O'REILLY®

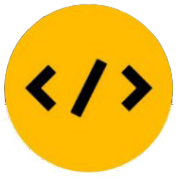
Глубокое обучение

Легкая разработка проектов на Python

Сет Вейдман

Deep Learning from Scratch

Building with Python from First Principles



Seth Weidman

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Глубокое обучение

Легкая разработка проектов на Python

Сет Вейдман

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2021

Сет Вейдман

Глубокое обучение: легкая разработка проектов на Python

Серия «Бестселлеры O'Reilly»

Перевели с английского И. Рузмайкина, А. Павлов

Руководитель проекта	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Обложка	*В. Мостипан*
Корректоры	*М. Молчанова, М. Одинокова*
Верстка	*Е. Неволайнен*

ББК 32.973.2-018.1
УДК 004.43

Вейдман Сет

В26 Глубокое обучение: легкая разработка проектов на Python. — СПб.: Питер, 2021. — 272 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1675-1

Взрывной интерес к нейронным сетям и искусственному интеллекту затронул уже все области жизни, и понимание принципов глубокого обучения необходимо каждому разработчику ПО для решения прикладных задач. Эта практическая книга представляет собой вводный курс для всех, кто занимается обработкой данных, а также для разработчиков ПО. Вы начнете с основ глубокого обучения и быстро перейдете к более сложным архитектурам, создавая проекты с нуля. Вы научитесь использовать многослойные, сверточные и рекуррентные нейронные сети. Только понимая принцип их работы (от «математики» до концепций), вы сделаете свои проекты успешными. В этой книге:

- Четкие схемы, помогающие разобраться в нейросетях, и примеры рабочего кода.
- Методы реализации многослойных сетей с нуля на базе простой объектно-ориентированной структуры.
- Примеры и доступные объяснения сверточных и рекуррентных нейронных сетей.
- Реализация концепций нейросетей с помощью популярного фреймворка PyTorch.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1492041412 англ.

ISBN 978-5-4461-1675-1

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 02.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 22.01.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 500. Заказ

Оглавление

Предисловие

Если вы уже пытались узнать что-то о нейронных сетях и глубоком обучении, то, скорее всего, столкнулись с изобилием ресурсов, от блогов до массовых открытых онлайн-курсов различного качества и даже книг. У меня было именно так, когда я начал изучать эту тему несколько лет назад. Однако если вы читаете это предисловие, вполне вероятно, что вы нигде не нашли достаточно полноценного описания нейронных сетей. Все эти ресурсы напоминают попытку нескольких слепцов описать слона (https://oreil.ly/r5YxS). Вот что привело меня к написанию этой книги.

Ресурсы по нейронным сетям обычно делятся на две категории. Некоторые из них касаются в основном концептуальной и математической части и содержат как рисунки, которые, как правило, встречаются в объяснениях нейронных сетей, так и круги, соединенные линиями со стрелками на концах, а также подробные математические объяснения того, что происходит, чтобы вы могли «вникнуть в матчасть». Пример этого — очень хорошая книга Яна Гудфеллоу и др. «Deep Learning»[1].

На других ресурсах — много кода, запустив который вы видите, как снижается ошибка и «обучается» нейронная сеть. Например, следующий пример из документации PyTorch действительно задает и обучает простую нейронную сеть случайными данными:

```
# N - размер партии; D_in - входной размер;
# H - скрытое измерение; D_out - размер вывода.
N, D_in, H, D_out = 64, 1000, 100, 10

# Создать случайные входные и выходные данные
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Произвольно инициализировать веса
w1 = torch.randn(D_in, H, device=device, dtype=dtype)
w2 = torch.randn(H, D_out, device=device, dtype=dtype)
```

[1] *Гудфеллоу Я., Бенджио И., Курвилль А.* Глубокое обучение / пер. с англ. А. А. Слинкина. — 2-е изд., испр. — М.: ДМК Пресс, 2018. — 652 с. — *Примеч. ред.*

```
learning_rate = 1e-6
for t in range(500):
    # Прямой проход: вычислить прогнозируемое y
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    # Вычислить и вывести потери
    loss = (y_pred – y).pow(2).sum().item()
    print(t, loss)

    # Backprop для вычисления градиентов w1 и w2 относительно потерь
    grad_y_pred = 2.0 * (y_pred – y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # Обновить веса с помощью градиентного спуска
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Очевидно, что такие объяснения не дают понимания того, что на самом деле происходит: лежащих в основе математических принципов, отдельных компонентов нейронной сети, как они работают вместе и т. д.[1]

А каким должно быть хорошее объяснение? Тут полезно посмотреть, как объясняются другие концепции информатики: например, если вы хотите узнать об алгоритмах сортировки, то в учебнике найдется:

- объяснение принципа работы алгоритма простыми словами;
- визуальное объяснение работы алгоритма — схема, график;
- математическое объяснение почему алгоритм работает[2];
- псевдокод, реализующий алгоритм.

[1] Этот пример был задуман как иллюстрация библиотеки PyTorch для тех, кто уже разбирается в нейронных сетях, а не как поучительное руководство. Тем не менее многие учебники придерживаются именно такого стиля, давая только код вместе с краткими пояснениями.

[2] В частности, в случае алгоритмов сортировки — почему мы получаем отсортированный список.

Эти элементы объяснения нейронных сетей редко (или никогда) объединяются, хотя мне кажется очевидным, что правильное объяснение нейронных сетей должно быть именно таким. Эта книга — попытка восполнить этот пробел.

Для понимания нейронных сетей нужно несколько мысленных моделей

Я не исследователь и не доктор наук. Но я профессионально преподавал Data Science: провел пару буткемпов по Data Science в компании Metis, а затем в течение года путешествовал по всему миру с Metis, проводя однодневные семинары для компаний из разных отраслей. На этих семинарах я рассказывал о машинном обучении и основных концепциях разработки программного обеспечения. Мне всегда нравилось преподавать, и в последнее время, занимаясь в основном машинным обучением и статистикой, меня интересует вопрос, как лучше всего объяснить технические понятия. Что касается нейронных сетей, я обнаружил, что наиболее сложной задачей является создание правильной «мысленной модели» того, что представляет собой нейронная сеть, тем более что для понимания нейронных сетей полностью требуется не одна, а несколько ментальных моделей и все они освещают различные (но все же важные) аспекты работы нейронных сетей. Следующие четыре предложения являются правильными ответами на вопрос: «Что такое нейронная сеть?»

- Нейронная сеть — это математическая функция, которая принимает входные и производит выходные данные.
- Нейронная сеть — это вычислительный граф, через который протекают многомерные массивы.
- Нейронная сеть состоит из слоев, каждый из которых может рассматриваться как ряд «нейронов».
- Нейронная сеть — это универсальный аппроксиматор функций, который теоретически может представить решение любой контролируемой проблемы обучения.

Не сомневаюсь, что читатели уже слышали одно или несколько из этих понятий раньше и в целом понимают, для чего нужны нейронные сети. Однако для полного понимания нужно осознать их все и показать, как

они связаны. Как тот факт, что нейронная сеть может быть представлена в виде вычислительного графа, сопоставить, например, с понятием слоя? Для более точного понимания мы реализуем все эти концепции с нуля в Python и соединим их, создавая рабочие нейронные сети, которые вы можете обучать на своем компьютере дома. Несмотря на то что мы уделим немало времени деталям реализации, *целью реализации этих моделей в Python будет укрепление и уточнение нашего понимания концепций. Здесь спешка не нужна.*

Я хотел бы, чтобы после прочтения этой книги у вас было такое глубокое понимание всех этих мысленных моделей и результатов их работы, чтобы понимать, каким образом нейронные сети должны быть *реализованы*. После этого все концепции, связанные с обучением в будущих проектах, станут для вас проще.

Структура книги

Первые три главы наиболее важные и сами по себе достойны отдельных книг.

1. В главе 1 я покажу, как представлять математические функции в виде последовательности операций, связанных вместе, чтобы сформировать вычислительный граф. Также я покажу, как это представление позволяет нам вычислять производные выходов этих функций по отношению к их входам, используя правило цепи. В конце главы расскажу об очень важной операции — умножении матрицы — и покажу, как она может вписаться в математическую функцию и в то же время позволит нам вычислить производные, которые понадобятся для глубокого обучения.
2. В главе 2 мы будем использовать строительные блоки, которые сделали в главе 1 с целью создания и обучения модели для решения реальных проблем: в частности, мы будем использовать их для построения моделей линейной регрессии и нейронных сетей для прогнозирования цен на жилье на основе реальных данных. Я покажу, что нейронная сеть работает лучше, чем линейная регрессия, и попытаюсь объяснить почему. Подход к построению моделей «сначала суть» в этой главе должен дать вам очень хорошее представление о том, как работают нейронные сети, а также покажет ограниченные

возможности пошагового подхода для определения моделей глубокого обучения. И тут на сцену выходит глава 3.

3. В главе 3 мы возьмем строительные блоки из подхода «сначала суть» из первых двух глав и используем их для построения компонентов более высокого уровня, которые составляют все модели глубокого обучения: слои, модели, оптимизаторы и т. д. Мы закончим эту главу обучением модели глубокого обучения, заданной с нуля, на наборе данных из главы 2 и покажем, что она работает лучше, чем простая нейронная сеть.

4. Как выясняется, существует несколько теоретических предпосылок того, что нейронная сеть с заданной архитектурой действительно найдет хорошее решение для данного набора данных при обучении с использованием стандартных методов обучения, которые мы будем использовать в этой книге. В главе 4 мы поговорим о хитростях, применяемых в процессе обучения, которые обычно увеличивают вероятность того, что нейронная сеть найдет хорошее решение, и по возможности дадим математическое описание, почему они работают.

5. В главе 5 мы обсудим фундаментальные идеи, лежащие в основе сверточных нейронных сетей (CNN), разновидности архитектуры нейронных сетей, специализирующихся на распознавании изображений. Существует множество объяснений принципов работы CNN, поэтому я сосредоточусь на самых основных понятиях о CNN и их отличиях от обычных нейронных сетей: в частности, как CNN делают так, что каждый слой нейронов превращается в «карты признаков», и как два из этих слоев (каждый из которых состоит из нескольких карт объектов) связываются друг с другом посредством сверточных фильтров. Кроме того, что мы будем писать обычные слои в нейронной сети с нуля, а также сверточные слои с нуля, чтобы укрепить понимание того, как они работают.

6. В первых пяти главах мы создадим миниатюрную библиотеку нейронных сетей, которая определяет нейронные сети как серию слоев, которые сами состоят из серии операций, которые передают входные данные вперед и градиенты назад. Но большинство нейронных сетей реализуются на практике не так. Вместо этого используется техника, называемая автоматическим дифференцированием. Я приведу краткий обзор автоматического дифференцирования в начале главы 6 и далее использую его для основной темы главы: рекур-

рентных нейронных сетей (RNN), архитектуры нейронных сетей, обычно используемых для анализа данных, в которых точки данных появляются последовательно, например временны́х данных или естественного языка. Я объясню работу «классических RNN» и двух вариантов: GRU и LSTM (и конечно, мы реализуем все их с нуля). Далее мы опишем элементы, которые являются общими для всех этих вариантов RNN, и некоторые различия между этими вариантами.

7. В заключение, в главе 7, я покажу, как все, что мы делали с нуля в главах 1–6, может быть реализовано с использованием высокопроизводительной библиотеки с открытым исходным кодом PyTorch. Изучение такой структуры очень важно для развития вашего знания нейронных сетей; но погружение и изучение структуры без предварительного понимания того, как и почему работают нейронные сети, серьезно ограничит ваше обучение в долгосрочной перспективе. Цель такого порядка глав в этой книге — дать вам возможность писать чрезвычайно высокопроизводительные нейронные сети (с помощью PyTorch), настраивая при этом вас на долгосрочное обучение и успех (через изучение основ). В конце мы приведем краткую иллюстрацию того, как нейронные сети могут использоваться для обучения без учителя.

Моя идея состояла в том, чтобы написать книгу, которую я сам бы хотел почитать, когда только начинал изучать эту тему несколько лет назад. Надеюсь, книга будем вам полезна. Вперед!

Условные обозначения

В книге используются следующие типографские обозначения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных среды, операторов и ключевых слов.

Моноширинный жирный

Обозначает команды или другой текст, который должен вводиться пользователем.

Моноширинный курсив

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Теорема Пифагора: $a^2 + b^2 = c^2$.

Так обозначаются советы, предложения и примечания общего характера.

Использование примеров кода

Дополнительный материал (примеры кода, упражнения и т. д.) можно скачать по адресу репозитория книги на GitHub по адресу oreil.ly/deep-learning-github.

Благодарности

Благодарю своего редактора Мелиссу Поттер вместе с командой из O'Reilly, которые были внимательны и отвечали на мои вопросы на протяжении всего процесса.

Выражаю особую благодарность нескольким людям, чья работа по созданию технических концепций в области машинного обучения, доступная для более широкой аудитории, вдохновила меня, и тем, кого мне посчастливилось узнать лично: это, в частности, Брэндон Ропер, Джоэл Грус, Джереми Уотт и Эндрю Траск.

Благодарю своего босса в Metis и директора в Facebook, которые поддерживали меня, когда я пытался выкроить время на работу над этим проектом.

Благодарю Мэта Леонарда, который недолгое время был моим соавтором, после чего наши пути разошлись. Мэт помог организовать код в мини-

малистичном стиле — lincoln — и дал очень полезную обратную связь по поводу сырых вариантов первых двух глав, написав свои собственные версии больших разделов этих глав.

Наконец, благодарю своих друзей Еву и Джона, которые вдохновили меня на решительный шаг и фактически заставили меня начать писать. Я также хотел бы поблагодарить моих многочисленных друзей в Сан-Франциско, которые терпели мое волнение, переживали вместе со мной по поводу книги и оказывали всяческую поддержку, хотя в течение многих месяцев я не мог найти время потусоваться с ними.

От издательства

Некоторые иллюстрации снабжены QR-кодом. Перейдя по ссылке, вы сможете посмотреть их цветную версию.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

ГЛАВА 1

Математическая база

> Не нужно запоминать эти формулы. Если вы поймете принципы, по которым они строятся, то сможете придумать собственную систему обозначений.
>
> *Джон Кохран, методическое пособие Investments Notes, 2006*

В этой главе будет заложен фундамент для понимания работы нейронных сетей — вложенные математические функции и их производные. Мы пройдем весь путь от простейших строительных блоков до «цепочек» составных функций, вплоть до функции многих переменных, внутри которой происходит умножение матриц. Умение находить частные производные таких функций поможет вам понять принципы работы нейронных сетей, речь о которых пойдет в следующей главе.

Каждую концепцию мы будем рассматривать с трех сторон:

- математическое представление в виде формулы или набора уравнений;
- код, по возможности содержащий минимальное количество дополнительного синтаксиса (для этой цели идеально подходит язык Python);
- рисунок или схема, иллюстрирующие происходящий процесс.

Благодаря такому подходу мы сможем исчерпывающе понять, как и почему работают вложенные математические функции. С моей точки зрения, любая попытка объяснить, из чего состоят нейронные сети, не раскрывая все три аспекта, будет неудачной.

И начнем мы с такой простой, но очень важной математической концепции, как функция.

Функции

Как описать, что такое функция? Разумеется, я мог бы ограничиться формальным определением, но давайте рассмотрим эту концепцию с разных сторон, как ощупывающие слона слепцы из притчи.

Математическое представление

Вот два примера функций в математической форме записи:

- $f_1(x) = x^2$.
- $f_2(x) = \max(x, 0)$.

Записи означают, что функция f_1 преобразует входное значение x в x^2, а функция f_2 возвращает наибольшее значение из набора $(x, 0)$.

Визуализация

Вот еще один способ представления функций:

1. Нарисовать плоскость xy (где x соответствует горизонтальной оси, а y — вертикальной).
2. Нарисовать на этой плоскости набор точек, x-координаты которых (обычно равномерно распределенные) соответствуют входным значениям функции, а y-координаты — ее выходным значениям.
3. Соединить эти точки друг с другом.

Французский философ и математик Рене Декарт первым использовал подобное представление, и его начали активно применять во многих областях математики, в частности в математическом анализе. Пример графиков функций показан на рис. 1.1.

Есть и другой способ графического представления функций, который почти не используется в матанализе, но удобен, когда речь заходит о моделях глубокого обучения. Функцию можно сравнить с черным ящиком, который принимает значение на вход, преобразует его внутри по каким-то правилам и возвращает новое значение. На рис. 1.2 показаны две уже знакомые нам функции как в общем виде, так и для отдельных входных значений.

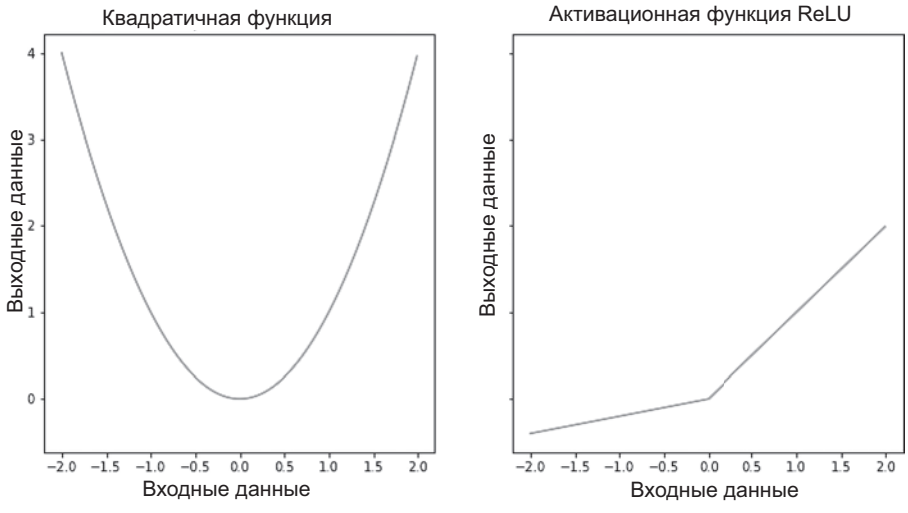


Рис. 1.1. Две непрерывные дифференцируемые функции

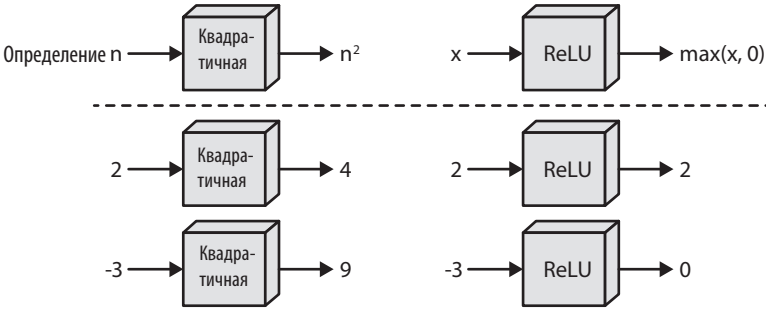


Рис. 1.2. Другой способ представления тех же функций

Код

Наконец, можно описать наши функции с помощью программного кода. Но для начала я скажу пару слов о библиотеке NumPy, которой мы воспользуемся.

Примечание № 1. NumPy

Библиотека NumPy для Python содержит реализации вычислительных алгоритмов, по большей части написанные на языке C и оптимизиро-

ванные для работы с многомерными массивами. Данные, с которыми работают нейронные сети, всегда хранятся в *многомерных массивах*, чаще всего в дву- или трехмерных. Объекты `ndarray` из библиотеки NumPy дают возможность интуитивно и быстро работать с этими массивами. Например, если сохранить данные в виде обычного или многомерного списка, обычный синтаксис языка Python не позволит выполнить поэлементное сложение или умножение списков, зато эти операции прекрасно реализуются с помощью объектов `ndarray`:

```
print("операции со списками на языке Python:")
a = [1,2,3]
b = [4,5,6]
print("a+b:", a+b)
try:
    print(a*b)
except TypeError:
    print("a*b не имеет смысла для списков в языке Python")
print()
print("операции с массивами из библиотеки numpy:")
a = np.array([1,2,3])
b = np.array([4,5,6])
print("a+b:", a+b)
print("a*b:", a*b)

операции со списками на языке Python:
a+b: [1, 2, 3, 4, 5, 6]
a*b не имеет смысла для списков в языке Python

операции с массивами из библиотеки numpy:
a+b: [5 7 9]
a*b: [ 4 10 18]
```

Объект `ndarray` обладает и таким важным для работы с многомерными массивами атрибутом, как количество измерений. Измерения еще называют осями. Их нумерация начинается с 0, соответственно первая ось будет иметь индекс 0, вторая — 1 и т. д. В частном случае двумерного массива нулевую ось можно сопоставить строкам, а первую — столбцам, как показано на рис. 1.3.

Эти объекты позволяют интуитивно понятным способом совершать различные операции с элементами осей. Например, суммирование строки или столбца двумерного массива приводит к «свертке» вдоль

соответствующей оси, возвращая массив на одно измерение меньше исходного:

```
print('a:')
print(a)
print('a.sum(axis=0):', a.sum(axis=0))
print('a.sum(axis=1):', a.sum(axis=1))

a:
[[1 2]
 [3 4]]
a.sum(axis=0): [4 6]
a.sum(axis=1): [3 7]
```

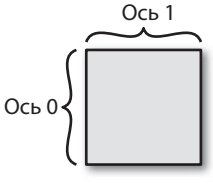


Рис. 1.3. Двумерный массив из библиотеки NumPy, в котором ось с индексом 0 соответствует строкам, а ось с индексом 1 — столбцам

Наконец, объект `ndarray` поддерживает такую операцию, как сложение с одномерным массивом. Например, к двумерному массиву *a*, состоящему из *R* строк и *C* столбцов, можно прибавить одномерный массив *b* длиной *C*, и библиотека NumPy выполнит сложение для элементов каждой строки массива a[1]:

```
a = np.array([[1,2,3],
              [4,5,6]])

b = np.array([10,20,30])

print("a+b:\n", a+b)

a+b:
[[11 22 33]
 [14 25 36]]
```

[1] Позднее это позволит нам легко добавлять смещение к результатам умножения матриц.

Примечание № 2. Функции с аннотациями типов

Как я уже упоминал, код в этой книге приводится как дополнительная иллюстрация, позволяющая более наглядно представить объясняемые концепции. Постепенно эта задача будет усложняться, так как функции с несколькими аргументами придется писать как часть сложных классов. Для повышения информативности такого кода мы будем добавлять в определение функций аннотации типов; например, в главе 3 нейронные сети будут инициализироваться вот так:

```
def __init__(self,
             layers: List[Layer],
             loss: Loss, learning_rate: float = 0.01) -> None:
```

Такое определение сразу дает представление о назначении класса. Вот для сравнения функция `operation`:

```
def operation(x1, x2):
```

Чтобы понять назначение этой функции, потребуется вывести тип каждого объекта и посмотреть, какие операции с ними выполняются. А теперь переопределим эту функцию следующим образом:

```
def operation(x1: ndarray, x2: ndarray) -> ndarray:
```

Сразу понятно, что функция берет два объекта `ndarray`, вероятно, каким-то способом комбинирует и выводит результат этой комбинации. В дальнейшем мы будем снабжать аннотациями типов все определения функций.

Простые функции в библиотеке NumPy

Теперь мы готовы написать код определенных нами функций средствами библиотеки NumPy:

```
def square(x: ndarray) -> ndarray:
    '''
    Возведение в квадрат каждого элемента объекта ndarray.
    '''
    return np.power(x, 2)

def leaky_relu(x: ndarray) -> ndarray:
    '''
    Применение функции "Leaky ReLU" к каждому элементу ndarray.
    '''
    return np.maximum(0.2 * x, x)
```

Библиотека NumPy позволяет применять многие функции к объектам `ndarray` двумя способами: `np.function_name (ndarray)` или `ndarray.function_name`. Например, функцию `relu` можно было написать как `x.clip (min = 0)`. В дальнейшем мы будем пользоваться записью вида `np.function_name (ndarray)`. И даже когда альтернативная запись короче, как, например, в случае транспонирования двумерного объекта `ndarray`, мы будем писать не `ndarray.T`, а `np.trans pose (ndarray, ( 1, 0))`.

Постепенно вы привыкнете к трем способам представления концепций, и это поможет по-настоящему понять, как происходит глубокое обучение.

Производные

Понятие производной функции, скорее всего, многим из вас уже знакомо. Производную можно определить как скорость изменения функции в рассматриваемой точке. Мы подробно рассмотрим это понятие с разных сторон.

Математическое представление

Математически производная определяется как предел отношения приращения функции к приращению ее аргумента при стремлении приращения аргумента к нулю:

$$\frac{df}{du}(a) = \lim_{\Delta \to 0} \frac{f(a+\Delta) - f(a-\Delta)}{2 \times \Delta}.$$

Можно численно оценить этот предел, присвоив переменной Δ маленькое значение, например 0.001:

$$\frac{df}{du}(a) = \frac{f(a+0.001) - f(a-0.001)}{0.002}.$$

Теперь посмотрим на графическое представление нашей производной.

Визуализация

Начнем с общеизвестного способа: если нарисовать касательную к декартову представлению функции *f*, производная функции в точке касания будет равна угловому коэффициенту касательной. Вычислить этот коэффициент, или тангенс угла наклона прямой, можно, взяв разность значений функции *f* при $a - 0.001$ и $a + 0.001$ и поделив на величину приращения, как показано на рис. 1.4.

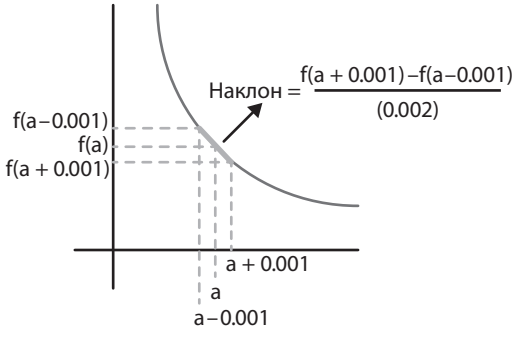


Рис. 1.4. Производная как угловой коэффициент

На рисунке производную можно представить в виде множителя, кратно которому меняется выходное значение функции при небольшом изменении подаваемого на вход значения. Фактически мы меняем значение входного параметра на очень маленькую величину и смотрим, как при этом поменялось значение на выходе. Схематично это представлено на рис. 1.5.

Рис. 1.5. Альтернативный способ визуализации концепции производной

Со временем вы увидите, что для понимания глубокого обучения второе представление оказывается важнее первого.

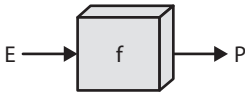
Код

И наконец, код для вычисления приблизительного значения производной:

```
from typing import Callable

def deriv(func: Callable[[ndarray], ndarray], input_: ndarray,
                        delta: float = 0.001) -> ndarray:
    '''
    Вычисление производной функции "func" в каждом элементе массива
    "input_".
    '''
    return (func(input_ + delta) - func(input_ - delta)) / (2 * delta)
```

Выражение «*P* — это функция *E*» (я намеренно использую тут случайные символы) означает, что некая функция *f* берет объекты *E* и превращает в объекты *P*, как показано на рисунке. Другими словами, *P* — это результат применения функции *f* к объектам *E*:



А вот так выглядит соответствующий код:

```
def f(input_: ndarray) -> ndarray:
    # Какое-то преобразование
    return output

P = f(E)
```

Вложенные функции

Вот мы и дошли до концепции, которая станет фундаментом для понимания нейронных сетей. Это вложенные, или составные, функции. Дело в том, что две функции f_1 и f_2 можно связать друг с другом таким образом, что выходные данные одной функции станут входными для другой.

Визуализация

Наглядно представить концепцию вложенной функции можно с помощью рисунка.

На рис. 1.6 мы видим, что данные подаются в первую функцию, преобразуются, выводятся и становятся входными данными для второй функции, которая и дает окончательный результат.

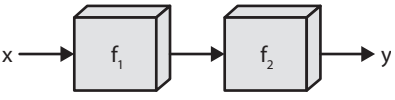


Рис. 1.6. Вложенные функции

Математическое представление

В математической нотации вложенная функция выглядит так:

$$f_2\left(f_1\left(x\right)\right) = y.$$

Такое представление уже сложно назвать интуитивно понятным, потому что читать эту запись нужно не по порядку, а изнутри наружу. Хотя, казалось бы, это должно читаться как «функция f_2 функции f_1 переменной x», но на самом деле мы вычисляем f_1 от переменной x, а затем — f_2 от полученного результата.

Код

Чтобы представить вложенные функции в виде кода, для них первым делом нужно определить тип данных:

```
from typing import List

# Function принимает в качестве аргумента объекты ndarray и выводит
# объекты ndarray
Array_Function = Callable[[ndarray], ndarray]

# Chain — список функций
Chain = List[Array_Function]
```

Теперь определим прохождение данных по цепочке из двух функций:

```
def chain_length_2(chain: Chain, a: ndarray) -> ndarray:
    '''
    Вычисляет подряд значение двух функций в объекте "Chain".
```

```
    '''
    assert len(chain) == 2, \
    "Длина объекта 'chain' должна быть равна 2"

    f1 = chain[0]
    f2 = chain[1]

    return f2(f1(x))
```

Еще одна визуализация

Так как составная функция, по сути, представляет собой один объект, ее можно представить в виде $f_1 f_2$, как показано на рис. 1.7.

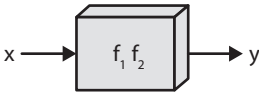


Рис. 1.7. Альтернативное представление вложенных функций

Из математического анализа известно, что если все функции, из которых состоит составная функция, дифференцируемы в рассматриваемой точке, то и составная функция, как правило, дифференцируема в этой точке! То есть функция $f_1 f_2$ — это просто обычная функция, от которой можно взять производную, — а именно производные составных функций лежат в основе моделей глубокого обучения.

Для вычисления производных сложных функций нам потребуется формула, чем мы и займемся далее.

Цепное правило

Цепное правило (или правило дифференцирования сложной функции) в математическом анализе позволяет вычислять производную композиции двух и более функций на основе индивидуальных производных. С математической точки зрения модели глубокого обучения представляют собой составные функции, а в следующих главах вы увидите, что понимание того, каким способом берутся производные таких функций, потребуется для обучения этих моделей.

Математическое представление

В математической нотации теорема утверждает, что для значения x

$$\frac{df_2}{du}(x) = \frac{df_2}{du}\left(f_1(x)\right) \times \frac{df_1}{du}(x),$$

где u — вспомогательная переменная, представляющая собой входное значение функции.

Производную функции f одной переменной можно обозначить как $\frac{df}{du}$. Вспомогательная переменная в данном случае может быть любой, ведь запись $f(x) = x^2$ и $f(y) = y^2$ означает одно и то же. Я обозначил эту переменную u.

Но позднее нам придется иметь дело с функциями нескольких переменных, например x и y. И в этом случае между $\frac{df}{dx}$ и $\frac{df}{dy}$ уже будет принципиальная разница.

Именно поэтому в начале раздела мы записали производные с помощью вспомогательной переменной u и будем в дальнейшем использовать ее для производных функций одной переменной.

Визуализация

Формула из предыдущего раздела не слишком помогает понять суть цепного правила. Давайте посмотрим на рис. 8, иллюстрирующий, что же такое производная в простом случае $f_1 f_2$.

Из рисунка интуитивно понятно, что производная составной функции *должна* представлять собой произведение производных входящих в нее функций. Предположим, что *производная* первой функции при $u = 5$ дает значение 3, то есть $\frac{df_1}{du}(5) = 3$.

Затем предположим, что *значение* первой функции при величине входного параметра 5 равно 1, то есть $f_1(5) = 1$. Производную этой функции при $u = 1$ приравняем к -2, то есть $\frac{df_2}{du}(1) = -2$.

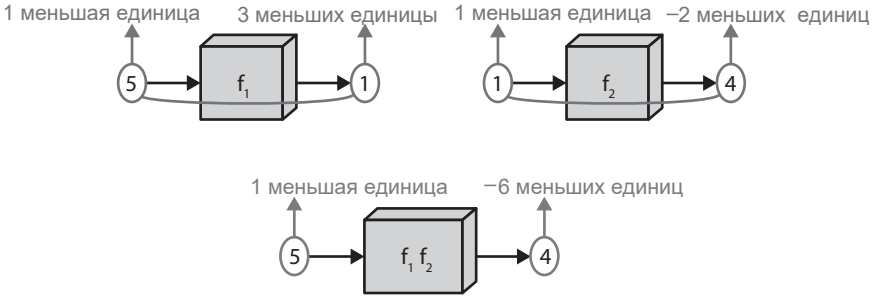


Рис. 1.8. Цепное правило

Теперь вспомним, что эти функции связаны друг с другом. Соответственно если при изменении входного значения второго черного ящика на 1 мы получим на выходе значение –2, то изменение входного значения до 3 даст нам изменение выходного значения на величину $-2 \times 3 = -6$. Именно поэтому в формуле для цепного правила фигурирует произведение:

$$\frac{df_2}{du}\left(f_1(x)\right) умножить\ на \frac{df_1}{du}(x).$$

Как видите, рассмотрение математической записи цепного правила с рисунком позволяет определить выходное значение вложенной функции по ее входному значению. Теперь посмотрим, как может выглядеть код, вычисляющий значение такой производной.

Код

Первым делом напишем код, а потом покажем, что он корректно вычисляет производную вложенной функции. В качестве примера рассмотрим уже знакомые квадратичную функцию и сигмоиду, которая применяется в нейронных сетях в качестве функции активации:

```
def sigmoid(x: ndarray) -> ndarray:
    '''
    Применение сигмоидной функции к каждому элементу объекта ndarray.
    '''
    return 1 / (1 + np.exp(-x))
```

А этот код использует цепное правило:

```
def chain_deriv_2(chain: Chain,
                  input_range: ndarray) -> ndarray:
    '''
    Вычисление производной двух вложенных функций:
    (f2(f1(x))' = f2'(f1(x)) * f1'(x) с помощью цепного правила
    '''

    assert len(chain) == 2, \
    "Для этой функции нужны объекты 'Chain' длиной 2"

    assert input_range.ndim == 1, \
    "Диапазон входных данных функции задает 1-мерный объект ndarray"

    f1 = chain[0]
    f2 = chain[1]

    # df1/dx
    f1_of_x = f1(input_range)

    # df1/du
    df1dx = deriv(f1, input_range)

    # df2/du(f1(x))
    df2du = deriv(f2, f1(input_range))

    # Поэлементно перемножаем полученные значения
    return df1dx * df2du
```

На рис. 1.9 показан результат применения цепного правила:

```
PLOT_RANGE = np.arange(-3, 3, 0.01)

chain_1 = [square, sigmoid]
chain_2 = [sigmoid, square]

plot_chain(chain_1, PLOT_RANGE)
plot_chain_deriv(chain_1, PLOT_RANGE)

plot_chain(chain_2, PLOT_RANGE)
plot_chain_deriv(chain_2, PLOT_RANGE)
```

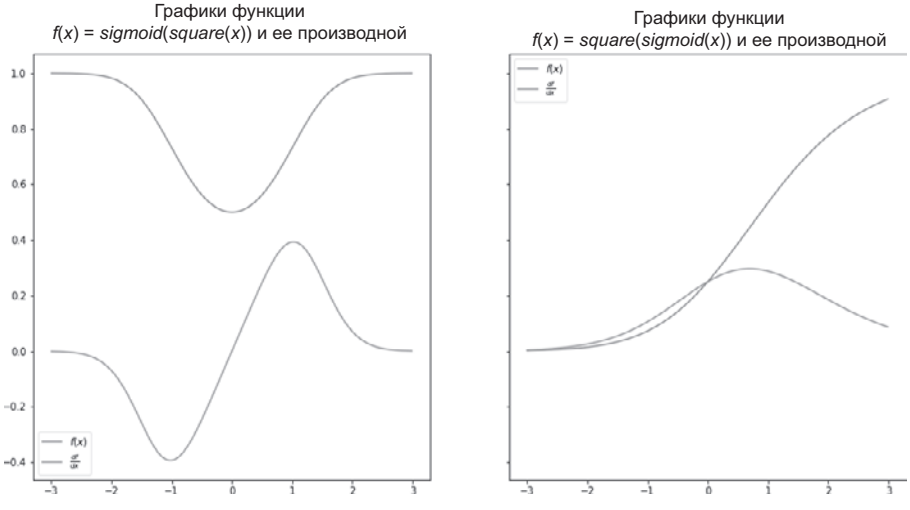


Рис. 1.9. Результат применения цепного правила

Кажется, цепное правило работает. Там, где функция наклонена вверх, ее производная положительна, там, где она параллельна оси абсцисс, производная равна нулю; при наклоне функции вниз ее производная отрицательна.

Как математически, так и с помощью кода мы можем вычислять производную «составных» функций, таких как $f_1 f_2$, если обе эти функции дифференцируемы.

С математической точки зрения модели глубокого обучения представляют собой цепочки из функций. Поэтому сейчас мы рассмотрим более длинный пример, чтобы в дальнейшем вы смогли экстраполировать эти знания на более сложные модели.

Более длинная цепочка

Возьмем три дифференцируемых функции f_1, f_2 и f_3 и попробуем вычислить производную $f_1 f_2 f_3$. Мы уже знаем, что функция, составленная из любого конечного числа дифференцируемых функций, тоже дифференцируема.

Математическое представление

Дифференцирование происходит по следующей формуле:

$$\frac{df_3}{du}(x) = \frac{df_3}{du}\left(f_2\left(f_1(x)\right)\right) \times \frac{df_2}{du}\left(f_1(x)\right) \times \frac{df_1}{du}(x).$$

Здесь работает та же схема, что и в случае цепочки из двух функций $\frac{df_2}{du}(x) = \frac{df_2}{du}\left(f_1(x)\right) \times \frac{df_1}{du}(x)$, но формула не позволяет интуитивно понять, что именно происходит!

Визуализация

Лучшее представление о том, как работает эта формула, нам даст рис. 1.10.

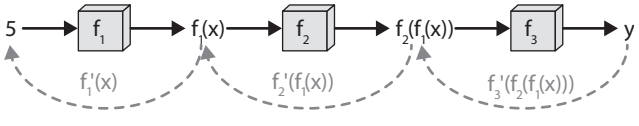


Рис. 1.10. Вычисление производной трех вложенных функций

Ход рассуждений в данном случае будет таким же, как и ранее. Представим, что все три функции как бы нанизаны на струну, входное значение функции $f_1 f_2 f_3$ обозначим a, а выходное — b. При изменении a на небольшое значение Δ результат $f_1(a)$ изменится на $\frac{df_1}{du}(x)$, умноженное на Δ. Следующий шаг в цепочке — функция $f_2(f_1(x))$ изменится на $\frac{df_2}{du}\left(f_1(x)\right) \times \frac{df_1}{du}(x)$, умноженное на Δ. Аналогичным образом рассчитывается изменение на третьем шаге. Это будет полная формула, написанная в соответствии с цепным правилом и умноженная на Δ. Потратьте некоторое время на изучение этого принципа. А когда мы начнем писать код, многие вещи постепенно прояснятся.

Код

Теперь напишем код вычисления производной по приведенной в предыдущем разделе формуле. Что интересно, на этом простом примере мы уже видим отправной момент, с которого начинается процесс прямого и обратного распространения по нейронной сети:

```
def chain_deriv_3(chain: Chain,
                  input_range: ndarray) -> ndarray:

    '''
    Вычисление производной трех вложенных функций:
    (f3(f2(f1)))' = f3'(f2(f1(x))) * f2'(f1(x)) *
    f1'(x) с помощью цепного правила
    '''

    assert len(chain) == 3, \
    "Для этой функции нужны объекты 'Chain' длиной 3"
    f1 = chain[0]
    f2 = chain[1]
    f3 = chain[2]

    # f1(x)
    f1_of_x = f1(input_range)

    # f2(f1(x))
    f2_of_x = f2(f1_of_x)

    # df3du
    df3du = deriv(f3, f2_of_x)

    # df2du
    df2du = deriv(f2, f1_of_x)

    # df1dx
    df1dx = deriv(f1, input_range)

    # Поэлементно перемножаем полученные значения
    return df1dx * df2du * df3du
```

При вычислении производной вложенных функций по цепному правилу происходит интересная вещь. Дело в том, что эта операция осуществляется в два этапа:

1. Сначала мы идем вперед, вычисляя значения `f1_of_x` и `f2_of_x`. Это можно назвать *прямым проходом* (forward pass).
2. Затем эти значения используются для расчета компонентов производной. При этом мы идем в обратную сторону.

В конце мы перемножаем эти значения и получаем производную.

Теперь на примере трех уже определенных нами функций — `sigmoid`, `square` и `leaky_relu` — посмотрим, как все работает.

```
PLOT_RANGE = np.range(-3, 3, 0.01)
plot_chain([leaky_relu, sigmoid, square], PLOT_RANGE)
plot_chain_deriv([leaky_relu, sigmoid, square], PLOT_RANGE)
```

Результат показан на рис. 1.11.

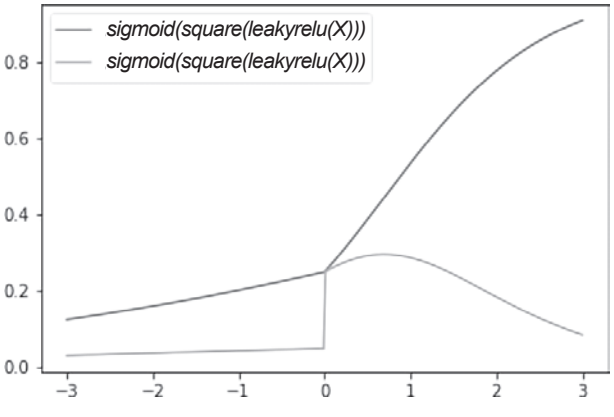


Рис. 1.11. Для трех вложенных функций цепное правило тоже работает

Сравнив графики производных с наклонами исходных функций, мы убедимся в корректной работе цепного правила.

Теперь можно рассмотреть сложные функции нескольких переменных. Они подчиняются тем же самым принципам и намного больше подходят для описания того, что происходит во время глубокого обучения.

Функции нескольких переменных

К этому моменту вы должны были понять, как формируются сложные функции, и уметь представлять эти функции в виде набора блоков с входными и выходными данными. Выше я показал как формулы для взятия производных таких функций, так и поэтапный процесс их вычисления с прямым и обратным проходами.

Функции, с которыми приходится иметь дело в глубоком обучении, часто имеют целый набор входных данных, которые в процессе обработки складываются, умножаются или комбинируются каким-то другим способом. Вычислить производную такой функции тоже несложно. В качестве примера рассмотрим простой сценарий: функция двух переменных, которая вычисляет их сумму и затем передает в другую функцию.

Математическое представление

В данном примере лучше начать с математики. Пусть входные данные представляют переменные x и y. Действие функции можно разбить на два этапа. Сначала функция, которую мы обозначим греческой буквой α, выполняет сложение входных данных. Греческие буквы и дальше будут использоваться в качестве имен функций. Результат действия функции обозначим a. С формальной точки зрения все очень просто:

$$a = \alpha(x, y) = x + y.$$

Затем передадим a в некую функцию σ (это может быть любая непрерывная функция, например сигмоид, квадратичная функция или другая функция по вашему выбору). Результат ее работы обозначим переменной s:

$$s = \sigma(a).$$

При этом ничто не мешает обозначить всю функцию f и написать:

$$f(x, y) = \sigma(x + y).$$

С математической точки зрения это более точная запись, но она не дает представления о том, что на самом деле мы последовательно выполняем две операции. Лучше всего это будет видно на рисунке, который приведен ниже.

Визуализация

Теперь, когда мы добрались до функций нескольких переменных, окончательно стало понятно, что наши схемы со стрелками, указывающими на порядок выполнения операций, представляют собой *вычислительные графы* (computational graphs). Например, на рис. 1.12 показан вычислительный граф описанной выше функции *f*.

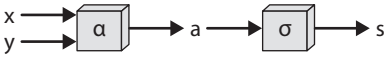


Рис. 1.12. Функция двух переменных

Мы видим, что в функцию α подаются два элемента входных данных, а результат ее работы *a* передается в функцию σ.

Код

Код вычислений в данном случае выглядит очень просто, но обратите внимание на дополнительный проверочный оператор:

```
def multiple_inputs_add(x: ndarray,
                        y: ndarray,
                        sigma: Array_Function) -> float:
    '''
    Функция сложения двух переменных, прямой проход.
    '''
    assert x.shape == y.shape

    a = x + y
    return sigma(a)
```

Эта функция не похожа на те, с которыми мы имели дело раньше. Если в предыдущих случаях наши функции по отдельности обрабатывали каждый элемент объекта `ndarray`, то теперь мы сначала проверяем форму этих объектов, чтобы удостовериться, что с ними можно проводить указанную в коде операцию. Такие проверки всегда выполняются для функций нескольких переменных. Для такой простой операции, как сложение, достаточно проверить идентичность форм. Только в этом случае мы сможем выполнять операцию поэлементно.

Производные функций нескольких переменных

Вряд ли вас удивит тот факт, что производную такой функции можно брать по обеим ее переменным.

Визуализация

По сути, мы делаем то же самое, что и в случае функции одной переменной: идем назад по нашему графу, вычисляя производные всех составляющих функций, а затем получаем производную путем перемножения результатов (рис. 1.13).

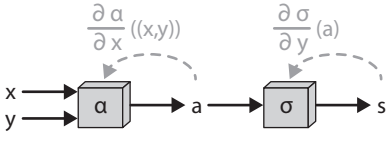


Рис. 1.13. Обратный проход через вычислительный граф функции нескольких переменных

Математическое представление

К функциям нескольких переменных тоже применимо цепное правило. Так как в рассматриваемом случае речь идет о вложенной функции $f(x,y)=\sigma(\alpha(x,y))$, то получаем:

$$\frac{\partial f}{\partial x}=\frac{\partial \sigma}{\partial u}(\alpha(x,y))\times\frac{\partial \alpha}{\partial x}((x,y))=\frac{\partial \sigma}{\partial u}(x+y)\times\frac{\partial \alpha}{\partial x}((x,y)).$$

Результат дифференцирования по второй переменной $\frac{\partial f}{\partial y}$ будет идентичным.

Теперь обратите внимание, что:

$$\frac{\partial \alpha}{\partial x}((x,y))=1,$$

так как для каждой единицы приращения по x при любых значениях x приращение α тоже составляет единицу (это же справедливо и для y).

С учетом сказанного можно написать код, вычисляющий производную такой функции.

Код

```
def multiple_inputs_add_backward(x: ndarray,
                                 y: ndarray,
                                 sigma: Array_Function) -> float:
    '''
    Вычисление производной нашей простой функции по обеим переменным.
    '''

    # "Прямой проход"
    a = x + y

    # Вычисление производных
    dsda = deriv(sigma, a)

    dadx, dady = 1, 1

    return dsda * dadx, dsda * dady
```

В качестве самостоятельной работы рассмотрите такую же функцию, в которой переменные x и y не складываются, а перемножаются.

А мы перейдем к более сложному примеру, который практически полностью воспроизводит происходящее в глубоком обучении. От функции, рассмотренной в этом разделе, она будет отличаться только векторными аргументами.

Функции нескольких переменных с векторными аргументами

В глубоком обучении используются функции, на вход которых подаются *векторы* или *матрицы*. Эти объекты можно складывать, перемножать, а для векторов существует еще и такая операция, как скалярное произведение. Ниже мы рассмотрим примеры применения к этим функциям цепного правила.

Именно эти техники позволяют понять, почему работает глубокое обучение. Основная цель глубокого обучения — подбор модели, наилучшим образом описывающей данные. Фактически мы ищем функцию, которая точнее всего сможет сопоставить результаты *наблюдений* (служащие ее входными данными) с определенным *шаблоном* (который играет роль выходных данных). Входные данные удобно представлять в виде матриц, строки которых содержат результаты наблюдений, а столбцы — соответствующие количественные характеристики. Подробно это будет обсуждаться в следующей главе, а пока важно понять математическую базу происходящего, то есть научиться вычислять производные сложных функций, в которых происходит скалярное умножение векторов или умножение матриц.

Математическое представление

При работе с нейронными сетями единицы информации или результаты наблюдений обычно представляют в виде набора признаков. Каждый признак обозначается как x_1, x_2 и т. д., до последнего признака x_n:

$$X = \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix}.$$

Например, в следующей главе мы построим нейронную сеть, которая будет прогнозировать цену на жилье; в этом примере x_1, x_2 и далее — это числовые характеристики, такие как площадь дома или его расстояние до ближайшей школы.

Создание новых признаков из уже существующих

Возможно, самая распространенная операция в нейронных сетях — определение взвешенной суммы признаков. Именно этот параметр усиливает определенные признаки и снимает акцент с других. Его можно рассматривать как новую функцию, полученную комбинированием существующих. Математически это скалярное произведение вектора характеристик и имеющего такой же размер вектора весов: w_1, w_2 и дальше до w_n. Давайте рассмотрим эту концепцию более подробно.

Математическое представление

Если определить вектор весов отдельных признаков как

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix},$$

скалярное произведение двух векторов будет выглядеть так:

$$N = v(X, W) = X \times W = x_1 \times w_1 + x_2 \times w_2 + \ldots + x_n \times w_n .$$

Обратите внимание, что эта операция представляет собой частный случай *умножения матриц*. Фактически мы умножаем вектор-строку X на вектор-столбец W.

Теперь посмотрим, как изобразить это графически.

Визуализация

Простой способ изображения скалярного произведения показан на рис. 1.14.

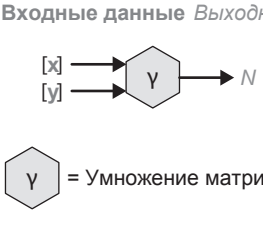


Рис. 1.14. Скалярное произведение двух векторов

Мы видим, что функция принимает два аргумента, роль которых могут играть объекты `ndarray`, и превращает их в один объект `ndarray`.

Но работу функций нескольких переменных можно представить и другими способами. Например, выделив отдельные операции и входные данные, как показано на рис. 1.15 и 1.16.

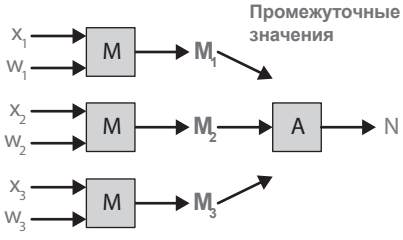


Рис. 1.15. Другой способ представления операции умножения матриц

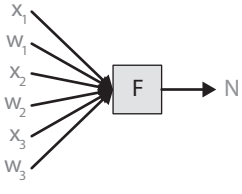


Рис. 1.16. Третий способ предоставления умножения матриц

Дело в том, что произведение матриц, как и его частный случай, скалярное произведение, — это краткое представление для набора отдельных операций, которое, как мы увидим в следующем разделе, сокращает и процедуру вычисления производных.

Код

И наконец, несложный код, реализующий операцию умножения:

```
def matmul_forward(X: ndarray,
                   W: ndarray) -> ndarray:
    '''
    Прямой проход при умножении матриц.
    '''

    assert X.shape[1] == W.shape[0], \
    '''
    Для операции умножения число столбцов первого массива должно
    совпадать с числом строк второго; у нас же число столбцов
    первого массива равно {0}, а число строк второго равно {1}.
    '''
```

```
.format(X.shape[1], W.shape[0])

# умножение матриц
N = np.dot(X, W)

return N
```

Этот код содержит оператор проверки, гарантирующий допустимость операции умножения. Раньше такая проверка не требовалась, так как мы имели дело с объектами `ndarray` одинакового размера, операции над которыми выполнялись поэлементно.

Производные функции нескольких векторных переменных

Производная функции одной переменной, например $f(x) = x^2$ или $f(x) = \text{sigmoid}(x)$, находится очень легко — достаточно применить соответствующее правило. А как должна выглядеть производная векторных функций? Обозначим скалярное произведение $\nu(X,W) = N$ и попробуем ответить на вопрос, что такое $\frac{\partial N}{\partial X}$ и $\frac{\partial N}{\partial W}$.

Визуализация

По сути, мы хотим сделать что-то, представленное на рис. 1.17.

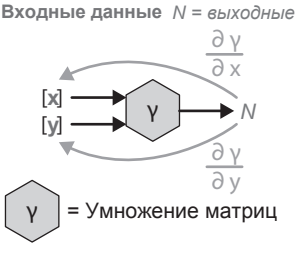


Рис. 1.17. Обратный проход операции умножения матриц

В случае обычных операций сложения и умножения такие производные вычисляются просто, как вы видели в ранее рассмотренных примерах.

Но что делать в случае умножения матриц? Давайте посмотрим, как это выглядит математически.

Математическое представление

Первым делом следует определить «производную по матрице». Если вспомнить, что матрица — это удобная форма представления набора чисел, легко понять, что производная по матрице означает производную по каждому из ее элементов. Для вектора-строки X она будет выглядеть так:

$$\frac{\partial \nu}{\partial X} = \begin{bmatrix} \frac{\partial \nu}{\partial x_1} & \frac{\partial \nu}{\partial x_2} & \frac{\partial \nu}{\partial x_3} \end{bmatrix}.$$

Но функция ν дает число: $N = x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$. Посмотрев на него внимательно, мы увидим, что изменение x_1 на ϵ единиц меняет N на $w_1 \times \epsilon$ единиц. То же самое происходит с остальными x_i элементами. Поэтому:

$$\frac{\partial \nu}{\partial x_1} = w_1,$$

$$\frac{\partial \nu}{\partial x_2} = w_2,$$

$$\frac{\partial \nu}{\partial x_3} = w_3.$$

В результате получаем:

$$\frac{\partial \nu}{\partial X} = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} = W^T.$$

Этот удивительно элегантный результат дает ключ к тому, почему глубокое обучение работает и может быть так точно реализовано.

Используя аналогичные рассуждения, получим:

$$\frac{\partial \nu}{\partial W} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = X^T.$$

Код

Сложную часть рассуждений, то есть математический вывод ответа, мы уже проделали. Написать код легко:

```
def matmul_backward_first(X: ndarray,
                          W: ndarray) -> ndarray:
    '''
    Обратный проход для операции умножения матриц по первому аргументу.
    '''

