая большая)
self.encoder_inputs.append(tf.placeholder(tf.int32,
shape=[None],
name="encoder{0}".format(i)))
for i in xrange(buckets[-1][1] + 1):
self.decoder_inputs.append(tf.placeholder(tf.int32,
shape=[None],
name="decoder{0}".format(i)))
self.target_weights.append(tf.placeholder(dtype,
shape=[None],
name="weight{0}".format(i)))
# Our targets are decoder inputs shifted by one. (Наши цели — вводы
      декодера со сдвигом 1)
targets = [self.decoder_inputs[i + 1]
for i in xrange(len(self.decoder_inputs) — 1)]
```

Вычисляем выходные данные и ошибку в функции `seq2seq.model_with_buckets`. Эта функция создает модель `seq2seq`, совместимую с группами, и вычисляет ошибку либо усреднением по всей примерной последовательности, либо как взвешенную ошибку перекрестной энтропии для последовательности логитов:

```
# Training outputs and losses. (Выводы и потери при обучении)
if forward_only:
self.outputs, self.losses = seq2seq.model_with_buckets(
self.encoder_inputs, self.decoder_inputs, targets,
self.target_weights, buckets, lambda x, y:
seq2seq_f(x, y, True),
softmax_loss_function=softmax_loss_function)
# If we use output projection, we need to project outputs
# for decoding. (Если используется проекция вывода, для декодирования
      необходимо проецировать выводы)
if output_projection is not None:
for b in xrange(len(buckets)):
self.outputs[b] = [
tf.matmul(output, output_projection[0]) +
output_projection[1]
for output in self.outputs[b]
]
```

```
else:
self.outputs, self.losses = seq2seq.model_with_buckets(
self.encoder_inputs, self.decoder_inputs, targets,
self.target_weights, buckets,
lambda x, y: seq2seq_f(x, y, False),
softmax_loss_function=softmax_loss_function)
```

Наконец, нужно обновить параметры модели (ведь это обучающиеся переменные) при помощи градиентного спуска. Мы берем обычный стохастический спуск с обрезанием градиента, но с тем же успехом можно использовать любой оптимизатор: результаты улучшатся, и обучение пойдет гораздо быстрее. Затем сохраняем все переменные.

```
# Gradients and SGD update operation for training the model. (Градиенты
      и стохастический градиентный спуск позволяют обновить операцию
      для обучения модели)
params = tf.trainable_variables()
if not forward_only:
self.gradient_norms = []
self.updates = []
opt = tf.train.GradientDescentOptimizer(
self.learning_rate)
for b in xrange(len(buckets)):
gradients = tf.gradients(self.losses[b], params)
clipped_gradients, norm = tf.clip_by_global_norm(
gradients,
max_gradient_norm)
self.gradient_norms.append(norm)
self.updates.append(opt.apply_gradients(
zip(clipped_gradients, params), global_step=
self.global_step))
self.saver = tf.train.Saver(tf.all_variables())
```

Описав высокоуровневые детали графа вычислений, переходим к последнему и самому нижнему уровню модели — внутренним элементам `seq2seq.embedding_attention_seq2seq()`. При инициализации модели некоторые флаги и аргументы передаются как параметры функции. Подробнее изучим аргумент `feed_previous`. При его значении `true` декодер будет использовать выведенный логит на временном шаге `T` как вход на временном шаге `T+1`. Он станет последовательно декодировать очередной токен на основе всех предыдущих. Такой тип декодирования, при котором каждое выходное значение зависит от всех предыдущих, можно назвать *авторегрессионным*:

```
def embedding_attention_seq2seq(encoder_inputs,
decoder_inputs,
cell,
num_encoder_symbols,
num_decoder_symbols,
embedding_size,
output_projection=None,
feed_previous=False,
dtype=None,
scope=None,
initial_state_attention=False):
```

Сначала создаем обертку для декодера.

```
with variable_scope.variable_scope(
scope or "embedding_attention_seq2seq", dtype=dtype)
as scope:
dtype = scope.dtype
encoder_cell = rnn_cell.EmbeddingWrapper(
cell,
embedding_classes=num_encoder_symbols,
embedding_size=embedding_size)
encoder_outputs, encoder_state = rnn.rnn(
encoder_cell, encoder_inputs, dtype=dtype)
```

В следующем фрагменте кода вычисляем конкатенацию выходных данных кодера, на которые нужно обратить внимание. Это важно, поскольку позволяет декодеру проходить по этим состояниям как по распределению:

```
# First calculate a concatenation of encoder outputs
# to put attention on. (Сначала вычисляем конкатенацию выводов кодера,
      на которые нужно обратить внимание)
           top_states = [
array_ops.reshape(e, [-1, 1, cell.output_size]) for e
in encoder_outputs
]
attention_states = array_ops.concat(1, top_states)
```

Создаем декодер. Если флаг `output_projection` не выбран, нейрон оборачивается, чтобы использовать проекцию выходного значения:

```
output_size = None
if output_projection is None:
```

```
cell = rnn_cell.OutputProjectionWrapper(cell,
num_decoder_symbols)
output_size = num_decoder_symbols
```

Отсюда вычисляем выходные значения и состояния с помощью `embedding_attention_decoder`:

```
if isinstance(feed_previous, bool):
       return embedding_attention_decoder(
       decoder_inputs,
       encoder_state,
       attention_states,
       cell,
       num_decoder_symbols,
       embedding_size,
       output_size=output_size,
       output_projection=output_projection,
       feed_previous=feed_previous,
       initial_state_attention=initial_state_attention)
```

Стоит отметить, что `embedding_attention_decoder` — небольшое улучшение по сравнению с `attention_decoder`, описанным в предыдущем разделе. Здесь выходные значения проецируются на обученное пространство векторного представления, что обычно повышает производительность. Функция `loop`, которая просто описывает динамику рекуррентного нейрона с векторным представлением, вызывается на этом шаге:

```
def embedding_attention_decoder(decoder_inputs,
       initial_state,
       attention_states,
       cell,
       num_symbols,
       embedding_size,
       output_size=None,
       output_projection=None,
       feed_previous=False,
       update_embedding_for_previous=
       True,
       dtype=None,
       scope=None,
       initial_state_attention=False):
       if output_size is None:
```

```
        output_size = cell.output_size
        if output_projection is not None:
        proj_biases = ops.convert_to_tensor(output_projection[1],
              dtype=dtype)
        proj_biases.get_shape().assert_is_compatible_with(
              [num_symbols])
        with variable_scope.variable_scope(
        scope or "embedding_attention_decoder", dtype=dtype)
        as scope:
        embedding = variable_scope.get_variable("embedding",
        [num_symbols,
              embedding_size])
        loop_function = _extract_argmax_and_embed(
        embedding, output_projection,
        update_embedding_for_previous) if feed_previous
        else None
        emb_inp = [
        embedding_ops.embedding_lookup(embedding, i) for i in
        decoder_inputs
  ]
  return attention_decoder(
        emb_inp,
        initial_state,
        attention_states,
        cell,
        output_size=output_size,
        loop_function=loop_function,
        initial_state_attention=initial_state_attention)
```

Последний шаг — рассмотреть собственно `attention_decoder`. Как видно из названия, основное свойство этого декодера — вычислять набор весов внимания по скрытым состояниям, которые выдаются при кодировании. После профилактических проверок изменяем размер скрытых признаков на нужный:

```
  def attention_decoder(decoder_inputs,
  initial_state,
  attention_states,
  cell,
  output_size=None,
```

```
loop_function=None,
dtype=None,
scope=None,
initial_state_attention=False):
if not decoder_inputs:
raise ValueError("Must provide at least 1 input to attention
decoder.") (Необходимо предоставить по меньшей мере один ввод декодеру
      внимания)
if attention_states.get_shape()[2].value is None:
raise ValueError("Shape[2] of attention_states must be known:
      (Форма [2] attention_states должна быть известна)
%s" %
attention_states.get_shape())
if output_size is None:
output_size = cell.output_size
with variable_scope.variable_scope(
scope or "attention_decoder", dtype=dtype) as scope:
dtype = scope.dtype
batch_size = array_ops.shape(decoder_inputs[0])[0] # Needed
# for
#reshaping.
attn_length = attention_states.get_shape()[1].value
if attn_length is None:
attn_length = array_ops.shape(attention_states)[1]
attn_size = attention_states.get_shape()[2].value
# To calculate W1 * h_t we use a 1-by-1 convolution,
# need to reshape before. (Для вычисления W1 * h_t используем
      свертку 1 к 1, которую нужно предварительно переформатировать)
hidden = array_ops.reshape(attention_states,
[-1, attn_length, 1, attn_size])
hidden_features = []
v = []
attention_vec_size = attn_size # Size of query vectors
for attention. (Размер векторов запросов для внимания)
k = variable_scope.get_variable("AttnW_0",
[1, 1, attn_size,
attention_vec_size])
hidden_features.append(nn_ops.conv2d(hidden, k,
[1, 1, 1, 1], "SAME"))
```

```
v.append(
variable_scope.get_variable("AttnV_0",
[attention_vec_size]))
state = initial_state
```

Сейчас мы определим метод `attention()`, который получает вектор запросов и возвращает вектор весов внимания по скрытым состояниям. Этот метод реализует тот же подход к вниманию, что и в предыдущем разделе:

```
def attention(query):
"""Put attention masks on hidden using hidden_features
and query."""  (Накладываем маски внимания на скрытый слой с помощью
      hidden_features и запросов)
ds = [] # Results of attention reads will be
# stored here. (Результаты чтения внимания будут храниться здесь)
if nest.is_sequence(query): # (если запрос n-мерен, делаем его плоским)
# flatten it.
query_list = nest.flatten(query)
for q in query_list: # Check that ndims == 2 if
# specified. (Проверяем, что ndims == 2, если указано)
ndims = q.get_shape().ndims
if ndims:
assert ndims == 2
query = array_ops.concat(1, query_list)
# query = array_ops.concat(query_list, 1)
with variable_scope.variable_scope("Attention_0"):
y = linear(query, attention_vec_size, True)
y = array_ops.reshape(y, [-1, 1, 1,
attention_vec_size])
# Attention mask is a softmax of v^T * tanh(…). (Маска внимания —
      функция мягкого максимума v^T * tanh(…))
s = math_ops.reduce_sum(v[0] * math_ops.tanh(
hidden_features[0] + y),
[2, 3])
a = nn_ops.softmax(s)
# Now calculate the attention-weighted vector d. (Теперь вычисляем
      вектор весов внимания d)
d = math_ops.reduce_sum(
array_ops.reshape(a, [-1, attn_length, 1, 1]) *
hidden, [1, 2])
ds.append(array_ops.reshape(d, [-1, attn_size]))
return ds
```

При помощи этой функции вычисляем внимание по каждому из состояний вывода, начиная с начального:

```
outputs = []
prev = None
batch_attn_size = array_ops.stack([batch_size, attn_size])
attns = [array_ops.zeros(batch_attn_size, dtype=dtype)]
for a in attns: # Ensure the second shape of attention
# vectors is set. (Убеждаемся, что вторая форма векторов внимания задана)
a.set_shape([None, attn_size])
if initial_state_attention:
attns = attention(initial_state)
```

Выполняем тот же цикл для остальных входных данных. Проводим профилактическую проверку, чтобы убедиться, что входные данные на текущем шаге имеют нужный размер. Затем запускаем ячейки РНС и запрос внимания. Они объединяются и передаются на выход в соответствии с той же динамикой:

```
for i, inp in enumerate(decoder_inputs):
if i > 0:
variable_scope.get_variable_scope().reuse_variables()
# If loop_function is set, we use it instead of
# decoder_inputs. (Если задана loop_function, используем ее вместо
      вводов декодера)
if loop_function is not None and prev is not None:
with variable_scope.variable_scope("loop_function",
reuse=True):
inp = loop_function(prev, i)
# Merge input and previous attentions into one vector of
# the right size. (Сливаем ввод и предыдущие внимания в один вектор
      нужного размера)
input_size = inp.get_shape().with_rank(2)[1]
if input_size.value is None:
raise ValueError("Could not infer input size from input:
%s" % inp.name) (Нельзя вывести размер ввода из ввода)
x = linear([inp] + attns, input_size, True)
# Run the RNN. (Запускаем РНС)
cell_output, state = cell(x, state)
# Run the attention mechanism. (Запускаем механизм внимания)
if i == 0 and initial_state_attention:
```

```
with variable_scope.variable_scope(
variable_scope.get_variable_scope(), reuse=True):
attns = attention(state)
else:
attns = attention(state)
with variable_scope.variable_scope(
"AttnOutputProjection"):
output = linear([cell_output] + attns, output_size,
True)
if loop_function is not None:
prev = output
outputs.append(output)
return outputs, state
```

Мы успешно завершили подробную реализацию весьма изощренной нейронной системы машинного перевода. Продуктивные системы содержат дополнительные хитрости, которые не поддаются простым обобщениям, к тому же они обучаются на больших вычислительных серверах, чтобы обеспечивать передовые технологические возможности. Так, эта модель обучалась в течение четырех дней на восьми GPU NVIDIA Telsa M40. Показываем графики перплексии на рис. 7.31 и 7.32, а также изменение темпа обучения со временем.

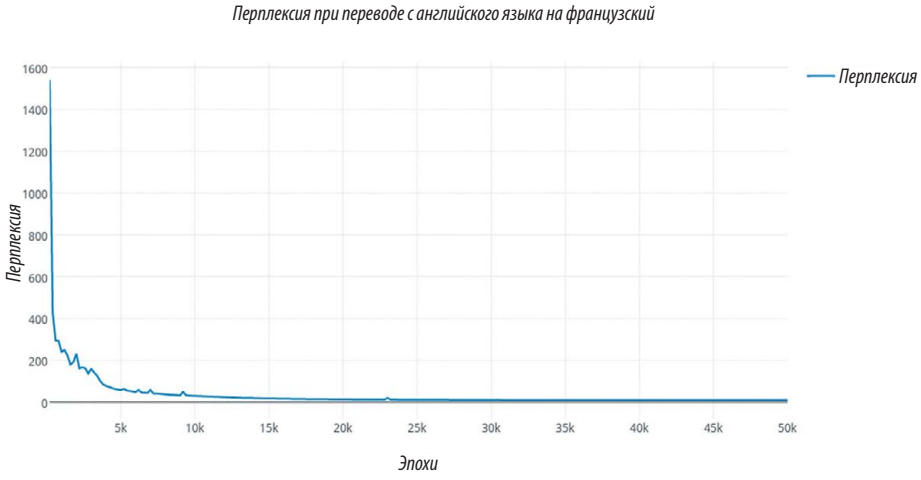


Рис. 7.31. *График перплексии по обучающим данным во времени. После 50 тысяч эпох перплексия снижается примерно с 6 до 4 — вполне приличный результат для нейронной системы машинного перевода*

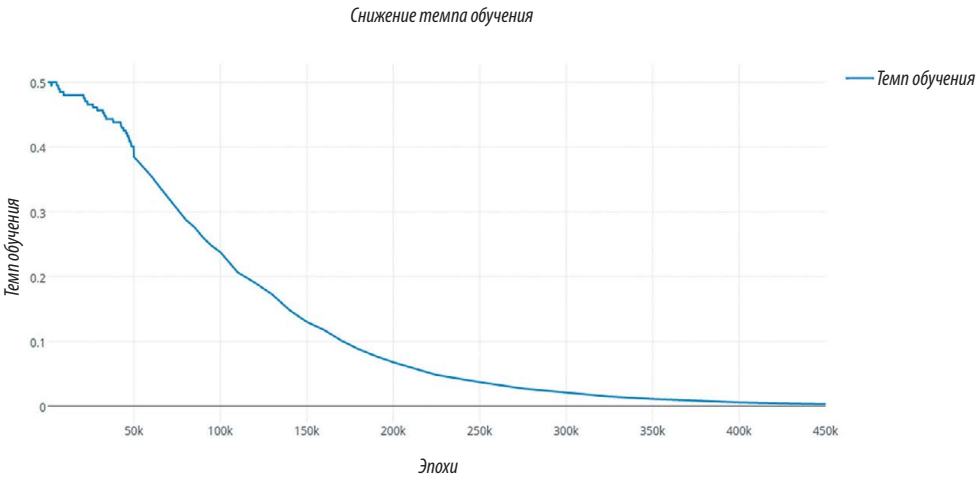


Рис. 7.32. *График темпа обучения во времени; в отличие от перплексии, темп почти гладко снижается до 0. Это означает, что к моменту окончания обучения модель достигнет стабильного состояния*

Чтобы ярче показать модель с вниманием, можно визуализировать внимание, которое вычисляет декодер `LSTM` при переводе предложения с английского языка на французский. Так, мы знаем, что, когда кодер `LSTM` обновляет состояние ячейки, чтобы сжать предложение до непрерывного векторного представления, он также вычисляет скрытые состояния на каждом шаге. Мы знаем, что декодер `LSTM` вычисляет выпуклую сумму по этим скрытым состояниям, и ее можно считать механизмом внимания: когда определенному скрытому состоянию соответствует больший вес, можно считать, что модель обращает больше внимания на токен, введенный на этом шаге.

Именно такую визуализацию мы и показываем на рис. 7.33. Английское предложение, которое необходимо перевести, приведено в верхней строке, а перевод на французский — в первом столбце.

Чем светлее квадрат, тем больше внимания декодер уделил столбцу при декодировании этого элемента ряда. Таким образом, `(i, j)`-й элемент карты внимания показывает меру внимания, которая была уделена `j`-й метке в английском предложении при переводе `i`-й метки во французском.

Сразу можно увидеть, что механизм, судя по всему, работает неплохо. Внимание в основном уделяется именно нужным областям, хотя в предсказаниях модели и присутствует легкий шум. Возможно, добавление дополнительных слоев в сеть поможет добиться более четкого внимания.

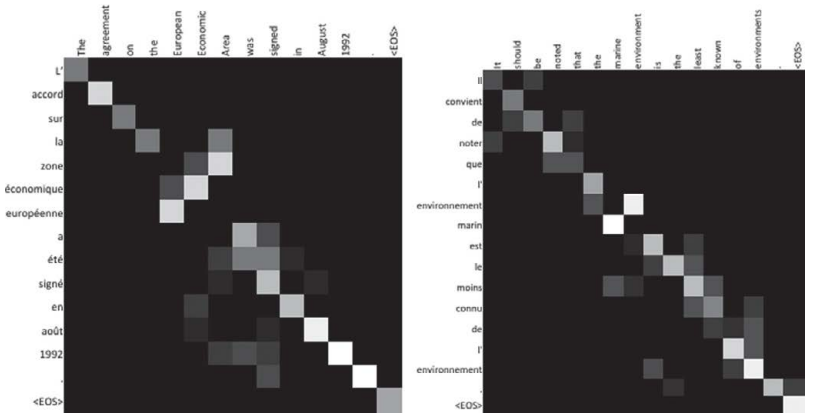


Рис. 7.33. *Мы можем непосредственно визуализировать веса, когда декодер наблюдает скрытые состояния кодера. Чем светлее квадрат, тем больше внимания уделяется элементу*

Отметим особо, что словосочетание the European Economic Area («европейская экономическая зона») переводится на французский в обратном порядке — zone economique europeenne, и этот переворот отражается в весах внимания! Такие схемы могут быть еще интереснее, если переводить с английского на какой-то другой язык, который не так гладко разбивается слева направо. Разобрав и воплотив одну из важнейших архитектур, мы можем перейти к изучению поразительных новых достижений рекуррентных нейронных сетей и углубиться в более сложные аспекты обучения.

Резюме

В этой главе мы занимались анализом последовательностей. Мы разобрали, как заставить сети с прямым распространением сигнала обрабатывать последовательности, постарались понять работу рекуррентных нейронных сетей и остановились на многочисленных областях применения механизмов внимания: от переводов с языка на язык до расшифровки аудиофайлов.

ГЛАВА 8

Нейронные сети с дополнительной памятью

Мостафа Самир[86]

Мы уже убедились в том, насколько эффективной может быть РНС при решении таких сложных проблем, как машинный перевод. Но мы пока не раскрыли ее потенциал полностью. В главе 7 мы говорили о теоретических доказательствах того, что архитектура РНС обеспечит универсальное представление функций. Еще более точный результат можно получить при помощи *полноты РНС по Тьюрингу*. При грамотной архитектуре и адекватной установке параметров им будут под силу любые вычислительные задачи, решаемые компьютерными алгоритмами или машиной Тьюринга*.

Нейронные машины Тьюринга

Однако добиться такой универсальности на практике крайне сложно. Причина в том, что мы имеем дело с огромным полем поиска вариантов архитектур и значений параметров РНС — настолько большим, что при помощи градиентного спуска очень трудно найти решение произвольной задачи. Ниже мы рассмотрим ряд подходов с переднего края исследований, позволяющих начать реализовывать этот потенциал.

Рассмотрим, например, очень простую проблему понимания чтения.

```
Мэри вышла в коридор. Она взяла там стакан с молоком. Затем она
вернулась в офис, где увидела яблоко, и взяла его.
Сколько предметов в руках у Мэри?
```

* Машина Тьюринга — абстрактная вычислительная машина, предложенная Аланом Тьюрингом в 1936 году. Включает неограниченную в обе стороны ленту, разделенную на ячейки, и управляющее устройство с головками чтения и записи данных на нее. Устройство может находиться в одном из множества состояний, заданных заранее. *Прим. науч. ред.*

Ответ тривиален: два! Но что в нашем мозге породило этот вариант? Если бы мы думали о том, как ответить на этот вопрос на понимание с помощью простой компьютерной программы, алгоритм, вероятно, выглядел бы так.

