O'REILLY®

Глубокое обучение

Легкая разработка проектов на Python

Сет Вейдман

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.

Authorized Russian translation of the English edition of Deep Learning from Scratch ISBN 9781492041412 © 2019 Seth Weidman
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1675-1

© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Бестселлеры O'Reilly», 2021

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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 70х100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 500. Заказ

Оглавление

Предисловие **8**

Для понимания нейронных сетей нужно несколько мысленных моделей 10

Структура книги 11

Условные обозначения 13

Использование примеров кода 14

Благодарности..14

От издательства 15

Глава 1. Математическая база...16

Функции 17

Производные..22

Вложенные функции..24

Цепное правило 26

Более длинная цепочка 30

Функции нескольких переменных 34

Производные функций нескольких переменных.......................................36

Функции нескольких переменных с векторными аргументами..................37

Создание новых признаков из уже существующих 38

Производные функции нескольких векторных переменных......................41

Производные векторных функций: продолжение......................................43

Вычислительный граф для двух матриц..47

Самое интересное: обратный проход...51

Заключение...58

Глава 2. Основы глубокого обучения..59

Обучение с учителем..60

Алгоритмы обучения с учителем...62

Линейная регрессия 63
Обучение модели 69
Оценка точности модели 73
Код...74
Основы нейронных сетей...79
Обучение и оценка нейронной сети...86
Заключение...90

Глава 3. Основы глубокого обучения...91
Определение глубокого обучения: первый проход 91
Строительные блоки нейросети: операции 93
Строительные блоки нейросети: слои...97
Блочное строительство..100
Класс NeuralNetwork и, возможно, другие 107
Глубокое обучение с чистого листа 111
Trainer и Optimizer 115
Собираем все вместе 119
Заключение и следующие шаги 122

Глава 4. Расширения...123
Немного о понимании нейронных сетей.....................................124
Многопеременная логистическая функция активации с перекрестно-энтропийными потерями...126
Эксперименты 135
Импульс 138
Скорость обучения 142
Инициализация весов..145
Исключение, или дропаут..149
Заключение..153

Глава 5. Сверточная нейронная сеть 155
Нейронные сети и обучение представлениям 155
Слои свертки 160

Реализация операции многоканальной свертки 167
Свертка: обратный проход 171
Использование операции для обучения CNN 184
Заключение 188
Глава 6. Рекуррентные нейронные сети 190
Ключевое ограничение: работа с ветвлениями 191
Автоматическое дифференцирование 194
Актуальность рекуррентных нейронных сетей 199
Введение в рекуррентные нейронные сети 201
RNN: код 209
Заключение 230
Глава 7. Библиотека PyTorch 231
Класс PyTorch Tensor 231
Глубокое обучение с PyTorch 233
Сверточные нейронные сети в PyTorch 242
P. S. Обучение без учителя через автокодировщик 251
Заключение 261
Приложение А. Глубокое погружение 262
Цепное правило 262
Градиент потерь с учетом смещения 266
Свертка с помощью умножения матриц 266
Об авторе 272
Об обложке 272

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.

