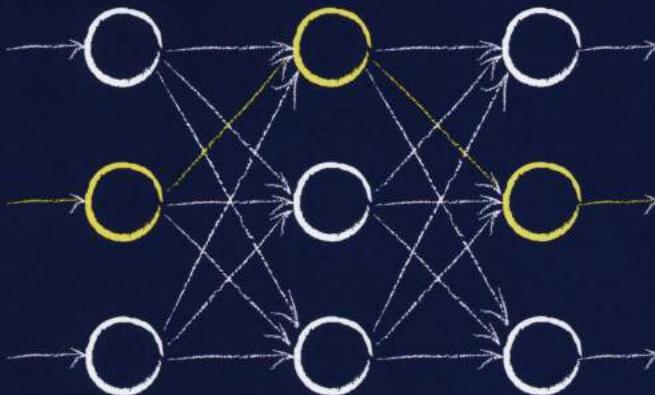


ПОЛНОЦВЕТНОЕ ИЗДАНИЕ

# СОЗДАЕМ НЕЙРОННУЮ СЕТЬ



*Математические идеи, лежащие в основе  
работы нейронных сетей, и поэтапное создание  
собственной нейронной сети на языке Python*

ТАРИК РАШИД

## ГЛАВА 2

# Создаем нейронную сеть на Python

*Чтобы по-настоящему в чем-то разобраться,  
нужно сделать это самому.*

*Начинай с малого... затем наращивай.*

В этой главе мы создадим собственную нейронную сеть. Для этого мы используем компьютер, поскольку, как вы уже знаете, нам придется выполнить тысячи вычислений. С помощью компьютеров это можно сделать очень быстро и без потери точности.

Мы будем сообщать компьютеру, что ему следует сделать, используя понятные ему инструкции. Компьютерам трудно понять обычный человеческий язык с присущими ему неточностью и неоднозначностью, который мы применяем в повседневном общении. Если уж люди часто не могут договориться между собой, то что говорить о компьютерах!

## Python

Мы будем использовать язык программирования Python. С него удобно начинать, поскольку он прост в изучении. Инструкции, написанные на Python одними людьми, легко читают и понимают другие люди. Кроме того, этот язык очень популярен и применяется во многих областях, включая научные исследования, преподавание, глобальные инфраструктуры, а также анализ данных и искусственный интеллект. Python все шире изучают в школах, а достигшая невероятных масштабов популярность микрокомпьютеров Raspberry Pi сделала Python доступным для еще более широкого круга людей, включая детей и студентов.

В приложении вы найдете инструкции относительно того, как настроить Raspberry Pi Zero для выполнения всей работы по созданию собственной нейронной сети с помощью Python. Raspberry Pi Zero — это чрезвычайно дешевый небольшой компьютер, стоимость которого в настоящее время составляет около 5 долларов. Это не опечатка — он действительно стоит всего 5 долларов!

О Python, как и о любом другом языке программирования, можно рассказать многое чего, но в книге мы сосредоточимся на создании собственной нейронной сети и будем изучать Python лишь в том объеме, который необходим для достижения этой конкретной цели.

## Интерактивный Python = IPython

Мы не будем самостоятельно устанавливать Python вместе со всеми возможными расширениями, предназначенными для математических вычислений и построения графиков, поскольку эта процедура может сопровождаться различными ошибками в процессе установки, вызванными неопытностью пользователя. Вместо этого мы воспользуемся готовым решением с заранее подготовленными пакетами, которое называется IPython.

Оболочка IPython содержит язык программирования Python и несколько расширений для выполнения численного и графического анализа данных, включая те, которые нам понадобятся. Она предоставляет удобное средство интерактивной разработки Jupyter Notebook (блокнот)<sup>1</sup>, напоминающее обычный блокнот, которое идеально подходит для оперативной проверки новых идей и анализа результатов. При этом отпадает необходимость заботиться о размещении файлов программ, интерпретаторов и библиотек, что могло бы отвлекать ваше внимание от сути задачи, особенно если что-то идет не так.

Посетите сайт [ipython.org](http://ipython.org), на котором предлагаются различные варианты установки IPython. Я использую пакет Anaconda, который можно загрузить на сайте <http://www.continuum.io/downloads>.

---

<sup>1</sup>Автором использовалась интерактивная оболочка IPython Notebook. В последних версиях IPython, в том числе в той, которая использовалась при подготовке перевода, эта оболочка называется Jupyter Notebook, чем подчеркивается ее совместимость не только с Python, но и с другими языками программирования. — Примеч. ред.

The screenshot shows the Anaconda 4.3.0 download page. At the top, there are links for 'Download For Windows', 'Download for OSX', and 'Download for Linux'. Below this, the 'Anaconda 4.3.0' section is shown for Windows. It includes a note about the BSD license, a 'Changelog' link, and a list of installation steps:

1. Download the installer
2. Optional: Verify data integrity with [MD5 or SHA-256](#) [More info](#)
3. Double-click the `.exe` file to install Anaconda and follow the instructions on the screen

Below the steps is a note: 'Behind a firewall? Use these [zipped Windows installers](#)'. To the right, two download options are presented:

- Python 3.6 version**
  - 64-BIT INSTALLER (422M)**
  - [32-BIT INSTALLER \(348M\)](#)
- Python 2.7 version**
  - 64-BIT INSTALLER (413M)**
  - [32-BIT INSTALLER \(339M\)](#)

Возможно, к тому времени, когда вы его посетите, сайт будет выглядеть иначе, но сути дела это не меняет. Сначала перейдите на вкладку, соответствующую используемой вами операционной системе: Windows, OS X или Linux. После этого обязательно выберите для загрузки версию **Python 3.6**, а не 2.7.

Версия Python 3 внедряется все более высокими темпами, и будущее принадлежит ей. Python 2.7 уже надежно прижился, но нам нужно смотреть вперед и начинать использовать Python 3 при всяком удобном случае, особенно для новых проектов. Сейчас большинство компьютеров **64-разрядные**, и для них следует загружать именно эту версию Python. Установка 32-разрядной версии может понадобиться разве что для компьютеров, выпущенных более десяти лет назад.

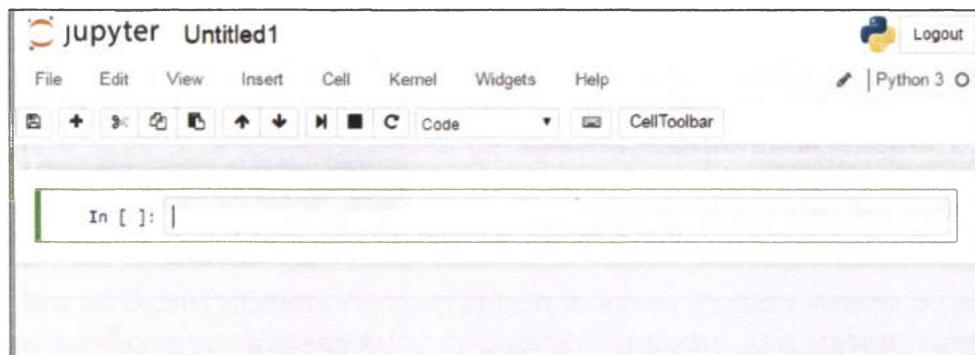
Установите IPython, следуя приведенным на сайте инструкциям. Этот процесс не должен вызвать у вас никаких затруднений.

## Простое введение в Python

Мы будем предполагать, что вы успешно справились с установкой IPython, следуя приведенным на сайте инструкциям, и теперь у вас есть доступ к этой оболочке.

## Блокноты

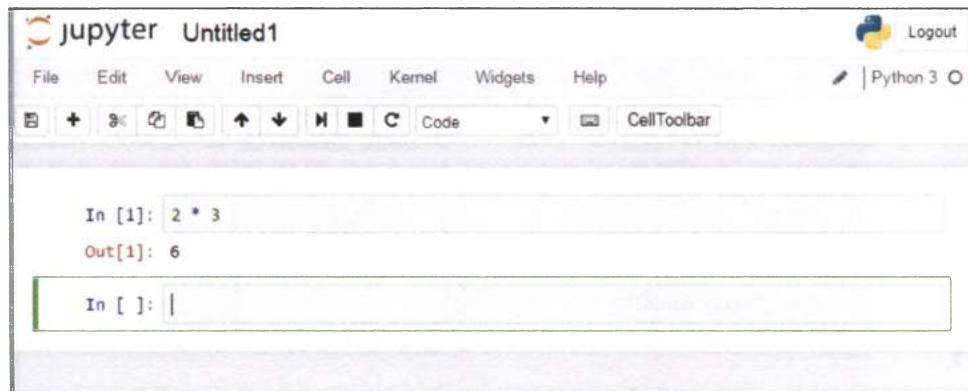
Запустив интерактивную оболочку Jupyter Notebook (Блокнот), щелкните на кнопке New у правого края окна и выберите в открывшемся меню пункт Python 3, что приведет к открытию пустого **блокнота**.



Блокнот интерактивен, т.е. ожидает от вас ввода команды, выполняет то, что вы ему приказали, выводит ответ и вновь переходит в состояние ожидания следующей команды или вопроса. В общем, он ведет себя как послушный робот-лакей, знающий толк в математике и обладающий тем дополнительным качеством, что никогда не устает.

При решении задач даже средней сложности имеет смысл разбивать их на части. Так легче структурировать логику задачи, а если что-то пойдет не так, особенно в большом проекте, вам будет проще найти ту его часть, в которой произошла ошибка. В терминологии IPython такие части называются **ячейками (cells)**. В показанном выше блокноте ячейка пуста, и вы, наверное, заметили мерцающий курсор, который приглашает вас ввести в нее какую-нибудь команду.

Давайте прикажем компьютеру что-то сделать. Например, попросим его перемножить два числа, скажем, умножить 2 на 3. Введите в ячейку текст “`2*3`” (кавычки вводить не следует) и щелкните на кнопке `run cell` (выполнить ячейку), напоминающей кнопку воспроизведения в проигрывателе. Компьютер должен быстро выполнить ваш приказ и вернуть следующий результат.



Как видите, компьютер выдал правильный результат: “6”. Только что мы предоставили компьютеру нашу первую инструкцию и успешно получили корректный ответ. Это была наша первая компьютерная программа!

Не обращайте внимания на надписи “In [1]” и “Out[1]”, которыми в IPython помечаются инструкция и ответ. Так оболочка напоминает вам о том, что именно вы просили сделать (`input`) и что вы получаете в ответ (`output`). Числа в скобках указывают на последовательность вопросов и ответов, что весьма удобно, когда вы то и дело вносите в код исправления и заново выполняете инструкции.

## Python — это просто

Когда я говорил, что Python — это простой язык программирования, я был совершенно серьезен. Перейдите к следующей ячейке “In []”, введите представленный ниже код и выполните его, щелкнув на кнопке запуска. В отношении инструкций, записанных на компьютерном языке, широко применяется термин **код**. Вместо щелчка на кнопке запуска кода можете воспользоваться комбинацией клавиш `<Ctrl+Enter>`, если вам, как и мне, этот способ более удобен.

```
print("Hello World!")
```

В ответ компьютер должен просто вывести в окне фразу “Hello World!”

```
In [1]: 2 * 3
Out[1]: 6

In [2]: print("Hello World!")
Hello World!
```

Как видите, ввод инструкции, предписывающей вывод фразы “Hello World!”, не привел к удалению предыдущей ячейки с содержащимися в ней собственной инструкцией и собственным выводом. Это средство оказывается очень полезным при поэтапном создании решений из нескольких частей.

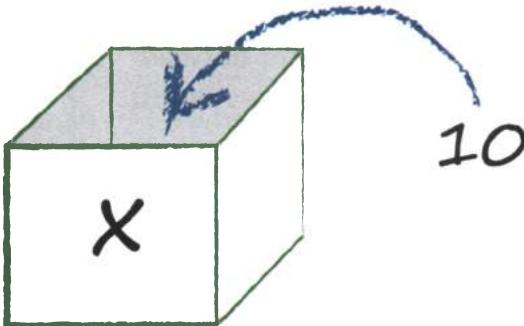
Посмотрим, что произойдет при выполнении следующего кода, который демонстрирует одну ключевую идею. Введите и выполните этот код в новой ячейке. Если новая пустая ячейка не отображается в окне, щелкните на кнопке с изображением знака “плюс”, после наведения на которую указателя мыши высвечивается подсказка *Insert Cell Below* (вставить ячейку снизу).

```
x = 10
print(x)
print(x+5)

y = x+7
print(y)

print(z)
```

Первая строка,  $x = 10$ , выглядит как математическая запись, утверждающая, что  $x$  равно 10. В Python это означает, что в виртуальное хранилище под названием  $x$  заносится значение 10. Данную простую концепцию иллюстрирует следующая диаграмма.



Значение “10” остается в хранилище до тех пор, пока в нем не возникнет необходимость. Вас не должна озадачить инструкция `print(x)`, поскольку это инструкция вывода информации на экран, с которой вы уже сталкивались. Она выдаст хранящееся в `x` значение, которое равно 10. Но почему будет выведено “10”, а не “`xx` можно вычислить и получить значение 10, которое и выводится. В следующей строке, которая содержит инструкцию `print(x+5)`, вычисляется выражение `x+5`, приводящее в конечном счете к значению  $10+5$ , или 15. Поэтому мы ожидаем, что на экран будет выведено “15”.

Следующая строка с инструкцией `y=x+7` также не собьет вас с толку, если вы будете руководствоваться той идеей, что Python стремится выполнить все возможные вычисления. В этой инструкции мы предписываем сохранить значение в новом хранилище, которое теперь называется `y`, но при этом возникает вопрос, а какое именно значение мы хотим сохранить? В инструкции указано выражение `x+7`, которое равно  $10+7$ , т.е. 17. Таким образом, именно это значение и будет сохранено в `y`, и следующая инструкция выведет его на экран.

А что произойдет со строкой `print(z)`, если мы не назначили `z` никакого значения, как это было сделано в случае `x` и `y`? Мы получим сообщение об ошибке, в вежливой форме информирующее нас о некорректности предпринимаемых нами действий и пытающееся быть максимально полезным, чтобы мы могли устраниТЬ ошибку. Должен заметить, что сообщения об ошибках не всегда успешноправляются со своей задачей — оказать помощь пользователю (причем этот недостаток характерен для большинства языков программирования).

Результаты выполнения описанного кода, включая вежливое сообщение об ошибке “name ‘z’ is not defined” (имя ‘z’ не определено), можно увидеть на следующей иллюстрации.

The screenshot shows a Jupyter Notebook window titled "jupyter Untitled4". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, and a Python 3 icon. The code editor contains three cells:

- In [1]: `2 * 3`  
Out[1]: 6
- In [2]: `print("Hello World!")`  
Hello World!
- In [3]:  

```
x = 10
print(x)
print(x+5)

y = x+7
print(y)

print(z)
```

  
10  
15  
17

A horizontal dashed line separates the code from the error output. Below it, the error message is displayed:

```
NameError: name 'z' is not defined
```

The bottom cell In [ ]: is empty.

Вышеупомянутые хранилища, обозначенные как *x* и *y* и используемые для хранения таких значений, как 10 и 17, называют **переменными**. В языках программирования переменные используются для создания обобщенных наборов инструкций по аналогии с тем, как математики используют алгебраические символы наподобие “*x*” и “*y*” для формулировки утверждений общего характера.