```
1. назначить ячейку памяти для подсчета.
2. инициализировать подсчет значением 0.
3. для каждого слова в тексте.
      3.1. если слово — взяла.
            3.1.1. увеличить число.
4. выдать значение.
```

Оказывается, мозг подходит к решению задачи почти так же, как и эта простая программа. Приступая к чтению, мы выделяем место в памяти (как и она) и помещаем туда полученные данные. Сначала мы фиксируем местоположение Мэри — судя по первому предложению, это коридор. Из второго получаем информацию о предметах, которые несет Мэри. Сейчас это стакан молока. Когда мы видим третье предложение, мозг модифицирует значение первой ячейки памяти: уже не коридор, а офис. К концу четвертого предложения вторая ячейка тоже модифицируется, вместив не только молоко, но и яблоко. Когда мы доходим до вопроса, мозг быстро обращается ко второй ячейке и получает оттуда информацию о том, что предметов два. В нейронауках и когнитивной психологии такая система кратковременного хранения информации и управления ею называется кратковременной памятью. Именно она лежит в основе исследований, которые мы будем рассматривать в этой главе.

В 2014 году Алекс Грейвз и коллеги из Google DeepMind впервые осветили этот вопрос в статье «Нейронные машины Тьюринга»[87], где ввели новую нейронную архитектуру под тем же названием — *нейронная машина Тьюринга* (Neural Turig Machine, NTM). Она состоит из контроллерной нейронной сети (обычно PHC) с внешним запоминающим устройством, созданным по принципу рабочей памяти мозга.

Рисунок 8.1 показывает, что то же сходство сохраняется и для архитектуры нейронной машины Тьюринга с внешней памятью вместо RAM, головками считывания/записи вместо шины считывания/записи и контроллера с сетью вместо CPU, за исключением того, что контроллер обучается программе, а в CPU она поступает в готовом виде.

Если рассмотреть NTM в свете беседы о полноте PHC по Тьюрингу, окажется, что дополнение PHC внешней памятью для кратковременного хранения позволяет вырезать из поля поиска существенный сегмент. Ведь теперь не нужно работать с PHC, которые способны и манипулировать информацией, и хранить ее: достаточно найти те, которые могут обрабатывать информацию с внешнего носителя. Такое сужение поля поиска помогает нам частично раскрыть потенциал PHC, до которого ранее было тяжело добраться. Это

очевидно по разнообразию задач, которым может научиться NTM: от копирования последовательностей входных данных до эмулирования н-граммных моделей и сортировки данных по приоритетам. К концу главы мы увидим, как расширенная NTM может научиться выполнять задачи на понимание прочитанного, вроде рассмотренной выше, методом градиентного поиска!

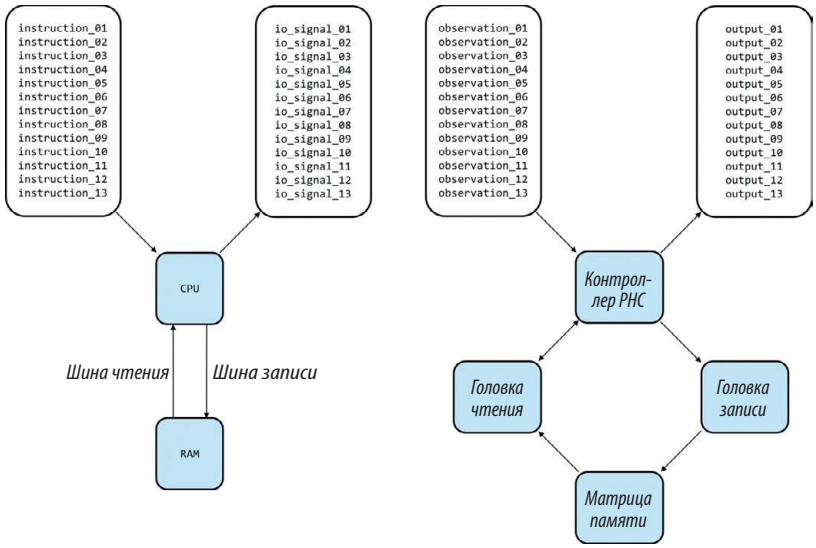


Рис. 8.1. *Сравнение архитектуры современного компьютера, в который поступает готовая программа (слева), с нейронной машиной Тьюринга, которая обучается (справа). Здесь по одной головка чтения и записи, но в NTM их может быть и по нескольку*

Доступ к памяти на основе внимания

Чтобы обучать NTM по методу градиентного поиска, нужно убедиться, что вся архитектура дифференцируема: можно вычислить градиент исходящих потерь по отношению к параметрам модели, обрабатывающим входные данные. Это свойство называется *сквозной дифференцируемостью* — от входа к выходу. Если мы попробуем получить доступ к памяти NTM так, как цифровые компьютеры обращаются к `RAM`, через дискретные значения адресов, мы получим разрывность в градиентах вывода и не сможем больше обучать модель градиентным методом. Нам нужен способ постоянного доступа к памяти и возможность «сосредоточиться» на конкретной ее ячейке. И такой постоянной концентрации можно достичь благодаря методам внимания!

Вместо дискретного адреса в памяти мы даем каждой головке возможность создать нормализованный функцией мягкого максимума вектор внимания

того же размера, что и число ячеек памяти. Так мы получим доступ ко всем ячейкам памяти одновременно в размытом виде. Каждое значение вектора будет показывать, насколько мы собираемся концентрироваться на ячейке или насколько вероятно то, что нам понадобится доступ к ней. Например, чтобы прочесть вектор на временном шаге t из матрицы памяти $N \times W$, обозначенной M_t (где N — число ячеек, а W — размер ячейки), мы создаем вектор внимания, или вектор весов w_t размера N, и вектор считывания можно будет вычислить через произведение $\mathbf{r}_t = M_t^\top w_t$, где $\top$ обозначает операцию транспонирования матрицы. Рисунок 8.2 показывает, как эти веса относятся к конкретной ячейке. Теперь мы можем извлечь вектор чтения, содержащий примерно ту же информацию, что и ячейка памяти.

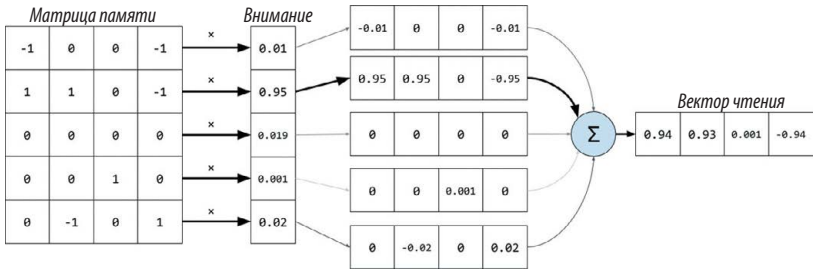


Рис. 8.2. *Именно так размытое, основанное на внимании чтение может выдать вектор, содержащий примерно ту же информацию, что и соответствующая ячейка*

Тот же метод используется и для головки записи: создается вектор весов w_t, который служит для стирания определенной информации из памяти, указанной контроллером вектора стирания e_t с W значений от 0 до 1; от этих значений и зависит, что стирать и что хранить. Затем взвешивание проводится для записи в матрицу стертой памяти новой информации, тоже определяемой контроллером в векторе записи v_t, которая содержит W значений:

$$M_t = M_{t-1} \circ \left(E - w_t e_t^\top\right) + w_t v_t^\top,$$

где E — матрица единиц, а $\circ$ — поэлементное умножение. Как и в случае с чтением, взвешивание w_t подсказывает, куда направить операции стирания (первый член уравнения) и записи (второй член).

Механизмы адресации памяти в NTM

Теперь, когда мы понимаем, как NTM способна непрерывно обращаться к памяти при помощи взвешивания внимания, осталось понять, как эти веса порождаются и какие формы обращения к памяти представляют. Для этого стоит проанализировать, что NTM должны делать со своей памятью.

Судя по модели, работу которой они имитируют (машина Тьюринга), они должны получать доступ к ячейке по ее значению и уметь двигаться вперед или назад из нее.

Первый способ поведения может быть реализован механизмом доступа, который мы назовем *адресацией по содержанию*. При такой форме обращения контроллер выдает значение, которое ищет и которое мы будем называть ключом k_t, затем определяет степень его сходства с информацией, сохраненной в каждой ячейке, и сосредоточивает внимание на самой похожей из них. Такое взвешивание можно вычислить следующим образом:

$$\mathrm{C}(M,k,\beta)=\frac{\exp\left(\beta \mathrm{D}\left(M,k\right)\right)}{\sum_{i=0}^{N}\exp\left(\beta \mathrm{D}\left(M\left[i\right],k\right)\right)},$$

где D — некая мера сходства, например косинусная близость. Это уравнение — нормализованное распределение функции мягкого максимума по результатам сходства. Но здесь есть дополнительный параметр β, призванный при необходимости заставить затухать веса внимания. Мы называем его мощностью ключа. Его основной смысл в том, что для некоторых задач ключ, выданный контроллером, не очень близок к каким-то данным в памяти, что приведет к единообразным с виду весам внимания. На рис. 8.3 показано, как мощность ключа позволяет контроллеру учиться выходить из этой ситуации, чтобы больше сосредоточиваться на одной наиболее вероятной ячейке; затем контроллер учится тому, какое значение мощности выдавать для каждого порождаемого ключа.

Чтобы переходить по памяти вперед-назад, сначала надо понять, где мы сейчас. Эту информацию мы получаем взвешиванием доступа на предыдущем шаге w_{t-1}. Поэтому, чтобы сохранить информацию о текущем местоположении с новым взвешиванием на основании содержания w_t^c, которое мы только что провели, проводим интерполяцию между двумя взвешиваниями, используя скаляр g_t, лежащий между 0 и 1:

$$w_t^g = g_t w_t^c + \left(1-g_t\right) w_{t-1}.$$

Назовем g_t *вентилем интерполяции*. Он тоже порождается контроллером и контролирует информацию, которую мы хотим использовать на текущем временном шаге. Если значение вентиля близко к 1, мы делаем акцент на обращении по содержанию. Если же оно близко к 0, мы передаем информацию о текущем положении и игнорируем обращение по содержанию. Контроллер учится использовать этот вентиль и, например, может обращать его в 0 при необходимости итерации через последовательные ячейки, когда важнее всего информация о текущем местоположении. Тип информации, которую он пропускает, определяется *вентильным взвешиванием* w_t^g.

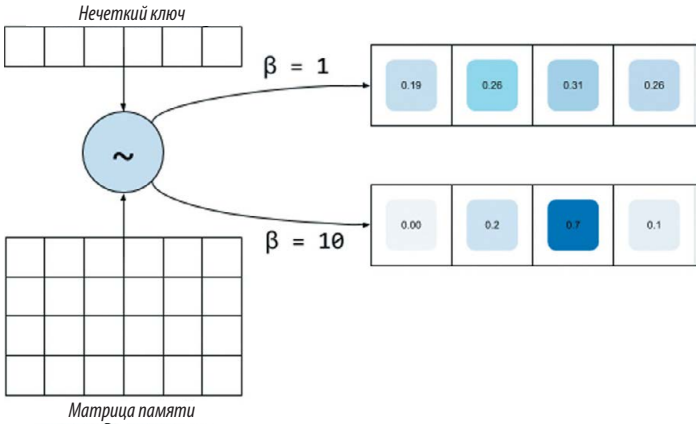


Рис. 8.3. *Нечеткий ключ с похожими результатами, образующими почти единообразный и бесполезный вектор внимания. Увеличение его мощности позволяет сконцентрироваться на наиболее вероятной ячейке*

Чтобы начать двигаться по памяти, нужно найти способ взять текущее вентильное взвешивание и сместить фокус с одной ячейки на другую. Это можно сделать, выполнив свертку вентильного взвешивания *сдвиговым взвешиванием* s_t, которое тоже выдается контроллером. Это нормализованный функцией мягкого максимума вектор внимания размера $(n + 1)$, где n — четное целое число, указывающее на количество возможных сдвигов вокруг данной ячейки при вентильном взвешивании. Например, при размере 3 можно сказать, что возможны два сдвига: один вперед, один назад. На рис. 8.4 показано, как сдвиговое взвешивание способно перемещать ячейку концентрации при вентильном взвешивании. Смещение происходит при свертке вентильного взвешивания сдвиговым — примерно так же, как мы сворачивали изображения при помощи карт признаков в главе 5. Единственное исключение — случай, когда сдвиговое взвешивание выходит за рамки вентильного. Вместо дополнения, которое мы применяли ранее, воспользуемся поворотным сверточным оператором: вышедшие за пределы веса применяются к значениям на другом конце вентильного взвешивания, как показано в средней панели рис. 8.4. Эту операцию поэлементно можно выразить так:

$$\tilde{w}_t[i] = \sum_{j=0}^{|s_t|} w_t^g \left[\left(i + \frac{|s_t| - 1}{2} - j \right) \bmod N \right] s_t[j].$$

При введении операции сдвига веса наших головок могут свободно перемещаться по памяти вперед и назад. Но возникает проблема, если в какой-то момент сдвиговое взвешивание окажется недостаточно резким.

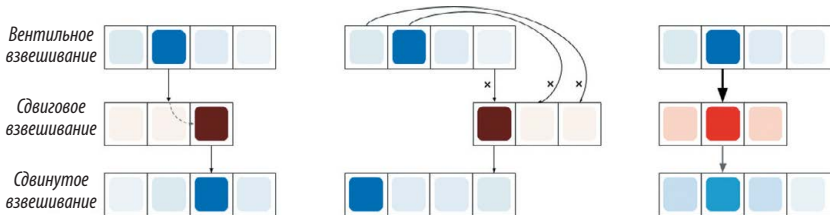


Рис. 8.4. *Слева: сдвиговое взвешивание, направленное направо, смещает вентильное взвешивание на одну ячейку вправо. В центре: поворотная свертка на сдвиговом взвешивании, нацеленном влево, сдвигает вентильное взвешивание влево. Справа: нерезкое сдвиговое взвешивание к центру оставляет вентильное на месте, но рассеивает его*

По природе операции свертки нерезкое сдвиговое взвешивание (правая панель рис. 8.4) рассеивает исходные вентильные по окрестностям, что снижает его концентрацию. Для борьбы с этим эффектом размывания мы выполняем еще одну операцию над сдвиговыми взвешиваниями: заострение. Контроллер выдает последний скаляр $\gamma_t \geq 1$, который заостряет сдвиговое взвешивание:

$$w_t = \frac{\tilde{w}_t^{\gamma_t}}{\sum_{i=0}^{N} \tilde{w}_t \left[i\right]^{\gamma_t}}.$$

Процесс, начинающийся с интерполяции и заканчивающийся вектором весов после заострения, — второй механизм адресации в NTM, *механизм на основе ячейки*. Сочетая оба эти механизма, NTM может использовать память, чтобы учиться выполнять разные задачи. Одна из них, которая к тому же позволит нам лучше рассмотреть NTM в действии, — задача копирования, представленная на рис. 8.5. В ней мы задаем последовательность случайных двоичных векторов, которые оканчиваются специальным символом, и требуем перенести входную последовательность в выходную.

Визуализация показывает, как во время ввода NTM начинает записывать входные данные шаг за шагом в последовательные ячейки памяти. Во время вывода NTM возвращается к первому вектору и проходит по следующим ячейкам, читая и выводя записанную ранее входную последовательность. В первой работе по NTM приводится несколько других визуализаций, обученных на других задачах. Эти визуализации показывают, что архитектура способна пользоваться механизмами адресации для адаптации и обучения решению разных задач.

Ограничимся текущим пониманием NTM и пропустим этап реализации. Остаток главы посвятим анализу недостатков NTM и тому, как их помогла устранить инновационная архитектура: дифференцируемый нейронный

компьютер (differentiable neural computer, DNC). В конце опробуем реализацию этой архитектуры для простых задач понимания чтения, вроде уже рассмотренной выше.

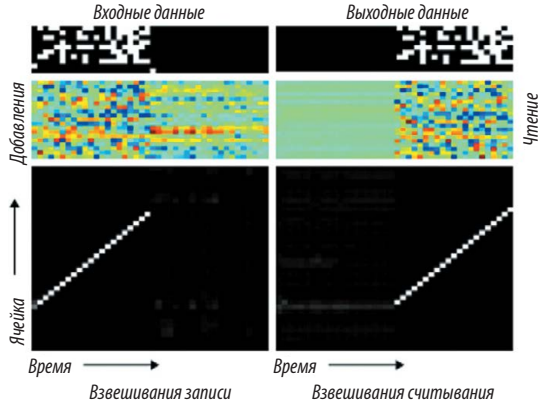


Рис. 8.5. *Визуализация NTM, обученной на задаче копирования. Слева: сверху вниз показаны входные данные модели, векторы записи и взвешивания по всем ячейкам памяти во времени. Справа: сверху вниз показаны вывод модели, векторы считывания и взвешивания считывания по всем ячейкам памяти с течением времени*[88]

Дифференцируемый нейронный компьютер

При всех своих достоинствах NTM обладают рядом ограничений, связанных с механизмами их памяти. Первое таково: NTM не способны гарантировать, что записанные данные не будут путаться или перекрываться. Дело в самой природе «дифференцируемой» операции записи: мы фиксируем новые данные по всей памяти, и лишь отчасти этот процесс регулируется вниманием. Обычно механизмы внимания учатся концентрировать веса строго в одной ячейке и NTM стремится к самому свободному от вмешательства поведению, но это не гарантировано. И даже в этом случае, когда в ячейку памяти что-то записано, снова использовать ее уже не получится, даже если данные утратят важность.

Невозможность освобождать и повторно использовать ячейки — второе ограничение архитектуры NTM. В результате новые данные записываются в новые ячейки, которые, скорее всего, как и в вышеприведенной задаче копирования, окажутся смежными. Это единственный способ хранения в NTM временной информации о данных: в порядке поступления. Если головка записи перепрыгнет в другое место памяти при записи

последовательных данных, головка чтения не сможет восстановить временную связь между информацией, записанной до и после скачка. Таково третье ограничение NTM.

В октябре 2016 года Алекс Грейвз и коллеги по DeepMind опубликовали статью под названием «Гибридные вычисления с помощью нейронной сети с динамической внешней памятью»[89], где представили новую нейронную архитектуру с дополненной памятью — *дифференцируемый нейронный компьютер* (DNC). Это улучшение NTM, призванное устранить ограничения, которые мы упоминали выше. Как и NTM, DNC включает контроллер, взаимодействующий с внешней памятью. Память состоит из N слов размера W, составляющих матрицу $N \times W$, которую мы назовем M. Контроллер принимает вектор входных данных размером X и R векторов размером W, прочитанных из памяти на предыдущем шаге, где R — число головок считывания. Затем он пропускает их через нейтральную сеть и выдает два вида данных.

- *Вектор интерфейса*, который содержит всю необходимую информацию для запроса в память (то есть записи и чтения).
- *Предварительную версию выходного вектора* размером Y.

Внешняя память принимает вектор интерфейса, проводит необходимую запись единственной головкой и читает R новых векторов из памяти. Она выдает только что прочитанные векторы в контроллер, где те суммируются с предварительным выходным вектором. Получается итоговый выходной вектор размером Y.

На рис. 8.6 наглядно показана работа DNC, которую мы только что описали. В отличие от NTM, DNC сохраняют и остальные структуры данных вместе с памятью, что позволяет отслеживать ее состояние. Как мы вскоре увидим, благодаря этим структурам и новым разумным механизмам внимания DNC способны успешно преодолеть ограничения NTM.

Чтобы архитектура была дифференцируемой, DNC получают доступ к памяти через вектора весов размера N, элементы которых определяют, насколько головки концентрируются в каждой ячейке. Имеется R взвешиваний для головок чтения $w_t^{r,1}, \ldots, w_t^{r,R}$, где t обозначает временной шаг. И есть одно взвешивание записи w_t^w для единственной головки. Получив эти взвешивания, можно внести обновления в матрицу памяти:

$$M_t = M_{t-1} \circ \left(E - w_t^w e_t^\top\right) + w_t^w v_t^\top,$$

где e_t, v_t — векторы *стирания* и *записи*, которые нам уже знакомы по NTM и поступают от контроллера по вектору интерфейса как указания о том, что стереть из памяти и что в нее записать.

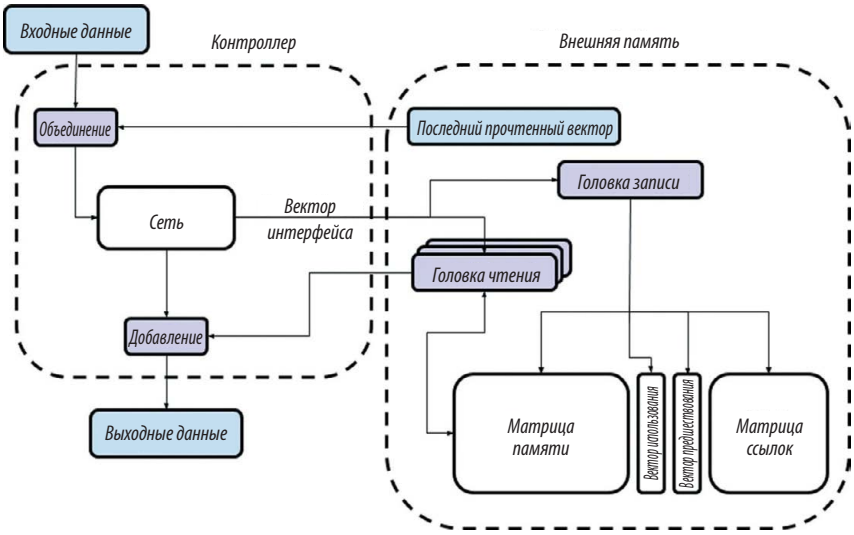


Рис. 8.6. *Обзор архитектуры и схемы работы DNC. Внешняя память DNC отличается от NTM несколькими дополнительными структурами данных, а также механизмами внимания, используемыми для доступа*

Получив обновленную матрицу M_t, мы можем прочесть новые векторы считывания $r_t^1, r_t^2, \ldots, r_t^R$ при помощи следующего уравнения для каждого взвешивания:

$$r_t^i = M_t^\top w_t^{r,i}.$$

Пока создается впечатление, что в процедуре записи и считывания DNC и NTM все шаги совпадают. Разница станет заметна, когда мы перейдем к разбору механизмов внимания, используемых в `DNC` для получения этих взвешиваний. Обе машины используют механизм адресации по содержанию `C(M, k, β)`, о котором шла речь выше, но механизмы DNC более совершенны; более эффективно и обращение к памяти.

Запись без помех в DNC