    # обратный проход
    dNdX = np.transpose(W, (1, 0))

    return dNdX
```

Рассчитанный здесь параметр `dNdX` представляет собой частную производную каждого элемента вектора X по скалярному произведению N. Этот показатель мы в дальнейшем будем называть градиентом X. Дело в том, что каждому элементу вектора X, например x_3, в массиве `dNdx` соответствует элемент (в нашем случае это `dNdX [2]`), представляющий собой частную производную скалярного произведения N по x_3. Далее в книге термин «градиент» будет обозначать многомерный аналог частной производной; точнее говоря, массив частных производных функции по каждому из ее аргументов.

Производные векторных функций: продолжение

Разумеется, модели глубокого обучения включают в себя целые цепочки операций как с векторными аргументами, так и с поэлементной обработкой поданного на вход массива `ndarray`. Поэтому сейчас мы рассмотрим процесс вычисления производной составной функции, которая включает в себя *оба* вида аргументов. Предположим, что функция $\nu(X, W)$ вычисляет скалярное произведение векторов X и W и передает полученный результат в функцию σ. Мы хотим получить ее частные производные, или, если использовать новую терминологию, вычислить градиент этой новой функции по переменным X и W. Уже в следующей главе я подробно расскажу, как это связано с работой нейронных сетей, а пока же мы просто учимся находить градиенты вычислительных графов произвольной сложности.

Визуализация

Схема на рис. 1.18 отличается от схемы на рис. 1.17 только добавленной в конец функцией σ.

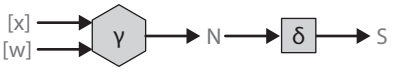


Рис. 1.18. Уже знакомая схема, к которой добавлена еще одна функция

Математическое представление

Формула такой функции выглядит очень просто:

$$s = f(X,W) = \sigma\left(\nu(X,W)\right) = \sigma\left(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3\right).$$

Код

Теперь напишем ее код:

```
def matrix_forward_extra(X: ndarray,
                         W: ndarray,
                         sigma: Array_Function) -> ndarray:
    '''
    Вычисление функции, в которой результат умножения матриц
    передается в следующую функцию.
    '''
    assert X.shape[1] == W.shape[0]

    # умножение матриц
    N = np.dot(X, W)

    # подаем результат умножения матриц на выход функции сигма
    S = sigma(N)

    return S
```

Вычисление производной

Обратный проход в этом случае представляет собой всего лишь небольшое расширение предыдущего примера.

Математическое представление

Так как $f(X, W)$ — это вложенная функция, то есть $f(X, W) = \sigma(\nu(X, W))$, ее частная производная, например, по X должна выглядеть так:

$$\frac{\partial f}{\partial X} = \frac{\partial \sigma}{\partial u}(\nu(X, W)) \times \frac{\partial \nu}{\partial X}(X, W).$$

Но первая часть этого уравнения — это всего лишь:

$$\frac{\partial \sigma}{\partial u}(\nu(X, W)) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3).$$

Так как σ — непрерывная функция, производную которой можно вычислить в любой точке, просто подставим в нее значение $x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3$. В предыдущем разделе мы вывели, что $\frac{\partial \nu}{\partial X}(X, W) = W^T$. Сделаем подстановку и получим:

$$\frac{\partial f}{\partial X} = \frac{\partial \sigma}{\partial u}(\nu(X, W)) \times \frac{\partial \nu}{\partial X}(X, W) = \frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3) \times W^T.$$

Как и в предыдущем примере, появляется вектор, совпадающий по направлению с вектором X, что в финале дает число $\frac{\partial \sigma}{\partial u}(x_1 \times w_1 + x_2 \times w_2 + x_3 \times w_3)$, умноженное на вектор-строку.

Визуализация

Схема из рис. 1.19 обратного прохода рассматриваемой функции напоминает схему из предыдущего примера, которую вы видели на рис. 1.17. Просто сейчас появился еще один множитель. Он представляет собой производную функции σ, оцененную для значения, которое мы получили в качестве результата умножения матриц.

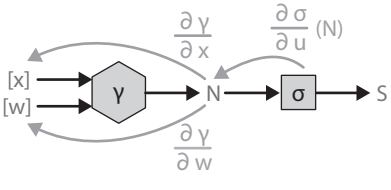


Рис. 1.19. Обратный проход для векторной составной функции

Код

Код для обратного прохода выглядит очень просто:

```
def matrix_function_backward_1(X: ndarray,
                               W: ndarray,
                               sigma: Array_Function) -> ndarray:
    '''
    Вычисление частной производной функции по первому аргументу.
    '''
    assert X.shape[1] == W.shape[0]

    # умножение матриц
    N = np.dot(X, W)

    # передача результата умножения матриц в функцию сигма
    S = sigma(N)

    # обратный проход
    dSdN = deriv(sigma, N)

    # вычисление dNdX
    dNdX = np.transpose(W, (1, 0))

    # произведение результатов; так как dNdX имеет размерность 1x1,
    # порядок множителей не имеет значения
    return np.dot(dSdN, dNdX)
```

Обратите внимание, что, как и в предыдущем примере, мы вычисляем значение функции во время прямого прохода (здесь оно обозначено просто как *N*), а затем используем это значение для обратного прохода.

Проверка корректности результата

Как определить, правильно ли мы нашли производные? Есть простой способ. Нужно немного поменять входное значение и посмотреть, как это отразится на результате. Например, в рассматриваемом случае мы увидим, что вектор *X* равен:

```
print(X)
```

```
[[ 0.4723 0.6151 -1.7262]]
```

Если увеличить x_3 на 0,01, то есть с –1,726 до –1,716, значение функции, которое мы вычисляем во время прямого прохода, тоже должно немного увеличиться, что мы и видим на рис. 1.20.

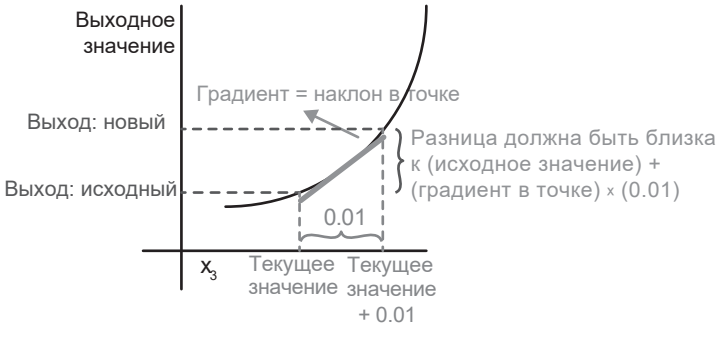


Рис. 1.20. Проверка градиента

А теперь выведем значение производной *matrix_function_backward_1*. Мы увидим, что наш градиент равен `-0.1121`:

```
print(matrix_function_backward_1(X, W, sigmoid))
```

```
[[ 0.0852 -0.0557 -0.1121]]
```

Если мы правильно рассчитали градиент, увеличение переменной x_3 на 0,01 должно привести к уменьшению производной примерно на `0,01 × -0,1121 = -0,001121`. Любой другой результат, как в большую, так и в меньшую сторону, будет означать, что цепное правило не работает. В данном случае вычисления[1] показывают, что мы все посчитали верно!

В заключение рассмотрим пример, в основе которого лежит весь ранее изученный материл. Этот пример непосредственно касается моделей, которые мы будем создавать в следующей главе.

Вычислительный граф для двух матриц

Как в машинном обучении в целом, так и в глубоком обучении в частности на вход подаются два двумерных массива, один из которых пред-

[1] Код для материалов этой главы вы найдете в репозитории GitHub oreil.ly/2ZUwKOZ.

ставляет набор данных X, а второй — их веса W. В следующей главе мы подробно поговорим о причинах такого представления данных, а пока сосредоточимся на математической стороне дела. В частности, мы покажем, что цепное правило работает даже после перехода от скалярного произведения векторов к произведению матриц и что написать код для вычисления такой производной по-прежнему очень просто.

Математические расчеты, как и в предыдущих случаях, не сложные, но объемные. При этом они дают краткий и лаконичный результат. Разумеется, мы рассмотрим процесс пошагово и свяжем его с кодом и схемами.

Математическое представление

Пусть

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix},$$

а

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}.$$

Это может быть набор данных, в котором каждому наблюдению сопоставлены три признака. Три строки соответствуют трем наблюдениям, для которых мы хотим получить некий прогноз.

Проделаем с этими матрицами следующие операции:

1. Найдем их произведение $N = \nu(X, W)$.
2. Подадим выходные данные функции N на вход дифференцируемой функции σ, обозначив это как $S = \sigma(N)$.

Как и прежде, требуется найти градиенты S по аргументам X и W и определить, можно ли в этом случае воспользоваться цепным правилом.

Отличие от ранее рассмотренных случаев состоит в том, что теперь придется иметь дело с матрицами, а не с числами. Возникает вопрос, что такое градиент одной матрицы по другой?

Нам доступны различные действия с многомерными массивами, но чтобы корректно определить понятие «градиент» по выходным данным, нужно суммировать (или каким-то другим способом превратить в одно число) последний массив последовательности. Только тогда будет иметь смысл вопрос: «Насколько изменение каждого элемента матрицы X повлияет на конечный результат?»

Поэтому мы добавим функцию лямбда, суммирующую элементы функции S.

Теперь опишем это языком формул. Начнем с произведения матриц X и W:

$$X \times W = \begin{bmatrix} x_{11} \times w_{11} + x_{12} \times w_{21} + x_{13} \times w_{31} & x_{11} \times w_{12} + x_{12} \times w_{22} + x_{13} \times w_{32} \\ x_{21} \times w_{11} + x_{22} \times w_{21} + x_{23} \times w_{31} & x_{21} \times w_{12} + x_{22} \times w_{22} + x_{23} \times w_{32} \\ x_{31} \times w_{11} + x_{32} \times w_{21} + x_{33} \times w_{31} & x_{31} \times w_{12} + x_{32} \times w_{22} + x_{33} \times w_{32} \end{bmatrix} =$$

$$= \begin{bmatrix} XW_{11} & XW_{12} \\ XW_{21} & XW_{22} \\ XW_{31} & XW_{32} \end{bmatrix}.$$

Элемент результирующей матрицы, расположенный в ряду i и столбце j, для удобства обозначим XW_{ij}.

Передадим полученную матрицу в функцию σ, что означает применение этой функции ко всем элементам произведения матриц $X \times W$:

$$\sigma(X \times W) = \begin{bmatrix} \sigma(x_{11} \times w_{11} + x_{12} \times w_{21} + x_{13} \times w_{31}) & \sigma(x_{11} \times w_{12} + x_{12} \times w_{22} + x_{13} \times w_{32}) \\ \sigma(x_{21} \times w_{11} + x_{22} \times w_{21} + x_{23} \times w_{31}) & \sigma(x_{21} \times w_{12} + x_{22} \times w_{22} + x_{23} \times w_{32}) \\ \sigma(x_{31} \times w_{11} + x_{32} \times w_{21} + x_{33} \times w_{31}) & \sigma(x_{31} \times w_{12} + x_{32} \times w_{22} + x_{33} \times w_{32}) \end{bmatrix} =$$

$$= \begin{bmatrix} \sigma(XW_{11}) & \sigma(XW_{12}) \\ \sigma(XW_{21}) & \sigma(XW_{22}) \\ \sigma(XW_{31}) & \sigma(XW_{32}) \end{bmatrix}.$$

После этого остается найти сумму всех элементов:

$$L = \Lambda(\sigma(X \times W)) = \Lambda\left(\begin{bmatrix} \sigma(XW_{11}) & \sigma(XW_{12}) \\ \sigma(XW_{21}) & \sigma(XW_{22}) \\ \sigma(XW_{31}) & \sigma(XW_{32}) \end{bmatrix}\right) =$$

$$= \sigma(XW_{11}) + \sigma(XW_{12}) + \sigma(XW_{21}) + \sigma(XW_{22}) + \sigma(XW_{31}) + \sigma(XW_{32}).$$

Теперь все свелось к уже знакомой задаче из математического анализа: есть функция L и нужно найти ее градиент по переменным X и W, чтобы узнать, насколько на нее повлияет изменение каждого элемента входных матриц (x_{11}, w_{21} и т. д.). Математически это записывается следующим образом:

$$\frac{\partial \Lambda}{\partial u}(X) = \begin{bmatrix} \frac{\partial \Lambda}{\partial u}(x_{11}) & \frac{\partial \Lambda}{\partial u}(x_{12}) & \frac{\partial \Lambda}{\partial u}(x_{13}) \\ \frac{\partial \Lambda}{\partial u}(x_{21}) & \frac{\partial \Lambda}{\partial u}(x_{22}) & \frac{\partial \Lambda}{\partial u}(x_{23}) \\ \frac{\partial \Lambda}{\partial u}(x_{31}) & \frac{\partial \Lambda}{\partial u}(x_{32}) & \frac{\partial \Lambda}{\partial u}(x_{33}) \end{bmatrix}.$$

Теперь давайте посмотрим, как наша задача описывается языком схем, и напишем соответствующий код.

Визуализация

Концептуально мы делаем то же, что и в предыдущих примерах с вычислительными графами для многих переменных. Вы без труда сможете понять рис. 1.21.

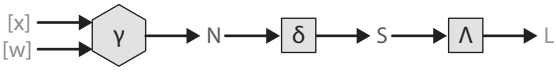


Рис. 1.21. Граф прямого прохода сложной функции

Мы просто передаем данные в функцию и утверждаем, что и в этом случае цепное правило позволит вычислить нужные градиенты.

Код

Вот как может выглядеть наш код:

```
def matrix_function_forward_sum(X: ndarray,
                                W: ndarray,
                                sigma: Array_Function) -> float:
    '''
    Прямой проход функции объектов ndarray X и W и функции sigma.
```

```
    '''
    assert X.shape[1] == W.shape[0]

    # умножение матриц
    N = np.dot(X, W)

    # передача результата умножения матриц в функцию сигма
    S = sigma(N)

    # сумма всех элементов
    L = np.sum(S)

    return L
```

Самое интересное: обратный проход

Пришло время выполнить обратный проход и показать, каким образом, даже в случае умножения матриц, мы можем вычислить градиент N по каждому элементу входных объектов `ndarray`[1]. Как только вы поймете, как все происходит, то без проблем сможете перейти к обучению моделей, которым мы и займемся в главе 2. Первым делом вспомним, что нужно делать.

Визуализация

Выполняем уже знакомую по предыдущим примерам процедуру; рис. 1.22 должен быть вам знаком, как и рис. 1.21.

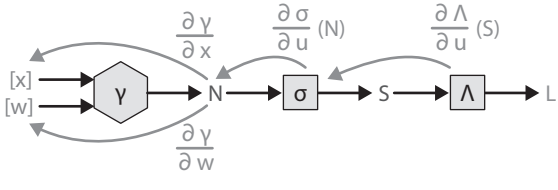


Рис. 1.22. Обратный проход через сложную функцию

[1] Так как градиенты N по X и по W вычисляется одинаково, мы рассмотрим только градиент по X.

Нужно вычислить частную производную каждого элемента функции, оценить ее по входным данным и перемножить результаты, чтобы получить окончательную производную. Давайте по очереди рассмотрим каждую из частных производных.

Математическое представление

Отимечу, что все вычисления можно произвести, что называется, в лоб. Ведь L — это функция переменных x_{11}, x_{12} и т. д., до переменной x_{33}.

Но эта задача выглядит очень сложной. В конце концов, смысл цепного правила в том, что оно позволяет разбивать сложные функции на составные части, производить вычисления с этими частями и перемножать результаты. Именно это дает возможность легко писать код вычисления производных: достаточно пошагово выполнить прямой проход, сохранить результат и использовать его для оценки нужных нам производных во время обратного прохода.

Я покажу, что этот подход работает и в случае матриц.

Обозначим функцию $\Lambda\left(\sigma\left(\nu\left(X,W\right)\right)\right)$ как L. Будь это обычная функция скалярных переменных, то в соответствии с цепным правилом можно было бы написать:

$$\frac{\partial\Lambda}{\partial X}\left(X\right)=\frac{\partial\nu}{\partial X}\left(X,W\right)\times\frac{\partial\sigma}{\partial u}\left(N\right)\times\frac{\partial\Lambda}{\partial u}\left(S\right).$$

После чего оставалось бы по очереди вычислить каждую из трех частных производных. Именно так мы поступали раньше в примере с тремя вложенными функциями. Согласно рис. 1.22, этот подход должен сработать и сейчас.

Вычислить последний компонент просто. Мы хотим узнать, насколько возрастет функция L при увеличении каждого элемента функции S. Как вы помните, L представляет собой сумму всех элементов функции S, и производная будет:

$$\frac{\partial\Lambda}{\partial u}\left(S\right)=\begin{bmatrix}1 & 1\\ 1 & 1\\ 1 & 1\end{bmatrix},$$

так как увеличение любого элемента функции S, например, на 0.46 единицы будет увеличивать Λ на те же 0.46 единицы.

Дальше следует компонент $\frac{\partial \sigma}{\partial u}(N)$. Это производная функции σ, оцененная для рассматриваемого значения элементов произведения матриц N. Перейдя к ранее использовавшемуся обозначению XW, получим:

$$\begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix}.$$

Ничто не мешает выполнить поэлементное умножение двух производных и вычислить $\frac{\partial L}{\partial u}(N)$:

$$\frac{\partial \Lambda}{\partial \mathrm{u}}(N) = \frac{\partial \Lambda}{\partial \mathrm{u}}(S) \times \frac{\partial \sigma}{\partial \mathrm{u}}(N) =$$

$$= \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix}.$$

А дальше начинаются сложности. Ведь согласно схеме и цепному правилу, дальше нужно вычислить $\frac{\partial \nu}{\partial u}(X)$. Еще раз напомню, что N — это вывод функции ν, то есть результат умножения матриц X и W. Фактически мы хотим понять, насколько увеличение каждого элемента матрицы X (размером 3×3) повлияет на каждый элемент матрицы N (размером 3×2). Не совсем понятно, каким образом выполнить такой переход между матрицами разной формы и будет ли это вообще иметь смысл.

Если помните, раньше нам везло. Ведь матрица X оказывалась транспонированной матрицей W, и наоборот. И мы получали, что $\frac{\partial \nu}{\partial u}(X) = W^T$,

а $\frac{\partial \nu}{\partial u}(W) = X^T$. Попробуем найти что-то аналогичное в рассматриваемом случае. Фактически нам нужно определить, что скрывается под знаком «?» в выражении:

$$\frac{\partial \Lambda}{\partial u}(X) = \frac{\partial \Lambda}{\partial u}(\sigma(N)) \times ? = \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix} \times ?$$

Если вспомнить, как происходит умножение матриц, станет понятно, что знак «?» — это W^T, как в более простом примере со скалярным произведением векторов! Это легко проверить, напрямую посчитав частные производные функции L по каждому элементу матрицы X. Проделав это[1], мы увидим, что в результате действительно получается выражение:

$$\frac{\partial \Lambda}{\partial u}(X) = \frac{\partial \Lambda}{\partial u}(S) \times \frac{\partial \sigma}{\partial u}(N) \times W^T,$$

в котором первая операция умножения выполняется поэлементно, а вторая — в соответствии с правилами умножения матриц.

Это означает, что *даже когда операции в вычислительном графе включают в себя умножение матриц, в результате которого на выходе появляется матрица, размером и формой отличающаяся от входных матриц, мы все равно можем выполнить обратный проход, используя цепное правило*. Без этого обучение моделей глубокого обучения было бы намного более сложным делом. Вы убедитесь в этом уже в следующей главе.

Код

На базе сделанного выше заключения напишем код, который, я надеюсь, позволит еще лучше понять происходящий процесс:

```
def matrix_function_backward_sum_1(X: ndarray,
                                   W: ndarray,
                                   sigma: Array_Function) -> ndarray:
```

[1] Полный расчет вы найдете в разделе «Цепное правило» приложения А.

```
    '''
    Вычисление производной функции по первому матричному аргументу.
    '''
    assert X.shape[1] == W.shape[0]

    # умножение матриц
    N = np.dot(X, W)

    # передача результата умножения матриц в функцию сигма
    S = sigma(N)

    # суммирование всех элементов
    L = np.sum(S)

    # примечание: я буду ссылаться на производные по их количеству
    # здесь, в отличие от математического представления, где мы
    # ссылались на их имена функций.
    # dLdS — just 1s
    dLdS = np.ones_like(S)

    # dSdN
    dSdN = deriv(sigma, N)

    # dLdN
    dLdN = dLdS * dSdN

    # dNdX
    dNdX = np.transpose(W, (1, 0))

    # dLdX
    dLdX = np.dot(dSdN, dNdX)
    return dLdX
```

Удостоверимся, что все работает:

```
np.random.seed(190204)
X = np.random.randn(3, 3)
W = np.random.randn(3, 2)

print("X:")
print(X)
```

```
print("L:")
print(round(matrix_function_forward_sum(X, W, sigmoid), 4))
print()
print("dLdX:")
print(matrix_function_backward_sum_1(X, W , sigmoid))
```

```
X:
[[-1.5775 -0.6664 0.6391]
 [-0.5615 0.7373 -1.4231]
 [-1.4435 -0.3913 0.1539]]
L:
2.3755

dLdX:
[[ 0.2489 -0.3748 0.0112]
 [ 0.126 -0.2781 -0.1395]
 [ 0.2299 -0.3662 -0.0225]]
```

Так как у нас, как и в предыдущем примере, `dLdX` обозначает градиент L по X, то, например, верхний левый элемент будет иметь значение $\frac{\partial \Lambda}{\partial x_{11}}(X,W) = 0.2489$.

Если предположить, что наши расчеты корректны, увеличение элемента x_{11} на 0.001 должно приводить к увеличению L на 0.01×0.2489. И мы легко можем убедиться, что это действительно так:

```
X1 = X.copy()
X1[0, 0] += 0.001

print(round(
        (matrix_function_forward_sum(X1, W, sigmoid) – \
         matrix_function_forward_sum(X, W, sigmoid)) / 0.001, 4))
0.2489
```

Как видите, мы верно посчитали наш градиент!

Визуальное представление результатов

Фактически мы рассмотрели, что происходит с элементом x_{11} при прохождении через сигмоиду, а затем сложили результаты. Внутри сигмоиды выполняется много операций, например умножение матриц, в процессе

которого происходит объединение девяти элементов матрицы X с шестью элементами матрицы W. Все эти операции можно изобразить в виде единой функции WNSL, как представлено на рис. 1.23.

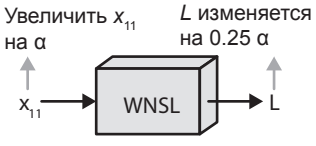


Рис. 1.23. Представление вложенных функций как единой функции WNSL

Это дифференцируемая функция, так как все входящие в нее функции дифференцируемы. Мы рассматриваем одно входное значение x_{11}, поэтому для определения градиента нужно всего лишь найти значение $\frac{dL}{dx_{11}}$.

Для наглядности построим график зависимости L от x_{11}.

Еще раз выведем нашу матрицу X и увидим, что x_{11} = `-1.5775`:

```
print("X:")
print(X)

X:
[[-1.5775 -0.6664 0.6391]
 [-0.5615 0.7373 -1.4231]
 [-1.4435 -0.3913 0.1539]]
```

На рис. 1.24 показаны значения функции L в зависимости от элемента x_{11} при постоянных значениях всех остальных элементов[1].

Как видите, при изменении компонента x_{11} на 2 функция L увеличивается примерно на 0.5 (от 2.1 до 2.6), что и дает только что рассчитанный нами наклон 0.5/2 = 0.25!

Как видите, сложные математические операции с матрицами не помешали нам правильно вычислять частные производные матрицы L по элементам матрицы X. Аналогично можно посчитать градиент L по элементам W.

[1] Полную функцию вы найдете на сайте книги (https://oreil.ly/deep-learning-github).

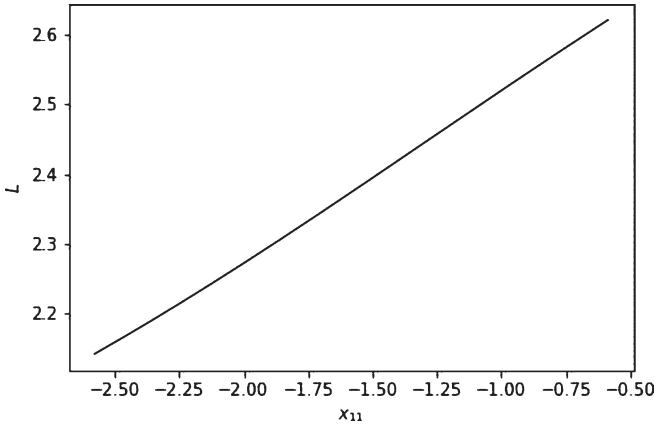


Рис. 1.24. Зависимость L от элемента x_{11}

Градиент L по W равен X^T. Но так как при вычислении такого градиента имеет значение порядок перемножаемых матриц, компонент X^T окажется в выражении первым:

$$\frac{\partial \Lambda}{\partial u}(W) = X^T \times \frac{\partial \Lambda}{\partial u}(S) \times \frac{\partial \sigma}{\partial u}(N).$$

Соответственно в коде после строки `dNdW = np.transpose(X, (1, 0))` будет идти строка `dLdW = np.dot(dNdW, dSdN)`, а не `dLdX = np.dot(dSdN, dNdX)`, как раньше.

Заключение

Материал этой главы научит вас понимать вложенные математические функции и принцип их работы путем представления их в виде соединенных линиями последовательных блоков, каждый из которых представляет собой отдельную функцию, входящую в состав сложной. В частности, вам не составит труда написать код для вычисления частных производных таких функций по любому аргументу, даже когда в качестве аргументов выступают двумерные объекты `ndarray`, и проверить правильность сделанных расчетов. Все это позволит вам строить и обучать нейронные сети и модели глубокого обучения с нуля в последующих главах. Вперед!

ГЛАВА 2

Основы глубокого обучения

В главе 1 вы познакомились с математическими элементами, составляющими базу глубокого обучения: вложенными, непрерывными и дифференцируемыми функциями. Вы научились представлять их в виде вычислительных графов, каждый узел которых соответствует одной простой функции. В частности, я показал способ оценки производных вложенных функций: достаточно оценить производные всех составляющих такой функции при заданных аргументах и перемножить полученные результаты. Это возможно благодаря цепному правилу. Рассмотренные в предыдущей главе примеры позволили убедиться, что такой подход корректно работает в случае функций, на вход которых подаются объекты `ndarray` из библиотеки NumPy, и такие же объекты мы видим на выходе.

Более того, был рассмотрен даже пример функции с несколькими объектами `ndarray` в качестве аргументов, которая осуществляла *умножение матриц* и давала на выходе матрицу другого размера, то есть фактически меняла форму входных данных. На вход мы подавали матрицу X (массив `ndarray` размером B × N) и матрицу W (массив `ndarray` размером N × M), а на выходе получали массив размером B × M. Вы могли убедиться, что когда умножение матриц `v` (X, W) происходит внутри вложенной функции, для оценки производных по входным данным можно использовать простое выражение. В частности, вместо $\frac{\partial \nu}{\partial u}(W)$ можно подставить X^T, а вместо $\frac{\partial \nu}{\partial u}(X)$ — W^T.

Пришло время применить эту информацию на практике. В этой главе мы:

1. Построим на базе этих элементов модель линейной регрессии.
2. Посмотрим на процесс обучения этой модели.
3. Расширим эту модель до однослойной нейронной сети.

В главе 3 все это позволит нам сразу перейти к построению моделей глубокого обучения.

Но прежде, чем мы начнем, хотел бы напомнить, что представляет собой такой способ машинного обучения, как обучение с учителем.

Обучение с учителем

В общем виде машинное обучение можно описать как построение алгоритмов, выявляющих или «изучающих» присутствующие в данных *закономерности*; обучение с учителем представляет собой подраздел машинного обучения, специализирующийся на поиске взаимосвязей между *характеристиками уже подготовленных данных*[1].

В этой главе мы рассмотрим типичную для обучения с учителем задачу: поиск связи между характеристиками дома и его стоимостью. Очевидно, что цена дома зависит от таких характеристик, как количество комнат, их метраж, близость к школам и другой социальной инфраструктуре, а также от того, хотят ли люди приобрести дом в собственность или предпочитают аренду. Все эти данные *уже измерены*, и цель обучения с учителем состоит в выявлении корреляции между ними.

Под «измеренными» данными я имею в виду представление каждой характеристики в виде числа. Это легко сделать, когда речь заходит о таких свойствах, как количество или площадь комнат. Но на цену дома будут влиять и, к примеру, взятые с сайта TripAdvisor отзывы о районе, где дом располагается. Тут уже возникает вопрос, каким образом перевести эти неструктурированные данные, то есть обычную речь, в цифровое представление. Кроме того, остаются еще и неоднозначные характеристики, например ценность дома. Впрочем, очевидно, что эту конкретную характеристику можно выразить через такой параметр, как цена[2].

---

[1] Другая разновидность машинного обучения — обучение без учителя — ищет взаимосвязи в данных без предварительной подготовленной обучающей выборки.

[2] Хотя при решении такой задачи в реальности с этим параметром возникнут определенные сложности. Например, как определить цену дома, который много десятилетий не выставлялся на продажу? Впрочем, в книге мы будем рассматривать примеры, в которых числовое представление данных не вызывает подобных сложностей.

После преобразования всех «характеристик» в числа нужно выбрать структуру для их представления. В машинном обучении принята почти универсальная структура, которая, как оказалось, облегчает процесс вычислений. Это представление одного наблюдения, например одного дома, в виде строки данных. Такие строки складываются друг на друга, формируя «пакеты» данных, которые и передаются моделям в виде двумерных массивов `ndarray`. Свои прогнозы модели выдают также в виде двумерных объектов `ndarray`, по одному прогнозу на наблюдение.

При этом длина каждой строки в массиве определяется количеством признаков данных. В общем случае одна характеристика может давать целый набор признаков. Тогда мы говорим, что точка данных относится к одной или нескольким *категориям*. Например, дом может быть облицован красным кирпичом, коричневым кирпичом или плиткой из сланца[1]. Процесс извлечения признаков из того, что мы воспринимаем как характеристики наблюдений, называется *проектированием признаков*. Детальное рассмотрение этого процесса выходит за рамки этой книги. В данной главе мы решим задачу, в которой для каждого наблюдения будет 13 характеристик, и каждой из них мы поставим в соответствие один числовой признак.

Как я уже говорил, обучение с учителем в конечном счете сводится к выявлению взаимосвязей между характеристиками данных. На практике мы выбираем одну характеристику и пытаемся предсказать ее поведение. Такая характеристика называется *целевой*. Выбор целевой характеристики зависит от поставленной задачи. Например, если нужно *описать* взаимосвязь между ценами домов и количеством комнат в них, в качестве целевого признака можно выбрать как цену, так и количество комнат. В обоих случаях обученная модель опишет взаимосвязь между этими двумя характеристиками. Если же наша задача научить модель *оценивать ранее не выставлявшиеся на продажу дома*, целевым признаком должна стать цена, потому что именно ее будет предсказывать обученная модель исходя из остальных признаков.

Схематично принцип обучения с учителем показан на рис. 2.1. На рисунке представлен как самый высокий уровень поиска взаимосвязей между данными, так и самый низкий уровень классификации этих связей между целевыми и исходными признаками после обучения модели.

---

[1] Большинству читателей, скорее всего, известно, что такие признаки называют категориальными.

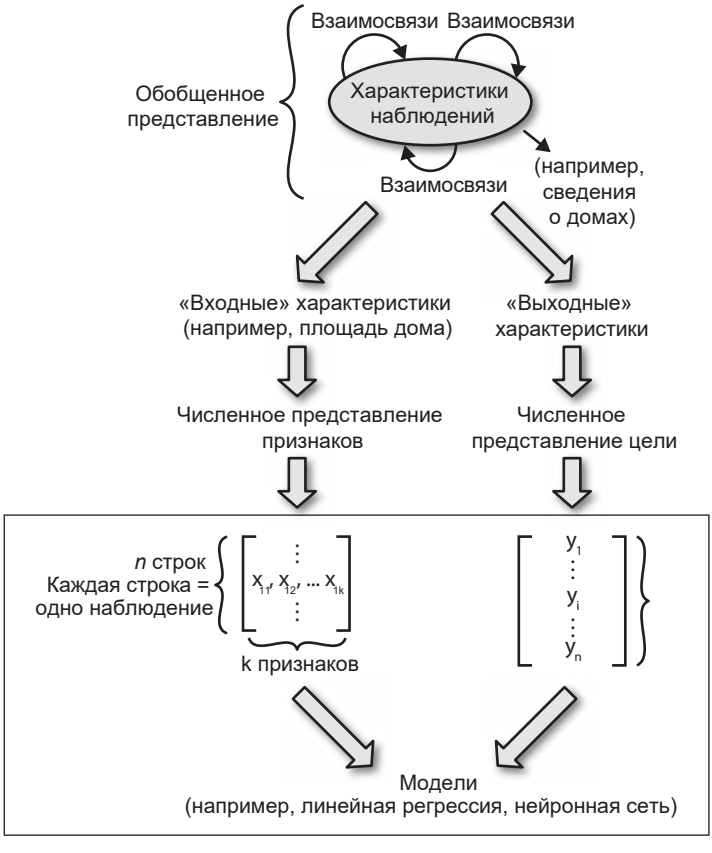


Рис. 2.1. Схема процесса обучения с учителем

Практически все время мы будем заниматься тем, что показано в нижней части рис. 2.1. При этом зачастую собрать правильный набор данных, корректно определить задачу и выполнить проектирование признаков намного сложнее, чем осуществить моделирование. Но эта книга посвящена именно моделям глубокого обучения и принципам их работы, поэтому сосредоточимся именно на этой теме.

Алгоритмы обучения с учителем

Теперь, когда вы получили общее представление о назначении алгоритмов для обучения с учителем, напомню, что эти алгоритмы пред-

ставляют собой просто вложенные функции. С этой точки зрения цель обучения с учителем состоит в *поиске* функции, которая принимает в качестве входных данных массивы `ndarray` и их же дает на выходе. Эта функция должна преобразовывать массивы предоставленных ей признаков в массив, значения которого как можно точнее аппроксимируют целевые данные.

Мы будем представлять данные в виде матрицы X, состоящей из n строк, каждая из которых содержит наблюдение с k признаками, причем все эти признаки числовые. Фактически каждое наблюдение представляет собой вектор $x_i = \begin{bmatrix} x_{i1} & x_{i2} & x_{i3} & \dots & x_{ik} \end{bmatrix}$, и совокупно все эти наблюдения составляются в пакет. Вот как будет выглядеть пакет из трех наблюдений:

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \cdots & x_{3k} \end{bmatrix}.$$

Каждому пакету наблюдений будет сопоставлен набор целей. Элемент такого пакета представляет собой номер цели для соответствующего наблюдения. Его можно представить как вектор-столбец:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Наша задача в рамках обучения с учителем будет состоять в построении функции, принимающей пакеты наблюдений, представленные в виде структуры X_{batch}, и продуцирующей векторы p_i (которые мы интерпретируем как «предсказания»), достаточно близкие к целевым значениям y_i.

По сути, мы уже готовы приступить к построению первой модели на базе реальных данных. Начнем мы с такой распространенной модели, как линейная регрессия.

Линейная регрессия

Линейную регрессию часто представляют следующей формулой:

$$y_i = \beta_0 + \beta_1 \times x_1 + \ldots + \beta_n \times x_k + \epsilon.$$

Фактически мы предполагаем, что каждую целевую переменную можно представить как линейную комбинацию k признаков и константы β_0, которая корректирует «базовое» значение прогноза (в частности, это прогноз при нулевом значении всех признаков).

Из определения непонятно, каким образом мы будем писать код и обучать эту модель. Поэтому нужно выразить модель в виде функций, с которыми мы работали в главе 1. И проще всего начать со схемы.

Визуализация

Как представить линейную регрессию в виде вычислительного графа? На уровне отдельных элементов модель выглядит так: мы вычисляем произведение каждой пары x_i и w_i, а затем суммируем полученные результаты, как показано на рис. 2.2.

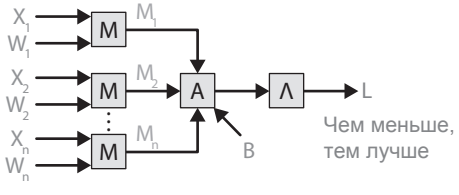


Рис. 2.2. Алгоритм линейной регрессии, представленный в виде отдельных операций умножения и сложения

Но в главе 1 мы видели, как представление этой операции через умножение матриц позволяет не только кратко записать функцию, но и корректно рассчитать ее производную, что подходит для последующего обучения модели.

Но как это сделать? Для начала рассмотрим сценарий, в котором отсутствует свободный член (в выражении он обозначен β_0). Результат работы модели линейной регрессии можно представить как скалярное произведение каждого вектора наблюдений $x_i = \begin{bmatrix} x_1 & x_2 & x_3 & \dots & x_k \end{bmatrix}$ и вектора параметров, который мы назовем W:

$$W = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \dots \\ w_k \end{bmatrix}.$$

В этом случае предсказание выражается формулой:

$$p_i = x_i \times W = w_1 \times x_{i1} + w_2 \times x_{i2} + \ldots + w_k \times x_{ik}.$$

Как видите, в случае модели линейной регрессии «генерацию предсказания» можно представить с помощью всего одной операции: скалярного произведения.

Для пакета наблюдений будет использоваться другая операция: *умножение матриц*. Например, если есть пакет из трех наблюдений, записанный в виде матрицы:

$$X_{batch} = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \cdots & x_{3k} \end{bmatrix},$$

его умножение на вектор W даст набор предсказаний:

$$P_{batch} = X_{batch} \times W = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \cdots & x_{3k} \end{bmatrix} \times \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \ldots \\ w_k \end{bmatrix} =$$

$$= \begin{bmatrix} x_{11} \times w_1 & x_{12} \times w_2 & x_{13} \times w_3 & \ldots & x_{1k} \times w_k \\ x_{21} \times w_1 & x_{22} \times w_2 & x_{23} \times w_3 & \cdots & x_{2k} \times w_k \\ x_{31} \times w_1 & x_{32} \times w_2 & x_{33} \times w_3 & \cdots & x_{3k} \times w_k \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Используем этот факт, а также знакомые вам по предыдущей главе принципы взятия производных, чтобы обучить модель.

Обучение модели

Что означает «обучение» модели? Говоря простыми словами, модели[1] берут данные, каким-то образом комбинируют их с *параметрами* и дают предсказания. Например, модель линейной регрессии берет данные в виде матрицы X и набор параметров в виде вектора W и, перемножив их, дает вектор предсказаний:

[1] По крайней мере, те модели, которые будут рассматриваться в этой книге.

$$P_{batch} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Однако для обучения модели требуется информация о точности полученных прогнозов. Чтобы ее получить, добавим вектор целевых значений y_{batch}, связанный с подаваемым на вход модели набором наблюдений X_{batch}, и посчитаем величину отклонения сделанных прогнозов от ожидаемых. Такая мера количества ошибок называется функцией потерь. Часто для этой цели используют среднеквадратическую ошибку (mean squared error, MSE):

$$MSE\left(P_{batch}, y_{batch}\right) = MSE\left(\begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}, \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right) = \frac{\left(y_1 - p_1\right)^2 + \left(y_2 - p_2\right)^2 + \left(y_3 - p_3\right)^2}{3}.$$

Обозначим полученное число L. Это крайне важный параметр, ведь теперь мы можем по методике, которую я дал в главе 1, посчитать его *градиент* по каждому элементу вектора W. Этими данными мы воспользуемся для *обновления всех элементов вектора W в направлении, уменьшающем значение L*. Многократное повторение этой процедуры и называется «обучением» модели. Ниже вы убедитесь, что такой подход действительно работает. Чтобы лучше понять, как вычислить нужные градиенты, видоизменим диаграмму с рис. 2.2.

Линейная регрессия: еще одна визуализация и математическая модель

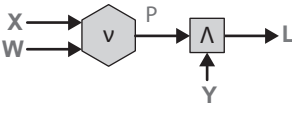


Рис. 2.3. Вычислительный граф для уравнения линейной регрессии. Жирным выделены входные данные, буква W обозначает веса

Представим функцию потерь как набор вложенных функций:

$$L = \Lambda\left(\nu\left(X, W\right), Y\right).$$

Свободный член

Благодаря диаграмме несложно понять, как добавить в нашу модель свободный член. Ведь это еще один элемент диаграммы, отвечающий за «смещение» (bias), как показано на рис. 2.4.

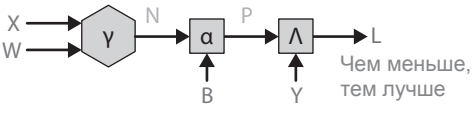


Рис. 2.4. Вычислительный граф для функции линейной регрессии с добавленным членом смещения

Прежде чем переходить к написанию кода, нужно понять, что поменялось в математическом представлении после добавления смещения. К уже знакомому нам скалярному произведению в каждом элементе p_i вектора предсказания будет прибавляться константа b:

$$P_{batch\_with\_bias} = x_i \times W + b = \begin{bmatrix} x_{11} \times w_1 & x_{12} \times w_2 & x_{13} \times w_3 & \dots & x_{1k} \times w_k \\ x_{21} \times w_1 & x_{22} \times w_2 & x_{23} \times w_3 & \cdots & x_{2k} \times w_k \\ x_{31} \times w_1 & x_{32} \times w_2 & x_{33} \times w_3 & \cdots & x_{3k} \times w_k \end{bmatrix} = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \end{bmatrix}.$$

Обратите внимание, что поскольку в линейной регрессии всего одно пересечение линии оценки, к каждому наблюдению добавляется *одно и то же* значение смещения. В следующем разделе мы поговорим о том, как это влияет на вычисление производных.

Код

Теперь мы готовы написать функцию, которая принимает данные наших наблюдений X_{batch}, целевые значения y_{batch} и дает на выходе прогноз и функцию потерь. Напомню, что в случае вложенных функций вычисление производных происходит в два этапа. Сначала во время «прямого прохода» входные данные последовательно пропускаются через набор операций с сохранением полученного результата. Затем следует «обратный проход», во время которого сохраненные результаты применяются для вычисления соответствующих производных.

Результаты прямого прохода мы будем сохранять в словарь. Чтобы отделить их от параметров (которые также потребуются во время обратного прохода), поместим параметры в отдельный словарь:

```
def forward_linear_regression(X_batch: ndarray,
                              y_batch: ndarray,
                              weights: Dict[str, ndarray])
                              -> Tuple[float, Dict[str, ndarray]]:
    '''
    Прямой проход для линейной регрессии.
    '''
    # проверяем совпадение размеров X и y
    assert X_batch.shape[0] == y_batch.shape[0]

    # проверяем допустимость умножения матриц
    assert X_batch.shape[1] == weights['W'].shape[0]

    # проверяем, что B это объект ndarray размером 1x1
    assert weights['B'].shape[0] == weights['B'].shape[1] == 1

    # вычисления
    N = np.dot(X_batch, weights['W'])

    P = N + weights['B']

    loss = np.mean(np.power(y_batch – P, 2))

    # сохранение информации, полученной во время прямого прохода
    forward_info: Dict[str, ndarray] = {}
    forward_info['X'] = X_batch
    forward_info['N'] = N
    forward_info['P'] = P
    forward_info['y'] = y_batch

    return loss, forward_info
```

Теперь все готово к обучению модели. Поговорим о том, что это такое и как реализуется.

Обучение модели

Воспользуемся инструментами и методами, с которыми мы познакомились в прошлой главе, и вычислим $\frac{\partial L}{\partial w_i}$ для всех элементов w_i вектора W, а также $\frac{\partial L}{\partial b}$. Для этого выполним обратный проход, во время которого оценим частные производные вложенных функций при заданных входных значениях, а затем перемножим полученные результаты.

Диаграмма для операции вычисления градиентов

Концепция того, что мы хотим получить, представлена на рис. 2.5.

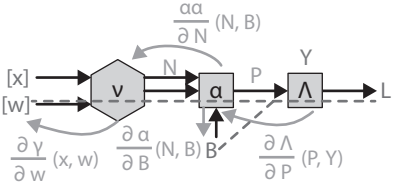


Рис. 2.5. Обратный проход по вычислительному графу линейной регрессии

Еще раз напомню, что мы просто будем вычислять частные производные каждой из вложенных функций, двигаясь изнутри наружу, а затем оценивать их для значений, полученных во время прямого прохода. После чего останется получить произведение результатов.

Математическое представление

Согласно рис. 2.5, мы хотим получить следующее произведение:

$$\frac{\partial \Lambda}{\partial P}(P, Y) \times \frac{\partial \alpha}{\partial N}(N, B) \times \frac{\partial \nu}{\partial W}(X, W).$$

Первым делом вычислим элемент $\frac{\partial \Lambda}{\partial P}(P, Y) \partial \Lambda$. Так как $\Lambda(P, Y) = (Y - P)^2$, для всех элементов векторов Y и P получим:

$$\frac{\partial \Lambda}{\partial P}(P, Y) = -1 \times (2 \times (Y - P)).$$

Вот как это выражение выглядит в виде кода:

```
dLdP = -2 * (Y – P)
```

Следующим идет элемент, содержащий матрицы: $\frac{\partial \alpha}{\partial N}(N, B)$. Но так как под обозначением α скрывается всего лишь свободный член, можно применить уже знакомую по предыдущей главе логику: маленькое увеличение любого элемента матрицы N приведет к такому же увеличению вектора P, который равен $\alpha(N, B) = N + B$. Соответственно производная рассматриваемого элемента будет матрицей, заполненной единицами, по форме совпадающей с матрицей N.

Код для такой производной выглядит очень просто:

```
dPdN = np.ones_like(N)
```

Ну и наконец, компонент $\frac{\partial \nu}{\partial W}(X, W)$. В конце предыдущей главы мы установили, что при вычислении производных вложенных функций, в случае когда внутри осуществляется умножение матриц, можно провести следующую замену:

$$\frac{\partial \nu}{\partial W}(X, W) = X^T.$$

В виде кода это выглядит так:

```
dNdW = np.transpose(X, (1, 0))
```

Сделаем то же самое для свободного члена. Так как мы его просто прибавляем, его частная производная будет равна 1:

```
dPdB = np.ones_like(weights['B'])
```

Осталось получить произведение всех производных, проследив за правильным порядком умножения содержащих матрицы компонентов `dNdW` и `dNdX`.

Код для вычисления градиента

Напомню, что нужно взять все, что мы вычислили и ввели во время прямого прохода (на рис. 2.5 эти компоненты обозначены как X, W, N, B, P и y), и рассчитать частные производные $\frac{\partial \Lambda}{\partial W}$ и $\frac{\partial \Lambda}{\partial B}$. Это реализует приведенный ниже код. Входные данные W и B представлены в виде словарей, первый из которых, содержащий веса, называется `weights`, а второй, содержащий остальные параметры, — `forward_info`:

```
def loss_gradients(forward_info: Dict[str, ndarray],
                   weights: Dict[str, ndarray]) -> Dict[str, ndarray]:
    '''
    Вычисление dLdW и dLdB для модели линейной регрессии.
    '''
    batch_size = forward_info['X'].shape[0]

    dLdP = -2 * (forward_info['y'] - forward_info['P'])

    dPdN = np.ones_like(forward_info['N'])

    dPdB = np.ones_like(weights['B'])

    dLdN = dLdP * dPdN

    dNdW = np.transpose(forward_info['X'], (1, 0))

    # умножение матриц, в котором первым идет компонент
    # dNdW (см. примечание в конце предыдущей главы)
    dLdW = np.dot(dNdW, dLdN)

    # суммирование по измерению, представляющему размер набора
    # (объяснение ниже)
    dLd B = (dLdP * dPdB).sum(axis=0)

    loss_gradients: Dict[str, ndarray] = {}
    loss_gradients['W'] = dLdW
    loss_gradients['B'] = dLdB

    return loss_gradients
```

Как видите, мы просто вычисляем все производные и перемножаем их, проследив за порядком следования матриц[1]. Вскоре вы убедитесь, что это действительно работает. Впрочем, после того как в предыдущей главе мы интуитивно доказали применимость цепного правила, удивления быть не должно.

Вычисленный градиент функции потерь мы сохраняем в словарь: веса — как ключи, а параметры, увеличивающие влияние весов на потери, — как значения. Словарь весов структурирован аналогичным способом. Следовательно, перебирать веса модели можно следующим образом:

```
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Именно такой способ хранения указанных параметров ничем не обусловлен. Можно сохранить их и другим способом, просто в этом случае они будут просматриваться в другом порядке и ссылаться на них мы будем по-другому.

Обучение модели

Теперь нужно создать цикл из следующих операций:

1. Выбор набора данных.
2. Прямой проход модели.
3. Обратный проход модели с применением данных, полученных во время прямого прохода.
4. Применение вычисленных градиентов для обновления весов.

Репозиторий Jupyter Notebook к этой главе (https://oreil.ly/2TDV5q9) содержит код функции `train`, предназначенной для обучения нашей модели. Он реализует все вышеуказанные шаги и, кроме того, перемешивает данные, чтобы они подавались в функцию в случайном порядке. Вот ключевые строки кода, которые повторяются внутри цикла `for`:

[1] Кроме того, мы суммируем элементы `dLdB` вдоль оси 0; зачем это нужно, я подробно объясню чуть позже.

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)

for key in weights.keys(): # 'weights' и 'loss_grads' имеют
                           # одинаковые ключи
    weights[key] -= learning_rate * loss_grads[key]
```

После этого обучающая функция запускается определенное количество раз на всем тренировочном наборе данных:

```
train_info = train(X_train, y_train,
                   learning_rate = 0.001,
                   batch_size=23,
                   return_weights=True,
                   seed=80718)
```

Обучающая функция возвращает кортеж `train_info`, один из элементов которого — это параметры или веса, показывающие, как в процессе тренировки изменилась модель.

В глубоком обучении термины «параметры» и «веса» используются как синонимы. В этой книге они тоже равнозначны.

Оценка точности модели

Но как понять, насколько корректно построенная модель раскрывает взаимосвязи в данных? Обучающая выборка представляет собой лишь часть от общей совокупности данных. А перед нами строит задача построить модель, выявляющую взаимосвязи во всей совокупности, несмотря на ограниченность тренировочных данных.

Всегда существует опасность, что модель начнет выбирать взаимосвязи, существующие в обучающей выборке, но отсутствующие в совокупности данных. Представьте, что в выборку случайно попали дома с тремя ванными комнатами, облицованные желтым сланцем, которые предлагаются по относительно низкой цене. Нейронная сеть обнаружит эту закономерность, хотя в общей совокупности данных она не наблюдается. Это

явление называется *переобучением* (overfitting). Как понять, что у модели может быть подобный недостаток?

Чтобы избежать такой ситуации, данные, предназначенные для обучения модели, разбивают на *обучающий набор* (training set) и *тестовый набор* (testing set). Первый используется для обучения модели (то есть для итеративного обновления весов), после чего точность работы модели оценивается на тестовых данных.

В основе этого подхода лежит простая логика. Если модель смогла обнаружить взаимосвязи, которые работают и на *остальной части обучающей выборки* (то есть на всем наборе данных), велика вероятность, что они присутствуют и в *общей совокупности данных*.

Код

Давайте попробуем оценить нашу модель на тестовом наборе. Первым делом напишем функцию, генерирующую предсказания, обрезав уже знакомую нам функцию `forward_loss`:

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]):
    '''
    Генерация предсказаний для модели линейной регрессии.
    '''
    N = np.dot(X, weights['W'])

    return N + weights['B']
```

Теперь возьмем веса, которые возвращает обучающая функция, и напишем:

```
preds = predict(X_test, weights) # weights = train_info[0]
```

Насколько хороши эти предсказания? Пока ответа на этот вопрос нет. Ведь мы еще не знаем, работает ли выбранный подход — определение модели как набора операций и ее обучение путем итеративной корректировки параметров, для которой мы вычисляем частные производные функции потерь по различным параметрам. Так что будет здорово, если окажется, что все это хоть как-то работает.

Для проверки результатов построим график зависимости предсказанных значений от фактических. В идеальном случае все точки должны оказаться на прямой линии с наклоном в 45 градусов. Реальный результат показан на рис. 2.6.

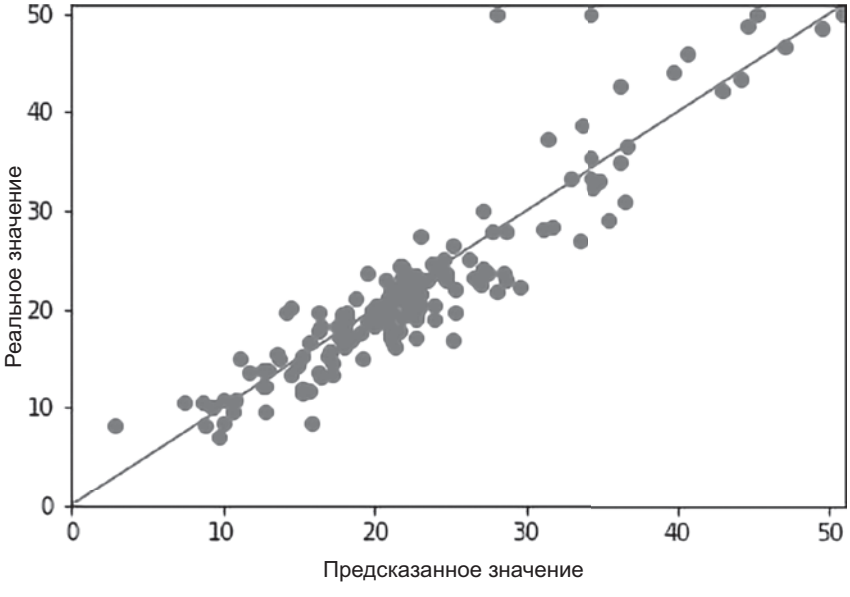


Рис. 2.6. Сравнение предсказанных и действительных значений для модели линейной регрессии

График выглядит вполне приемлемо, поэтому количественная оценка точности работы модели имеет смысл. Это можно сделать двумя способами:

- Вычислить абсолютное значение среднего расстояния между предсказаниями модели и фактическими значениями. Эту метрику называют *средним модулем отклонения* (mean absolute error, MAE):

```
def mae(preds: ndarray, actuals: ndarray):
    '''
    Вычисление среднего линейного отклонения.
    '''
    return np.mean(np.abs(preds – actuals))
```

- Вычислить средний квадрат расстояния между предсказаниями модели и фактическими значениями. Эту метрику называют *корнем из среднего квадрата отклонения* (root mean squared error, RMSE):

```
def rmse(preds: ndarray, actuals: ndarray):
    '''
    Вычисление корня из среднего квадрата отклонения.
    '''
    return np.sqrt(np.mean(np.power(preds – actuals, 2)))
```

Для рассматриваемой модели были получены значения:

```
Mean absolute error: 3.5643
Root mean squared error: 5.0508
```

Корень из среднего квадрата отклонения — распространенная метрика, поскольку она находится в одном масштабе с целевыми значениями. Разделив это число на среднее от целевого показателя, мы увидим, насколько полученный прогноз далек от фактического значения. В рассматриваемом случае среднее значение параметра `y_test` составляет `22.0776`. Соответственно прогнозы цен на недвижимость в этой модели в среднем отклоняются от фактических на $5.0508/22.0776 \cong 22.9\%$.