## Автоматизация работы

Компьютеры великолепно подходят для многократного выполнения однотипных задач — они не размышляют и производят вычисления намного быстрее, чем это удается делать людям с помощью калькуляторов!

Посмотрим, сможем ли мы заставить компьютер вывести квадраты первых десяти натуральных чисел, начиная с нуля: 0 в квадрате, 1 в квадрате, 2 в квадрате и т.д. Мы ожидаем получить следующий ряд чисел: 0, 1, 4, 9, 16, 25 и т.д.

Мы могли бы самостоятельно произвести все эти вычисления, а затем просто использовать инструкции вывода `print(0)`, `print(1)`, `print(4)` и т.д. Это сработает, но ведь наша цель заключается в том, чтобы всю вычислительную работу вместо нас выполнил компьютер. Более того, при этом мы упустили бы возможность создать типовой набор инструкций, позволяющий выводить квадраты чисел в любом заданном диапазоне. Для этого нам нужно сначала вооружиться некоторыми новыми идеями, к освоению которых мы сейчас и приступим.

Введите в очередную пустую ячейку следующий код:

```
list( range(10) )
```

Вы должны получить список из десяти чисел от 0 до 9. Это просто замечательно: мы заставили компьютер выполнить всю работу по созданию списка, дав ему соответствующее поручение. Теперь мы хозяева, а компьютер — наш слуга!

```
In [5]: list( range(10) )
Out[5]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Вероятно, вы удивлены тем, что список содержит числа от 0 до 9, а не от 1 до 10. Это не случайно. Многое из того, что связано с компьютерами, начинается с 0, а не с 1. Я много раз спотыкался на этом, полагая, что компьютерный список начинается с 1, а не с 0. Упорядоченные списки широко используются в качестве счетчиков при многократном выполнении некоторых вычислений и, в частности, когда используются итеративные функции.

Возможно, вы заметили, что мы опустили ключевое слово `print`, которое использовали при выводе фразы “Hello World!”, как, впрочем, обошлись без него и при вычислении произведения  $2 \times 3$ . Использование ключевого слова `print` не является обязательным при работе с Python в интерактивном режиме, поскольку оболочка знает, что мы хотим увидеть результат введенной инструкции.

Распространенным способом многократного выполнения вычислений с помощью компьютера является использование так называемых циклов. Слово “цикл” правильно передает суть происходящего, когда некоторые действия повторяются снова и снова, возможно, даже бесконечное число раз. Вместо того чтобы приводить формальное определение цикла, гораздо полезнее рассмотреть конкретный простой пример. Введите и выполните в новой ячейке следующий код.

```
for n in range(10):
    print(n)
    pass
print("готово")
```

Здесь имеются три новых элемента, которые мы последовательно обсудим. Первая строка содержит выражение `range(10)`, с которым вы уже знакомы. Оно создает список чисел от 0 до 9.

Конструкция `for n in` — это тот элемент, который создает цикл и заставляет выполнять некоторые действия для каждого числа из списка, организуя счетчик путем назначения текущего значения переменной `n`. Ранее мы уже обсуждали присваивание значений переменным, и здесь происходит то же самое: при первом проходе цикла выполняется присваивание `n=0`, а при последующих — присваивание `n=1, n=2` и так до тех пор, пока мы не дойдем до последнего элемента списка, для которого будет выполнено присваивание `n=9`.

Вы, несомненно, сразу же поймете смысл инструкции `print(n)` в следующей строке, которая просто выводит текущее значение `n`. Но обратите внимание на отступ перед текстом `print(n)`. В Python отступы играют важную роль, поскольку намеренно используются для того, чтобы показать подчиненность одних инструкций другим, в данном случае циклу, созданному с помощью конструкции `for n in`. Инструкция `pass` сигнализирует о конце цикла, и следующая строка, записанная с использованием обычного отступа, не является частью цикла. Это означает, что слово “готово” будет

выведено только один раз, а не десять. Представленный ниже результат подтверждает наши ожидания.

```
Out[5]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [6]: for n in range(10):
    print(n)
    pass
print("готово")

0
1
2
3
4
5
6
7
8
9
готово
```

Теперь вам должно быть понятно, что для получения квадратов чисел следует выводить на экран значения  $n \cdot n$ . В действительности мы можем поступить еще лучше и выводить фразы наподобие “Квадрат числа 3 равен 9”. Заметьте, что в инструкции переменные не заключаются в кавычки и поэтому вычисляются.

```
for n in range(10):
    print("Квадрат числа", n, "равен", n*n)
    pass
print("готово")
```

Результат приведен ниже.

```
In [7]: for n in range(10):
    print("Квадрат числа", n, "равен", n*n)
    pass
print("готово")

Квадрат числа 0 равен 0
Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
Квадрат числа 3 равен 9
Квадрат числа 4 равен 16
Квадрат числа 5 равен 25
Квадрат числа 6 равен 36
Квадрат числа 7 равен 49
Квадрат числа 8 равен 64
Квадрат числа 9 равен 81
готово
```

Это уже довольно сильный пример! Мы можем заставить компьютер очень быстро выполнить большой объем работы, используя минимальный набор инструкций. Точно так же мы, если бы захотели, легко могли бы увеличить число итераций до пятидесяти или даже тысячи с помощью выражений `range(50)` или `range(1000)`. Попробуйте сделать это самостоятельно!

## Комментарии

Прежде чем знакомиться с другими чудесными и невероятными по своим возможностям командами Python, взгляните на приведенный ниже простой код.

```
# следующая инструкция выводит куб числа 2
print(2**3)
```

Первая строка начинается с символа решетки (`#`). Python игнорирует все строки, начинающиеся с этого символа. Однако эти строки не бесполезны: с их помощью мы можем оставлять в коде полезные комментарии, которые помогут понять его предназначение другим людям или даже нам самим, если мы обратимся к нему после долгого перерыва.

Поверьте мне, вы скажете себе “спасибо” за то, что не поленились снабдить код комментариями, особенно если речь идет о сложных или менее очевидных фрагментах кода. Я не раз пытался расшифровать написанный собственноручно код, задавая себе вопрос: “А чего, собственно говоря, я хотел этим добиться?”

## Функции

В главе 1 мы интенсивно работали с математическими функциями. Мы относились к ним как к некоторым машинам, которые получают входные данные, выполняют некую работу и выдают результат. Эти функции действовали самостоятельно, и мы могли их многократно использовать.

Многие языки программирования, включая Python, упрощают создание повторно используемых инструкций. Подобно математическим функциям такие многократно используемые фрагменты кода, если они определены надлежащим образом, могут действовать автономно и обеспечивают создание более короткого элегантного кода.

Почему сокращается объем кода? Потому что вызывать функцию, обращаясь к ее имени, во сто крат лучше, чем каждый раз полностью записывать образующий эту функцию код.

А что означает “определены надлежащим образом”? Это означает, что вы четко определили, какие входные данные ожидает функция и какой тип выходных данных она выдает. Некоторые функции в качестве входных данных принимают только числа, и поэтому им нельзя передавать состоящие из букв слова.

И вновь, вы лучше всего сможете оценить плодотворность идеи использования функций, если мы обратимся к конкретному примеру. Введите и выполните следующий код.

```
# функция, которая принимает два числа в качестве входных данных
# и выводит их среднее значение
def среднее(x, y):
    print("первое число -", x)
    print("второе число -", y)
    a = (x + y) / 2.0
    print("среднее значение равно", a)
    return a
```

Обсудим, что делают эти команды. Первые две строки, начинающиеся с символов `#`, Python игнорирует, но мы используем их в качестве комментария для потенциальных читателей кода. Следующая за ними конструкция `def среднее(x, y):` сообщает Python, что мы собираемся определить новую повторно используемую функцию. Здесь `def` (от англ. *define* — определить) — ключевое слово; `среднее` — имя, которое мы даем функции. Его можно выбирать произвольно, но лучше всего использовать описательные имена, которые напомнят нам о том, для чего данная функция фактически предназначена. Конструкция в круглых скобках `(x, y)` сообщает Python, что функция принимает два входных значения, которые в пределах последующего определения функции обозначаются как `x` и `y`. В некоторых языках программирования требуется, чтобы вы указывали, объекты какого типа представляют переменные, но Python этого не делает, и впоследствии может лишь по-дружески пожурить вас за использование переменной не по назначению, например за то, что вы пытаетесь использовать слово, как будто оно является числом, или еще за какую-нибудь несуразицу подобного рода.

Теперь, когда мы объявили Python о своем намерении определить функцию, мы должны сообщить, что именно делает эта функция. Определение функции вводится с отступом, что отражено в приведенном выше коде. В некоторых языках для того, чтобы было понятно, к каким частям программы относятся те или иные фрагменты кода, используется множество всевозможных скобок, но создатели Python осознавали, что обилие скобок затрудняет чтение кода, тогда как отступы позволяют с первого взгляда получить представление о его структуре. В отношении целесообразности такого подхода мнения разделились, поскольку люди часто попадают в ловушку, забывая о важности отступов, но лично мне данная идея очень нравится! Это одна из самых полезных идей, зародившихся в чересчур заумном мире компьютерного программирования.

Понять смысл кода в определении функции среднее( $x, y$ ) вам будет совсем несложно, поскольку в нем используется все то, с чем вы уже сталкивались. Этот код выводит числа, передаваемые функции при ее вызове. Для вычисления среднего значения выводить аргументы вообще не обязательно, но мы делаем это для того, чтобы было совершенно понятно, что происходит внутри функции. В следующей инструкции вычисляется величина  $(x + y) / 2.0$ , и полученное значение присваивается переменной  $a$ . Мы выводим среднее значение исключительно для контроля того, что происходит в коде. Последняя инструкция — `return a`. Она завершает определение функции и сообщает Python, что именно должна вернуть функция в качестве результата, подобно машинам, которые мы ранее обсуждали.

Запустив этот код, вы увидите, что ничего не произошло. Никакие числа не выведены. Дело в том, что мы всего лишь определили функцию, но пока что не вызвали ее. На самом деле Python зарегистрировал функцию и будет держать ее наготове до тех пор, пока мы не захотим ее использовать.

Ведите в следующей ячейке `среднее(2, 4)`, чтобы активизировать функцию для значений 2 и 4. Кстати, в мире компьютерного программирования это называется **вызовом функции**. В соответствии с нашими ожиданиями данная функция выведет два входных значения и их вычисленное среднее. Кроме того, вы увидите ответ, выведенный отдельно, поскольку вызовы функций в интерактивных сессиях Python сопровождаются последующим выводом возвращаемого ими

значения. Ниже показано определение функции, а также результаты ее вызова с помощью инструкции среднее(2, 4) и инструкции среднее(200, 301) с большими значениями. Поэкспериментируйте самостоятельно, вызывая функцию с различными входными значениями.

```
In [9]: # функция, которая принимает два числа в качестве входных данных
# и выводит их среднее значение
def среднее(x,y):
    print("первое число -", x)
    print("второе число -", y)
    a = (x + y) / 2.0
    print("среднее значение равно", a)
    return a

In [10]: среднее(2,4)
первое число - 2
второе число - 4
среднее значение равно 3.0

Out[10]: 3.0

In [11]: среднее(200,301)
первое число - 200
второе число - 301
среднее значение равно 250.5

Out[11]: 250.5
```

Возможно, вы обратили внимание на то, что в коде функции среднее двух значений находится путем деления не на 2, а на 2,0. Почему мы так поступили? Это особенность Python, которая мне самому не нравится. Если бы мы делили на 2, то результат был бы округлен до ближайшего целого в меньшую сторону, поскольку Python рассматривал бы 2 просто как целое число. Это не изменило бы результат вызова среднее(2, 4), поскольку  $6/2$  равно 3, т.е. целому числу. Однако в случае вызова среднее(200, 301) среднее значение, равное  $501/2$ , т.е. 250,5, было бы округлено до значения 250. Я считаю это неразумным, но об этом стоит помнить, если ваш код ведет себя не совсем так, как ожидается. Деление же на 2,0 сообщает Python, что в наши намерения действительно входит работа с числами, которые могут иметь дробную часть, и мы не хотим, чтобы они округлялись.

Немного передохнем и поздравим себя. Мы определили повторно используемую функцию — один из наиболее важных и мощных элементов как математики, так и компьютерного программирования.

Мы будем применять повторно используемые функции при написании кода собственной нейронной сети. Например, имеет смысл создать повторно используемую функцию, которая реализовала бы вычисления с помощью сигмоиды, чтобы ее можно было вызывать много раз.

## Массивы

Массивы — это не более чем таблицы значений, и они действительно оказываются очень кстати. Как и в случае таблиц, вы можете ссылаться на конкретные ячейки по номерам строк и столбцов. Наверное, вам известно, что именно таким способом можно ссылаться на ячейки в электронных таблицах (например, B1 или C5) и производить над ними вычисления (например, C3+D7).

	A	B	C	D
1	это ячейка A1	это ячейка B1		
2	это ячейка A2	это ячейка B3		
3				
4				
5		это ячейка C5		
6				
7				
8				

Когда дело дойдет до написания кода нашей нейронной сети, мы используем массивы для представления матриц входных сигналов, весовых коэффициентов и выходных сигналов. Но не только этих, а также матриц, представляющих сигналы внутри нейронной сети и их распространение в прямом направлении, и матриц, представляющих обратное распространение ошибок. Поэтому давайте познакомимся с массивами. Введите и выполните следующий код:

```
import numpy
```

Что делает этот код? Команда `import` сообщает Python о необходимости привлечения дополнительных вычислительных ресурсов из другого источника для расширения круга имеющихся инструментов.

В некоторых случаях эти дополнительные инструменты являются частью Python, но они не находятся в состоянии готовности к немедленному использованию, чтобы без надобности не отягощать Python. Чаще всего расширения не относятся к основному инструментарию Python, но создаются сторонними разработчиками в качестве вспомогательных ресурсов, доступных для всеобщего использования. В данном случае мы импортируем дополнительный набор инструментов, объединенных в единый модуль под названием **numpy**. Пакет numpy очень популярен и предоставляет ряд полезных средств, включая массивы и операции над ними.

Введите в следующей ячейке приведенный ниже код.

```
a = numpy.zeros( [3,2] )
print(a)
```

В этом коде импортированный модуль numpy используется для создания массива размерностью  $3 \times 2$ , во всех ячейках которого содержатся нулевые значения. Мы сохраняем весь массив в переменной **a**, после чего выводим ее на экран. Результат подтверждает, что массив действительно состоит из нулей, хранящихся в виде таблицы с тремя строками и двумя столбцами.