Первое ограничение NTM, о котором мы говорили, — невозможность обеспечить запись без помех. Интуитивный способ решения этой проблемы — разработать архитектуру, которая будет жестко концентрироваться на единственной свободной ячейке памяти, не дожидаясь, пока NTM научится это делать. Чтобы следить, какие ячейки свободны, а какие заняты, нам нужна новая структура данных, способная удерживать информацию такого рода. Назовем ее *`вектором использования`*. Вектор использования u_t — вектор

размера *N*, где каждый элемент имеет значение от 0 до 1. Это отражает степень использования соответствующей ячейки памяти: `0` — полностью свободная, `1` — полностью занятая. Вектор использования изначально содержит нули $u_0 = \mathbf{0}$ и обновляется на каждом шаге при появлении информации. Благодаря ей становится ясно, что ячейка, на которую нужно обратить особое внимание, имеет наименьшее значение использования. Чтобы получить такое взвешивание, нужно отсортировать вектор использования и получить список индексов ячеек в восходящем порядке. Такой список называется *свободным*, обозначим его как φ_t. Благодаря ему можно создать промежуточное взвешивание — *взвешивание выделения* a_t, которое определяет, какая ячейка памяти отводится под новые данные. Вычисляется a_t так:

$$a_t\left[\phi_t\left[j\right]\right] = \left(1 - u_t\left[\phi_t\left[j\right]\right]\right)\prod_{i=1}^{j-1} u_t\left[\phi_t\left[i\right]\right],$$

где $j \in 1,\ldots,N$.

На первый взгляд, уравнение кажется непонятным. Его проще осознать, если взять числовой пример. Пусть u_t = `[1, 0.7, 0.2, 0.4]`. Расчеты вы сможете провести сами. В итоге у вас должно получиться такое взвешивание выделения: a_t = `[0, 0.024, 0.8, 0.12]`. После вычислений становится ясно, как работает формула: `1 - u_t[φ_t[j]]` делает вес ячейки пропорциональным степени ее незанятости. Отметим, что произведение $\prod_{i=1}^{j-1} u_t\left[\phi_t\left[j\right]\right]$ становится все меньше при итерациях по свободному списку, поскольку мы постоянно перемножаем числа из множества от 0 до 1. Оно дополнительно снижает вес ячейки при переходе от наименее используемой ячейки к наиболее используемой. В результате самая свободная ячейка получает наибольший вес, а самая занятая — наименьший. Так мы можем гарантировать способность сосредоточиваться на одной ячейке, не рассчитывая, что модель научится этому с нуля: отсюда большая надежность и сокращение времени обучения.

Имея взвешивание выделения a_t и взвешивание просмотра c_t^w, которые мы получаем из механизма адресации по содержанию: $c_t^w = C\left(M_{t-1}, k_t^w, \beta_t^w\right)$, где k_t^w, β_t^w — ключ и сила просмотра, полученные по вектору интерфейса, можно вывести итоговое взвешивание записи:

$$w_t^w = g_t^w\left[g_t^a a_t + \left(1 - g_t^a\right)c_t^w\right],$$

где g_t^w, g_t^a — значения от `0` до `1`, именуемые вентилем записи и вентилем выделения, которые мы тоже получаем из контроллера через вектор интерфейса. Вентили контролируют операцию записи, причем g_t^w сначала определяет, должна ли вообще делаться запись, а g_t^a показывает, будем ли мы добавлять информацию в новую ячейку при помощи взвешивания выделения или изменим текущее значение, указанное просмотровым взвешиванием.

Повторное использование памяти в DNC

А если при вычислении взвешивания выделения обнаружится, что все ячейки используются, то есть $\mathrm{u}_t = \mathbf{1}$? Тогда все взвешивания обратятся в 0 и в память не поступят новые данные. Потому-то необходимо уметь освобождать и заново использовать память.

Чтобы знать, какие ячейки можно освободить, а какие нет, строим *вектор удержания* ψ_t размера N, который указывает, какое количество данных в каждой ячейке следует сохранить. Каждый элемент принимает значение от 0 до 1, где 0 указывает, что ячейка может быть освобождена, а 1 — что она должна быть полностью сохранена. Вектор вычисляется так:

$$\psi_t = \prod_{i=1}^{R} \left(1 - f_t^i w_{t-1}^{r,i}\right).$$

Уравнение сообщает, что уровень, до которого ячейку следует освободить, пропорционален тому, сколько данных прочтено из нее на последних нескольких шагах разными головками (представлены значениями взвешиваний $w_{t-1}^{r,i}$). Но постоянное освобождение ячейки памяти сразу после чтения ее данных нежелательно, поскольку эти данные могут нам понадобиться позже.

Оставим контроллеру возможность решить, когда освобождать, а когда сохранять ячейку после прочтения, выдавая набор из R свободных вентилей $f_t^1, \ldots, f_t^R$, значения которых составляют от 0 до 1. Это определяет, насколько нужно освобождать ячейки, на основе того, что их только что прочли. Контроллер обучается использованию этих вентилей для достижения желаемого поведения. Как только вектор удержания получен, можно с его помощью обновить вектор использования, чтобы отражать все освобождения и сохранения:

$$\mathrm{u}_t = \left(\mathrm{u}_{t-1} + \mathrm{w}_{t-1}^{w} - \mathrm{u}_{t-1} \circ \mathrm{w}_{t-1}^{w}\right) \circ \psi_t.$$

Это уравнение можно прочесть так: ячейка будет использована, если она была сохранена (ее значение $\psi_t \approx 1$) и либо уже в работе, либо в нее только что произведена запись (о чем свидетельствует значение $\mathrm{u}_{t-1} + \mathrm{w}_{t-1}^{w}$). Вычитание поэлементного произведения $\mathrm{u}_{t-1} \circ \mathrm{w}_{t-1}^{w}$ возвращает выражение в диапазон между 0 и 1, значение использования вновь становится действительным (если взвешивание записи при предыдущем использовании вышло за пределы 1).

Обновление использования перед вычислением выделения позволяет ввести свободную память для новых данных. Мы теперь также можем более эффективно использовать (впервые или заново) ограниченные ресурсы памяти, преодолевая второе ограничение NTM.

Временное связывание записей DNC

Учитывая динамические механизмы управления памятью, используемые в DNC, каждый раз, когда надо выделить ячейку памяти, мы получаем наименее используемую. Выходит, здесь нет позиционной связи между ней и предыдущей записью. При таком типе доступа к памяти непрерывный способ сохранения временны́х отношений, принятый в NTM, не подходит. Нам нужно вести непосредственную фиксацию порядка записи данных.

В DNC это достигается при помощи еще двух структур данных, дополняющих матрицу памяти и вектор использования. Первая называется *вектором предшествования* p_t. Это вектор размера N, который должен представлять распределение вероятностей по ячейкам, так что каждое значение соответствует вероятности того, что соответствующая ячейка была последней, в которую производилась запись. Предшествование изначально задается равным нулю ($\mathrm{p}_0 = \mathbf{0}$) и обновляется на следующих шагах:

$$\mathrm{p}_t = \left(1 - \sum_{i=1}^{N} \mathrm{w}_t^w[i]\right)\mathrm{p}_{t-1} + \mathrm{w}_t^w.$$

Обновление происходит при первом умножении предыдущих значений предшествования на коэффициент, пропорциональный тому, что только что записано в память (определяется по сумме компонентов взвешивания записи). Затем значение взвешивания добавляется в значение после сброса, и ячейка с большим взвешиванием (та, в которую запись произведена только что) тоже получит высокое значение вектора предшествования.

Вторая структура данных, необходимая для записи временной информации, — *матрица ссылок* L_t. Это матрица размером $N \times N$, в которой элемент $\mathrm{L}_t[i, j]$ имеет значение от 0 до 1 и показывает, насколько вероятно, что ячейка i была занята после j. На начальном этапе все значения обнулены, а диагональные элементы сохраняются на нулевом уровне в течение времени $\mathrm{L}_t[i, i] = 0$, поскольку бессмысленно фиксировать, что ячейка была записана после себя, когда предыдущие данные уже перезаписаны и утрачены. Все остальные элементы в матрице обновляются по такому принципу:

$$\mathrm{L}_t[i, j] = \left(1 - \mathrm{w}_t^w[i] - \mathrm{w}_t^w[j]\right)\mathrm{L}_{t-1}[i, j] + \mathrm{w}_t^w[i]\mathrm{p}_{t-1}[j].$$

Уравнение создано по тому же шаблону, что и другие правила обновления. Сначала ссылочный элемент сбрасывается при помощи коэффициента, пропорционального уровню записи в ячейки i, j. Затем ссылка обновляется корреляцией (здесь представлена в виде умножения) между взвешиванием записи в ячейке i и предыдущим значением предшествования в j. Так устраняется третье ограничение NTM: мы можем фиксировать временную информацию независимо от того, как головка записи движется по памяти.

Понимание головки чтения DNC

Когда головка записи заканчивает обновлять матрицу памяти и соответствующие структуры данных, в игру вступает головка чтения. Ее задача проста: просматривать значения в памяти и проводить итерации вперед и назад по времени между данными. Возможность просмотра достигается простой адресацией по содержанию: для каждой головки записи `i` мы вычисляем промежуточное значение $c_t^{r,i} = C\left(M_t, k_t^{r,i}, \beta_t^{r,i}\right)$, где $k_t^{r,1}, \ldots, k_t^{r,R}$ и $\beta_t^{r,1}, \ldots, \beta_t^{r,R}$ — два набора ключей чтения R и мощностей, полученных от контроллера по вектору интерфейса.

Чтобы получить итерации вперед и назад, нужно сделать так, чтобы взвешивания могли делать шаг вперед или назад из ячейки, откуда только что производилось чтение. Для движения вперед это достигается умножением матрицы ссылок на последние прочтенные взвешивания. Это сдвигает веса из последней прочтенной ячейки в ячейку из последней записи, указанной в матрице ссылок, и создает промежуточное переднее взвешивание для каждой головки считывания `i`: $f_t^i = L_t w_{t-1}^{r,i}$. Точно так же создаем и промежуточное заднее взвешивание, умножая транспонированную матрицу ссылок на последние прочтенные взвешивания $b_t^i = L_{t-1}^{\top} w_{t-1}^{r,i}$. Теперь можно создать новые взвешивания для каждого чтения в соответствии со следующим правилом:

$$w_t^{r,i} = \pi_t^i[1] b_t^i + \pi_t^i[2] c_t^i + \pi_t^i[3] f_t^i,$$

где $\pi_t^1, \ldots, \pi_t^R$ — *режимы чтения*. Это распределения функции мягкого максимума по трем элементам, которые поступают из контроллера вектора интерфейса. Три значения определяют, какое внимание головка чтения должна уделить каждому механизму: заднему, просмотровому и переднему соответственно. Контроллер обучается использовать эти режимы, чтобы указать памяти, как считывать данные.

Сеть контроллера DNC

Теперь, поняв, как работает внешняя память в архитектуре DNC, осталось выяснить, как устроен контроллер, координирующий все операции в памяти. Работа его проста: в его основе лежит нейронная сеть (рекуррентная или с прямым распространением сигнала), которая читает входные данные на текущем шаге вместе с векторами с предыдущего шага и выдает вектор, размер которого зависит от архитектуры сети. Обозначим этот вектор как $N(\chi_t)$, где N — любая функция, вычисляемая нейронной сетью, а χ_t — конкатенация входного вектора на текущем шаге и последних прочитанных векторов $\chi_t = \left[x_t; r_{t-1}^1; \ldots; r_{t-1}^R\right]$. Конкатенация последних прочитанных векторов служит

той же цели, что и скрытое состояние в обычной LSTM: связать выходные данные с прошлым. От вектора, исходящего из нейронной сети, нам нужна информация двух видов. Первый — вектор интерфейса ζ_t. Как мы уже видели, он содержит всю информацию, чтобы память могла выполнять свою работу. Вектор ζ_t можно рассматривать как конкатенацию уже известных отдельных элементов, как показано на рис. 8.7.

$$\zeta_t = [\underbrace{k_t^{r,1}; \ldots; k_t^{r,R}}_{\text{Каждый размером } W}; \overbrace{\beta_t^{r,1}; \ldots; \beta_t^{r,R}}^{\text{Каждый размером 1}}; k_t^w; \beta_t^w; \underbrace{e_t; v_t}_{\text{Размер } W}; \overbrace{f_t^1; \ldots; f_t^R; g_t^a; g_t^w}^{\text{Каждый размером 1}}; \underbrace{\pi_t^1; \ldots; \pi_t^R}_{\text{Каждый размером 3}}]$$

Рис. 8.7. *Вектор интерфейса, разложенный на компоненты*

Суммируя размеры по компонентам, мы можем считать вектор ζ_t как один большой вектор размером $(R \times W + 3W + 5R + 3)$. Чтобы получить такой вектор на выводе сети, мы создаем матрицу W_ζ обучаемых весов $|N| \times (R \times W + 3W + 5R + 3)$, где $|N|$ — размер выходных данных сети:

$$\zeta_t = W_\zeta N(\chi t).$$

Прежде чем передать вектор ζ_t в память, надо убедиться, что каждый компонент имеет корректное значение. Например, все вентили и вектор стирания должны составлять от 0 до 1, так что мы пропускаем их через сигмоидную функцию, чтобы обеспечить соответствие этому требованию:

$$e_t = \sigma(e_t), f_t^i = \sigma(f_t^i), g_t^a = \sigma(g_t^a), g_t^w = \sigma(g_t^w), \text{ где } \sigma(z) = \frac{1}{1+e^{-z}}.$$

Все просмотровые мощности должны иметь значение не менее 1, так что мы сначала пропускаем их через функцию *oneplus*:

$$\beta_t^{r,i} = \text{oneplus}(\beta_t^{r,i}), \beta_t^w = \text{oneplus}(\beta_t^w), \text{ где } \text{oneplus}(z) = 1 + \log(1 + e^z).$$

Наконец, режимы чтения должны иметь корректное распределение функции мягкого максимума:

$$\pi_t^i = \text{softmax}(\pi_t^i), \text{ где } \text{softmax}(z) = \frac{e^z}{\sum_j e^{z_j}}.$$

После этих преобразований вектор интерфейса можно передать в память; и пока он руководит операциями, нам нужен второй элемент данных от нейронной сети — предварительная версия выходного вектора v_t. Это вектор того же размера, что и окончательный выходной, но им не являющийся. Используя еще одну матрицу W_y обучаемых весов $|N| \times Y$, можно получить этот вектор по формуле:

$$v_t = W_y N(\chi_t).$$

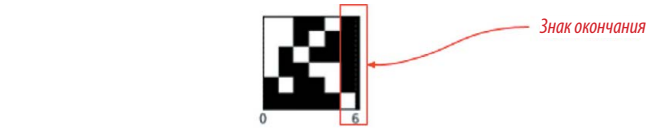
Он дает возможность связать окончательный выходной вектор не только с выходными данными сети, но и с недавно прочитанными из памяти векторами r_t. Из третьей матрицы W_r обучаемых весов $(R \times W) \times Y$ мы можем получить окончательный выходной вектор:

$$y_t = v_t + W_r \left[\mathrm{r}_t^1; \ldots; \mathrm{r}_t^R \right].$$

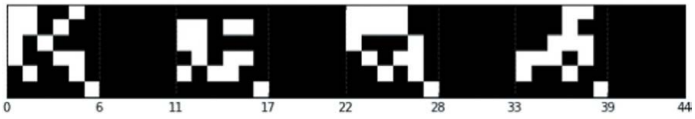
Если контроллер ничего не знает о памяти, кроме размера слова W, обученный контроллер можно масштабировать до большей памяти с большим количеством ячеек без необходимости повторного обучения. Вдобавок то, что нам не пришлось указывать конкретной структуры нейронной сети или конкретной функции потерь, делает DNC универсальным решением, которое можно применить к самому широкому спектру задач обучения.

Визуализация работы DNC

Один из способов увидеть DNC в деле — обучить его на простой задаче, которая позволит посмотреть на взвешивания и значения параметров и визуализировать их удобным для интерпретации способом. Возьмем проблему копирования, с которой мы уже имели дело при разговоре об NTM, но в несколько видоизмененной форме.



а — одна входная последовательность



б — серия входных последовательностей

Рис. 8.8. *Ввод одной последовательности и серии входных последовательностей*

Вместо того чтобы копировать одну последовательность двоичных векторов, мы будем дублировать серии таких последовательностей. На рис. 8.8 (а) показана одна входная последовательность. После ее обработки и копирования на выходе `DNC` завершил бы свою программу, а его память была бы перезагружена, и нам не удалось бы изучить процесс обработки в динамике. Поэтому мы будем рассматривать ряд последовательностей, показанных на рис. 8.8 (б), как единый ввод.

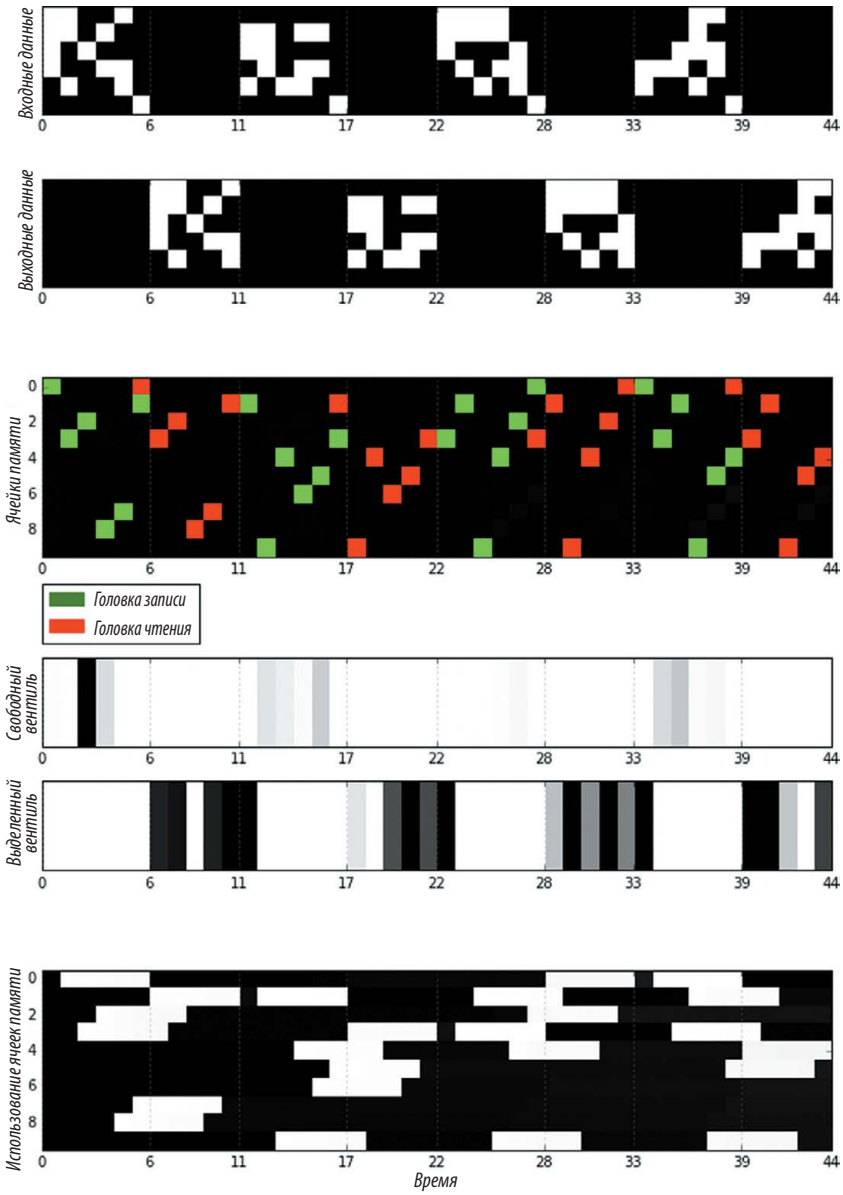


Рис. 8.9. *Визуализация работы DNC над проблемой копирования*

На рис. 8.9 показана визуализация действий DNC, обученного на серии размера 4, где каждая последовательность содержит пять двоичных векторов и знак окончания. Здесь всего 10 ячеек памяти, и все 20 векторов ввода сохранить нельзя. Контроллер с прямым распространением сигнала

обеспечивает, чтобы никакие данные не хранились в рекуррентном состоянии, а единственная головка чтения использована для большей наглядности. Эти ограничения должны заставить DNC научиться освобождению и повторному использованию памяти для успешного копирования всего ввода. Так и происходит.

На визуализации видно, как DNC записывает каждый из пяти векторов последовательности в одну ячейку памяти. После получения знака окончания головка чтения начинает считывать из ячеек в соответствии с порядком записи. Можно видеть, как занятые и свободные вентили чередуют активацию между фазами записи и чтения для каждой последовательности в серии. На графике вектора использования внизу заметно, что после записи в ячейку памяти ее значение использования становится равным 1, а затем снижается до 0 сразу после считывания, показывая, что ячейка освобождена и может быть использована снова.

Эта визуализация — часть открытой реализации архитектуры DNC, выполненной Мостафой Самиром[90]. В следующем разделе мы познакомимся с важными приемами, которые позволят нам реализовать более простую версию DNC для работы с проблемой понимания при чтении.

Реализация DNC в TensorFlow

Реализация архитектуры DNC — по сути, прямое применение математики, о которой мы только что говорили. Полная реализация приведена в репозитории кода для этой книги, здесь же мы сосредоточимся на самых трудных местах и попутно расскажем о некоторых новых методах работы с `TensorFlow`.

Основная часть реализации находится в файле *`mem_ops.py`*: там реализованы все наши механизмы внимания и доступа. Файл импортируется и используется контроллером.

Сложными здесь могут показаться две операции: обновление матрицы ссылок и расчет выделения взвешиваний. Обе можно выполнить наивным способом — с помощью циклов `for`. Но работа с ними для создания графа вычисления — обычно не лучшая идея. Рассмотрим сначала операцию обновления матрицы ссылок. Вот как она выглядит при цикловой реализации:

```
def Lt(L, wwt, p, N):
      L_t = tf.zeros([N,N], tf.float32)
      for i in range(N):
            for j in range(N):
                  if i == j:
```

```
                continue
            _mask = np.zeros([N,N], np.float32);
            _mask[i,j] = 1.0
            mask = tf.convert_to_tensor(_mask)
            link_t = (1 — wwt[i] — wwt[j]) * L[i,j] +
            wwt[i] * p[j]
            L_t += mask * link_t
    return L_t
```

Здесь мы воспользовались уловкой, поскольку `TensorFlow` не поддерживает назначения для элементов тензоров. Можно понять, что тут не так, если вспомнить, что `TensorFlow` — образец *символического* программирования, при котором каждое обращение к `API` не проводит операцию и не изменяет состояние программы, а определяет узел графа вычислений как символ для операции, которую мы хотим выполнить. После того как граф полностью определен, для него задаются конкретные значения, и он выполняется. Получается, как на рис. 8.10, в большинстве итераций цикла `for` к графу вычислений добавляется новый набор узлов, соответствующий телу цикла. Поэтому для N ячеек памяти мы получаем $(N_2 - N)$ идентичных копий одних и тех же узлов — по одной на итерацию.

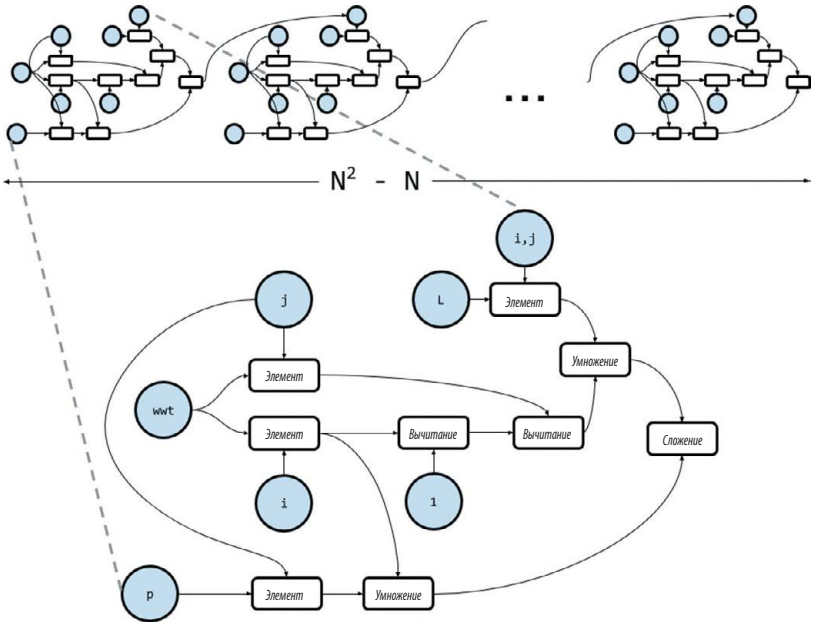


Рис. 8.10. *Граф вычислений операции обновления матрицы ссылок, созданный с помощью цикла for*

Каждая копия отъедает немного оперативной памяти и времени на обработку. Если N — небольшое число, например 5, получится 20 одинаковых копий, что не так плохо. Но если нам нужна большая память, например $N = 256$, то будет уже 65 280 одинаковых копий узлов, а это катастрофа и для памяти, и для времени обработки!

Один из способов преодоления этой проблемы — *векторизация*. Мы выполняем над массивом операцию, которая изначально определяется для конкретных элементов, и перезаписываем ее как операцию над всем массивом. Для обновления матрицы ссылок перезапись может происходить так:

$$\mathrm{L}_t = \left[\left(1 - \mathrm{w}_t^w \oplus \mathrm{w}_t^w\right) \circ \mathrm{L}_{t-1} + \mathrm{w}_t^w \mathrm{p}_{t-1}\right] \circ \left(1 - I\right).$$

Здесь I — единичная матрица, а $\mathrm{w}_t^w \mathrm{p}_{t-1}$ — векторное произведение. Чтобы получить векторизацию, мы задаем новый оператор — попарное сложение векторов, обозначаемое $\oplus$. Его можно определить так:

$$u \oplus v = \begin{pmatrix} u_1 & + & v_1 & \ldots & u_1 & + & v_n \\ & . & . & & & . & \\ & . & & . & & . & \\ & . & & & . & . & \\ u_n & + & v_1 & \ldots & u_n & + & v_n \end{pmatrix}.$$

Этот оператор требует дополнительной памяти, но все же не так много, как при цикловой реализации. При векторизованном переформулировании правила обновления мы можем записать более эффективную с точки зрения памяти и времени реализацию:

```
def Lt(L, wwt, p, N):
# we only need the case of adding a single vector to itself (нам нужен
      только случай добавления единичного вектора к самому себе)
def pairwise_add(v):
n = v.get_shape().as_list()[0]
# an NxN matrix of duplicates of u along the columns (матрицы NxN копий
      u по столбцам)
V = tf.concat(1, [v] * n)
return V + V
I = tf.constant(np.identity(N, dtype=np.float32))
updated = (1 — pairwise_add(wwt)) * L + tf.matmul(wwt, p)
updated = updated * (1 — I) # eliminate self-links
return updated
```

Примерно то же можно сделать и для правила выделения взвешиваний. Одно правило для каждого элемента вектора мы разбиваем на несколько операций, которые будут работать со всем вектором сразу.

1. При сортировке вектора использования для получения списка свободной памяти берем также сам вектор отсортированного использования.

2. Вычисляем вектор кумулятивного произведения отсортированного использования. Каждый элемент соответствует части произведения в первоначальном поэлементном правиле.

3. Умножаем вектор кумулятивного произведения на (`1`-вектор отсортированного использования). Полученный вектор — выделение взвешивания, но в отсортированном порядке, а не в исходном порядке ячейки записи.

4. Для каждого элемента неупорядоченного выделения взвешивания берем его значение и помещаем в соответствующий индекс списка свободной памяти. Полученный вектор — то выделение взвешивания, которое нам и нужно.

На рис. 8.11 приведен этот же процесс с числовыми примерами.

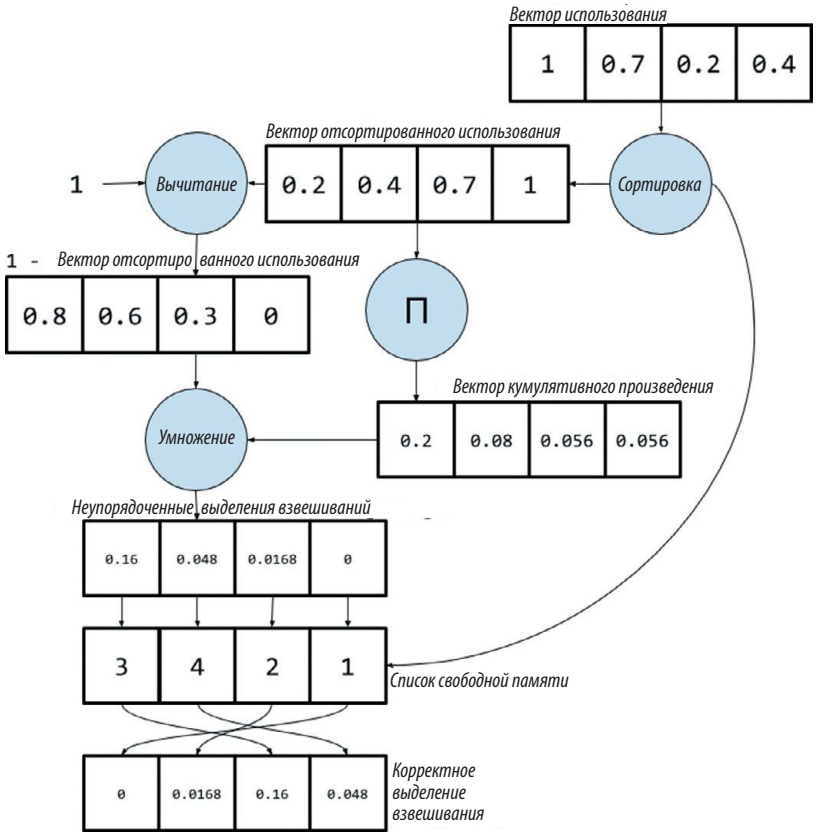


Рис. 8.11. *Векторизованный процесс расчета выделения взвешивания*

Может показаться, что для сортировки на шаге 1 и переупорядочивания весов на шаге 4 нам все равно понадобятся циклы, но, к счастью, TensorFlow предлагает такие символические операции, которые позволяют выполнять эти действия, не прибегая к циклам Python.

Для сортировки мы будем использовать `tf.nn.top_k`. Эта операция принимает тензор и число *k* и выдает как отсортированные высшие значения *k* в порядке убывания, так и их индексы. Чтобы получить вектор отсортированного использования в порядке возрастания, нужно взять высшие значения *N* вектора, обратного вектору использования. Привести отсортированные значения к исходным знакам можно, умножив получившийся вектор на `-1`:

```
sorted_ut, free_list = tf.nn.top_k(-1 * ut, N)

sorted_ut *= -1
```

Для переупорядочивания весов выделения воспользуемся новой структурой данных TensorFlow под названием `TensorArray`. Эти массивы можно считать символической альтернативой списку Python. Сначала получаем пустой массив тензоров размера `N`, содержащий веса в верном порядке, затем помещаем туда значения в правильном порядке методом экземпляра `scatter(indices, values)`. Он получает в качестве второго аргумента тензор и рассеивает значения по первому измерению по всему массиву, причем первый аргумент — список индексов ячеек, по которым мы хотим рассеять соответствующие значения. В нашем случае первый аргумент — список свободной памяти, а второй — неупорядоченные выделения взвешиваний. Получив массив с весами в нужных местах, используем еще один метод экземпляра — `pack()` — для помещения всего массива в объект `Tensor`:

```
empty_at = tf.TensorArray(tf.float32, N)

full_at = empty_at.scatter(free_list, out_of_location_at)

a_t = full_at.pack()
```

Последняя часть реализации, где необходимо введение циклов, — собственно цикл контроллера, который проходит по каждому шагу входной последовательности, обрабатывая ее. Поскольку векторизация работает только при поэлементном определении операций, цикл контроллера векторизовать нельзя. К счастью, TensorFlow снова дает нам возможность избежать циклов `for` в Python, что существенно отразилось бы на производительности; это метод *символического цикла*. Он работает так же, как большинство символических операций: вместо разворачивания

настоящего цикла в граф определяется узел, который пройдет как цикл при выполнении самого графа.

Задать символический цикл можно при помощи `tf.while_loop(cond, body, loop_vars)`. Аргумент `loop_vars` — список изначальных значений тензоров и/или их массивов, которые проходят по каждой итерации цикла; он может быть и вложенным. Два других аргумента — вызываемые объекты (функции или лямбда выражения), которые передаются в этот список переменных цикла при каждой итерации. Первый аргумент `cond` представляет условие цикла. Пока этот вызываемый объект возвращает значение `true`, цикл будет продолжать работу. Второй аргумент `body` — тело цикла, которое исполняется при каждой итерации. Этот вызываемый объект и отвечает за модификацию переменных цикла и возвращение их на следующей итерации. Такие перемены, однако, не должны затрагивать форму тензора, которая на каждой итерации остается неизменной. После выполнения цикла возвращается список его переменных с их значениями после последней итерации.

Чтобы лучше понять, как используются символические циклы, приведем простой пример. Допустим, нам дан вектор значений, и мы хотим получить их кумулятивную сумму. Это достигается с помощью `tf.while_loop`, как в следующем коде:

```
values = tf.random_normal([10])
index = tf.constant(0)
values_array = tf.TensorArray(tf.float32, 10)
cumsum_value = tf.constant(0.)
cumsum_array = tf.TensorArray(tf.float32, 10)
values_array = values_array.unpack(values)
def loop_body(index, values_array, cumsum_value, cumsum_array):
       current_value = values_array.read(index)
       cumsum_value += current_value
       cumsum_array = cumsum_array.write(index, cumsum_value)
       index += 1
       return (index, values_array, cumsum_value, cumsum_array)
_, _, _, final_cumsum = tf.while_loop(
       cond= lambda index, *_: index < 10,
       body= loop_body,
```

```
        loop_vars= (index, values_array, cumsum_value,
             cumsum_array)
)
cumsum_vector = final_cumsum.pack()
```

Сначала мы используем `unpack(values)` массива тензоров, чтобы распаковать значения тензора по первому измерению по массиву. В теле цикла мы получаем значение текущего индекса методом `read(index)` из массива. Затем высчитываем кумулятивную сумму и добавляем ее к массиву кумулятивной суммы методом `write(index, value)`, который записывает заданное значение в массив на заданном индексе. Наконец, после полного выполнения цикла, мы получаем итоговый массив кумулятивной суммы и упаковываем его в тензор. Примерно так же реализуется цикл `DNC` по шагам входной последовательности.

Обучение DNC чтению и пониманию

Ранее в этой главе, когда мы говорили о нейронных `н`-граммах, мы упомянули, что ИИ в данном случае еще не может отвечать на вопросы по прочитанному тексту. Теперь мы достигли точки, когда можем построить систему, которая делает то же, что DNC, в применении к набору данных `bAbI`.

`bAbI` — синтетический набор данных, который состоит из 20 наборов историй, вопросов по ним и ответов. Любая группа данных представляет отдельную задачу, касающуюся рассуждений и выводов на основании текста. В той версии, которую будем использовать мы, для каждой задачи есть 10 тысяч вопросов для обучения и 1000 для тестирования. Например, следующая история (из которой адаптирован уже известный нам отрывок) взята из задачи «списки и наборы», в которой ответы на вопросы — списки или наборы предметов, упомянутых в истории:

```
1 Мэри взяла там молоко.
2 Мэри пошла в офис.
3 Что несет Мэри? молоко 1
4 Мэри взяла там яблоко.
5 Сандра пошла в спальню.
6 Что несет Мэри? молоко,яблоко 1 4
```

Этот пример взят непосредственно из набора данных. История описана в пронумерованных предложениях, начиная с 1. Каждый вопрос заканчивается вопросительным знаком, а слова, которые за ним следуют, — ответы.

Если ответ состоит более из двух или более слов, те разделены запятыми. Числа за ответами — контрольные сигналы, указывающие на предложения, которые содержат слова из ответа.

Усложняя задачу, мы откажемся от контрольных сигналов, чтобы система научилась читать текст и сама находить ответы. В соответствии с исходной работой по DNC проведем предварительную обработку набора данных, убрав все числительные и знаки препинания, за исключением `"?"` и `"."`, переведя все слова в нижний регистр, а все слова в ответе заменив дефисами `"-"` во входящей последовательности. Получим `159` уникальных слов и знаков (лексиконов) по всем заданиям. Преобразуем каждый лексикон в прямой унитарный вектор размера `159`; никаких плотных векторных представлений, только слова. Наконец, мы сочетаем все 200 тысяч тренировочных вопросов для обучения модели на них в совокупности, при этом тестовые вопросы каждого задания отделяем друг от друга, чтобы далее проверять обученную модель на каждом задании отдельно. Весь этот процесс реализован в файле репозитория кода *`preprocess.py`*.

Мы случайным образом выбираем историю из закодированных обучающих данных, пропускаем ее через DNC на контроллер `LSTM` и получаем соответствующую выходную последовательность. Затем измеряем потери между выходной последовательностью и желаемой с помощью функции потерь мягкого максимума перекрестной энтропии, но только для шагов, которые содержат ответы. Все остальные игнорируются: мы назначаем вектору весов значения 1 на шагах с ответами и значения 0 в остальных местах. Этот процесс реализован в файле *`train_babi.py`*.

После того как модель обучена, тестируем ее работоспособность на оставшихся тестовых вопросах. Нашей метрикой будет процент вопросов, на которые модель не смогла ответить в рамках каждого задания. Ответ — слово с наибольшим значением функции мягкого максимума на выходе, то есть наиболее вероятное слово. Считается, что ответ верный, если все слова в нем правильные. Если модель не смогла ответить более чем на 5% вопросов в задании, считается, что она не справилась. Процедуру тестирования можно найти в файле *`test_babi.py`*.

После обучения модели на примерно 500 тысяч итераций (это может занять очень много времени!) оказывается, что с большинством заданий она справилась очень хорошо. Но она плохо выполняет более сложные задачи, такими как *ориентирование*, где надо отвечать на вопросы о том, как попасть из одного места в другое. В нижеследующем отчете сравниваются результаты нашей модели со средними значениями из первой работы по DNC.

Задание	Результат, %	Среднее значение
Один факт	0	9±12,6%
Два факта	11,88	39,2±20,5%
Три факта	27,8	39,6±16,4%
Две связи аргументов	1,4	0,4±0,7%
Три связи аргументов	1,7	1,5±1%
Вопросы "да"/"нет"	0,5	6,9±7,5%
Подсчет	4,9	9,8±7%
Списки/наборы	2,1	5,5±5,9%
Простое отрицание	0,8	7,7±8,3%
Неопределенное знание	1,7	9,6±11,4%
Базовое тождество по референту	0,1	3,3±5,7%
Конъюнкция	0	5,0±6,3%
Составное тождество по референту	0,4	3,1±3,6%
Временное рассуждение	11,8	11±7,5%
Базовая дедукция	45,44	27,2±20,1%
Базовая индукция	56,43	53,6±1,9%
Позиционное рассуждение	39,02	32,4±8%
Рассуждение о размерах	8,68	4,2±1,8%
Ориентирование	98,21	64,6±37,4%
Мотивация действующих лиц	2,71	0±0,1%
Средняя квадратичная ошибка	15,78	16,7±7,6%
Не выполнено (ошибок > 5%)	8	11,2±5,4%

Резюме

В этой главе мы рассмотрели проблемы переднего края науки о глубоком обучении — работу с NTM и DNC, завершив реализацией модели, которая может решать задачу понимания чтения.

В последней главе мы начнем изучать иную сферу: обучение с подкреплением. Мы познакомимся с новым классом задач и подготовим алгоритмические основы решения при помощи уже созданных нами инструментов глубокого обучения.

ГЛАВА 9

Глубокое обучение с подкреплением

Николас Локашо[91]

В этой главе мы рассмотрим обучение с подкреплением — раздел машинного обучения, требующего взаимодействия и обратной связи. Это необходимо для создания агентов, которые будут не просто воспринимать и интерпретировать мир, но и взаимодействовать с ним. Мы расскажем, как внедрить глубокие нейронные сети в структуру обучения с подкреплением, и обсудим последние достижения и улучшения в этой области.

Глубокое обучение с подкреплением и игры Atari

Применение глубоких нейронных сетей к обучению с подкреплением стало важным прорывом в 2014 году, когда лондонский стартап DeepMind поразил специалистов по машинному обучению, представив глубокую нейронную сеть, которая справлялась с играми компании Atari лучше, чем люди. Эта сеть, получившая название `Deep Q-Network (DQN)`, стала первым масштабным успешным применением обучения с подкреплением с глубокими нейронными сетями. Она оказалась особенно примечательной, потому что одна и та же архитектура без изменений смогла освоить 49 разных игр, различающихся правилами, целями и стратегиями. Создатели DeepMind свели воедино многие традиционные идеи обучения с подкреплением, разработав и несколько новаторских методов, которые оказались ключевыми для успеха. В этой главе мы рассмотрим реализацию `DQN`, как она представлена в публикации в журнале Nature под названием «Управление на уровне человека с помощью глубокого обучения с подкреплением»[92]. Но сначала подробнее рассмотрим суть метода (рис. 9.1).

Рис. 9.1. *Агент глубокого обучения с подкреплением играет в Breakout. Изображение из агента DQN OpenAI Gym[93], который будет реализован в этой главе*

Что такое обучение с подкреплением?

По сути, это обучение путем взаимодействия со средой. Процесс включает агента, среду и сигнал вознаграждения. Агент решает совершить действие в среде и за это получает соответствующее вознаграждение. Способ,

которым он выбирает, что совершить, называется *стратегией*. Агент хочет увеличить вознаграждение, так что он должен научиться оптимальной стратегии взаимодействия со средой (рис. 9.2).

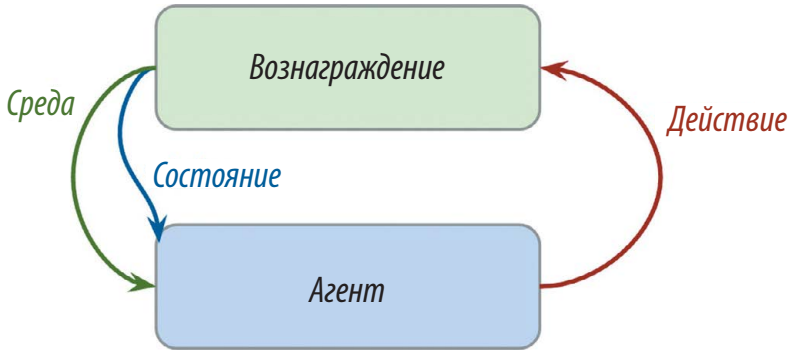


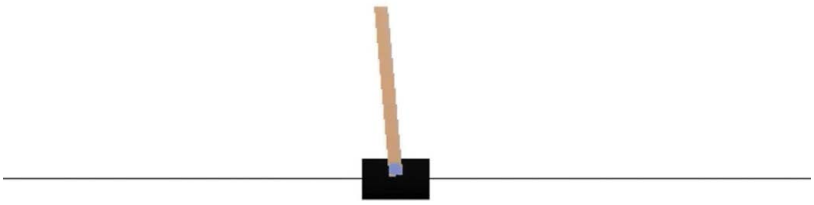
Рис. 9.2. *Схема обучения с подкреплением*

Обучение с подкреплением отличается от остальных типов, о которых мы говорили ранее. При традиционном подходе с учителем у нас есть данные и метки, а задача — предсказывать последние на основании данных. В освоении навыков без учителя у нас есть только данные, а задача — поиск структур в их основе. В обучении же с подкреплением нет ни данных, ни меток. Сигнал поступает от вознаграждений, получаемых от среды.

Обучение с подкреплением сейчас интересно многим причастным к работе над искусственным интеллектом, поскольку это общая структура создания разумных агентов. Агент учится взаимодействовать со средой, чтобы увеличить общее вознаграждение. Это лучше соответствует модели развития человека. Да, мы можем построить очень хорошую и точную модель классификации изображений кошек и собак, обучив ее на тысячах рисунков. Но такой подход не используется в начальных школах. Люди взаимодействуют со средой, усваивая представления о мире, на основе которых смогут позже принимать решения. Практическое применение обучения с подкреплением обнаруживаются во многих передовых технологиях: автомобилях без водителя, роботизированном управлении двигателем, играх, контроле кондиционирования воздуха, оптимизации рекламы и стратегиях торговли на фондовом рынке.

В качестве иллюстрации рассмотрим простой пример для решения проблемы управления — балансировку шеста. В задаче есть тележка с шестом, который прикреплен к нему на шарнире и может раскачиваться. Есть также агент, который управляет тележкой, — двигает ее влево или вправо. Есть

среда, которая вознаграждает агента, если шест направлен вверх, и штрафует, если тот падает вниз (рис. 9.3).



Эпизод 1000

Рис. 9.3. *Простой агент обучения с подкреплением, балансирующий шест. Изображение из агента OpenAI Gym Policy Gradient, который будет создан в этой главе*

Марковские процессы принятия решений (MDP)

В нашем примере с балансировкой шеста есть несколько важных элементов, которые можно формализовать как `марковские процессы принятия решений (MDP)`. Вот они.

Состояние

У тележки есть ряд возможных положений на оси *x*. У шеста — ряд возможных углов.

Действие

Агент может совершить действие — сдвинуть тележку влево или вправо.

Переход состояний

Когда агент действует, среда меняется: тележка двигается, шест изменяет угол и скорость.

Вознаграждение

Если агент хорошо балансирует шест, он получает позитивное вознаграждение. Если шест падает, следует негативное подкрепление.

MDP определяется следующим:

- S, конечное множество возможных состояний;
- A, конечное множество действий;
- $P(r, s' | s, a)$, функция перехода между состояниями;
- R, функция вознаграждения.

MDP дают математическую структуру для моделирования принятия решений в заданной среде (рис. 9.4).

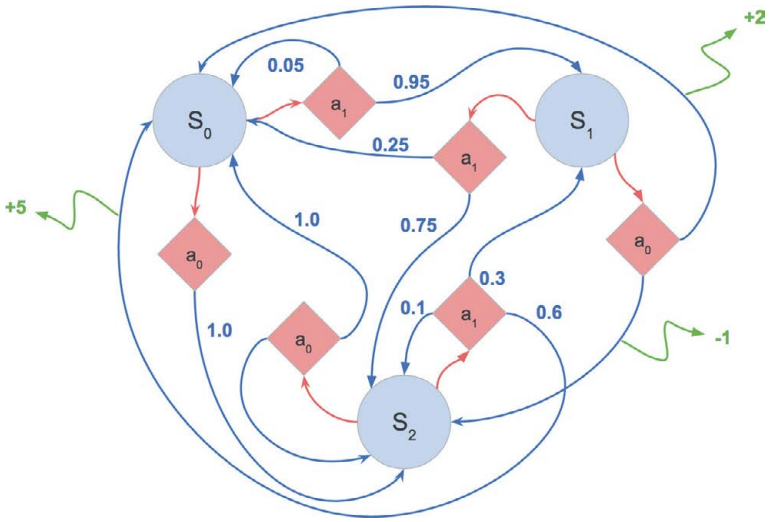


Рис. 9.4. *Пример марковского процесса принятия решений. Голубые кружки обозначают состояния среды. Красные ромбы соответствуют возможным действиям. Стрелки от ромбов к кругам отображают переход из одного состояния в другое. Числа при них соответствуют вероятности действия. Числа в конце зеленых стрелок показывают вознаграждение, которое выдается агенту за выполнение соответствующего перехода*

Когда агент совершает действие в структуре MDP, образуется *эпизод*. Он состоит из серии кортежей состояний, действий и вознаграждений. Эпизоды сменяются, пока среда не достигает конечного состояния: например, экрана Game Over в играх Atari или падения шеста в примере с тележкой и шестом. Следующее уравнение показывает все переменные эпизода:

$$(s_0, a_0, r_0), (s_1, a_1, r_1), \ldots (s_n, a_n, r_n).$$

В примере с тележкой (cart) и шестом (pole) состояние среды может быть кортежем из положения тележки и угла шеста, например: $(x_{cart}, \theta_{pole})$.

СТРАТЕГИЯ

Цель MDP — найти оптимальную стратегию для агента. Стратегия — способ действия в зависимости от текущего состояния. Формально ее можно представить в виде функции π, которая выбирает действие a, выполняемое агентом в состоянии s. Цель MDP — найти стратегию максимального увеличения ожидаемой будущей выгоды: $\max_\pi E\left[R_0 + R_1 + ...R_t \mid \pi\right]$. Здесь R отражает *будущую выгоду* от каждого эпизода. А теперь дадим ее более строгое определение.

БУДУЩАЯ ВЫГОДА

Будущая выгода — ожидаемые вознаграждения. Выбор оптимального действия требует учитывать не только непосредственные результаты, но и долгосрочные последствия. Например, агент-альпинист, получающий вознаграждения за достижение высоты, может решить немного спуститься, чтобы перейти на более удобный путь к вершине горы.

Мы хотим, чтобы наши агенты были оптимизированы в отношении *будущей выгоды*. Для этого им стоит учитывать отдаленные последствия своих действий. Например, в пинг-понге агент получает вознаграждение, когда соперник не может отбить его удар. Но действия, ведущие к этому (входные данные, которые обусловили положение ракетки, позволяющее нанести решающий удар), происходят за много шагов до получения вознаграждения, и его стоит считать отложенным.

Мы можем включить отложенные вознаграждения в общий сигнал, создав *выгоду* для каждого шага, которая будет учитывать как немедленные, так и будущие вознаграждения. Наивный подход к вычислению *будущей выгоды* на данном шаге — простая сумма вроде следующей:

$$R_t = \sum_{k=0}^{T} r_{t+k}.$$

Мы можем вычислить все выгоды, R, где $R = \{R_0, R_1, \ldots R_i, \ldots R_n\}$, с помощью следующего кода:

```
def calculate_naive_returns(rewards):
""" Calculates a list of naive returns given a
list of rewards.""" (Вычисляет список наивных выгод на основании списка
      вознаграждений)
total_returns = np.zeros(len(rewards))
total_return = 0.0
for t in range(len(rewards), 0):
```

```
total_return = total_return + reward
total_returns[t] = total_return
return total_returns
```

Этот подход включает будущие выгоды, и агент может научиться оптимальной общей стратегии. Здесь будущие выгоды ценятся так же, как и немедленные. Но это-то и беспокоит. При бесконечном числе шагов это выражение может свестись к бесконечности, так что нужно установить для него предел. Более того, если на каждом этапе все выгоды расценивать одинаково, агент может оптимизировать действия для очень отдаленной выгоды, и получится стратегия, в которой не учитываются срочность или иной вариант зависимости вознаграждения от времени.

Поэтому следует оценивать будущие вознаграждения чуть ниже, чтобы наши агенты могли научиться получать их быстрее. Этого можно достичь с помощью *дисконтирования будущих выгод*.

ДИСКОНТИРОВАНИЕ БУДУЩИХ ВЫГОД

Для реализации этого подхода мы умножаем вознаграждение текущего состояния на коэффициент дисконтирования γ в степени текущего шага. Тем самым мы штрафуем агентов, которые совершают много действий до получения положительного вознаграждения. Благодаря дисконтированию наш агент будет выбирать вознаграждения в недалеком будущем, что позволит развить хорошую стратегию. Это вознаграждение можно выразить так:

$$R_t = \sum_{k=0}^{T} \gamma^t r_{t+k+1}.$$

Коэффициент дисконтирования γ отражает уровень снижения, которого мы хотим достичь, и может принимать значение от 0 до 1. Высокий γ соответствует небольшому дисконтированию, низкий — значительному. Типичное значение гиперпараметра γ — между 0,99 и 0,97.

Реализовать дисконтирование выгоды можно так:

```
def discount_rewards(rewards, gamma=0.98):
       discounted_returns = [0 for _ in rewards]
       discounted_returns[-1] = rewards[-1]
       for t in range(len(rewards)-2, -1, -1): # iterate backwards
       discounted_returns[t] = rewards[t] +
       discounted_returns[t+1]*gamma
       return discounted_returns
```

Исследование и использование

Обучение с подкреплением — по сути, метод проб и ошибок. В таких условиях агент, опасающийся провала, будет не очень-то эффективен. Рассмотрим такой сценарий. Мышь помещают в лабиринт, показанный на рис. 9.5. Агент должен управлять ею для получения максимального вознаграждения. Если мышь находит воду, она получает +1; если контейнер с ядом (красный), то –10; за нахождение сыра дается +100. После вознаграждения эпизод заканчивается. Оптимальная стратегия должна помочь мыши успешно добраться до сыра и съесть его.

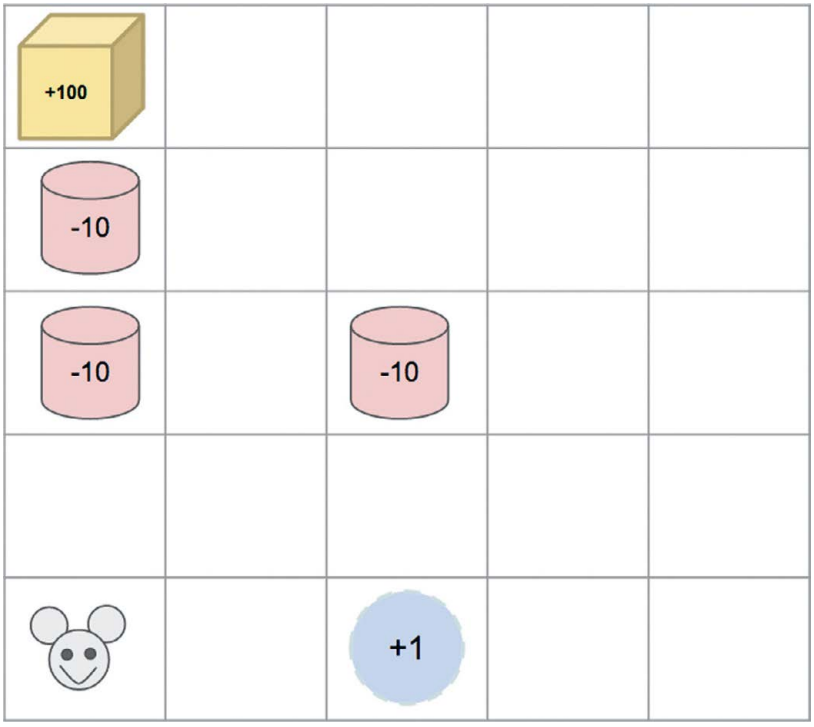


Рис. 9.5. *Ситуация, в которой оказываются многие мыши*

В первом эпизоде мышь идет влево, попадает в ловушку и получает –10. Во втором она избегает движения налево, поскольку результат оказался отрицательным, сразу выпивает воду, находящуюся справа, и получает +1.

После двух эпизодов может показаться, что мышь нашла хорошую стратегию. Животное продолжает и дальше следовать ей и получает скромную, но гарантированную награду +1. Поскольку агент следует жадной

стратегии, всегда выбирая лучшее действие в модели, он оказывается в ловушке *локального максимума*.

Чтобы предотвратить такие ситуации, агенту полезно отклониться от рекомендации модели и выбрать неоптимальное действие, чтобы лучше *исследовать* среду.

Вместо того чтобы сразу сделать шаг направо и *использовать* среду, получив воду и гарантированные +1, агент может однажды шагнуть налево и отправиться в более опасные области в поисках лучшей стратегии. Однако чрезмерное исследование приведет к отсутствию награды. Недостаточные же приведут в ловушку локального максимума. Такой баланс *исследования и использования* необходим для обучения успешной стратегии.

ϵ-ЖАДНОСТЬ

Стратегия уравновешивания дилеммы исследования и использования называется *ϵ-жадностью*. Это простая стратегия, связанная с выбором на каждом шаге между наиболее рекомендуемым действием и каким-нибудь случайным.

Вероятность того, что агент предпочтет случайное действие, определяется значением ϵ.

Реализовать стратегию ϵ-жадности можно так:

```
def epsilon_greedy_action(action_distribution, epsilon=1e-1):
      if random.random() < epsilon:
      return np.argmax(np.random.random(
      action_distribution.shape))
      else:
      return np.argmax(action_distribution)
```

НОРМАЛИЗОВАННЫЙ АЛГОРИТМ ϵ-ЖАДНОСТИ

При обучении модели с подкреплением вначале мы часто хотим выбрать исследование, поскольку наша модель мало знает мир. Потом, когда она хорошо познакомится со средой и научится хорошей стратегии, нам будет желательно, чтобы агент больше доверял себе, — это приведет к дальнейшей оптимизации стратегии. И мы отказываемся от идеи фиксированного ϵ и периодически нормализуем его со временем, начиная с небольшого значения и после каждого эпизода обучения увеличивая его на какой-то

коэффициент. Типичные условия нормализованных сценариев ϵ-жадности включают нормализацию с 0,99 до 0,1 за 10 тысяч подходов. Реализовать нормализацию можно так:

```
def epsilon_greedy_action_annealed(action_distribution,
            percentage,
            epsilon_start=1.0,
            epsilon_end=1e-2):
      annealed_epsilon = epsilon_start*(1.0-percentage) +
      epsilon_end*percentage
      if random.random() < annealed_epsilon:
            return np.argmax(np.random.random(
            action_distribution.shape))
      else:
            return np.argmax(action_distribution)
```

Изучение стратегии и ценности

Мы определили архитектуру обучения с подкреплением, поговорили о дисконтировании будущей выгоды и рассмотрели компромиссы использования и исследования. Но пока мы не упоминали о том, как собираемся научить агента максимизировать выгоду.

Подходы к вопросу делятся на две большие категории: изучение стратегии и изучение ценности. В первом случае мы непосредственно обучаем стратегии, которая доводит выгоду до максимума. Во втором мы изучаем ценность каждой пары «состояние + действие». Для осваивающего велосипед изучение стратегии соответствует мыслям о том, как нажатие на правую педаль при падении влево восстановит равновесие. Если же действовать методом изучения ценности, нужно определять ценности для различных положений машины и действий, которые вы можете предпринять в этих условиях. Мы поговорим об обоих подходах, а начнем с изучения стратегии.

ИЗУЧЕНИЕ СТРАТЕГИИ ПРИ ПОМОЩИ ГРАДИЕНТА ПО СТРАТЕГИЯМ

В обычном освоении навыков с учителем можно использовать стохастический градиентный спуск для обновления параметров и минимизации потерь, вычисленных на основе выходных данных сети и правильной метки. Оптимизируем выражение:

$$\arg\min_{\theta} \sum_i \log p\left(y_i \mid x_i;\theta\right).$$

В обучении с подкреплением правильной метки нет — только сигналы вознаграждения. Но и здесь можно применить стохастический градиентный спуск для оптимизации весов, использовав *градиенты по стратегиям*[94]. Мы можем воспользоваться действиями, которые совершает агент, и выгодами, связанными с ними, чтобы веса модели принимали действия, которые приводят к высоким наградам, и отвергали ведущие к негативному подкреплению. Оптимизируем выражение:

$$\arg\min_{\theta} -\sum_i R_i \log p\left(y_i \mid x_i;\theta\right),$$

где y_i — действие агента на шаге t, а R_i — дисконтированная будущая выгода. Мы умножаем потери на значение выгоды, так что, если модель выбирает действие, ведущее к отрицательному результату, то потери возрастают. Если модель продолжает упорствовать, она получит еще больший штраф, ведь мы учитываем вероятность того, что она выберет это действие. Определив функцию потерь, мы можем применить стохастический градиентный спуск для их минимизации и обучения грамотной стратегии.

Тележка с шестом и градиенты по стратегиям

Сейчас мы реализуем агент градиента по стратегиям для решения проблемы тележки с шестом — классической задачи в обучении с подкреплением. Мы воспользуемся средой из `OpenAi Gym`, созданной как раз для этого задания.

OPENAI GYM

`OpenAI Gym` — инструментальная библиотека `Python` для разработки агентов с подкреплением. Она предлагает простой в использовании интерфейс для взаимодействия с рядом сред и содержит более сотни открытых реализаций самых распространенных сред для обучения с подкреплением. `OpenAI Gym` ускоряет разработку агентов обучения с подкреплением, беря на себя всю работу по симуляции среды и позволяя исследователям сосредоточиться на агенте и алгоритмах. Еще одно преимущество `OpenAI Gym` в том, что исследователи могут сравнивать свои результаты с другими, поскольку все получают одну и ту же стандартизированную среду для решения задачи. Мы воспользуемся задачей тележки с шестом для создания агента, который сможет легко взаимодействовать с этой средой.

СОЗДАНИЕ АГЕНТА

Чтобы создать агента, который сможет взаимодействовать со средой `OpenAI`, определим класс `PGAgent`, который будет содержать архитектуру модели, ее веса и гиперпараметры:

```
class PGAgent(object):
      def __init__(self, session, state_size, num_actions,
           hidden_size, learning_rate=1e-3,
           explore_exploit_setting=
           'epsilon_greedy_annealed_1.0->0.001'):
           self.session = session
           self.state_size = state_size
           self.num_actions = num_actions
           self.hidden_size = hidden_size
           self.learning_rate = learning_rate
           self.explore_exploit_setting = explore_exploit_setting
           self.build_model()
           self.build_training()
      def build_model(self):
           with tf.variable_scope('pg-model'):
           self.state = tf.placeholder(
           shape=[None, self.state_size],
           dtype=tf.float32)
           self.h0 = slim.fully_connected(self.state,
           self.hidden_size)
           self.h1 = slim.fully_connected(self.h0,
           self.hidden_size)
           self.output = slim.fully_connected(
           self.h1, self.num_actions,
           activation_fn=tf.nn.softmax)
           # self.output = slim.fully_connected(self.h1,
           self.num_actions)
      def build_training(self):
           self.action_input = tf.placeholder(tf.int32,
           shape=[None])
           self.reward_input = tf.placeholder(tf.float32,
           shape=[None])
```