Хорошая ли это точность? В репозиторий Jupyter Notebook к этой главе (https://oreil.ly/2TDV5q9) добавлен результат, полученный на этом же наборе данных для модели линейной регрессии, реализованной средствами Sci-Kit Learn — самой популярной библиотеки Python для машинного обучения. Эта модель дает средний модуль отклонения `3.5666`, а корень из среднего квадрата отклонения `5.0482`, что практически совпадает с результатами, которые дала наша модель, построенная на базе математических формул. Это показывает, что такой подход вполне применим для построения моделей машинного обучения. Чуть позже мы расширим его на области нейронных сетей и глубокого обучения.

Определение самого важного признака

Перед началом моделирования признаки были масштабированы так, чтобы в результате среднее значение стало равным 0 и среднеквадратическое отклонение — 1. Зачем это нужно, подробно расскажу в главе 4. В случае линейной регрессии такое преобразование позволяет интерпретировать абсолютные значения коэффициентов как меру важности признаков.

Чем выше коэффициент, тем важнее признак. Вот коэффициенты в рассматриваемом случае:

```
np.round(weights['W'].reshape(-1), 4)
```

```
array([-1.0084, 0.7097, 0.2731, 0.7161, -2.2163, 2.3737, 0.7156,
       -2.6609, 2.629, -1.8113, -2.3347, 0.8541, -4.2003])
```

Последний коэффициент больше всего по модулю, соответственно он и является самым важным. Сравнение этого признака с целевым значением показано на рис. 2.7.

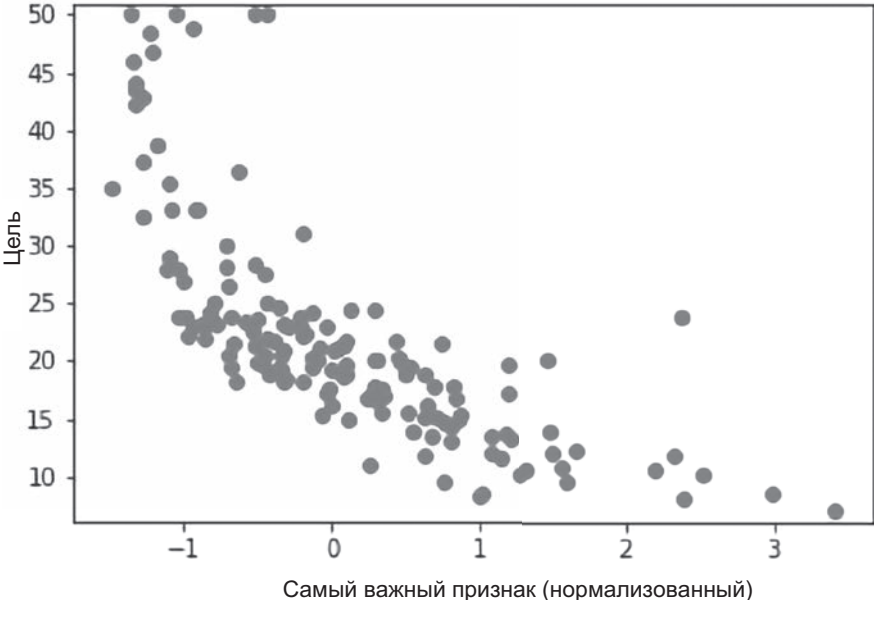


Рис. 2.7. Сравнение самого важного признака с целевым значением в модели линейной регрессии

Мы видим, что по мере роста значения признака значение целевого параметра уменьшается, причем это нелинейная зависимость. Изменение целевого параметра при изменении значения признака с –2 до –1 и с 1 до 2 будет *разным*. Позднее мы еще вернемся к этому моменту.

На рис. 2.8 к этому графику добавлено соотношение между самым важным признаком и *предсказаниями модели*. Чтобы получить это соотношение,

мы пропустили через обученную модель данные, обработанные следующим образом:

- Все признакам было присвоено их среднее значение.
- За 40 итераций проведена интерполяция значений самого важного признака от –1.5 до 3.5, что примерно соответствует диапазону этого признака после масштабирования.

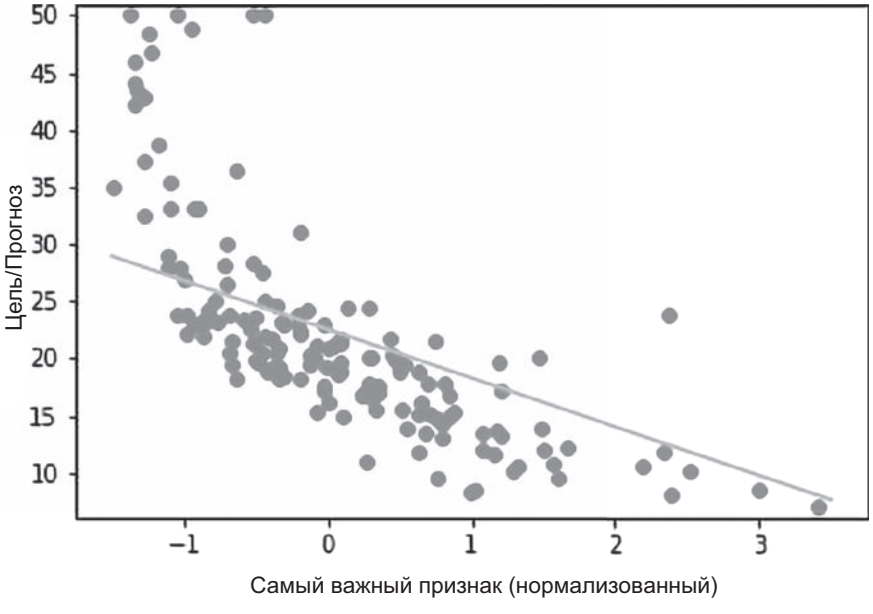


Рис. 2.8. Сравнение самого важного признака с целевым значением и предсказаниями, данными моделью линейной регрессии

График наглядно показывает ограниченность модели линейной регрессии. Несмотря на нелинейное соотношение между самым важным признаком и целью, в результате обучения извлекается только линейная связь. Такое поведение обусловлено внутренней структурой модели.

Получается, что для определения более сложных, нелинейных закономерностей требуется другая модель. Но как ее построить? Ответом на этот вопрос станут нейронные сети.

Основы нейронных сетей

Я показал, как на базе теоретической информации построить и обучить модель линейной регрессии. Но как, используя аналогичную цепочку рассуждений, спроектировать модель, которая сможет обнаруживать нелинейные взаимосвязи? Давайте создадим *много* моделей линейной регрессии, чьи результаты работы пропустим через нелинейную функцию, после чего применим еще одну модель линейной регрессии, которая и даст прогноз. Как вы вскоре увидите, вычислять градиенты для этой более сложной модели можно тем же способом, что и для модели линейной регрессии.

Шаг 1. Набор моделей линейной регрессии

Как выглядит создание «набора линейных регрессий»? В модели линейной регрессии происходит умножение матриц с наборами параметров. Матрица данных X, имеющая форму `[размер_пакета, число_признаков]`, умножается на матрицу весов W формы `[число_признаков, 1]`, в результате мы получаем матрицу формы `[размер_пакета, 1]`, то есть *взвешенную сумму* исходных признаков для каждого наблюдения в пакете. Поэтому набор линейных регрессий эквивалентен умножению матрицы входных данных на матрицу весов, имеющую форму `[размер_пакета, число_выводов]`, что даст *для каждого наблюдения* `число_выводов` взвешенных сумм исходных признаков.

Что это за взвешенные суммы? Их можно рассматривать как «извлеченные признаки», то есть как комбинации оригинальных признаков, способствующие последующему процессу обучения и обеспечивающие точность прогноза. Сколько таких признаков нужно в рассматриваемом случае? Так как оригинальных признаков у нас 13, создадим такое же число извлеченных признаков.

Шаг 2. Нелинейная функция

Все эти взвешенные суммы подаются на вход какой-то нелинейной функции. Первой мы опробуем в этой роли знакомую по главе 1 функцию `sigmoid`, показанную на рис. 2.9.

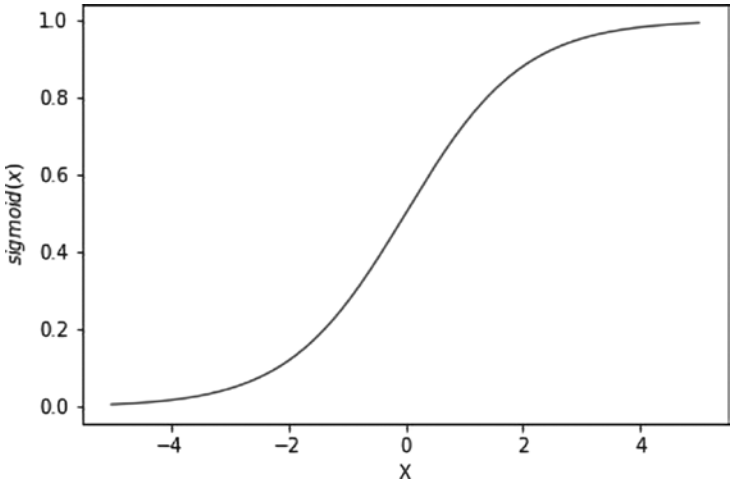


Рис. 2.9. Сигмоидная функция в диапазоне x от –5 до 5

Почему мы не взяли, например, квадратичную функцию $f(x) = x^2$? Во-первых, нам нужна *монотонная* функция, «сохраняющая» информацию о знаке вводимых значений. Предположим, две из наших линейных регрессий дали значения –3 и 3. Квадратичная функция в обоих случаях даст значение 9, то есть информация о разнице знаков в исходных данных будет «потеряна».

Во-вторых, сигмоида — нелинейная функция, что позволит нашей нейронной сети извлекать нелинейные взаимосвязи между признаками и целью.

Наконец, производную сигмоиды можно легко выразить через саму функцию, что выгодно с точки зрения вычислений:

$$\frac{\partial \sigma}{\partial u}(x) = \sigma(x) \times (1 - \sigma(x)) \cdot$$

Вы в этом убедитесь, как только мы дойдем до обратного прохода по нашей нейронной сети.

Шаг 3. Еще одна линейная регрессия

Сигмоидная функция преобразует 13 извлеченных признаков таким образом, что их значения оказываются в диапазоне от 0 до 1. После чего

они используются как входные данные для обычной функции линейной регрессии.

Для обучения полученной функции мы повторим действия, выполнявшиеся с моделью линейной регрессии, а именно воспользуемся цепным правилом, чтобы выяснить, насколько увеличение весов увеличит (или уменьшит) потери, и обновим веса в направлении, уменьшающем потери. Через некоторое количество итераций (по крайней мере, мы на это надеемся) мы получим более точную модель, которая «выучила» нелинейную взаимосвязь между признаками и целью.

Теперь посмотрим, как это выглядит схематично.

Визуализация

Новая версия модели представлена на рис. 2.10.

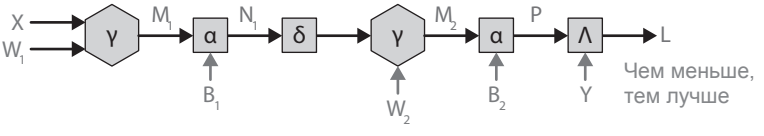


Рис. 2.10. Представление шагов 1–3 в виде вычислительного графа

Как и прежде, все начинается с умножения и сложения матриц. Первая матрица, которую мы используем для преобразования входных признаков, называется матрицей *весов* (weight matrix), а вторая матрица, которую мы добавляем к полученному набору признаков, называется *смещениями* (bias). Поэтому мы обозначили их W_1 и B_1 соответственно.

Результат этих операций передается в сигмоидную функцию, после чего процесс повторяется с *другим* набором весов и смещений, которые мы обозначим как W_2 и B_2. Эта методика и даст в итоге предсказание P.

Еще визуализация?

Позволяет ли подобное пошаговое описание получить интуитивное представление о происходящем? Этот вопрос касается основной темы книги: для полного понимания нейронных сетей нужно рассмотреть несколько представлений, каждое из которых показывает свой аспект их работы.

Диаграмма с рис. 2.10 не помогает интуитивно понять «структуру» сети, зато наглядно показывает способ обучения модели. Во время обратного прохода оцениваются значения частных производных всех составляющих функции для существующих входных данных, после чего простым умножением мы считаем градиенты потерь по каждому из весов. Эту процедуру вы уже видели в главе 1.

Существует и другой, более стандартный способ представления нейронной сети. Исходные признаки рисуются в виде окружностей. В нашем случае их будет 13. Вторые 13 окружностей будут обозначать результат работы набора линейных регрессий и сигмоиды. Причем каждая окружность из второго набора — функция 13 оригинальных признаков, поэтому из каждой окружности будет выходить 13 линий[1].

При этом 13 выходных данных используются для генерации единственного предсказания, которое мы обозначим еще одной окружностью, как показано на рис. 2.11, демонстрирующем окончательный вариант схемы[2].

Тем, кто уже что-то читал о нейронных сетях, скорее всего, знакома подобная схема. Такое представление сразу показывает, что перед вами нейронная сеть и сколько у нее слоев. Но по нему невозможно понять математическую основу происходящего и способы обучения сети.

Поэтому при всей важности этой диаграммы я добавил ее в первую очередь, чтобы продемонстрировать *связь* между ней и основным способом представления нейронных сетей: в виде соединенных линиями блоков, представляющих функции, которые выполняются во время прямого прохода, и производные, которые вычисляются во время обратного прохода. В следующей главе я покажу, как напрямую превратить эти рисунки в код, записывая каждую функцию как класс `Python`, наследующий от базового класса `Operation`.

Код

Этот код имеет ту же структуру, что и код для более простой модели линейной регрессии, которую мы рассматривали выше. Веса задаются

[1] Разумеется, возможна ситуация, когда выходные данные связаны только с частью исходных признаков; так происходит, например, в сверточных нейронных сетях.

[2] На самом деле это не совсем полная схема: на ней отсутствуют 169 линий, обозначающих связи между первыми двумя «слоями» объектов. Впрочем, нарисованного достаточно для иллюстрации идеи.

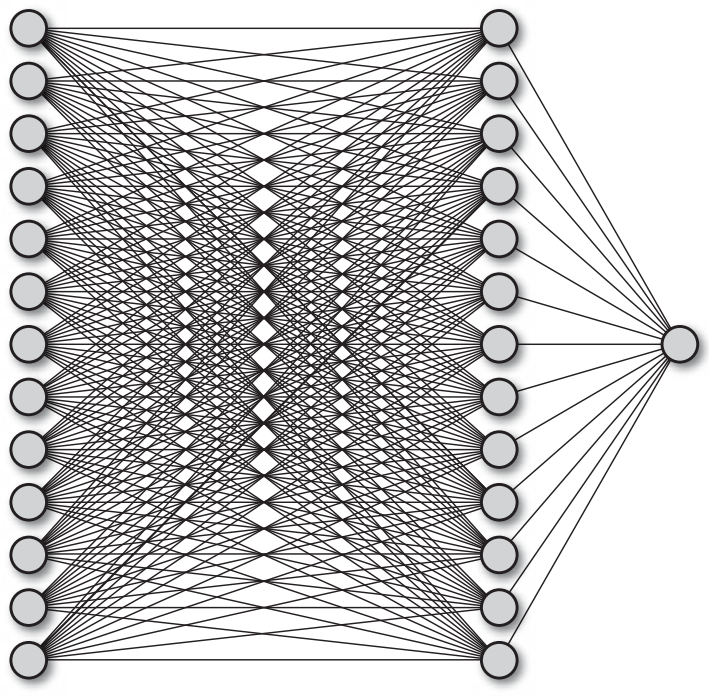


Рис. 2.11. Более распространенное (но во многих отношениях менее полезное) визуальное представление нейронной сети

с помощью словаря `weights`, а возвращает функция величину потерь и словарь `forward_info`, содержащий информацию о результатах прямого прохода. Просто на этот раз мы подставляем в код данные об операциях с рис. 2.10:

```
def forward_loss(X: ndarray,
                 y: ndarray,
                 weights: Dict[str, ndarray]
                 ) -> Tuple[Dict[str, ndarray], float]:
    '''
    Прямой проход через модель нейронной сети и определение величины
    потерь.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']
```

```
O1 = sigmoid(N1)

M2 = np.dot(O1, weights['W2'])

P = M2 + weights['B2']

loss = np.mean(np.power(y - P, 2))

forward_info: Dict[str, ndarray] = {}
forward_info['X'] = X
forward_info['M1'] = M1
forward_info['N1'] = N1
forward_info['O1'] = O1
forward_info['M2'] = M2
forward_info['P'] = P
forward_info['y'] = y

return forward_info, loss
```

Несмотря на более сложную схему, последовательность операций не меняется. Мы производим запрограммированные вычисления и сохраняем полученные результаты в словарь `forward_info`.

Обратный проход

Обратный проход выполняется так же, как и в случае более простой модели линейной регрессии, просто на этот раз он включает большее количество шагов.

Визуализация

Напомню, какие шаги нам предстоят:

1. Вычисление значений всех производных при заданных входных данных.
2. Умножение полученных результатов.

Вы снова убедитесь, что благодаря цепному правилу все работает. Частные производные, которые нам предстоит вычислить, показаны на рис. 2.12.

В результате мы получим градиент потерь по каждому из весов.

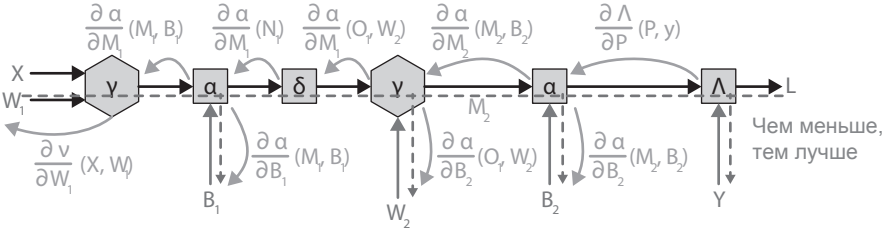


Рис. 2.12. Во время обратного прохода мы получим произведение всех частных производных, связанных с операциями в нашей нейронной сети

Математическое представление и код

Все частные производные, с которыми нам предстоит иметь дело, и соответствующие им строки кода представлены в табл. 2.1.

Таблица 2.1. Список производных для нейронной сети

Производная	Код
$\frac{\partial \Lambda}{\partial P}(P, y)$	`dLdP = -(forward_info[y] - forward_info[P])`
$\frac{\partial \alpha}{\partial M_2}(M_2, B_2)$	`np.ones_like(forward_info[M2])`
$\frac{\partial \alpha}{\partial B_2}(M_2, B_2)$	`np.ones_like(weights[B2])`
$\frac{\partial \nu}{\partial W_2}(O_1, W_2)$	`dM2dW2 = np.transpose(forward_info[O1], (1, 0))`
$\frac{\partial \nu}{\partial O_1}(O_1, W_2)$	`dM2dO1 = np.transpose(weights[W2], (1, 0))`
$\frac{\partial \sigma}{\partial u}(N_1)$	`dO1dN1 = sigmoid(forward_info[N1]) × (1 - sigmoid(forward_info[N1])`
$\frac{\partial \alpha}{\partial M_1}(M_1, B_1)$	`dN1dM1 = np.ones_like(forward_info[M1])`
$\frac{\partial \alpha}{\partial B_1}(M_1, B_1)$	`dN1dB1 = np.ones_like(weights[B1])`
$\frac{\partial \nu}{\partial W_1}(X, W_1)$	`dM1dW1 = np.transpose(forward_info[X], (1, 0))`

Выражения для градиента потерь по элементам смещения `dLdB1` и `dLdB2` нужно суммировать вдоль каждой строки, так как в пакете данных к каждой строке добавляется один и тот же элемент смещения. Подробнее см. раздел «Градиент потерь с учетом смещения» приложения А.

Полный градиент потерь

Полный код функции `loss_gradients` вы найдете в репозитории Jupyter Notebook к этой главе (https://oreil.ly/2TDV5q9). Эта функция вычисляет все частные производные из табл. 2.1 и перемножает их, давая на выходе градиенты потерь по каждому объекту `ndarray` с весами:

- `dLdW2`;
- `dLdB2`;
- `dLdW1`;
- `dLdB1`.

Остается получить сумму элементов `dLdB1` и `dLdB2` вдоль оси 0, как описано в разделе «Градиент потерь с учетом смещения» в приложении А.

Наша первая нейронная сеть готова! Посмотрим, превосходит ли она модель линейной регрессии.

Обучение и оценка нейронной сети

Прямой и обратный проходы для нейронной сети выполнялись так же, как и для созданной в начале главы модели линейной регрессии. Одинаковыми будут и схемы обучения и оценка обеих моделей. На каждой итерации в функцию подаются данные, для которых выполняется прямой проход, затем во время обратного прохода вычисляются градиенты потерь по весам, и на базе этой информации веса обновляются. Фактически внутри обучающего цикла мы можем использовать уже знакомый код:

```
forward_info, loss = forward_loss(X_batch, y_batch, weights)

loss_grads = loss_gradients(forward_info, weights)
```

```
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Разница состоит в содержимом функций `forward_loss` и `loss_gradients`, а также в данных словаря `weights`, в котором теперь будет не два, а четыре ключа (`W1`, `B1`, `W2` и `B2`). Фактически это и есть основная идея книги: даже для сложных архитектур в случае простых моделей используются одни и те же математические принципы и высокоуровневые обучающие процедуры.

Строка кода для получения предсказаний тоже останется без изменений:

```
preds = predict(X_test, weights)
```

Просто на этот раз будет применяться другая функция `predict`:

```
def predict(X: ndarray,
            weights: Dict[str, ndarray]) -> ndarray:
    '''
    Генерация предсказаний моделью нейронной сети.
    '''
    M1 = np.dot(X, weights['W1'])

    N1 = M1 + weights['B1']

    O1 = sigmoid(N1)

    M2 = np.dot(O1, weights['W2'])

    P = M2 + weights['B2']

    return P
```

Рассчитаем для проверки средний модуль отклонения и корень из среднего квадрата отклонения:

```
Mean absolute error: 2.5289
Root mean squared error: 3.6775
```

Оба параметра значительно ниже, чем в предыдущей модели! Рис. 2.13 показывает, как предсказанные значения соотносятся с целевыми. Здесь тоже видны улучшения.

Визуально точки лежат ближе к прямой с наклоном 45 градусов, чем на рис. 2.6. Попробуйте самостоятельно запустить код из репозитория Jupyter Notebook к этой главе (https://oreil.ly/2TDV5q9)!

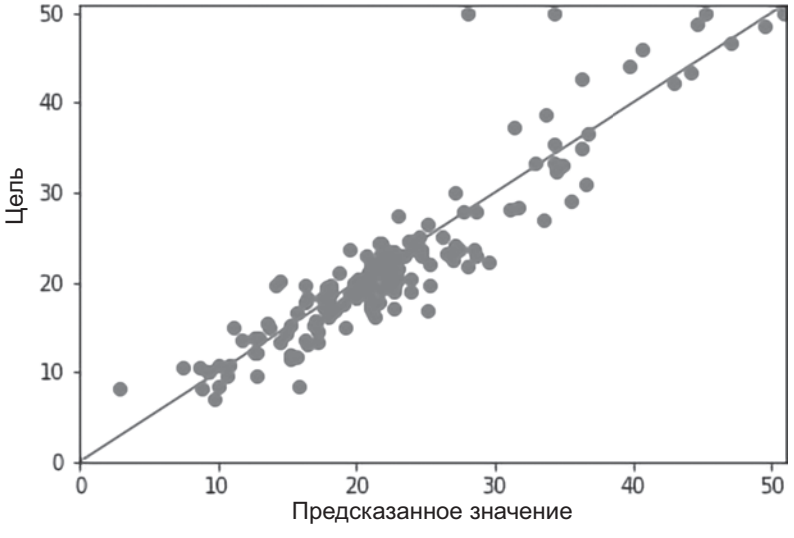


Рис. 2.13. Предсказанные и целевые значения при решении задачи регрессии нейронной сетью

Почему все получилось именно так

Почему эта модель работает лучше предыдущей? Напомню, что в предыдущем случае соотношение между самым важным признаком и целью было *нелинейным*, в то время как модель умела выявлять только линейные взаимосвязи. И я высказал предположение, что мы сможем увеличить точность модели, добавив в нее нелинейную функцию.

Рис. 2.14 демонстрирует уже знакомый нам график нормализованных значений самого важного признака вместе с целевыми и прогнозируемыми значениями. Последние получены путем передачи в модель средних значений остальных признаков, причем величина наиболее важного признака, как и раньше, варьируется от –3.5 до 1.5.

Мы видим, что теперь обнаруженная взаимосвязь (а) нелинейна и (б) лучше аппроксимирует соотношение между целью и прогнозами (пред-

ставленное точками). Что собственно и требовалось. Так что добавление в нашу модель нелинейной функции позволило путем последовательного обновления весов и обучения модели обнаружить нелинейную связь между входными и выходными данными.

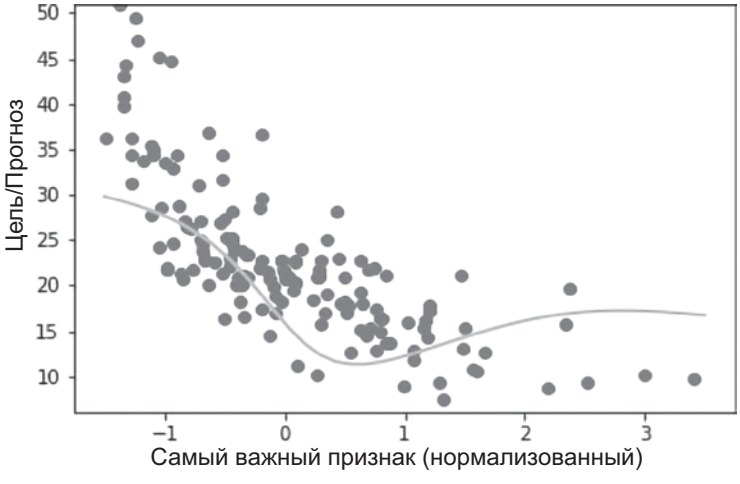


Рис. 2.14. Сравнение самого важного признака с целевым значением в модели нейронной сети

Это первая причина, по которой наша нейронная сеть работает лучше модели линейной регрессии. Кроме того, нейронная сеть может изучать взаимосвязи между *комбинациями* исходных признаков и цели, а не только между отдельными признаками. Ведь нейронная сеть использует умножение матриц для создания 13 «изученных признаков», каждый из которых представляет собой комбинацию всех исходных, и уже к этим извлеченным признакам применяется еще одна линейная регрессия. Анализ данных, который вы можете найти на сайте книги, показал наиболее важные извлеченные признаки. Это:

$$-4.44 \times \text{признак}_6 - 2.77 \times \text{признак}_1 - 2.07 \times \text{признак}_7 + \ldots$$

и:

$$4.43 \times \text{признак}_2 - 3.39 \times \text{признак}_4 - 2.39 \times \text{признак}_1 + \ldots$$

Вместе с 11 другими извлеченными признаками они будут включены в последние два слоя нейронной сети.

Именно эти две вещи — обнаружение нелинейных взаимосвязей между признаками и целью и обнаружение взаимосвязей между *комбинациями* признаков и целью — обеспечивают бóльшую по сравнению с моделью линейной регрессии точность нейронных сетей при решении практических задач.

Заключение

Из этой главы вы узнали, как с помощью принципов, изложенных в главе 1, создать и обучить две стандартные модели машинного обучения. Я показал, как представить в виде вычислительного графа простую модель из классической статистики — модель линейной регрессии. Подобное представление позволяет рассчитывать градиенты потерь по различным параметрам модели и обучать модель, подавая ей на вход данные из обучающего набора и обновляя параметры в направлении, уменьшающем потери.

Но, как мы увидели позже, эта модель ограничена, так как обнаруживает только линейные закономерности между признаками и целью. Попытка построить модель, выявляющая нелинейные взаимосвязи, привела к созданию нашей первой нейронной сети. Вы узнали, как создать такую сеть с нуля и как обучить ее с помощью уже знакомой процедуры, с помощью которой мы обучали модель линейной регрессии. Вы убедились, что нейронная сеть работает лучше модели линейной регрессии, и теперь знаете, почему так получается.

Мы неспроста ограничились в этой главе двумя относительно простыми моделями. Ведь их определение выполнялось вручную. Для прямого прохода потребовалось написать код 6 операций, а для обратного — 17. Но проницательные читатели могли заметить, что готовый код содержит много повторений. И достаточно корректно выбрать уровень абстракции, как можно будет отказаться от определения моделей с точки зрения отдельных операций, которым мы занимались в этой главе. Это позволит создавать более сложные модели, в том числе модели глубокого обучения. Именно этим мы и займемся в следующей главе. Вперед!

ГЛАВА 3

Основы глубокого обучения

Возможно, вы удивитесь, но мы уже освоили весь необходимый математический аппарат и основные понятия, чтобы ответить на ключевые вопросы о моделях глубокого обучения, которые были заданы в начале книги. Теперь вы понимаете, *как* работают нейронные сети, работа с которыми представляет собой вычисления, связанные с умножением матриц, вычислением потерь и частных производных по отношению к этим потерям, а также разобрались, *почему* вся эта магия работает (спасибо цепному правилу). Первые созданные нами нейросети мы представили как множество «строительных блоков», где каждый такой блок представлял собой какую-либо математическую функцию. В этой главе мы превратим эти строительные блоки в абстрактные классы Python, чтобы в дальнейшем использовать их для построения моделей глубокого обучения. Как и обещано — все с чистого листа!

Мы также сопоставим нейросетевые описания с более традиционными описаниями моделей глубокого обучения, которые вы, возможно, уже знаете. Например, мы узнаем, что такое модели с «несколькими скрытыми слоями». Здесь вся суть в том, чтобы сопоставить высокоуровневое описание с тем, как все работает на самом деле. Пока что мы давали только низкоуровневое описание. В первой части этой главы мы сопоставим это описание моделей с общими понятиями более высокого уровня, такими как «слои», которые в конечном итоге позволят нам более легко описывать более сложные модели.

Определение глубокого обучения: первый проход

Что такое «модель глубокого обучения»? В предыдущей главе мы писали, что это математическая функция в виде вычислительного графа. Модель должна была сопоставить входные данные из некоторого набора данных признаков (данные, например, могут соответствовать характеристикам

домов) с выходными данными, полученными из соответствующего распределения (например, цены этих домов). Оказалось, что если модель будет функцией, в которой для всех входных данных задавался бы свой вес, то функцию можно было бы «подогнать», чтобы получить правильный выход. Для этого нужно:

1. Пропустить наблюдения через модель, отслеживая величины, вычисленные по пути во время этого «прямого прохода».
2. Рассчитать *потери*, показывающие, насколько прогнозы нашей модели отличаются от *целевых результатов*.
3. Используя величины, вычисленные на прямом проходе, и цепное правило, описанное в главе 1, вычислить, как сильно каждый из входных *параметров* влияет на величину ошибки.
4. Изменить значения параметров, чтобы при повторном проходе ошибка стала меньше.

В первой модели у нас была линейная комбинация, превращающая признаки в целевые данные (что эквивалентно традиционной модели линейной регрессии). Проблема была ожидаема: даже при оптимальном подборе весов модель отражала просто линейное отношение между признаками и целью.

Затем мы определили структуру функции, которая применяет сначала линейные операции, потом нелинейную операцию (сигмоиду), а затем еще один набор линейных операций. Оказалось, что теперь модель стала лучше понимать взаимоотношение между входом и выходом, а заодно научилась изучать отношения между *комбинациями* наших входных признаков и целевыми данными.

При чем же тут модели глубокого обучения? Начнем грубо — это будут последовательности операций, в которых задействованы как *минимум две* непоследовательные нелинейные функции.

Вскоре я покажу, откуда берется такое определение, но сначала отметим, что поскольку модели глубокого обучения — это последовательность операций, процесс их обучения будет идентичен тому, что мы уже видели ранее. Работоспособность модели обусловлена дифференцируемостью модели по отношению к ее входам, и как упоминалось в главе 1, комбинация дифференцируемых функций тоже является дифференцируемой, и если

отдельные функции дифференцируемы, то вся функция будет дифференцируемой и мы сможем обучить ее, используя описанные выше 4 шага.

Но есть и разница. До сих пор мы вычисляли эти производные путем ручного кодирования прямого и обратного проходов, а затем перемножали соответствующие величины, чтобы получить производные. Для простой модели нейронной сети, показанной в главе 2, потребовалось 17 шагов. Из-за такого описания не сразу понятно, как можно усложнить модель (и в чем будет сама сложность) или хотя бы заменить сигмоиду на другую нелинейную функцию. Чтобы научиться создавать сколь угодно глубокие и сложные модели глубокого обучения, нужно подумать о том, в какой момент на этих 17 этапах мы можем создать нечто более многоразовое, нежели отдельные операции. Это нечто должно позволить создавать разные модели. В качестве первого шага мы попытаемся сопоставить используемые нами операции с традиционными описаниями нейронных сетей, состоящими из «слоев», «нейронов» и т. д.

Сначала нужно создать абстракцию для представления отдельных операций, над которыми мы работали до сих пор, а не продолжать кодировать одно и то же умножение матриц и прибавление отклонений.

Строительные блоки нейросети: операции

Класс `Operation` будет реализовывать одну из функций в сети. Мы уже знаем, что на высоком уровне у такой функции должен быть прямой и обратный методы, каждый из которых получает в качестве входных данных и выводит объект `ndarray`. Некоторые операции, например матричное умножение, тоже принимают на вход `ndarray`: параметры. В нашем классе `Operation` или, возможно, в другом классе, который наследуется от него, параметры должны храниться в переменной экземпляра.

Еще одна мысль: существуют два типа класса `Operation`: некоторые, например матричное умножение, возвращают `ndarray`, но в форме, отличной от входной. И наоборот, некоторые операции, такие как сигмоидальная функция, просто применяют некоторую функцию к каждому элементу входного `ndarray`. Как обобщить это? Давайте рассмотрим объекты `ndarray`, которые проходят через наши операции: каждая операция передает выходные данные следующему слою на прямом проходе, а на обратном проходе получает «выходной градиент», который будет представлять частную

производную потерь относительно каждого элемента вывода `Operation` (которая вычисляется другими операциями, составляющими сеть). Кроме того, на обратном проходе операция отправляет назад «входной градиент» — частную производную потерь по каждому элементу ввода.

Эти факты накладывают несколько важных ограничений на работу классов `Operation`, но это помогает правильно считать градиент:

- Форма *выходного* градиента `ndarray` должна соответствовать форме выходного файла.
- Форма *входного* градиента, который `Operation` отправляет назад во время обратного прохода, должна соответствовать форме входа `Operation`.

Вы поймете суть, взглянув на рисунок ниже.

Визуализация

На рис. 3.1 мы привели пример операции `O`, которая получает входные данные от операции `N` и передает выходные данные другой операции `P`.

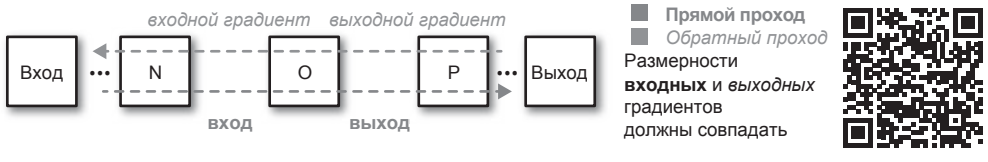


Рис. 3.1. Operation с вводом и выводом

На рис. 3.2 рассмотрен случай класса `Operation` с параметрами.

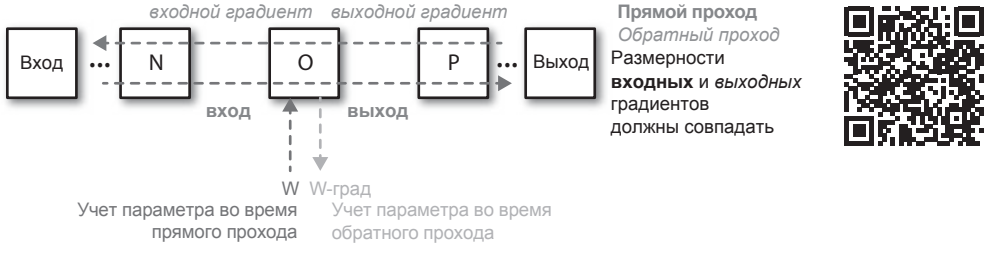


Рис. 3.2. ParamOperation с входом, выходом и параметрами

Код

Теперь мы можем написать базовую часть нашей нейронной сети, класс `Operation`:

```
class Operation(object):
    '''
    Базовый класс операции в нейросети.
    '''
    def __init__(self):
        pass

    def forward(self, input_: ndarray):
        '''
        Хранение ввода в атрибуте экземпляра self._input
        Вызов функции self._output().
        '''
        self.input_ = input_

        self.output = self._output()

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Вызов функции self._input_grad().
        Проверка совпадения размерностей.
        '''
        assert_same_shape(self.output, output_grad)

        self.input_grad = self._input_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        return self.input_grad


    def _output(self) -> ndarray:
        '''
        Метод _output определяется для каждой операции.
        '''
        raise NotImplementedError()
```

```
    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Метод _input_grad определяется для каждой операции.
        '''
        raise NotImplementedError()
```

Для каждой операции, которую мы определяем, нужно будет задать функции `_output` и `_input_grad`.

Такие базовые классы нужны скорее для понимания; важно, чтобы наше представление процесса глубокого обучения соответствовало этой схеме: вперед передаются входные данные, а обратно — градиент, и размерности полученного «спереди» соответствуют тому, что отправляется «назад», и наоборот.

Позже определим конкретные операции, которые мы уже использовали, например умножение матриц. Но сначала определим еще один класс, который наследует от `Operation` и который мы будем использовать специально для операций с параметрами:

```
class ParamOperation(Operation):
    '''
    Операция с параметрами.
    '''

    def __init__(self, param: ndarray) -> ndarray:
        '''
        Метод ParamOperation
        '''
        super().__init__()
        self.param = param

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Вызов self._input_grad и self._param_grad.
        Проверка размерностей.
        '''

        assert_same_shape(self.output, output_grad)
```

```
        self.input_grad = self._input_grad(output_grad)
        self.param_grad = self._param_grad(output_grad)

        assert_same_shape(self.input_, self.input_grad)
        assert_same_shape(self.param, self.param_grad)

        return self.input_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Во всех подклассах ParamOperation должна быть реализация
        метода _param_grad.
        '''
        raise NotImplementedError()
```

Подобно базовому классу `Operation`, отдельная операция `ParamOperation` должна определять функцию `_param_grad` в дополнение к функциям `_output` и `_input_grad`.

Теперь мы формализовали строительные блоки нейронной сети, которые использовали в наших моделях до сих пор. Мы могли бы пропустить и определить нейронные сети непосредственно в терминах этих операций, но есть промежуточный класс, вокруг которого мы плясали в течение полутора глав и который мы определим в первую очередь: слой.

Строительные блоки нейросети: слои

С точки зрения операций слой — это набор линейных операций, за которыми следует нелинейная операция. Например, в нейронной сети из предыдущей главы было пять операций: две линейные операции — умножение на вес и добавление смещения, потом сигмоида, а затем еще две линейные операции. В этом случае мы бы сказали, что первые три операции, вплоть до нелинейной, будут составлять первый слой, а последние две операции — второй слой. Входные данные — это тоже особый вид слоя, называемый *входным* слоем (с точки зрения нумерации слоев этот слой не считается, так что пусть он будет «нулевым»). Последний слой аналогично называется *выходным* слоем. Средний слой — «первый» — называется *скрытым*, так как только на этом слое значения явно не отображаются во время обучения.

Выходной слой тоже особенный, поскольку к нему *не нужно* применять нелинейную операцию. Это связано с тем, что нам надо на выходе из этого слоя получать значения между отрицательной бесконечностью и бесконечностью (или, по крайней мере, между 0 и бесконечностью), а нелинейные функции обычно «сдавливают» входные данные до некоторого подмножества этого диапазона (например, сигмоидальная функция сокращает свой ввод до значения от 0 до 1).

Визуализация

Для наглядности на рис. 3.3 показана схема нейронной сети из предыдущей главы, при этом отдельные операции разбиты на слои.

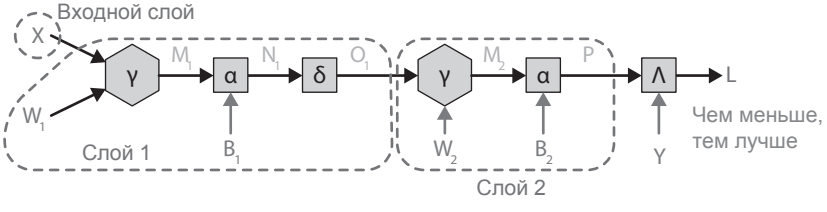


Рис. 3.3. Нейронная сеть из предыдущей главы с группировкой операций на слои

Из рисунка видно, что входные данные представляют «входной» слой, следующие три операции (до сигмоиды) — следующий слой, а последние две операции — последний слой.

Выглядит довольно громоздко. Но так и должно быть, так как представление нейронной сети в виде последовательности операций хоть и позволяет понять, как сети обучаются и работают, является слишком «низкоуровневым» для чего-либо более сложного, чем двухслойная нейронная сеть. Поэтому наиболее распространенный способ представления нейронных сетей — это слои (рис. 3.4).

Связь с мозгом

Осталось затронуть пару вопросов, чтобы все встало на свои места. Можно сказать, что на каждом слое находится определенное количество нейронов, равное *размерности вектора наблюдений на выходе слоя*. Таким образом,

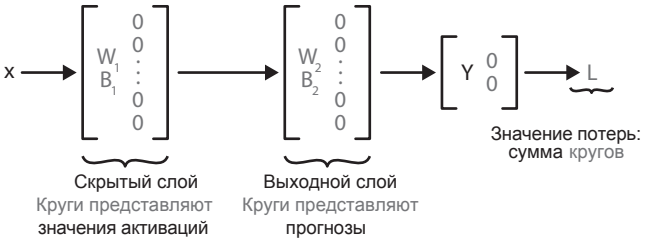


Рис. 3.4. Нейронная сеть из предыдущей главы с точки зрения слоев

у нейронной сети из предыдущего примера 13 нейронов на входном слое, 13 нейронов на скрытом слое и один нейрон на выходном слое.

Нейроны в человеческом мозге тоже могут получать входные сигналы от многих других нейронов, после чего отправляют свой собственный сигнал вперед, если полученные ими сигналы совокупно достаточно «мощные» для активации. Нейроны в контексте нейронных сетей работают похожим образом: их выходные сигналы зависят от входов, но входы преобразуются в выходы с помощью нелинейной функции. Таким образом, эта нелинейная функция называется функцией активации, а выходящие из нее значения — активациями для этого слоя[1].

Теперь, когда мы разобрались со слоями, то можем сформулировать более традиционное определение глубокого обучения: *модели глубокого обучения — это нейронные сети с несколькими скрытыми слоями*.

Такое определение эквивалентно тому, что мы говорили ранее о классах `Operation`, поскольку слой — это просто серия операций, завершаемая нелинейной операцией.

Теперь, когда мы определили базовый класс для наших операций, посмотрим, как из него составить модели, которые мы видели в предыдущей главе.

---

[1] Среди всех функций активации сигмоида, которая отображает входные данные на диапазон между 0 и 1, наиболее точно имитирует фактическую активацию нейронов в головном мозге, но в целом функции активации могут иметь и более гладкие и линейные формы.

Блочное строительство

Какие именно операции нужно совершить для реализации упомянутых в предыдущей главе моделей? Мы рассмотрели уже три:

- матричное умножение входного вектора на матрицу весов;
- добавление отклонения;
- сигмоидная функция активации.

Начнем с операции `WeightMultiply`:

```
class WeightMultiply(ParamOperation):
    '''
    Умножение весов в нейронной сети.
    '''

    def __init__(self, W: ndarray):
        '''
        Инициализация класса Operation с self.param = W.
        '''
        super().__init__(W)

    def _output(self) -> ndarray:
        '''
        Вычисление выхода.
        '''
        return np.dot(self.input_, self.param)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Вычисление выходного градиента.
        '''
        return np.dot(output_grad, np.transpose(self.param, (1, 0)))

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Вычисление градиента параметров.
        '''
        return np.dot(np.transpose(self.input_, (1, 0)), output_grad)
```

Здесь реализовано умножение матриц на прямом проходе, а также правила «возврата градиентов» входов и параметров на обратном проходе (это правило мы обсудили в конце главы 1). Как вы вскоре увидите, мы можем использовать этот класс как строительный блок для составления слоев.

Далее идет операция суммирования, которую мы назовем `BiasAdd`:

```
class BiasAdd(ParamOperation):
    '''
    Прибавление отклонений.
    '''

    def __init__(self,
                 B: ndarray):
        '''
        Инициализация класса Operation с self.param = B.
        Проверка размерностей.
        '''
        assert B.shape[0] == 1

        super().__init__(B)

    def _output(self) -> ndarray:
        '''
        Вычисление выхода.
        '''
        return self.input_ + self.param

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Вычисление входного градиента.
        '''
        return np.ones_like(self.input_) * output_grad

    def _param_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Вычисление градиента параметров.
        '''
        param_grad = np.ones_like(self.param) * output_grad
        return np.sum(param_grad,
                      axis=0).reshape(1, param_grad.shape[1])
```

Наконец, реализуем сигмоиду:

```
class Sigmoid(Operation):
    '''
    Сигмоидная функция активации.
    '''

    def __init__(self) -> None:
        '''пока ничего не делаем'''
        super().__init__()

    def _output(self) -> ndarray:
        '''
        Вычисление выхода.
        '''
        return 1.0/(1.0+np.exp(-1.0 * self.input_))

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        '''
        Вычисление входного градиента.
        '''
        sigmoid_backward = self.output * (1.0 — self.output)
        input_grad = sigmoid_backward * output_grad
        return input_grad
```

Мы просто реализовали математику, описанную в предыдущей главе.

И для сигмоиды, и для `ParamOperation` шагом обратного прохода, где мы вычисляем:

`input_grad = <что-то> * output_grad`,

является шаг, где мы применяем цепное правило, а соответствующее правило для `WeightMultiply`:

`np.dot (output_grad, np.transpose (self.param, (1, 0)))`,

как мы отмечали в главе 1, является аналогом цепного правила, когда рассматриваемая функция является матричным умножением.

С операциями закончили, теперь можно переходить к определению класса `Layer`.

Шаблон слоя

Написанный класс `Operation` поможет написать класс `Layer`:

- Методы прямого и обратного прохода просто включают последовательную отправку входных данных через последовательность операций, что мы и показывали на графиках. Это самое важное, а все остальное — обертка и сопутствующие действия:
  - определение правильной последовательности `Operation` в функции `_setup_layer` и инициализация и сохранение параметров в этих `Operation` (то же самое делается в функции `_setup_layer`);
  - сохранение правильных значений в `self.input_` и `self.output` для прямого метода;
  - выполнение проверки правильности в обратном методе.
- Наконец, функции `_params` и `_param_grads` просто извлекают параметры и их градиенты (относительно потерь) из `ParamOperations` в слое.

Вот как все это выглядит:

```
class Layer(object):
    '''
    Слой нейронов в нейросети.
    '''
def __init__(self,
             neurons: int):
    '''
    Число нейронов примерно соответствует «ширине» слоя
    '''
    self.neurons = neurons
    self.first = True
    self.params: List[ndarray] = []
    self.param_grads: List[ndarray] = []
    self.operations: List[Operation] = []

    def _setup_layer(self, num_in: int) -> None:
        '''
        Функция _setup_layer реализуется в каждом слое.
        '''
        raise NotImplementedError()
```

```
    def forward(self, input_: ndarray) -> ndarray:
        '''
        Передача входа вперед через серию операций.
        '''
        if self.first:
            self._setup_layer(input_)
            self.first = False

        self.input_ = input_

        for operation in self.operations:

            input_ = operation.forward(input_)

        self.output = input_

        return self.output

    def backward(self, output_grad: ndarray) -> ndarray:
        '''
        Передача output_grad назад через серию операций.
        Проверка размерностей.
        '''
        assert_same_shape(self.output, output_grad)

        for operation in reversed(self.operations):
            output_grad = operation.backward(output_grad)

        input_grad = output_grad

        self._param_grads()
        return input_grad

    def _param_grads(self) -> ndarray:
        '''
        Извлечение _param_grads из операций слоя.
        '''

        self.param_grads = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
```

```
                self.param_grads.append(operation.param_grad)

    def _params(self) -> ndarray:
        '''
        Извлечение _params из операций слоя.
        '''

        self.params = []
        for operation in self.operations:
            if issubclass(operation.__class__, ParamOperation):
                self.params.append(operation.param)
```

Подобно тому как мы перешли от абстрактного определения класса `Operation` к реализации конкретных операций, необходимых для нейронной сети, сделаем то же самое со слоями.

Полносвязный слой

Операции мы называли по смыслу — `WeightMultiply`, `BiasAdd` и т. д. А как назвать слой? Слой `LinearNonLinear` или как?

Определяющей характеристикой этого слоя является то, что *каждый выходной нейрон является функцией всех входных нейронов*. Именно так и работает умножение матриц: если матрица состоит из *n* строк и столбцов, результатом умножения будет *n* новых признаков, каждый из которых представляет собой взвешенную линейную комбинацию всех *n* входных объектов[1]. Таким образом, эти слои часто называют полносвязными слоями, или плотными слоями.

С названием определились, теперь определим класс `Dense` с точки зрения операций, которые мы уже определили. Уже понятно, что все, что нам нужно сделать, это поместить операции, определенные в предыдущем разделе, в список функции `_setup_layer`.

```
class Dense(Layer):
    '''
    Полносвязный слой, наследующий от Layer.
```

[1] Как мы увидим в главе 5, это относится не ко всем слоям: например, в сверточных слоях каждый выходной объект является комбинацией лишь небольшого подмножества входных объектов.