```
In [2]: import numpy
In [3]: a = numpy.zeros( [3,2] )
print(a)
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

А теперь модифицируем содержимое этого массива и заменим некоторые из его нулей другими значениями. В приведенном ниже коде показано, как сослаться на конкретные ячейки для того, чтобы заменить хранящиеся в них значения новыми. Это напоминает обращение к нужным ячейкам электронной таблицы.

```
a[0,0] = 1
a[0,1] = 2
a[1,0] = 9
a[2,1] = 12
print(a)
```

Первая строка обновляет ячейку, находящуюся в строке 0 и столбце 0, заменяя все, что в ней до этого хранилось, значением 1. В остальных строках другие ячейки аналогичным образом обновляются, а последняя строка выводит массив на экран с помощью инструкции `print(a)`. На приведенной ниже иллюстрации показано, что собой представляет массив после всех изменений.

```
In [2]: import numpy  
In [3]: a = numpy.zeros( [3,2] )  
print(a)  
[[ 0.  0.]  
 [ 0.  0.]  
 [ 0.  0.]]  
In [4]: a[0,0] = 1  
a[0,1] = 2  
a[1,0] = 9  
a[2,1] = 12  
print(a)  
[[ 1.  2.]  
 [ 9.  0.]  
 [ 0.  12.]]
```

Теперь, когда вам известно, как присваивать значения элементам массива, вас, вероятно, интересует, каким образом можно узнать значение отдельного элемента, не выводя на экран весь массив? Это делается очень просто. Чтобы сослаться на содержимое ячейки массива, которое мы хотим вывести на экран или присвоить другой переменной, достаточно воспользоваться выражением наподобие `a[0,1]` или `a[1,0]`. Именно это демонстрирует приведенный ниже фрагмент кода.

```
print(a[0,1])  
v = a[1,0]  
print(v)
```

Запустив этот пример, вы увидите, что первая инструкция `print` выводит значение 2,0, т.е. содержимое ячейки [0,1]. Следующая инструкция присваивает значение элемента массива `a[1,0]` переменной `v` и выводит значение этой переменной. Как и ожидалось, выводится значение 9,0.

```
In [5]: print(a[0,1])
v = a[1,0]
print(v)
```

```
2.0
9.0
```

Нумерация столбцов и строк начинается с 0, а не с 1. Верхний левый элемент обозначается как [0,0], а не как [1,1]. Отсюда следует, что правому нижнему элементу соответствует обозначение [2,1], а не [3,2]. Иногда я сам попадаю впросак на этом, потому что забываю о том, что в компьютерном мире многие вещи начинаются с 0, а не с 1. Если мы попытаемся, к примеру, обратиться к элементу массива `a[0,2]`, то получим сообщение об ошибке.

```
In [6]: # попытка обращения к несуществующему элементу массива
a[0,2]

-
IndexError                                     Traceback (most recent call last)
t)
<ipython-input-6-99555f2824e5> in <module>()
      1 # попытка обращения к несуществующему элементу массива
----> 2 a[0,2]

IndexError: index 2 is out of bounds for axis 1 with size 2
```

Массивы, или матрицы, пригодятся нам для нейронных сетей, поскольку позволяют упростить инструкции при выполнении интенсивных вычислений, связанных с распространением сигналов и обратным распространением ошибок в сети, что обсуждалось в главе 1.

## Графическое представление массивов

Как и в случае больших числовых таблиц и списков, понять смысл чисел, содержащихся в элементах массива большой размерности, довольно трудно. В то же время их визуализация позволяет быстро получить общее представление о том, что они означают. Одним из способов графического отображения двухмерных числовых массивов является их представление в виде двухмерных поверхностей,

окраска которых в каждой точке зависит от значения соответствующего элемента массива. Способ установления соответствия между цветом поверхности и значением элемента массива вы выбираете по своему усмотрению. Например, можно преобразовывать значения в цвет, используя какую-либо цветовую шкалу, или закрасить всю поверхность белым цветом за исключением черных точек, которым соответствуют значения, превышающие определенный порог.

Давайте представим в графическом виде созданный ранее небольшой массив размерностью  $3 \times 2$ .

Но сначала мы должны расширить возможности Python для работы с графикой. Для этого необходимо импортировать дополнительный код Python, написанный другими людьми. Это все равно что взять у товарища книгу рецептов, поставить ее на свою книжную полку и пользоваться ею для приготовления блюд, которые раньше не умели готовить.

Ниже приведена инструкция, с помощью которой мы импортируем нужный нам пакет для работы с графикой:

```
import matplotlib.pyplot
```

Здесь `matplotlib.pyplot` — это имя новой “книги рецептов”, которую мы на время одолживаем. Во всех подобных случаях имя модуля или библиотеки, предоставляющей в ваше распоряжение дополнительный код, указывается после ключевого слова `import`. В процессе работы с Python часто приходится импортировать дополнительные средства, облегчающие жизнь программиста за счет повторного использования полезного кода, предложенного сторонними разработчиками. Возможно, и вы разработаете когда-нибудь код, которым поделитесь с коллегами!

Кроме того, мы должны дополнительно сообщить IPython о том, что любую графику следует отображать в блокноте, а не в отдельном окне. Это делается с помощью следующей директивы:

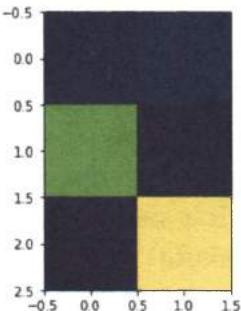
```
%matplotlib inline
```

Теперь мы полностью готовы к тому, чтобы представить массив в графическом виде. Введите и выполните следующий код:

```
matplotlib.pyplot.imshow(a, interpolation="nearest")
```

```
In [6]: import numpy  
a = numpy.zeros( [3,2] )  
a[0,0] = 1  
a[0,1] = 2  
a[1,0] = 9  
a[2,1] = 12  
  
In [7]: import matplotlib.pyplot  
%matplotlib inline  
  
In [8]: matplotlib.pyplot.imshow(a, interpolation="nearest")
```

```
Out[8]: <matplotlib.image.AxesImage at 0x8240048>
```



Потрясающе! Нам удалось представить содержимое массива в виде цветной диаграммы. Вы видите, что ячейки, содержащие одинаковые значения, закрашены одинаковыми цветами. Позже мы используем ту же функцию `imshow()` для визуализации значений, которые будем получать из нашей нейронной сети.

В состав пакета IPython входит богатый набор инструментов, предназначенных для визуализации данных. Вам следует изучить их самостоятельно, чтобы оценить диапазон их возможностей. Даже функция `imshow()` предлагает множество опций графического представления данных, таких как использование различных цветовых палитр.

## Объекты

Вам следует познакомиться с еще одним фундаментальным понятием, которое используется в Python, — понятием **объекта**. Объекты в чем-то схожи с повторно используемыми функциями, поскольку, однажды определив их, вы сможете впоследствии обращаться к ним множество раз. Но по сравнению с функциями объекты способны на гораздо большее.

Наилучший способ знакомства с объектами — это не обсуждение абстрактных понятий, а рассмотрение конкретного примера. Взгляните на следующий код.

```
# класс объектов Dog (собака)
class Dog:

    # собаки могут лаять
    def bark(self):
        print("Гав!")
        pass
    pass
```

Начнем с того, с чем мы уже знакомы. Прежде всего, код включает функцию `bark()`. Как нетрудно заметить, при вызове данной функции она выведет слово “Гав!” Это довольно просто.

А теперь рассмотрим код в целом. Вы видите ключевое слово `class`, за которым следуют имя `Dog` (собака) и структура, напоминающая функцию. Вы сразу же можете провести параллели между этой структурой и определением функции, которое также снабжается именем. Отличаются же они тем, что для определения функций используется ключевое слово `def`, тогда как для определения объектов используется ключевое слово `class`.

Прежде чем углубиться в обсуждение того, какое отношение эта структура, называемая **классом**, имеет к объектам, вновь обратимся к простому, но реальному коду, который оживляет эти понятия.

```
sizzles = Dog()
sizzles.bark()
```

В первой строке создается переменная `sizzles`, источником которой является, судя по всему, вызов функции. В действительности `Dog()` — это особая функция, которая создает экземпляр класса `Dog`. Теперь вы увидели, как создавать различные сущности из определений классов. Эти сущности называются **объектами**. В данном случае мы создали из определения класса `Dog` объект `sizzles` и можем считать, что этот объект является собакой!

В следующей строке для объекта `sizzles` вызывается функция `bark()`. С этим вы уже немного знакомы, поскольку ранее успели поработать с функциями. Но вам нужно еще привыкнуть к тому,

что функция `bark()` вызывается так, как если бы она была частью объекта `sizzles`. Это возможно, потому что данную функцию имеют все объекты, созданные на основе класса `Dog`, ведь именно в нем она и определена.

Опишем все это простыми словами. Мы создали переменную `sizzles`, разновидность класса `Dog`. Переменная `sizzles` — это объект, созданный по шаблону класса `Dog`. Объекты — это экземпляры класса.

Следующий пример показывает, что мы к этому времени успели сделать, и подтверждает, что функция `sizzles.bark()` действительно выводит слово “Гав!”

```
In [9]: # класс объектов Dog (собака)
class Dog:
```

```
    # собаки могут лаять
    def bark(self):
        print("Гав!")
        pass
    pass
```

```
In [10]: sizzles = Dog()
```

```
In [11]: sizzles.bark()
Гав!
```

Возможно, вы обратили внимание на непонятный элемент `self` в определении функции — `bark(self)`. Он здесь для того, чтобы указывать Python, к какому объекту приписывается функция при ее создании. Я считаю, что это должно быть очевидным, поскольку определение функции `bark()` включено в определение класса, а значит, Python и без того известно, к какому объекту ее следует прикрепить, но это мое личное мнение.

Рассмотрим примеры более полезного использования объектов и классов. Взгляните на следующий код.

```
sizzles = Dog()
```

```
mutley = Dog()
```

```
sizzles.bark()
```

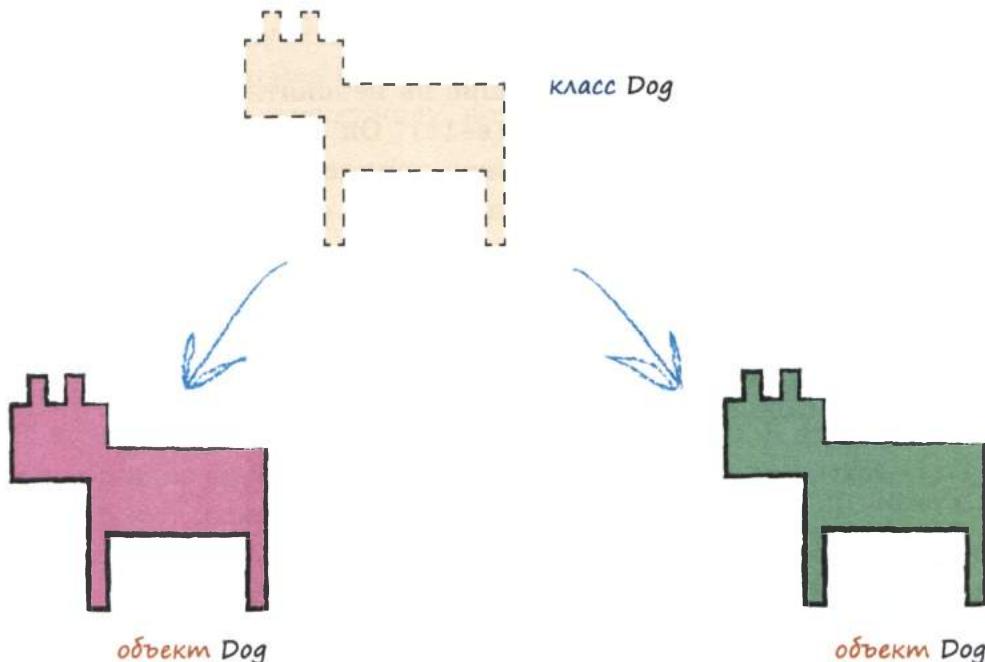
```
mutley.bark()
```

```
In [12]: sizzles = Dog()  
mutley = Dog()  
  
sizzles.bark()  
mutley.bark()
```

Гав!  
Гав!

Это уже интересно! Мы создали два объекта: `sizzles` и `mutley`. Важно понимать, что оба они были созданы на основе одного и того же определения класса `Dog`. Замечательно! Сначала мы определяем, как объекты должны выглядеть и как должны себя вести, а затем создаем их реальные экземпляры.

В этом и состоит суть различия между **классами** и **объектами**: первые представляют собой описания объектов, а вторые — реальные экземпляры классов, созданные в соответствии с этими описаниями. Класс — это рецепт приготовления пирога, а объект — сам пирог, изготовленный по данному рецепту. Процесс создания объектов на основе описания класса можно пояснить с помощью следующей наглядной иллюстрации.



А что нам дает создание объектов на основе класса? К чему все эти хлопоты? Не лучше ли было просто вывести фразу “Гав!” без какого-либо дополнительного кода?

Прежде всего, это имеет смысл делать тогда, когда возникает необходимость в создании множества однотипных объектов. Создавая эти объекты по единому образцу на основе класса, а не описывая полностью каждый из них по отдельности, вы экономите массу времени. Но реальное преимущество объектов заключается в том, что они обеспечивают компактное объединение данных и функциональности в одном месте. Централизованная организация фрагментов кода в виде объектов, которым они естественным образом принадлежат, значительно облегчает понимание структуры программы, особенно в случае сложных задач, что является большим плюсом для программистов. Собаки лают. Кнопки реагируют на щелчки. Динамики издают звуки. Принтеры печатают или жалуются, если закончилась бумага. Во многих компьютерных системах кнопки, динамики и принтеры действительно являются объектами, функции которых вы активизируете.

Функции, принадлежащие объектам, называют **методами**. С включением функций в состав объектов вы уже сталкивались, когда мы добавляли функцию `bark()` в определение класса `Dog`, и созданные на основе этого класса объекты `sizzles` и `mutley` включали в себя данный метод. Вы видели, что оба они “лаяли” в рассмотренном примере!

Нейронные сети принимают некоторые входные данные (входной сигнал), выполняют некоторые вычисления и выдают выходные данные (выходной сигнал). Вы также знаете, что их можно тренировать. Вы уже видели, что эти действия — тренировка и выдача ответа — являются естественными функциями нейронной сети, т.е. их можно рассматривать в качестве функций объекта нейронной сети. Вы также помните, что с нейронными сетями естественным образом связаны относящиеся к ним внутренние данные — весовые коэффициенты связей между узлами. Вот почему мы будем создавать нашу нейронную сеть в виде объекта.

Чтобы вы получили более полное представление о возможностях объектов, давайте добавим в класс переменные, предназначенные для хранения специфических данных конкретных объектов, а также методы, позволяющие просматривать и изменять эти данные.

Взгляните на приведенное ниже обновленное определение класса Dog. Оно включает ряд новых элементов, которые мы рассмотрим по отдельности.

```
# определение класса объектов Dog
class Dog:

    # метод для инициализации объекта внутренними данными
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # получить состояние
    def status(self):
        print("имя собаки: ", self.name)
        print("температура собаки: ", self.temperature)
        pass

    # задать температуру
    def setTemperature(self, temp):
        self.temperature = temp;
        pass

    # собаки могут лаять
    def bark(self):
        print("Гав!")
        pass
    pass
```

Прежде всего, хочу обратить ваше внимание на то, что мы добавили в класс Dog три новые функции. У нас уже была функция bark(), а теперь мы дополнительно включили в класс функции `__init__()`, `status()` и `setTemperature()`. Процедура добавления новых функций достаточно очевидна. Чтобы собака могла не только лаять с помощью функции bark(), мы при желании могли бы дополнительно предоставить ей возможность чихать с помощью функции sneeze().

Но что это за переменные, указанные в круглых скобках после имен новых функций? Например, функция `setTemperature` фактически определена как `setTemperature(self, temp)`. Функция со странным названием `__init__` фактически определена как `__init__(self, petname, temp)`. Эти переменные, получение значений которых функции ожидают во время вызова, называются параметрами. Помните функцию вычисления среднего `avg(x, y)`, с которой вы уже

сталкивались? Определение функции `avg()` явно указывало на то, что функция ожидает получения двух параметров. Следовательно, функция `__init__()` нуждается в параметрах `petname` и `temp`, а функция `setTemperature()` — только в параметре `temp`.