```
        # Select the logits related to the action taken (Выбираем
        логиты, относящиеся к данному действию)
        self.output_index_for_actions = (tf.range(
        0, tf.shape(self.output)[0]) *
        tf.shape(self.output)[1]) +
        self.action_input
        self.logits_for_actions = tf.gather(
        tf.reshape(self.output, [-1]),
        self.output_index_for_actions)
        self.loss = - \
        tf.reduce_mean(tf.log(self.logits_for_actions) *
        self.reward_input)
        self.optimizer = tf.train.AdamOptimizer(
        learning_rate=self.learning_rate)
        self.train_step = self.optimizer.minimize(self.loss)
    def sample_action_from_distribution(
        self, action_distribution,
        epsilon_percentage):
        # Choose an action based on the action probability
        # distribution and an explore vs exploit (Выбираем действие
        на основании распределения вероятностей действий
        и соотношения исследования и использования)
        if self.explore_exploit_setting == 'greedy':
        action = greedy_action(action_distribution)
        elif self.explore_exploit_setting ==
        'epsilon_greedy_0.05':
        action = epsilon_greedy_action(action_distribution,
               0.05)
        elif self.explore_exploit_setting ==
               'epsilon_greedy_0.25':
        action = epsilon_greedy_action(action_distribution,
               0.25)
        elif self.explore_exploit_setting ==
        'epsilon_greedy_0.50':
        action = epsilon_greedy_action(action_distribution,
               0.50)
        elif self.explore_exploit_setting ==
```

```
            'epsilon_greedy_0.90':
            action = epsilon_greedy_action(action_distribution,
                   0.90)
            elif self.explore_exploit_setting ==
            'epsilon_greedy_annealed_1.0->0.001':
            action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage, 1.0,
                   0.001)
            elif self.explore_exploit_setting ==
            'epsilon_greedy_annealed_0.5->0.001':
            action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage, 0.5,
                   0.001)
            elif self.explore_exploit_setting ==
            'epsilon_greedy_annealed_0.25->0.001':
            action = epsilon_greedy_action_annealed(
            action_distribution, epsilon_percentage, 0.25,
                   0.001)
            return action
      def predict_action(self, state, epsilon_percentage):
            action_distribution = self.session.run(
            self.output, feed_dict={self.state: [state]})[0]
            action = self.sample_action_from_distribution(
            action_distribution, epsilon_percentage)
            return action
```

СОЗДАНИЕ МОДЕЛИ И ОПТИМИЗАТОРА

Разберем некоторые важные функции. В `build_model()` мы определяем архитектуру модели как трехслойную нейронную сеть. Модель возвращает слой из трех узлов, каждый из которых представляет распределение вероятностей действия. В `build_training()` мы реализуем оптимизатор градиента по стратегии. Выражаем целевые потери так, как уже говорили, умножая предсказание вероятности действия на вознаграждение за него и суммируем все результаты, формируя мини-пакет. Определив цели, можно использовать `tf.AdamOptimizer`, который обновит веса в соответствии с градиентом для минимизации потерь.

СЕМПЛИРОВАНИЕ ДЕЙСТВИЙ

Задаем функцию `predict_action`, которая семплирует действие на основании выходного распределения вероятностей действия модели. Мы поддерживаем разные стратегии семплирования, о которых говорилось выше, чтобы достичь равновесия исследования и использования, в том числе жадную, ϵ-жадную и нормализованную ϵ-жадную стратегии.

ФИКСАЦИЯ ИСТОРИИ

Мы будем объединять градиенты запусков множества эпизодов, так что полезно будет записывать кортежи состояния, действия и вознаграждения. Для этого реализуем историю и память эпизода.

```
class EpisodeHistory(object):
      def __init__(self):
            self.states = []
            self.actions = []
            self.rewards = []
            self.state_primes = []
            self.discounted_returns = []
      def add_to_history(self, state, action, reward,
            state_prime):
            self.states.append(state)
            self.actions.append(action)
            self.rewards.append(reward)
            self.state_primes.append(state_prime)
            class Memory(object):
      def __init__(self):
            self.states = []
            self.actions = []
            self.rewards = []
            self.state_primes = []
            self.discounted_returns = []
      def reset_memory(self):
            self.states = []
            self.actions = []
            self.rewards = []
```

```
            self.state_primes = []
            self.discounted_returns = []
        def add_episode(self, episode):
            self.states += episode.states
            self.actions += episode.actions
            self.rewards += episode.rewards
            self.discounted_returns += episode.discounted_returns
```

ОСНОВНАЯ ФУНКЦИЯ ГРАДИЕНТА ПО СТРАТЕГИЯМ

Соединим всё это в нашей основной функции, которая создаст среду `OpenAI Gym` для примера `CartPole` (тележка с шестом), задаст пример агента и заставит его взаимодействовать со средой `CartPole` и обучаться на ней.

```
def main(argv):
    # Configure Settings (Конфигурируем настройки)
    total_episodes = 5000
    total_steps_max = 10000
    epsilon_stop = 3000
    train_frequency = 8
    max_episode_length = 500
    render_start = -1
    should_render = False
    explore_exploit_setting =
    'epsilon_greedy_annealed_1.0->0.001'
    env = gym.make('CartPole-v0')
    state_size = env.observation_space.shape[0] # 4 for
    # CartPole-v0
    num_actions = env.action_space.n # 2 for CartPole-v0
    solved = False
    with tf.Session() as session:
    agent = PGAgent(session=session, state_size=state_size,
    num_actions=num_actions,
    hidden_size=16,
    explore_exploit_setting=
    explore_exploit_setting)
    session.run(tf.global_variables_initializer())
```

```
episode_rewards = []
batch_losses = []
global_memory = Memory()
steps = 0
for i in tqdm.tqdm(range(total_episodes)):
state = env.reset()
episode_reward = 0.0
episode_history = EpisodeHistory()
epsilon_percentage = float(min(i/float(
epsilon_stop), 1.0))
for j in range(max_episode_length):
action = agent.predict_action(state,
epsilon_percentage)
state_prime, reward, terminal, _ =
env.step(action)
if (render_start > 0 and i >
render_start and should_render) \
or (solved and should_render):
env.render()
episode_history.add_to_history(
state, action, reward, state_prime)
state = state_prime
episode_reward += reward
steps += 1
if terminal:
episode_history.discounted_returns =
discount_rewards(
episode_history.rewards)
global_memory.add_episode(
episode_history)
if np.mod(i, train_frequency) == 0:
feed_dict = {
agent.reward_input: np.array(
global_memory.discounted_returns),
agent.action_input: np.array(
global_memory.actions),
agent.state: np.array(
```

```
global_memory.states)}
_, batch_loss = session.run(
[agent.train_step, agent.loss],
feed_dict=feed_dict)
batch_losses.append(batch_loss)
global_memory.reset_memory()
episode_rewards.append(episode_reward)
break
if i % 10:
if np.mean(episode_rewards[:-100]) >
100.0:
solved = True
else:
solved = False
```

Этот код обучит агента `CartPole` успешно и надежно удерживать шест в равновесии.

РАБОТА PGAGENT В ПРИМЕРЕ С ТЕЛЕЖКОЙ С ШЕСТОМ

Рисунок 9.6 — таблица средних вознаграждений нашего агента на каждом шаге обучения. Мы пробуем восемь разных методов семплирования, а лучший результат достигается при помощи нормализованной ϵ-жадной стратегии (от 1 до 0,001).

Отметим, что в целом стандартный ϵ-жадный алгоритм дает очень плохие результаты. Обсудим, почему так происходит. Если задано верхнее значение $\epsilon = 0{,}9$, мы совершаем случайные действия 90% времени.

Даже если модель научится выполнять идеальные действия, это будет использовано всего в 10% случаев. А вот если значение ϵ низкое — 0,05, мы в подавляющем большинстве случаев совершаем действия, которые модель считает оптимальными. Эффективность выше, но велика вероятность скатиться к локальному максимуму вознаграждения, поскольку почти нет возможности исследовать другие стратегии.

Итак, ϵ-жадный алгоритм не дает хороших результатов ни при 0,05, ни при 0,9: исследованию уделяется либо слишком много, либо слишком мало внимания. Вот почему нормализация ϵ оказывается хорошей стратегией семплирования. Она позволяет модели сначала исследовать, а затем использовать, что необходимо для изучения хорошей стратегии.

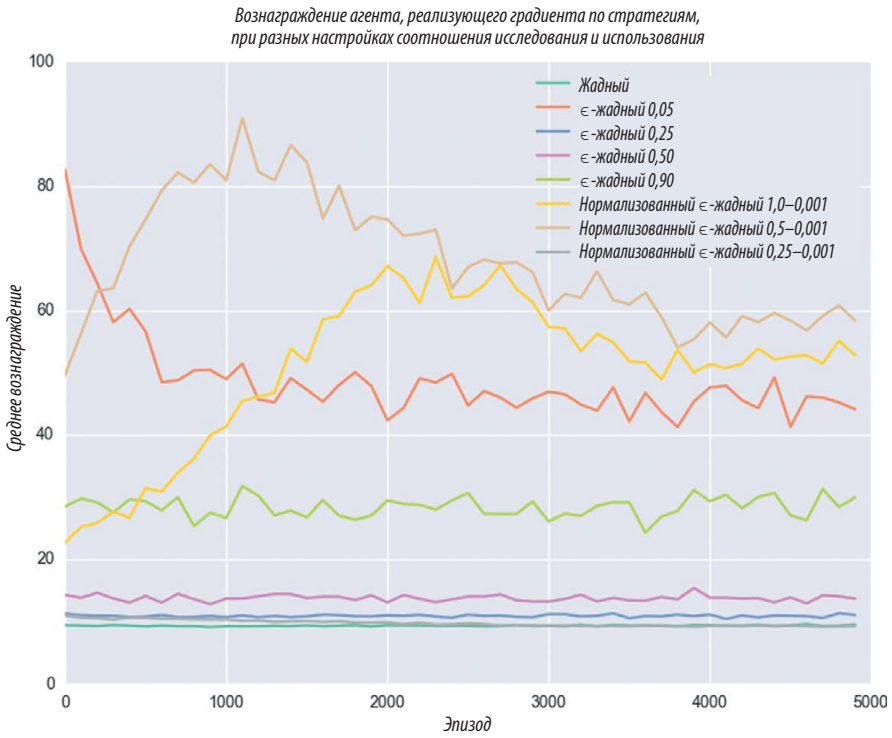


Рис. 9.6. *Конфигурация соотношения исследования и использования влияет на скорость и успешность обучения*

Q-обучение и глубокие Q-сети

`Q`-обучение — категория модели с подкреплением, именуемая обучением ценности. Вместо непосредственного исследования стратегии мы будем усваивать ценности состояний и действий.

`Q`-обучение связано с исследованием *`Q`-функции*, которая отражает качество пары (состояние, действие). `Q(s, a)` — функция, которая рассчитывает максимальную дисконтированную будущую выгоду от совершения действия `a` в состоянии `s`.

Значение `Q` отражает ожидаемые долгосрочные выгоды, если мы в соответствующем состоянии и совершаем соответствующее действие, а затем идеально выполняем все последующие (чтобы получить максимальную ожидаемую будущую выгоду). Формально это можно выразить так:

$$Q^*(s_t, a_t) = \max_\pi E\left[\sum_{i=t}^{T} \gamma^i r^i\right].$$

Возможно, вы задаетесь вопросом: как узнать значения Q? Ведь даже людям тяжело понять, насколько хорошо то или иное действие, поскольку надо знать, как вы собираетесь поступать в будущем. Ожидаемые выгоды зависят от нашей долгосрочной стратегии. Это похоже на проблему курицы и яйца: чтобы оценить пару (состояние, действие), нужно знать все идеальные дальнейшие действия. А чтобы знать, какие будущие действия окажутся идеальными, нужно иметь точно рассчитанные стоимости состояния и действия.

УРАВНЕНИЕ БЕЛЛМАНА

Мы решаем эту дилемму, определяя значения Q как функцию от будущих значений Q. Такие отношения называются уравнением Беллмана, которое утверждает, что максимальная будущая выгода от действия a — текущая выгода плюс максимальная будущая на следующем шаге от совершения следующего действия a':

$$Q^*(s_t, a_t) = E\left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a')\right].$$

Это рекурсивное определение позволяет установить соответствие между значениями Q в прошлом и будущем, и уравнение удобно задает правило обновления. Мы можем обновить предыдущие значения Q так, чтобы они основывались на будущих. И здесь очень удачно, что мы точно знаем одно верное значение Q: это Q для самого последнего действия перед окончанием эпизода.

Для этого состояния мы точно знаем, что следующее действие привело к новому вознаграждению, и можем точно задать значения Q. Теперь можно использовать правило обновления для распространения этого значения на предыдущий шаг:

$$\hat{Q}_j \to \hat{Q}_{j+1} \to \hat{Q}_{j+2} \to \ldots \to Q^*.$$

Такое обновление называется *итерацией по значениям*.

Первое значение Q оказывается неверным, но это приемлемо. С каждой итерацией мы можем обновлять его при помощи верного значения в будущем. После одной итерации последнее значение Q верно, ведь это вознаграждение с последнего состояния и действия перед окончанием эпизода. Затем мы проводим обновление Q и устанавливаем тем самым его

значение для второй с конца пары (состояние, действие). На следующей итерации мы можем гарантировать, что верны два последних значения Q, и т. д. Благодаря итерации по значениям гарантируется схождение к конечному оптимальному значению Q.

ПРОБЛЕМЫ ИТЕРАЦИИ ПО ЦЕННОСТЯМ

Итерация по ценностям устанавливает связь между парами состояний и действий и значениями Q, и мы создаем таблицу этих связей, которая называется *Q-таблицей*.

Коротко поговорим о ее размере. Итерация по ценностям — утомительный процесс, который требует полного обхода всех пар (состояние, действие). Например, в игре `Breakout` 100 кирпичиков могут либо присутствовать, либо нет, а также есть 50 возможных положений ударной лопатки, 250 возможных позиций шарика и три действия — и уже здесь такой объем, который во много раз превосходит сумму всех вычислительных возможностей человечества. А в стохастических средах объем Q-таблицы будет еще больше — возможно, даже бесконечным. И тогда найти Q-значения всех пар (действие, состояние) станет невозможно. Этот подход явно не сработает. Как же тогда заниматься Q-обучением?

АППРОКСИМАЦИЯ Q-ФУНКЦИИ

Размер Q-таблицы делает наивный подход неосуществимым для любой реальной задачи. Но что если ослабить требования к оптимальной Q-функции? Если обучать аппроксимацию Q-функции, можно использовать модель для ее оценки.

Вместо того чтобы пытаться исследовать каждую пару (состояние, действие) ради обновления Q-таблицы, можно обучить функцию, которая будет аппроксимировать ее и даже строить обобщения за пределами своего опыта. И нам не придется вести утомительный поиск по всем возможным Q-значениям для обучения функции.

ГЛУБОКАЯ Q-СЕТЬ (DQN)

Этим руководствовались в DeepMind при работе над глубокой Q-сетью (`Deep Q-Network, DQN`). `DQN` берет глубокую нейронную сеть, которая на основе полученного изображения (состояния) оценивает Q-значение для всех возможных действий.

ОБУЧЕНИЕ DQN

Мы хотим обучить сеть аппроксимировать Q-функции. Выразим ее аппроксимацию как функцию параметров нашей модели:

$$\hat{Q}_\theta(s,a|\theta) \sim Q^*(s,a).$$

Помните, что Q-обучение — это обучение ценности. Мы осваиваем не саму стратегию, а ценность каждой пары (действие, состояние), независимо от их качества. Аппроксимацию Q-функции нашей модели мы выразили как `Qtheta`, и мы хотели бы, чтобы она была близка к ожидаемому вознаграждению. Используя уравнение Беллмана, рассмотренное выше, мы можем выразить его так:

$$R_t^* = \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta)\right).$$

Наша цель — минимизировать разницу между аппроксимацией Q и следующим значением Q:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \hat{Q}(s_t, a_t|\theta) - R_t^*.$$

Раскрытие этого выражения дает нам полную целевую функцию:

$$\min_\theta \sum_{e \in E} \sum_{t=0}^{T} \hat{Q}(s_t, a_t|\theta) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'|\theta)\right)$$

Она полностью дифференцируема как функция параметров нашей модели, и можно найти для нее градиенты для использования в стохастическом градиентном спуске и минимизации потерь.

СТАБИЛЬНОСТЬ ОБУЧЕНИЯ

Вы наверняка уже заметили проблему: мы определяем функцию потерь на основе разницы предсказанного Q-значения нашей модели для этого шага и для следующего. Получается, потери вдвойне зависят от параметров модели.

При каждом обновлении параметров Q-значения сдвигаются, а мы используем их для дальнейших обновлений. Высокая корреляция обновлений может привести к циклам обратной связи и нестабильности в обучении,

поскольку параметры порой значительно колеблются и функция потерь не сходится.

Чтобы устранить эту проблему корреляции, можно использовать пару простых инженерных хитростей: это целевая Q-сеть и воспроизведение опыта.

ЦЕЛЕВАЯ Q-СЕТЬ

Вместо постоянного обновления одной сети по отношению к самой себе можно снизить взаимозависимость, введя вторую, которая называется целевой. Наша функция потерь относится к случаям Q-функции, $\hat{Q}(s_t, a_t|\theta)$ и $\hat{Q}(s_{t+1}, a'|\theta)$.

Мы представим первое Q как предсказательную сеть, а второе будет выдаваться целевой Q-сетью. Последняя — копия предсказательной сети с задержкой обновления параметров.

Мы обновляем целевую Q-сеть в соответствии с предсказательной только через каждые несколько пакетов. Это дает необходимую стабильность Q-значениям, и теперь можно должным образом изучить хорошую Q-функцию.

ПОВТОРЕНИЕ ОПЫТА

Есть еще один источник досадной нестабильности в обучении: высокие корреляции последних действий. Если обучать DQN на пакетах из недавнего опыта, все пары (состояние, действие) будут взаимосвязаны. Это вредно, поскольку мы хотим, чтобы градиенты пакета представляли весь градиент; а если данные нерепрезентативны для распределения данных, пакетный градиент не будет точным приближением истинного.

Поэтому нам нужно разбить корреляцию данных в пакетах. Это можно осуществить при помощи *повторения опыта*. Мы сохраняем весь опыт агента в таблице, а чтобы создать пакет, проводим случайную выборку. Опыт хранится в таблице в виде кортежей (s_i, a_i, r_i, s_{i+1}). Из этих четырех значений можно вычислить функцию потерь, а с ней и градиент для оптимизации сети.

Таблица воспроизведения опыта больше похожа на очередь. Опыт, который агент получил на ранних стадиях обучения, может не отражать тот, с которым сталкивается уже обученный агент, так что полезно время от времени удалять очень старый опыт из таблицы.

ОТ Q-ФУНКЦИИ К СТРАТЕГИИ

Q-обучение — парадигма обучения ценностям, а не алгоритм освоения стратегии. Мы не обучаем прямо стратегию действия в среде. А можем ли мы разработать ее на основе данных Q-функции? Если мы нашли хорошую аппроксимацию, мы знаем ценность каждого действия для каждого состояния. И теперь можно тривиально выработать оптимальную стратегию: просмотреть Q-функцию на предмет всех действий в текущем состоянии, выбрать действие с максимальным значением Q, перейти в новое состояние и повторить то же. Если Q-функция оптимальна, выработанная на ее основе стратегия тоже будет оптимальной. И тогда мы можем выразить оптимальную стратегию так:

$$\pi\left(s;\theta\right)=\arg\max_{a'}\hat{Q}^{*}\left(s,a';\theta\right).$$

Можно также прибегнуть к техникам семплирования, о которых мы говорили выше, для выработки стохастической стратегии, которая порой отклоняется от рекомендаций Q-функции для варьирования соотношения исследования и использования.

DQN И МАРКОВСКОЕ ПРЕДПОЛОЖЕНИЕ

DQN — тоже марковский процесс принятия решений, который опирается на *марковское предположение*, что следующее состояние `s_i + 1` зависит только от текущего `s_i` и действия `a_i`, а не от какого-либо предыдущего состояния или действия. Это несправедливо во многих средах, где состояние игры не может быть отражено в едином кадре. Например, в пинг-понге скорость шарика (важный фактор успеха) не может быть получена по одному кадру. Марковское предположение делает моделирование принятия решений гораздо проще и надежнее, но часто ценой мощности модели.

РЕШЕНИЕ ПРОБЛЕМЫ МАРКОВСКОГО ПРЕДПОЛОЖЕНИЯ В DQN

DQN решает эту проблему обращением к *истории состояний*. Вместо обработки одного кадра как игрового состояния DQN считает текущим состоянием четыре последних кадра, что позволяет ей использовать информацию, зависимую от времени. Это своего рода инженерная уловка, и в конце главы мы рассмотрим более эффективные методы работы с последовательностью состояний.

ИГРА В BREAKOUT ПРИ ПОМОЩИ DQN

Сведем воедино всё, что мы узнали в этой главе, и займемся реализацией DQN для игры в Breakout. Начнем с определения нашего DQNAgent.