```
    '''
    def __init__(self,
                 neurons: int,
                 activation: Operation = Sigmoid()) -> None:
        '''
        Для инициализации нужна функция активации.
        '''
        super().__init__(neurons)
        self.activation = activation

    def _setup_layer(self, input_: ndarray) -> None:
        '''
        Определение операций для полносвязного слоя.
        '''
        if self.seed:
            np.random.seed(self.seed)

        self.params = []

        # веса
        self.params.append(np.random.randn(input_.shape[1],
                                           self.neurons))

        # отклонения
        self.params.append(np.random.randn(1, self.neurons))

        self.operations = [WeightMultiply(self.params[0]),
                           BiasAdd(self.params[1]),
                           self.activation]
        return None
```

Обратите внимание, что если мы сделаем активацию по умолчанию линейной, то это то же самое, что активации нет как таковой, а на выход слоя передается то же, что и на входе.

Что понадобится еще помимо `Operation` и `Layer`? Чтобы обучить нашу модель, точно понадобится класс `NeuralNetwork`, которым мы обернем `Layer` так же, как класс `Layer` обернут вокруг `Operation`. Мы не знаем, какие еще классы понадобятся, поэтому пока сделаем `NeuralNetwork`, а дальше определимся по ходу дела.

Класс NeuralNetwork и, возможно, другие

Что должен делать наш класс `NeuralNetwork`? Он должен учиться на данных, а точнее, собирать пакеты «наблюдений» (`X`) и «правильных ответов» (`y`), и изучать взаимосвязь между `X` и `y`, подстраивая функцию, которая позволит преобразовать `X` в предсказания `p`, близкие к `y`.

Как именно будет происходить это обучение, учитывая только что определенные классы `Layer` и `Operation`? Вспоминая, как работала модель из последней главы, реализуем следующее:

1. Нейронная сеть принимает на вход набор данных `X` и последовательно пропускает его через каждый слой (а на самом деле — через ряд операций), и полученным результатом будет прогноз.
2. Прогноз сравнивается с `y`, чтобы рассчитать потери и сгенерировать «градиент потерь», который является частной производной потери по каждому элементу в последнем слое в сети (который и создает прогноз).
3. Наконец, этот градиент проходит по сети в обратном направлении через каждый уровень и вычисляются «градиенты параметров» — частная производная потерь по каждому из параметров. Результаты сохраняются.

Визуализация

На рис. 3.5 показано описание нейронной сети с точки зрения слоев.

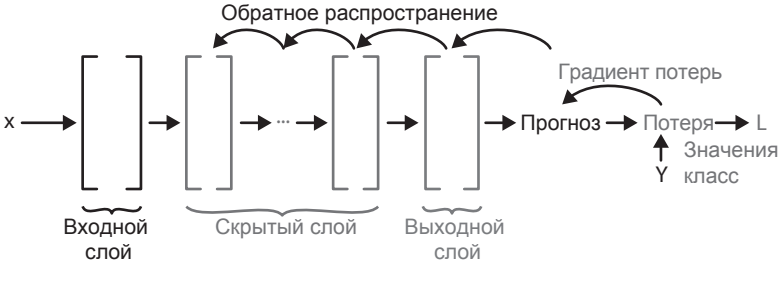


Рис. 3.5. Обратное распространение со слоями вместо операций

Код

Как это реализовать? Мы хотим, чтобы наша нейронная сеть в конечном итоге работала со слоями так же, как наши слои работали с операциями. То есть метод `forward` должен получать `X` в качестве входных данных и просто делать что-то вроде:

```
for layer in self.layers:
    X = layer.forward (X)

return X
```

Аналогично мы хотим, чтобы обратный метод брал градиент и делал что-то вроде:

```
for layer in reversed(self.layers):
    grad = layer.backward(grad)
```

Откуда берется этот самый `grad`? Он берется от потери, специальной функции, которая принимает прогноз с `y`:

- Вычисляет одно число, представляющее собой «штраф» для сети, которая делает такой прогноз.
- Отправляет назад градиент для каждого элемента прогноза относительно потери. Этот градиент — это то, что последний слой в сети получит как вход для своей обратной функции.

В примере из предыдущей главы функция потерь была квадратом разности между прогнозом и целью, и градиент прогноза относительно потери вычислялся соответствующим образом.

Как это реализовать? Потери — это важно, поэтому для них надо будет сделать собственный класс. Кроме того, этот класс может быть реализован аналогично классу `Layer`, за исключением того, что метод `forward` будет выдавать число с плавающей точкой вместо объекта `ndarray`, которое будет передано на следующий уровень. Перейдем к коду.

Класс Loss

Базовый класс `Loss` будет похож на `Layer` — методы `forward` и `backward` будут проверять соответствие размерностей `ndarrays` и выполнять два

метода, _output и _input_grad, которые должен реализовывать любой подкласс Loss:

```
class Loss(object):
    '''
    Потери нейросети.
    '''

    def __init__(self):
        '''Пока ничего не делаем'''
        pass

    def forward(self, prediction: ndarray, target: ndarray) -> float:
        '''
        Вычисление значения потерь.
        '''
        assert_same_shape(prediction, target)

        self.prediction = prediction
        self.target = target

        loss_value = self._output()

        return loss_value

    def backward(self) -> ndarray:
        '''
        Вычисление градиента потерь по входам функции потерь.
        '''
        self.input_grad = self._input_grad()

        assert_same_shape(self.prediction, self.input_grad)

        return self.input_grad

    def _output(self) -> float:
        '''
        Функция _output должна реализовываться всем подклассами
        класса Loss.
        '''
        raise NotImplementedError()
```

```
    def _input_grad(self) -> ndarray:
        '''
        Функция _input_grad должна реализовываться всем подклассами
        класса Loss.
        '''
        raise NotImplementedError()
```

Как и в классе `Operation`, мы проверяем, что градиент, который потери отправляют назад, имеет ту же форму, что и прогноз, полученный в качестве входных данных от последнего уровня сети:

```
class MeanSquaredError(Loss):

    def __init__(self)
        '''пока ничего не делаем'''
        super().__init__()

    def _output(self) -> float:
        '''
        вычисление среднего квадрата ошибки.
        '''
        loss =
            np.sum(np.power(self.prediction – self.target, 2)) /
            self.prediction.shape[0]

        return loss

    def _input_grad(self) -> ndarray:
        '''
        Вычисление градиента ошибки по входу MSE.
        '''

        return 2.0 * (self.prediction – self.target) /
        self.prediction.shape[0]
```

Здесь мы просто кодируем прямое и обратное правила формулы среднеквадратичной потери.

Это последний требуемый строительный блок. Давайте рассмотрим, как эти части сочетаются друг с другом, а затем приступим к построению модели.

Глубокое обучение с чистого листа

В конечном итоге мы хотим создать класс `NeuralNetwork`, как на рис. 3.5, и использовать эту нейросеть для моделей глубокого обучения. Прежде чем мы начнем писать код, давайте точно опишем, каким будет этот класс и как он будет взаимодействовать с классами `Operation`, `Layer` и `Loss`, которые мы только что определили:

1. `NeuralNetwork` будет в качестве атрибута получать список экземпляров `Layer`. Слои будут такими, как было определено ранее — с прямым и обратным методами. Эти методы принимают объекты `ndarray` и возвращают объекты `ndarray`.
2. Каждый `Layer` будет иметь список операций `Operation`, сохраненный в атрибуте `operations` слоя функцией `_setup_layer`.
3. Эти операции, как и сам слой, имеют методы прямого и обратного преобразования, которые принимают в качестве аргументов объекты `ndarray` и возвращают объекты `ndarray` в качестве выходных данных.
4. В каждой операции форма `output_grad`, полученная в методе `backward`, должна совпадать с формой выходного атрибута `Layer`. То же самое верно для форм `input_grad`, передаваемых в обратном направлении методом `backward` и атрибутом `input_`.
5. Некоторые операции имеют параметры (которые хранятся в атрибуте `param`). Эти операции наследуют от класса `ParamOperation`. Те же самые ограничения на входные и выходные формы применяются к слоям и их методам `forward` и `backward` — они берут объекты `ndarray`, и формы входных и выходных атрибутов и их соответствующие градиенты должны совпадать.
6. У класса `NeuralNetwork` также будет класс `Loss`. Этот класс берет выходные данные последней операции из `NeuralNetwork` и цели, проверяет, что их формы одинаковы, и, вычисляя значение потерь (число) и `ndarray loss_grad`, которые будут переданы в выходной слой, начинает обратное распространение.

Реализация пакетного обучения

Мы уже говорили о шагах обучения модели по одной партии за раз. Повторим:

1. Подаем входные данные через функцию модели («прямой проход») для получения прогноза.
2. Рассчитываем потери.
3. Вычисляем градиенты потерь по параметрам с использованием цепного правила и значений, вычисленных во время прямого прохода.
4. Обновляем параметры на основе этих градиентов.

Затем мы передаем новый пакет данных и повторяем эти шаги.

Перенести эти шаги платформу `NeuralNetwork` очень просто:

1. Получаем объекты `ndarray` `X` и `y` в качестве входных данных.
2. Передаем `X` по слоям.
3. Используем `Loss` для получения значения потерь и градиента потерь для обратного прохода.
4. Используем градиент потерь в качестве входных данных для метода `backward`, вычисления `param_grads` для каждого слоя в сети.
5. Вызываем на каждом слое функцию `update_params`, которая будет брать скорость обучения для `NeuralNetwork`, а также только что рассчитанные `param_grads`.

Наконец, у нас есть полное определение нейронной сети, на которой можно выполнять пакетное обучение. Теперь напишем код.

Нейронная сеть: код

Код выглядит весьма просто:

```
class NeuralNetwork(object):
    '''
    Класс нейронной сети.
    '''
    def __init__(self, layers: List[Layer],
                 loss: Loss,
                 seed: float = 1)
        '''
        Нейросети нужны слои и потери.
        '''
```

```
        self.layers = layers
        self.loss = loss
        self.seed = seed
        if seed:
            for layer in self.layers:
                setattr(layer, "seed", self.seed)

    def forward(self, x_batch: ndarray) -> ndarray:
        '''
        Передача данных через последовательность слоев.
        '''
        x_out = x_batch
        for layer in self.layers:
            x_out = layer.forward(x_out)

        return x_out

    def backward(self, loss_grad: ndarray) -> None:
        '''
        Передача данных назад через последовательность слоев.
        '''
        grad = loss_grad
        for layer in reversed(self.layers):
            grad = layer.backward(grad)

        return None

    def train_batch(self,
                    x_batch: ndarray,
                    y_batch: ndarray) -> float:
        '''
        Передача данных вперед через последовательность слоев.
        Вычисление потерь.
        Передача данных назад через последовательность слоев.
        '''

        predictions = self.forward(x_batch)

        loss = self.loss.forward(predictions, y_batch)

        self.backward(self.loss.backward())
```

```
        return loss

    def params(self):
        '''
        Получение параметров нейросети.
        for layer in self.layers:
            yield from layer.params

    def param_grads(self):
        '''
        Получение градиента потерь по отношению к параметрам нейросети.
        '''
        for layer in self.layers:
            yield from layer.param_grads
```

С помощью этого класса `NeuralNetwork` мы можем реализовать модели из предыдущей главы модульным, гибким способом и определить другие модели для представления сложных нелинейных отношений между входом и выходом. Например, ниже показано, как создать две модели, которые мы рассмотрели в предыдущей главе: линейную регрессию и нейронную сеть[1]:

```
linear_regression = NeuralNetwork(
   layers=[Dense(neurons = 1)],
           loss = MeanSquaredError(),
           learning_rate = 0.01
           )

neural_network = NeuralNetwork(
   layers=[Dense(neurons=13,
                 activation=Sigmoid()),
           Dense(neurons=1,
                 activation=Linear())],
           loss = MeanSquaredError(),
           learning_rate = 0.01
           )
```

Почти готово. Теперь мы просто многократно передаем данные через сеть, чтобы модель начала учиться. Но чтобы сделать этот процесс понятнее и проще в реализации для более сложных сценариев глубокого обучения,

[1] Скорость обучения 0.01 найдена оптимальной путем экспериментов.

надо определить другой класс, который будет выполнять обучение, а также дополнительный класс, который выполняет «обучение» или фактическое обновление параметров `NeuralNetwork` с учетом градиентов, вычисленных при обратном проходе. Давайте определим эти два класса.

Trainer и Optimizer

Есть сходство между этими классами и кодом, который мы использовали для обучения сети в главе 2. Там для реализации четырех шагов, описанных ранее для обучения модели, мы использовали следующий код:

```
# Передаем X_batch вперед и вычисляем потери
forward_info, loss = forward_loss(X_batch, y_batch, weights)

# Вычисляем градиент потерь по отношению к каждому весу
loss_grads = loss_gradients(forward_info, weights)

# обновляем веса
for key in weights.keys():
    weights[key] -= learning_rate * loss_grads[key]
```

Этот код находился внутри цикла `for`, который неоднократно передавал данные через функцию, определяющую и обновляющую нашу сеть.

Теперь, когда у нас есть нужные классы, мы, в конечном счете, сделаем это внутри функции подгонки в классе `Trainer`, который в основном будет оберткой вокруг функции `train`, использованной в предыдущей главе. (Полный код для этой главы в можно найти на странице книги на GitHub.) Основное отличие состоит в том, что внутри этой новой функции первые две строки из предыдущего блока кода будут заменены этой строкой:

```
neural_network.train_batch (X_batch, y_batch)
```

Обновление параметров, которое происходит в следующих двух строках, будет происходить в отдельном классе `Optimizer`. И наконец, цикл `for`, который ранее охватывал все это, будет выполняться в классе `Trainer`, который оборачивает `NeuralNetwork` и `Optimizer`.

Далее обсудим, почему нам нужен класс `Optimizer` и как он должен выглядеть.

Optimizer

В модели, которую мы описали в предыдущей главе, у слоев есть простое правило для обновления весов на основе параметров и их градиентов. В следующей главе мы узнаем, что есть множество других правил обновления. Например, некоторые правила учитывают данные не только от текущего набора данных, но и от всех предыдущих. Создание отдельного класса `Optimizer` даст нам гибкость в замене одного правила обновления на другое, что мы более подробно рассмотрим в следующей главе.

Описание и код

Базовый класс `Optimizer` будет принимать `NeuralNetwork`, и каждый раз, когда вызывается пошаговая функция `step`, будет обновлять параметры сети на основе их текущих значений, их градиентов и любой другой информации, хранящейся в `Optimizer`:

```
class Optimizer(object):
    '''
    Базовый класс оптимизатора нейросети.
    '''
    def __init__(self,
        lr: float = 0.01):
        '''
        У оптимизатора должна быть начальная скорость обучения.
        '''
        self.lr = lr


    def step(self) -> None:
        '''
        У оптимизатора должна быть функция "step".
        '''
        pass
```

И вот как это выглядит с простым правилом обновления (*стохастический градиентный спуск*):

```
class SGD(Optimizer):
    '''
    Стохастический градиентный оптимизатор.
    '''
    def __init__(self,
```

```
                lr: float = 0.01) -> None:
        '''пока ничего'''
        super().__init__(lr)

    def step(self):
        '''
        Для каждого параметра настраивается направление, при этом
        амплитуда регулировки зависит от скорости обучения.
        '''
        for (param, param_grad) in zip(self.net.params(),
                                       self.net.param_grads()):

            param -= self.lr * param_grad
```

Обратите внимание, что хотя наш класс `NeuralNetwork` не имеет метода `_update_params`, мы используем методы `params()` и `param_grads()` для извлечения правильных `ndarrays` для оптимизации.

Класс `Optimizer` готов, теперь нужен `Trainer`.

Trainer

Помимо обучения модели класс `Trainer` также связывает `NeuralNetwork` с `Optimizer`, гарантируя правильность обучения. Вы, возможно, заметили в предыдущем разделе, что мы не передавали нейронную сеть при инициализации нашего оптимизатора. Вместо этого мы назначим `NeuralNetwork` атрибутом `Optimizer` при инициализации класса `Trainer`:

```
setattr(self.optim, 'net', self.net)
```

В следующем подразделе я покажу упрощенную, но рабочую версию класса `Trainer`, которая пока содержит только метод `fit`. Этот метод выполняет несколько эпох обучения и выводит значение потерь после некоторого заданного числа эпох. В каждую эпоху мы будем:

- перемешивать данные в начале эпохи;
- передавать данные через сеть в пакетном режиме, обновляя параметры.

Эпоха заканчивается, когда мы пропускаем весь обучающий набор через `Trainer`.

Код класса Trainer

Ниже приведен код простой версии класса `Trainer`. Мы скрываем два вспомогательных метода, использующихся во время выполнения функции `fit`: `generate_batches`, генерирующий пакеты данных из `X_train` и `y_train` для обучения, и `per mute_data`, перемешивающий `X_train` и `y_train` в начале каждой эпохи. Мы также включили аргумент `restart` в функцию `train`: если он имеет значение `True` (по умолчанию), то будет повторно инициализировать параметры модели в случайные значения при вызове функции `train`:

```
class Trainer(object):
    '''
    Обучение нейросети.
    '''
    def __init__(self,
                 net: NeuralNetwork,
                 optim: Optimizer)
        '''
        Для обучения нужны нейросеть и оптимизатор. Нейросеть
         назначается атрибутом экземпляра оптимизатора.
        '''
        self.net = net
        setattr(self.optim, 'net', self.net)

    def fit(self, X_train: ndarray, y_train: ndarray,
            X_test: ndarray, y_test: ndarray,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32,
            seed: int = 1,
            restart: bool = True) -> None:
        '''
        Подгонка нейросети под обучающие данные за некоторое число
        эпох. Через каждые eval_every эпох выполняется оценка.
        '''
        np.random.seed(seed)

        if restart:
            for layer in self.net.layers:
                layer.first = True
```

```
        for e in range(epochs):

            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self.generate_batches(X_train, y_train,
                                                    batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):

                self.net.train_batch(X_batch, y_batch)

                self.optim.step()

            if (e+1) % eval_every == 0:

                test_preds = self.net.forward(X_test)

                loss = self.net.loss.forward(test_preds, y_test)

                print(f"Validation loss after {e+1} epochs is
                     {loss:.3f}")
```

В полной версии этой функции в хранилище GitHub книги (https://oreil.ly/2MV0aZI) мы также реализовали *раннюю остановку*, которая выполняет следующие действия:

1. Сохраняет значение потерь каждые `eval_every` эпох.
2. Проверяет, стали ли потери меньше после настройки.
3. Если потери не стали ниже, модель откатывается на шаг назад.

Теперь у нас есть все необходимое для обучения этих моделей!

Собираем все вместе

Ниже приведен код для обучения сети с использованием всех классов `Trainer` и `Optimizer` и двух моделей, определенных ранее, — `linear_regression` и `neural_network`. Мы установим скорость обучения равной 0.01, максимальное количество эпох — 50 и будем оценивать наши модели каждые 10 эпох:

```
optimizer = SGD(lr=0.01)
trainer = Trainer(linear_regression, optimizer)

trainer.fit(X_train, y_train, X_test, y_test,
            epochs = 50,
            eval_every = 10,
            seed=20190501);

Validation loss after 10 epochs is 30.295
Validation loss after 20 epochs is 28.462
Validation loss after 30 epochs is 26.299
Validation loss after 40 epochs is 25.548
Validation loss after 50 epochs is 25.092
```

Использование тех же функций оценки моделей из главы 2 и помещение их в функцию `eval_regression_model` дают нам следующие результаты:

```
eval_regression_model(linear_regression, X_test, y_test)

Mean absolute error: 3.52

Root mean squared error 5.01
```

Это похоже на результаты линейной регрессии, которую мы использовали в предыдущей главе, а это подтверждает, что наша структура работает.

Запустив тот же код с моделью `neural_network` со скрытым слоем с 13 нейронами, мы получим следующее:

```
Validation loss after 10 epochs is 27.434
Validation loss after 20 epochs is 21.834
Validation loss after 30 epochs is 18.915
Validation loss after 40 epochs is 17.193
Validation loss after 50 epochs is 16.214

eval_regression_model(neural_network, X_test, y_test)

Mean absolute error: 2.60

Root mean squared error 4.03
```

Опять же, эти результаты похожи на те, что мы видели в предыдущей главе, и они значительно лучше, чем линейная регрессия.

Наша первая модель глубокого обучения (с нуля)

С настройками покончено, запустим первую модель:

```
deep_neural_network = NeuralNetwork(
    layers=[Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=13,
                  activation=Sigmoid()),
            Dense(neurons=1,
                  activation=LinearAct())],
   loss=MeanSquaredError(),
   learning_rate=0.01
)
```

Пока не будем фантазировать и просто добавим скрытый слой с той же размерностью, что и первый слой, так что наша сеть теперь имеет два скрытых слоя, каждый из которых содержит 13 нейронов.

Обучение с использованием той же скорости обучения и периодичности оценки, что и в предыдущих моделях, дает следующий результат:

```
Validation loss after 10 epochs is 44.134
Validation loss after 20 epochs is 25.271
Validation loss after 30 epochs is 22.341
Validation loss after 40 epochs is 16.464
Validation loss after 50 epochs is 14.604

eval_regression_model(deep_neural_network, X_test, y_test)

Mean absolute error: 2.45

Root mean squared error 3.82
```

Наконец-то мы перешли непосредственно к глубокому обучению, но тут уже будут реальные задачи, и без фокусов и хитростей наша модель глубокого обучения будет работать не намного лучше, чем простая нейронная сеть с одним скрытым слоем.

Что еще более важно, полученная структура легко расширяема. Мы могли бы весьма просто реализовать другие виды `Operation`, обернуть их в новые слои и сразу вставить их, предполагая, что в них определены `_output` и `_input_grad` и что размерности данных совпадают с размерностями их

соответствующих градиентов. Аналогично мы могли бы попробовать другие функции активации и посмотреть, уменьшит ли это показатели ошибок. Призываю взять наш код с GitHub (https://oreil.ly/deep-learning-github) и попробовать!

Заключение и следующие шаги

В следующей главе я расскажу о нескольких приемах, которые будут необходимы для правильной тренировки наших моделей, чтобы решать более сложные задачи[1]. Мы попробуем другие функции потерь и оптимизаторы. Я также расскажу о дополнительных приемах настройки скоростей обучения и их изменения в процессе обучения, а также покажу, как реализовать это в классах `Optimizer` и `Trainer`. Наконец, мы рассмотрим прореживание (Dropout), узнаем, что это такое и зачем оно нужно для повышения устойчивости обучения. Вперед!

[1] Даже в этой простой задаче незначительное изменение гиперпараметров может привести к тому, что модель глубокого обучения не справится с двухслойной нейронной сетью. Возьмите код с GitHub и попробуйте сами!

ГЛАВА 4

Расширения

Из первых трех глав мы узнали, что такое модели глубокого обучения и как они должны работать, а затем создали первую модель глубокого обучения и научили ее решать относительно простую задачу прогнозирования цен на жилье на основе некоторых признаков. Однако в реальных задачах успешно обучить модели глубокого обучения не так просто. В теории такие модели действительно позволяют найти оптимальное решение любой задачи, которую можно свести к обучению с учителем, но вот на практике не все так просто. Теория говорит о том, что данная архитектура модели позволяет найти оптимальное решение проблемы. Однако помимо теории есть хорошие и понятные методы, которые повышают вероятность удачного обучения нейронной сети, и эта глава именно об этом.

Начнем с математического анализа задачи нейронной сети: поиска минимума функции. Затем покажем ряд методов, которые помогут выполнить эту задачу, на классическом примере с рукописными цифрами. Мы начнем с функции потерь, которая используется в задачах классификации в глубоком обучении, и продемонстрируем, что она значительно ускоряет обучение (пока мы говорили только о регрессии, а функцию потерь и задачу классификации не рассматривали). Рассмотрим новые функции активации и покажем, как они влияют на обучение, а также поговорим об их плюсах и минусах. Далее рассмотрим самое важное (и простое) улучшение для уже знакомого нам метода стохастического градиентного спуска, а также кратко поговорим о возможностях продвинутых оптимизаторов. В заключение мы рассмотрим еще три метода улучшения работы нашей системы: снижение скорости обучения, инициализация весов и отсев. Как мы увидим, все эти методы помогут нашей нейронной сети находить оптимальные решения.

В первой главе мы сначала приводили рисунок со схемой, затем математическую модель, и потом код для рассматриваемой концепции. В этой главе какой-то конкретной схемы у методов не будет, поэтому мы сначала

дадим краткое описание метода, а затем перейдем к математике (которая будет намного проще, чем в первой главе). Завершим рассмотрение кодом, который реализует метод с использованием введенных нами ранее строительных блоков. Итак, начнем с краткого описания задачи нейронной сети: поиска минимума функции.

Немного о понимании нейронных сетей

Как мы уже видели, в нейронных сетях содержатся весовые коэффициенты, или веса. Значения весов и входные данные `x` и `y` позволяют вычислить «потери». На рис. 4.1 показана схема описанной нейронной сети.

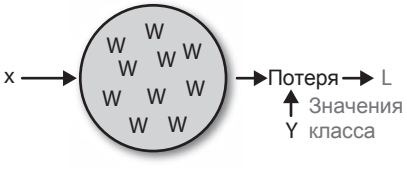


Рис. 4.1. Простое представление нейронной сети с весами

На самом деле каждый отдельный вес имеет некоторую сложную нелинейную связь с характеристиками `x`, целью `y`, другими весами и, в конечном счете, потерей `L`. Если мы построили график, варьируя значение веса, сохраняя постоянными значения с другими весами, `x` и `y`, и вычерчивая полученное значение потери `L`, мы могли видеть что-то вроде того, что показано на рис. 4.2.

Запуская обучение нейронной сети, мы задаем весам начальные значения в пределах, показанных на рис. 4.2. Затем, используя градиенты, которые рассчитываем во время обратного распространения, мы итеративно обновляем значения весов, основываясь на наклоне кривой и текущем значении веса[1]. На рис. 4.3 показана геометрическая интерпретация настройки весов нейронной сети в зависимости от градиента и скорости обучения. Слева стрелками показано, что это правило применяется многократно с меньшей скоростью обучения, чем в области справа. Обратите внима-

[1] Кроме того, как мы видели в главе 3, мы умножаем эти градиенты на скорость обучения, что позволяет точно контролировать процесс настройки весов.

ние, что в обоих случаях изменение значения пропорционально наклону кривой при данном значении веса (более крутой наклон означает более сильное изменение).

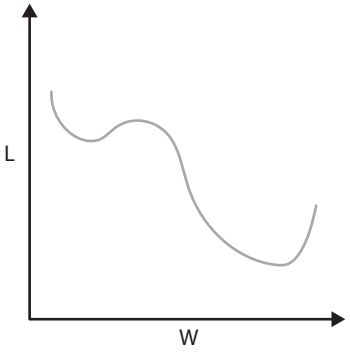


Рис. 4.2. Вес нейронной сети против ее потери

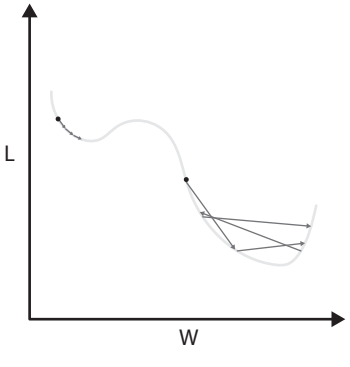


Рис. 4.3. Настройка весов нейронной сети в зависимости от значения градиента и скорости обучения

Конечная цель обучения модели — задать весам такие значения, чтобы значение потерь нашло глобальный минимум. Как видно из рис. 4.3, если шаг обучения слишком мал, мы рискуем попасть в локальный минимум, который будет менее оптимален, чем глобальный (этот сценарий показан синими стрелками). Если шаг слишком велик, мы рискуем «перепрыгнуть» глобальный минимум, даже если находимся рядом с ним (этот сценарий показан красными стрелками). Это самая главная дилемма на-

стройки скорости обучения: слишком малая скорость ведет к попаданию в локальный минимум, а слишком большая ведет к промаху.

На деле все еще сложнее. В нейронной сети могут быть тысячи, а может, и миллионы весов, и в этом случае мы ищем глобальный минимум в тысяче- или миллионмерном пространстве. Более того, поскольку мы на каждой итерации будем обновлять значения весов, а также передавать различные значения `X` и `y`, сама кривая, на которой мы ищем минимум, тоже постоянно меняется! Из-за этого нейронные сети много лет были объектом споров и скепсиса, так как казалось, что оптимальное решение найти невозможно. Ян Лекун и соавт. в статье 2015 года высказались следующим образом:

> В частности, было принято считать, что простой градиентный спуск оказывается пойман в локальный минимум, который не позволит уменьшить среднюю ошибку. На практике неоптимальные локальные минимумы редко являются проблемой для больших сетей. Независимо от начальных условий система почти всегда достигает решений очень похожего качества. Недавние теоретические и эмпирические результаты убедительно свидетельствуют о том, что локальные минимумы — это не такая уж и проблема.

На рис. 4.3 хорошо видно, почему скорость обучения не должна быть слишком большой или слишком маленькой, отсюда интуитивно понятно, почему приемы, которые мы собираемся изучить в этой главе, действительно работают. Теперь, понимая цель нейронных сетей, мы приступим к работе. Начнем с многопеременной логистической активационной функции кросс-энтропийных потерь, которая работает в значительной степени благодаря своей способности обеспечивать более крутые градиенты весов, чем функция среднеквадратичных потерь, которую мы видели в предыдущей главе.

Многопеременная логистическая функция активации с перекрестно-энтропийными потерями

В главе 3 в качестве функции потерь мы использовали среднеквадратическую ошибку (MSE). Функция обладала свойством выпуклости,

то есть чем дальше прогноз от цели, тем круче был бы начальный градиент, который класс `Loss` отправлял обратно в слои сети. Из-за этого увеличивались градиенты, полученные параметрами. Но оказывается, что в задачах классификации это не предел мечтаний, поскольку в этом случае значения, которые выводит наша сеть, должны интерпретироваться как вероятности в диапазоне от 0 до 1, и вектор вероятностей должен иметь равную 1 сумму для каждого наблюдения, которое мы передали через нашу сеть. Многопеременная логистическая функция активации (будем использовать сокращенное английское название — softmax) с перекрестно-энтропийными потерями позволяет за счет этого получить более крутые градиенты, чем средний квадрат ошибки, при тех же входных данных. У этой функции два компонента: первый — это функция `softmax`, а второй — это перекрестно-энтропийные потери. Рассмотрим их подробнее.

Компонент № 1: функция softmax

Для задачи классификации с N возможными категориями нейронная сеть выдаст вектор из N значений для каждого наблюдения. Для задачи с тремя категориями вектор может быть таким:

```
[5, 3, 2]
```

Математическое представление

Поскольку речь идет о задаче классификации, мы знаем, что результат следует интерпретировать как вектор вероятностей того, что данное наблюдение относится к категории 1, 2 или 3 соответственно. Один из способов преобразования этих значений в вектор вероятностей — нормализация, сложение и деление на сумму:

$$\text{Normalize}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \dfrac{x_1}{x_1 + x_2 + x_3} \\ \dfrac{x_2}{x_1 + x_2 + x_3} \\ \dfrac{x_3}{x_1 + x_2 + x_3} \end{bmatrix}.$$

Но есть способ, который дает более крутые градиенты и имеет пару тузов в рукаве, — `softmax`. Эта функция для вектора длины 3 будет иметь вид:

$$\text{Softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \frac{e^{x_1}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_2}}{e^{x_1}+e^{x_2}+e^{x_3}} \\ \frac{e^{x_3}}{e^{x_1}+e^{x_2}+e^{x_3}} \end{bmatrix}.$$

Наглядное представление

Суть функции `softmax` заключается в том, что она усиливает максимальное значение по сравнению с другими, заставляя нейронную сеть стать «чувствительнее» к прогнозам в задаче классификации. Давайте сравним результаты функций `normalize` и `softmax` для данного выше вектора вероятностей:

```
normalize(np.array([5,3,2]))

array([0.5, 0.3, 0.2])

softmax(np.array([5,3,2]))

array([0.84, 0.11, 0.04])
```

Видно, что исходное максимальное значение 5 выделилось сильнее, а два других стали ниже, чем после нормализации. Таким образом, функция `softmax` — это нечто среднее между нормализацией значений и фактическим применением функции `max` (которая в данном случае приведет к выводу массива (`[1.0, 0.0, 0.0]`)), отсюда и название «softmax» — «мягкий максимум».

Компонент № 2: перекрестно-энтропийная потеря

Напомним, что любая функция потерь берет на вход вектор вероятностей $\begin{bmatrix} p_1 \\ \dots \\ p_n \end{bmatrix}$ и вектор фактических значений $\begin{bmatrix} y_1 \\ \dots \\ y_n \end{bmatrix}$.

Математическое представление

Функция перекрестно-энтропийной потери для каждого индекса *i* в этих векторах:

$$CE(p_i, y_i) = -y_i \times \log(p_i) - (1 - y_i) \times \log(1 - p_i).$$

Наглядное представление

Чтобы понять, чем такая функция потерь хороша, отметим, что, поскольку каждый элемент *y* равен 0 или 1, предыдущее уравнение сводится к:

$$CE(p_i, y_i) = \begin{cases} -\log(1 - p_i) & при\ y_i = 0 \\ -\log(p_i) & при\ y_i = 1. \end{cases}$$

Теперь все проще. Если $y = 0$, то график зависимости значения этой потери от значения среднеквадратичной потери на интервале от 0 до 1 выглядит так, как показано на рис. 4.4.

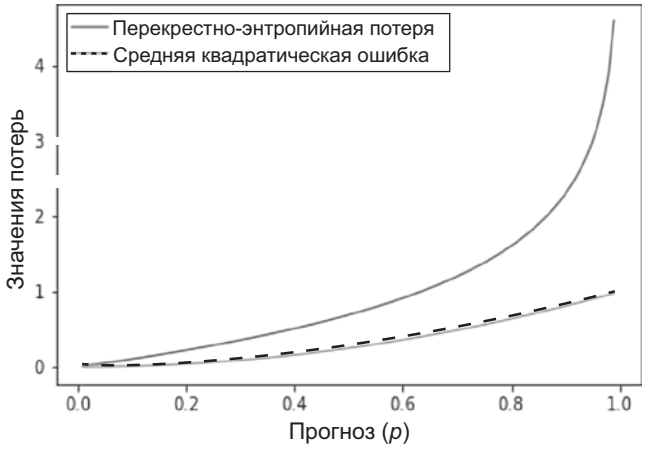


Рис. 4.4. Сравнение перекрестно-энтропийной функции потерь и СКО при $y = 0$

Штрафы у перекрестно-энтропийной функции потерь намного выше[1], и кроме того, они становятся больше с увеличением скорости, стремясь

[1] Мы можем быть более конкретными: среднее значение $-log(1 - x)$ за интервал от 0 до 1 оказывается равным 1, тогда как среднее значение x^2 за тот же интервал составляет всего 1/3.

к бесконечности, когда разница между нашим прогнозом и целью стремится к 1! График для случая $y = 1$ похож, только «перевернут» (то есть он повернут на 180 градусов вокруг линии $x = 0.5$).

Таким образом, для задач, где результат должен лежать между 0 и 1, перекрестно-энтропийная функция потерь создает более крутые градиенты, чем MSE. Но настоящая магия происходит, когда мы объединяем эту функцию потерь с функцией `softmax` — сначала проводим выход нейронной сети через функцию `softmax`, чтобы нормализовать его до нужного диапазона, а затем подаем полученные вероятности в перекрестно-энтропийную функцию потерь.

Давайте посмотрим, как это выглядит в сценарии с тремя категориями, который мы уже упоминали. Выражение для компонента вектора потерь $i = 1$, то есть первого компонента потерь для данного наблюдения, которое мы обозначим как SCE_1, имеет вид:

$$SCE_1 = -y_1 \times \log\left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right) - (1 - y_1) \times \log\left(1 - \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}\right).$$

Это выражение дает более сложный градиент для данной функции потерь. Но есть более элегантное выражение, которое легко и записать, и реализовать:

$$\frac{\partial SCE_1}{\partial x_1} = \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}} - y_1.$$

Это означает, что полный градиент перекрестно энтропийной функции с `softmax` равен:

$$\text{softmax}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) - \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Готово! Как и было обещано, итоговая реализация тоже будет простой:

```
softmax_x = softmax(x, axis = 1)
loss_grad = softmax_x – y
```

Теперь будем писать код.

Код

В главе 3 мы говорили, что классу `Loss` требуются два двумерных массива: один с прогнозами сети, а другой с целевыми значениями. Количество строк в каждом массиве представляет собой размер пакета, а количество столбцов — это число категорий n в задаче классификации. Каждая строка представляет собой наблюдение в наборе данных, причем значения n в строке — это вероятности принадлежности этого наблюдения к каждому из n классов. Таким образом, нам придется применить функцию `softmax` к каждой строке в массиве прогнозов. Так мы получаем первую потенциальную проблему: полученные числа мы будем передавать в функцию `log`, чтобы вычислить потери. Тут есть проблема, так как функция `log(x)` уходит в отрицательную бесконечность, когда x стремится к 0, а $1 - x$ стремится в бесконечность, когда x стремится к 1.

Соберем все вместе!

```
class SoftmaxCrossEntropyLoss(Loss):
    def __init__(self, eps: float=1e-9)
        super().__init__()
        self.eps = eps
        self.single_output = False

    def _output(self) -> float:

        # применение функции softmax к каждой строке (наблюдению)
        softmax_preds = softmax(self.prediction, axis=1)

        # захват выхода softmax, чтобы предотвратить неустойчивость
        self.softmax_preds = np.clip(softmax_preds, self.eps, 1 -
                                     self.eps)

        # вычисление потерь
    softmax_cross_entropy_loss = (
        -1.0 * self.target * np.log(self.softmax_preds) - \
           (1.0 - self.target) * np.log(1 - self.softmax_preds)
)

        return np.sum(softmax_cross_entropy_loss)

    def _input_grad(self) -> ndarray:

        return self.softmax_preds - self.target
```

Вскоре я покажу пару экспериментов с набором данных MNIST, которые покажут преимущества этой функции потерь перед MSE. Но сначала давайте обсудим компромиссы, связанные с выбором функции активации, и посмотрим, есть ли что-то получше сигмоиды.

Примечание о функциях активации

В главе 2 мы утверждали, что сигмоида — хорошая функция активации, потому что:

- она нелинейная и монотонная;
- вносит в модель некоторую нормализацию, приводя промежуточные элементы в диапазон от 0 до 1.

Однако у сигмоиды есть и недостаток, как у MSE: она производит относительно плоские градиенты во время обратного прохода. Градиент, который передается в сигмовидную функцию (или любую другую функцию) на обратном проходе, показывает, насколько выходные данные функции в конечном итоге влияют на потери. Поскольку максимальный наклон сигмовидной функции равен 0,25, градиенты в лучшем случае будут разделены на 4 при передаче к предыдущей операции в модели. Хуже того, когда вход сигмоиды меньше −2 или больше 2: градиент, который получают эти входы, будет почти равен 0, так как функция `sigmoid(x)` почти плоская при $x = -2$ или $x = 2$. Это означает, что любые параметры, влияющие на эти входные данные, получат небольшие градиенты и сеть будет учиться медленно[1]. Кроме того, если в слоях нейронной сети используется несколько сигмоид одна за другой, эта проблема будет усугубляться, что приведет к дальнейшему уменьшению градиентов.

А какой будет функция активации, у которой будут противоположные преимущества и недостатки?

---

[1] Для понимания: представьте, что вес w вносит вклад в функцию f (так что $f = w \times x_1 + ...$) и во время прямого прохода нашей нейронной сети $f = -10$ для некоторых наблюдений. Поскольку функция `sigmoid(x)` является такой плоской при `x = -10`, изменение значения w почти не повлияет на прогноз модели и, следовательно, на потери.

Другая крайность: Rectified Linear Unit

Функция активации Recified Linear Unit (блок линейной ректификации), или ReLU, — это популярная функция, в своих плюсах и минусах противоположная сигмоиде. ReLU равна 0, если x меньше 0, и x в противном случае. Схема показана на рис. 4.5.

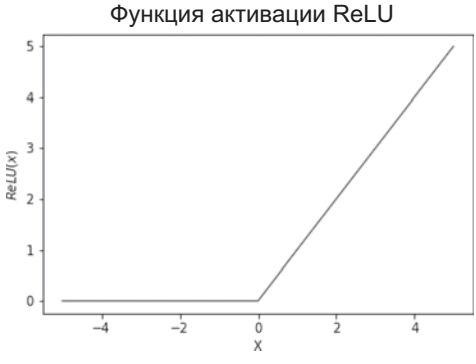


Рис. 4.5. Функция активации ReLU

Эта функция активации монотонная и нелинейная. Она создает более крутые градиенты, чем сигмоида: 1, если входные данные для функции больше 0, и 0 в противном случае, то есть среднее значение равно 0.5, тогда как максимальный градиент сигмоиды равен 0.25. Функция активации ReLU популярна в нейронных сетях глубокого обучения, потому что ее недостаток (а именно тот факт, что она генерирует резкое и несколько произвольное различие между значениями, меньшими или большими, чем 0) компенсируется другими методами, а вот ее преимущество (генерация больших градиентов) имеет решающее значение для настройки весов.

Но есть некоторая золотая середина между этими двумя функциями активации, которую мы будем использовать в этой главе: гиперболический тангенс.

Золотая середина: гиперболический тангенс

Функция гиперболического тангенса (tanh) по форме аналогична сигмоиде, но отображает входные данные в значения от –1 до 1 (рис. 4.6).

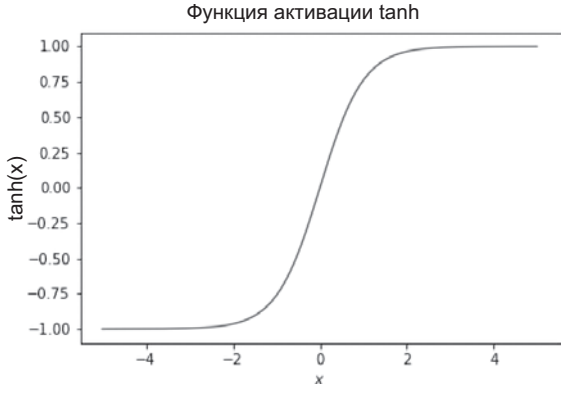


Рис. 4.6. Функция активации tanh

Эта функция дает значительно более крутые градиенты, чем сигмоида, — максимальный градиент tanh равен 1 в отличие от 0.25 у сигмоиды. На рис. 4.7 показаны градиенты этих двух функций.

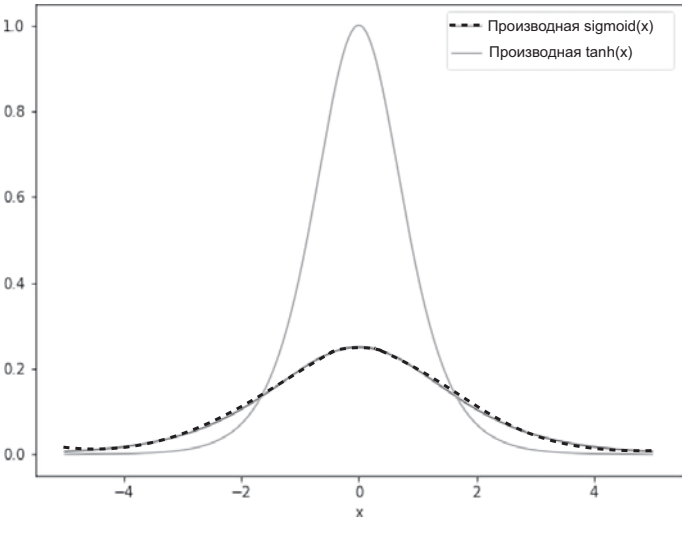


Рис. 4.7. Сигмоидная производная против производной tanh

Кроме того, точно так же, как $f(x) = sigmoid(x)$ имеет простую производную $f'(x) = sigmoid(x) \times (1 - sigmoid(x))$, так и $f(x) = tanh\ (x)$ имеет простую производную $f'(x) = 1 - tanh(x)^2$.

Выбор функции активации всегда влечет за собой компромиссы независимо от архитектуры: нам нужна функция активации, которая позволит нашей сети изучать нелинейные отношения между входом и выходом, не добавляя при этом лишней сложности. Например, функция активации «Leaky ReLU» допускает небольшой отрицательный наклон, когда вход для функции ReLU меньше 0, что позволяет функции ReLU отправлять градиенты назад, а функция активации «ReLU6» перекрывает положительный конец ReLU на уровне 6, добавляя сети нелинейности. Тем не менее обе эти функции активации сложнее, чем ReLU, и для простой задачи чересчур сложные функции активации могут усложнить обучение сети. Таким образом, в моделях, которые мы будем показывать далее в этой книге, мы будем использовать функцию активации tanh, которая является разумным компромиссом.

Теперь, когда мы выбрали функцию активации, опробуем ее в деле.

Эксперименты

Вернемся к теме из начала главы и покажем, почему перекрестно-энтропийная функция потерь `softmax` настолько распространена в глубоком обучении[1]. Мы будем использовать набор данных MNIST, представляющий собой черно-белые изображения написанных от руки цифр размером 28×28 пикселей, каждый из которых окрашен в диапазоне от 0 (белый) до 255 (черный). Кроме того, этот набор данных разделен на обучающий набор из 60 000 изображений и тестовый набор из 10 000 дополнительных изображений. В репозитории этой книги на GitHub (https://oreil.ly/2H7rJvf) есть дополнительная функция для считывания изображений и их меток в обучающие и тестовые наборы:

```
X_train, y_train, X_test, y_test = mnist.load()
```

Наша цель — обучить нейронную сеть распознавать цифру на изображении.

[1] Например, в учебнике по классификации MNIST TensorFlow используется функция `softmax_cross_entropy_with_logits`, а `nn.CrossEntropyLoss` в PyTorch фактически вычисляет внутри нее функцию `softmax`.

Предварительная обработка данных

Для выполнения классификации нужно будет преобразовать наши векторы меток изображений в `ndarray` той же формы, что и прогнозы. Метка «0» будет сопоставляться с вектором с единичным значением первого элемента и 0 на остальных позициях, метка «1» превращается в вектор 1 на второй позиции (с индексом 1) и т. д. (https://oreil.ly/2KTRm3z):

$$[0,\ \ 2,\ \ 1] \Rightarrow \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \end{bmatrix}.$$

Наконец, всегда полезно масштабировать данные так, чтобы среднее значение равнялось 0 и дисперсия равнялась 1, как мы это делали в предыдущих главах. Однако в данном случае каждая точка данных является изображением, и мы не будем выполнять такое масштабирование, поскольку это приведет к изменению значений соседних пикселей, что, в свою очередь, может привести к искажению изображения! Вместо этого мы выполним глобальное масштабирование нашего набора данных путем вычитания общего среднего и деления на общую дисперсию (обратите внимание, что мы используем статистику из обучающего набора для масштабирования тестового набора):

```
X_train, X_test = X_train - np.mean(X_train), X_test - np.mean(X_train)
X_train, X_test = X_train / np.std(X_train), X_test / np.std(X_train)
```

Модель

Наша модель должна выдавать 10 выходов для каждого входа. Каждый выход представляет собой вероятность принадлежности изображения к одному из 10 классов. Поскольку на выходе мы получаем вероятность, на последнем слое будем использовать сигмоидную функцию активации. В этой главе мы хотим проиллюстрировать, действительно ли всякие трюки по улучшению обучения помогают, поэтому мы будем использовать двухслойную нейронную сеть с количеством нейронов в скрытом слое, близким к среднему геометрическому числу наших входов (784) и количеству выходов (10): $89 \approx \sqrt{784 \times 10}$.

Теперь давайте сравним сеть, обученную с помощью функции MSE, с сетью, в которой используется перекрестно-энтропийная функция потерь

softmax. Значения потерь приведены для каждого наблюдения (напомним, что в среднем перекрестно-энтропийные значения потерь будут втрое больше, чем MSE). Если мы запустим:

```
model = NeuralNetwork(
    layers=[Dense(neurons=89,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Sigmoid())],
            loss = MeanSquaredError(),
seed=20190119)

optimizer = SGD(0.1)

trainer = Trainer(model, optimizer)
trainer.fit(X_train, train_labels, X_test, test_labels,
            epochs = 50,
            eval_every = 10,
            seed=20190119,
            batch_size=60);

calc_accuracy_model(model, X_test)
```

это даст нам:

```
Validation loss after 10 epochs is 0.611
Validation loss after 20 epochs is 0.428
Validation loss after 30 epochs is 0.389
Validation loss after 40 epochs is 0.374
Validation loss after 50 epochs is 0.366

The model validation accuracy is: 72.58%
```

Теперь давайте проверим наш собственный тезис: перекрестно-энтропийная функция потери softmax поможет нашей модели учиться быстрее.

Эксперимент: перекрестно-энтропийная функция потери softmax

Сначала давайте изменим предыдущую модель:

```
model = NeuralNetwork(
     layers=[Dense(neurons=89,
                   activation=Tanh()),
```

```
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20190119)
```

Поскольку теперь мы пропускаем выходные данные модели через функцию softmax, сигмоида уже не нужна.

Запустим обучение на 50 эпох, что дает нам следующие результаты:

```
Validation loss after 10 epochs is 0.630
Validation loss after 20 epochs is 0.574
Validation loss after 30 epochs is 0.549
Validation loss after 40 epochs is 0.546
Loss increased after epoch 50, final loss was 0.546, using the model
from epoch 40

The model validation accuracy is: 91.01%
```

Действительно, замена нашей функции потерь на функцию с более крутыми градиентами дает огромное повышение точности нашей модели[1].

Мы можем добиться большего, даже не меняя нашу архитектуру. В следующем разделе мы рассмотрим импульс — самое важное и простое улучшение для стохастического градиентного спуска, который мы использовали до сих пор.

Импульс

До сих пор мы использовали только одно правило обновления весовых коэффициентов — мы просто брали производную потери по весам и смещали значения весов в направлении градиента. Это означает, что наша функция `_update_rule` в оптимизаторе выглядела так:

[1] Вы могли бы сказать, что перекрестно-энтропийная функция потери `softmax` побеждает нечестно, так как функция `softmax` нормализует полученные значения, тогда как СКО просто получает 10 входов, пропущенных через сигмоиду без нормализации. Однако на сайте https://oreil.ly/2H7rJvf я показал, что MSE работает хуже, чем перекрестно-энтропийная функция потери, даже после нормализации входных данных.