Заглянем внутрь этих функций. Начнем с функции с необычным названием `__init__()`. Зачем ей присвоено такое замысловатое имя? Это специальное имя, и Python будет вызывать функцию `__init__()` каждый раз при создании нового объекта данного класса. Это очень удобно для выполнения любой подготовительной работы до фактического использования объекта. Так что же именно происходит в этой математической функции инициализации? Мы создаем две новые переменные: `self.name` и `self.temperature`. Вы можете узнать их значения из переменных `petname` и `temp`, передаваемых функции. Часть `self.` в именах означает, что эти переменные являются собственностью объекта, т.е. принадлежат данному конкретному объекту и не зависят от других объектов `Dog` или общих переменных Python. Мы не хотим смешивать имя данной собаки с именем другой! Если это кажется вам слишком сложным, не беспокойтесь, все значительно упростится, когда мы приступим к рассмотрению конкретного примера.

Следующая на очереди — функция `status()`, которая действительно проста. Она не принимает никаких параметров и просто выводит значения переменных `name` и `temperature` объекта `Dog`.

Наконец, функция `setTemperature()` принимает параметр `temp`, значение которого при ее вызове присваивается внутренней переменной `self.temperature`. Это означает, что даже после того, как объект создан, вы можете в любой момент изменить его температуру, причем это можно сделать столько раз, сколько потребуется. Мы не будем тратить время на обсуждение того, почему все эти функции, включая `bark()`, принимают атрибут `self` в качестве первого параметра. Это особенность Python, которая лично меня немного раздражает, но таков ход эволюции Python. По замыслу разработчиков это должно напоминать Python, что функция, которую вы собираетесь определить, принадлежит объекту, ссылкой на который служит `self`. Но ведь, по сути, это и так очевидно, ведь код функции находится внутри класса. Таким образом, вас не должно удивлять то, что горячие дискуссии по этому поводу разгорелись даже в среде

профессиональных программистов на языке Python, так что вы не одиноки в компании тех, у кого такой подход вызывает недоумение.

Чтобы прояснить все, о чем мы говорили, рассмотрим конкретный пример. В приведенном ниже коде вы видите обновленный класс Dog, определенный с новыми функциями, и новый объект `lassie`, создаваемый с параметрами, один из которых задает его имя как "Lassie", а другой устанавливает его температуру равной 37.

```
In [1]: # определение класса объектов Dog
class Dog:

    # метод для инициализации объекта внутренними данными
    def __init__(self, petname, temp):
        self.name = petname;
        self.temperature = temp;

    # получить состояние
    def status(self):
        print("имя собаки: ", self.name)
        print("температура собаки: ", self.temperature)
        pass

    # задать температуру
    def setTemperature(self,temp):
        self.temperature = temp;
        pass

    # собаки могут лаять
    def bark(self):
        print("Гав!")
        pass

In [2]: # создать новый объект собаки на основе класса Dog
lassie = Dog("Lassie", 37)

In [3]: lassie.status()
Имя собаки: Lassie
температура собаки: 37
```

Как видите, вызов функции `status()` для объекта `lassie` класса `Dog` обеспечивает вывод его имени и текущего значения температуры. С момента создания объекта эта температура не изменилась.

Попытаемся изменить температуру объекта и проверим, действительно ли она изменилась, введя следующий код.

```
lassie.setTemperature(40)
lassie.status()
```

Результат представлен ниже.

```
In [2]: # создать новый объект собаки на основе класса Dog  
lassie = Dog("Lassie", 37)
```

```
In [3]: lassie.status()
```

```
имя собаки: Lassie  
температура собаки: 37
```

```
In [4]: lassie.setTemperature(40)  
lassie.status()
```

```
имя собаки: Lassie  
температура собаки: 40
```

Как видите, вызов функции `setTemperature(40)` действительно изменил внутреннее значение температуры объекта.

Мы должны быть довольны собой, поскольку узнали довольно много об объектах, которые многие считают сложной темой, но для нас она оказалась не таким уж трудным орешком!

Сейчас нам известно достаточно много о Python, чтобы мы могли самостоятельно создать собственную нейронную сеть.

## Проект нейронной сети на Python

Теперь мы можем приступить к созданию собственной нейронной сети с помощью языка Python, знакомиться с которым вы только что закончили. Мы будем двигаться постепенно, отрабатывая каждый шаг, и создадим свою программу на Python по частям.

Начинать с малого, а затем постепенно наращивать — весьма мудрый подход при создании компьютерного кода даже средней сложности.

С учетом того опыта, который мы к этому моменту накопили, кажется вполне естественным начать с построения скелета класса нейронной сети. Не будем терять времени!

### Скелет кода

Кратко обрисуем, что должен собой представлять класс нейронной сети. Мы знаем, что он должен содержать по крайней мере три функции:

- инициализация — задание количества входных, скрытых и выходных узлов;
- тренировка — уточнение весовых коэффициентов в процессе обработки предоставленных для обучения сети тренировочных примеров;
- опрос — получение значений сигналов с выходных узлов после предоставления значений входящих сигналов.

Возможно, в данный момент мы не сможем предложить идеальное определение класса, и впоследствии понадобится включить в него новые функции, но давайте начнем с этого минимального набора.

Наш начальный код может иметь примерно следующий вид.

```
# определение класса нейронной сети
class neuralNetwork:

    # инициализировать нейронную сеть
    def __init__():
        pass

    # тренировка нейронной сети
    def train():
        pass

    # опрос нейронной сети
    def query():
        pass
```

Для начала довольно неплохо. В действительности это достаточно надежный скелет, который в процессе нашей работы будет постепенно обрасти плотью в виде работающего кода.

## Инициализация сети

Начнем с инициализации. Мы знаем, что нам необходимо задать количество узлов входного, скрытого и выходного слоев. Эти данные определяют конфигурацию и размер нейронной сети. Вместо того чтобы жестко задавать их в коде, мы предусмотрим установку соответствующих значений в виде параметров во время создания объекта нейронной сети. Благодаря этому можно будет без труда создавать новые нейронные сети различного размера.

В основе нашего решения лежит одно важное соображение. Хорошие программисты, ученые-компьютерщики и математики при всяком удобном случае стараются писать обобщенный код, не зависящий от конкретных числовых значений. Это хорошая привычка, поскольку она вынуждает нас более глубоко продумывать решения, расширяющие применимость программы. Следуя ей, мы сможем использовать наши программы в самых разных сценариях. В данном случае это означает, что мы будем пытаться разрабатывать код для нейронной сети, поддерживающий как можно больше открытых опций и использующий как можно меньше предположений, чтобы его можно было легко применять в различных ситуациях. Мы хотим, чтобы один и тот же класс мог создавать как небольшие нейронные сети, так и очень большие, требуя лишь задания желаемых размеров сети в качестве параметров.

Кроме того, нам нельзя забывать о коэффициенте обучения. Этот параметр также можно устанавливать при создании новой нейронной сети. Посмотрите, как может выглядеть функция инициализации `__init__()` в подобном случае.

```
# инициализировать нейронную сеть
def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
    # задать количество узлов во входном, скрытом и выходном слое
    self.inodes = inputnodes
    self.hnodes = hiddennodes
    self.onodes = outputnodes

    # коэффициент обучения
    self.lr = learningrate
    pass
```

Добавим этот код в наше определение класса нейронной сети и попытаемся создать объект небольшой сети с тремя узлами в каждом слое и коэффициентом обучения, равным 0,3.

```
# количество входных, скрытых и выходных узлов
input_nodes = 3
hidden_nodes = 3
output_nodes = 3

# коэффициент обучения равен 0,3
learning_rate = 0.3
```

```
# создать экземпляр нейронной сети
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)
```

Конечно же, данный код позволяет получить объект сети, но такой объект пока что не будет особенно полезным, потому что не содержит ни одной функции, способной выполнять полезную работу. Впрочем, тут нет ничего плохого, это нормальная практика — начинать с малого и постепенно наращивать код, попутно находя и устраивая ошибки.

Исключительно для проверки того, что мы ничего не упустили, ниже показано, как выглядит блокнот IPython с определением класса нейронной сети и кодом для создания объекта.

```
In [1]: # определение класса нейронной сети
class neuralNetwork:

    # инициализировать нейронную сеть
    def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
        # задать количество узлов во входном, скрытом и выходном слое
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # коэффициент обучения
        self.lr = learningrate
        pass

    # тренировка нейронной сети
    def train():
        pass

    # опрос нейронной сети
    def query():
        pass

In [2]: # количество входных, скрытых и выходных узлов
input_nodes = 3
hidden_nodes = 3
output_nodes = 3

# коэффициент обучения равен 0.3
learning_rate = 0.3

# создать экземпляр нейронной сети
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)
```

```
In [ ]:
```

Что дальше? Мы сообщили объекту нейронной сети, сколько узлов разных типов нам необходимо иметь, но для фактического создания узлов пока что ничего не сделали.

## Весовые коэффициенты — сердце сети

Итак, нашим следующим шагом будет создание сети, состоящей из узлов и связей. Наиболее важная часть этой сети — **весовые коэффициенты связей** (веса). Они используются для расчета распространения сигналов в прямом направлении, а также обратного распространения ошибок, и именно весовые коэффициенты уточняются в попытке улучшить характеристики сети.

Ранее вы уже видели, насколько удобна компактная запись весов в виде матриц. Следовательно, мы можем создать следующие матрицы:

- матрицу весов для связей между входным и скрытым слоями,  $W_{\text{входной\_скрытый}}$ , размерностью `hidden_nodes × input_nodes`;
- другую матрицу для связей между скрытым и выходным слоями,  $W_{\text{скрытый\_выходной}}$ , размерностью `output_nodes × hidden_nodes`.

При необходимости вернитесь к главе 1, чтобы понять, почему первая матрица имеет размерность `hidden_nodes × input_nodes`, а не `input_nodes × hidden_nodes`.

Вспомните из главы 1, что начальные значения весовых коэффициентов должны быть небольшими и выбираться случайным образом. Следующая функция из пакета `numpy` генерирует массив случайных чисел в диапазоне от 0 до 1, где размерность массива равна `rows × columns`:

```
numpy.random.rand(rows, columns)
```

Все хорошие программисты ищут в Интернете онлайновую документацию по функциям Python и даже находят полезные функции, о существовании которых и не подозревали. Для поиска любой информации, касающейся программирования, лучше всего использовать Google. Выполните такой поиск для функции `numpy.random.rand()`, если хотите узнать о ней больше.

Если мы собираемся использовать расширения пакета `numpy`, мы должны импортировать эту библиотеку в самом начале кода. Прежде всего удостоверимся, что нужная нам функция работает. В приведенном ниже примере используется матрица размерностью  $3 \times 3$ . Как видите, матрица заполнилась случайными значениями в диапазоне от 0 до 1.

```
In [1]: import numpy  
  
In [2]: numpy.random.rand(3,3)  
Out[2]: array([[ 0.5222268,  0.38405384,  0.52803768],  
               [ 0.97782786,  0.76440116,  0.53243937],  
               [ 0.59079114,  0.93147621,  0.61986779]])
```

Данный подход можно улучшить, поскольку весовые коэффициенты могут иметь не только положительные, но и отрицательные значения и изменяться в пределах от  $-1,0$  до  $+1,0$ . Для простоты мы вычтем  $0,5$  из этих граничных значений, перейдя к диапазону значений от  $-0,5$  до  $+0,5$ . Ниже представлены результаты применения этого изящного подхода, которые демонстрируют, что некоторые весовые коэффициенты теперь имеют отрицательные значения.

```
In [3]: numpy.random.rand(3,3) - 0.5  
Out[3]: array([[-0.10804685,  0.36596034,  0.027477 ],  
               [ 0.22649054, -0.40888039,  0.43435292],  
               [ 0.27549563,  0.35735947, -0.1666345 ]])
```

Мы готовы создать матрицу начальных весов в нашей программе на Python. Эти весовые коэффициенты составляют неотъемлемую часть нейронной сети и служат ее характеристиками на протяжении всей ее жизни, а не времененным набором данных, которые исчезают сразу же после того, как функция отработала. Это означает, что они должны быть частью процесса инициализации и быть доступными для других методов, таких как функции тренировки и опроса сети.

Ниже приведен код, включая комментарии, который создает две матрицы весовых коэффициентов, используя значения переменных `self.inodes`, `self.hnodes` и `self.onodes` для задания соответствующих размеров каждой из них.

```
# Матрицы весовых коэффициентов связей wih (между входным и скрытым # слоями) и who (между скрытым и выходным слоями).  
# Весовые коэффициенты связей между узлом i и узлом j следующего слоя # обозначены как w_i_j:  
# w11 w21  
# w12 w22 и т.д.
```

```
self.wih = (numpy.random.rand(self.hnodes, self.inodes) - 0.5)
self.who = (numpy.random.rand(self.onodes, self.hnodes) - 0.5)
```

Великолепная работа! Мы реализовали то, что составляет саму сердцевину нейронной сети, — матрицы связей между ее узлами!

## По желанию: улучшенный вариант инициализации весовых коэффициентов

Описанное в этом разделе обновление кода не является обязательным, поскольку это всего лишь простое, но популярное усовершенствование процесса инициализации весовых коэффициентов.

В конце главы 1 мы обсуждали различные способы подготовки данных и инициализации коэффициентов. Так вот, некоторые предпочитают несколько усовершенствованный подход к созданию случайных начальных значений весов. Для этого весовые коэффициенты выбираются из нормального распределения с центром в нуле и со стандартным отклонением, величина которого обратно пропорциональна корню квадратному из количества входящих связей на узел.

Это легко делается с помощью библиотеки `numpy`. Опять-таки, в отношении поиска онлайновой документации Google незаменим. Функция `numpy.random.normal()` описана по такому адресу:

<https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.normal.html>

Она поможет нам с извлечением выборки из нормального распределения. Ее параметрами являются центр распределения, стандартное отклонение и размер массива `numpy`, если нам нужна матрица случайных чисел, а не одиночное число.

Обновленный код инициализации весовых коэффициентов будет выглядеть примерно так.

```
self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
                                (self.hnodes, self.inodes))
self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
                                (self.onodes, self.hnodes))
```

Как видите, центр нормального распределения установлен здесь в 0,0. Стандартное отклонение вычисляется по количеству узлов

в следующем слое с помощью функции `pow(self.hnodes, -0.5)`, которая просто возводит количество узлов в степень  $-0,5$ . Последний параметр определяет конфигурацию массива `numpy`.

## Опрос сети

Казалось бы, логично начать с разработки кода, предназначенного для тренировки нейронной сети, заполнив инструкциями функцию `train()`, которая на данном этапе пуста, но мы отложим ее на время и займемся более простой функцией `query()`. Благодаря этому мы постепенно приобретем уверенность в своих силах и получим опыт в использовании как языка Python, так и матриц весовых коэффициентов в объекте нейронной сети.

Функция `query()` принимает в качестве аргумента входные данные нейронной сети и возвращает ее выходные данные. Все довольно просто, но вспомните, что для этого нам нужно передать сигналы от узлов входного слоя через скрытый слой к узлам выходного слоя для получения выходных данных. При этом, как вы помните, по мере распространения сигналов мы должны сглаживать их, используя весовые коэффициенты связей между соответствующими узлами, а также применять сигмоиду для уменьшения выходных сигналов узлов.

В случае большого количества узлов написание для каждого из них кода на Python, осуществляющего сглаживание весовых коэффициентов, суммирование сигналов и применение к ним сигмоиды, превратилось бы в сплошной кошмар.