```
# DQNAgent
        class DQNAgent(object):
        def __init__(self, session, num_actions,
        learning_rate=1e-3, history_length=4,
        screen_height=84, screen_width=84,
        gamma=0.98):
        self.session = session
        self.num_actions = num_actions
        self.learning_rate = learning_rate
        self.history_length = history_length
        self.screen_height = screen_height
        self.screen_width = screen_width
        self.gamma = gamma
        self.build_prediction_network()
        self.build_target_network()
        self.build_training()
        def build_prediction_network(self):
        with tf.variable_scope('pred_network'):
        self.s_t = tf.placeholder('float32', shape=[
        None,
        self.history_length,
        self.screen_height,
        self.screen_width],
        name='state')
        self.conv_0 = slim.conv2d(self.s_t, 32, 8, 4,
        scope='conv_0')
        self.conv_1 = slim.conv2d(self.conv_0, 64, 4, 2,
        scope='conv_1')
        self.conv_2 = slim.conv2d(self.conv_1, 64, 3, 1,
        scope='conv_2')
        shape = self.conv_2.get_shape().as_list()
        self.flattened = tf.reshape(
        self.conv_2, [-1, shape[1]*shape[2]*shape[3]])
```

```
self.fc_0 = slim.fully_connected(self.flattened,
512, scope='fc_0')
self.q_t = slim.fully_connected(
self.fc_0, self.num_actions, activation_fn=None,
scope='q_values')
self.q_action = tf.argmax(self.q_t, dimension=1)
def build_target_network(self):
with tf.variable_scope(‘target_network'):
self.target_s_t = tf.placeholder(‘float32',
shape=[None, self.history_length,
self.screen_height, self.screen_width],
name='state')
self.target_conv_0 = slim.conv2d(
self.target_s_t, 32, 8, 4, scope='conv_0')
self.target_conv_1 = slim.conv2d(
self.target_conv_0, 64, 4, 2, scope='conv_1')
self.target_conv_2 = slim.conv2d(
self.target_conv_1, 64, 3, 1, scope='conv_2')
shape = self.conv_2.get_shape().as_list()
self.target_flattened = tf.reshape(
self.target_conv_2, [-1,
shape[1]*shape[2]*shape[3]])
self.target_fc_0 = slim.fully_connected(
self.target_flattened, 512, scope='fc_0')
self.target_q = slim.fully_connected(
self.target_fc_0, self.num_actions,
activation_fn=None, scope='q_values')
def update_target_q_weights(self):
pred_vars = tf.get_collection(
tf.GraphKeys.GLOBAL_VARIABLES, scope=
‘pred_network')
target_vars = tf.get_collection(
tf.GraphKeys.GLOBAL_VARIABLES, scope=
‘target_network')
for target_var, pred_var in zip(target_vars, pred_vars):
weight_input = tf.placeholder(‘float32',
name='weight')
```

```
target_var.assign(weight_input).eval(
{weight_input: pred_var.eval()})
def build_training(self):
self.target_q_t = tf.placeholder('float32', [None],
name='target_q_t')
self.action = tf.placeholder('int64', [None],
name='action')
action_one_hot = tf.one_hot(
self.action, self.num_actions, 1.0, 0.0,
name='action_one_hot')
q_of_action = tf.reduce_sum(
self.q_t * action_one_hot, reduction_indices=1,
name='q_of_action')
self.delta = tf.square((self.target_q_t − q_of_action))
self.loss = tf.reduce_mean(self.delta, name='loss')
self.optimizer = tf.train.AdamOptimizer(
learning_rate=self.learning_rate)
self.train_step = self.optimizer.minimize(self.loss)
def sample_action_from_distribution(self,
action_distribution, epsilon_percentage):
# Choose an action based on the action probability
# distribution (Выбираем действие на основе распределения
     вероятностей действия)
action = epsilon_greedy_action_annealed(
action_distribution, epsilon_percentage)
return action
def predict_action(self, state, epsilon_percentage):
action_distribution = self.session.run(
self.q_t, feed_dict={self.s_t: [state]})[0]
action = self.sample_action_from_distribution(
action_distribution, epsilon_percentage)
return action
def process_state_into_stacked_frames(self, frame,
past_frames, past_state=None):
full_state = np.zeros(
(self.history_length, self.screen_width,
self.screen_height))
```

```
if past_state is not None:
for i in range(len(past_state)-1):
full_state[i, :, :] = past_state[i+1,
:, :]
full_state[-1, :, :] = imresize(to_grayscale(frame),
(self.screen_width,
self.screen_height))
/255.0
else:
all_frames = past_frames + [frame]
for i, frame_f in enumerate(all_frames):
full_state[i, :, :] = imresize(
to_grayscale(frame_f), (self.screen_width,
self.screen_height))/255.0
full_state = full_state.astype('float32')
return full_state
```

Здесь происходит много интересного, так что разберем это подробнее.

СОЗДАНИЕ АРХИТЕКТУРЫ

Мы создали две `Q`-сети: предсказательную и целевую. Определение их архитектуры одинаково, поскольку это, по сути, одна сеть, но у второго варианта есть задержка в обновлении параметров. Поскольку мы учимся играть в `Breakout` по чистому пиксельному вводу, наше состояние игры — массив пикселей.

Пропускаем это изображение через три сверточных слоя, а затем через два полносвязных, получая на выходе `Q`-значения для каждого из потенциальных действий.

ЗАНЕСЕНИЕ КАДРОВ В СТЕК

Вы, возможно, заметили, что входные данные состояния имеют размер `[None, self.history_length, self.screen_height, self.screen_width]`. Для моделирования и включения чувствительных ко времени переменных состояния, таких как скорость, `DQN` использует не одно изображение, а группу последовательных, которая называется *историей*. Каждая картинка рассматривается как отдельный канал. Мы

создаем объединяемые изображения при помощи вспомогательной функции `process_state_into_stacked_frames(self, frame, past_frames, past_state=None)`.

ЗАДАНИЕ ОБУЧАЮЩИХ ОПЕРАЦИЙ

Наша функция потерь выводится из выражения цели, приведенного выше в этой главе:

$$\min_{\theta} \sum_{e \in E} \sum_{t=0}^{T} \hat{Q}(s_t, a_t | \theta) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a' | \theta)\right)$$

Мы хотим, чтобы предсказательная сеть равнялась целевой плюс выгода на текущем шаге. Это можно выразить в чистом коде `TensorFlow` в виде разницы между выводами предсказательной и целевой сетей. Этот градиент мы используем для обновления и обучения предсказательной сети с помощью `AdamOptimizer`.

ОБНОВЛЕНИЕ ЦЕЛЕВОЙ Q-СЕТИ

Чтобы гарантировать стабильную среду обучения, мы обновляем целевую `Q`-сеть через каждые четыре пакета. Правило обновления довольно простое: мы приравниваем ее веса к весам предсказательной сети. Для этого используется функция `update_target_q_network(self)`. Можно применить `tf.get_collection()` для получения переменных областей действия предсказательной и целевой сетей. Мы можем пройти в цикле по этим переменным и запустить операцию `tf.assign()` для приравнивания весов целевой `Q`-сети к весам предсказательной.

РЕАЛИЗАЦИЯ ПОВТОРЕНИЯ ОПЫТА

Мы уже говорили, что повторение опыта может помочь устранить корреляцию обновлений градиентных пакетов для повышения качества `Q`-обучения и выработанной на его основе стратегии. Рассмотрим кратко простой случай реализации повторения опыта. Задействуем метод `add_episode(self, episode)`, который берет весь эпизод (объект `EpisodeHistory`) и добавляет его в `ExperienceReplayTable`. Затем он контролирует переполнение таблицы и удаляет оттуда самые старые значения опыта.

Когда дело доходит до выборки из таблицы, можно обратиться к `sample_batch(self, batch_size)` для случайной компоновки пакета:

```
class ExperienceReplayTable(object):
      def __init__(self, table_size=5000):
      self.states = []
      self.actions = []
      self.rewards = []
      self.state_primes = []
      self.discounted_returns = []
      self.table_size = table_size
      def add_episode(self, episode):
      self.states += episode.states
      self.actions += episode.actions
      self.rewards += episode.rewards
      self.discounted_returns += episode.discounted_returns
      self.state_primes += episode.state_primes
      self.purge_old_experiences()
      def purge_old_experiences(self):
      if len(self.states) > self.table_size:
      self.states = self.states[-self.table_size:]
      self.actions = self.actions[-self.table_size:]
      self.rewards = self.rewards[-self.table_size:]
      self.discounted_returns = self.discounted_returns[
      -self.table_size:]
      self.state_primes = self.state_primes[
      -self.table_size:]
      def sample_batch(self, batch_size):
      s_t, action, reward, s_t_plus_1, terminal = [], [],
      [], [], []
      rands = np.arange(len(self.states))
      np.random.shuffle(rands)
      rands = rands[:batch_size]
      for r_i in rands:
      s_t.append(self.states[r_i])
      action.append(self.actions[r_i])
      reward.append(self.rewards[r_i])
```

```
        s_t_plus_1.append(self.state_primes[r_i])
        terminal.append(self.discounted_returns[r_i])
        return np.array(s_t), np.array(action),
        np.array(reward), np.array(s_t_plus_1),
        np.array(terminal)
```

ОСНОВНОЙ ЦИКЛ DQN

Сведем всё воедино в основной функции, которая создаст среду `OpenAI Gym` для `Breakout`, выведет образец `DQNAgent` и заставит его взаимодействовать со средой и успешно обучаться игре в `Breakout`:

```
def main(argv):
        # Configure Settings (Конфигурируем настройки)
        run_index = 0
        learn_start = 100
        scale = 10
        total_episodes = 500*scale
        epsilon_stop = 250*scale
        train_frequency = 4
        target_frequency = 16
        batch_size = 32
        max_episode_length = 1000
        render_start = total_episodes – 10
        should_render = True
        env = gym.make('Breakout-v0')
        num_actions = env.action_space.n
        solved = False
        with tf.Session() as session:
        agent = DQNAgent(session=session,
        um_actions=num_actions)
        session.run(tf.global_variables_initializer())
        episode_rewards = []
        batch_losses = []
        replay_table = ExperienceReplayTable()
        global_step_counter = 0
        for i in tqdm.tqdm(range(total_episodes)):
```

```
frame = env.reset()
past_frames = [frame] * (agent.history_length-1)
state = agent.process_state_into_stacked_frames(
frame, past_frames, past_state=None)
episode_reward = 0.0
episode_history = EpisodeHistory()
epsilon_percentage = float(min(i/float(
epsilon_stop), 1.0))
for j in range(max_episode_length):
action = agent.predict_action(state,
epsilon_percentage)
if global_step_counter < learn_start:
action = random_action(agent.num_actions)
# print(action)
frame_prime, reward, terminal, _ = env.step(
action)
state_prime =
agent.process_state_into_stacked_frames(
frame_prime, past_frames,
past_state=state)
past_frames.append(frame_prime)
past_frames = past_frames[-4:]
if (render_start > 0 and (i >
render_start)
and should_render) or (solved and
should_render):
env.render()
episode_history.add_to_history(
state, action, reward, state_prime)
state = state_prime
episode_reward += reward
global_step_counter += 1
if j == (max_episode_length – 1):
terminal = True
if terminal:
episode_history.discounted_returns =
discount_rewards(
```