```
update = self.lr*kwargs['grad']
kwargs['param'] -= update
```

Для начала объясним, зачем нам нужно что-то здесь менять и вводить понятие инертности.

Об инертности наглядно

Вспомните рис. 4.3, на котором построено значение отдельного параметра в зависимости от значения потерь в сети. Представьте, что значение параметра постоянно обновляется в одном и том же направлении и значение потери продолжает уменьшаться с каждой итерацией. Рабочая точка движется вниз по склону, и величина обновления на каждом временном шаге будет аналогична «скорости» параметра. Однако в реальном мире объекты останавливаются и меняют направление не мгновенно, потому что у них есть инерция. То есть скорость объекта в данный момент является не только функцией сил, действующих на тело в данный момент, но и накопленных ранее скоростей. Пользуясь этой аналогией, мы далее введем в нашу модель инертность.

Реализация инертности в классе оптимизатора

Внедрение импульса или инерции означает, что величина обновления на каждом шаге будет вычисляться как средневзвешенное значение от предыдущих обновлений, причем веса будут уменьшаться в геометрической прогрессии от недавних к давним. Тогда нам надо будет задать степень затухания, которая определяет, насколько величина обновления зависит от накопленной скорости или от текущего значения.

Математическое представление

Математически, если наш параметр инертности равен μ, а градиент на каждом временном шаге равен ∇_t, наше обновление веса будет иметь вид:

$$\text{обновление} = \nabla_t + \mu \times \nabla_{t-1} + \mu^2 \times \nabla_{t-2} + \ldots$$

Например, если бы наш параметр инерции был равен 0.9, мы бы умножили градиент предыдущего значения на 0.9 градиент двух шагов назад — на $0.9^2 = 0.81$, трех шагов назад — на $0.9^3 = 0.729$ и т. д. После этого мы бы

прибавили все это к градиенту от текущего временного шага, чтобы получить обновленное значение общего веса для текущего временного шага.

Код

Как это реализовать? Вычислять бесконечную сумму всякий раз, когда мы хотим обновить значения весов? Нет, все проще. Наш `Optimizer` будет отслеживать лишь некоторое количество предыдущих шагов и прибавлять их к текущему. Затем на каждом временном шаге мы будем использовать текущий градиент, изменяя вес и смещая накопленные значения. С физикой тут мало связи, поэтому мы будем называть эту величину «скоростью».

Как обновлять скорость? В два этапа:

1. Умножаем скорость на параметр инерции.
2. Добавляем градиент.

Тогда начиная с $t = 1$ скорость будет иметь значения:

1. ∇_1
2. $\nabla_2 + \mu \times \nabla_1$
3. $\nabla_3 + \mu \times (\nabla_2 + \mu \times \nabla_1) = \mu \times (\nabla_2 + \mu^2 \times \nabla_1)$

Саму скорость можно использовать для обновления параметров. Добавим это в новый подкласс `Optimizer`, который мы назовем `SGDMomentum`. В этом классе будут функции `step` и `_update_rule`:

```
def step(self) -> None:
    '''
    если итерация первая: инициализируем «скорости» для всех параметров.
    иначе вызываем функцию _update_rule.
    '''
    if self.first:
        # задаем скорости для первой итерации
        self.velocities = [np.zeros_like(param)
                           for param in self.net.params()]
        self.first = False

    for (param, param_grad, velocity) in zip(self.net.params(),
                                             self.net.param_grads(),
                                             self.velocities):
```

```
        # передаем скорости в функцию "_update_rule"
        self._update_rule(param=param,
                          grad=param_grad,
                          velocity=velocity)

def _update_rule(self, **kwargs) -> None:
    '''
    обновление по скорости и инерции.
    '''

    # обновление скорости
    kwargs['velocity'] *= self.momentum
    kwargs['velocity'] += self.lr * kwargs['grad']

    # Обновляем параметры
    kwargs['param'] -= kwargs['velocity']
```

Посмотрим, поможет ли новый оптимизатор процессу обучения.

Эксперимент: стохастический градиентный спуск с инерцией

Выполним обучение той же нейронной сети с одним скрытым слоем на наборе данных MNIST, используя оптимизатор `SGDMomentum(lr = 0.1, momentum = 0.9)` вместо `SGD(lr = 0.1)`:

```
Validation loss after 10 epochs is 0.441
Validation loss after 20 epochs is 0.351
Validation loss after 30 epochs is 0.345
Validation loss after 40 epochs is 0.338
Loss increased after epoch 50, final loss was 0.338, using the model
from epoch 40

The model validation accuracy is: 95.51%
```

Видно, что потери стали значительно ниже, а точность стала значительно выше, а значит, инерция сделала свое дело[1]!

[1] Более того, инерция не единственный способ использовать информацию за пределами градиента текущей партии данных для обновления параметров. Мы кратко рассмотрим другие правила обновления в приложении А, а реализованы они в библиотеке Lincoln, которую мы добавили на репозиторий GitHub книги (https://oreil.ly/2MhdQ1B).

В качестве альтернативы мы могли бы изменять скорость обучения, причем не только вручную, но и автоматически, по некоторому правилу.

Наиболее распространенные из таких правил рассмотрим далее.

Скорость обучения

> [Скорость обучения] часто является самым важным гиперпараметром, и ее всегда нужно настраивать правильно.
>
> *Иешуа Бенжио, «Practical recommendations for gradient-based training of deep architectures», 2012*

Причина снижения скорости обучения была показана на рис. 4.3 в предыдущем разделе: в начале обучения удобнее делать «большие» шаги, но по мере обучения мы рано или поздно достигнем точки, в которой начнем «пропускать» минимум. Но такая проблема вообще может не возникнуть, так как, если соотношение между нашими весами и потерями «плавно снижается» по мере приближения к минимуму, как на рис. 4.3, величина градиентов будет автоматически уменьшаться при уменьшении наклона. Но такая ситуация может и не произойти, и в любом случае снижение скорости обучения может дать нам более тонкий контроль над процессом.

Типы снижения скорости обучения

Существуют разные способы снижения скорости обучения. Самый простой: линейное затухание, при котором скорость обучения линейно уменьшается от первоначального значения до некоторого конечного значения в конце каждой эпохи. То есть на шаге t, если скорость обучения, с которой мы хотим начать, равна α_{start}, а конечная скорость обучения равна α_{end}, то скорость обучения на каждом временном шаге равна:

$$\alpha_t = \alpha_{start} - \left(\alpha_{start} - \alpha_{end}\right) \times \frac{t}{N},$$

где N — общее число эпох.

Еще один рабочий метод — экспоненциальное затухание, при котором скорость обучения снижается на некоторую долю. Формула простая:

$$\alpha_t = \alpha \times \delta^t,$$

где

$$\delta = \frac{\alpha_{end}^{\frac{1}{N-1}}}{\alpha_{start}}.$$

Реализовать это просто: у класса `Optimizer` будет атрибут конечной скорости обучения `final_lr`, до которой начальная скорость обучения будет постепенно снижаться:

```
def __init__(self,
             lr: float = 0.01,
             final_lr: float = 0,
             decay_type: str = 'exponential')
    self.lr = lr
    self.final_lr = final_lr
    self.decay_type = decay_type
```

В начале обучения мы можем вызвать функцию `_setup_decay`, которая вычисляет, как сильно скорость обучения будет снижаться в каждую эпоху:

```
self.optim._setup_decay ()
```

Эти вычисления реализуют линейное и экспоненциальное затухание скорости обучения, которые мы только что видели:

```
def _setup_decay(self) -> None:

    if not self.decay_type:
        return
    elif self.decay_type == 'exponential':
        self.decay_per_epoch = np.power(self.final_lr / self.lr,
                                        1.0 / (self.max_epochs-1))
    elif self.decay_type == 'linear':
        self.decay_per_epoch = (self.lr – self.final_lr) /
                               (self.max_epochs-1)
```

В конце каждой эпохи уменьшаем скорость обучения:

```
def _decay_lr(self) -> None:

    if not self.decay_type:
        return

    if self.decay_type == 'exponential':
        self.lr *= self.decay_per_epoch

    elif self.decay_type == 'linear':
        self.lr -= self.decay_per_epoch
```

Наконец, мы будем вызывать функцию `_decay_lr` из класса `Trainer` в функции `fit` в конце каждой эпохи:

```
if self.optim.final_lr:
    self.optim._decay_lr()
```

Теперь проведем несколько экспериментов и посмотрим на результат.

Эксперименты: затухание скорости обучения

Попробуем обучить ту же модель, добавив затухание скорости обучения. Мы инициализируем скорость обучения таким образом, чтобы «средняя скорость обучения» за цикл была равна предыдущей скорости обучения, равной 0.1. При линейном затухании скорости обучения сделаем снижение с 0.15 до 0.05, а для экспоненциального затухания начальная скорость будет равна 0.2, а снижаться будет до 0.05. Для линейного затухания:

```
optimizer = SGDMomentum(0.15, momentum=0.9, final_lr=0.05,
                        decay_ type='linear')
```

получим:

```
Validation loss after 10 epochs is 0.403
Validation loss after 20 epochs is 0.343
Validation loss after 30 epochs is 0.282
Loss increased after epoch 40, final loss was 0.282, using the model
from epoch 30
The model validation accuracy is: 95.91%
```

Для экспоненциального затухания:

```
optimizer = SGDMomentum(0.2, momentum=0.9, final_lr=0.05,
                        decay_ type='exponential')
```

получим:

```
Validation loss after 10 epochs is 0.461
Validation loss after 20 epochs is 0.323
Validation loss after 30 epochs is 0.284
Loss increased after epoch 40, final loss was 0.284, using the model
from epoch 30
The model validation accuracy is: 96.06%
```

Потери в «лучших моделях» были равны 0.282 и 0.284, что значительно ниже, чем значение 0.338, полученное раньше!

Далее поговорим о том, как задавать начальные веса.

Инициализация весов

Как мы упоминали при разговоре о функциях активации, некоторые функции активации, такие как сигмоида и гиперболический тангенс, имеют самые крутые градиенты при нулевых входах, а затем быстро сглаживаются по мере удаления входов от 0. Это может привести к снижению эффективности этих функций, так как, если у многих входов значения далеки от 0, мы получим очень маленькие градиенты на обратном проходе.

Оказывается, это основная проблема в нейронных сетях, с которыми мы работаем. Рассмотрим скрытый слой в сети MNIST, о которой мы говорили. На этот слой приходит 784 единицы данных, а затем они умножаются на матрицу весов, в результате чего получается n нейронов (а затем при необходимости добавляется смещение для каждого нейрона). На рис. 4.8 показано распределение этих n значений в скрытом слое нашей нейронной сети (с 784 входами) до и после прохода через функцию активации `Tanh`.

После прохода через функцию активации большинство активаций приняли значения –1 или 1! Это связано с тем, что каждая функция математически определена как:

$$f_n = w_{1,n} \times x_1 + \ldots + w_{784,n} \times x_{784} + b_n .$$

Поскольку мы инициализировали каждый вес с дисперсией 1 ($\mathrm{Var}(w_{i,j}) = 1$ и $\mathrm{Var}(b_n) = 1$) и $\mathrm{Var}(X_1 + X_2) = \mathrm{Var}(X_1) + \mathrm{Var}(X_2)$ для независимых случайных величин X_1 и X_2, мы имеем:

$$\mathrm{Var}(f_n) = 785.$$

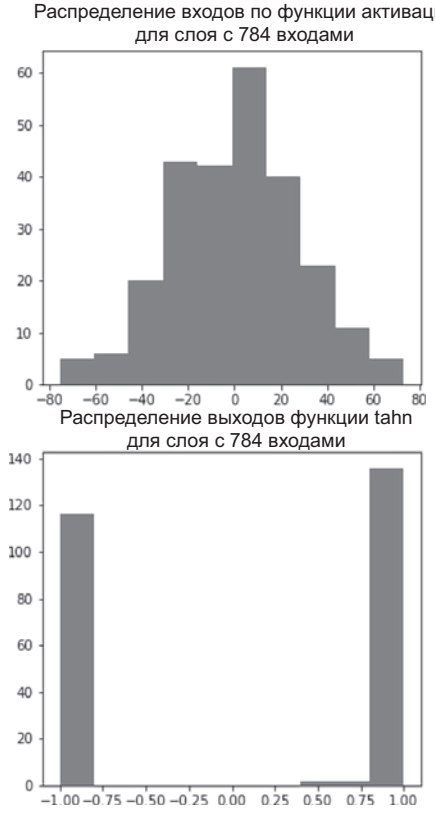


Рис. 4.8. Распределение входов по функции активации

Это дает стандартное отклонение ($\sqrt{785}$) чуть более 28, что отражает разброс значений, который показан в верхней половине рис. 4.8.

Кажется, у нас проблема. Но в том ли она заключается, что передаваемые в функции активации признаки не могут быть «слишком распределенными»? Если бы проблема была в этом, мы бы просто поделили признаки на некоторое значение, чтобы уменьшить их дисперсию. Но тогда очевидный вопрос: откуда мы знаем, на сколько делить? Ответ: значения следует масштабировать в зависимости от количества нейронов, передаваемых в слой. Если бы у нас была многослойная нейронная сеть и в одном слое было 200 нейронов, а в следующем 100, слой из 200 нейронов давал бы большее широкое распределение, чем 100 нейронов. Это нежелательно, так как мы не хотим, чтобы масштаб признаков, которые наша нейронная

сеть изучает во время обучения, зависел от их количества. Аналогично мы не хотим, чтобы прогнозы нашей сети зависели от масштаба входных признаков. Предсказания нашей модели не должны изменяться от умножения или деления всех признаков на 2.

Существует несколько способов исправить это, и здесь мы рассмотрим один наиболее распространенный вариант: мы можем отрегулировать начальную дисперсию весов по количеству нейронов в слоях, чтобы значения, передаваемые вперед следующему слою во время прямого прохода и назад во время обратного прохода, имели примерно одинаковый масштаб. Нельзя забывать об обратном проходе, поскольку в этом случае у нас та же проблема: дисперсия градиентов, которые слой получает во время обратного распространения, будет напрямую зависеть от количества объектов в следующем слое, поскольку именно он отправляет градиенты назад к рассматриваемому слою.

Математическое представление и код

Как конкретно решить проблему? Если в каждом слое есть n входных нейронов и n выходных нейронов, дисперсия для каждого веса, которая будет поддерживать постоянство дисперсии результирующих характеристик на прямом проходе, будет равна:

$$\frac{1}{n_{in}}.$$

Аналогично для обратного прохода:

$$\frac{1}{n_{out}}.$$

Компромисс, который часто называют инициализацией Глорота[1], выглядит так:

$$\frac{2}{n_{in} + n_{out}}.$$

Код будет простым — мы добавляем аргумент `weight_init` к каждому слою и добавляем следующее в нашу функцию `_setup_layer`:

[1] Метод был предложен Глоротом и Бенжио в статье 2010 года *Understanding the difficulty of training deep feedforward neural networks.*

```
if self.weight_init == "glorot":
    scale = 2/(num_in + self.neurons)
else:
    scale = 1.0
```

Теперь наши модели будут выглядеть так:

```
model = NeuralNetwork(
     layers=[Dense(neurons=89,
                   activation=Tanh(),
                   weight_init="glorot"),
             Dense(neurons=10,
                   activation=Linear(),
                   weight_init="glorot")],
             loss = SoftmaxCrossEntropy(),
seed=20190119)
```

при этом для каждого слоя задается `weight_init = "glorot"`.

Эксперименты: инициализация весов

Запустим все те же модели, но теперь с инициализацией Глорота:

```
Validation loss after 10 epochs is 0.352
Validation loss after 20 epochs is 0.280
Validation loss after 30 epochs is 0.244
Loss increased after epoch 40, final loss was 0.244, using the model
from epoch 30
The model validation accuracy is: 96.71%
```

для модели с линейным затуханием скорости обучения и

```
Validation loss after 10 epochs is 0.305
Validation loss after 20 epochs is 0.264
Validation loss after 30 epochs is 0.245
Loss increased after epoch 40, final loss was 0.245, using the model
from epoch 30
The model validation accuracy is: 96.71%
```

для модели с экспоненциальным затуханием скорости обучения. Снова имеет место существенное снижение потерь: с 0.282 и 0.284, которые были ранее, до 0.244 и 0.245!

Обратите внимание, что все эти фокусы не привели к увеличению времени обучения или размера нашей модели. Мы всего лишь подстроили процесс обучения, основываясь на понимании конечной цели нейронной сети, которое мы изложили ранее. В этой главе мы рассмотрим еще один прием. Вы, возможно, заметили, что *ни одна из моделей, которые мы использовали в этой главе, не была моделью глубокого обучения*. Это были просто нейронные сети с одним скрытым слоем. Дело в том, что без техники дропаута, которую мы сейчас изучим, трудно обучать модели глубокого обучения без переобучения.

Исключение, или дропаут

В этой главе мы изменили процедуру обучения нейронной сети, что позволило приблизиться к глобальному минимуму. Возможно, вы заметили, что мы не попробовали, казалось бы, самую очевидную вещь: добавить в сеть больше слоев или больше нейронов. Дело в том, что простое добавление «огневой мощи» в большинстве архитектур нейронных сетей лишь затруднит поиск хорошо обобщенного решения. Простое количественное расширение нейронной сети позволит ей моделировать *более* сложные отношения между входом и выходом, но также влечет за собой риск переобучения. *Методика исключения, или дропаута, позволила бы нам увеличивать мощь сети, в большинстве случаев снижая вероятность переобучения сети.*

Что же такое этот дропаут?

Определение

Дропаут — это случайный выбор некоторой части p нейронов в слое и установка их равными 0 во время прямого прохода. Этот странный трюк снижает мощь сети, но часто позволяет избавиться от переобучения. Это особенно верно в более глубоких сетях, где признаки представляют собой несколько уровней абстракции, удаленных из исходных элементов.

Несмотря на то что дропаут может помочь нашей сети избежать переобучения, мы все же хотим помочь ей делать правильные прогнозы, когда это будет нужно. Таким образом, операция `Dropout` будет иметь два режима: режим «обучения», в котором применяется дропаут, и режим «вывода», в котором его не будет. Тут будет другая проблема: применение

дропаута к слою уменьшает число передаваемых вперед значений в $1 - p$ раз в среднем, то есть веса в следующих слоях получат меньше значений с величиной M, вместо этого они получат величину $M * (1 - p)$. Мы хотим имитировать этот сдвиг при работе сети в режиме вывода, поэтому помимо отключения дропаута мы умножим все значения на $1 - p$.

В виде кода будет понятнее.

Код

Мы можем реализовать дропаут как отдельную операцию, которую мы добавим в конец каждого слоя. Это будет выглядеть следующим образом:

```
class Dropout(Operation):

    def __init__(self,
                 keep_prob: float = 0.8):
                 super().__init__()
                 self.keep_prob = keep_prob

    def _output(self, inference: bool) -> ndarray:
        if inference:
            return self.inputs * self.keep_prob
        else:
            self.mask = np.random.binomial(1, self.keep_prob,
                                           size=self.inputs.shape)
            return self.inputs * self.mask

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad * self.mask
```

На прямом проходе при применении дропаута мы сохраняем «маску», в которой хранится информация об обнуленных нейронах. На обратном проходе мы умножаем градиент, полученный операцией, на эту маску. Это происходит потому, что дропаут делает градиент равным 0 для обнуленных значений (поскольку изменение их значений теперь не будет влиять на потери), а другие градиенты оставим без изменений.

Учет дропаута в остальной модели

Возможно, вы заметили, что мы включили флаг `inference` в метод `_output`, который определяет, применяется ли дропаут. Чтобы этот флаг вызы-

вался правильно, мы должны добавить его в нескольких других местах во время обучения:

1. Методы пересылки `Layer` и `NeuralNetwork` будут принимать `inference` в качестве аргумента (по умолчанию равным `False`) и передавать флаг в каждую `Operation`, и каждая `Operation` будет вести себя поразному в зависимости от режима.
2. Напомним, что в классе `Trainer` мы оцениваем обученную модель на тестовом наборе через каждые `eval_every` эпох. Во время оценки мы каждый раз будем оценивать флаг `inference`, равный `True`:

   ```
   test_preds = self.net.forward(X_test, inference=True)
   ```

3. Наконец, мы добавляем ключевое слово `dropout` в класс `Layer`, и полная подпись функции `__init__` для класса `Layer` теперь выглядит следующим образом:

   ```
   def __init__(self,
                neurons: int,
                activation: Operation = Linear(),
                dropout: float = 1.0,
                weight_init: str = "standard")
   ```

   Мы добавляем операцию дропаута с помощью вот такой функции в `_setup_layer`:

   ```
    if self.dropout < 1.0:
        self.operations.append(Dropout(self.dropout))
   ```

Готово! Давайте посмотрим, как это работает.

Эксперимент: дропаут

Во-первых, видно, что добавление в модель дропаута действительно снижает потери. Добавим дропаут, равный 0.8 (чтобы 20% нейронов обнулилось), в первый слой, чтобы наша модель выглядела следующим образом:

```
mnist_soft = NeuralNetwork(
   layers=[Dense(neurons=89,
                 activation=Tanh(),
                 weight_init="glorot",
                 dropout=0.8),
           Dense(neurons=10,
```

```
                activation=Linear(),
                weight_init="glorot")],
           loss = SoftmaxCrossEntropy(),
seed=20190119)
```

и обучим модель с теми же гиперпараметрами, что и раньше (экспоненциальное снижение скорости обучения с 0.2 до 0.05):

```
Validation loss after 10 epochs is 0.285
Validation loss after 20 epochs is 0.232
Validation loss after 30 epochs is 0.199
Validation loss after 40 epochs is 0.196
Loss increased after epoch 50, final loss was 0.196, using the model
from epoch 40
The model validation accuracy is: 96.95%
```

Мы снова значительно уменьшили потери по сравнению с тем, что мы видели ранее: величина потери стала равна 0.196 по сравнению с 0.244 ранее.

Дропаут проявляет себя во всей красе, когда мы добавляем больше слоев. Давайте заменим модель, которую мы использовали в этой главе, на модель глубокого обучения. Пусть в первом скрытом слое будет в два раза больше нейронов, чем в скрытом слое ранее (178), а во втором скрытом слое — вдвое меньше (46). Наша модель выглядит так:

```
model = NeuralNetwork(
    layers=[Dense(neurons=178,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=46,
                  activation=Tanh(),
                  weight_init="glorot",
                  dropout=0.8),
            Dense(neurons=10,
                  activation=Linear(),
                  weight_init="glorot")],
                  loss = SoftmaxCrossEntropy(),
seed=20190119)
```

Обратите внимание на дропаут в первых двух слоях.

Обучив модель, увидим еще одно уменьшение потерь и повышение точности!

```
Validation loss after 10 epochs is 0.321
Validation loss after 20 epochs is 0.268
Validation loss after 30 epochs is 0.248
Validation loss after 40 epochs is 0.222
Validation loss after 50 epochs is 0.217
Validation loss after 60 epochs is 0.194
Validation loss after 70 epochs is 0.191
Validation loss after 80 epochs is 0.190
Validation loss after 90 epochs is 0.182
Loss increased after epoch 100, final loss was 0.182, using the
model from epoch 90
The model validation accuracy is: 97.15%
```

Но это улучшение невозможно без дропаута. Ниже приведены обучения той же модели без дропаута:

```
Validation loss after 10 epochs is 0.375
Validation loss after 20 epochs is 0.305
Validation loss after 30 epochs is 0.262
Validation loss after 40 epochs is 0.246
Loss increased after epoch 50, final loss was 0.246, using the model
from epoch 40
The model validation accuracy is: 96.52%
```

Без дропаута модель глубокого обучения работает хуже, чем модель с одним скрытым слоем, несмотря на вдвое большее число параметров и время обучения! Это показывает, насколько дропаут важен при обучении моделей глубокого обучения. Он сыграл важную роль в определении модели-победителя на ImageNet 2012 года, что положило начало современной эре глубокого обучения[1]. Без дропаута вы, возможно, не читали бы эту книгу!

Заключение

В этой главе мы разобрали некоторые из наиболее распространенных методов улучшения обучения нейронной сети, начав с описания ее работы и цели на низком уровне. В конце хотелось бы привести перечень того,

[1] Подробнее об этом в статье Г. Хинтона и соавт. *Improving neural networks by preventing co-adaptation of feature detectors.*

что вы можете попробовать выжать из своей нейронной сети независимо от задачи:

- Учитывайте инерцию — важный метод оптимизации — при определении правила обновления веса.
- Скорость обучения лучше снижать постепенно, используя линейное или экспоненциальное затухание, или более современный метод, например косинусоидальное затухание. Наиболее эффективные шаблоны скорости работают не только в зависимости от номера эпохи, но также и от величины потерь, то есть скорость обучения снижается только тогда, когда не удается снизить потери. Попробуйте реализовать это!
- Убедитесь, что масштаб инициализации весов является функцией количества нейронов в вашем слое (это делается по умолчанию в большинстве библиотек нейронных сетей).
- Добавьте дропаут, особенно если в вашей сети есть несколько полностью связанных слоев подряд.

Далее мы перейдем к обсуждению архитектур, специально придуманных для конкретных областей, начиная со сверточных нейронных сетей, используемых для распознавания изображений. Вперед!

ГЛАВА 5

Сверточная нейронная сеть

В этой главе мы рассмотрим сверточные нейронные сети (англ.: *convolutional neural networks* — CNN). CNN — это стандартная архитектура нейронной сети, используемая при работе с изображениями в широком спектре задач. Ранее мы работали только с полносвязными нейронными сетями, которые мы реализовали в виде нескольких полносвязных слоев. Начнем с обзора некоторых ключевых элементов этих сетей и объясним, зачем нам для изображений другая архитектура. Затем мы рассмотрим CNN так же, как и другие понятия в этой книге: сначала мы обсудим, как они работают на высоком уровне, затем перейдем к низкому уровню и, наконец, закодируем все это с нуля[1]. К концу этой главы вы получите достаточно полное представление о том, как работают CNN, чтобы уметь использовать их для своих задач, а также для самостоятельного изучения существующих модификаций CNN, таких как ResNets, DenseNets и Octave Convolutions.

Нейронные сети и обучение представлениям

Нейронные сети получают данные о наблюдениях, причем каждое наблюдение представлено некоторым числом n признаков. Мы приводили два разных примера.

Первый: набор данных о ценах на жилье, где каждое наблюдение состояло из 13 признаков — числовых характеристик этого дома. Второй: набор данных MNIST рукописных цифр; поскольку изображения были представлены 784 пикселями (28 пикселей в ширину и 28 пикселей в высоту),

[1] Наш код для сверточных сетей будет крайне неэффективным. В разделе «Градиент потерь с учетом смещения» приложения А будет предложена более эффективная реализация операции многоканальной свертки, которую мы опишем в этой главе с использованием библиотеки NumPy.

каждое наблюдение было представлено 784 признаками, содержащими яркость или темноту каждого пикселя.

В каждом случае, масштабируя данные, мы смогли построить модель, которая достаточно точно предсказывала результат для этого набора данных. Во всех случаях модель нейронной сети с одним скрытым слоем работала лучше, чем модель без этого скрытого слоя. Почему? Одна из причин заключается в том, что нейронная сеть может изучать нелинейные отношения между входом и выходом. Также в машинном обучении для эффективного предсказания нам часто нужны линейные комбинации наших исходных функций. Предположим, что значения пикселей для цифры из набора MNIST составляют от x_1 до x_{784}. Например, когда x_1 имеет яркость выше среднего, x_{139} ниже среднего и x_{237} тоже ниже среднего, на изображении, скорее всего, нарисована цифра 9. Есть много таких комбинаций, каждая из которых положительно или отрицательно влияет на вероятность того, что нарисована та или иная цифра. Нейронные сети могут автоматически обнаруживать комбинации этих «важных» признаков в процессе обучения. Изначально этот процесс начинается с создания случайных комбинаций исходных элементов путем умножения на случайную весовую матрицу. В процессе обучения нейронная сеть учится находить полезные комбинации и игнорировать бесполезные. Процесс этого отбора важных комбинаций называется обучением представлениям, и это главная причина успеха нейронных сетей в разных областях. Результат показан на рис. 5.1.

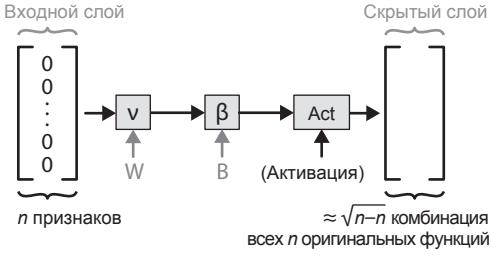


Рис. 5.1. Нейронные сети, которые мы видели до сих пор, получают *n* признаков, а затем отбирают из них полезные комбинации

Надо ли вводить что-то новое для работы с изображениями? Ответ «да» основан на том, что на изображениях интересные «комбинации призна-

ков» (пикселей) обычно берутся из пикселей, расположенных близко друг к другу. Маловероятно, что полезная комбинация получится из 9 случайно выбранных пикселей, в отличие от участка пикселей 3 × 3. Это важный факт: порядок элементов имеет значение, поскольку это соседние пиксели на изображении, тогда как в данных о ценах на жилье порядок элементов не имеет значения. Как это реализовать?

Архитектура для данных изображений

Мы так же будем генерировать комбинации признаков, и каждая из них должна представлять собой комбинацию пикселей из небольшого прямоугольного пятна во входном изображении (рис. 5.2).

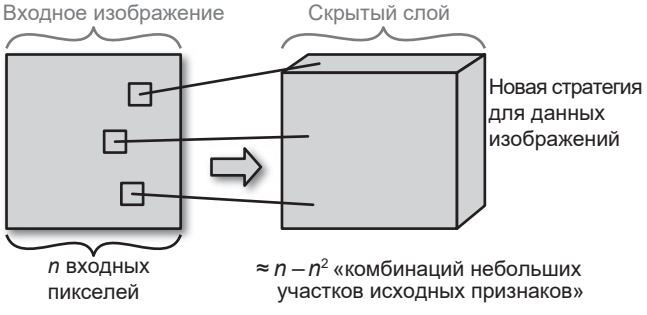


Рис. 5.2. С помощью данных изображения мы можем определить каждый признак как функцию небольшого фрагмента данных и, таким образом, определить от n до n^2 выходных нейронов

Если нейронная сеть изучает комбинации всех входных пикселей во входном изображении, она получается неэффективной, поскольку в этом случае игнорируется принцип, описанный в предыдущем разделе: большинство интересных комбинации функций на изображениях располагается на небольших участках. Однако раньше было несложно рассчитать новые признаки, которые представляли собой комбинации всех входных признаков: если бы у нас было f входных признаков и мы хотим вычислить n новых признаков, нужно просто умножить `ndarray`, содержащий наши входные объекты, на матрицу $f \times n$. А как вычислить сразу много комбинаций пикселей из локальных фрагментов входного изображения? Ответ — операция свертки.

Операция свертки

Прежде чем мы опишем операцию свертки, давайте проясним, что подразумевается под «признаками, представляющими собой комбинацию пикселей из некоторого участка изображения». Допустим, у нас есть входное изображение I размером 5×5:

$$I = \begin{bmatrix} i_{11} & i_{11} & i_{13} & i_{14} & i_{15} \\ i_{21} & i_{22} & i_{23} & i_{24} & i_{25} \\ i_{31} & i_{32} & i_{33} & i_{34} & i_{35} \\ i_{41} & i_{42} & i_{43} & i_{44} & i_{45} \\ i_{51} & i_{52} & i_{53} & i_{54} & i_{55} \end{bmatrix}.$$

Предположим, мы хотим вычислить новый признак из участка 3×3 пикселя в середине. Аналогично определению старых признаков в уже рассмотренных нейронных сетях мы определим новый признак, который является функцией этого участка 3×3. Для этого определим множество весов 3×3, W:

$$w = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}.$$

Затем мы просто возьмем скалярное произведение W на соответствующую часть I, чтобы получить на выходе значение признака, который обозначим как o_{33} (o означает «выход», а цифры означают координаты центра):

$$o_{33} = w_{11} \times i_{22} + w_{12} \times i_{23} + w_{13} \times i_{24} + w_{21} \times i_{32} + w_{22} \times i_{33} + w_{23} \times i_{34} + w_{31} \times i_{42} + w_{32} \times i_{43} + \\ + w_{33} \times i_{44}.$$

Это значение будет обрабатываться так же, как и рассмотренные ранее признаки: к нему может быть добавлено смещение, а затем значение проходит через функцию активации, в результате получится «нейрон», или «выученный признак», который будет передаваться на последующие уровни сети. То есть наши признаки — это функция некоторого участка изображения.

Как интерпретировать такие признаки? Оказывается, что рассчитанные таким образом признаки позволяют понять, присутствует ли в данном

месте изображения визуальный шаблон, определяемый весами. Из теории компьютерного зрения давно известно, что массивы чисел 3×3 или 5×5 могут играть роль «детекторов шаблонов», если взять их скалярное произведение со значениями пикселей в каждой зоне изображения. Например, скалярное произведение следующего массива чисел 3×3:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

с заданным участком входного изображения позволит определить, есть ли в этом месте изображения край. Известны и другие подобные матрицы, позволяющие определять наличие углов, вертикальных или горизонтальных линий и т. д.[1]

Теперь предположим, что мы использовали один и тот же набор весов W, чтобы найти в каждом месте входного изображения некоторый шаблон. Чтобы «примерить» набор W ко всему изображению, надо взять скалярное произведение W на пиксели в каждом месте изображения. В итоге мы получим новое изображение O, размером практически идентичное исходному изображению (размер может немного отличаться в зависимости от того, как делать расчет по краям). Это изображение O будет своего рода «картой признаков», на которой показано расположение шаблона W во входном изображении. Именно этот процесс происходит в сверточных нейронных сетях и называется *сверткой*, а результатом свертки является *карта признаков*.

Изложенный принцип лежит в основе работы CNN. Прежде чем мы сможем организовать полноценную работу, как в предыдущих главах, нужно будет (буквально) добавить новое измерение.

Операция многоканальной свертки

Сверточные нейронные сети отличаются от обычных нейронных сетей тем, что генерируют на порядок больше признаков, а каждый признак является функцией небольшого фрагмента входного изображения. Теперь конкретнее: при наличии n входных пикселей только что описанная

[1] ru.qwe.wiki/wiki/Kernel_(image_processing)

операция свертки создаст n выходных объектов, по одному для каждого местоположения шаблона во входном изображении. Но на самом деле в сверточном слое происходит нечто посложнее: мы создаем f наборов из n признаков, каждый с соответствующим (изначально случайным) набором весов, определяющим визуальный шаблон. Найденный на изображении шаблон будет зафиксирован на карте объектов. Эти карты создаются с помощью операций свертки (рис. 5.3).

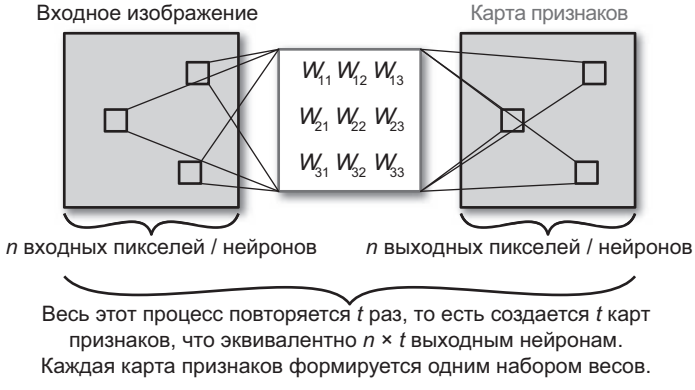


Рис. 5.3. Для входного изображения с n пикселями мы определяем f карт объектов, каждая из которых имеет примерно такой же размер, что и исходное изображение. Всего получается $n \times f$ выходных нейронов, каждый из которых является функцией небольшого участка исходного изображения

С принципами разобрались, но давайте добавим ясности. Каждый «набор признаков», построенный определенным набором весов, называется картой объектов. В контексте сверточного слоя число карт объектов называется количеством каналов слоя, поэтому эта операция называется многоканальной сверткой. Кроме того, f наборов весов W_i называются сверточными фильтрами[1].

Слои свертки

Теперь, когда мы изучили операцию многоканальной свертки, пора подумать о том, как вставить эту операцию в слой нейронной сети. Ранее

[1] А также ядрами.

слои наших нейронных сетей были относительно просты: мы подавали им на вход двумерные объекты `ndarray` и получали такие же на выходе. Но в сверточных сетях у нас на выходе будет 3D-матрица, размерность которой будет число каналов (как у «карт объектов») × высота изображения × ширина изображения.

Возникает вопрос: как передать `ndarray` вперед в другой сверточный слой, чтобы создать «глубокую сверточную» нейронную сеть? Мы знаем, как выполнить операцию свертки изображения с одним каналом и нашими фильтрами. А как выполнить многоканальную свертку с несколькими каналами на входе, когда два сверточных слоя будут соединены вместе? Понимание этого является ключом к пониманию глубоких сверточных нейронных сетей.

Рассмотрим, что происходит в нейронной сети с полносвязными слоями: в первом скрытом слое у нас, например, h_1 признаков, которые являются комбинациями всех исходных признаков из входного слоя. В следующем слое объекты представляют собой комбинации всех объектов предыдущего уровня, так что у нас может быть h_2 «признаков от признаков». Чтобы создать следующий слой из h_2 признаков, мы используем веса $h_1 \times h_2$, чтобы представить, что каждый из объектов h_2 является функцией каждого из объектов h_1 в предыдущем слое.

Аналогичный процесс происходит в первом слое сверточной нейронной сети: сначала мы преобразуем входное изображение в m_1 карт признаков с помощью m_1 сверточных фильтров. Карта показывает, присутствует ли каждый из m_1 шаблонов, описанных весами, в каждом месте входного изображения. Поскольку разные слои полносвязной нейронной сети могут содержать разное количество нейронов, следующий слой сверточной нейронной сети может содержать m_2 признаков. Чтобы сеть могла понимать шаблоны, она ищет в определенном месте изображения «шаблоны шаблонов» — сочетания визуальных шаблонов из предыдущего уровня. Если выходной сигнал сверточного слоя представляет собой трехмерную матрицу формы m_2 каналов × высоту изображения × ширину изображения, то каждое конкретное местоположение на изображении на одной из карт характеристик m_2 представляет собой линейную комбинацию свертки m_1 различных фильтров для этого же положения в каждой из соответствующих карт объектов m_1 из предыдущего уровня. Каждое местоположение в каждой из m_2 карт представляет собой комбинацию визуальных характеристик m_1, уже изученных в предыдущем сверточном слое.

Реализация

Когда мы поняли, как соединяются два многоканальных сверточных слоя, мы сможем реализовать операцию: аналогично тому, как требуется $h_1 \times h_2$ весов для связи слоев с h_1 и h_2 нейронами, нам нужно $m_1 \times m_2$ сверточных фильтров для соединения сверточного слоя с m_1 каналами со слоем с m_2 каналами. Зная это, можно задать размеры `ndarrays`, которые будут составлять вход, выход и параметры операции многоканальной свертки:

1. Форма входных данных:
   - Размер пакета.
   - Входные каналы.
   - Высота изображения.
   - Ширина изображения.
2. Форма выходных данных:
   - Размер пакета.
   - Число выходных каналов.
   - Высота изображения.
   - Ширина изображения.
3. Сами сверточные фильтры будут иметь форму:
   - Входные каналы.
   - Выходные каналы.
   - Высота фильтра.
   - Ширина фильтра.

Порядок измерений в разных библиотеках может отличаться, но сами измерения одни и те же.

Во время реализации операции свертки нам все это понадобится.

Различия между сверточными и полносвязными слоями

В начале главы мы обсудили различия между сверточными и полносвязными слоями на высоком уровне. На рис. 5.4 мы снова привели это сравнение.

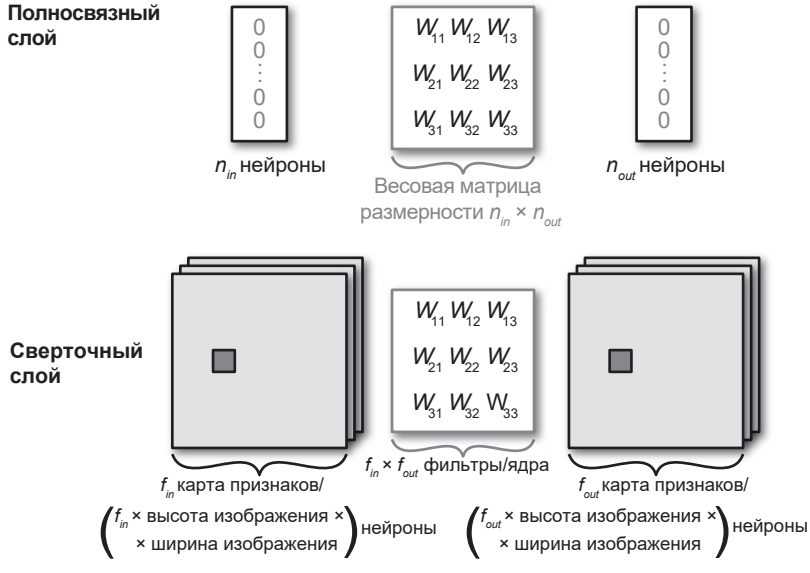


Рис. 5.4. Сравнение сверточных и полносвязных слоев

Еще одно отличие между этими типами слоев заключается в механизме интерпретации отдельных нейронов:

- В полносвязном слое каждый нейрон обнаруживает, присутствует ли *конкретная комбинация признаков, изученных предыдущим слоем*, в текущем наблюдении.
- В сверточном слое каждый нейрон обнаруживает, присутствует ли *конкретная комбинация визуальных шаблонов*, изученных предыдущим слоем, в данном месте входного изображения.

Перед реализацией самой сети надо решить еще один вопрос: как использовать для создания прогнозов пространственные `ndarrays`, которые мы получаем в качестве выходных данных.

Создание прогнозов с помощью сверточных слоев: flatten-слой

Мы рассмотрели, как сверточные слои изучают признаки, показывающие наличие шаблона на изображении, а затем сохраняют эти объекты на карты объектов. Как использовать слои карт для составления прогнозов? При использовании полносвязных нейронных сетей для определения того, к какому из 10 классов принадлежит изображение (см. предыдущую главу), нужно было убедиться, что последний слой имеет размерность 10; затем нужно передать эти 10 чисел в функцию перекрестно-энтропийной потери `softmax`, чтобы превратить прогноз в вероятности. Теперь нужно выяснить, что мы можем сделать в случае сверточного слоя, где у нас есть трехмерный `ndarray` для наблюдения формы m каналов × высоту изображения × ширину изображения.

Напомним, что каждый нейрон показывает, присутствует ли конкретная комбинация визуальных шаблонов (или шаблонов шаблонов) в данном месте на изображении. Это ничем не отличается от признаков, которые мы бы получили с помощью полносвязной нейронной сети: первый полносвязный слой будет представлять признаки отдельных пикселей, второй — признаки этих признаков и т. д. В полносвязном варианте мы бы обрабатывали каждый «признак признаков», который сеть выучила как один нейрон и который будет использоваться в качестве входных данных для прогнозирования того, к какому классу принадлежит изображение.

Оказывается, то же самое можно сделать и со сверточными нейронными сетями — мы рассматриваем m карт объектов как m × *высоту изображения* × *ширину изображения* нейронов и используем операцию flatten, чтобы сжать эти три измерения (количество каналов, высоту изображения и ширину изображения) в одномерный вектор, после чего можем использовать простое матричное умножение для составления окончательного прогноза. Работает это так: каждый отдельный нейрон тут делает примерно то же, что и нейроны в полносвязном слое, то есть говорит, присутствует ли данный визуальный признак (или комбинация признаков) в заданном месте в изображения. Такой нейрон можно считать последним слоем нейронной сети[1].

[1] Вот почему важно понимать, что свертка создает m карт фильтров и m × высота × × ширина изображения отдельных нейронов. Как и в случае с нейронными сетями, ключевым моментом является одновременное хранение нескольких уровней интерпретации в уме и видение связи между ними.

Позже в этой главе мы реализуем flatten-слой. Но прежде чем углубиться в детали реализации, давайте поговорим о другом виде слоя, таком же важном для сверточных сетей.

Пулинговый слой

Пулинговые слои тоже часто используются в сверточных нейронных сетях. Они снижают разрешение карты признаков, созданной операцией свертки. Чаще всего используется размер пула, равный 2, то есть каждая область 2×2 каждой карты признаков приводится либо к максимальному, либо к среднему значению для этой области. Тогда для изображения $n \times n$ все изображение сведется к размеру $\frac{n}{2} \times \frac{n}{2}$. Это показано на рис. 5.5.

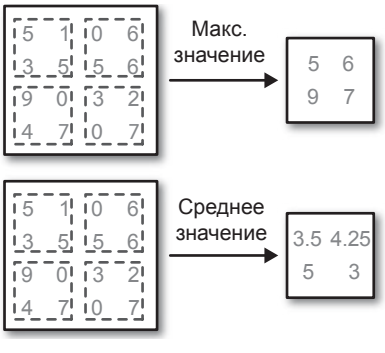


Рис. 5.5. Иллюстрация пулинга по максимальному и среднему для изображения размером 4×4. Каждый участок 2×2 приводится к среднему или максимальному значению для этого участка

Основным преимуществом пулинга является понижение дискретизации изображения, чтобы оно содержало в четыре раза меньше пикселей, чем на предыдущем уровне, то есть объем вычислений, необходимых для обучения сети, снижается в 4 раза. Более того, в сети можно использовать несколько уровней пулинга, как делалось во многих архитектурах на заре CNN. Недостатком пулинга является то, что из изображения с пониженной дискретизацией можно извлечь только одну четвертую часть информации. Однако сети с пулингом хорошо себя показали в тестах по распознаванию изображений, и это наводит на мысль, что потеря информации была не столь значительна по сравнению с увеличением скорости вычислений. Многие решили, что это трюк, который не стоит использовать, даже если он сработал. Как писал Джеффри Хинтон на Reddit AMA в 2014 году:

«Операция пулинга, используемая в сверточных нейронных сетях, является большой ошибкой, и сам факт ее работоспособности является катастрофой». И действительно, в самых последних архитектурах CNN (например, Residual Networks или ResNets[1]) этот метод используется по минимуму или не используется вообще. В этой книге мы не собираемся реализовывать пулы слоев, и рассмотрели их здесь для полноты обзора.

Применение CNN вне изображений

Все описанное выше довольно стандартно для работы с изображениями с использованием нейронных сетей: изображения обычно представлены в виде набора m_1 каналов пикселей, где $m_1 = 1$ для черно-белых изображений и $m_1 = 3$ для цветных изображений. Затем некоторое число m_2 операций свертки применяется к каждому каналу (с помощью карт $m_1 \times m_2$), причем этот шаблон действует для нескольких слоев. Все это используется весьма часто. Реже используется идея разделения данных на «каналы» и последующая обработка этих данных с использованием CNN, причем этот метод подходит не только для изображений. Например, именно так была создана AlphaGo DeepMind, которая показала, что нейронные сети могут научиться играть в го. Приведу выдержку из статьи[2]:

Нейронная сеть принимает на вход стек изображений 19 × 19 × 17, содержащий 17 плоскостей двоичных объектов. Восемь характерных плоскостей X_t состоят из двоичных значений, указывающих наличие камней текущего игрока ($X_{t_i} = 1$, если пересечение *i* содержит камень цвета игрока на временном шаге *t*; 0, если пересечение пусто, содержит камень противника или если $t < 0$). Еще восемь конструктивных плоскостей Y_t представляют соответствующие элементы для камней противника. Последняя характерная плоскость, *C*, представляет цвет для воспроизведения и имеет постоянное значение либо 1, если ходят черные, либо 0, если ходят белые. Эти плоскости объединяются вместе для получения входных объектов $s_t = X_t$, Y_t, X_{t-1}, Y_{t-1},…, X_{t-7}, Y_{t-7}, *C*. Необходимы также старые значения признаков X_t и Y_t, поскольку имеет значение не только текущее положение камней, в связи с тем что повторы запрещены.

[1] См. оригинальную статью *ResNet Deep Residual Learning for Image Recognition*, Kaiming He et al.

[2] DeepMind (Дэвид Сильвер и др.), *Mastering the Game of Go Without Human Knowledge*, 2017.

В их игре поле представлялось в виде «изображения» размером 19 × 19 пикселей с 17 каналами! Шестнадцать из этих каналов кодируют то, что произошло на восьми последних ходах, которые сделал каждый игрок. Это было необходимо для того, чтобы запретить повтор предыдущих шагов. Семнадцатый канал представляет собой сетку 19 × 19, состоящую из всех 1 или 0, в зависимости от того, чей сейчас ход[1]. CNN и их операции многоканальной свертки обычно применяются к изображениям, но сама идея представления данных в виде некоторого пространственного измерения с несколькими «каналами» применима и в других задачах.

Чтобы по-настоящему понять операцию многоканальной свертки, мы должны реализовать ее с чистого листа. Давайте приступим!

Реализация операции многоканальной свертки

Оказывается, что реализация этой жутко сложной операции с четырехмерным `ndarray` на входе и четырехмерным `ndarray` параметров будет намного понятнее, если сначала рассмотреть одномерный случай. Справившись с этим, конечный результат можно будет получить простым добавлением циклов `for`. Как обычно, сначала будут схемы, затем математические представления и код на Python.

Прямой проход

Свертка в одном измерении, по сути, аналогична свертке в двух измерениях: мы берем одномерный вход и одномерный сверточный фильтр, а затем создаем выходные данные, перемещая фильтр вдоль входных данных.

Давайте предположим, что вход имеет длину 5:

$$Input : [t_1, t_2, t_3, t_4, t_5].$$

А размер «шаблонов», которые мы хотим обнаружить, имеет длину 3:

$$Filter : [w_1, w_2, w_3].$$

[1] Год спустя DeepMind опубликовали результаты, используя аналогичный принцип с шахматами — только на этот раз, чтобы закодировать более сложный набор правил шахмат, на входе было 119 каналов! См. DeepMind (Дэвид Сильвер и др.), *A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play*.

Схемы и математическое представление

Первый элемент вывода создается путем свертки первого элемента фильтром:

$$Output\ feature\ 1: t_1 w_1 + t_2 w_2 + t_3 w_3.$$

Второй элемент выходных данных создается путем перемещения фильтра на одну единицу вправо и его свертки со следующими значениями:

$$Output\ feature\ 2: t_2 w_1 + t_3 w_2 + t_4 w_3.$$

Для следующего выходного значения нам не хватило места:

$$Output\ feature\ 3: O_3 = t_3 w_1 + t_4 w_2 + t_5 w_3.$$

Весь ввод обработан, и в результате мы получили только три элемента, хотя начали с пяти! Как быть?

Дополнение изображения

Чтобы избежать уменьшения размера выходных данных в результате операции свертки, сделаем трюк, используемый во всех сверточных нейронных сетях: добавим входным данным по краям нули, чтобы выход стал такого же размера, как и вход. Так мы избегаем проблемы, которая у нас возникла.

Как мы уже поняли, для фильтра размера 3 нужна одна единица отступа по краям, чтобы размер входных данных был правильным. Поскольку мы почти всегда используем нечетные размеры фильтров, размер отступа должен быть равен размеру фильтра, деленному на 2 и округленному до ближайшего целого числа.

Теперь входных данные нумеруются не от i_1 до i_5, а от i_0 до i_6, где i_0 и i_6 равны 0. Затем можно вычислить вывод свертки как:

$$o_1 = i_0 \times w_1 + i_1 \times w_2 + i_2 \times w_3$$

и так до:

$$o_5 = i_4 \times w_1 + i_5 \times w_2 + i_6 \times w_3.$$

Теперь размеры входа и выхода совпадают. Как это реализовать?

Код

Все просто. Прежде чем сделать это, давайте подведем итоги:

1. Мы хотим получить выход такого же размера, как и вход.
2. Чтобы сделать это, не «сжимая» выход, нужно добавить данных на вход.
3. Следует написать цикл, который проходит по входным данным и свертывает каждую позицию с помощью фильтра.

Начнем с ввода и фильтра:

```
input_1d = np.array([1,2,3,4,5])
param_1d = np.array([1,1,1])
```

Функция, добавляющая данные с концов входного вектора:

```
def _pad_1d(inp: ndarray,
            num: int) -> ndarray:
    z = np.array([0])
    z = np.repeat(z, num)
    return np.concatenate([z, inp, z])

_pad_1d(input_1d, 1)

array([0., 1., 2., 3., 4., 5., 0.])
```

А что насчет самой свертки? Заметьте, что для каждого элемента в выводе, который мы хотим произвести, есть соответствующий элемент в дополненном вводе, где «запускается» операция свертки. Как только мы выясним, с чего начать, то просто пройдемся по всем элементам фильтра, выполняя умножение для каждого элемента и добавляя результат к итогу.

Как найти этот «соответствующий элемент»? Легко! Первый элемент выхода берется от первого элемента входа с учетом отступа:

```
def conv_1d(inp: ndarray,
            param: ndarray) -> ndarray:

    # назначение размерностей
    assert_dim(inp, 1)
    assert_dim(param, 1)
```

```
    # отступы
    param_len = param.shape[0]
    param_mid = param_len // 2
    input_pad = _pad_1d(inp, param_mid)

    # инициализация вывода
    out = np.zeros(inp.shape)

    # свертка
    for o in range(out.shape[0]):
        for p in range(param_len):
            out[o] += param[p] * input_pad[o+p]

    # проверка, что форма не меняется
    assert_same_shape(inp, out)

    return out

conv_1d_sum(input_1d, param_1d)

array([ 3., 6., 9., 12., 9.])
```

Все просто. Прежде чем перейти к обратному проходу (а тут уже будет сложно), давайте поговорим о важном гиперпараметре: шаге.

Немного о шаге

Ранее отмечалось, что операции пулинга являются одним из способов уменьшения разрешения изображений. Во многих ранних архитектурах сверточных сетей это позволяло значительно сократить объем вычислений без ущерба точности. Однако этот метод потерял популярность из-за своего недостатка: на следующий слой передается лишь одна четвертая часть данных изображения.

Чаще используется другой метод — изменяется шаг операции свертки. Шагом называется количество единиц, на которые фильтр движется по изображению. В примере выше мы использовали шаг 1, то есть каждый фильтр проверялся на каждом элементе входа, поэтому выход получался такого же размера, что и вход. При шаге, равном 2, фильтр «просматривает» элементы изображения через один и размер выхода будет вдвое меньше входа. Использование шага = 2 дает следующее: мы передаем на

следующий слой вдвое меньше данных, но при этом информация о самом изображении не теряется, так как уменьшение числа данных происходит за счет пропуска, а не удаления. Таким образом, использование шага больше 1 является более удачным методом снижения объема данных в современных CNN.

Для целей этой книги будем придерживаться шага 1, а использование других значений я оставлю вам в качестве самостоятельного упражнения. Так будет проще писать обратный проход.

Свертка: обратный проход

С обратным проходом сложнее. Давайте вспомним, что нужно сделать: раньше мы производили вывод операции свертки, используя входные данные и параметры. Теперь требуется вычислить:

- Частную производную потерь по каждому элементу входных данных операции свертки — ранее `inp`.
- Частную производную потерь по каждому элементу фильтра — ранее `param_1d`.

Вспомните, как работают `ParamOperations`, которые затрагивались в главе 4: в методе `backward` им передается выходной градиент, показывающий, насколько каждый элемент выходных данных в конечном итоге влияет на потери. Затем этот выходной градиент используется для вычисления градиентов входных данных и параметров. Таким образом нужно написать функцию, которая принимает `output_grad` той же формы, что и `input`, и создает `input_grad` и `param_grad`.

Как проверить правильность вычисленных градиентов? Вспомним идею из первой главы: известно, что частная производная суммы по любому из ее входов равна 1 (если сумма $s = a + b + c$, то $\frac{\partial s}{\partial a} = \frac{\partial s}{\partial b} = \frac{\partial s}{\partial c} = 1$). Таким образом, можно вычислить значения `input_grad` и `param_grad` с помощью наших функций `_input_grad` и `_param_grad` (которые мы напишем в ближайшее время) и выхода `put_grad`, заполненного единицами. Затем проверим правильность этих градиентов, изменив элементы входных данных на некоторую величину α, и посмотрим, изменится ли полученная сумма в α раз.

Каким должен быть градиент?

По этому принципу рассчитаем, каким *должен быть* элемент вектора градиента:

```
def conv_1d_sum(inp: ndarray,
                param: ndarray) -> ndarray:
    out = conv_1d(inp, param)
    return np.sum(out)

# случайное увеличение пятого элемента на 1
input_1d_2 = np.array([1,2,3,4,6])
param_1d = np.array([1,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d_2, param_1d))

39.0
41.0
```

Итак, градиент пятого элемента входа *должен быть* равен 41 – 39 = 2.

Давайте порассуждаем, как вычислять такой градиент, не просто рассчитав разницу между этими двумя суммами. Здесь все становится интересно.

Вычисление градиента одномерной свертки

Мы видим, что увеличение этого элемента на входе увеличивает выход на 2. Если внимательно посмотреть на результат, можно увидеть, как именно это происходит:

$$
\begin{aligned}
Output: \big[t_0 w_1 + t_1 w_2 + t_2 w_3, &= O_1 \\
t_1 w_1 + t_2 w_2 + t_3 w_3, &= O_2 \\
t_2 w_1 + t_3 w_2 + t_4 w_3, &= O_3 \\
t_3 w_1 + t_4 w_2 + t_5 w_3, &= O_4 \\
t_4 w_1 + t_5 w_2 + t_6 w_3 \big] &= O_5
\end{aligned}
$$

Этот элемент ввода обозначен t_5. Он появляется в выводе в двух местах:

- В o_4 умножается на w_3.
- В o_5 умножается на w_2.

Чтобы понять, как входные данные сопоставляются с суммой выходных данных, обратите внимание, что при наличии элемента o_6 элемент t_5 тоже бы умножался на w_1.

Следовательно, влияние t_5 на потери, которое мы можем обозначить как $\frac{\partial L}{\partial t_5}$, будет равно:

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_4} \times w_3 + \frac{\partial L}{\partial o_5} \times w_2 + \frac{\partial L}{\partial o_6} \times w_1.$$

Конечно, в этом простом примере, когда потеря — это просто сумма, $\frac{\partial L}{\partial o_1} = 1$ для всех элементов выхода (за исключением элементов отступа, для которых это количество равно 0).

Эту сумму очень легко вычислить: это сумма $w_2 + w_3$, которая равна 2, поскольку $w_2 = w_3 = 1$.

Каков общий шаблон?

Теперь давайте рассмотрим общий шаблон для элемента ввода. Тут нужно правильно отслеживать индексы. Поскольку мы превращаем математические выкладки в код, то используем i-й элемент выходного градиента как o_i^{grad} (поскольку в конечном итоге мы будем обращаться к нему через `output_grad [i]`). Тогда:

$$\frac{\partial L}{\partial t_5} = o_4^{grad} \times w_3 + o_5^{grad} \times w_2 + o_6^{grad} \times w_1 .$$

Аналогично:

$$\frac{\partial L}{\partial t_3} = o_2^{grad} \times w_3 + o_3^{grad} \times w_2 + o_4^{grad} \times w_1 ,$$

а также:

$$\frac{\partial L}{\partial t_4} = o_3^{grad} \times w_3 + o_4^{grad} \times w_2 + o_5^{grad} \times w_1 .$$

Здесь есть определенная закономерность, преобразовать которую в код немного сложно, тем более что индексы на выходе увеличиваются,

а индексы на весах уменьшаются. Это реализуется через вложенный цикл `for`:

```
# param: в нашем случае ndarray формы (1,3)
# param_len: целое число 3
# inp: в нашем случае ndarray формы (1,5)
# input_grad: всегда ndarray той же формы, что и «inp»
# output_pad: в нашем случае ndarray формы (1,7)
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

Такой код увеличивает индексов весов, в то же время уменьшая вес на выходе.

Пока не совсем неочевидно, но самое сложное здесь — это понять, что к чему относится. Добавление сложности, например размеров пакетов, сверток с двумерными входами или входов с несколькими каналами, — это просто вопрос добавления дополнительных циклов `for`, как мы увидим в следующих нескольких разделах.

Вычисление градиента параметра

Аналогично можно рассуждать о том, как увеличение элемента фильтра должно увеличить производительность. Во-первых, давайте увеличим (произвольно) первый элемент фильтра на одну единицу и посмотрим, как это повлияет на сумму:

```
input_1d = np.array([1,2,3,4,5])
# увеличиваем случайно выбранный элемент на 1
param_1d_2 = np.array([2,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d, param_1d_2))
```

```
39.0
49.0
```

Получится $\frac{\partial L}{\partial w_1} = 10$.