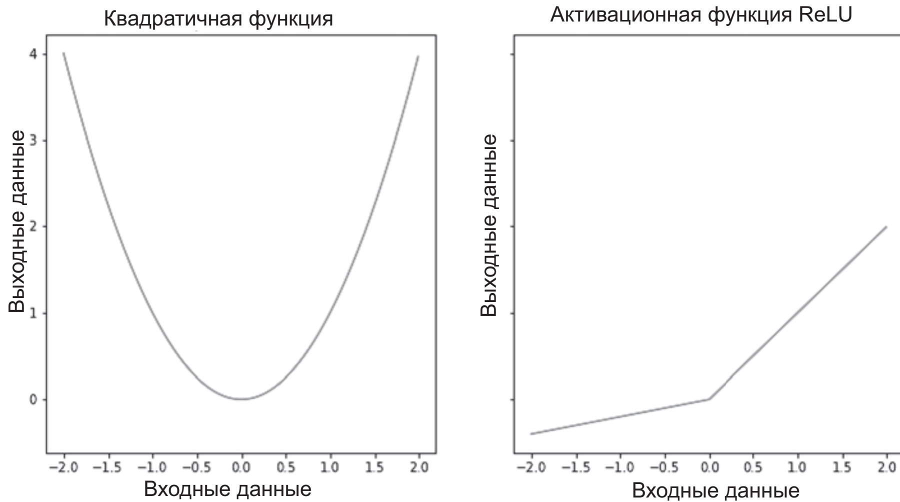


Рис. 1.1. Две непрерывные дифференцируемые функции

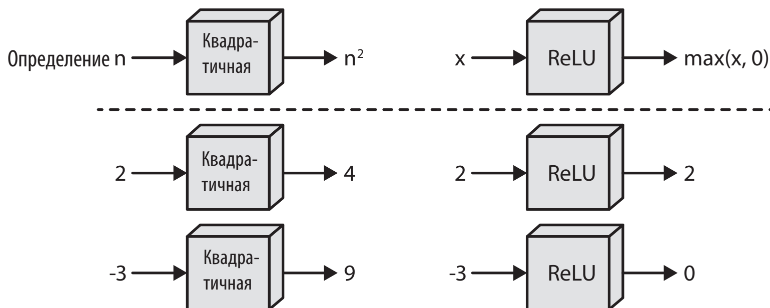


Рис. 1.2. Другой способ представления тех же функций

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]
```

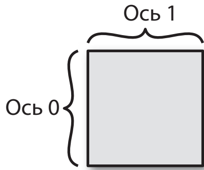


Рис. 1.3. Двумерный массив из библиотеки NumPy, в котором ось с индексом 0 соответствует строкам, а ось с индексом 1 — столбцам

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.

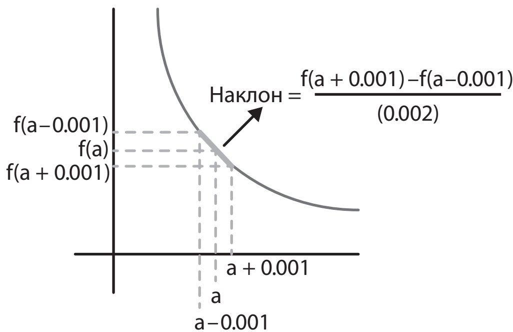


Рис. 1.4. Производная как угловой коэффициент

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.

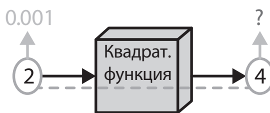


Рис. 1.5. Альтернативный способ визуализации концепции производной

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

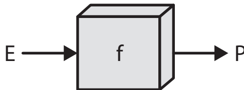
Код

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.

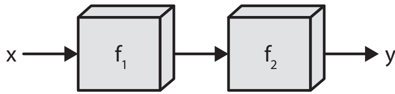


Рис. 1.6. Вложенные функции

Математическое представление

В математической нотации вложенная функция выглядит так:

$$f_2(f_1(x)) = y.$$

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.

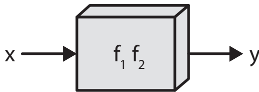


Рис. 1.7. Альтернативное представление вложенных функций

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

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

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 $\frac{df_2}{du}(1)=-2$.

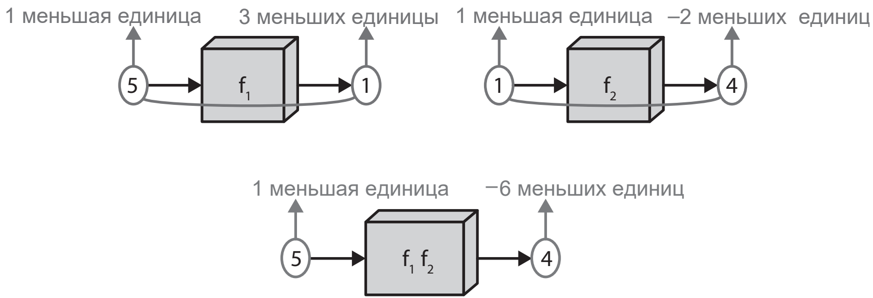


Рис. 1.8. Цепное правило

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 × 3 = –6. Именно поэтому в формуле для цепного правила фигурирует произведение:

$$\frac{df_2}{du}\big(f_1(x)\big)\textit{умножить на}\frac{df_1}{du}(x).$$

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)
```

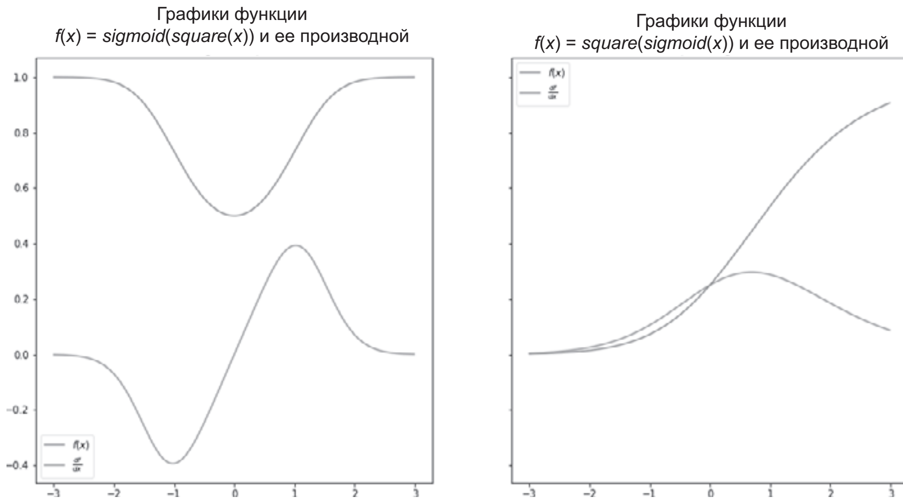
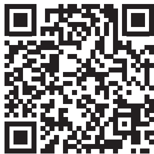


Рис. 1.9. Результат применения цепного правила



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.