К счастью, нам не нужно писать детальный код для каждого узла, поскольку мы уже знаем, как записать подобные инструкции в простой и компактной матричной форме. Ниже показано, как можно получить входящие сигналы для узлов скрытого слоя путем сочетания матрицы весовых коэффициентов связей между входным и скрытым слоями с матрицей входных сигналов:

$$x_{\text{скрытый}} = w_{\text{входной\_скрытый}} \cdot I$$

В этом выражении замечательно не только то, что в силу его краткости нам легче его записать, но и то, что такие компьютерные языки, как Python, распознают матрицы и эффективно выполняют все

расчеты, поскольку им известно об однотипности всех стоящих за этим вычислений.

Вы будете удивлены простотой соответствующего кода на языке Python. Ниже представлена инструкция, которая показывает, как применить функцию скалярного произведения библиотеки `numpy` к матрицам весов и входных сигналов:

```
hidden_inputs = numpy.dot(self.wih, inputs)
```

Вот и все!

Эта короткая строка кода Python выполняет всю работу по объединению всех входных сигналов с соответствующими весами для получения матрицы сглаженных комбинированных сигналов в каждом узле скрытого слоя. Более того, нам не придется ее переписывать, если в следующий раз мы решим использовать входной или скрытый слой с другим количеством узлов. Этот код все равно будет работать!

Именно эта мощь и элегантность матричного подхода являются причиной того, что перед этим мы не пожалели потратить время и усилия на его рассмотрение.

Для получения выходных сигналов скрытого слоя мы просто применяем к каждому из них сигмоиду:

$$O_{\text{скрытый}} = \text{сигмоида} (X_{\text{скрытый}})$$

Это не должно вызвать никаких затруднений, особенно если сигмоида уже определена в какой-нибудь библиотеке Python. Оказывается, так оно и есть! Библиотека `scipy` в Python содержит набор специальных функций, в том числе сигмоиду, которая называется `expit()`. Не спрашивайте меня, почему ей присвоили такое дурацкое имя. Библиотека `scipy` импортируется точно так же, как и библиотека `numpy`.

```
# библиотека scipy.special содержит сигмоиду expit()
import scipy.special
```

Поскольку в будущем мы можем захотеть поэкспериментировать с функцией активации, настроив ее параметры или полностью заменив другой функцией, лучше определить ее один раз в объекте нейронной сети во время его инициализации. После этого мы сможем

неоднократно ссылаться на нее, точно так же, как на функцию `query()`. Такая организация программы означает, что в случае внесения изменений нам придется сделать это только в одном месте, а не везде в коде, где используется функция активации.

Ниже приведен код, определяющий функцию активации, который мы используем в разделе инициализации нейронной сети.

```
# использование сигмоиды в качестве функции активации  
self.activation_function = lambda x: scipy.special.expit(x)
```

Что делает этот код? Он не похож ни на что, с чем мы прежде сталкивались. Что это за `lambda`? Не стоит пугаться, здесь нет ничего страшного. Все, что мы сделали, — это создали функцию наподобие любой другой, только с использованием более короткого способа записи, называемого **лямбда-выражением**. Вместо привычного определения функции в форме `def имя()` мы использовали волшебное слово `lambda`, которое позволяет создавать функции быстрым и удобным способом, что называется, “на лету”. В данном случае функция принимает аргумент `x` и возвращает `scipy.special.expit()`, а это есть не что иное, как сигмоида. Функции, создаваемые с помощью лямбда-выражений, являются безымянными или, как предпочитают говорить опытные программисты, **анонимными**, но данной функции мы присвоили имя `self.activation_function()`. Это означает, что всякий раз, когда потребуется использовать функцию активации, ее нужно будет вызвать как `self.activation_function()`.

Итак, возвращаясь к нашей задаче, мы применим функцию активации к сглаженным комбинированным входящим сигналам, поступающим на скрытые узлы. Соответствующий код совсем не сложен.

```
# рассчитать исходящие сигналы для скрытого слоя  
hidden_outputs = self.activation_function(hidden_inputs)
```

Таким образом, сигналы, исходящие из скрытого слоя, описываются матрицей `hidden_outputs`.

Мы прошли промежуточный скрытый слой, а как быть с последним, выходным слоем? В действительности распространение сигнала от скрытого слоя до выходного ничем принципиально не отличается от предыдущего случая, поэтому способ расчета остается тем же, а значит, и код будет аналогичен предыдущему.

Ниже приведен итоговый фрагмент кода, объединяющий расчеты сигналов скрытого и выходного слоев.

```
# рассчитать входящие сигналы для скрытого слоя
hidden_inputs = numpy.dot(self.wih, inputs)
# рассчитать исходящие сигналы для скрытого слоя
hidden_outputs = self.activation_function(hidden_inputs)

# рассчитать входящие сигналы для выходного слоя
final_inputs = numpy.dot(self.who, hidden_outputs)
# рассчитать исходящие сигналы для выходного слоя
final_outputs = self.activation_function(final_inputs)
```

Если отбросить комментарии, здесь всего четыре строки кода, выделенные полужирным шрифтом, которые выполняют все необходимые расчеты: две — для скрытого слоя и две — для выходного слоя.

## Текущее состояние кода

Сделаем паузу и переведем дыхание, чтобы посмотреть, как выглядит в целом код, который мы к этому времени создали. А выглядит он так.

```
# определение класса нейронной сети
class neuralNetwork:

    # инициализировать нейронную сеть
    def __init__(self, inputnodes, hiddennodes, outputnodes,
     learningrate):
        # задать количество узлов во входном, скрытом и выходном слое
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # Матрицы весовых коэффициентов связей wih и who.
        # Весовые коэффициенты связей между узлом i и узлом j
        # следующего слоя обозначены как w_i_j:
        # w11 w21
        # w12 w22 и т.д.
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
     (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
     (self.onodes, self.hnodes))

    # коэффициент обучения
```

```

self.lr = learningrate

# использование сигмоиды в качестве функции активации
self.activation_function = lambda x: scipy.special.expit(x)

pass

# тренировка нейронной сети
def train() :
    pass

# опрос нейронной сети
def query(self, inputs_list):
    # преобразовать список входных значений
    # в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)
    # рассчитать исходящие сигналы для скрытого слоя
    hidden_outputs = self.activation_function(hidden_inputs)

    # рассчитать входящие сигналы для выходного слоя
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # рассчитать исходящие сигналы для выходного слоя
    final_outputs = self.activation_function(final_inputs)

    return final_outputs

```

Но это только определение класса, перед которым в первой ячейке блокнота IPython следует поместить код, импортирующий модули numpy и scipy.special.

```

import numpy
# библиотека scipy.special с сигмоидой expit()
import scipy.special

```

Попутно отмечу, что функции query() в качестве входных данных потребуются только входные сигналы input\_list. Ни в каких других входных данных она не нуждается.

Мы достигли значительного прогресса и теперь можем вернуться к недостающему фрагменту — функции train(). Вспомните, что тренировка включает две фазы: первая — это расчет выходного сигнала, что и делает функция query(), а вторая — обратное распространение

ошибок, информирующее вас о том, каковы должны быть поправки к весовым коэффициентам.

Прежде чем переходить к написанию кода функции `train()`, осуществляющей тренировку сети на примерах, протестируем, как работает уже имеющийся код. Для этого создадим небольшую сеть и предоставим ей некоторые входные данные. Очевидно, что никакого реального смысла результаты содержать не будут, но нам важно лишь проверить работоспособность всех созданных функций.

В следующем примере создается небольшая сеть с входным, скрытым и выходным слоями, содержащими по три узла, которая опрашивается с использованием случайно выбранных входных сигналов (1.0, 0.5 и -1.5).

```
In [2]: # количество входных, скрытых и выходных узлов  
input_nodes = 3  
hidden_nodes = 3  
output_nodes = 3  
  
# коэффициент обучения равен 0.3  
learning_rate = 0.3  
  
# создать экземпляр нейронной сети  
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)  
  
In [3]: n.query([1.0, 0.5, -1.5])  
[1.0, 0.5, -1.5]  
Out[3]: array([[ 0.58812286],  
               [ 0.38342223],  
               [ 0.57511807]])
```

Нетрудно заметить, что при создании объекта нейронной сети необходимо задавать значение коэффициента обучения, даже если он не используется. Это объясняется тем, что определение класса нейронной сети включает функцию инициализации `__init__()`, которая требует предоставления данного аргумента. Если его не указать, программа не сможет быть выполнена, и отобразится сообщение об ошибке.

Вы также могли заметить, что входные данные передаются в виде списка, который в Python обозначается квадратными скобками. Вывод также представлен в виде числового списка. Эти значения не имеют никакого реального смысла, поскольку мы не тренировали

сеть, но тот факт, что во время выполнения программы не возникли ошибки, должен нас радовать.

## Тренировка сети

Приступим к решению несколько более сложной задачи тренировки сети. Ее можно разделить на две части.

- Первая часть — расчет выходных сигналов для заданного тренировочного примера. Это ничем не отличается от того, что мы уже можем делать с помощью функции `query()`.
- Вторая часть — сравнение рассчитанных выходных сигналов с желаемым ответом и обновление весовых коэффициентов связей между узлами на основе найденных различий.

Сначала запишем готовую первую часть.

```
# тренировка нейронной сети
def train(self, inputs_list, targets_list):
    # преобразовать список входных значений в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)
    # рассчитать исходящие сигналы для скрытого слоя
    hidden_outputs = self.activation_function(hidden_inputs)

    # рассчитать входящие сигналы для выходного слоя
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # рассчитать исходящие сигналы для выходного слоя
    final_outputs = self.activation_function(final_inputs)

    pass
```

Этот код почти совпадает с кодом функции `query()`, поскольку процесс передачи сигнала от входного слоя к выходному остается одним и тем же.

Единственным отличием является введение дополнительного параметра `targets_list`, передаваемого при вызове функции, поскольку невозможно тренировать сеть без предоставления ей тренировочных примеров, которые включают желаемые или целевые значения:

```
def train(self, inputs_list, targets_list)
```

Список `targets_list` преобразуется в массив точно так же, как список `input_list`:

```
targets = numpy.array(targets_list, ndmin=2).T
```

Теперь мы очень близки к решению основной задачи тренировки сети — уточнению весов на основе расхождения между расчетными и целевыми значениями.

Будем решать эту задачу поэтапно.

Прежде всего, мы должны вычислить ошибку, являющуюся разностью между желаемым целевым выходным значением, предоставленным тренировочным примером, и фактическим выходным значением. Она представляет собой разность между матрицами (`targets - final_outputs`), рассчитываемую поэлементно. Соответствующий код выглядит очень просто, что еще раз подтверждает мощь и красоту матричного подхода.

```
# ошибка = целевое значение - фактическое значение
output_errors = targets - final_outputs
```

Далее мы должны рассчитать обратное распространение ошибок для узлов скрытого слоя. Вспомните, как мы распределяли ошибки между узлами пропорционально весовым коэффициентам связей, а затем рекомбинировали их на каждом узле скрытого слоя. Эти вычисления можно представить в следующей матричной форме:

$$\text{ошибки}_{\text{скрытый}} = \text{веса}^T_{\text{скрытый\_выходной}} \cdot \text{ошибки}_{\text{выходной}}$$

Код, реализующий эту формулу, также прост в силу способности Python вычислять скалярные произведения матриц с помощью модуля `numpy`.

```
# ошибки скрытого слоя - это ошибки output_errors,
# распределенные пропорционально весовым коэффициентам связей
# и рекомбинированные на скрытых узлах
hidden_errors = numpy.dot(self.who.T, output_errors)
```

Итак, мы получили то, что нам необходимо для уточнения весовых коэффициентов в каждом слое. Для весов связей между скрытым и выходным слоями мы используем переменную `output_errors`.

Для весов связей между входным и скрытым слоями мы используем только что рассчитанную переменную `hidden_errors`.

Ранее нами было получено выражение для обновления веса связи между узлом  $j$  и узлом  $k$  следующего слоя в матричной форме.

$$\Delta W_{jk} = \alpha * E_k * \text{сигмоида}(O_k) * (1 - \text{сигмоида}(O_k)) \cdot O_j^T$$

Величина  $\alpha$  — это коэффициент обучения, а сигмоида — это функция активации, с которой вы уже знакомы. Вспомните, что символ "\*" означает обычное поэлементное умножение, а символ "." — скалярное произведение матриц. Последний член выражения — это транспонированная ( $T$ ) матрица исходящих сигналов предыдущего слоя. В данном случае транспонирование означает преобразование столбца выходных сигналов в строку.

Это выражение легко транслируется в код на языке Python. Сначала запишем код для обновления весов связей между скрытым и выходным слоями.

```
# обновить весовые коэффициенты связей между скрытым и выходным слоями
self.who += self.lr * numpy.dot((output_errors * final_outputs *
    (1.0 - final_outputs)), numpy.transpose(hidden_outputs))
```

Это довольно длинная строка кода, но цветовое выделение поможет вам разобраться в том, как она связана с приведенным выше математическим выражением. Коэффициент обучения `self.lr` просто умножается на остальную часть выражения. Есть еще матричное умножение, выполняемое с помощью функции `numpy.dot()`, и два элемента, выделенных синим и красным цветами, которые отображают части, относящиеся к ошибке и сигмоидам из следующего слоя, а также транспонированная матрица исходящих сигналов предыдущего слоя.

Операция `+=` означает увеличение переменной, указанной слева от знака равенства, на значение, указанное справа от него. Поэтому `x+=3` означает, что `x` увеличивается на 3. Это просто сокращенная запись инструкции `x=x+3`. Аналогичный способ записи допускается и для других арифметических операций. Например, `x/=3` означает деление `x` на 3.

Код для уточнения весовых коэффициентов связей между входным и скрытым слоями будет очень похож на этот. Мы воспользуемся симметрией выражений и просто перепишем код, заменяя в нем имена переменных таким образом, чтобы они относились к предыдущим слоям. Ниже приведен суммарный код для двух наборов весовых коэффициентов, отдельные элементы которого выделены цветом таким образом, чтобы сходные и различающиеся участки кода можно было легко заметить.

```
# обновить весовые коэффициенты связей между скрытым и выходным слоями  
self.who += self.lr * numpy.dot((output_errors * final_outputs *  
↳(1.0 - final_outputs)), numpy.transpose(hidden_outputs))
```

```
# обновить весовые коэффициенты связей между входным и скрытым слоями  
self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs *  
↳(1.0 - hidden_outputs)), numpy.transpose(inputs))
```

Вот и все!

Даже не верится, что вся та работа, для выполнения которой нам потребовалось множество вычислений, и все те усилия, которые мы вложили в разработку матричного подхода и способа минимизации ошибок сети методом градиентного спуска, свелись к паре строк кода! Отчасти мы обязаны этим языку Python, но фактически это закономерный результат нашего упорного труда, вложенного в упрощение того, что легко могло стать сложным и приобрести устрашающий вид.