Мы внимательно изучали выходные данные и увидели, какие элементы фильтра влияют на них, а также добавили отступы, чтобы шаблон был виднее. Получилось, что:

$$w_1^{grad} = t_0 \times o_1^{grad} + t_1 \times o_2^{grad} + t_2 \times o_3^{grad} + t_3 \times o_4^{grad} + t_4 \times o_5^{grad}.$$

А поскольку для суммы все элементы o_i^{grad} равны 1, а t_0 равно 0, имеем:

$$w_1^{grad} = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 3 + 4 = 10.$$

Это подтверждает полученный ранее расчет.

Код

Этот код будет проще, чем код для входного градиента, поскольку на этот раз индексы движутся в одном направлении. В том же вложенном цикле код будет таким:

```
# param: в нашем случае ndarray формы (1,3)
# param_grad: ndarray такой же формы, что и param
# inp: в нашем случае ndarray формы (1,5)
# input_pad: ndarray такой же формы (1,7)
# output_grad: в нашем случае ndarray формы (1,5)
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        param_grad[p] += input_pad[o+p] * output_grad[o]
```

Наконец, мы можем объединить эти два вычисления и написать функцию для вычисления входного градиента и градиента фильтра с помощью следующих шагов:

1. Взять вход и фильтр в качестве аргументов.
2. Вычислить вывод.
3. Сделать отступ на входе и выходном градиенте (для ввода `input_pad` и `output_pad`).
4. Как показано ранее, использовать выходной градиент с отступом и фильтр для вычисления градиента.
5. Точно так же использовать выходной градиент (без отступа) и дополненный вход для вычисления градиента фильтра.

Целиком функция, которая оборачивает показанный выше код, приведена в https://oreil.ly/2H99xkJ.

На этом реализация свертки в 1D закончена! Как мы увидим в следующих нескольких разделах, переход к двумерным входам, пакетам двумерных входов или даже многоканальным пакетам двумерных входов будет на удивление прост.

Пакеты, 2D-свертки и многоканальность

Во-первых, дадим функциям свертки возможность работать с пакетами входов — двумерными входами, первое измерение которых представляет собой размер пакета ввода, а второе — длину последовательности 1D:

```
input_1d_batch = np.array ([[0,1,2,3,4,5,6],
                            [1,2,3,4,5,6,7]])
```

Выполним те же шаги, которые были определены ранее: сначала добавим входные данные, используя их для вычисления выходных данных, а затем добавим выходной градиент для вычисления градиентов и входа, и фильтров.

Одномерная свертка с пакетами: прямой проход

Единственное отличие от добавления второго измерения (размера пакета) состоит в том, что нужно дополнить и вычислить выходные данные для каждого наблюдения отдельно (как мы делали ранее), а затем сложить результаты, чтобы получить пакет выходных данных. Тогда функция `conv_1d`, например, станет такой:

```
def conv_1d_batch(inp: ndarray,
                  param: ndarray) -> ndarray:

    outs = [conv_1d(obs, param) for obs in inp]
    return np.stack(outs)
```

Одномерная свертка с пакетами: обратный проход

Обратный проход аналогичен: для вычисления входного градиента теперь используется цикл `for` для вычисления входного градиента из

предыдущего раздела, но уже для каждого наблюдения. Результаты суммируются:

```
# "_input_grad" — это функция, содержащая цикл for из предыдущей версии:
# она принимает одномерный вход и фильтр, а также output_gradient,
# и вычисляет входной градиент
grads = [_input_grad(inp[i], param, out_grad[i])[1] for i in
        range(batch_size)]
np.stack(grads)
```

Градиент для фильтра при работе с серией наблюдений будет немного другим. Это связано с тем, что фильтр свертывается с каждым наблюдением и поэтому связан с каждым наблюдением на выходе. Таким образом, чтобы вычислить градиент параметра, мы должны пройтись по всем наблюдениям и увеличить соответствующие значения градиента параметра. Тем не менее для этого достаточно добавить внешний цикл `for` для вычисления градиента параметра, который мы видели ранее:

```
# param: в нашем случае ndarray формы (1,3)
# param_grad: ndarray такой же формы, что и param
# inp: в нашем случае ndarray формы (1,5)
# input_pad: ndarray формы (1,7)
# output_grad: в нашем случае ndarray формы (1,5)
for i in range(inp.shape[0]): # inp.shape[0] = 2
    for o in range(inp.shape[1]): # inp.shape[0] = 5
        for p in range(param.shape[0]): # param.shape[0] = 3
            param_grad[p] += input_pad[i][o+p] * output_grad[i][o]
```

Это измерение «накладывается» поверх оригинальной одномерной свертки. Превращение свертки в двумерную тоже выполняется просто.

Двумерная свертка

2D-свертка — это просто расширение одномерного случая, так как соединение входа и выхода через фильтры в каждой отдельной размерности будет таким, как в одномерном случае. Действия будут такими же:

1. На прямом проходе мы:
   - дополняем входные данные;
   - используем новый вход с отступами и параметры для вычисления результата.

2. На обратном проходе для вычисления входного градиента мы:
   - соответствующим образом заполняем выходной градиент;
   - используем этот выходной градиент с дополнением, а также входные данные и параметры, чтобы вычислить как входной градиент, так и градиент параметров.
3. На обратном проходе для вычисления градиента параметра мы:
   - правильно дополняем ввод;
   - проходим по элементам дополненного ввода и соответствующим образом увеличиваем градиент параметра.

2D-свертка: кодирование прямого прохода

В одномерном случае код с учетом входных данных и параметров прямого прохода выглядел следующим образом:

```
# input_pad: вход с добавленными отступами

out = np.zeros_like(inp)

for o in range(out.shape[0]):
    for p in range(param_len):
        out[o] += param[p] * input_pad[o+p]
```

Для двумерных сверток получаем:

```
# input_pad: вход с добавленными отступами

out = np.zeros_like(inp)

for o_w in range(img_size): # проход по высоте изображения
    for o_h in range(img_size): # проход по ширине изображения
        for p_w in range(param_size): # проход по высоте параметра
            for p_h in range(param_size): # проход по ширине параметра
                out[o_w][o_h] += param[p_w][p_h] * input_pad[o_
                w+p_w][o_h+p_h]
```

Каждый цикл разбивался на два вложенных цикла.

Расширение до двух измерений при наличии пакета изображений тоже похоже на одномерный случай: мы просто добавляем цикл `for` снаружи циклов, показанных здесь.

2D-свертка: написание кода для обратного прохода

Конечно же, как и в прямом проходе, мы можем использовать для обратного прохода те же индексы, что и в одномерном случае. Напомню, что в одномерном случае код имел вид:

```
input_grad = np.zeros_like(inp)

for o in range(inp.shape[0]):
    for p in range(param_len):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

В 2D-случае код выглядит так:

```
# output_pad: вывод с отступами
input_grad = np.zeros_like(inp)

for i_w in range(img_width):
    for i_h in range(img_height):
        for p_w in range(param_size):
            for p_h in range(param_size):
                input_grad[i_w][i_h] +=
               output_pad[i_w+param_size-p_w-1][i_h+param_size-p_h-1] \
                * param[p_w][p_h]
```

Обратите внимание, что индексирование на выходе будет таким же, как в одномерном случае, однако происходит в двух измерениях; в одномерном случае было:

```
output_pad[i+param_size-p-1] * param[p]
```

в 2D-случае есть:

```
output_pad[i_w+param_size-p_w-1][i_h+param_size-p_h-1] * param[p_w][p_h]
```

Другие факты одномерного случая тоже остались актуальны:

- для пакета входных изображений выполняется предыдущая операция для каждого наблюдения, а затем результаты суммируются;

- для градиента параметра производится проход через все изображения в пакете и добавляются компоненты от каждого к соответствующим местам в градиенте параметра[1]:

```
# input_pad: входные данные с отступами

param_grad = np.zeros_like(param)

for i in range(batch_size): # equal to inp.shape[0]
    for o_w in range(img_size):
        for o_h in range(img_size):
            for p_w in range(param_size):
                for p_h in range(param_size):
                    param_grad[p_w][p_h] += input_pad[i][o_w+p_w]
                                            [o_h+p_h] \
                    * output_grad[i][o_w][o_h]
```

Код для полноценной многоканальной свертки почти готов. В данный момент код сворачивает фильтры по двумерному вводу и производит двумерный вывод. Как уже говорилось, на каждом сверточном слое есть не только нейроны, расположенные вдоль этих двух измерений, но и некоторое количество «каналов», равное количеству карт признаков, которые создает слой. Рассмотрим этот момент подробнее.

Последний элемент: добавление «каналов»

Как учесть случаи, когда и ввод, и вывод являются многоканальными? Как и в случае добавления пакетов, ответ прост: добавляются два внешних цикла `for` в уже знакомый код — один цикл для входных каналов и другой для выходных каналов. Зацикливая все комбинации входного и выходного каналов, мы делаем каждую карту выходных объектов комбинацией всех карт входных объектов.

Чтобы это работало, следует *всегда* представлять изображения как трехмерные `ndarrays`. Черно-белые изображения будут иметь один канал, а цветные изображения — три (красный, синий и зеленый). Затем, независимо от количества каналов, работа продолжается, как описано ранее, с использованием ряда карт признаков, созданных из изображения, каж-

[1] Полную реализацию см. на веб-сайте книги (https://oreil.ly/2H99xkJ).

дый из которых представляет собой комбинацию сверток, полученных от всех каналов в изображении (или из каналов на предыдущем слое, если речь идет о слоях в сети).

Прямой проход

А теперь напишем код для вычисления вывода сверточного слоя с учетом четырехмерных `ndarrays` на входе и параметров:

```
def _compute_output_obs(obs: ndarray,
                        param: ndarray) -> ndarray:
    '''
    obs: [channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    assert_dim(obs, 3)
    assert_dim(param, 4)

    param_size = param.shape[2]
    param_mid = param_size // 2
    obs_pad = _pad_2d_channel(obs, param_mid)

    in_channels = fil.shape[0]
    out_channels = fil.shape[1]
    img_size = obs.shape[1]

    out = np.zeros((out_channels,) + obs.shape[1:])
    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for o_w in range(img_size):
                for o_h in range(img_size):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            out[c_out][o_w][o_h] += \
                            param[c_in][c_out][p_w][p_h]
                            * obs_pad[c_in][o_w+p_w][o_h+p_h]
    return out

def _output(inp: ndarray,
            param: ndarray) -> ndarray:
    '''
```

```
    obs: [batch_size, channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    outs = [_compute_output_obs(obs, param) for obs in inp]

    return np.stack(outs)
```

Обратите внимание, что функция `_pad_2d_channel` добавляет отступы.

Код вычислений аналогичен коду в более простом 2D-случае (без каналов), показанном ранее, за исключением того, что теперь имеется `fil[c_out][c_in][p_w][p_h]` вместо обычного `fil[p_w][p_h]`, поскольку в массиве фильтров есть еще два измерения и лишние `c_out` × `c_in` элементов.

Обратный проход

Обратный проход аналогичен и выполняется так же, как и в простом 2D-случае:

1) для входных градиентов вычисляются градиенты каждого наблюдения по отдельности (для этого добавляем выходной градиент), а затем градиенты складываются;
2) используется выходной градиент с отступами для градиента параметра, выполняется проход в цикле по наблюдениям и применяются соответствующие значения для каждого из них, чтобы обновить градиент параметра.

Ниже приведен код вычисления выходного градиента:

```
def _compute_grads_obs(input_obs: ndarray,
                       output_grad_obs: ndarray,
                       param: ndarray) -> ndarray:
    '''
    input_obs: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    input_grad = np.zeros_like(input_obs)
    param_size = param.shape[2]
    param_mid = param_size // 2
    img_size = input_obs.shape[1]
```

```
    in_channels = input_obs.shape[0]
    out_channels = param.shape[1]
    output_obs_pad = _pad_2d_channel(output_grad_obs, param_mid)

    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for i_w in range(input_obs.shape[1]):
                for i_h in range(input_obs.shape[2]):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            input_grad[c_in][i_w][i_h] += \
                            output_obs_pad[c_out][i_w+param_size-p_w-1]
                            [i_h+param_size-p_h-1] \
                            * param[c_in][c_out][p_w][p_h]

    return input_grad

def _input_grad(inp: ndarray, output_grad: ndarray, param: ndarray)
-> ndarray:

    grads = [_compute_grads_obs(inp[i], output_grad[i], param)
          for i in  range(
        output_grad.shape[0])]

    return np.stack(grads)
```

А вот и градиент параметра:

```
def _param_grad(inp: ndarray, output_grad: ndarray, param: ndarray)
-> ndarray:
    '''
    inp: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    param_grad = np.zeros_like(param)
    param_size = param.shape[2]
    param_mid = param_size // 2
    img_size = inp.shape[2]
    in_channels = inp.shape[1]
    out_channels = output_grad.shape[1]
```

```
    inp_pad = _pad_conv_input(inp, param_mid)
    img_shape = output_grad.shape[2:]

    for i in range(inp.shape[0]):
        for c_in in range(in_channels):
            for c_out in range(out_channels):
                for o_w in range(img_shape[0]):
                    for o_h in range(img_shape[1]):
                        for p_w in range(param_size):
                            for p_h in range(param_size):
                                param_grad[c_in][c_out][p_w][p_h] += \
                                inp_pad[i][c_in][o_w+p_w][o_h+p_h] \
                                * output_grad[i][c_out][o_w][o_h]
    return param_grad
```

Эти три функции — `_output`, `_input_grad` и `_param_grad` — именно то, что нужно для создания класса `Conv2DOperation`, формирующего ядро `Conv2DLayers`, которое мы будем использовать в наших CNN! Осталось проработать всего пару деталей, а потом можно применять эту операцию в сверточной сети.

Использование операции для обучения CNN

Чтобы получить работающую модель CNN, нужно реализовать еще немного:

1) реализовать операцию flatten, рассмотренную ранее в этой главе, чтобы модель могла делать прогнозы;
2) включить этот класс `Operation`, а также `Conv2DOpOperation` в слой `Conv2D`;
3) написать более быструю версию `Conv2DOperation`. Напишем ее здесь, а подробности рассмотрим в разделе «Цепное правило» приложения А.

Операция flatten

Для завершения сверточного слоя понадобится еще одна операция: операция flatten. Результатом операции свертки будет трехмерный массив для каждого наблюдения измерения (число каналов, `img_height`, `img_width`). Но если мы не передадим эти данные в другой сверточный слой, сначала

нужно будет преобразовать их в вектор для каждого наблюдения. Поскольку каждый нейрон кодирует присутствие шаблона в данном месте на изображении, мы легко можем «сплющить» этот трехмерный `ndarray` в одномерный вектор и передать его вперед. Операция flatten, показанная здесь, делает именно это, учитывая тот факт, что в сверточных слоях, как и в любом другом слое, первым измерением нашего `ndarray` всегда является размер пакета:

```
class Flatten(Operation):
    def __init__(self):
    super().__init__()

    def _output(self) -> ndarray:
    return self.input.reshape(self.input.shape[0], -1)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
    return output_grad.reshape(self.input.shape)
```

Это последняя требуемая операция. Теперь обернем наши `Operation` в `Layer`.

Готовый слой Conv2D

Таким образом, весь сверточный слой будет выглядеть примерно так:

```
class Conv2D(Layer):

    def __init__(self,
                 out_channels: int,
                 param_size: int,
                 activation: Operation = Sigmoid(),
                 flatten: bool = False) -> None:
        super().__init__()
        self.out_channels = out_channels
        self.param_size = param_size
        self.activation = activation
        self.flatten = flatten

    def _setup_layer(self, input_: ndarray) -> ndarray:

        self.params = []
```

```
        conv_param = np.random.randn(self.out_channels,
                                     input_.shape[1], # входные каналы
                                     self.param_size,
                                     self.param_size)
        self.params.append(conv_param)

        self.operations = []
        self.operations.append(Conv2D(conv_param))
        self.operations.append(self.activation)

        if self.flatten:
            self.operations.append(Flatten())

        return None
```

В зависимости от того, хотим ли мы передавать выходные данные этого слоя в другой сверточный слой или в полносвязный связанный слой для предсказаний, применятся (или нет) операция flatten.

Пара слов о скорости и альтернативной реализации

Читатели, знакомые с понятием вычислительной сложности, могут сказать, что такой код катастрофически медленный: для вычисления градиента параметра пришлось написать семь вложенных циклов! В этом нет ничего плохого, поскольку нам нужно было прочувствовать и понять принцип работы CNN, написав все с нуля. Но можно написать по-другому:

1) из входных данных извлекаются участки `image_height` × `image_width` × `num_channels` размера `filter_height` × `filter_width` из набора тестов;
2) для каждого участка выполняется скалярное произведение на соответствующий фильтр, соединяющий входные каналы с выходными каналами;
3) складываем результаты скалярных произведений, чтобы сформировать результат.

Проявив смекалку, можно выразить почти все описанные ранее операции через пакетное умножение матриц, реализованное с помощью функции NumPy `mul`. В приложении А и на веб-сайте книги показано, как это сделать, а пока достаточно сказать, что такая реализация позволяет писать

относительно небольшие сверточные нейронные сети, которые будут обучаться за разумное количество времени. А это, в свою очередь, открывает простор для экспериментов!

Эксперименты

Даже если мы используем функцию `matmul` и изменение формы, обучение модели на одну эпоху с одним слоем свертки все равно займет около 10 минут, поэтому ограничимся демонстрацией модели только с одним сверточным слоем с 32 каналами (количество выбрано условно):

```
model = NeuralNetwork(
        layers=[Conv2D(out_channels=32,
                       param_size=5,
                       dropout=0.8,
                       weight_init="glorot",
                       flatten=True,
                      activation=Tanh()),
                Dense(neurons=10,
                      activation=Linear())],
        loss = SoftmaxCrossEntropy(),
        seed=20190402)
```

Обратите внимание, что у этой модели $32 \times 5 \times 5 = 800$ параметров в первом слое, но эти параметры используются для создания $32 \times 28 \times 28 = 25\,088$ нейронов, или «изученных признаков». В полносвязном слое со скрытым размером 32 получится $784 \times 32 = 25\,088$ *параметров* и всего 32 нейрона.

Несколько простых проб и ошибок — обучение этой модели на нескольких сотнях пакетов с разными скоростями обучения и наблюдение за полученными в результате потерями — показывают, что скорость обучения 0.01 работает лучше, чем скорость обучения 0.1, когда первый слой сверточный, а не полносвязный. Обучение сети за одну эпоху с оптимизатором `SGDMomentum (lr = 0.01, momentum = 0.9)` дает:

```
Validation accuracy after 100 batches is 79.65%
Validation accuracy after 200 batches is 86.25%
Validation accuracy after 300 batches is 85.47%
Validation accuracy after 400 batches is 87.27%
Validation accuracy after 500 batches is 88.93%
```

```
Validation accuracy after 600 batches is 88.25%
Validation accuracy after 700 batches is 89.91%
Validation accuracy after 800 batches is 89.59%
Validation accuracy after 900 batches is 89.96%
Validation loss after 1 epochs is 3.453

Model validation accuracy after 1 epoch is 90.50%
```

Из результата видно, что мы можем обучить сверточную нейронную сеть с нуля, что в итоге дает MNIST более 90% точности всего за один проход по обучающему набору[1]!

Заключение

В этой главе мы поговорили о сверточных нейронных сетях. Мы начали с общих понятий о сверточных сетях и о том, в чем они схожи и чем отличаются от полносвязных нейронных сетей, а затем описали их работу на низком уровне, реализовав базовую операцию многоканальной свертки с нуля в Python.

Начиная с высокого уровня сверточные слои создают примерно на порядок больше нейронов, чем полносвязные слои, которые мы видели до этого, причем каждый нейрон представляет собой комбинацию всего лишь нескольких признаков из предыдущего слоя, а не всех элементов предыдущего уровня, как в полносвязных слоях. На уровне ниже мы увидели, что нейроны фактически сгруппированы в «карты признаков», каждая из которых показывает, присутствует ли определенный шаблон или их комбинация в данном месте на изображении. Такие карты признаков называются «каналами» сверточного слоя.

Несмотря на все отличия от классов `Operation`, которые мы использовали в плотных слоях, операция свертки вписывается в тот же шаблон, что и другие операции `ParamOperation`, которые мы видели:

- у него есть метод `_output`, который вычисляет вывод с учетом его входных данных и параметров;
- у него есть методы `_input_grad` и `_param_grad`, которые при заданном `output_grad` той же формы, что и выходные данные операции, вычис-

[1] Полный код можно найти в разделе главы в репозитории книги на GitHub.

ляют градиенты той же формы, что и входные данные и параметры соответственно.

Разница тут лишь в том, что `_input`, `output` и `params` теперь являются четырехмерными объектами `ndarray`, тогда как в случае полностью связанных слоев они были двумерными.

Полученные знания сформируют прочный фундамент для изучения или применения сверточных нейронных сетей в будущем. Далее рассмотрим еще один распространенный вид архитектуры нейронных сетей: рекуррентные нейронные сети, предназначенные для работы с данными, появляющимися в последовательностях, а не просто с несвязанными пакетами, с которыми мы имели дело в случаях с домами и изображениями. Вперед!

ГЛАВА 6

Рекуррентные нейронные сети

В этой главе рассмотрим рекуррентные нейронные сети (*recurrent neural networks*, RNN), класс нейросетей, предназначенных для обработки последовательных данных. В нейронных сетях, с которыми мы работали до этого, каждая полученная партия данных рассматривалась как набор независимых наблюдений. Сеть не знала ничего о предыдущих или будущих данных цифр из MNIST, будь то полносвязная нейронная сеть из главы 4 или сверточная нейронная сеть из главы 5. Однако многие виды данных являются по своей природе упорядоченными. Это могут быть зависящие от времени последовательности финансовых данных или языковые данные, в которых символы, слова и предложения упорядочены, и т. д. Цель рекуррентных нейронных сетей — научиться принимать *последовательности* таких данных и возвращать правильный прогноз, например цену товара на следующий день или следующее слово в предложении.

Для работы с упорядоченными данными с помощью полносвязных нейронных сетей, которые мы рассматривали в первых нескольких главах, потребуются три модификации. Во-первых, потребуется добавить «новое измерение» в данные, которые мы подаем на вход нейронной сети. Ранее данные, которые мы подавали нейронным сетям, были, по существу, двумерными — у каждого `ndarray` было одно измерение, отвечающее за число наблюдений, и еще одно, обозначающее число признаков[1]. Можно представить это иначе — *каждое наблюдение* представляет собой одномерный вектор. В рекуррентных нейронных сетях у данных тоже будет измерение, представляющее количество наблюдений, но каждое наблюдение будет представлено в виде двумерного массива: одно измерение обозначает длину последовательности данных, а второе — сами признаки

[1] Мы обнаружили, что наблюдения удобно располагать по строкам, а элементы — по столбцам, но это не обязательно. В любом случае данные должны быть двумерными.

в каждом элементе последовательности. Таким образом, на вход RNN будет подаваться трехмерный `ndarray` вида `[batch_size, sequence_length, num_features]`, т. е. пакет последовательностей.

Второе: для работы с трехмерными входными данными нужно использовать новый тип архитектуры нейронных сетей, которому как раз и посвящена эта глава. Но именно с третьей модификации мы и начнем обзор. Для работы с такой новой формой данных придется использовать совершенно другую структуру и другие абстракции. Почему? И в полносвязных, и в сверточных нейронных сетях каждая «операция», даже если она фактически представляет собой множество отдельных сложений и умножений (как в случае умножения матриц или свертки), может быть описана как цельная «мини-фабрика», которая и на прямом, и на обратном проходе принимает на вход один `ndarray` и выдает также один `ndarray` на выходе (при это может использоваться еще один `ndarray` с параметрами операции для выполнения вычислений). Как оказалось, рекуррентные нейронные сети так реализовать не получится. Прежде чем мы разберемся, почему, подумайте вот над чем: какие характеристики архитектуры нейронной сети делают невозможным использование созданной нами платформы? Ответ станет для вас озарением, но для описания полного пути его получения потребовалось бы углубиться в детали реализации, которые выходят за рамки этой книги. Давайте рассмотрим ключевое ограничение платформы, которую использовали до сих пор.

Ключевое ограничение: работа с ветвлениями

Оказывается, наша структура не позволяет обучать модели с помощью вычислительных графов, показанных на рис. 6.1.

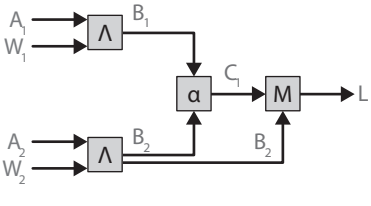


Рис. 6.1. Вычислительный граф, который приводит к сбою в классе Operation: поскольку во время прямого прохода обрабатывается много разных данных, это значит, что мы не можем отправлять «назад» градиенты во время обратного прохода, как раньше

В чем проблема? Преобразование прямого прохода в код выглядит хорошо (обратите внимание: `Add` и `Multiply` приведены здесь исключительно для демонстрационных целей):

```
a1 = torch.randn(3,3)
w1 = torch.randn(3,3)

a2 = torch.randn(3,3)
w2 = torch.randn(3,3)

w3 = torch.randn(3,3)

# операции
wm1 = WeightMultiply(w1)
wm2 = WeightMultiply(w2)
add2 = Add(2, 1)
mult3 = Multiply(2, 1)

b1 = wm1.forward(a1)
b2 = wm2.forward(a2)
c1 = add2.forward((b1, b2))
L = mult3.forward((c1, b2))
```

Проблемы начинаются на обратном проходе. Допустим, мы хотим использовать уже привычное цепное правило для вычисления производной от `L` по `w1`. Раньше мы просто обращались к каждой операции в обратном порядке. Здесь же, из-за повторного использования `b2` *во время прямого прохода*, этот подход не сработает. Например, если бы мы начали с обратного вызова на `mult3`, то у нас были бы градиенты для обоих входов `c1` и `b2`. А если бы затем мы вызвали функцию `backward` на `add2`, то не смогли бы подать на вход градиент для `c1`, потому что нужен еще и градиент для `b2`, так как он тоже влияет на значение потери `L`. Таким образом, для правильного выполнения обратного прохода уже не получится просто перемещаться по операциям в обратном порядке. Вместо этого придется написать что-то вроде следующего:

```
c1_grad, b2_grad_1 = mult3.backward (L_grad)

b1_grad, b2_grad_2 = add2.backward (c1_grad)

# объединение градиентов, показывающее, что b2
```

```
# используется на прямом проходе дважды
b2_grad = b2_grad_1 + b2_grad_2

a2_grad = wm2.backward (b2_grad)

a1_grad = wm1.backward (b1_grad)
```

На этом этапе можно полностью отказаться и от использования класса `Operation`. Вместо этого можно сохранить все величины, которые вычисляются на прямом проходе, и повторно использовать их на обратном, как делалось в главе 2! Всегда можно реализовать сколь угодно сложные нейронные сети, вручную прописывая отдельные вычисления, которые необходимо выполнить на прямом и обратном проходах. В главе 2 мы так уже делали, когда записывали 17 отдельных операций для обратного прохода двухслойной нейронной сети (и что-то подобное мы сделаем в этой главе внутри «ячеек RNN»). Класс `Operation` мы вводили с целью создать гибкую структуру, которая позволит на высоком уровне описать нейронную сеть и сделать так, чтобы все вычисления низкого уровня «просто работали». Такая структура хорошо иллюстрирует многие ключевые понятия нейронных сетей, но сейчас мы увидели ее недостатки.

У этой проблемы есть элегантное решение: автоматическое дифференцирование. Это совершенно иной способ реализации нейронных сетей[1]. Мы рассмотрим эту концепцию в достаточной степени, чтобы понять, как она работает, но слишком углубляться не станем, так как созданию полнофункциональной среды автоматического дифференцирования пришлось бы посвятить несколько глав. Кроме того, в следующей главе, посвященной PyTorch, мы рассмотрим использование высокопроизводительного фреймворка автоматического дифференцирования. Сама концепция автоматического дифференцирования довольно важна, и нам надо поговорить о ней, прежде чем мы перейдем к RNN. Мы разработаем базовую структуру для этого механизма и покажем, как она помогает решить проблему с повторным использованием объектов во время прямого прохода, о которой мы говорили ранее.

---

[1] Стоит упомянуть альтернативное решение этой проблемы, которым автор Даниэль Сабинас поделился в своем блоге: он представляет операции в виде графа, а затем использует поиск по ширине для вычисления градиентов на обратном проходе в правильном порядке. В результате получается структура, похожая на TensorFlow. В его постах в блоге все это рассказывается логично и понятно.

Автоматическое дифференцирование

Как мы уже видели, существуют архитектуры нейронных сетей, в которых наш класс `Operation` не позволяет с легкостью вычислить градиенты выходных данных относительно входных данных, а именно это нам нужно для обучения моделей. Автоматическое дифференцирование позволяет вычислять эти градиенты совершенно иначе: вместо того чтобы составлять сеть из классов `Operation`, мы определяем класс, который оборачивается вокруг самих данных и позволяет данным отслеживать операции, выполняемые над ними. В результате данные смогут постоянно накапливать градиенты по мере прохождения через операции. Чтобы лучше понять, как будет работать это «накопление градиента», напишем код[1].

Реализация накопления градиента

Чтобы автоматически отслеживать градиенты, мы должны переписать методы Python, которые выполняют основные операции над данными. В Python использование операторов `+` или `-` фактически вызывает скрытые методы `__add__` и `__sub__`. Например, вот как это работает с оператором сложения:

```
a = array([3,3])
print("Addition using '__add__':", a.__add__(4))
print("Addition using '+':", a + 4)

Addition using '__add__': [7 7]
Addition using '+': [7 7]
```

Благодаря этому мы можем написать класс, который оборачивается вокруг типичного «числа» Python (`float` или `int`) и перезаписывает методы `add` и `mul`:

```
Numberable = Union[float, int]

def ensure_number(num: Numberable) -> NumberWithGrad:
    if isinstance(num, NumberWithGrad):
```

[1] Для более глубокого понимания того, как внедрить автоматическое дифференцирование, почитайте книгу *Grokking Deep Learning* Эндрю Траска (Manning) (На русском: *Траск Э.* Грокаем глубокое обучение. — СПб.: Питер, 2020. — 352 с.).

```
        return num
    else:
        return NumberWithGrad(num)

class NumberWithGrad(object):

    def __init__(self,
                 num: Numberable,
                 depends_on: List[Numberable] = None,
                 creation_op: str = ''):
        self.num = num
        self.grad = None
        self.depends_on = depends_on or []
        self.creation_op = creation_op

    def __add__(self,
                other: Numberable) -> NumberWithGrad:
        return NumberWithGrad(self.num + ensure_number(other).num,
                              depends_on = [self,
                              ensure_number(other)],
                              creation_op = 'add')

    def __mul__(self,
                other: Numberable = None) -> NumberWithGrad:

        return NumberWithGrad(self.num * ensure_number(other).num,
                              depends_on = [self, _number(other)],
                              creation_op = 'mul')

    def backward(self, backward_grad: Numberable = None) -> None:
        if backward_grad is None: # first time calling backward
            self.grad = 1
        else:
            # В этих строках реализовано накопление градиентов.
            # Если градиент пока не существует, он становится равен
            # backward_grad
            if self.grad is None:
                self.grad = backward_grad
            # В противном случае backward_grad добавляется
            # к существующему градиенту
            else:
                self.grad += backward_grad
```

```
        if self.creation_op == "add":
            # Назад отправляется self.grad, так как увеличение
            # любого из этих элементов приведет к такому же
            # увеличению выходного значения
            self.depends_on[0].backward(self.grad)
            self.depends_on[1].backward(self.grad)

        if self.creation_op == "mul":

            # Расчет производной по первому элементу
            new = self.depends_on[1] * self.grad
            # Отправка производной по этому элементу назад
            self.depends_on[0].backward(new.num)

            # Расчет производной по второму элементу
            new = self.depends_on[0] * self.grad
            # Отправка производной по этому элементу назад
            self.depends_on[1].backward(new.num)
```

В этом коде много чего происходит, поэтому давайте распакуем класс `NumberWithGrad` и посмотрим, как он работает. Напомним, что этот класс позволяет писать простые операции и автоматически рассчитывать градиенты. Например:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 5
```

Как сильно увеличение на ϵ в данном случае увеличит значение `c`? Довольно очевидно, что увеличение получится в 4 раза. И действительно, если мы в этом классе сначала напишем:

```
c.backward (),
```

то затем можно просто написать вот так, не используя циклы `for`:

```
print(a.grad)
```

```
4
```

Как это работает? Фундаментальный секрет описанного выше класса заключается в том, что каждый раз, когда над объектом `NumberWithGrad` выполняется операция `+` или `*`, создается новый объект `NumberWithGrad`,

зависящий от NumberWithGrad. Затем, когда на объекте NumberWithGrad происходит обратный вызов, как ранее для c, все градиенты для всех объектов NumberWithGrad, использованных для создания c, вычисляются автоматически. И действительно, градиент в результате рассчитывается не только для a, но и для b:

```
print(b.grad)
```

```
1
```

Но главное преимущество такой структуры заключается в том, что NumberWithGrads *накапливают* градиенты, что позволяет многократно использовать их во время серии вычислений, гарантированно получая правильный градиент. Мы покажем это на тех же операциях, которые ранее вызывали вопросы, но будем использовать NumberWithGrad, а затем подробно рассмотрим, как все работает.

Иллюстрация автоматического дифференцирования

В данной серии вычислений переменная a используется многократно:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = c * (a + 2)
```

Нетрудно посчитать, что после всего этого d = 75, но вопрос заключается в другом: как сильно увеличится значение a при увеличении значения d? Сначала мы можем найти ответ на этот вопрос математически. У нас есть:

$d = (4a + 3) \times (a + 2) = 4a^2 + 11a + 6.$

Тогда, используя степенное правило:

$$\frac{\partial d}{\partial a} = 8a + 11.$$

Следовательно, для a = 3 значение этой производной должно быть 8 × 3 + + 11 = 35. Проверим это численно:

```
def forward(num: int):
    b = num * 4
    c = b + 3
```

```
    return c * (num + 2)

print(round(forward(3.01) – forward(2.99)) / 0.02), 3)
```

```
35.0
```

Теперь обратите внимание, что мы получим тот же результат, если будем вычислять градиент с помощью системы автоматического дифференцирования:

```
a = NumberWithGrad(3)

b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e.backward()
print(a.grad)
```

```
35
```

Объясним, что случилось

Мы увидели, что цель автоматического дифференцирования состоит в том, чтобы сделать фундаментальными единицами анализа *сами объекты данных* — числа, объекты `ndarray`, `Tensor` и т. д. — а не `Operation`, как было раньше.

У всех методов автоматического дифференцирования есть общие черты:

- У каждого метода есть класс, который оборачивается вокруг фактически вычисляемых данных. Здесь оборачивается `NumberWithGrad` вокруг чисел с плавающей точкой и целых чисел. К примеру, в PyTorch аналогичный класс называется `Tensor`.
- Общие операции, такие как сложение, умножение и умножение матриц, переопределяются так, чтобы они всегда возвращали экземпляр этого класса. В предыдущем случае мы реализовали `NumberWithGrad` и `NumberWithGrad` или `NumberWithGrad` с `float` или `int`.
- Класс `NumberWithGrad` должен содержать информацию о том, как вычислять градиенты, с учетом информации о том, что происходит на прямом проходе. Ранее мы делали это путем включения в класс аргумента `creation_op`, в котором записывалось, как был создан класс `NumberWithGrad`.

- На обратном проходе градиенты передаются в обратном направлении с использованием базового типа данных, а не «обертки». В нашем случае это означает, что градиенты будут иметь тип `float` и `int`, а не `NumberWithGrad`.
- Как мы говорили в начале этого раздела, автоматическое дифференцирование позволяет повторно использовать значения, вычисленные на прямом проходе, — в предыдущем примере мы без проблем дважды использовали значение `a`. Секрет заключается в этих строках:

  ```
  if self.grad is None:
      self.grad = backward_grad
  else:
      self.grad += backward_grad
  ```

  Здесь говорится, что после получения нового градиента, `backward_grad`, объект `NumberWithGrad` должен либо использовать в качестве этого значения градиент `NumberWithGrad`, либо просто добавить его значение к существующему градиенту `NumberWithGrad`. Это позволяет `NumberWithGrad` накапливать градиенты, когда в модели повторно используются соответствующие объекты.

На этом с автоматическим дифференцированием закончим. Давайте теперь обратимся к структуре модели, ради которой затеяли все эти объяснения, поскольку для вычисления прогноза требуется повторное использование некоторых значений во время прямого прохода.

Актуальность рекуррентных нейронных сетей

Как говорилось в начале этой главы, рекуррентные нейронные сети предназначены для обработки данных, представленных в виде последовательностей: каждое наблюдение теперь представляет собой не вектор с `n` объектами, а двумерный массив размерности `n` объектов на `t` временных шагов (рис. 6.2).

В следующих нескольких разделах я расскажу, как в RNN содержатся такие данные, но сначала давайте попытаемся понять, зачем они нужны. Почему обычные нейронные сети с прямой связью не подходят для обработки таких данных? Можно представить каждый временной шаг в виде независимого набора признаков. Например, одно наблюдение может содержать признаки, соответствующие времени `t = 1`, а целевым

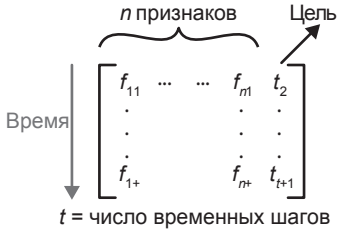


Рис. 6.2. Последовательности данных: на каждом из t временных шагов у нас есть n признаков

станет набор `t = 2`. Затем следующее наблюдение может иметь признаки от времени `t = 2`, а целевым станет набор `t = 3`, и т. д. Если бы мы хотели для составления прогнозов использовать данные *нескольких* временных шагов, а не одного временного шага, то могли бы использовать функции от `t = 1` и `t = 2`, чтобы прогнозировать целевое значение `t = 3`, а от `t = 2` и `t = 3`, чтобы прогнозировать целевое значение при `t = 4` и т. д.

Однако обработка каждого временного шага как независимого набора не учитывает тот факт, что данные расположены в определенной последовательности. Как учесть последовательный характер данных для составления более точных прогнозов? Решение будет примерно таким:

1. Использовать признаки от временного шага `t = 1`, чтобы составить прогнозы для соответствующей цели при `t = 1`.
2. Использовать признаки от временного шага `t = 2`, *а также информацию от* `t = 1`, *включая значение цели при* `t = 1`, чтобы составить прогнозы для `t = 2`.
3. Использовать признаки от `t = 3`, *а также накопленную информацию с* `t = 1` *и* `t = 2`, чтобы составить прогнозы при `t = 3`.
4. Далее на каждом шаге использовать информацию от всех предыдущих шагов, чтобы составить прогноз.

Чтобы сделать это, нам придется передавать наши данные через нейронную сеть по одному элементу последовательности за раз, причем сначала должны использоваться данные с первого временного шага, затем со следующего временного шага и т. д. Кроме того, мы хотим, чтобы по мере обработки новых элементов последовательности наша нейронная сеть «накапливала» информацию о том, что видела раньше. В оставшейся части этой главы будет подробно рассмотрена реализация этого в рекуррентных нейронных сетях. Существует несколько вариантов реализации рекур-

рентных нейронных сетей, но у всех них есть общая базовая структура последовательной обработки данных. Сначала обсудим эту структуру, а затем рассмотрим, чем отличаются варианты.

Введение в рекуррентные нейронные сети

Начнем с того, что на высоком уровне обсудим, как передаются данные через «прямую» нейронную сеть. В таком типе сети данные передаются через несколько *слоев*. Для одного наблюдения результат работы — это «представление» наблюдения на этом слое. После первого слоя это представление состоит из признаков, которые являются комбинациями исходных признаков. После следующего слоя оно состоит из комбинаций этих представлений или «признаков признаков» исходных элементов и т. д. для последующих слоев сети. Таким образом, после каждого прямого прохода сеть будет на выходах каждого из своих слоев содержать множество представлений исходного наблюдения (рис. 6.3).

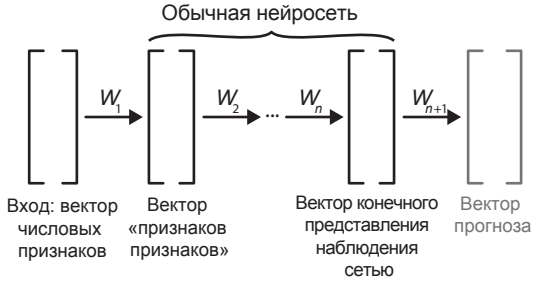


Рис. 6.3. Обычная нейронная сеть, в которой наблюдение передается вперед и преобразуется в разные представления после каждого слоя

Однако когда через сеть пройдет следующий набор наблюдений, эти представления удалятся. Ключевым нововведением рекуррентных нейронных сетей и всех их вариантов является передача этих представлений обратно в сеть вместе со следующим набором наблюдений. Это будет выглядеть вот так:

1. На первом временном шаге `t = 1` мы пропускаем через сеть наблюдение с первого временного шага (возможно, с некоторыми случайно инициализированными представлениями). Получаем прогноз для `t = 1`, а также представления на каждом слое.

2. На следующем временном шаге мы пропускаем наблюдение со второго временного шага, `t = 2` вместе с представлениями, вычисленными на первом временном шаге (которые, опять же, являются просто выходами слоев нейронной сети), и каким-то образом объединяем их (именно по способам объединения и различаются варианты RNN, о которых чуть позже). И та и другая информация используется для вычисления прогноза для `t = 2`, а также обновленных представлений на каждом слое, которые теперь являются функцией входных данных, передаваемых при `t = 1` и при `t = 2`.
3. На третьем временном шаге мы передаем наблюдение от `t = 3`, а также представления, которые теперь включают информацию от `t = 1` и `t = 2`, используем эту информацию, чтобы составить прогнозы для `t = 3`, а также обновляем представления каждого слоя, которые теперь содержат информацию из временных шагов 1–3.

Этот процесс изображен на рис. 6.4.

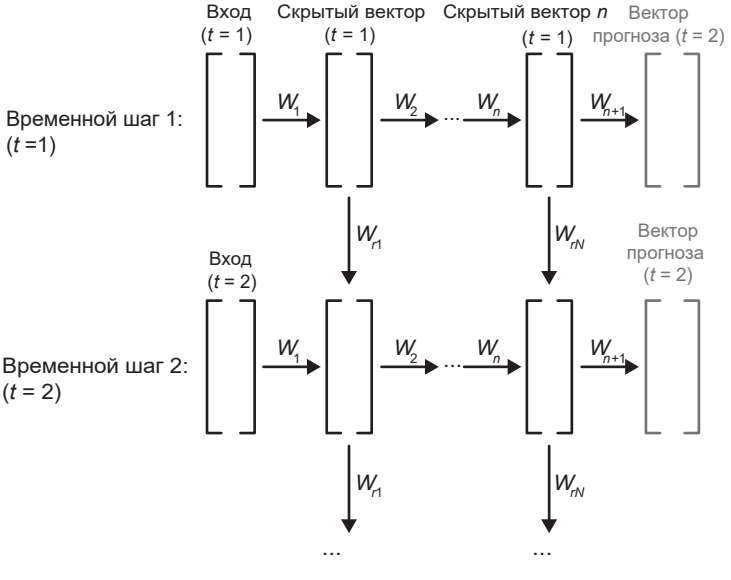


Рис. 6.4. В рекуррентных сетях представления каждого слоя используются на следующих временных шагах

У каждого слоя есть «постоянное» представление, постепенно обновляющееся по мере прохождения новых наблюдений. Именно поэтому RNN не

позволяют использовать придуманную нами структуру `Operation`, которую мы сделали в предыдущих главах: объект `ndarray`, содержащий состояние каждого слоя, постоянно обновляется и многократно используется для составления прогнозов для последовательности данных в RNN. Поскольку мы не можем использовать структуру из предыдущей главы, начать придется с понимания того, какие классы потребуются для работы с RNN.

Первый класс для RNN: RNNLayer

Исходя из описания того, чего мы ожидаем от RNN, становится ясно, что нам понадобится класс `RNNLayer`, который будет передавать последовательность данных по одному элементу последовательности за раз. Давайте теперь подробно рассмотрим, как такой класс должен работать. Как мы говорили в этой главе, RNN работает с данными, в которых каждое наблюдение является двумерным и имеет размерность `(sequence_length, num_features)`; и поскольку с вычислительной точки зрения всегда более эффективно передавать данные в пакетном режиме, класс `RNNLayer` должен принимать трехмерные объекты `ndarray` размера `(batch_size, sequence_length, num_features)`. Однако в предыдущем разделе я объяснил, что мы хотим передавать наши данные через `RNNLayer` по одному элементу последовательности за раз. Как это сделать, имея формат данных, данные `(batch_size, sequence_length, num_features)`? А вот как:

1. Выберем двумерный массив по второй размерности начиная с `data [:, 0, :]`. Этот `ndarray` будет иметь форму `(batch_size, num_features)`.
2. Инициализируем «скрытое состояние» для объекта `RNNLayer`, которое будет постоянно обновляться по мере передачи элементов последовательности, оно будет иметь форму `(batch_size, hidden_size)`. Этот `ndarray` хранит «накопленную информацию» слоя о данных, которые были переданы на предыдущих временных шагах.
3. Пропустим эти два `ndarray` вперед через первый временной шаг в этом слое. В конечном итоге наш `RNNLayer` будет выводить `ndarrays` не той же размерности, что были на входе, в отличие от плотных слоев, поэтому выходные данные будут иметь форму `(batch_size, num_outputs)`. Кроме того, нужно обновить представление нейронной сети для каждого наблюдения: на каждом временном шаге наш `RNNLayer` должен также выводить `ndarray` формы `(batch_size, hidden_size)`.
4. Выбираем следующий двумерный массив из данных: `data [:, 1, :]`.

5. Передаем эти данные, а также значения представлений RNN, выведенных на первом временном шаге, на второй временной шаг на этом слое, чтобы получить еще один вывод формы (`batch_size`, `num_outputs`), а также обновленные представления формы (`batch_size`, `hidden_size`).
6. Продолжаем эти действия, пока через слой не пройдут все временные шаги в количестве `sequence_length`. Затем объединяем все результаты, чтобы получить выходные данные этого слоя формы (`batch_size`, `sequence_length`, `num_outputs`).

Этот алгоритм задает представление о том, как должен работать класс `RNNLayer`, и мы разберемся еще лучше, когда будем писать код. Но это еще не все — нам понадобится еще один класс для получения данных и обновления скрытого состояния слоя на каждом шаге. Для этого мы будем использовать `RNNNode`.

Второй класс для RNN: RNNNode

Исходя из описания, которое мы привели в предыдущем разделе, класс `RNNNode` должен иметь метод `forward` со следующими входами и выходами:

- Два `ndarray` в качестве входных данных:
  - один для ввода данных в сеть, с формой `[batch_size, num_fea tures]`;
  - один для представлений наблюдений на этом временном шаге, с формой `[batch_size, hidden_size]`.
- Два `ndarray` в качестве выходных данных:
  - один для выходных сети на этом временном шаге, с формой `[batch_size, num_outputs]`;
  - один для обновленных представлений наблюдений на этом временном шаге, с формой `[batch_size, hidden_size]`.

Далее мы покажем, как классы `RNNNode` и `RNNLayer` будут работать вместе.

Объединение двух классов

Класс `RNNLayer` оборачивается вокруг списка объектов `RNNNode` и (по крайней мере) будет содержать метод `forward`, у которого будут следующие входные и выходные данные:

- входные данные: пакет последовательностей наблюдений формы `[batch_size, sequence_length, num_features]`;
- выходные данные: выход нейронной сети для этих последовательностей формы `[batch_size, sequence_length, num_outputs]`.

На рис. 6.5 показан порядок передачи данных через RNN с двумя слоями `RNNL` по пять узлов RNN в каждом. На каждом временном шаге входные данные, изначально имеющие размерность `feature_size`, последовательно передаются через первый `RNNNode` в каждом `RNNLayer`, при этом сеть в конечном итоге выводит на этом временном шаге прогноз размерности `output_size`. Кроме того, каждый `RNNNode` передает «скрытое состояние» следующему `RNNNode` в каждом слое. Как только данные каждого из пяти временных шагов пройдут через все слои, мы получим окончательный набор прогнозов формы `(5, output_size)`, где `output_size` должен быть того же размера, что и цель. Затем эти прогнозы сравниваются с целевыми значениями, и рассчитывается градиент потерь на обратном проходе. На рис. 6.5 все это показано вместе на примере того, как данные формата 5×2 `RNNNode` проходят от первого до последнего (10) слоя,

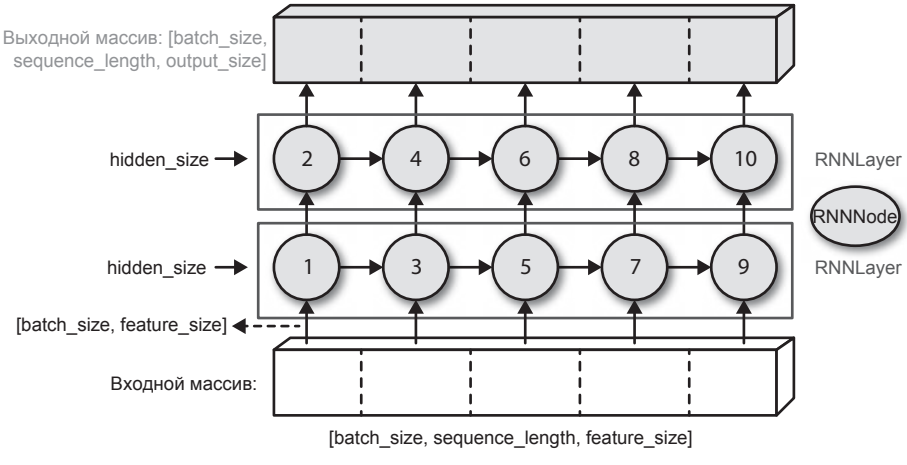


Рис. 6.5. Порядок передачи данных через RNN с двумя слоями, разработанную для обработки последовательностей длиной 5

В качестве альтернативы данные могут проходить через RNN в порядке, показанном на рис. 6.6. Каким бы порядок ни был, в целом, должно произойти следующее:

- Каждый слой должен обрабатывать данные до следующего слоя, например на рис. 6.5 пункт 2 не может произойти раньше 1, а 4 не может произойти раньше 3.
- Аналогично каждый слой должен обрабатывать все свои временные шаги по порядку — например, на рис. 6.5 пункт 4 не может происходить раньше 2, а 3 не может происходить раньше 1.
- Последний слой должен иметь размерность выходных данных `feature_size` для каждого наблюдения.

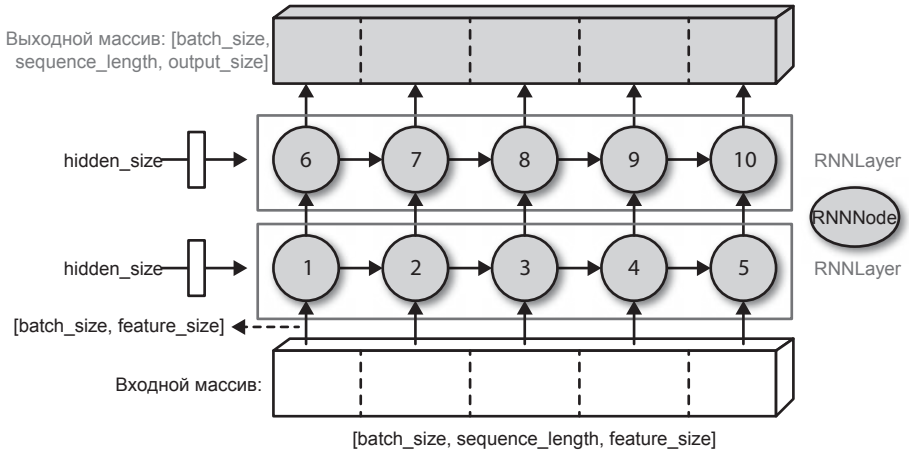


Рис. 6.6. Другой вариант порядка прохождения данных через ту же RNN на прямом проходе

Все это касается прямого прохода RNN. А что насчет обратного прохода?

Обратный проход

Обратное распространение через рекуррентные нейронные сети часто описывается как отдельный алгоритм, называемый «обратное распространение по времени». Название соответствует тому, как это происходит, но звучит намного сложнее, чем на самом деле. Помня наше рассуждение о том, как данные передаются через RNN, мы можем описать, что происходит на обратном проходе, следующим образом: обратный проход по RNN есть передача градиентов в обратном направлении через сеть в порядке, обратном тому, в каком мы передавали входные данные на прямом проходе. То есть мы делаем то же самое, что и в обычных сетях.

Исходя из рис. 6.5 и 6.6, на прямом проходе происходит следующее:

1. Изначально есть серия наблюдений, каждое из которых имеет форму (`feature_size`, `sequence_length`).
2. Эти входные данные разбиваются на отдельные элементы `sequence_length` и передаются в сеть по одному.
3. Каждый элемент проходит через все слои, и в конечном итоге получается выход размера `output_size`.
4. Одновременно с этим слой передает скрытое состояние для расчетов в этом слое на следующем временном шаге.
5. Это проделывается для всех временных шагов `sequence_length`, в результате чего получается выход размера (`output_size`, `sequence_length`).

Обратное распространение работает так же, но наоборот:

1. Изначально у нас есть градиент формы `[output_size, sequence_length]`, который говорит о том, как сильно каждый элемент выходных данных (тоже размера `[output_size, sequence_length]`) в конечном итоге влияет на потери для данной серии наблюдений.
2. Эти градиенты разбиваются на отдельные элементы `sequence_length` и пропускаются через слои в обратном порядке.
3. Градиент каждого элемента передается через все слои.
4. Одновременно с этим слои передают *градиент потерь по отношению к скрытому состоянию для данного временного шага* назад в вычисления слоев на предыдущие шаги.
5. Это продолжается для всех временных шагов `sequence_length`, пока градиенты не будут переданы назад каждому слою в сети, что позволит вычислить градиент потерь по каждому весу, как мы это делаем в обычных нейросетях.

Соотношение между обратным и прямым проходами показано на рис. 6.7, из которого видно, как данные проходят через RNN во время обратного прохода. Вы, конечно, заметите, что это то же самое, что и на рис. 6.5, но с перевернутыми стрелками и измененными числами.

На высоком уровне прямой и обратный проходы для слоя `RNNLay` очень похожи на проходы слоя в обычной нейронной сети: на вход приходит

ndarray определенной формы, на выходе получается ndarray другой формы, а на обратном проходе приходит выходной градиент той же формы, что и их выходные данные, и создается входной градиент той же формы, что и входные данные. Но есть важное отличие в том, как в RNNLayer обрабатываются градиенты веса по сравнению с другими слоями, поэтому кратко рассмотрим этот вопрос, прежде чем перейти к написанию кода.

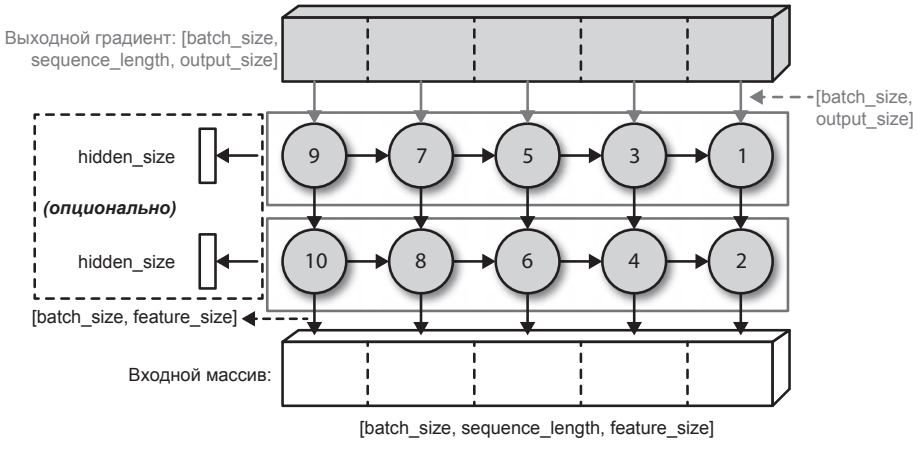


Рис. 6.7. На обратном проходе RNN передают данные в направлении, противоположном тому, как данные передаются во время прямого прохода

Накопление градиентов для весов в РНН

В рекуррентных нейронных сетях, как и в обычных нейросетях, у каждого слоя свой набор весов. Это означает, что один и тот же набор весов будет влиять на вывод слоя на всех временных шагах sequence_length. Следовательно, во время обратного распространения один и тот же набор весов будет получать разные градиенты sequence_length. Например, в кружке под номером «1» в схеме обратного распространения, показанной на рис. 6.7, второй слой получит градиент для последнего временного шага, а в кружке под номером «3» слой получит градиент предпоследнего шага. В обоих случаях будет использоваться один и тот же набор весов. Таким образом, во время обратного распространения нужно будет накапливать градиенты для весов по последовательности временных шагов. Это означает, что независимо от выбранного способа хранения весов придется обновлять градиенты следующим образом:

```
weight_grad += grad_from_time_step
```

Этот подход отличается от слоев `Dense` и `Conv2D`, в которых мы просто сохраняли параметры в аргументе `param_grad`.

Мы раскрыли, как работают RNN и какие классы потребуются для их реализации; теперь поговорим подробнее.

RNN: код

Давайте начнем с тех вещей, в которых реализация RNN будет аналогична другим реализациям нейронных сетей, которые мы рассмотрели в этой книге:

1. RNN по-прежнему передает данные по слоям, при этом на прямом проходе передаются данные, а на обратном проходе — градиенты. Таким образом, независимо от того, каким окажется эквивалент нашего класса NeuralNetwork, у него все равно будет список RNNLayer в качестве атрибута слоев и прямой проход будет реализован примерно так:

```
def forward(self, x_batch: ndarray) -> ndarray:

    assert_dim(ndarray, 3)

    x_out = x_batch
    for layer in self.layers:
        x_out = layer.forward(x_out)

return x_out
```

2. Реализация `Loss` у RNN будут такой же, как и раньше: выходной `ndarray` производится последним `Layer` и сравнивается с вектором `y_batch`, вычисляется одно значение и градиент этого значения относительно входа в `Loss` с той же формой возвращается в качестве вывода. Нужно изменить функцию `softmax`, чтобы она работала соответствующим образом с `ndarrays` формы `[batch_size, sequence_length, feature_size]`, но это не проблема.

3. Класс `Trainer` в основном не меняется: мы циклически перебираем наши обучающие данные, выбираем пакеты входных данных и пакеты выходных данных и непрерывно передаем их через нашу модель, получая значения потерь, которые говорят нам, обучается ли наша модель, и обновляем весовые коэффициенты после каждой партии. Кстати, о весах...

4. Класс `Optimizer` остается прежним. Как мы увидим, придется на каждом временном шаге обновлять способ извлечения `params` и `param_grads`, но «правила обновления» (которые мы записали в функции `_update_rule` в нашем классе) остаются прежними.

Интересные вещи появляются именно в классе `Layer`.

Класс RNNLayer

Ранее мы давали классу `Layer` набор `Operation`, который передавал данные вперед, а градиенты — назад. Слои `RNNLayer` будут совершенно другими. В них должно поддерживаться «скрытое состояние», которое постоянно обновляется по мере поступления новых данных и каким-то образом «объединяется» с данными на каждом временном шаге. Как именно это должно работать? За основу можно взять рис. 6.5 и 6.6. В них предполагается, что у каждого `RNNLayer` должен быть список `RNNNode` в качестве атрибута, после чего каждый элемент последовательности из входных данных слоя должен проходить через каждый `RNNNode`, по одному за раз. Каждый `RNNNode` будет принимать этот элемент последовательности, а также «скрытое состояние» для этого слоя и создавать выходные данные для слоя на этом временном шаге, а также обновлять скрытое состояние слоя.

Чтобы прояснить все это, давайте углубимся в код: рассмотрим по порядку, как должен инициализироваться класс `RNNLayer`, как он должен передавать данные вперед во время прямого прохода и как должен отправлять данные назад во время обратного хода.

Инициализация

У экземпляра `RNNLayer` должны быть:

- целое значение `hidden_size`;
- целое значение `output_size`;
- `ndarray` `start_H` формы `(1, hidden_size)`, представляющей собой скрытое состояние слоя.

Кроме того, как и в обычных нейронных сетях, мы установим флаг `self.first = True` при инициализации слоя. При первой передаче данных в метод `forward` мы передадим полученный массив `ndarray` в метод `_init_params`, инициализируем параметры и установим `self.first = False`.

После инициализации нашего слоя нужно описать передачу данных вперед.

Метод forward

В целом, метод `forward` состоит из приема на вход `ndarray` `x_seq_in` формы `(batch_size, sequence_length, feature_size)` и ее последовательной передачи через все `RNNNode` данного слоя. В приведенном ниже коде `self.nodes` — это `RNNNode` для слоя, а `H_in` — скрытое состояние слоя:

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))

for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

Небольшое замечание о скрытом состоянии `H_in`: скрытое состояние `RNNLayer` обычно представлено в виде вектора, но операции в каждом `RNNNode` требуют, чтобы скрытое состояние имело размерность `ndarray (batch_size, hidden_size)`. Поэтому в начале каждого прямого прохода мы просто «повторяем» скрытое состояние:

```
batch_size = x_seq_in.shape[0]

H_in = np.copy(self.start_H)

H_in = np.repeat(H_in, batch_size, axis=0)
```

После прямого прохода мы берем среднее значение по наблюдениям, составляющим пакет, чтобы получить обновленное скрытое состояние для этого слоя:

```
self.start_H = H_in.mean(axis=0, keepdims=True)
```

Кроме того, из этого кода видно, что у `RNNNode` должен быть метод `forward`, который принимает два массива со следующими размерностями:

- `(batch_size, feature_size)`;
- `(batch_size, hidden_size)`

и возвращает два массива размерностей:

- `(batch_size, output_size)`;
- `(batch_size, hidden_size)`.

Мы рассмотрим реализацию класса `RNNNode` (и ее варианты) в следующем разделе. Но сначала давайте рассмотрим метод `backward` для класса `RNNLayer`.

Метод backward

Так как выходом метода `forward` является `x_seq_out`, у метода `backward` на входе должен быть градиент той же формы, что и `x_seq_out`, под названием `x_seq_out_grad`. Двигаясь в направлении, противоположном прямому методу, мы передаем этот градиент в обратном направлении через узлы RNN, в конечном итоге возвращая `x_seq_in_grad` формы `(batch_size, sequence_length, self.feature_size)` в качестве градиента для всего слоя:

```
h_in_grad = np.zeros((batch_size, self.hidden_size))

sequence_length = x_seq_out_grad.shape[1]

x_seq_in_grad = np.zeros((batch_size, sequence_length,
                          self.feature_size))

for t in reversed(range(sequence_length)):

    x_out_grad = x_seq_out_grad[:, t, :]

    grad_out, h_in_grad = \
        self.nodes[t].backward(x_out_grad, h_in_grad, self.params)

    x_seq_in_grad[:, t, :] = grad_out
```

То есть у `RNNNode` должен быть метод `backward`, который, следуя шаблону, является противоположностью метода `forward`, принимая два массива фигур:

- `(batch_size, output_size)`;
- `(batch_size, hidden_size)`

и возвращая два массива фигур:

- `(batch_size, feature_size)`;
- `(batch_size, hidden_size)`.

И это работа `RNNLayer`. Теперь кажется, что осталось только описать ядро рекуррентной нейронной сети: узлы `RNNNode`, где происходят реальные вычисления. Прежде чем мы это сделаем, давайте проясним роль `RNNNode` и их вариантов в общей работе `RNN`.

Основные элементы узлов RNNNode

При работе с RNN разговор обычно начинается именно с узлов `RNNNode`. Но мы рассмотрим узлы последними, так как до этого момента посредством схем и описаний пытались понять самую суть RNN: как структурируются данные и как данные и скрытые состояния передаются между слоями. Оказывается, существует несколько способов реализации самих `RNNNode`, фактической обработки данных с заданным временным шагом и обновления скрытого состояния слоя. Первый способ — это так называемые «обычные» рекуррентные нейронные сети, которые мы будем также называть «классическими RNN». Однако существуют и другие, более сложные способы реализации RNN. Пример такого способа — это вариант с узлами RNN под названием GRU, что означает «Gated Recurrent Units» (управляемые рекуррентные блоки). Часто говорят, что GRU и другие варианты RNN значительно отличаются от классических RNN, но важно помнить, что в них все равно используется одна и та же структура слоев, которую мы уже рассмотрели. Например, во всех этих сетях одинаково передаются данные во времени и обновляются скрытые состояния на каждом шаге. Единственное отличие между ними — это внутренняя работа этих «узлов».

Еще раз подчеркнем: если бы мы реализовали `GRULayer` вместо `RNNLayer`, код был бы точно таким же! Ядро прямого прохода выглядит следующим образом:

```
sequence_length = x_seq_in.shape[1]

x_seq_out = np.zeros((batch_size, sequence_length, self.output_size))
```

```
for t in range(sequence_length):

    x_in = x_seq_in[:, t, :]

    y_out, H_in = self.nodes[t].forward(x_in, H_in, self.params)

    x_seq_out[:, t, :] = y_out
```

Единственное отличие состоит в том, что каждый «узел» в `self.nodes` будет реализован как `GRUNode` вместо `RNNNode`. Метод `backward` тоже не изменится.

Это также почти справедливо для самого известного варианта классических RNN: LSTM, или ячеек «долгая краткосрочная память» (англ. Long Short Term Memory). Единственная разница состоит в том, что в слоях `LSTMLayer` нужно, чтобы слой запоминал две величины и обновлял их, когда элементы последовательности передаются во времени: в дополнение к «скрытому состоянию» в слое хранится «состояние ячейки», что позволяет лучше моделировать долгосрочные зависимости. В результате возникают небольшие различия между реализациями `LSTMLayer` и `RNNLayer`. Так, у слоев `LSTMLayer` будет два массива `ndarray` для хранения состояния слоя с течением времени:

- `Ndarray start_H` формы `(1, hidden_size)`, где хранится скрытое состояние слоя;
- `Ndarray start_C` формы `(1, cell_size)`, где хранится состояние ячейки слоя.

Каждый узел `LSTMNode` должен принимать входные данные, а также скрытое состояние и состояние ячейки. На прямом проходе это будет выглядеть так:

```
y_out, H_in, C_in = self.nodes[t].forward(x_in, H_in, C_in self.params)
```

а в методе `backward`:

```
grad_out, h_in_grad, c_in_grad = \
    self.nodes[t].backward(x_out_grad, h_in_grad, c_in_grad, self.params)
```

Мы упомянули всего три варианта, но их существует намного больше, например LSTM со «смотровыми глазка́ми» хранят состояние ячейки в дополнение к скрытому состоянию, а некоторые из них поддержива-

ют только скрытое состояние[1]. Но в целом, слой, состоящий из узлов `LSTMPeepholeConnectionNode`, будет вписываться в `RNNLayer` так же, как и все другие реализации, и методы `forward` и `backward` будут такими же. Рассмотренная базовая структура RNN — способ, которым данные направляются вперед через слои, а также вперед с течением времени и затем назад во время обратного прохода, — уникальная черта рекуррентных сетей. Реальные структурные различия между классическими RNN и RNN на основе LSTM относительно невелики, а вот их характеристики могут существенно отличаться.

Теперь давайте посмотрим на реализацию `RNNNode`.

«Классические» узлы RNN

RNN принимают данные по одному элементу последовательности за раз; например, если мы хотим спрогнозировать цену на нефть, на каждом временном шаге RNN будет получать информацию о функциях, которые мы используем для прогнозирования цены на данном временном шаге. Кроме того, RNN хранит в своем «скрытом состоянии» кодировку, представляющую совокупную информацию о том, что произошло на предыдущих временных шагах. Нам нужно превратить эти данные (а именно признаки текущей точки данных и накопленную информацию от всех предыдущих шагов) в прогноз для этого шага, а также в обновленное скрытое состояние.

Чтобы понять, как RNN делает это, вспомним, что происходит в обычной нейронной сети. В прямой нейронной сети каждый слой получает набор «изученных признаков» от предыдущего слоя. Каждый такой набор представляет собой комбинацию исходных признаков, которую сеть «считает» полезными. Затем слой умножает эти признаки на матрицу весов, что позволяет слою изучать объекты, которые являются комбинациями признаков, полученными слоем в качестве входных данных. Чтобы нормализовать выход, мы добавляем к этим новым признакам «смещение» и пропускаем их через функцию активации.

В рекуррентных нейронных сетях нужно сделать так, чтобы наше обновленное скрытое состояние включало в себя и входные данные, и старое

[1] Загляните в «Википедию», чтобы почитать о разных вариантах LSTM, по адресу oreil.ly/2TysrXj.

скрытое состояние. Это похоже на то, что происходит в обычных нейронных сетях:

1. Сначала мы объединяем ввод и скрытое состояние. Затем мы умножаем это значение на матрицу весов, добавляем смещение и передаем результат через тангенциальную функцию активации. Это и будет наше обновленное скрытое состояние.
2. Затем мы умножаем это новое скрытое состояние на весовую матрицу, которая преобразует скрытое состояние в выход с требуемой размерностью. Например, если мы используем этот RNN для прогнозирования одного непрерывного значения на каждом временном шаге, нам нужно умножить скрытое состояние на весовую матрицу размера `(hidden_size, 1)`.

Таким образом, наше обновленное скрытое состояние будет зависеть и от входных данных, полученных на текущем временном шаге, и от предыдущего скрытого состояния, а выходные данные будут являться результатом передачи этого обновленного скрытого состояния через операции полносвязанного слоя.

Пора написать код.

Пишем код для RNNNode

В приведенном ниже коде мы реализуем шаги, описанные абзацем выше. Позже мы сделаем то же самое с GRU и LSTM (как уже поступали с простыми математическими функциями, которые обсуждали в главе 1), и идея будет общая: мы сохраняем все значения, вычисленные на прямом проходе, как атрибуты, хранящиеся в классе `Node`, чтобы затем использовать их для вычисления обратного прохода:

```
def forward(self,
            x_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]
            ) -> Tuple[ndarray]:
    '''
    param x: массив numpy формы (batch_size, vocab_size)
    param H_prev: массив numpy формы (batch_size, hidden_size)
    return self.x_out: массив numpy формы (batch_size, vocab_size)
    return self.H: массив numpy формы (batch_size, hidden_size)
    '''
```

```
    self.X_in = x_in
    self.H_in = H_in

    self.Z = np.column_stack((x_in, H_in))

    self.H_int = np.dot(self.Z, params_dict['W_f']['value']) \
                                + params_dict['B_f']['value']

    self.H_out = tanh(self.H_int)

    self.X_out = np.dot(self.H_out, params_dict['W_v']['value']) \
                                    + params_dict['B_v']['value']

    return self.X_out, self.H_out
```

Еще одно замечание: поскольку здесь мы не используем класс `Param-Operations`, хранить параметры придется по-другому. Мы будем хранить их в словаре `params_dict`, который ссылается на параметры по имени. Кроме того, у каждого параметра будет два ключа: значение и производная (то есть градиент). В прямом проходе нам потребуется только значение.

RNNNode: обратный проход

На обратном проходе `RNNNode` просто вычисляет значения градиентов потерь по отношению к входам в `RNNNode`, учитывая градиенты потерь по отношению к выходам `RNNNode`. Здесь работает такая же логика, которую мы обсуждали в главах 1 и 2. Поскольку `RNNNode` предоставляется в виде последовательности операций, то можно просто вычислить производную каждой операции, рассчитанной на ее входе, и последовательно умножить эти производные на те, что были раньше (по правилам умножения матриц), чтобы получить массивы `ndarray`, где будут храниться градиенты потерь по отношению к каждому из входов. Напишем код:

```
def forward(self,
            x_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]
            ) -> Tuple[ndarray]:
    '''
    param x: массив numpy формы (batch_size, vocab_size)
    param H_prev: массив numpy формы (batch_size, hidden_size)
    return self.x_out: массив numpy формы (batch_size, vocab_size)
```

```
        return self.H: массив numpy формы (batch_size, hidden_size)
        '''
        self.X_in = x_in
        self.H_in = H_in
        self.Z = np.column_stack((x_in, H_in))

        self.H_int = np.dot(self.Z, params_dict['W_f']['value']) \
                                  + params_dict['B_f']['value']

        self.H_out = tanh(self.H_int)

        self.X_out = np.dot(self.H_out, params_dict['W_v']['value']) \
                                        + params_dict['B_v']['value']

        return self.X_out, self.H_out
```

Обратите внимание, что, как и в классе `Operaton` до этого, формы входов в функцию `backward` должны соответствовать формам выходов функции `forward`, а формы выходов `backward` — формам входов `forward`.

Ограничения классических узлов RNN

Вспомним, что цель `RNN` — выявлять зависимости в последовательностях данных. В терминах нашего примера о прогнозировании цены на нефть это означает, что мы должны иметь возможность выявить связь между последовательностью признаков, которые мы определили ранее, и тем, что произойдет с ценой на нефть на следующем шаге. Но сколько последних шагов нужно взять? В отношении цен на нефть можно предположить, что наиболее важной будет «вчерашняя» информация, то есть на один временной шаг назад, «позавчерашняя» будет уже чуть менее важной, и т. д — ценность информации, как правило, снижается по мере продвижения назад во времени.

Для многих задач это верно, но есть и такие области применения RNN, где требуется изучать много данных и длительные зависимости. Классический пример — *моделирование языка*, то есть построение модели, которая сможет предсказать следующий символ, слово или часть слова, учитывая весьма длинный ряд предыдущих слов или символов (это распространенная задача, и мы обсудим некоторые ее особенности позже в этой главе). Для этого классических RNN обычно недостаточно. Теперь, когда мы

знаем чуть больше, становится ясно, почему: на каждом временном шаге скрытое состояние умножается на одну и ту же весовую матрицу. Рассмотрим, что происходит, когда мы умножаем число на значение `x` снова и снова: если `x < 1`, число уменьшается в геометрической прогрессии до 0, и если `x > 1`, число увеличивается экспоненциально до бесконечности. У рекуррентных нейронных сетей та же проблема: поскольку на каждом временном шаге один и тот же набор весов умножается на скрытое состояние, градиент весов со временем становится либо чрезвычайно маленьким, либо чрезвычайно большим. Первая проблема известна как «затухание градиента», а вторая — «взрыв градиента». Оба эти явления затрудняют обучение RNN моделированию долгосрочных зависимостей (50–100 временных шагов), необходимых для высококачественного моделирования языка. Есть две известные модификации классических архитектур RNN, которые мы рассмотрим далее и у которых эта проблема в значительной степени решена.

Решение: узлы GRUNode

Классические RNN берут входные значения и скрытое состояние, объединяют их, а затем с помощью умножения матриц определяют то, насколько важна информация в скрытом состоянии по сравнению с входной информацией для прогнозирования выходных данных. Более продвинутые варианты RNN строятся вокруг идеи о том, что для моделирования долгосрочных зависимостей, таких как языковые, *мы иногда получаем информацию, которая говорит, что нужно «забывать» или «сбрасывать» наше скрытое состояние*. Простой пример — символы точки «.» или двоеточия «:». Если модель встречает один из этих символов, то знает, что нужно забыть все, что было ранее, и начинать моделировать новую последовательность с нуля.

Первый вариант RNN с этой идеей — это GRU, или Gated Recurrent Units (управляемые рекуррентные нейроны), названные так потому, что вход и предыдущее скрытое состояние проходят через серию «шлюзов».

1. Первый шлюз аналогичен операциям, которые выполняются в классических RNN: входное и скрытое состояния объединяются, умножаются на матрицу весов, а затем проходят через сигмоиду. Это шлюз «обновления».

2. Второй шлюз — это шлюз «сброса»: входное и скрытое состояния объединяются, умножаются на весовую матрицу, проходят через сигмоиду, а затем умножаются на предшествующее скрытое состояние. Это позволяет сети «учиться забывать», что было в скрытом состоянии, учитывая текущий вход.
3. Затем выход второго шлюза умножается на другую матрицу и передается через функцию `Tanh`, причем выходные данные получаются новым «потенциальным» скрытым состоянием.
4. Наконец, в скрытое состояние попадает шлюз обновления, умноженный на «потенциальное» новое скрытое состояние плюс старое скрытое состояние, умноженное на 1, минус шлюз обновления.

В этой главе мы рассмотрим два усовершенствованных варианта классических RNN: GRU и LSTM. LSTM более популярны и были изобретены задолго до GRU. Однако GRU — это более простая версия LSTM, в которой яснее видно, как идея шлюзов может позволить RNN «научиться сбрасывать» свое скрытое состояние с учетом входных данных, поэтому начнем именно с GRU.

GRUNode: схема

На рис. 6.8 узел `GRUNode` изображен как серия шлюзов. Каждый шлюз содержит операции плотного слоя: умножение на матрицу весов, добавление смещения и пропуск результата через функцию активации. Функции активации — сигмоида, и тогда результат оказывается в диапазоне от 0 до 1, либо `Tanh`, в этом случае получается диапазон от –1 до 1. Диапазон каждого промежуточного `ndarray`, созданного следующим, показан под именем массива.

На рис. 6.8, как и на рис. 6.9 и 6.10, входные данные узла окрашены в зеленый цвет, вычисленные промежуточные величины окрашены в синий цвет, а выходные данные — в красный. Все веса (на рисунке не показаны) содержатся в шлюзах.

Обратите внимание, что для обратного распространения через такую сеть нам нужно будет использовать последовательность экземпляров `Operation`, вычислить производную каждой операции по отношению к ее входу и перемножить результаты. Здесь это явно не показано, так как

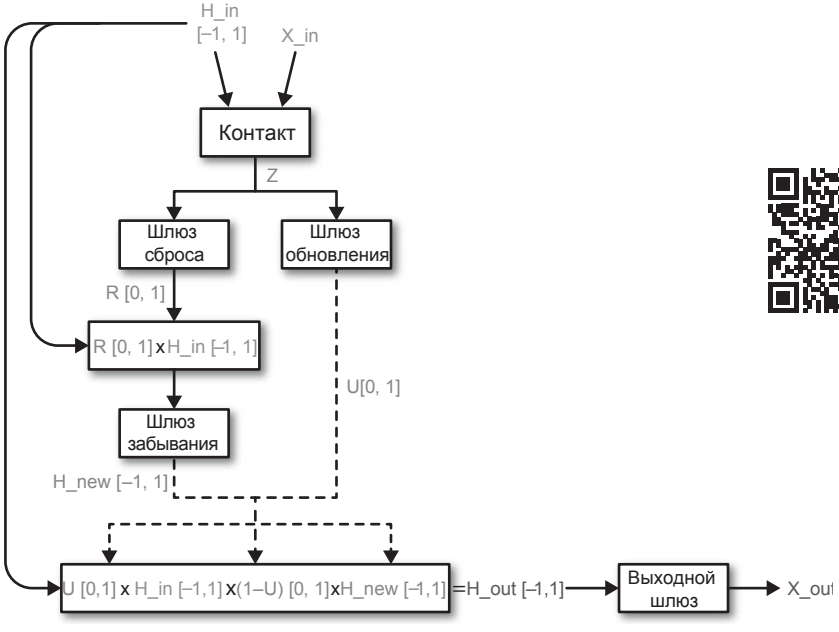


Рис. 6.8. Передача потока данных через GRUNode и его шлюзы, вычисление X_out и H_out

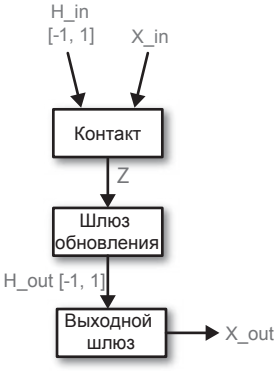


Рис. 6.9. Поток данных пропускается через RNNNode, проходя всего через два шлюза и создавая X_out и H_out

шлюзы (которые на самом деле являются группами из трех операций) показаны как один блок. Но все же мы знаем, как осуществлять обратное распространение через `Operation`, из которых состоит каждый шлюз, поэтому сейчас будем придерживаться терминов шлюзов.

Действительно, на рис. 6.9 показано представление классического RNNNode с использованием шлюзов.

Можно представлять Operation, которые мы описали ранее, как пропуск входа и скрытого состояния через два шлюза.

GRUNodes: код

В следующем коде реализован прямой проход через GRUNode, как описано ранее:

```
def forward(self,
            X_in: ndarray,
            H_in: ndarray,
            params_dict: Dict[str, Dict[str, ndarray]]) ->
            Tuple[ndarray]:
    '''
    param X_in: размер numpy формы (batch_size, vocab_size)
    param H_in: размер numpy формы (batch_size, hidden_size)
    return self.X_out: размер numpy формы (batch_size, vocab_size)
    return self.H_out: размер numpy формы (batch_size, hidden_size)
    '''
    self.X_in = X_in
    self.H_in = H_in

    # шлюз сброса
    self.X_r = np.dot(X_in, params_dict['W_xr']['value'])
    self.H_r = np.dot(H_in, params_dict['W_hr']['value'])

    # шлюз обновления
    self.X_u = np.dot(X_in, params_dict['W_xu']['value'])
    self.H_u = np.dot(H_in, params_dict['W_hu']['value'])

    # шлюзы
    self.r_int = self.X_r + self.H_r + params_dict['B_r']['value']
    self.r = sigmoid(self.r_int)
    self.u_int = self.X_r + self.H_r + params_dict['B_u']['value']
    self.u = sigmoid(self.u_int)

    # новое состояние
    self.h_reset = self.r * H_in
    self.X_h = np.dot(X_in, params_dict['W_xh']['value'])
    self.H_h = np.dot(self.h_reset, params_dict['W_hh']['value'])
```

```
    self.h_bar_int = self.X_h + self.H_h + params_dict['B_h']['value']
    self.h_bar = np.tanh(self.h_bar_int)

    self.H_out = self.u * self.H_in + (1 – self.u) * self.h_bar

    self.X_out = (
np.dot(self.H_out, params_dict['W_v']['value']) \
+ params_dict['B_v']['value']
)

    return self.X_out, self.H_out
```

Обратите внимание, что мы явным образом не объединяем массивы `X_in` и `H_in`, поскольку в отличие от `RNNNode`, где они всегда используются вместе, здесь они используются по отдельности в `GRUNodes`; в частности, мы используем `H_in` независимо от `X_in` в строке `self.h_reset = self.r * H_in`.

Метод `backward` можно найти на сайте книги (https://oreil.ly/2P0lG1G) — там реализован простой проход через операции, составляющие `GRUNode`, вычисляется производная каждой операции по отношению к ее входу, и результаты перемножаются.

LSTMNodes

Ячейки долгой краткосрочной памяти, или LSTM, являются наиболее популярной модификацией классических RNN. Причина заключается в том, что они были изобретены на заре глубокого обучения, еще в 1997 году[1], а все альтернативы, в том числе GRU, появились в последние несколько лет (например, GRU были предложены в 2014 году).

Как и GRU, LSTM дают RNN возможность «сбрасывать» или «забывать» свое скрытое состояние при получении новых входных данных. В GRU это достигается путем передачи входного и скрытого состояния через серию шлюзов, а также вычисления «потенциального» нового скрытого состояния с их помощью — `self.h_bar`, вычисленного с помощью шлюза `self.r`. После этого вычисляется окончательное скрытое состояние с использованием средневзвешенного значения предлагаемого нового скрытого состояния и старого скрытого состояния, что управляется шлюзом обновления:

```
self.H_out = self.u * self.H_in + (1 – self.u) * self.h_bar
```

[1] См. статью LSTM *Long Short-Term Memory*, автор Hochreiter и соавт. (1997).

В LSTM, напротив, используется *отдельный вектор «состояния», так называемое «состояние ячейки», которым определяется, нужно ли «забыть» то, что находится в скрытом состоянии*. Затем с помощью двух других шлюзов контролируется степень сброса или обновления *состояния ячейки*, а четвертый шлюз определяет степень обновления скрытого состояния, основываясь на последнем известном состоянии ячейки[1].

LSTMNode: визуализация

На рис. 6.10 показана схема `LSTMNode` с операциями, представленными в качестве шлюзов.

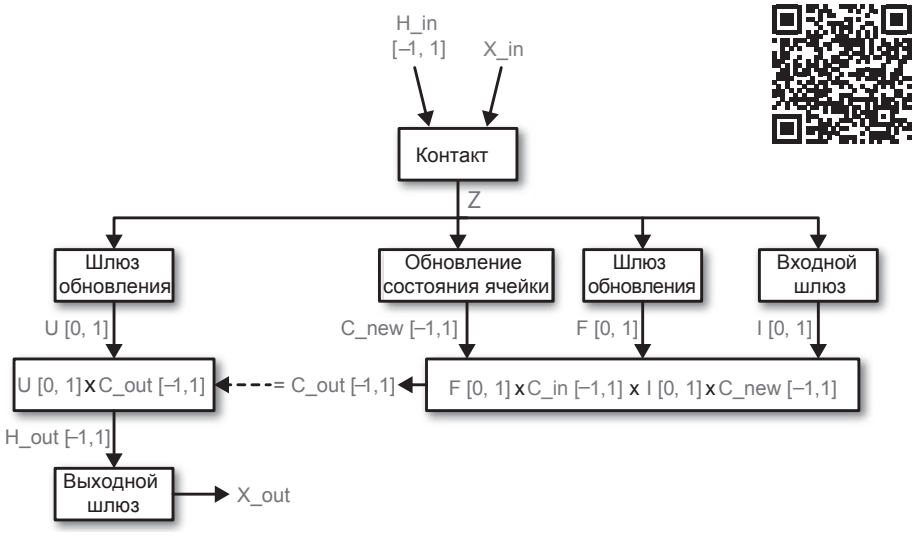


Рис. 6.10. Поток данных передается через LSTMNode, проходя через серию шлюзов и выводя обновленные состояния ячеек и скрытые состояния C_out и H_out соответственно вместе с фактическим выходом X_out

LSTM: код

Как и в случае `GRUNode`, полный код для `LSTMNode`, включая метод `backward` и пример, показывающий, как эти узлы вписываются в `LSTMLayer`, при-

[1] По крайней мере, стандартный вариант LSTM. Как уже упоминалось, есть и другие варианты, такие как «LSTM со смотровыми глазка́ми», шлюзы в которых расположены по-разному.

веден на сайте книги (https://oreil.ly/2P0lG1G). Здесь мы просто покажем метод forward:

```
def forward(self,
    X_in: ndarray,
    H_in: ndarray,
    C_in: ndarray,
    params_dict: Dict[str, Dict[str, ndarray]]):
    '''
    param X_in: массив numpy формы (batch_size, vocab_size)
    param H_in: массив numpy формы (batch_size, hidden_size)
    param C_in: массив numpy формы (batch_size, hidden_size)
    return self.X_out: массив numpy формы (batch_size, output_size)
    return self.H: массив numpy формы (batch_size, hidden_size)
    return self.C: массив numpy формы (batch_size, hidden_size)
    '''

    self.X_in = X_in
    self.C_in = C_in

    self.Z = np.column_stack((X_in, H_in))
    self.f_int = (
        np.dot(self.Z, params_dict['W_f']['value']) \
        + params_dict['B_f']['value']
        )
    self.f = sigmoid(self.f_int)

    self.i_int = (
        np.dot(self.Z, params_dict['W_i']['value']) \
        + params_dict['B_i']['value']
        )
    self.i = sigmoid(self.i_int)

    self.C_bar_int = (
        np.dot(self.Z, params_dict['W_c']['value']) \
        + params_dict['B_c']['value']
        )
    self.C_bar = tanh(self.C_bar_int)
    self.C_out = self.f * C_in + self.i * self.C_bar

    self.o_int = (
        np.dot(self.Z, params_dict['W_o']['value']) \
```

```
        + params_dict['B_o']['value']
        )
    self.o = sigmoid(self.o_int)
    self.H_out = self.o * tanh(self.C_out)

    self.X_out = (
        np.dot(self.H_out, params_dict['W_v']['value']) \
        + params_dict['B_v']['value']
        )

    return self.X_out, self.H_out, self.C_out
```

Это был последний компонент нашей инфраструктуры RNN, который понадобится для начала обучения моделей! Осталось рассмотреть еще один момент: как представить текстовые данные в форме, позволяющей передавать их в RNN.

Представление данных для языковой модели на основе RNN на уровне символов

Языковое моделирование — это одна из наиболее распространенных задач, для которых используются RNN. Как преобразовать последовательность символов в набор обучающих данных, чтобы RNN смогла предсказывать следующий символ? Самый простой способ — использовать *горячую кодировку*. Она работает следующим образом: каждая буква представляется в виде вектора размером, равным *размеру словаря* или количеству букв в общем наборе текста, по которому мы обучаем сеть (он рассчитывается заранее и жестко задан как гиперпараметр сети). Затем каждая буква представляется в виде вектора с 1 в позиции, соответствующей этой букве, и с 0 на других позициях. Наконец, векторы всех букв просто объединяются, формируя общее представление последовательности букв.

Ниже приведен простой пример того, как это будет выглядеть со словарем из четырех букв `a`, `b`, `c` и `d`, где первой буквой будет `a`, второй — `b` и т. д.:

$$abcdb\left[\begin{bmatrix}1\\0\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\right]=\begin{bmatrix}1&0&0&0&0\\0&1&0&0&0\\0&0&1&0&0\\0&0&0&1&0\end{bmatrix}.$$

Такой двумерный массив заменяет одно наблюдение формы `(sequence_length, num_features)` = `(5, 4)` в серии последовательностей. Если бы у нас был текст `abcdba` длины 6 и мы хотели бы передать последовательности длиной 5 в наш массив, первая последовательность была бы преобразована в предыдущую матрицу, а вторая последовательность имела бы вид:

$$bcdba\begin{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\\1\end{bmatrix}\begin{bmatrix}0\\1\\0\\0\end{bmatrix}\begin{bmatrix}1\\0\\0\\0\end{bmatrix}\end{bmatrix}=\begin{bmatrix}0&0&0&0&1\\1&0&0&1&0\\0&1&0&0&0\\0&0&1&0&0\end{bmatrix}.$$

Затем они объединяются, формируя входные данные для `RNN` формы `(batch_size, sequence_length, vocab_size)` = `(2, 5, 4)`. Продолжая «склеивать» фрагменты, мы можем взять необработанный текст и преобразовать его в набор последовательностей для передачи в RNN.

В разделе главы 6 в репозитории GitHub (https://oreil.ly/2P0lG1G) это записано в коде класса `RNNTrainer`, который может принимать необработанный текст, предварительно обрабатывать его, используя методы, описанные здесь, и подавать его в RNN партиями.

Другие задачи моделирования языка

Ранее мы этого не говорили, но, как видно из предыдущего кода, все варианты узлов `RNNNode` позволяют слоям `RNNLayer` выводить количество признаков, отличное от того, которое было на входе. Последний шаг у всех трех узлов — умножение конечного скрытого состояния сети на весовую матрицу, к которой мы получаем доступ через `params_dict[W_v]`. Второе измерение этой весовой матрицы определяет размерность вывода слоя. Это позволяет нам использовать одну и ту же архитектуру для разных задач моделирования языка, просто изменяя аргумент `output_size` в каждом слое.

Например, мы только что рассмотрели построение языковой модели с помощью «предсказания следующего символа». В этом случае размер вывода будет равен размеру словаря: `output_size = vocab_size`. А, например, для сентимент-анализа, последовательности, которые мы передаем, могут просто иметь метку «0» или «1» — положительную или отрицательную. В этом случае мы получим `output_size = 1`, а выходные данные

будем сравнивать с целью только после того, как мы передадим всю последовательность. Схема работы показана на рис. 6.11.

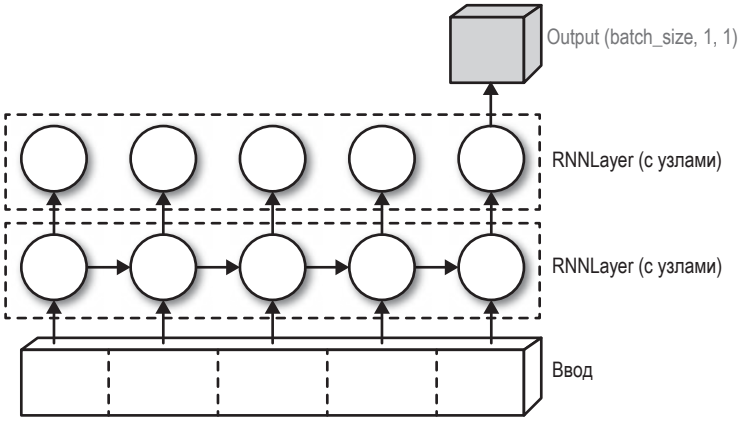


Рис. 6.11. Для сентимент-анализа RNN сравнивает прогноз с фактическими значениями и вырабатывает градиенты только для вывода последнего элемента последовательности; тогда обратное распространение будет продолжаться как обычно, при этом каждый из узлов, который не является последним, просто получит массив «X_grad_out» со всеми нулями

Такая структура позволяет реализовать различные задачи моделирования языка. Если точнее, в ней можно выполнять любые задачи моделирования, в которой данные являются последовательными и вводятся в сеть по одному элементу последовательности за раз.

Прежде чем закончить, рассмотрим еще один редко обсуждаемый аспект RNN: как можно смешивать и совмещать разные типы слоев — `GRULayer`, `LSTMLayer` и другие.

Объединение вариантов RNNLayer

Совместить различные типы `RNNLayer` очень просто: каждый слой `RNN` выводит `ndarray` формы `(batch_size, sequence_length, output_size)`, который можно передать на следующий слой. Как и в случае плотных слоев, указывать `input_shape` не нужно. Вместо этого мы просто задаем веса, основываясь на первом массиве, который слой получает в качестве входных данных, чтобы получить соответствующую форму с учетом входных данных. Таким образом, `RNNModel` может иметь атрибут `self.layers`:

```
[RNNLayer (hidden_size = 256, output_size = 128),
RNNLayer (hidden_size = 256, output_size = 62)]
```

Как и в случае с полносвязными нейронными сетями, мы просто должны быть уверены, что последний уровень производит вывод желаемой размерности; здесь, если мы имеем дело со словарем размера 62 и выполняем предсказание следующего символа, наш последний уровень должен иметь `размер_раздела 62`, так же как последний уровень в наших полностью подключенных нейронных сетях, имеющих дело с проблемой MNIST, должен иметь измерение 10.

После прочтения этой главы должно быть ясно (хотя это нечасто рассматривается при обработке RNN), что поскольку каждый вид слоя, который мы видели, имеет одинаковую базовую структуру, состоящую из последовательностей измерения `feature_size` и вывода последовательностей измерения `output_size`, то можно легко сложить разные виды слоев. Например, на веб-сайте книги (https://oreil.ly/2P0lG1G) мы обучаем `RNNModel` с атрибутом `self.layers`:

```
[GRULayer (hidden_size = 256, output_size = 128),
LSTMLayer (hidden_size = 256, output_size = 62)]
```

Иными словами, первый слой передает свои входные данные во времени с использованием `GRUNode`, а затем передает `ndarray` формы `(batch_size, sequence_length, 128)` на следующий слой, который впоследствии пропускает их через свои `LSTMNode`.

Соединяем все вместе

Классическое упражнение для иллюстрации эффективности RNN — научить нейросеть писать текст в определенном стиле. На веб-сайте книги (https://oreil.ly/2P0lG1G) мы выложили пример кода с моделью, определенной с использованием абстракций, описанных в этой главе. Модель обучается писать текст в стиле Шекспира. Единственный компонент, который мы не показали, — это класс `RNNTrainer`, который перебирает обучающие данные, предварительно обрабатывает их и передает их через модель. Основное различие между этим учителем и тем, что мы видели ранее, состоит в том, что в RNN при выборе пакета данных для передачи нужно сначала предварительно обработать его, закодировать каждую букву и объединить результирующие векторы в последовательность для

преобразования каждой строки длиной `sequence_length` в `ndarray` формы `(sequence_length, vocab_size)`.

Но как только данные предварительно обработаны и модель определена, RNN обучается так же, как и другие знакомые нам нейронные сети: партии передаются одна за одной, прогнозы модели сравниваются с целями для расчета потерь, а затем значения потерь обратно распространяются через операции, которые составляют модель для обновления весов.

Заключение

В этой главе мы изучили рекуррентные нейронные сети — особый тип архитектуры нейронных сетей, предназначенный для обработки последовательностей данных, а не отдельных операций. Вы узнали, что RNN состоят из слоев, которые передают данные во времени, обновляя свои скрытые состояния (и состояния их ячеек в случае LSTM) по мере работы. Мы узнали подробности о продвинутых вариантах RNN, GRU и LSTM и о том, как они передают данные через серию шлюзов на каждом временном шаге. Также мы узнали, что последовательности данных во всех модификациях обрабатываются примерно одинаковым образом, общая структура одинаковая и отличается лишь конкретными операциями, которые применяются на каждом временном шаге.

Надеемся, что теперь вы знаете чуть больше по этой сложной теме. В главе 7 мы завершим повествование, перейдя к практической стороне глубокого обучения, а также покажем, как реализовать то, о чем мы говорили до сих пор, с помощью фреймворка PyTorch, высокопроизводительного, основанного на автоматическом дифференцировании фреймворка для построения и обучения моделей глубокого обучения. Вперед!

ГЛАВА 7

Библиотека PyTorch

Из глав 5 и 6 вы узнали, как работают сверточные и рекуррентные нейронные сети, реализовав их с нуля. Понимание принципов, разумеется, необходимо, но одним лишь знанием нельзя заставить их работать над реальной проблемой. Теперь нам нужна возможность реализовать их в высокопроизводительной библиотеке. Можно было бы написать целую книгу по созданию высокопроизводительной библиотеки нейронных сетей, но это была бы совсем другая (или просто намного более длинная) книга для совсем другой аудитории. Вместо этого мы посвятим последнюю главу знакомству с PyTorch — набирающей популярность средой для нейронных сетей, основанной на автоматическом дифференцировании (мы упоминали ее в главе 6).

Как и в остальной части книги, мы напишем наш код так, чтобы он соответствовал принципам работы нейронных сетей, — напишем классы `Layer`, `Trainer` и т. д. При этом мы не будем писать наш код в соответствии с распространенными практиками PyTorch, но дадим ссылки на репозиторий GitHub (https://oreil.ly/2N4H8jz), чтобы вы могли больше узнать о том, как писать нейронные сети так, как это задумано PyTorch. Но прежде давайте начнем с изучения типа данных в ядре PyTorch, которое реализует автоматическое дифференцирование и, следовательно, способность четко выражать обучение нейронной сети: `Tensor`.

Класс PyTorch Tensor

В предыдущей главе мы показали, как класс `NumberWithGrad` накапливает градиенты, отслеживая операции, выполняемые над ним. Это означало, что если мы написали:

```
a = NumberWithGrad(3)

b = a * 4
```

```
c = b + 3
d = (a + 2)
e = c * d
e.backward()
```

тогда a.grad будет равно 35, что на самом деле является частной производной от e по отношению к a.

Класс Tensor в PyTorch работает как «ndarrayWithGrad»: он похож на NumberWith Grad, за исключением использования массивов (например, numpy) вместо простых чисел с плавающей точкой и целых чисел. Перепишем предыдущий пример с использованием класса Tensor. Сначала мы инициализируем Tensor вручную:

```
a = torch.Tensor ([[3., 3.,],
                   [3., 3.]], require_grad = True)
```

Обратите внимание на пару вещей:

- мы можем инициализировать класс Tensor, просто оборачивая данные, содержащиеся в нем, в torch.Tensor, как мы это делали с массивами ndarray;
- при такой инициализации Tensor мы должны передать аргумент require_grad = True, чтобы сказать Tensor накапливать градиенты.

Как только мы это сделаем, мы можем выполнить вычисления, как и раньше:

```
b = a * 4
c = b + 3
d = (a + 2)
e = c * d
e_sum = e.sum()
e_sum.backward()
```

По сравнению с примером NumberWithGrad здесь появился еще один шаг: мы должны *суммировать* e перед обратным вызовом на сумму. Мы уже говорили в первой главе, что не имеет смысла думать о «производной числа по массиву». Зато можно сказать, какой будет частная производная e_sum по отношению к каждому элементу a — и действительно, мы видим, что ответ соответствует тому, что мы выяснили в предыдущих главах:

```
print(a.grad)

tensor([[35., 35.],
        [35., 35.]], dtype=torch.float64)
```

Эта особенность PyTorch позволяет нам определять модели, задавая прямой проход, вычисляя потерю и вызывая функцию `.backward`, чтобы автоматически вычислить производную каждого из параметров относительно этой потери. В частности, не нужно думать о повторном использовании одного и того же количества в прямом проходе (что ранее не давала делать структура класса `Operation`, которую мы использовали в первых нескольких главах). Как показывает этот простой пример, градиенты сами начнут вычисляться правильно, как только мы вызовем результаты наших вычислений в обратном направлении.

В следующих нескольких разделах мы покажем, как фреймворк обучения, который мы рассмотрели ранее в книге, реализуется с помощью типов данных PyTorch.

Глубокое обучение с PyTorch

Как мы уже видели, у моделей глубокого обучения есть несколько элементов, которые совокупно создают обученную модель:

- `Model` (модель), которая содержит `Layers` (слои).
- `Optimizer` (оптимизатор).
- `Loss` (потери).
- `Trainer` (учитель).

Оказывается, что `Optimizer` и `Loss` реализуются в PyTorch одной строкой кода, а с `Model` и `Layer` все чуть сложнее. Давайте рассмотрим каждый из этих элементов по очереди.

Элементы PyTorch: Model, Layer, Optimizer и Loss

Ключевой особенностью PyTorch является возможность определять модели и слои как простые в использовании объекты, которые автоматически отправляют градиенты назад и сохраняют параметры, просто наследуя

их от класса `torch.nn.Module`. Позже в этой главе вы увидите, как эти части собираются вместе. Сейчас просто знайте, что слой `PyTorchLayer` записывается так:

```
from torch import nn, Tensor

class PyTorchLayer(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

а `PyTorchModel` можно записать так:

```
class PyTorchModel(nn.Module):

    def __init__(self) -> None:
        super().__init__()

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        raise NotImplementedError()
```

Иными словами, каждый подкласс `PyTorchLayer` или `PyTorchModel` просто должен реализовать методы `__init__` и `forward`, что позволит нам использовать их интуитивно понятными способами[1].

Флаг вывода

Как мы видели в главе 4, из-за отсева данных нужна возможность подстраивать поведение нашей модели в зависимости от того, работает ли она в режиме обучения или в режиме вывода. В PyTorch мы можем переключать модель или слой из режима обучения (поведение по умолчанию) в режим вывода, запустив функцию `m.eval` на модели или слое (любой

[1] Написание слоев и моделей таким способом с использованием PyTorch не рекомендуется и не применяется. Здесь мы пишем так только для целей иллюстрации понятий, которые рассмотрели до сих пор. Более распространенный способ построения блоков нейронной сети с помощью PyTorch приведен во вводном руководстве из официальной документации.

объект, который наследуется от `nn.Module`). Кроме того, в PyTorch есть элегантный способ быстро изменять поведение всех подклассов слоя с помощью функции `apply`. Если мы определим:

```
def inference_mode(m: nn.Module):
    m.eval()
```

тогда мы можем включить следующее:

```
if inference:
    self.apply(inference_mode)
```

в метод `forward` каждого подкласса `PyTorchModel` или `PyTorchLayer`, который мы определяем, получая желаемый флаг.

Давайте посмотрим, как все это соединить.

Реализация строительных блоков нейронной сети с использованием PyTorch: DenseLayer

Теперь у нас есть все предпосылки для того, чтобы начать реализовывать слои, которые мы видели ранее, но с применением операций PyTorch. Слой `DenseLayer` описывается следующим образом:

```
class DenseLayer(PyTorchLayer):
    def __init__(self,
                 input_size: int,
                 neurons: int,
                 dropout: float = 1.0,
                 activation: nn.Module = None) -> None:
        super().__init__()
        self.linear = nn.Linear(input_size, neurons)
        self.activation = activation
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 – dropout)

    def forward(self, x: Tensor,
                inference: bool = False) -> Tensor:
        if inference:
            self.apply(inference_mode)

        x = self.linear(x) # does weight multiplication + bias
```

```
        if self.activation:
            x = self.activation(x)
        if hasattr(self, "dropout"):
            x = self.dropout(x)

        return x
```

С помощью функции `nn.Linear` мы увидели наш первый пример операции PyTorch, которая автоматически выполняет обратное распространение. Этот объект не только реализует умножение веса и добавление члена смещения в прямом проходе, но также вызывает накопление градиентов `x`, поэтому производные потери по параметрам в обратном проходе вычисляются правильно. Также обратите внимание, что поскольку все операции PyTorch наследуются от `nn.Module`, мы можем вызывать их как математические функции: например, в предыдущем случае мы пишем `self.linear (x)`, а не `self.lin ear.forward (x)`. Это также относится и к самому `DenseLayer`, как мы увидим, когда будем использовать его в будущей модели.