```
episode_history.rewards)
replay_table.add_episode(episode_history)
if global_step_counter > learn_start:
if global_step_counter %
train_frequency == 0:
s_t, action, reward, s_t_plus_1,
terminal = \
replay_table.sample_batch(
batch_size)
q_t_plus_1 = agent.target_q.eval(
{agent.target_s_t:
s_t_plus_1})
terminal = np.array(terminal) + 0.
max_q_t_plus_1 = np.max(q_t_plus_1,
axis=1)
target_q_t = (1. — terminal) * \
agent.gamma * max_q_t_plus_1 +
reward
_, q_t, loss = agent.session.run(
[agent.train_step, agent.q_t,
agent.loss], {
agent.target_q_t: target_q_t,
agent.action: action,
agent.s_t: s_t
})
if global_step_counter %
target_frequency == 0:
agent.update_target_q_weights()
episode_rewards.append(episode_reward)
break
if i % 50 == 0:
ave_reward = np.mean(episode_rewards[-100:])
print(ave_reward)
if ave_reward > 50.0:
solved = False
else:
solved = False
```

РЕЗУЛЬТАТЫ DQNAGENT В BREAKOUT

Мы обучаем `DQNAgent` на 1000 эпизодах и рассматриваем кривую обучения. Чтобы достичь на Atari результатов выше, чем у человека, обычно время обучения растягивается на несколько дней. Но общая восходящая тенденция в вознаграждениях проявляется быстро, что показано на рис. 9.7.

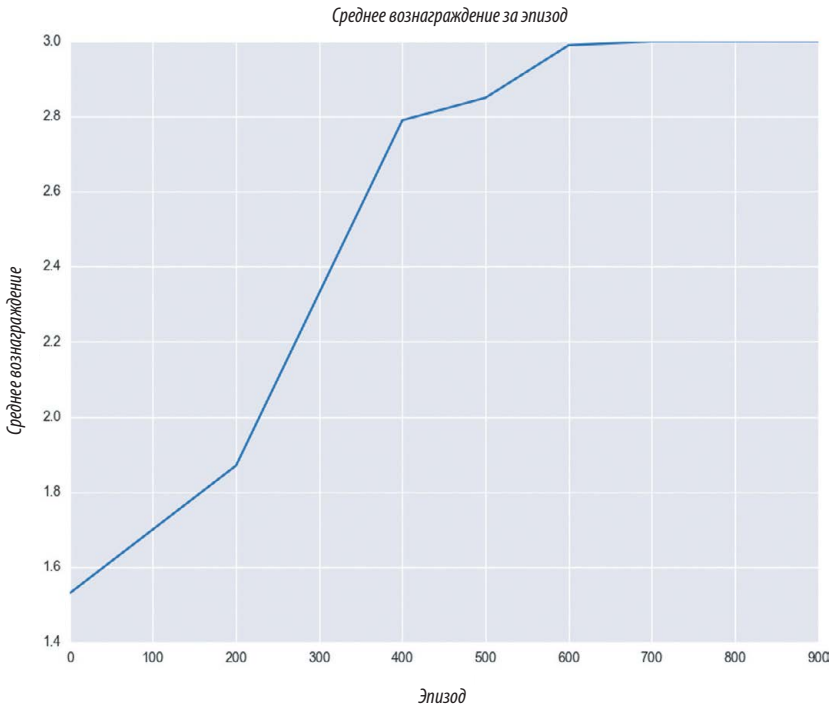


Рис. 9.7. *Агент DQN показывает все более успешные результаты в Breakout во время обучения, усвоив хорошую функцию-значение, и действует все менее случайно благодаря ϵ-жадной нормализации*

Улучшение и выход за пределы DQN

В 2013 году DQN прекрасно поработала над решением задач Atari, но имела ряд серьезных недостатков. Среди основных слабостей — слишком большое время обучения, неудачные результаты в определенных типах игр и необходимость повторного обучения для каждой новой задачи. Большинство исследований в области глубокого обучения с подкреплением в последние несколько лет было сосредоточено на устранении этих недостатков.

ГЛУБОКИЕ РЕКУРРЕНТНЫЕ Q-СЕТИ (DRQN)

Помните предположение Маркова — что следующее состояние зависит только от предыдущего и действия агента? Решение Маркова при помощи DQN, при котором четыре последовательных кадра помещаются в стек как отдельный канал, просто обходит проблему и во многом напоминает инженерную уловку. Почему кадров четыре, а не десять? Этот искусственный гиперпараметр ограничивает обобщающую способность модели. Как работать с произвольными последовательностями взаимосвязанных данных? Можно воспользоваться тем, что мы узнали в главе 6 о рекуррентных нейронных сетях, для моделирования последовательностей при помощи *глубоких рекуррентных Q-сетей* (DRQN).

В DRQN рекуррентный слой используется для передачи латентного знания о состоянии с одного временного шага на другой. Благодаря этому модель может обучиться тому, сколько кадров обладает информацией, которую нужно включить в состояние, и даже тому, как выкидывать неинформативные кадры или запоминать информативные старые.

DRQN даже расширили, включив нейронный механизм внимания, о чем говорится в работе Ивана Сорокина и коллег «Глубокие рекуррентные Q-сети с вниманием; DAQRN» 2015 года[95]. Поскольку DRQN работает с последовательностью данных, она может уделять внимание конкретным ее фрагментам. Такая способность концентрироваться на деталях и повышает эффективность, и обеспечивает интерпретируемость модели, создавая логическое обоснование для предпринимаемых действий.

DRQN оказалась лучше, чем DQN, в играх-«стрелялках» от первого лица, например DOOM[96], а также продемонстрировала лучшие результаты в некоторых играх Atari с длинными временными зависимостями, например Seaquest[97].

ПРОДВИНУТЫЙ АСИНХРОННЫЙ АГЕНТ-КРИТИК (A3C)

Продвинутый асинхронный агент-критик (Asynchronous advantage actor-critic, A3C) — новый подход к глубокому обучению с подкреплением, представленный в DeepMind 2016 года «Асинхронные методы глубокого обучения с подкреплением»[98]. Поговорим о том, что это и в чем его преимущества перед DQN.

Подход A3C *асинхронный*, то есть можно распараллелить агент по нескольким потокам, что значительно сократит время на обучение, поскольку ускоряется симуляция среды. A3C запускает одновременно несколько сред для сбора опыта. Помимо повышения скорости, этот подход дает еще

одно значительное преимущество: он еще больше снимает взаимозависимость опыта в пакетах, поскольку пакет одновременно заполняется опытом многих агентов, действующих по разным сценариям.

A3C использует метод *агента-критика*[99]. Он заключается в обучении как функции ценностей $V(s_t)$ — критик, так и стратегии $\pi(s_t)$ — агент. В этой главе мы выделили два разных подхода: обучение ценности и обучение стратегии. В A3C сочетаются сильные стороны обоих: функция ценностей критика улучшает стратегию агента.

A3C использует функцию преимущества вместо чистой дисконтированной будущей выгоды. При обучении стратегии мы хотим, чтобы агент получал штраф, выбирая действие, которое ведет к плохому подкреплению. A3C стремится к тому же, но критерием считает не вознаграждение, а преимущество, то есть разницу между предсказанным моделью и реальным качеством совершенного действия. Преимущество можно выразить так:

$$A_t = Q^*(s_t, a_t) - V(s_t).$$

У A3C есть функция ценности, `V(t)`, но она не выражает `Q`-функцию. Она оценивает преимущество, используя дисконтирование будущей выгоды как приближение `Q`-функции:

$$A_t = R_t - V(s_t).$$

Три этих метода, как оказалось, обеспечивают A3C преимущество перед большинством аналогов в сфере глубокого обучения с подкреплением. Агенты A3C могут научиться играть в Atari Breakout меньше чем за 12 часов, а агентам DQN на это может потребоваться три-четыре дня.

ПОДКРЕПЛЕНИЕ БЕЗ УЧИТЕЛЯ И ВСПОМОГАТЕЛЬНОЕ ОБУЧЕНИЕ (UNSUPERVISED REINFORCEMENT AND AUXILIARY LEARNING, UNREAL)

`UNREAL` — улучшение A3C, представленное в работе Макса Ядерберга и коллег «Обучение с подкреплением со вспомогательными заданиями без учителя»[100]. Эти авторы, как вы, наверное, уже догадались, тоже из DeepMind.

UNREAL решает проблему недостаточности вознаграждения. Обучение с подкреплением так сложно, поскольку агент просто получает вознаграждения, а определить, почему именно они увеличиваются или уменьшаются, сложно. Кроме того, мы должны обучить модель и хорошему представлению мира, и хорошей стратегии — только это обеспечит вознаграждение.

Если же обратная связь окажется слабой, как в случае с недостаточными вознаграждениями, это будет особенно сложно.

UNREAL задается вопросом о том, что можно освоить без вознаграждений, и ставит себе целью обучиться полезному представлению мира без учителя. Для этого оно добавляет несколько вспомогательных задач без учителя к общей цели.

Первое задание связано с обучением агента тому, как его действия влияют на среду. Он получает задачу контролировать значения пикселов на экране. Чтобы выработать набор значений в следующем кадре, агент должен выполнить определенное действие в текущем. Так он узнает, как его действия влияют на окружающий мир. Это помогает научиться представлению мира, которое учитывает и его действия.

Второе задание связано с обучением агента UNREAL *предсказанию вознаграждения*. Он получает последовательность состояний и задачу предсказать значение следующего вознаграждения. Если агент способен верно назвать его, то, возможно, у него уже есть хорошая модель будущего состояния окружающей среды, что будет полезно при выработке стратегии.

После выполнения этих вспомогательных задач без учителя UNREAL оказывается способен в 10 раз быстрее, чем A3C, обучаться в среде игры Labyrynth. Для UNREAL особенно важно обучение хорошим представлениям мира и тому, как освоение навыков без учителя может помочь в условиях слабой обратной связи или при решении проблем обучения с низкими ресурсами, например в модели с подкреплением.

Резюме

В этой главе мы поговорили об основах обучения с подкреплением, включая марковские процессы принятия решений, максимальное дисконтирование будущих вознаграждений и соотношение исследования и использования. Также мы рассказали о подходах к глубокому обучению с подкреплением, в том числе градиентах по стратегиям и глубоких Q-сетях, и осветили последние улучшения DQN и новые разработки в сфере глубокого обучения с подкреплением.

Обучение с подкреплением необходимо для создания агентов, которые могут не только воспринимать и интерпретировать мир, но и предпринимать действия и взаимодействовать с ним. Глубокое обучение с подкреплением уже сделало большие шаги к этой цели, создав успешных агентов, которые умеют играть в игры Atari, безопасно водят автомобили, выгодно торгуют на бирже, управляют роботами и способны на многое другое.

Примечания

Глава 1

1. Kuhn D. et al. Handbook of Child Psychology. Vol. 2. Cognition, Perception, and Language. Wiley, 1998.
2. LeCun Y., Bottou L., Bengio Y., Haffner P. Gradient-Based Learning Applied to Document Recognition // Proceedings of the IEEE. 1998. November. Vol. 86 (11). Pp. 2278–2324.
3. Rosenblatt F. The perceptron: A probabilistic model for information storage and organization in the brain // Psychological Review. 1958. Vol. 65. No. 6. P. 386.
4. Bubeck S. Convex optimization: Algorithms and complexity // Foundations and Trends® in Machine Learning. 2015. Vol. 8. No. 3–4. Pp. 231–357.
5. Restak R. M., Grubin D. The Secret Life of the Brain. Joseph Henry Press, 2001.
6. McCulloch W. S., Pitts W. A logical calculus of the ideas immanent in nervous activity // The Bulletin of Mathematical Biophysics. 1943. Vol. 5. No. 4. Pp. 115–133.
7. Mountcastle V. B. Modality and topographic properties of single neurons of cat's somatic sensory cortex // Journal of Neurophysiology. 1957. Vol. 20. No. 4. Pp. 408–434.
8. Nair V., Hinton G. E. Rectified Linear Units Improve Restricted Boltzmann Machines // Proceedings of the 27th International Conference on Machine Learning (ICML-10), 2010.

Глава 2

9. Rosenbloom P. The method of steepest descent // Proceedings of Symposia in Applied Mathematics. 1956. Vol. 6.
10. Rumelhart D. E., Hinton G. E., Williams R. J. Learning representations by backpropagating errors // Cognitive Modeling. 1988. Vol. 5. No. 3. P. 1.
11. http://stanford.io/2pOdNhy.
12. Nelder J. A., Mead R. A simplex method for function minimization // The Computer Journal. 1965. Vol. 7. No. 4. Pp. 308–313.
13. Tikhonov A. N., Glasko V. B. Use of the regularization method in nonlinear problems // USSR Computational Mathematics and Mathematical Physics. 1965. Vol. 5. No. 3. Pp. 93–107.
14. Srebro N., Rennie J. D. M., Jaakkola T. S. Maximum-Margin Matrix Factorization // NIPS. 2004. Vol. 17.
15. Srivastava N. et al. Dropout: A Simple Way to Prevent Neural Networks from Overfitting // Journal of Machine Learning Research. 2014. Vol. 15. No. 1. Pp. 1929–1958.

Глава 3

16. https://www.tensorflow.org/.
17. http://deeplearning.net/software/theano/ (http://bit.ly/2jtjGea); http://torch.ch/; http://caffe.berkeleyvision.org/; https://www.nervanasys.com/technology/neon/ (http://bit.ly/2r9XugB); https://keras.io/.
18. https://www.tensorflow.org/install/.
19. https://www.tensorflow.org/api_docs/python/tf/Variable.
20. https://www.tensorflow.org/api_docs/python/tf/random_normal.
21. https://www.tensorflow.org/api_docs/python/tf/assign.
22. http://bit.ly/2rtqoIA.
23. https://www.tensorflow.org/api_docs/python/tf/initialize_variables.

24. Abadi M. et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems // arXiv preprint arXiv: 1603.04467 (2016).
25. https://www.tensorflow.org/api_docs/python/tf/placeholder.
26. https://www.tensorflow.org/api_docs/python/tf/Session.
27. https://www.tensorflow.org/api_docs/python/tf/get_variable.
28. https://www.tensorflow.org/api_docs/python/tf/variable_scope.
29. https://www.tensorflow.org/api_docs/python/tf/device.
30. https://www.tensorflow.org/api_docs/python/tf/ConfigProto.
31. Cox D. R. The Regression Analysis of Binary Sequences // Journal of the Royal Statistical Society. Series B (Methodological). 1958. Pp. 215–242.
32. https://www.tensorflow.org/api_docs/python/tf/summary/scalar.
33. https://www.tensorflow.org/api_docs/python/tf/summary/histogram.
34. https://www.tensorflow.org/api_docs/python/tf/summary/merge_all.
35. https://www.tensorflow.org/get_started/graph_viz.
36. He K. et al. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification // Proceedings of the IEEE International Conference on Computer Vision. 2015.

Глава 4

37. Bengio Y. et al. Greedy Layer-Wise Training of Deep Networks // Advances in Neural Information Processing Systems. 2007. Vol. 19. P. 153.
38. Goodfellow I. J., Vinyals O., Saxe A. M. Qualitatively characterizing neural network optimization problems // arXiv preprint arXiv: 1412.6544 (2014).
39. Dauphin Y. N. et al. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization // Advances in Neural Information Processing Systems. 2014.
40. Sutskever I. et al. On the importance of initialization and momentum in deep learning // ICML (3). 2013. Vol. 28. Pp. 1139–1147.
41. Møller M. F. A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning // Neural Networks. 1993. Vol. 6. No. 4. Pp. 525–533.
42. Broyden C. G. A new method of solving nonlinear simultaneous equations // The Computer Journal. 1969. Vol. 12. No. 1. Pp. 94–99.
43. Bonnans J.-F. et al. Numerical Optimization: Theoretical and Practical Aspects. Springer Science & Business Media, 2006.
44. Duchi J., Hazan E., Singer Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization // Journal of Machine Learning Research. 2011. Vol. 12 (Jul.). Pp. 2121–2159.
45. Tieleman T., Hinton G. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude // COURSERA: Neural Networks for Machine Learning. 2012. Vol. 4. No. 2.
46. Kingma D., Ba J. Adam: A Method for Stochastic Optimization // arXiv preprint arXiv: 1412.6980 (2014).

Глава 5

47. Hubel D. H., Wiesel T. N. Receptive fields and functional architecture of monkey striate cortex // The Journal of Physiology. 1968. Vol. 195. No. 1. Pp. 215–243.
48. Cohen A. I. Rods and Cones // Physiology of Photoreceptor Organs. Springer Berlin Heidelberg, 1972. Pp. 63–110.
49. Viola P., Jones M. Rapid Object Detection using a Boosted Cascade of Simple Features // Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Vol. 1. IEEE, 2001.
50. Deng J. et al. ImageNet: A Large-Scale Hierarchical Image Database // Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference. IEEE, 2009.

51. Perronnin F., Sénchez J., Xerox Y. L. Large-scale image categorization with explicit data embedding // Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference. IEEE, 2010.
52. Krizhevsky A., Sutskever I., Hinton G. E. ImageNet Classification with Deep Convolutional Neural Networks // Advances in Neural Information Processing Systems. 2012.
53. LeCun Y. et al. Handwritten Digit Recognition with a Back-Propagation Network // Advances in Neural Information Processing Systems. 1990.
54. Hubel D. H., Wiesel T. N. Receptive fields of single neurones in the cat's striate cortex // The Journal of Physiology. 1959. Vol. 148. No. 3. Pp. 574–591.
55. https://www.tensorflow.org/api_docs/python/tf/nn/conv2d.
56. https://www.tensorflow.org/api_docs/python/tf/nn/max_pool.
57. Graham B. Fractional Max-Pooling // arXiv Preprint arXiv: 1412.6071 (2014).
58. Simonyan K., Zisserman A. Very Deep Convolutional Networks for Large-Scale Image Recognition // arXiv Preprint arXiv: 1409.1556 (2014).
59. Ioffe S., Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift // arXiv Preprint arXiv: 1502.03167. 2015.
60. Krizhevsky A., Hinton G. Learning Multiple Layers of Features from Tiny Images. 2009.
61. Maaten L. van der, Hinton G. Visualizing Data using t-SNE // Journal of Machine Learning Research. 2008. Vol. 9 (Nov.). Pp. 2579–2605.
62. http://cs.stanford.edu/people/karpathy/cnnembed/.
63. Gatys L. A., Ecker A. S., Bethge M. A Neural Algorithm of Artistic Style // arXiv Preprint arXiv: 1508.06576 (2015).
64. Karpathy A. et al. Large-scale Video Classification with Convolutional Neural Networks // Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014.
65. Abdel-Hamid O. et al. Applying Convolutional Neural Networks concepts to hybrid NN-HMM model for speech recognition // IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). Kyoto, 2012. Pp. 4277–4280.

Глава 6

66. Hinton G. E., Salakhutdinov R. R. Reducing the Dimensionality of Data with Neural Networks // Science. 2006. Vol. 313. No. 5786. Pp. 504–507.
67. Vincent P. et al. Extracting and Composing Robust Features with Denoising Autoencoders // Proceedings of the 25th International Conference on Machine Learning. ACM, 2008.
68. Bengio Y. et al. Generalized Denoising Auto-Encoders as Generative Models // Advances in Neural Information Processing Systems. 2013.
69. Ranzato M. et al. Efficient Learning of Sparse Representations with an Energy-Based Model // Proceedings of the 19th International Conference on Neural Information Processing Systems. MIT Press, 2006; Ranzato M., Szummer M. Semi-supervised Learning of Compact Document Representations with Deep Networks // Proceedings of the 25th International Conference on Machine Learning. ACM, 2008.
70. Makhzani A., Frey B. k-Sparse Autoencoders // arXiv preprint arXiv: 1312.5663 (2013).
71. Mikolov T. et al. Distributed Representations of Words and Phrases and their Compositionality // Advances in Neural Information Processing Systems. 2013.
72. Mikolov T., Chen K., Corrado G., Dean J. Efficient Estimation of Word Representations in Vector Space // ICLR Workshop, 2013.
73. https://www.tensorflow.org/api_docs/python/tf/nn/embedding_lookup.

Глава 7

74. Google News: https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit.
75. http://leveldb.org/.
76. http://www.cnts.ua.ac.be/conll2000/chunking/.

77. Nivre J. Incrementality in Deterministic Dependency Parsing // Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together. Association for Computational Linguistics, 2004.
78. Chen D., Manning C. D. A Fast and Accurate Dependency Parser Using Neural Networks // EMNLP. 2014.
79. https://github.com/tensorflow/models/tree/master/syntaxnet.
80. Andor D. et al. Globally Normalized Transition-Based Neural Networks // arXiv preprint arXiv: 1603.06042 (2016).
81. Там же.
82. Kilian J., Siegelmann H. T. The dynamic universality of sigmoidal neural networks // Information and computation. 1996. Vol. 128. No. 1. Pp. 48–56.
83. Kiros R. et al. Skip-Thought Vectors // Advances in neural information processing systems. 2015.
84. Bahdanau D., Cho K., Bengio Y. Neural Machine Translation by Jointly Learning to Align and Translate // arXiv preprint arXiv:1409.0473 (2014).
85. Этот код можно найти здесь: https://github.com/tensorflow/tensorflow/tree/r0.7/tensorflow/models/rnn/translate.

Глава 8

86. https://mostafa-samir.github.io/.
87. Graves A., Wayne G., Denihelka I. Neural Turing Machines // Cornell University, 2014 // https://arxiv.org/abs/1410.5401.
88. Там же.
89. Graves A., Wayne G., Reynolds M. et al. Hybrid computing using a neural network with dynamic external memory // Nature, 2016 // http://go.nature.com/2peM8m2.
90. https://github.com/Mostafa-Samir/DNC-tensorflow.

Глава 9

91. http://nicklocascio.com/.
92. Mnih V. et al. Human-level control through deep reinforcement learning // Nature. 2015. Vol. 518. No. 7540. Pp. 529–533.
93. Brockman G. et al. OpenAI Gym // arXiv preprint arXiv:1606.01540 (2016) // https://gym.openai.com//
94. Sutton R. S. et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation // NIPS. 1999. Vol. 99.
95. Sorokin I. et al. Deep Attention Recurrent Q-Network // arXiv preprint arXiv:1512.01693 (2015).
96. https://en.wikipedia.org/wiki/Doom_(1993_video_game).
97. https://en.wikipedia.org/wiki/Seaquest_(video_game).
98. Mnih V. et al. Asynchronous methods for deep reinforcement learning // International Conference on Machine Learning. 2016.
99. Konda V. R., Tsitsiklis J. N. Actor-Critic Algorithms // NIPS. 1999. Vol. 13.
100. Jaderberg M. et al. Reinforcement Learning with Unsupervised Auxiliary Tasks // arXiv preprint arXiv: 1611.05397 (2016).

Благодарности

Благодарим тех, кто помогал нам в работе над книгой. В первую очередь спасибо Мостафе Самиру и Сурье Бхупатираджу, которые внесли значительный вклад в главы 7 и 8. Мы очень признательны Мохамеду (Хассану) Кане и Анише Аталье, которые создавали первые варианты образцов кода в репозитории Github для этой книги.

Книга не состоялась бы без постоянной поддержки и опыта нашего издателя Шеннона Катта. Мы признательны за комментарии рецензентам — Айзеку Хоудзу, Дэвиду Анджеевски и Аарону Шумахеру, которые дали нам ценные и глубокие замечания еще на этапе черновиков. Наконец, мы благодарим за поддержку и советы во время работы над чистовиком всех наших друзей и членов семьи: Джеффа Дина, Нитина Будуму, Венката Будуму, а также Уильяма и Джека.

Несколько слов об обложке

Животное на обложке «Основ глубокого обучения» — рыба-единорог (Lophotus capellei). Она относится к семейству лофотовых и живет в глубоких водах Атлантического и Тихого океанов. Рыбы скрываются от исследователей, и о них мало что известно. Но некоторые из пойманных экземпляров достигали в длину почти двух метров.

Многие животные на обложках издательства O'Reilly относятся к видам, находящимся под угрозой; все они важны для мира. Узнать, как им помочь, можно на animals.oreilly.com. Изображение на обложке выполнено Карен Монтгомери на основе черно-белой гравюры из книги Ричарда Лидеккера Royal Natural History.

Об авторе

Нихиль Будума — один из основателей и главный научный сотрудник Remedy, компании из Сан-Франциско, которая создает новую систему управляемой данными первичной медицинской помощи. Уже в 16 лет он руководил лабораторией по созданию новых лекарственных средств в Университете Сан-Хосе и разрабатывал новые недорогие методы обследования для районов с ограниченными ресурсами. К 19 годам он имел уже две золотые медали Международной олимпиады по биологии. Затем он учился в Массачусетском технологическом университете, где занимался разработкой масштабных систем данных для оказания медицинской помощи, поддержания психического здоровья и медицинских разработок. В MIT он основал Lean On Me — национальную некоммерческую организацию, предоставляющую анонимную текстовую горячую линию в кампусах колледжей и использующую данные для поддержания психического и физического здоровья. Сейчас Нихиль в свободное время инвестирует в компании в сфере материальных технологий и данных в рамках своего венчурного фонда Q Venture Partners и руководя командой анализа данных бейсбольной команды Milwaukee Brewers.

Где купить наши книги

Специальное предложение для компаний

Если вы хотите купить сразу более 20 книг, например для своих сотрудников или в подарок партнерам, мы готовы обсудить с вами специальные условия работы. Для этого обращайтесь к нашему менеджеру по корпоративным продажам: +7 (495) 792-43-72, b2b@mann-ivanov-ferber.ru

Книготорговым организациям

Если вы оптовый покупатель, обратитесь, пожалуйста, к нашему партнеру — торговому дому «Эксмо», который осуществляет поставки во все книготорговые организации.

142701, Московская обл., г. Видное, Белокаменное ш., д. 1; +7 (495) 411-50-74; reception@eksmo-sale.ru

Адрес издательства «Эксмо» 125252, Москва, ул. Зорге, д. 1; +7 (495) 411-68-86; info@eksmo.ru /www.eksmo.ru

Санкт-Петербург
СЗКО Санкт-Петербург, 192029, г. Санкт-Петербург, пр-т Обуховской Обороны, д. 84е; +7 (812) 365-46-03 / 04; server@szko.ru

Нижний Новгород
Филиал «Эксмо» в Нижнем Новгороде, 603094, г. Нижний Новгород, ул. Карпинского, д. 29; +7 (831) 216-15-91, 216-15-92, 216-15-93, 216-15-94; reception@eksmonn.ru

Ростов-на-Дону
Филиал «Эксмо» в Ростове-на-Дону, 344023, г. Ростов-на-Дону, ул. Страны Советов, 44а; +7 (863) 303-62-10; info@rnd.eksmo.ru

Самара
Филиал «Эксмо» в Самаре, 443052, г. Самара, пр-т Кирова, д. 75/1, лит. «Е»; +7 (846) 269-66-70 (71…73); RDC-samara@mail.ru

Екатеринбург
Филиал «Эксмо» в Екатеринбурге, 620024, г. Екатеринбург, ул. Новинская, д. 2щ; +7 (343) 272-72-01 (02…08)

Новосибирск
Филиал «Эксмо» в Новосибирске, 630015, г. Новосибирск, Комбинатский пер., д. 3; +7 (383) 289-91-42; eksmo-nsk@yandex.ru

Хабаровск
Филиал «Эксмо» Новосибирск в Хабаровске, 680000, г. Хабаровск, пер. Дзержинского, д. 24, лит. «Б», оф. 1; +7 (4212) 910-120; eksmo-khv@mail.ru

Казахстан
«РДЦ Алматы», 050039, г. Алматы, ул. Домбровского, д. 3а; +7 (727) 251-59-89 (90, 91, 92); RDC-almaty@eksmo.kz

Украина
«Эксмо-Украина», Киев, ООО «Форс Украина», 04073, г. Киев, Московский пр-т, д. 9; +38 (044) 290-99-44; sales@forsukraine.com

Если у вас есть замечания и комментарии к содержанию, переводу, редактуре и корректуре, то просим написать на be_better@m-i-f.ru, так мы быстрее сможем исправить недочеты.

ПРОДАЖИ

МЕНЕДЖМЕНТ

ИСТОРИИ УСПЕХА

УПРАВЛЕНИЕ ПРОЕКТАМИ

ПЕРЕГОВОРЫ

HR

Научно-популярное издание

Нихиль **Будума**
при участии Николаса Локашо

ОСНОВЫ ГЛУБОКОГО ОБУЧЕНИЯ

Создание алгоритмов для искусственного интеллекта следующего поколения

Руководитель редакции *Артем Степанов*
Шеф-редактор *Ренат Шагабутдинов*
Ответственный редактор *Татьяна Рапопорт*
Научный редактор *Андрей Созыкин*
Литературный редактор *Ольга Свитова*
Арт-директор *Алексей Богомолов*
Верстка обложки *Наталия Майкова*
Верстка *Екатерина Матусовская*
Корректоры *Лев Зелексон, Олег Пономарев*

ООО «Манн, Иванов и Фербер»
www.mann-ivanov-ferber.ru
www.facebook.com/mifbooks
www.vk.com/mifbooks
www.instagram.com/mifbooks

O'REILLY®

Десятилетиями мы мечтаем о создании разумных машин с таким же мозгом, как у нас: роботов-помощников для уборки в доме; машин, которые управляли бы собой сами; микроскопов, автоматически выявляющих болезни. Но чтобы это стало реальностью, необходимо научить машины решать сложнейшие вычислительные задачи, а значит, разработать новый способ программирования компьютеров. Эта очень активная отрасль в исследованиях искусственного интеллекта получила название «глубокое обучение».

Глубокое обучение — это раздел машинного обучения, изучающий глубокие нейронные сети и выстраивающий процесс получения знаний на основе примеров. Компьютеру не задают огромный список правил решения задачи, а предоставляют модель, с помощью которой он может сравнивать примеры, а также краткий набор инструкций для ее модификации в случае ошибки.

Такие крупные компании, как Google, Microsoft и Facebook, уделяют большое внимание глубокому обучению и расширяют свои подразделения в этой сфере. Для всех прочих глубокое обучение пока остается сложным, многогранным и малопонятным предметом.

Цель этой книги — заполнить этот пробел. Авторы разбирают основные принципы решения задач в глубоком обучении, исторический контекст современных подходов к нему и способы внедрения его алгоритмов. Если вы интересуетесь или занимаетесь глубоким обучением, то эта книга для вас.

ISBN 978-5-00146-472-3

9 785001 464723 >

Максимально
полезные книги на сайте
mann-ivanov-ferber.ru

издательство
МАНН, ИВАНОВ И ФЕРБЕР

Like facebook.com/**mifbooks**

vk.com/**mifbooks**

instagram.com/**mifbooks**