## Полный код нейронной сети

Мы завершили разработку класса нейронной сети. Он приведен ниже для справки, и вы можете получить его в любой момент, воспользовавшись следующей ссылкой на GitHub — крупнейшем

веб-ресурсе для совместной разработки проектов, на котором люди бесплатно делятся своим кодом:

- [https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2\\_neural\\_network.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network.ipynb)

---

```
# определение класса нейронной сети
class neuralNetwork:
```

```
    # инициализировать нейронную сеть
    def __init__(self, inputnodes, hiddennodes, outputnodes,
                 learningrate):
        # задать количество узлов во входном, скрытом и выходном слое
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # Матрицы весовых коэффициентов связей wih и who.
        # Весовые коэффициенты связей между узлом i и узлом j
        # следующего слоя обозначены как w_i_j:
        # w11 w21
        # w12 w22 и т.д.
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
                                       (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
                                       (self.onodes, self.hnodes))

        # коэффициент обучения
        self.lr = learningrate

        # использование сигмоиды в качестве функции активации
        self.activation_function = lambda x: scipy.special.expit(x)

    pass

    # тренировка нейронной сети
    def train(self, inputs_list, targets_list):
        # преобразование списка входных значений
        # в двухмерный массив
        inputs = numpy.array(inputs_list, ndmin=2).T
        targets = numpy.array(targets_list, ndmin=2).T

        # рассчитать входящие сигналы для скрытого слоя
        hidden_inputs = numpy.dot(self.wih, inputs)
        # рассчитать исходящие сигналы для скрытого слоя
        hidden_outputs = self.activation_function(hidden_inputs)
```

```

# рассчитать входящие сигналы для выходного слоя
final_inputs = numpy.dot(self.who, hidden_outputs)
# рассчитать исходящие сигналы для выходного слоя
final_outputs = self.activation_function(final_inputs)

# ошибки выходного слоя =
# (целевое значение - фактическое значение)
output_errors = targets - final_outputs
# ошибки скрытого слоя - это ошибки output_errors,
# распределенные пропорционально весовым коэффициентам связей
# и рекомбинированные на скрытых узлах
hidden_errors = numpy.dot(self.who.T, output_errors)

# обновить весовые коэффициенты для связей между
# скрытым и выходным слоями
self.who += self.lr * numpy.dot((output_errors *
final_outputs * (1.0 - final_outputs)),
numpy.transpose(hidden_outputs))

# обновить весовые коэффициенты для связей между
# входным и скрытым слоями
self.wih += self.lr * numpy.dot((hidden_errors *
hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

pass

# опрос нейронной сети
def query(self, inputs_list):
    # преобразовать список входных значений
    # в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)
    # рассчитать исходящие сигналы для скрытого слоя
    hidden_outputs = self.activation_function(hidden_inputs)

    # рассчитать входящие сигналы для выходного слоя
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # рассчитать исходящие сигналы для выходного слоя
    final_outputs = self.activation_function(final_inputs)

    return final_outputs

```

Здесь не так много кода, особенно если принять во внимание, что он может быть использован с целью создания, тренировки и опроса трехслойной нейронной сети практически для любой задачи.

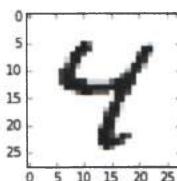
Далее мы займемся решением одной конкретной задачи: обучение нейронной сети распознаванию рукописных цифр.

## Набор рукописных цифр MNIST

Распознавание текста, написанного от руки, — это настоящий вызов для испытания возможностей искусственного интеллекта, поскольку эта проблема действительно трудна и размыта. Она не столь ясная и четко определенная, как перемножение одного множества чисел на другое.

Корректная классификация содержимого изображений с помощью компьютера, которую часто называют **распознаванием образов**, десятилетиями выдерживала атаки, направленные на ее разрешение. В последнее время в этой области наблюдается значительный прогресс, и решающую роль в наметившемся прорыве сыграли технологии нейронных сетей.

О трудности проблемы можно судить хотя бы по тому, что даже мы, люди, иногда не можем договориться между собой о том, какое именно изображение мы видим. В частности, предметом спора может послужить и написанная неразборчивым почерком буква. Взгляните на следующую цифру, написанную от руки. Что вы видите: 4 или 9?



Существует коллекция изображений рукописных цифр, используемых исследователями искусственного интеллекта в качестве популярного набора для тестирования идей и алгоритмов. То, что коллекция известна и пользуется популярностью, означает, что никому не составит труда проверить, как выглядит самая последняя из его безумных идей по сравнению с идеями других людей (т.е. различные идеи и алгоритмы тестируются с использованием одного и того же тестового набора).

Этим тестовым набором является база данных рукописных цифр под названием “MNIST”, предоставляемая авторитетным исследователем нейронных сетей Яном Лекуном для бесплатного всеобщего доступа по адресу <http://yann.lecun.com/exdb/mnist/>. Там же вы найдете сведения относительно успешности прежних и нынешних попыток корректного распознавания этих рукописных символов. Мы будем неоднократно обращаться к этому списку, чтобы проверить, в какой степени наши собственные идеи конкурентоспособны по сравнению с идеями, разрабатываемыми профессионалами!

Формат базы данных MNIST не относится к числу тех, с которыми легко работать, но, к счастью, другие специалисты создали соответствующие файлы в более простом формате (см., например, <http://pjreddie.com/projects/mnist-in-csv/>). Это так называемые CSV-файлы, в которых отдельные значения представляют собой обычный текст и разделены запятыми. Их содержимое можно легко просматривать в любом текстовом редакторе, и большинство электронных таблиц или программ, предназначенных для анализа данных, могут работать с CSV-файлами. Это довольно универсальный формат. На указанном сайте представлены следующие два файла:

- тренировочный набор ([http://www.pjreddie.com/media/files/mnist\\_train.csv](http://www.pjreddie.com/media/files/mnist_train.csv));
- тестовый набор ([http://www.pjreddie.com/media/files/mnist\\_test.csv](http://www.pjreddie.com/media/files/mnist_test.csv)).

**Тренировочный набор** содержит **60 000** промаркированных экземпляров, используемых для тренировки нейронной сети. Слово “промаркованные” означает, что для каждого экземпляра указан соответствующий правильный ответ.

**Меньший тестовый набор**, включающий **10 000** экземпляров, используется для проверки правильности работы идей или алгоритмов. Он также содержит корректные маркеры, позволяющие увидеть, способна ли наша нейронная сеть дать правильный ответ.

Использование независимых друг от друга наборов тренировочных и тестовых данных гарантирует, что с тестовыми данными нейронная сеть ранее не сталкивалась. В противном случае можно было бы схитрить и просто запомнить тренировочные данные, чтобы получить наибольшие, хотя и заработанные обманным путем, баллы.



Таким образом, первая запись представляет цифру “5”, о чем говорит первое значение, тогда как остальная часть текста в этой строке — это пиксельные значения написанной кем-то от руки цифры “5”. Вторая строка представляет рукописную цифру “0”, третья — цифру “4”, четвертая — цифру “1” и пятая — цифру “9”. Можете выбрать любую строку из файлов данных MNIST, и ее первое число укажет вам маркер для последующих данных изображения.

Однако увидеть, каким образом длинный список из 784 значений формирует изображение рукописной цифры “5”, не так-то просто. Мы должны вывести в графическом виде эти цифры и убедиться в том, что они действительно представляют цвета пикселей рукописной цифры.

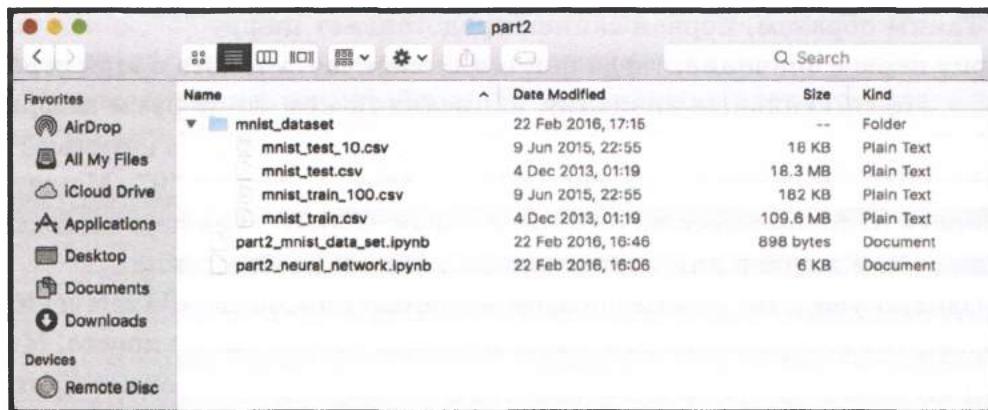
Прежде чем углубиться в рассмотрение деталей и приступить к написанию кода, загрузим небольшое подмножество набора данных, содержащихся в базе MNIST. Файлы данных MNIST имеют очень большой размер, тогда как работать с небольшими наборами значительно удобнее, поскольку это позволит нам экспериментировать, разрабатывать и испытывать свой код, что было бы затруднительно в случае длительных расчетов из-за большого объема обрабатываемых данных. Когда бы отработаем алгоритм и будем удовлетворены созданным кодом, можно будет вернуться к полному набору данных.

Ниже приведены ссылки на меньшие наборы MNIST, также подготовленные в формате CSV.

- **10 записей из тестового набора данных MNIST:**  
[https://raw.githubusercontent.com/makeyourownneurallnetwork/makeyourownneurallnetwork/master/mnist\\_dataset/mnist\\_test\\_10.csv](https://raw.githubusercontent.com/makeyourownneurallnetwork/makeyourownneurallnetwork/master/mnist_dataset/mnist_test_10.csv)
- **100 записей из тренировочного набора данных MNIST:**  
[https://raw.githubusercontent.com/makeyourownneurallnetwork/makeyourownneurallnetwork/master/mnist\\_dataset/mnist\\_train\\_100.csv](https://raw.githubusercontent.com/makeyourownneurallnetwork/makeyourownneurallnetwork/master/mnist_dataset/mnist_train_100.csv)

Если ваш браузер отображает данные вместо того, чтобы загрузить их автоматически, можете сохранить файл, выполнив команду меню Файл→Сохранить как или эквивалентную ей.

Сохраните файл в локальной папке, выбрав ее по своему усмотрению. Я храню свои данные в папке `mnist_dataset` вместе со своими блокнотами IPython, как показано на приведенном ниже снимке экрана. Сохранение блокнотов IPython и файлов данных в случайно выбираемых местах вносит в работу беспорядок.



Прежде чем с этими данными можно будет что-либо сделать, например построить график или обучить с их помощью нейронную сеть, необходимо обеспечить доступ к ним из кода на языке Python.

Открытие файлов и получение их содержимого в Python не составляет большого труда. Лучше всего показать, как это делается, на конкретном примере. Взгляните на следующий код.

```
data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
data_list = data_file.readlines()
data_file.close()
```

Здесь всего три строки кода. Обсудим каждую из них по отдельности.

Первая строка открывает файл с помощью функции `open()`. Вы видите, что в качестве первого из параметров ей передается имя файла. На самом деле это не просто имя файла `mnist_train_100.csv`, а весь путь доступа к нему, включая каталог (папку), в котором он находится. Второй параметр необязательный и сообщает Python, как мы хотим работать с файлом. Буква “`r`” означает, что мы хотим открыть файл только для чтения, а не для записи. Тем самым мы предотвращаем любое непреднамеренное изменение или удаление файла. Если мы попытаемся осуществить запись в этот файл или изменить его, Python не позволит этого сделать и выдаст ошибку. А что это за переменная `data_file`? Функция `open()` создает дескриптор (указатель), играющий роль ссылки на открываемый файл, и мы присваиваем его переменной `data_file`. Когда файл открыт, любые последующие действия с ним, например чтение, осуществляются через этот дескриптор.

Следующая строка проще. Мы используем функцию `readlines()`, ассоциированную с дескриптором файла `data_file`, для чтения всех строк содержимого файла в переменную `data_list`. Эта переменная содержит список, каждый элемент которого является строкой, представляющей строку файла. Это очень удобно, поскольку мы можем переходить к любой строке файла так же, как к конкретным записям в списке. Таким образом, `data_list[0]` — это первая запись, а `data_list[9]` — десятая и т.п.

Кстати, вполне возможно, что некоторые люди не рекомендовали вам использовать функцию `readlines()`, поскольку она считывает весь файл в память. Наверное, они советовали вам считывать по одной строке за раз, выполнять всю необходимую обработку для этой строки и лишь затем переходить к следующей. Нельзя сказать, что они неправы. Действительно, гораздо эффективнее работать поочередно с каждой строкой, а не считывать в память весь файл целиком. Однако наши файлы невелики, и использование функции `readlines()` значительно упрощает код, а для нас простота и ясность очень важны, поскольку мы изучаем Python.

Последняя строка закрывает файл. Считается хорошей практикой закрывать файлы и другие ресурсы после их использования. Если же оставлять их открытыми, то это может приводить к возникновению проблем. Что за проблемы? Некоторые программы могут отказываться осуществлять запись в файл, открытый в другом месте, если это приводит к несогласованности. В противном случае это напоминало бы ситуацию, когда два человека пытаются одновременно записывать разные тексты на одном и том же листе бумаги. Иногда компьютер может заблокировать файл во избежание такого рода конфликтов. Если не оставить за собой порядок после того, как необходимость в использовании файла уже отпала, у вас может накопиться много заблокированных файлов. По крайней мере, закрыв файл, вы предоставите компьютеру возможность освободить память, занимаемую уже ненужными данными.

Создайте новый пустой блокнот, выполните представленный ниже код и посмотрите, что произойдет, если попытаться вывести элементы списка.



- проигнорировать первое значение, являющееся маркером, извлечь оставшиеся  $28 \times 28 = 784$  значения и преобразовать их в массив, состоящий из 28 строк и 28 столбцов;
- отобразить массив!

И вновь, проще всего показать соответствующий простой код на Python и уже после этого подробно рассмотреть, что в нем происходит.

Прежде всего, мы не должны забывать о необходимости импортировать библиотеки расширений Python для работы с массивами и графикой.

```
import numpy
import matplotlib.pyplot
%matplotlib inline
```

А теперь взгляните на следующие три строки кода. Переменные окрашены различными цветами таким образом, чтобы было понятно, где и какие данные используются.

```
all_values = data_list[0].split(',')
image_array = numpy.asarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys',
                        interpolation='None')
```

В первой строке длинная первая запись `data_list[0]`, которую мы только что выводили, разбивается на отдельные значения с использованием запятой в качестве разделителя. Это делается с помощью функции `split()`, параметр которой определяет символ-разделитель. Результат помещается в переменную `all_values`. Можете вывести эти значения на экран и убедиться в том, что эта переменная действительно содержит нужные значения в виде длинного списка Python.

Следующая строка кода выглядит чуть более сложной, поскольку в ней происходит сразу несколько вещей. Начнем с середины. Запись списка в виде `all_values[1:]` указывает на то, что берутся все элементы списка за исключением первого. Тем самым мы игнорируем первое значение, играющее роль маркера, и берем лишь остальные 784 элемента. `numpy.asarray()` — это функция библиотеки `numpy`, преобразующая текстовые строки в реальные числа и создающая массив этих чисел. Постойте-ка, но почему текстовые

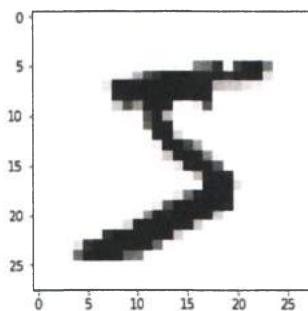
строки преобразуются в числа? Если файл был прочитан как текстовый, то каждая строка или запись все еще остается текстом. Извлечение из них отдельных элементов, разделенных запятыми, также дает текстовые элементы. Например, этим текстом могли бы быть слова “apple”, “orange123” или “567”. Текстовая строка “567” — это не то же самое, что число 567. Именно поэтому мы должны преобразовывать текстовые строки в числа, даже если эти строки выглядят как числа. Последний фрагмент инструкции `.reshape((28,28))` — гарантирует, что список будет сформирован в виде квадратной матрицы размером  $28 \times 28$ . Результирующий массив такой же размерности получает название `image_array`. Ух! Как много интересного всего лишь в одной строке кода!