Пример: моделирование цен на жилье в Бостоне в PyTorch

Используя этот слой в качестве строительного блока, мы можем реализовать уже знакомую модель цен на жилье, которую упоминали в главах 2 и 3. Напомним, что в этой модели был один скрытый слой с сигмовидной функцией активации. В главе 3 мы реализовали это в нашей объектно-ориентированной среде, в которой были класс для слоев и модель, а в качестве атрибута слоев был список длины 2. Точно так же мы можем определить класс `HousePricesModel`, который наследуется от `PyTorchModel`:

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13,
                 hidden_dropout: float = 1.0):
        super().__init__()
        self.dense1 = DenseLayer(13, hidden_size,
                                 activation=nn.Sigmoid(),
                                 dropout = hidden_dropout)
        self.dense2 = DenseLayer(hidden_size, 1)
```

```
    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)
        assert x.shape[1] == 13

        x = self.dense1(x)
        return self.dense2(x)
```

Теперь создаем экземпляр:

```
pytorch_boston_model = HousePricesModel (hidden_size = 13)
```

Обратите внимание, что в моделях `PyTorch` писать отдельный класс `Layer` не принято. Чаще всего просто определяют модели с точки зрения отдельных выполняемых операций, используя что-то вроде такого:

```
class HousePricesModel(PyTorchModel):

    def __init__(self,
                 hidden_size: int = 13):
        super().__init__()
        self.fc1 = nn.Linear(13, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, x: Tensor) -> Tensor:

        assert_dim(x, 2)

        assert x.shape[1] == 13

        x = self.fc1(x)
        x = torch.sigmoid(x)
        return self.fc2(x)
```

При создании своих моделей PyTorch вы, возможно, будете писать свой код именно так, а не создавать отдельный класс `Layer`. При чтении кода других пользователей вы почти всегда увидите что-то похожее на показанный выше код.

Слои и модели реализуются сложнее, чем оптимизаторы и потери, о которых мы расскажем далее.

Элементы PyTorch: Optimizer и Loss

Оптимизаторы и потери реализованы в PyTorch одной строкой кода. Например, потеря `SGDMomentum`, которую мы рассмотрели в главе 4, пишется так:

```
import torch.optim as optim

optimizer = optim.SGD(pytorch_boston_model.parameters(), lr=0.001)
```

В PyTorch модели передаются в оптимизатор в качестве аргумента. Такой способ гарантирует, что оптимизатор «указывает» на правильные параметры модели, поэтому знает, что обновлять на каждой итерации (ранее мы делали это с помощью класса `Trainer`).

Кроме того, среднеквадратическую потерю, которую мы видели в главе 2, и `SoftmaxCrossEntropyLoss`, которая обсуждалась в главе 4, тоже можно записать просто:

```
mean_squared_error_loss = nn.MSELoss ()
softmax_cross_entropy_loss = nn.CrossEntropyLoss ()
```

Как и слои, они наследуются от `nn.Module`, поэтому их можно вызывать так же, как слои.

Обратите внимание: в названии класса `nn.CrossEntropyLoss` отсутствует слово `softmax`, но сама операция `softmax` там выполняется, так что мы можем передавать «сырой» вывод нейронной сети, не реализуя `softmax` вручную.

Этот вариант `Loss` наследуется от `nn.Module`, как и `Layer` из примера выше, поэтому их можно вызывать одинаково, например с помощью `loss(x)` вместо `loss.forward (x)`.

Элементы PyTorch: Trainer

`Trainer` (учитель) объединяет все эти элементы. Какие к нему предъявляются требования? Мы знаем, что он должен реализовать общую модель обучения нейронных сетей, которая уже неоднократно встречалась в этой книге:

1) пакет данных передается через модель;
2) результаты и целевые значения передаются в функцию потерь, чтобы вычислить значение потерь;
3) вычисляется градиент потерь по всем параметрам;
4) оптимизатор обновляет параметры согласно некоторому правилу.

В PyTorch все это работает точно так же, за исключением двух небольших моментов:

- по умолчанию оптимизаторы сохраняют градиенты параметров (`param_grads`) после каждой итерации обновления параметров. Чтобы очистить эти градиенты перед следующим обновлением параметра, нужно вызывать функцию `self.optim.zero_grad`;
- как было показано ранее в простом примере с автоматическим дифференцированием, чтобы начать обратное распространение, нужно вызвать функцию `loss.backward` после вычисления значения потерь.

Таким образом мы получаем следующий код, который приводится в курсах по PyTorch и фактически будет использоваться в классе `PyTorchTrainer`. Как и класс `Trainer` из предыдущих глав, `PyTorchTrainer` принимает на вход оптимизатор, `PyTorchModel` и потери (либо `nn.MSELoss`, либо `nn.CrossEntropyLoss`) для пакета данных (`X_batch, y_batch`). Имея объекты как `self.optim`, `self.model` и `self.loss`, запускаем обучение модели следующим кодом:

```
# Сначала обнуляем градиенты
self.optim.zero_grad()

# пропускаем X_batch через модель
output = self.model(X_batch)

# вычисляем потери
loss = self.loss(output, y_batch)

# выполняем обратное распространение на потерях
loss.backward()

# вызываем функцию self.optim.step() (как и раньше), чтобы обновить
# параметры
self.optim.step()
```

Это самые важные строки. Остальной код для PyTorch `Trainer` в основном похож на код для `Trainer`, который мы видели в предыдущих главах:

```
class PyTorchTrainer(object):
    def __init__(self,
                 model: PyTorchModel,
                 optim: Optimizer,
                 criterion: _Loss):
        self.model = model
        self.optim = optim
        self.loss = criterion
        self._check_optim_net_aligned()

    def _check_optim_net_aligned(self):
        assert self.optim.param_groups[0]['params']\
        == list(self.model.parameters())

    def _generate_batches(self,
                          X: Tensor,
                          y: Tensor,
                          size: int = 32) -> Tuple[Tensor]:

        N = X.shape[0]

        for ii in range(0, N, size):
          X_batch, y_batch = X[ii:ii+size], y[ii:ii+size]

          yield X_batch, y_batch

    def fit(self, X_train: Tensor, y_train: Tensor,
            X_test: Tensor, y_test: Tensor,
            epochs: int=100,
            eval_every: int=10,
            batch_size: int=32):

        for e in range(epochs):
            X_train, y_train = permute_data(X_train, y_train)

            batch_generator = self._generate_batches(X_train, y_train,
                                                     batch_size)

            for ii, (X_batch, y_batch) in enumerate(batch_generator):
```

```
            self.optim.zero_grad()
            output = self.model(X_batch)
            loss = self.loss(output, y_batch)
            loss.backward()
            self.optim.step()

        output = self.model(X_test)
        loss = self.loss(output, y_test)
        print(e, loss)
```

Поскольку мы передаем `Model`, `Optimizer` и `Loss` в `Trainer`, то должны проверить, что параметры, на которые ссылается `Optimizer`, фактически совпадают с параметрами модели. Это делает функция `_check_optim_net_aligned`.

Теперь обучить модель проще простого:

```
net = HousePricesModel()
optimizer = optim.SGD(net.parameters(), lr=0.001)
criterion = nn.MSELoss()

trainer = PyTorchTrainer(net, optimizer, criterion)

trainer.fit(X_train, y_train, X_test, y_test,
            epochs=10,
            eval_every=1)
```

Этот код практически идентичен коду, который мы использовали для обучения моделей в структуре, созданной в первых трех главах. Неважно, что мы используем — PyTorch, TensorFlow или Theano, — основные шаги обучения модели глубокого обучения остаются неизменными!

Далее мы рассмотрим более продвинутые возможности PyTorch и покажем пару трюков для улучшения обучения, которое мы видели в главе 4.

Хитрости для оптимизации обучения в PyTorch

Из главы 4 мы знаем четыре таких хитрости:

- импульс;
- дропаут;

- инициализация веса;
- снижение скорости обучения.

В PyTorch это все легко реализовать. Например, чтобы включить в оптимизатор импульс, достаточно лишь добавить соответствующее слово в `SGD`, получая:

```
optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

Отсев данных тоже реализуется легко. В PyTorch есть встроенный модуль `nn.Linear (n_in, n_out)`, который вычисляет операции плотного слоя. Аналогично модуль `nn.Dropout (dropout_prob)` реализует операцию `Dropout`, с той лишь разницей, что в аргумент передается вероятность *исключения* (дропаута) нейрона, а не его сохранения, как это было в нашей реализации ранее.

Об инициализации весов думать вообще не надо: в большинстве операций `PyTorch` с параметрами, включая `nn.Linear`, веса автоматически масштабируются в зависимости от размера слоя.

Наконец, в `PyTorch` есть класс `lr_scheduler`, который можно использовать для снижения скорости обучения. Нужно выполнить импорт `from torch.optim import lr_scheduler`[1]. Теперь вы можете легко использовать эти методы в любом будущем проекте глубокого обучения, над которым вы работаете!

Сверточные нейронные сети в PyTorch

В главе 5 мы говорили о том, как работают сверточные нейронные сети, уделяя особое внимание операции многоканальной свертки. Мы видели, что операция преобразует пиксели входных изображений в слои нейронов, организованных в карты признаков, где каждый нейрон говорит, присутствует ли данный визуальный элемент (определенный в сверточном фильтре) в данном месте на изображении. Операция многоканальной свертки для двух входов и выходов имеет следующие формы:

---

[1] В репозитории GitHub книги, oreil.ly/301qxRk, вы можете найти пример кода, который реализует экспоненциальное снижение скорости обучения как часть PyTorch Trainer. Документацию по используемому там классу `ExponentialLR` можно найти на веб-сайте PyTorch (oreil.ly/2Mj9IhH).

- форма ввода данных `[batch_size, in_channels, image_height, image_width]`;
- форма ввода параметров `[in_channels, out_channels, filter_size, filter_size]`;
- форма вывода `[batch_size, out_channels, image_height, image_width]`.

Тогда операция многоканальной свертки в PyTorch:

```
nn.Conv2d (in_channels, out_channels, filter_size)
```

С этим определением можно легко обернуть `ConvLayer` вокруг этой операции:

```
class ConvLayer(PyTorchLayer):
    def __init__(self,
                 in_channels: int,
                 out_channels: int,
                 filter_size: int,
                 activation: nn.Module = None,
                 flatten: bool = False,
                 dropout: float = 1.0) -> None:
        super().__init__()

        # основная операция слоя
        self.conv = nn.Conv2d(in_channels, out_channels, filter_size,
                              padding=filter_size // 2)

        # те же операции "activation" и "flatten", что и ранее
        self.activation = activation
        self.flatten = flatten
        if dropout < 1.0:
            self.dropout = nn.Dropout(1 – dropout)

    def forward(self, x: Tensor) -> Tensor:

        # всегда выполняется операция свертки
        x = self.conv(x)

        # свертка выполняется опционально
        if self.activation:
            x = self.activation(x)
```

```
        if self.flatten:
            x = x.view(x.shape[0], x.shape[1] * x.shape[2] * x.shape[3])
        if hasattr(self, "dropout"):
            x = self.dropout(x)
        return x
```

В главе 5 мы автоматически добавляли выходные данные в зависимости от размера фильтра, чтобы размер выходного изображения соответствовал размеру входного изображения. PyTorch этого не делает. Чтобы добиться того же поведения, что и раньше, мы добавляем аргумент в параметр операции `nn.Conv2d` `padding` = `filter_size // 2`.

Теперь осталось лишь определить `PyTorchModel` с его операциями в функции `__init__` и последовательность операций в функции `forward`, после чего можно начать обучение. Далее следует простая архитектура, которую можно использовать в наборе данных `MNIST`, использующемся в главах 4 и 5. В ней есть:

- сверточный слой, который преобразует входной сигнал из 1 «канала» в 16 каналов;
- другой слой, который преобразует эти 16 каналов в 8 (каждый канал по-прежнему содержит 28 × 28 нейронов);
- два полносвязных слоя.

Схема нескольких сверточных слоев, за которыми следует меньшее количество полносвязных слоев, довольно обычна для сверточных архитектур. Здесь используется каждый по две штуки:

```
class MNIST_ConvNet(PyTorchModel):
    def __init__(self):
        super().__init__()
        self.conv1 = ConvLayer(1, 16, 5, activation=nn.Tanh(),
                               dropout=0.8)
        self.conv2 = ConvLayer(16, 8, 5, activation=nn.Tanh(),
                               flatten=True,
                               dropout=0.8)
        self.dense1 = DenseLayer(28 * 28 * 8, 32, activation=nn.Tanh(),
                                 dropout=0.8)
        self.dense2 = DenseLayer(32, 10)
```

```
    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)

        x = self.dense1(x)
        x = self.dense2(x)
        return x
```

Далее обучим эту модель так же, как обучали `HousePricesModel`:

```
model = MNIST_ConvNet ()
criterion = nn.CrossEntropyLoss ()
optimizer = optim.SGD (model.parameters (), lr = 0,01, momentum = 0,9)

trainer = PyTorchTrainer (model, optimizer, criterion)

trainer.fit (X_train, y_train,
             X_test, y_test,
             epochs = 5,
             eval_every = 1)
```

В отношении класса `nn.CrossEntropyLoss` есть важный момент. Напомним, что в нашей структуре из предыдущих глав класс `Loss` ожидал ввод той же формы, что и целевые данные. Для этого мы закодировали 10 различных целевых значений в данных `MNIST`, чтобы у каждой партии данных цель имела форму `[batch_size, 10]`.

В классе PyTorch `nn.CrossEntropyLoss`, который работает точно так же, как и предыдущий `SoftmaxCrossEntropyLoss`, этого делать не нужно. Эта функция потерь ожидает два объекта `Tensor`:

- `Tensor` предсказания размера `[batch_size, num_classes]`, похожий на наш класс `Softmax CrossEntropyLoss`.
- Целевой `Tensor` размера `[batch_size]` с различными значениями `num_classes`.

Таким образом, в предыдущем примере `y_train` — это массив размера `[60000]` (количество наблюдений в обучающем наборе `MNIST`), а `y_test` имеет размер `[10000]` (количество наблюдений в тестовом наборе).

Теперь, когда мы работаем с большими наборами данных, рассмотрим еще одну хорошую практику. Очевидно, что загружать все обучающие и тестовые наборы в память для обучения модели, как мы делаем с `X_train`, `y_train`, `X_test` и `y_test`, нерационально. У PyTorch есть способ обойти это: класс `DataLoader`.

Классы DataLoader и Transform

Напомним, что в модели MNIST из главы 2 мы слегка обрабатывали данные MNIST, вычитая из них глобальное среднее и деля на глобальное стандартное отклонение, чтобы «нормализовать» данные:

```
X_train, X_test = X_train - X_train.mean(), X_test - X_train.mean()
X_train, X_test = X_train / X_train.std(), X_test / X_train.std()
```

Для этого пришлось бы сначала полностью считать эти два массива в память. Но было бы намного эффективнее выполнить эту предварительную обработку «на лету», по мере поступления данных в нейронную сеть. В PyTorch есть встроенные функции, которые реализуют это и чаще всего используются с изображениями. Это преобразования через модуль `transforms` и `DataLoader` из `torch.utils.data`:

```
from torchvision.datasets import MNIST
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

Ранее мы считывали весь обучающий набор в `X_train` следующим образом:

```
mnist_trainset = MNIST(root="../data/", train=True)
X_train = mnist_trainset.train_data
```

Затем преобразовывали `X_train`, чтобы привести его к форме, подходящей для моделирования.

В PyTorch есть несколько удобных функций, которые позволяют нам выполнять множество преобразований для каждого пакета данных во время считывания. Это позволяет нам избежать считывания всего набора данных в память и использования преобразований PyTorch.

Сначала мы определяем список преобразований, которые необходимо выполнять для каждой партии считываемых данных. Например, следующие действия преобразуют каждое изображение MNIST в `Tensor` (большинство

наборов данных PyTorch по умолчанию являются «изображениями PIL», поэтому функция `transforms.ToTensor()` используется в первую очередь), а затем «нормализуют» набор данных — вычитая среднее значение, а затем деля на стандартное отклонение, используя общее среднее значение `MNIST` и стандартное отклонение 0.1305 и 0.3081 соответственно:

```
img_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1305,), (0.3081,))
])
```

Нормализация фактически вычитает среднее значение и стандартное отклонение *из каждого канала* входного изображения. Таким образом, при работе с цветными изображениями с тремя входными каналами используется преобразование `Normalize`, у которого два кортежа по три числа в каждом, например `transforms.Normalize ((0.1, 0.3, 0.6), (0.4, 0.2, 0.5))`. `DataLoader` из этого получает следующие задачи:

- Нормализовать первый канал, используя среднее значение 0.1 и стандартное отклонение 0.4.
- Нормализовать второй канал, используя среднее значение 0.3 и стандартное отклонение 0.2.
- Нормализовать третий канал, используя среднее значение 0.6 и стандартное отклонение 0.5.

После применения этих преобразований мы применяем их к набору данных, считывая их в пакетном режиме:

```
dataset = MNIST("../mnist_data/", transform=img_transforms)
```

Наконец, мы можем определить `DataLoader`, который принимает этот набор данных и определяет правила для последовательной генерации пакетов данных:

```
dataloader = DataLoader(dataset, batch_size=60, shuffle=True)
```

Затем мы можем модифицировать класс `Trainer`, чтобы использовать загрузчик данных для генерации пакетов, используемых для обучения

сети, вместо загрузки всего набора данных в память, а затем генерировать их вручную, используя функцию `batch_generator`, как мы делали раньше. На сайте книги (https://oreil.ly/2N4H8jz)[1] приведен пример обучения сверточной нейронной сети с использованием таких `DataLoaders`. В классе `Trainer` достаточно лишь заменить строку:

```
for X_batch, y_batch in enumerate(batch_generator):
```

на:

```
for X_batch, y_batch in enumerate(train_dataloader):
```

Кроме того, вместо того чтобы вводить весь обучающий набор в функцию подгонки, мы передаем `DataLoaders`:

```
trainer.fit(train_dataloader = train_loader,
            test_dataloader = test_loader,
            epochs=1,
            eval_every=1)
```

Используя эту архитектуру и вызывая метод `fit`, как мы только что сделали, получаем около 97% точности в MNIST после одной эпохи. Однако более важным, чем точность, является то, что вы видели, как реализовать концепции, рассмотренные нами из первых принципов, в высокопроизводительной среде. Теперь, когда вы понимаете как базовые концепции, так и структуру, я призываю вас изменить код в репозитории GitHub книги (https://oreil.ly/2N4H8jz) и попробовать другие сверточные архитектуры, другие наборы данных и т. д.

CNN были одной из двух продвинутых архитектур, которые мы рассмотрели ранее в книге; давайте теперь посмотрим, как реализовать самый продвинутый вариант RNN, который мы рассмотрели, LSTM, в PyTorch.

LSTM в PyTorch

В предыдущей главе мы рассмотрели, как создать LSTM с нуля. Наш слой `LSTMLayer` получал входной массив размера `[batch_size, sequence_length, feature_size]` и выводил `ndarray` размера `[batch_size, sequence_length, feature_size]`. Кроме того, каждый слой получал скрытое состояние и состояние ячейки, каждое из которых изначально имело форму

[1] См. раздел «Создание CNN с помощью PyTorch».

[1, hidden_size], затем расширялось до [batch_size, hidden_size] при передаче пакета, а затем сворачивалось обратно до [1, hidden_size] после завершения итерации.

Исходя из этого, метод __init__ для нашего LSTMLayer будет выглядеть следующим образом:

```
class LSTMLayer(PyTorchLayer):
    def __init__(self,
                 sequence_length: int,
                 input_size: int,
                 hidden_size: int,
                 output_size: int) -> None:
        super().__init__()
        self.hidden_size = hidden_size
        self.h_init = torch.zeros((1, hidden_size))
        self.c_init = torch.zeros((1, hidden_size))
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = DenseLayer(hidden_size, output_size)
```

Как и в случае со сверточными слоями, в PyTorch есть операция nn.lstm для реализации LSTM. Обратите внимание, что в нашем LSTMLayer мы сохраняли DenseLayer в атрибуте self.fc. Вы можете вспомнить из предыдущей главы, что последний шаг ячейки LSTM — это установка последнего скрытого состояния с помощью операций слоя Dense (умножение веса и добавление смещения), чтобы преобразовать скрытое состояние в размерность out put_size для каждой операции. PyTorch работает немного иначе: операция nn.lstm просто выводит скрытые состояния для каждого временного шага. Таким образом, чтобы позволить нашему LSTMLayer выводить измерение, отличное от его входных данных (а нам так и надо), мы добавляем DenseLayer в конце, чтобы преобразовать скрытое состояние в измерение output_size.

В этой модификации функция forward становится похожа на функцию LSTMLayer из главы 6:

```
def forward(self, x: Tensor) -> Tensor:

    batch_size = x.shape[0]

    h_layer = self._transform_hidden_batch(self.h_init,
                                           batch_size,
```

```
                                        before_layer=True)
    c_layer = self._transform_hidden_batch(self.c_init,
                                        batch_size,
                                        before_layer=True)

    x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))

    self.h_init, self.c_init = (
       self._transform_hidden_batch(h_out,
                                    batch_size,
                                    before_layer=False).detach(),
       self._transform_hidden_batch(c_out,
                                    batch_size,
                                    before_layer=False).detach()
                                   )
    x = self.fc(x)

    return x
```

Ключевая строка здесь, которая должна выглядеть знакомо с учетом реализации LSTM в главе 6, выглядит так:

```
x, (h_out, c_out) = self.lstm(x, (h_layer, c_layer))
```

Помимо этого, мы изменяем форму скрытого состояния и состояния ячейки до и после функции `self.lstm` с помощью вспомогательной функции `self._transform_hidden_batch`. Полная функция приведена в репозитории GitHub книги (https://oreil.ly/2N4H8jz).

Наконец, обернем модель:

```
class NextCharacterModel(PyTorchModel):
    def __init__(self,
                 vocab_size: int,
                 hidden_size: int = 256,
                 sequence_length: int = 25):
        super().__init__()
        self.vocab_size = vocab_size
        self.sequence_length = sequence_length

        # В этой модели всего один слой,
        # с одинаковыми размерностями входа и выхода
        self.lstm = LSTMLayer(self.sequence_length,
```

```
                            self.vocab_size,
                            hidden_size,
                            self.vocab_size)

    def forward(self,
                inputs: Tensor):
        assert_dim(inputs, 3) # batch_size, sequence_length, vocab_size

        out = self.lstm(inputs)

        return out.permute(0, 2, 1)
```

Функция `nn.CrossEntropyLoss` ожидает, что первые две размерности будут равны `batch_size` и распределению по классам. Однако в нашей реализации LSTM есть распределение по классам, когда последнее измерение (`vocab_size`) выходит из `LSTMLayer`. Поэтому, чтобы подготовить окончательный вывод модели для передачи в потерю, мы перемещаем измерение, содержащее распределение по буквам, во второе измерение, используя функцию `out.permute (0, 2, 1)`.

В репозитории GitHub книги (https://oreil.ly/2N4H8jz) я покажу, как написать класс `LSTMTrainer` для наследования от `PyTorchTrainer` и использования его для обучения `NextCharacterModel` для генерации текста. Мы используем ту же предварительную обработку текста, что и в главе 6: выбор последовательностей текста, горячее кодирование букв и группирование последовательностей горячих кодированных букв в пакеты.

Теперь вы знаете, как реализовать три рассмотренные нами архитектуры нейронных сетей (полносвязные, сверточные и рекуррентные) в PyTorch. В заключение мы кратко рассмотрим, как можно использовать нейронные сети для обучения без учителя.

P. S. Обучение без учителя через автокодировщик

В этой книге мы говорили о том, как используются модели глубокого обучения для решения задач обучения с учителем. Но есть и другая сторона машинного обучения: обучение без учителя. Она подразумевает так называемое «нахождение структуры в неразмеченных данных». Предпочитаю

считать это поиском взаимосвязей между признаками данных, которые еще не были измерены. В противовес этому, обучение с учителем — это поиск взаимосвязей между ранее измеренными признаками в данных.

Предположим, у вас был набор неразмеченных данных изображений. Вы ничего не знаете об этих изображениях — например, вы не уверены, существует ли 10 различных цифр, или 5, или 20 (эти изображения могут быть из какого-то древнего алфавита), — и вы хотите знать ответы на такие вопросы, как:

- Сколько есть разных цифр?
- Какие цифры внешне похожи друг на друга?
- Существуют ли «лишние» изображения, которые явно отличаются от других изображений?

Чтобы понять, как глубокое обучение может помочь в этом, остановимся и подумаем, а что же пытаются сделать модели глубокого обучения.

Обучение представлениям

Мы уже видели, что модели глубокого обучения могут научиться делать точные прогнозы. Они делают это путем преобразования входных данных, которые получают, в представления, которые постепенно становятся более абстрактными и более настроенными на решение любой задачи. В частности, последний слой сети, непосредственно перед слоем с самими предсказаниями (который будет иметь только один нейрон для задачи регрессии и `num_classes` нейронов для задачи классификации), является попыткой сети создать свое представление входных данных. Это максимально полезно для задачи прогнозирования (рис. 7.1).

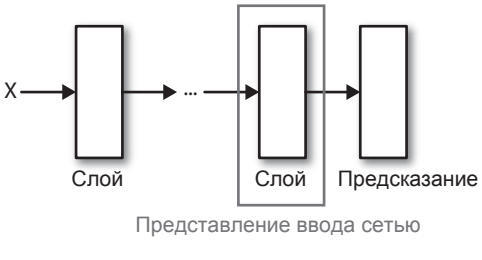


Рис. 7.1. Последний слой нейронной сети, непосредственно перед предсказаниями, содержит представление входных данных с точки зрения сети, которое кажется ей наиболее подходящим

После обучения модель может не только делать прогнозы для новых точек данных, *но и генерировать представления этих точек данных*. Затем их можно будет использовать для кластеризации, анализа сходства или обнаружения лишних данных (в дополнение к прогнозированию).

Не спешим вешать ярлыки

Ограничение этого подхода состоит в том, что для обучения модели нужна какая-то классификация, метки. А как научить модель генерировать «полезные» представления без каких-либо меток? Если у нас нет меток, придется сгенерировать представления наших данных с помощью того, что у нас есть: самих данных. Эта идея лежит в основе класса нейросетевых архитектур, известных как автокодировщики, которые включают обучение нейронных сетей для восстановления обучающих данных, заставляя сеть изучать представление каждой точки данных, наиболее полезной для этой реконструкции.

Визуализация

На рис. 7.2 показан общий принцип работы автокодировщика:

1. Один набор слоев преобразует данные в сжатое представление данных.
2. Другой набор слоев преобразует это представление в выходные данные того же размера и формы, что и исходные данные.

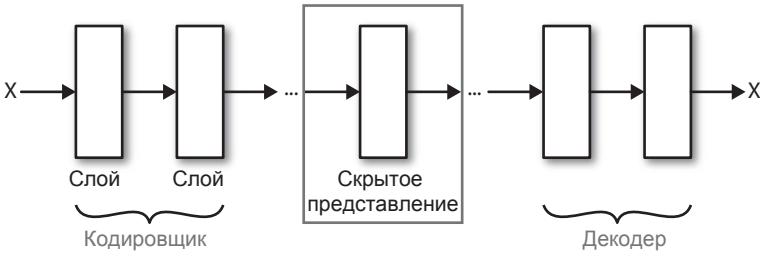


Рис. 7.2. У автокодировщика есть один набор слоев (который можно рассматривать как сеть «кодировщика»), который отображает входные данные в более низкоразмерное представление, и другой набор слоев (который можно рассматривать как сеть «декодера»), который отображает низкоразмерное представление обратно на вход; эта структура заставляет сеть изучать низкоразмерное представление, которое наиболее полезно для восстановления входных данных

Реализация такой архитектуры иллюстрирует некоторые особенности PyTorch, которые мы еще не имели возможности представить.

Реализация автокодировщика в PyTorch

Теперь мы покажем простой автокодировщик, который принимает входное изображение, пропускает его через два сверточных слоя, затем через плотный слой для генерации представления, а затем передает это представление обратно через плотный слой и два сверточных слоя для генерации выхода того же размера, что и вход. На этом примере я покажу две распространенные практики при реализации более сложных архитектур в PyTorch. Мы можем включить `PyTorchModels` в качестве атрибутов другой `PyTorchModel`, так же как ранее определили слои `PyTorchLayer` в качестве атрибутов таких моделей. В следующем примере мы реализуем наш автокодировщик как две `PyTorchModel` в качестве атрибутов: кодировщик и декодер. Как только мы обучим модель, то сможем использовать обученный кодировщик в качестве своей собственной модели для генерации представлений.

Определение кодировщика:

```
class Encoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Encoder, self).__init__()
        self.conv1 = ConvLayer(1, 14, activation=nn.Tanh())
        self.conv2 = ConvLayer(14, 7, activation=nn.Tanh(),
                               flatten=True)
        self.dense1 = DenseLayer(7 * 28 * 28, hidden_dim,
                                 activation=nn.Tanh())

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        x = self.conv1(x)
        x = self.conv2(x)
        x = self.dense1(x)
        return x
```

Определение декодера:

```
class Decoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Decoder, self).__init__()
        self.dense1 = DenseLayer(hidden_dim, 7 * 28 * 28,
                                 activation=nn.Tanh())

        self.conv1 = ConvLayer(7, 14, activation=nn.Tanh())
        self.conv2 = ConvLayer(14, 1, activation=nn.Tanh())

    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 2)

        x = self.dense1(x)

        x = x.view(-1, 7, 28, 28)
        x = self.conv1(x)
        x = self.conv2(x)

        return x
```

Если бы мы использовали шаг больше единицы, то просто не смогли бы использовать обычную свертку для преобразования кодирования в вывод. Вместо этого пришлось бы использовать *транспонированную свертку*, где размер изображения окажется больше, чем размер изображения на входе. Смотрите операцию `nn.ConvTrans pose2d` в документации PyTorch для получения дополнительной информации (https://oreil.ly/306qiV7).

Тогда сам автокодировщик может выглядеть так:

```
class Autoencoder(PyTorchModel):
    def __init__(self,
                 hidden_dim: int = 28):
        super(Autoencoder, self).__init__()

        self.encoder = Encoder(hidden_dim)

        self.decoder = Decoder(hidden_dim)
```

```
    def forward(self, x: Tensor) -> Tensor:
        assert_dim(x, 4)

        encoding = self.encoder(x)
        x = self.decoder(encoding)

        return x, encoding
```

Метод `forward` в классе `Autoencoder` иллюстрирует вторую распространенную практику в PyTorch: поскольку в конечном итоге мы захотим увидеть скрытое представление, создаваемое моделью, метод `forward` возвращает два элемента: кодирование, а также вывод, который будет использоваться для обучения сети.

Разумеется, придется видоизменить класс `Trainer`, чтобы приспособиться к этому методу. В частности, `PyTorchModel` в своем текущем виде выводит только один `Tensor` из своего метода `forward`. Как выясняется, несложно и полезно научить его выводить `Tuple of Tensors` по умолчанию, даже если этот `Tuple` имеет размер 1. Это позволит легко писать модели вроде `Autoencoder`. Нужно сделать всего три маленькие вещи. Во-первых, сделать сигнатуру функции метода `forward` нашего базового класса `PyTorchModel`:

```
def forward(self, x: Tensor) -> Tuple[Tensor]:
```

Затем в конце метода `forward` любой модели, которая наследуется от базового класса `PyTorch Model`, мы напишем `return x` вместо `return x`, как мы делали раньше.

Во-вторых, изменим класс `Trainer`, чтобы он всегда брал в качестве результата первый элемент того, что возвращает модель:

```
output = self.model(X_batch)[0]
...
output = self.model(X_test)[0]
```

Есть еще одна примечательная особенность модели `Autoencoder`: мы применяем функцию активации `Tanh` к последнему слою, что означает, что выходные данные модели будут лежать в диапазоне от –1 до 1. В любой модели выходные данные модели должны иметь тот же масштаб, что и целевые данные, а тут наши данные и являются целью. Таким образом, нужно масштабировать ввод в диапазоне от –1 до 1, как в следующем коде:

```
X_train_auto = (X_train - X_train.min())
               / (X_train.max() - X_train.min()) * 2 - 1
X_test_auto = (X_test - X_train.min())
               / (X_train.max() - X_train.min()) * 2 - 1
```

И в-третьих, можем обучить нашу модель, используя тренировочный код, который к настоящему времени должен выглядеть знакомым (в качестве размерности вывода кодирования наугад зададим 28):

```
model = Autoencoder(hidden_dim=28)
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

trainer = PyTorchTrainer(model, optimizer, criterion)

trainer.fit(X_train_auto, X_train_auto,
            X_test_auto, X_test_auto,
            epochs=1,
            batch_size=60)
```

Запустив этот код и обучив модель, мы можем посмотреть как на восстановленные изображения, так и на представления изображений, просто пропустив `X_test_auto` через модель (поскольку прямой метод был определен для возврата двух величин):

```
reconstructed_images, image_representations = model(X_test_auto)
```

Каждый элемент `reconstructed_images` — это `Tensor [1, 28, 28]`, который представляет собой лучшую попытку нейронной сети восстановить соответствующее исходное изображение после прохождения его через архитектуру автокодировщика, пропустившего изображение через слой с более низкой размерностью. На рис. 7.3 показано случайно выбранное восстановленное изображение и исходное изображение.

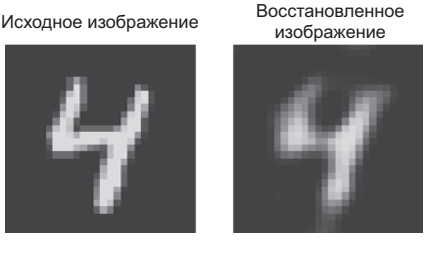


Рис. 7.3. Изображение из тестового набора MNIST вместе с восстановленным после его пропуска через автокодировщик

Визуально изображения выглядят одинаково, а это значит, что нейронная сеть действительно взяла исходные изображения размером 784 пикселя и сопоставила их с пространством более низкой размерности, а именно 28, и большая часть информации о 784-пиксельном изображении закодирована в векторе длиной 28. Как нам исследовать весь набор данных, чтобы увидеть, действительно ли нейронная сеть изучила структуру данных изображения без каких-либо меток? «Структура данных» здесь означает, что лежащие в основе данные на самом деле представляют собой изображения из 10 различных рукописных цифр. Таким образом, изображения, близкие к данному изображению в новом 28-мерном пространстве, в идеале должны иметь одну и ту же цифру или, по крайней мере, визуально быть очень похожими, поскольку мы, люди, различаем изображения именно по визуальному сходству. Можно проверить, так ли это, применив технику уменьшения размерности, изобретенную Лоренсом ван дер Маатеном в бытность его аспирантом у Джеффри Хинтона (одного из «отцов-основателей» нейронных сетей): t-распределенное стохастическое вложение соседей, или t-SNE. Метод t-SNE выполняет уменьшение размерности способом, аналогичным обучению нейронных сетей: начинает с начального представления нижнего измерения, а затем обновляет его, так что со временем оно приближается к решению со свойством, которое указывает, что «близко друг к другу» в многомерном пространстве означает «близко друг к другу» в низкоразмерном пространстве, и наоборот[1].

Сделаем следующее:

- Пропустим 10 000 изображений через t-SNE и уменьшим размерность до 2.
- Визуализируем получающееся двумерное пространство, окрашивая его различные точки реальной меткой (которую не видел автокодировщик).

На рис. 7.4 показан результат.

Получается, что изображения каждой цифры в основном сгруппированы в отдельном кластере; это показывает, что обучение нашей архитектуры автокодировщика позволило ей восстанавливать исходные изображения из низкоразмерного представления и в целом выявить общую структуру

[1] Оригинальная статья 2008 года — *Visualizing Data using t-SNE* Лоренса ван дер Маатена и Джеффри Хинтона (oreil.ly/2KIAaOt).

Рис. 7.4. Результат запуска t-SNE на 28-мерном пространстве автокодировщика

и закономерности у изображений без каких-либо меток[1]. Более того, помимо отдельных кластеров для всех 10 цифр мы видим, что визуально похожие цифры также расположены ближе друг к другу: сверху и чуть правее у нас есть группы цифр 3, 5 и 8, внизу рядом 4 и 9, и недалеко от

[1] Кроме того, это произошло без особых усилий: архитектура здесь очень проста, и мы не используем никаких хитростей, например снижения скорости обучения, поскольку обучение выполняется всего за одну эпоху. Это показывает, что основная идея использования подобной автокодировщику архитектуры для изучения структуры набора данных без меток в целом хороша и просто не сработала в данном конкретном случае.

них 7. Наконец, самые разные цифры — 0, 1 и 6 — образуют самые удаленные кластеры.

Более сильный тест и решение для обучения без учителя

Сейчас мы проверили, смогла ли модель выявить базовую структуру пространства входных изображений. Уже не должен удивлять тот факт, что сверточная нейронная сеть может изучать представления изображений цифр, наделяя при этом визуально похожие изображения похожими представлениями. Возьмемся за задачу покруче: попробуем проверить, обнаружит ли нейронная сеть «гладкое» базовое пространство: пространство, в котором любой вектор длиной 28, а не только векторы, полученные в результате передачи реальных цифр через сеть кодировщика, может быть сопоставлен с реалистично выглядящей цифрой. Оказывается, наш автокодировщик не может этого сделать. На рис. 7.5 показан результат генерации пяти случайных векторов длиной 28 и пропуска их через сеть декодера с использованием того факта, что автокодировщик содержал декодер в качестве атрибута:

```
test_encodings = np.random.uniform(low=-1.0, high=1.0, size=(5, 28))
test_imgs = model.decoder(Tensor(test_encodings))
```

Рис. 7.5. Результат пропуска пяти случайно сгенерированных векторов через декодер

Видно, что получающиеся изображения не похожи на цифры. Получается, что хотя автокодировщик и способен разумно отобразить наши данные в пространстве меньшего размера, он, по-видимому, не в состоянии освоить «гладкое» пространство.

Решение проблемы обучения нейронной сети представлению изображений в обучающем наборе в «гладком» базовом пространстве является одним из главных достижений порождающих состязательных сетей (generative adversarial networks, GAN). Изобретенные в 2014 году, GAN

наиболее широко известны тем, что позволяют нейронным сетям генерировать реалистичные изображения с помощью процедуры обучения, в которой две нейронные сети обучаются одновременно. В 2015 году GAN серьезно продвинулись, когда исследователи использовали их с глубоко сверточной архитектурой в обеих сетях не только для создания реалистично выглядящих 64 × 64 цветных изображений спален, но и для генерации большой выборки указанных изображений из случайно сгенерированных 100-размерных векторов[1]. Это признак того, что нейронные сети действительно изучили базовое представление «пространства» заданных немаркированных изображений. GAN заслуживают отдельной книги, поэтому подробно на них останавливаться я не буду.

Заключение

Теперь у вас есть глубокое понимание механики самых популярных передовых архитектур глубокого обучения, а также того, как реализовать эти архитектуры в одной из самых популярных высокопроизводительных сред глубокого обучения. Чтобы использовать модели глубокого обучения для решения реальных проблем, нужно лишь одно — практика. Вам предстоит (без особых проблем) читать чужой код и быстро осваивать детали и приемы реализации, заставляющие определенные архитектуры моделей работать над конкретными проблемами. Список рекомендуемых шагов приведен в репозитории GitHub книги (https://oreil.ly/2N4H8jz). Вперед!

[1] Ознакомьтесь с документацией DCGAN *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks* Алека Рэдфорда и др. (arxiv.org/abs/1511.06434), а также с документацией по PyTorch (oreil.ly/2TEspgG).

ПРИЛОЖЕНИЕ А

Глубокое погружение

Рассмотрим подробно некоторые технические области. Это важная для понимания информация, но необязательная.

Цепное правило

Сначала поясним, почему мы можем заменить W^T в выражении $\frac{\partial v}{\partial u}(X)$ по цепному правилу из главы 1.

Известно, что L означает следующее:

$$\sigma(XW_{11}) + \sigma(XW_{12}) + \sigma(XW_{21}) + \sigma(XW_{31}) + \sigma(XW_{32}).$$

а это лишь сокращение:

$$\sigma(XW_{11}) = \sigma(x_{11} \times w_{11} + x_{12} \times w_{21} + x_{13} \times w_{31}),$$
$$\sigma(XW_{12}) = \sigma(x_{11} \times w_{12} + x_{12} \times w_{22} + x_{13} \times w_{32})$$

и так далее. Давайте распишем одно из этих выражений. Как бы это выглядело, если бы мы взяли частную производную, скажем, $\sigma(XW_{11})$ по каждому элементу X (что, в конечном счете, мы и хотим сделать со всеми шестью компонентами L)?

Поскольку

$$\sigma(XW_{11}) = \sigma(x_{11} \times w_{11} + x_{12} \times w_{21} + x_{13} \times w_{31}),$$

нетрудно понять, что частная производная этого по x_1 благодаря очень простому применению цепного правила имеет вид:

$$\frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{11}.$$

Поскольку единственное, на что умножается x_{11} в выражении XW_{11}, это w_{11}, частная производная по отношению ко всему остальному равна 0.

Итак, вычисляя частную производную $\sigma(XW_{11})$ по всем элементам X, получим следующее общее выражение для $\frac{\partial\sigma(XW_{11})}{\partial X}$:

$$\frac{\partial\sigma(XW_{11})}{\partial X}=\begin{bmatrix}\frac{\partial\sigma}{\partial u}(XW_{11})\times w_{11} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{21} & \frac{\partial\sigma}{\partial u}(XW_{11})\times w_{31}\\ 0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}.$$

Аналогично мы можем, например, определить частную производную от $\sigma(XW_{32})$ по каждому элементу X:

$$\begin{bmatrix} 0 & 0 & 0\\ 0 & 0 & 0\\ \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{12} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{22} & \frac{\partial\sigma}{\partial u}(XW_{32})\times w_{32}\end{bmatrix}.$$

Теперь у нас есть все компоненты для вычисления $\frac{\partial\Lambda}{\partial X}(S)$. Мы можем просто вычислить шесть матриц той же формы, что и предыдущие матрицы, и сложить результаты вместе.

Обратите внимание, что математика стала запутанной, но не сложной. Вы можете пропустить следующие вычисления и сразу перейти к заключению — там одно простое выражение. Но работа с выражениями даст вам большее понимание того, насколько удивительно прост вывод. А что есть жизнь, как не радость от чего-то классного?

Здесь всего два шага. Во-первых, напишем, что $\frac{\partial\Lambda}{\partial X}(S)$ есть сумма шести только что описанных матриц:

$$\frac{\partial \Lambda}{\partial X}(S) = \frac{\partial \sigma(XW_{11})}{\partial X} + \frac{\partial \sigma(XW_{12})}{\partial X} + \frac{\partial \sigma(XW_{21})}{\partial X} + \frac{\partial \sigma(XW_{22})}{\partial X} + \frac{\partial \sigma(XW_{31})}{\partial X} + \frac{\partial \sigma(XW_{32})}{\partial X} =$$

$$= \begin{bmatrix}
\frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{11} & \frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{21} & \frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{31} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{12} & \frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{22} & \frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{32} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{11} & \frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{21} & \frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{31} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{12} & \frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{22} & \frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{32} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{11} & \frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{21} & \frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{31} \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{12} & \frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{22} & \frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{32}
\end{bmatrix}.$$

Теперь объединим эту сумму в одну большую матрицу. Эта матрица поначалу не будет иметь какой-либо интуитивно понятной формы, но на самом деле она является результатом вычисления предыдущей суммы:

$$\frac{\partial \Lambda}{\partial X}(S) =$$

$$= \begin{bmatrix}
\frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{11} + \frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{12} + \frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{21} + \frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{22} + \frac{\partial \sigma}{\partial u}(XW_{11}) \times w_{31} + \frac{\partial \sigma}{\partial u}(XW_{12}) \times w_{32} \\
\frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{11} + \frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{12} + \frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{21} + \frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{22} + \frac{\partial \sigma}{\partial u}(XW_{21}) \times w_{31} + \frac{\partial \sigma}{\partial u}(XW_{22}) \times w_{32} \\
\frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{11} + \frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{12} + \frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{21} + \frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{22} + \frac{\partial \sigma}{\partial u}(XW_{31}) \times w_{31} + \frac{\partial \sigma}{\partial u}(XW_{32}) \times w_{32}
\end{bmatrix}.$$

Теперь самое интересное. Напомним, что:

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}.$$

При этом W спрятано в предыдущей матрице — оно просто транспонировано. Напомним, что:

$$\frac{\partial \Lambda}{\partial u}(S) = \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix}.$$

Получается, что предыдущая матрица эквивалентна следующему:

$$\frac{\partial \Lambda}{\partial u}(X) = \begin{bmatrix} \frac{\partial \sigma}{\partial u}(XW_{11}) & \frac{\partial \sigma}{\partial u}(XW_{12}) \\ \frac{\partial \sigma}{\partial u}(XW_{21}) & \frac{\partial \sigma}{\partial u}(XW_{22}) \\ \frac{\partial \sigma}{\partial u}(XW_{31}) & \frac{\partial \sigma}{\partial u}(XW_{32}) \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix} = \frac{\partial \Lambda}{\partial u}(S) \times W^T.$$

Далее, помните, что мы искали что-то, чтобы заменить вопросительный знак в следующем уравнении:

$$\frac{\partial \Lambda}{\partial u}(X) = \frac{\partial \Lambda}{\partial u}(S) \times ?$$

Что ж, выходит, что это W. Результат сам идет в руки. Также обратите внимание, что это тот же самый результат, который мы уже видели ранее. Опять же, это объясняет, почему глубокое обучение работает, и позволяет правильно его реализовать. Означает ли это, что мы можем заменить знак вопроса в предыдущем уравнении и сказать, что $\frac{\partial v}{\partial X}(X, W) = W^T$? Нет, не совсем. Но если перемножить два входа (X и W), чтобы получить результат N, и пропустить эти входы через некоторую нелинейную функцию σ, чтобы в итоге получить выход S, можно сказать следующее:

$$\frac{\partial v}{\partial X}(X, W) = \frac{\partial \sigma}{\partial u}(N) \times W^T.$$

Этот математический факт позволяет эффективно вычислять и выражать обновления градиента, используя обозначение умножения матриц. Кроме того, аналогичными рассуждениями можно понять, что:

$$\frac{\partial \sigma}{\partial W}(X, W) = X^T \times \frac{\partial \sigma}{\partial u}(N).$$

Градиент потерь с учетом смещения

Далее мы подробно рассмотрим, почему при вычислении производной потери по смещению в полносвязной нейронной сети мы суммируем по `axis=0`.

Добавление смещения в нейронную сеть происходит в следующем контексте: у нас есть пакет данных, представленный матрицей измерения `n` строк (размер пакета) на `f` столбцов (количество объектов), и мы прибавляем число к каждой из функций. Например, в примере нейронной сети из главы 2 у нас есть 13 функций, а смещение B имеет 13 чисел; первое число будет добавлено к каждой строке в первом столбце `M1 = np.dot(X, weights[W1])`, второе число будет добавлено к каждой строке во втором столбце и т. д. Далее, в сети $B2$ будет содержать одно число, которое будет просто добавлено к каждой строке в одном столбце `M2`. Таким образом, поскольку в каждую строку матрицы будут добавлены одинаковые числа, на обратном проходе нам нужно сложить градиенты вдоль измерения, представляющего строки, к которым был добавлен каждый элемент смещения. Вот почему мы суммируем выражения для `dLdB1` и `dLdB2` вдоль `axis=0`, например `dLdB1 = (dLdN1 × dN1dB1).sum(axis = 0)`. На рис. А.1 приведено визуальное объяснение с некоторыми комментариями.

Свертка с помощью умножения матриц

Наконец, мы покажем, как выразить пакетную операцию многоканальной свертки в контексте пакетного умножения матриц, чтобы эффективно реализовать ее в NumPy. Чтобы понять, как работает свертка, рассмотрим, что происходит на прямом проходе полносвязной нейронной сети:

- Получаем ввод размера `[batch_size, in_features]`.
- Умножаем его на параметр размера `[in_features, out_features]`.
- Получаем результирующий вывод размера `[batch_size, out_features]`.

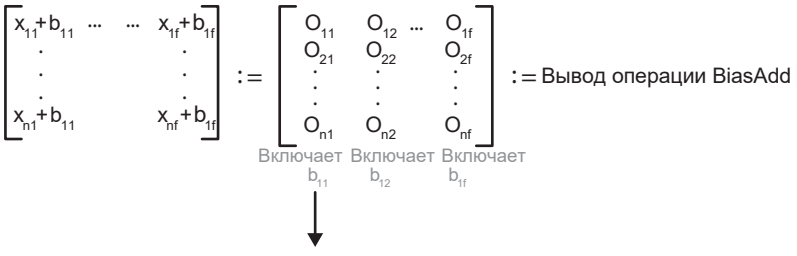


Рис. А.1. Краткое изложение того, почему вычисление производной выходного сигнала полносвязного слоя по смещению включает суммирование по `axis = 0`

В сверточном слое наоборот:

- Получаем ввод размера `[batch_size, in_channels, img_height, img_width]`.
- Сворачиваем его с параметром размера `[in_channels, out_channels, param_height, param_width]`.
- Получаем результирующий вывод размера `[batch_size, in_channels, img_height, img_width]`.

Ключ к тому, чтобы операция свертки выглядела более похожей на операцию передачи вперед, состоит в том, чтобы сначала извлечь «фрагменты изображения» размером `img_height` × `img_width` из каждого канала входного изображения. Как только эти фрагменты будут извлечены, входные данные могут быть изменены, так что операция свертки может быть выражена как умножение пакетной матрицы с использованием функции Num.Py `np.matmul`. Сначала:

```
def _get_image_patches(imgs_batch: ndarray,
                       fil_size: int):
    '''
    imgs_batch: [batch_size, channels, img_width, img_height]
    fil_size: int
    '''
    # подача изображений
    imgs_batch_pad = np.stack([_pad_2d_channel(obs, fil_size // 2)
                               for obs in imgs_batch])
    patches = []
    img_height = imgs_batch_pad.shape[2]
```

```
    # для каждого места на изображении...
    for h in range(img_height-fil_size+1):
        for w in range(img_height-fil_size+1):

            # ...берем фрагмент размером [fil_size, fil_size]
            patch = imgs_batch_pad[:, :, h:h+fil_size, w:w+fil_size]
            patches.append(patch)

    # Получение вывода размером
    # [img_height * img_width, batch_size, n_channels, fil_size,
    # fil_size]
    return np.stack(patches)
```

Затем можно вычислить вывод операции свертки следующим образом:

1. Взять фрагменты изображения размером [batch_size, in_channels, img_height x img_width, filter_size, filter_size].
2. Заменить их на [batch_size, img_height × img_width, in_channels × filter_size × filter_size].
3. Параметр формы должен быть [in_channels × filter_size × filter_size, out_channels].
4. После умножения пакетной матрицы результатом будет [batch_size, img_height × img_width, out_channels].
5. Заменить на [batch_size, out_channels, img_height, img_width].

```
def _output_matmul(input_: ndarray,
                   param: ndarray) -> ndarray:
    '''
    conv_in: [batch_size, in_channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    '''

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_height = input_.shape[2]
    patch_size = param.shape[0] * param.shape[2] * param.shape[3]

    patches = _get_image_patches(input_, param_size)

    patches_reshaped = (
        patches
```

```
        .transpose(1, 0, 2, 3, 4)
        .reshape(batch_size, img_height * img_height, -1)
        )

    param_reshaped = param.transpose(0, 2, 3, 1).reshape(patch_size, -1)

    output = np.matmul(patches_reshaped, param_reshaped)

    output_reshaped = (
        output
        .reshape(batch_size, img_height, img_height, -1)
        .transpose(0, 3, 1, 2)
    )

    return output_reshaped
```

Это был прямой проход. Для обратного прохода нужно рассчитать как градиент параметра, так и входные градиенты. Опять же, в качестве основы можно использовать способ для полносвязной нейронной сети. Начиная с градиента параметра градиент полносвязной нейронной сети:

```
np.matmul(self.inputs.transpose(1, 0), output_grad)
```

Это позволяет понять, как реализуется обратный проход через операцию свертки: здесь форма ввода — `[batch_size, in_channels, img_height, img_width]`, а полученный *градиент* вывода — `[batch_size, out_channels, img_height, img_width]`. Учитывая, что форма параметра имеет вид `[in_channels, out_channels, param_height, param_width]`, мы можем выполнить это преобразование с помощью следующих шагов:

1. Извлечь фрагменты изображения из входного изображения, что приведет к тому же результату, что и в прошлый раз, в форме `[batch_size, in_channels, img_height x img_width, filter_size, filter_size]`.
2. Используя умножение из полносвязного случая в качестве мотивации, изменить его, чтобы оно имело форму `[in_channels × param_height × param_width, batch_size × img_height × img_width]`.
3. Изменить форму выходных данных — из `[batch_size, out_channels, img_height, img_width]` на `[batch_size × img_height × img_width, out_channels]`.
4. Перемножить их, чтобы получить результат в форме `[in_channels × param_height × param_width, out_channels]`.

5. Изменить форму, чтобы получить окончательный градиент параметра формы [in_channels, out_channels, param_height, param_width].

Этот процесс реализован следующим образом:

```
def _param_grad_matmul(input_: ndarray,
                       param: ndarray,
                       output_grad: ndarray):
    '''
    input_: [batch_size, in_channels, img_width, img_height]
    param: [in_channels, out_channels, fil_width, fil_height]
    output_grad: [batch_size, out_channels, img_width, img_height]
    '''

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_size = input_.shape[2] ** 2
    in_channels = input_.shape[1]
    out_channels = output_grad.shape[1]
    patch_size = param.shape[0] * param.shape[2] * param.shape[3]
    patches = _get_image_patches(input_, param_sizes)

    patches_reshaped = (
        patches
        .reshape(batch_size * img_size, -1)
        )

    output_grad_reshaped = (
        output_grad
        .transpose(0, 2, 3, 1)
        .reshape(batch_size * img_size, -1)
    )

    param_reshaped = param.transpose(0, 2, 3, 1).reshape(patch_size, -1)

    param_grad = np.matmul(patches_reshaped.transpose(1, 0),
                           output_grad_reshaped)

    param_grad_reshaped = (
        param_grad
        .reshape(in_channels, param_size, param_size, out_channels)
        .transpose(0, 3, 1, 2)
    )

    return param_grad_reshaped
```

Аналогично можно получить входной градиент, имитируя операции в полносвязном слое, а именно:

```
np.matmul(output_grad, self.param.transpose(1, 0))
```

Код, приведенный ниже, вычисляет входной градиент:

```
def _input_grad_matmul(input_: ndarray,
                       param: ndarray,
                       output_grad: ndarray):

    param_size = param.shape[2]
    batch_size = input_.shape[0]
    img_height = input_.shape[2]
    in_channels = input_.shape[1]

    output_grad_patches = _get_image_patches(output_grad, param_size)

    output_grad_patches_reshaped = (
        output_grad_patches
        .transpose(1, 0, 2, 3, 4)
        .reshape(batch_size * img_height * img_height, -1)
    )

    param_reshaped = (
        param
        .reshape(in_channels, -1)
    )

    input_grad = np.matmul(output_grad_patches_reshaped, param_
                           reshaped.transpose(1, 0))

    input_grad_reshaped = (
        input_grad
        .reshape(batch_size, img_height, img_height, 3)
        .transpose(0, 3, 1, 2)
    )

    return input_grad_reshaped
```

Эти три функции образуют ядро `Conv2DOperation`, в частности его методы `_output`, `_param_grad` и `_input_grad`, которые есть в библиотеке Lincoln (https://oreil.ly/2KPdFay) в репозитории GitHub книги.

Об авторе

Сет Вейдман — data scientist, несколько лет практиковал и преподавал основы машинного обучения. Начинал как первый специалист по данным в Trunk Club, где создавал модели оценки потенциальных клиентов и системы рекомендаций. В настоящее время работает в Facebook, где создает модели машинного обучения в составе команды по инфраструктуре. В промежутках преподавал Data Science и машинное обучение на буткемпах и в команде по корпоративному обучению в компании Metis. Обожает объяснять сложные концепции, стремясь найти простое в сложном.

Об обложке

Птица на обложке — берберийская каменная куропатка (*Alectoris barbara*). Берберийская куропатка родом из Северной Африки, Гибралтара и Канарских островов, обитает в лесах, кустарниках и засушливых районах. Позже поселилась в Португалии, Италии и Испании.

Берберийские куропатки предпочитают ходить, а не летать. Они весят до полукилограмма (300–400 граммов) и имеют размах крыльев 45 см. Птицы имеют светло-серый окрас, за исключением красновато-коричневой шеи с белыми пятнами, темно-желтого брюшка и бело-коричневых полос на боку. Ноги, клюв и кольца вокруг глаз красные.

Берберийские куропатки питаются семенами, разнообразной растительностью и насекомыми. Весной самки откладывают от 10 до 16 яиц. Этот вид известен как территориальный. Часто издают резкие крякающие звуки, заявляя о своем присутствии.

Хотя берберийской куропатке ничто не угрожает в глобальном масштабе, многие животные на обложках книг O'Reilly находятся под угрозой исчезновения. Все они важны для мира.

Иллюстрация на обложке выполнена Карен Монтгомери на основе черно-белой гравюры *British Birds*.