Третья строка просто выводит на экран массив `image_array` с помощью функции `imshow()`, аналогично тому, с чем вы уже сталкивались. На этот раз мы выбрали цветовую палитру оттенков серого с помощью параметра `cmap='Greys'` для лучшей различимости рукописных символов.

Результат работы кода представлен ниже.

```
In [8]: all_values = data_list[0].split(',')
image_array = numpy.asarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

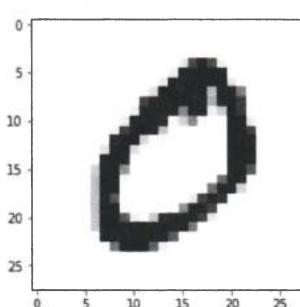
Out[8]: <matplotlib.image.AxesImage at 0x814bb70>
```



Вы видите графическое изображение цифры “5”, на что и указывал маркер. Если мы выберем следующую запись, `data_list[1]` с маркером 0, то получим показанное ниже изображение.

```
In [9]: all_values = data_list[1].split(',')
image_array = numpy.asarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys', interpolation='None')

Out[9]: <matplotlib.image.AxesImage at 0x82449e8>
```



Глядя на это изображение, вы с уверенностью можете сказать, что этой рукописной цифре действительно соответствует цифра 0.

## Подготовка тренировочных данных MNIST

Теперь, когда вы уже знаете, как получить данные из файлов MNIST и извлечь из них нужные записи, мы можем воспользоваться этим для визуализации данных. Мы хотим использовать эти данные для обучения нашей нейронной сети, но сначала мы должны продумать, как их следует подготовить, прежде чем предоставлять сети.

Вы уже видели, что нейронные сети работают лучше, если как входные, так и выходные данные конфигурируются таким образом, чтобы они оставались в диапазоне значений, оптимальном для функций активации узлов нейронной сети.

Первое, что мы должны сделать, — это перевести значения цветовых кодов из большего диапазона значений 0–255 в намного меньший, охватывающий значения от 0,01 до 1,0. Мы намеренно выбрали значение 0,01 в качестве нижней границы диапазона, чтобы избежать упомянутых ранее проблем с нулевыми входными значениями, поскольку они могут искусственно блокировать обновление весов. Нам необязательно выбирать значение 0,99 в качестве верхней границы допустимого диапазона, поскольку нет нужды избегать значений 1,0 для входных сигналов. Лишь выходные сигналы не могут превышать значение 1,0.

Деление исходных входных значений, изменяющихся в диапазоне 0–255, на 255 приведет их к диапазону 0–1,0. Последующее

умножение этих значений на коэффициент 0,99 приведет их к диапазону 0,0–0,99. Далее мы инкрементируем их на 0,01, чтобы вместить их в желаемый диапазон 0,01–1,0. Все эти действия реализует следующий код на языке Python.

```
scaled_input = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
print(scaled_input)
```

Результирующий вывод данных подтверждает, что они действительно принадлежат к диапазону значений от 0,01 до 1,00.

```
In [10]: # привести входные значения к диапазону 0,01 - 1,00
scaled_input = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
print(scaled_input)
```

```
[ 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.01  0.01  0.01  0.01  0.01  0.01
 0.01  0.208  0.62729412  0.99223529  0.62729412  0.20411765
 0.01  0.01  0.01  0.01  0.01  0.01  0.01]
```

Итак, мы осуществили подготовку данных MNIST путем их масштабирования и сдвига и теперь можем подавать их на вход нашей нейронной сети как с целью ее тренировки, так и с целью опроса.

На этом этапе нам также нужно продумать, что делать с выходными значениями нейронной сети. Ранее вы видели, что выходные значения должны укладываться в диапазон значений, обеспечиваемый функцией активации. Используемая нами логистическая функция не может выдавать такие значения, как –2,0 или 255. Ее значения охватывают диапазон чисел от 0,0 до 1,0, а фактически вы никогда не получите значений 0,0 или 1,0, поскольку логистическая функция не может их достигать и лишь асимптотически приближается к ним. Таким образом, по-видимому, нам придется масштабировать выходные значения в процессе тренировки сети.

Но вообще-то, мы должны задать самим себе вопрос более глубокого содержания. Что именно мы должны получить на выходе нейронной сети? Должно ли это быть изображение ответа? В таком случае нам нужно иметь  $28 \times 28 = 784$  выходных узла.

Если мы вернемся на шаг назад и задумаемся над тем, чего именно мы хотим от нейронной сети, то поймем, что мы просим ее классифицировать изображение и присвоить ему корректный маркер. Таким маркером может быть одно из десяти чисел в диапазоне от 0 до 9. Это означает, что выходной слой сети должен иметь 10 узлов, по одному на каждый возможный ответ, или маркер. Если ответом является “0”, то активизироваться должен первый узел, тогда как остальные узлы должны оставаться пассивными. Если ответом является “9”, то активизироваться должен последний узел выходного слоя при пассивных остальных узлах. Следующая иллюстрация поясняет эту схему на нескольких примерах выходных значений.

выходной слой	маркер	пример “5”	пример “0”	пример “9”
0	0	0.00	0.95	0.02
1	1	0.00	0.00	0.00
2	2	0.01	0.01	0.01
3	3	0.00	0.01	0.01
4	4	0.01	0.02	0.40
5	5	0.99	0.00	0.01
6	6	0.00	0.00	0.01
7	7	0.00	0.00	0.00
8	8	0.02	0.00	0.01
9	9	0.01	0.02	0.86

Первый пример соответствует случаю, когда сеть распознала входные данные как цифру “5”. Вы видите, что наибольший из исходящих сигналов выходного слоя принадлежит узлу с меткой “5”. Не

забывайте о том, что это шестой по счету узел, потому что нумерация узлов начинается с нуля. Все довольно просто. Остальные узлы производят сигналы небольшой величины, близкие к нулю. Вывод нулей мог оказаться следствием ошибок округления, но в действительности, как вы помните, функция активации никогда не допустит фактическое нулевое значение.

Следующий пример соответствует рукописному “0”. Наибольшую величину здесь имеет сигнал первого выходного узла, ассоциируемый с меткой “0”.

Последний пример более интересен. Здесь самый большой сигнал генерирует последний узел, соответствующий метке “9”. Однако и узел с меткой “4” дает сигнал средней величины. Обычно нейронная сеть должна принимать решение, основываясь на наибольшем сигнале, но, как видите, в данном случае она отчасти считает, что правильным ответом могло бы быть и “4”. Возможно, рукописное начертание символа затруднило его надежное распознавание? Такого рода неопределенности встречаются в нейронных сетях, и вместо того, чтобы считать их неудачей, мы должны рассматривать их как полезную подсказку о существовании другого возможного ответа.

Отлично! Теперь нам нужно превратить эти идеи в целевые массивы для тренировки нейронной сети. Как вы могли убедиться, если тренировочный пример помечен маркером “5”, то для выходного узла следует создать такой целевой массив, в котором малы все элементы, кроме одного, соответствующего маркеру “5”. В данном случае этот массив мог бы выглядеть примерно так: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].

В действительности эти числа нуждаются в дополнительном масштабировании, поскольку мы уже видели, что попытки создания на выходе нейронной сети значений 0 и 1, недостижимых в силу использования функции активации, приводят к большим весам и насыщению сети. Следовательно, вместо этого мы будем использовать значения 0,01 и 0,99, и потому целевым массивом для маркера “5” должен быть массив [0.01, 0.01, 0.01, 0.01, 0.01, 0.99, 0.01, 0.01, 0.01, 0.01].

А вот как выглядит код на языке Python, создающий целевую матрицу.

```
# количество выходных узлов - 10 (пример)
onodes = 10
targets = numpy.zeros(onodes) + 0.01
targets[int(all_values[0])] = 0.99
```

Первая строка после комментария просто устанавливает количество выходных узлов равным 10, что соответствует нашему примеру с десятью маркерами.

Во второй строке с помощью удобной функции `numpy.zeros()` создается массив, заполненный нулями. Желаемые размер и конфигурация массива задаются параметром при вызове функции. В данном случае создается одномерный массив, размер `onodes` которого равен количеству узлов в конечном выходном слое. Проблема нулей, которую мы только что обсуждали, устраняется путем добавления 0,01 к каждому элементу массива.

Следующая строка выбирает первый элемент записи из набора данных MNIST, являющийся целевым маркером тренировочного набора, и преобразует его в целое число. Вспомните о том, что запись читается из исходного файла в виде текстовой строки, а не числа. Как только преобразование выполнено, полученный целевой маркер используется для того, чтобы установить значение соответствующего элемента массива равным 0,99. Здесь все будет нормально работать, поскольку маркер “0” будет преобразован в целое число 0, являющееся корректным индексом данного маркера в массиве `targets[]`. Точно так же маркер “9” будет преобразован в целое число 9, и элемент `targets[9]` действительно является последним элементом этого массива.

Вот пример работы этого кода.

```
In [11]: # количество выходных узлов - 10 (пример)
onodes = 10
targets = numpy.zeros(onodes) + 0.01
targets[int(all_values[0])] = 0.99

In [12]: print(targets)
[ 0.99  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01]
```

Великолепно! Теперь мы знаем, как подготовить входные значения для тренировки и опроса нейронной сети, а выходные значения — для тренировки.

Обновим наш код с учетом проделанной работы. Ниже представлено состояние кода на данном этапе, включая последнее обновление. Вы также можете в любой момент получить его на сайте GitHub, используя следующую ссылку, в то время как мы продолжим его дальнейшую разработку:

- [https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2\\_neural\\_network\\_mnist\\_data.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data.ipynb)

Вы также можете ознакомиться с тем, как постепенно улучшался этот код, воспользовавшись следующей ссылкой:

- [https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2\\_neural\\_network\\_mnist\\_data.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2_neural_network_mnist_data.ipynb)

---

```
# Блокнот Python для книги "Создаем нейронную сеть".
# Код для создания 3-слойной нейронной сети вместе с
# кодом для ее обучения с помощью набора данных MNIST.
# (c) Tariq Rashid, 2016
# лицензия GPLv2

import numpy
# библиотека scipy.special содержит сигмоиду expit()
import scipy.special
# библиотека для графического отображения массивов
import matplotlib.pyplot
# гарантировать размещение графики в данном блокноте,
# а не в отдельном окне
%matplotlib inline

# определение класса нейронной сети
class neuralNetwork:

    # инициализировать нейронную сеть
    def __init__(self, inputnodes, hiddennodes, outputnodes,
    learningrate):
        # задать количество узлов во входном, скрытом и выходном слое
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # Матрицы весовых коэффициентов связей, wih и who.
        # Весовые коэффициенты связей между узлом i и узлом j
        # следующего слоя обозначены как w_i_j:
        # w11 w21
        # w12 w22 и т.д.
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
    (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
    (self.onodes, self.hnodes))

        # коэффициент обучения
        self.lr = learningrate
```

```

# использование сигмоиды в качестве функции активации
self.activation_function = lambda x: scipy.special.expit(x)

pass

# тренировка нейронной сети
def train(self, inputs_list, targets_list):
    # преобразование списка входных значений
    # в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)
    # рассчитать исходящие сигналы для скрытого слоя
    hidden_outputs = self.activation_function(hidden_inputs)

    # рассчитать входящие сигналы для выходного слоя
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # рассчитать исходящие сигналы для выходного слоя
    final_outputs = self.activation_function(final_inputs)

    # ошибки выходного слоя =
    # (целевое значение - фактическое значение)
    output_errors = targets - final_outputs
    # ошибки скрытого слоя - это ошибки output_errors,
    # распределенные пропорционально весовым коэффициентам связей
    # и рекомбинированные на скрытых узлах
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # обновить весовые коэффициенты для связей между
    # скрытым и выходным слоями
    self.who += self.lr * numpy.dot((output_errors *
        final_outputs * (1.0 - final_outputs)),
        numpy.transpose(hidden_outputs))

    # обновить весовые коэффициенты для связей между
    # входным и скрытым слоями
    self.wih += self.lr * numpy.dot((hidden_errors *
        hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

pass

# опрос нейронной сети
def query(self, inputs_list):
    # преобразовать список входных значений
    # в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)

```

```

# рассчитать исходящие сигналы для скрытого слоя
hidden_outputs = self.activation_function(hidden_inputs)

# рассчитать входящие сигналы для выходного слоя
final_inputs = numpy.dot(self.who, hidden_outputs)
# рассчитать исходящие сигналы для выходного слоя
final_outputs = self.activation_function(final_inputs)

return final_outputs

# количество входных, скрытых и выходных узлов
input_nodes = 784
hidden_nodes = 100
output_nodes = 10

# коэффициент обучения равен 0,3
learning_rate = 0.3

# создать экземпляр нейронной сети
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)

# загрузить в список тестовый набор данных CSV-файла набора MNIST
training_data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# тренировка нейронной сети

# перебрать все записи в тренировочном наборе данных
for record in training_data_list:
    # получить список значений, используя символы запятой ','
    # в качестве разделителей
    all_values = record.split(',')
    # масштабировать и сместить входные значения
    inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # создать целевые выходные значения (все равны 0,01, за исключением
    # желаемого маркерного значения, равного 0,99)
    targets = numpy.zeros(output_nodes) + 0.01

    # all_values[0] - целевое маркерное значение для данной записи
    targets[int(all_values[0])] = 0.99
    n.train(inputs, targets)
    pass

```

В самом начале этого кода мы импортируем графическую библиотеку, добавляем код для задания размеров входного, скрытого и выходного слоев, считываем малый тренировочный набор данных MNIST, а затем тренируем нейронную сеть с использованием этих записей.

Почему мы выбрали 784 входных узла? Вспомните о том, что это число равно произведению  $28 \times 28$ , представляющему количество пикселей, из которых состоит изображение рукописной цифры.

Выбор ста скрытых узлов не имеет столь же строгого научного обоснования. Мы не выбрали это число большим, чем 784, из тех соображений, что нейронная сеть должна находить во входных значениях такие особенности или шаблоны, которые можно выразить в более короткой форме, чем сами эти значения. Поэтому, выбирая количество узлов меньшим, чем количество входных значений, мы заставляем сеть пытаться находить ключевые особенности путем обобщения информации. В то же время, если выбрать количество скрытых узлов слишком малым, будут ограничены возможности сети в отношении определения достаточного количества отличительных признаков или шаблонов в изображении. Тем самым мы лишили бы сеть возможности выносить собственные суждения относительно данных MNIST. С учетом того, что выходной слой должен обеспечивать вывод 10 маркеров, а значит, должен иметь десять узлов, выбор промежуточного значения 100 для количества узлов скрытого слоя представляется вполне разумным.

В связи с этим следует сделать одно важное замечание: идеального общего метода для выбора количества скрытых узлов не существует. В действительности не существует и идеального метода выбора количества скрытых слоев. В настоящее время наилучшим подходом является проведение экспериментов до тех пор, пока не будет получена конфигурация сети, оптимальная для задачи, которую вы пытаетесь решить.

## Тестирование нейронной сети

Справившись с тренировкой сети, по крайней мере на небольшом подмножестве из ста записей, мы должны проверить, как она работает, и сделаем это, используя тестовый набор данных.

Прежде всего, необходимо получить тестовые записи. Соответствующий код очень похож на тот, который мы использовали для получения тренировочных данных.

```
# загрузить в список тестовый набор данных CSV-файла набора MNIST
test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

Мы распакуем эти данные точно так же, как и предыдущие, поскольку они имеют аналогичную структуру.

Прежде чем создавать цикл для перебора всех тестовых записей, посмотрим, что произойдет, если мы вручную выполним одиночный тест. Ниже представлены результаты опроса уже обученной нейронной сети, выполненного с использованием первой записи тестового набора данных.

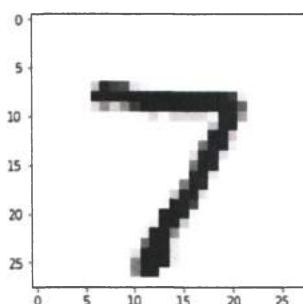
```
In [2]: # загрузить в список тестовый набор данных CSV-файла набора MNIST
test_data_file = open("mnist_dataset/mnist_test_10.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

```
In [3]: # получить первую тестовую запись
all_values = test_data_list[0].split(',')
# добавить маркер
print(all_values[0])
```

7

```
In [4]: image_array = numpy.asarray(all_values[1:]).reshape((28,28))
matplotlib.pyplot.imshow(image_array, cmap='Greys',
interpolation='None')
```

```
Out[4]: <matplotlib.image.AxesImage at 0x8c6e080>
```



```
In [5]: n.query((numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01)
```

```
Out[5]: array([[ 0.03305615],
 [ 0.00522053],
 [ 0.01038686],
 [ 0.07915526],
 [ 0.0167299 ],
 [ 0.02322546],
 [ 0.00498213],
 [ 0.79226727],
 [ 0.01799863],
 [ 0.01735059]])
```

Как видите, в качестве маркера первой записи тестового набора сеть определила символ “7”. Именно этого ответа мы ожидали, опрашивая ее.

Графическое отображение пиксельных значений подтверждает, что рукописной цифрой действительно является цифра “7”.

В результате опроса обученной сети мы получаем список чисел, являющихся выходными значениями каждого из выходных узлов. Сразу же бросается в глаза, что одно из выходных значений намного превышает остальные, и этому значению соответствует маркер “7”. Это восьмой элемент списка, поскольку первому элементу соответствует маркер “0”.

У нас все сработало!

Этот момент заслуживает того, чтобы мы им насладились, и полностью окупает все затраченные нами до сих пор усилия!

Мы обучили нашу нейронную сеть и добились того, что она смогла определить цифру, предоставленную ей в виде изображения. Вспомните, что до этого сеть не сталкивалась с данным изображением, поскольку оно не входило в тренировочный набор данных. Следовательно, нейронная сеть оказалась в состоянии корректно классифицировать незнакомый ей цифровой символ. Это поистине впечатляющий результат!

С помощью всего лишь нескольких строк кода на языке Python мы создали нейронную сеть, способную делать то, что многие люди сочли бы проявлением искусственного интеллекта, — распознавать изображения цифр, написанных рукой человека.

Этот результат впечатляет еще больше, если учесть, что для обучения сети была использована ничтожно малая часть полного набора тренировочных данных. Вспомните, что этот набор включает 60 тысяч записей, а мы использовали только 100 из них. У меня даже были сомнения относительно того, что у нас вообще что-либо получится!

Продолжим в том же духе и напишем код, позволяющий проверить, насколько хорошо нейронная сеть справляется с остальной частью набора данных, и провести подсчет правильных результатов, чтобы впоследствии мы могли оценивать плодотворность своих будущих идей по совершенствованию способности сети учиться, а также сравнивать наши результаты с результатами, полученными другими людьми.

Сначала ознакомьтесь с приведенным ниже кодом, а затем мы его обсудим.

```
# тестирование нейронной сети

# журнал оценок работы сети, первоначально пустой
scorecard = []

# перебрать все записи в тестовом наборе данных
for record in test_data_list:
    # получить список значений из записи, используя символы
    # запятой (',') в качестве разделителей
    all_values = record.split(',')
    # правильный ответ - первое значение
    correct_label = int(all_values[0])
    print(correct_label, "истинный маркер")
    # масштабировать и сместить входные значения
    inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # опрос сети
    outputs = n.query(inputs)
    # индекс наибольшего значения является маркерным значением
    label = numpy.argmax(outputs)
    print(label, "ответ сети")
    # присоединить оценку ответа сети к концу списка
    if (label == correct_label):
        # в случае правильного ответа сети присоединить
        # к списку значение 1
        scorecard.append(1)
    else:
        # в случае неправильного ответа сети присоединить
        # к списку значение 0
        scorecard.append(0)
    pass

pass
```

Прежде чем войти в цикл, обрабатывающий все записи тестового набора данных, мы создаем пустой список **scorecard**, который будет служить нам журналом оценок работы сети, обновляемым после обработки каждой записи.

В цикле мы делаем то, что уже делали раньше: извлекаем значения из текстовой записи, в которой они разделены запятыми. Первое значение, указывающее правильный ответ, сохраняется в отдельной переменной. Остальные значения масштабируются, чтобы их можно было использовать в качестве входных данных для передачи запроса нейронной сети. Ответ нейронной сети сохраняется в переменной **outputs**.

Далее следует довольно интересная часть кода. Мы знаем, что наибольшее из значений выходных узлов рассматривается сетью

в качестве правильного ответа. Индекс этого узла, т.е. его позиция, соответствует маркеру. Эта фраза просто означает, что первый элемент соответствует маркеру “0”, пятый — маркеру “4” и т.д. К счастью, существует удобная функция библиотеки `numpy`, которая находит среди элементов массива максимальное значение и сообщает его индекс. Это функция `numpy.argmax()`. Для получения более подробных сведений о ней можете посетить веб-страницу по следующему адресу:

<https://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.argmax.html>

Возврат этой функцией значения 0 означает, что правильным ответом сеть считает “0” и т.д.

В последнем фрагменте кода полученный маркер сравнивается с известным корректным маркером. Если оба маркера одинаковы, в журнал записывается “1”, в противном случае — “0”.

Кроме того, я включил в некоторых местах кода полезную команду `print()`, чтобы мы сами могли отслеживать правильные и предсказываемые значения. Ниже представлены результаты выполнения этого кода вместе с выведенными записями нашего рабочего журнала.

```
7 истинный маркер
7 ответ сети
2 истинный маркер
0 ответ сети
1 истинный маркер
1 ответ сети
0 истинный маркер
0 ответ сети
4 истинный маркер
4 ответ сети
1 истинный маркер
1 ответ сети
4 истинный маркер
4 ответ сети
9 истинный маркер
3 ответ сети
5 истинный маркер
1 ответ сети
9 истинный маркер
7 ответ сети
```

In [8]: `print(scorecard)`

```
[1, 0, 1, 1, 1, 1, 1, 0, 0, 0]
```

На этот раз не все у нас хорошо! Вы видите, что имеется несколько несовпадений. Последняя выведенная строка результатов показывает,

что из десяти тестовых записей правильно были распознаны 6. Таким образом, доля правильных результатов составила 60%. На самом деле это не так уж и плохо, если принять во внимание небольшой размер тренировочного набора, который мы использовали.

Дополним код фрагментом, который будет выводить относительную долю правильных ответов в виде дроби.

```
# рассчитать показатель эффективности в виде
# доли правильных ответов
scorecard_array = numpy.asarray(scorecard)
print ("эффективность = ", scorecard_array.sum() /
    scorecard_array.size)
```

Эта доля рассчитывается как количество всех записей в журнале, содержащих “1”, деленное на общее количество записей (размер журнала). Вот каким получается результат.

```
In [8]: print(scorecard)
[1, 0, 1, 1, 1, 1, 0, 0, 0]

In [9]: # рассчитать показатель эффективности в виде
# доли правильных ответов
scorecard_array = numpy.asarray(scorecard)
print ("эффективность = ", scorecard_array.sum() / scorecard_array.size)
эффективность = 0.6
```

Как и ожидалось, мы получили показатель эффективности сети, равный 0,6, или 60%.

## Тренировка и тестирование нейронной сети с использованием полной базы данных

Далее мы добавим в нашу основную программу новый код, разработанный для тестирования эффективности сети.

При этом потребуется изменить имена файлов, поскольку теперь они должны указывать на полный набор тренировочных данных, насчитывающий 60 тысяч записей, и набор тестовых данных, насчитывающий 10 тысяч записей. Ранее мы сохранили эти наборы в файлах с именами `mnist_dataset/mnist_train.csv` и `mnist_dataset/mnist_test.csv`. Нам предстоит серьезная работа!

Не забывайте о том, что блокнот с этим кодом можно получить на сайте GitHub по следующему адресу:

[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2\\_neural\\_network\\_mnist\\_data.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network_mnist_data.ipynb)

Вы также можете ознакомиться с тем, как постепенно улучшался этот код, воспользовавшись следующей ссылкой:

[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2\\_neural\\_network\\_mnist\\_data.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/commits/master/part2_neural_network_mnist_data.ipynb)

В соответствии с результатами обучения нашей простой трехслойной нейронной сети с использованием полного набора, включающего 60 тысяч примеров, и последующего тестирования на 10 тысячах записей показатель общей эффективности сети составляет **0,9473**. Это очень неплохо. Точность распознавания составила почти 95%!

```
In [12]: # рассчитать показатель эффективности в виде
# доли правильных ответов
scorecard_array = numpy.asarray(scorecard)
print ("эффективность = ", scorecard_array.sum() / scorecard_array.size)
эффективность = 0.9473
```

Этот показатель, равный почти 95%, можно сравнить с аналогичными результатами эталонных тестов, которые можно найти по адресу <http://yann.lecun.com/exdb/mnist/>. Вы увидите, что в некоторых случаях наши результаты даже лучше эталонных и почти сравнимы с приведенными на указанном сайте результатами для простейшей нейронной сети, эффективность которой составила 95,3%.

Это вовсе не так плохо. Мы должны быть довольны тем, что наша первая попытка продемонстрировала эффективность на уровне нейронной сети профессиональных исследователей.

Кстати, вас не должно удивлять, что даже в случае современных быстродействующих домашних компьютеров обработка всех 60 тысяч тренировочных примеров, для каждого из которых необходимо вычислить распространение сигналов от 784 входных узлов через сто скрытых узлов в прямом направлении, а также обратное распространение ошибок и обновление весов, занимает ощутимое время.

На моем ноутбуке прохождение всего тренировочного цикла заняло около двух минут. Для вашего компьютера длительность вычислений может быть иной.

## Улучшение результатов: настройка коэффициента обучения

Получение 95%-го показателя эффективности при тестировании набора данных MNIST нашей нейронной сетью, созданной на основе простейших идей и с использованием простого кода на языке Python, — это совсем неплохо, и ваше желание остановиться на этом можно было бы считать вполне оправданным.

Однако мы попытаемся улучшить разработанный к этому моменту код, внеся в него некоторые усовершенствования.

Прежде всего, мы можем попытаться настроить коэффициент обучения. Перед этим мы задали его равным 0,3, даже не тестируя другие значения.

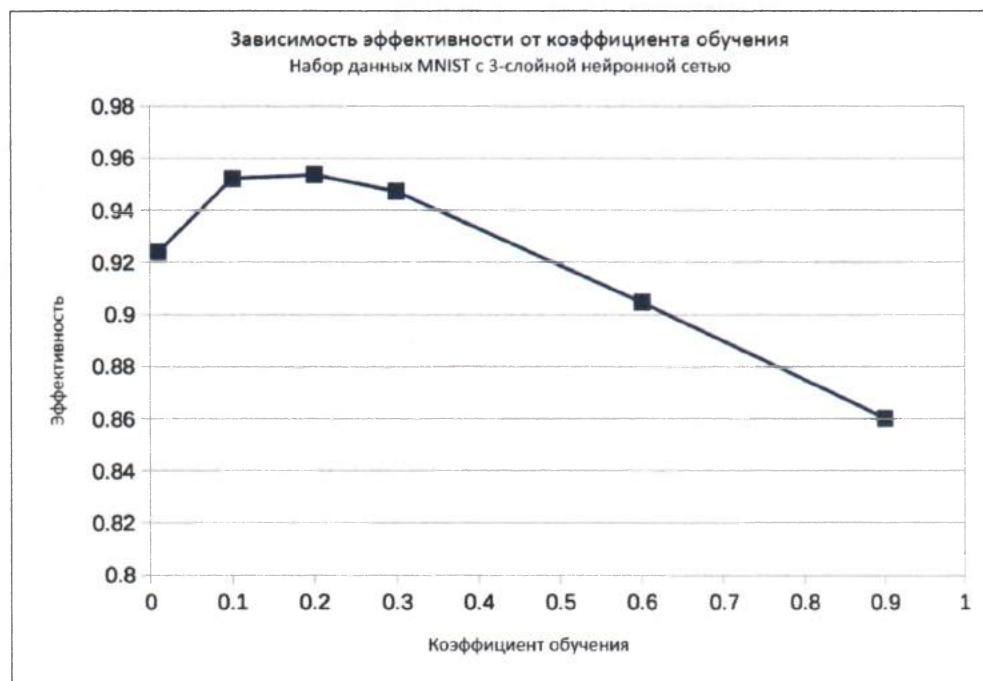
Давайте удвоим это значение до величины **0,6** и посмотрим, как это скажется на способности нейронной сети обучаться. Выполнение кода с таким значением коэффициента обучения дает показатель эффективности **0,9047**. Этот результат хуже прежнего. По-видимому, увеличение коэффициента обучения нарушает монотонность процесса минимизации ошибок методом градиентного спуска и сопровождается перескоками через минимум.

Повторим вычисления, установив коэффициент обучения равным **0,1**. На этот раз показатель эффективности улучшился до **0,9523**, что почти совпадает с результатом ранее упомянутого эталонного теста, который проводился с использованием тысячи скрытых узлов. Но ведь наш результат был получен с использованием намного меньшего количества узлов!

Что произойдет, если мы пойдем еще дальше и уменьшим коэффициент обучения до **0,01**? Оказывается, это также приводит к уменьшению показателя эффективности сети до **0,9241**. По-видимому, слишком малые значения коэффициента обучения снижают эффективность. Это представляется логичным, поскольку малые шаги уменьшают скорость градиентного спуска.

Описанные результаты представлены ниже в виде графика. Это не совсем научный подход, поскольку нам следовало бы провести такие

эксперименты множество раз, чтобы исключить фактор случайности и влияние неудачного выбора маршрутов градиентного спуска, но в целом он позволяет проиллюстрировать общую идею о существовании некоего оптимального значения коэффициента обучения.



Вид графика подсказывает нам, что оптимальным может быть значение в интервале от 0,1 до 0,3, поэтому испытаем значение **0,2**. Показатель эффективности получится равным **0,9537**. Мы видим, что этот результат немного лучше тех, которые мы имели при значениях коэффициента обучения, равных 0,1 или 0,3. Идею построения графиков вы должны взять на вооружение и в других сценариях, поскольку графики способствуют пониманию общих тенденций лучше, чем длинные числовые списки.

Итак, мы остановимся на значении коэффициента обучения 0,2, которое проявило себя как оптимальное для набора данных MNIST и нашей нейронной сети.

Кстати, если вы самостоятельно выполните этот код, то ваши оценки будут немного отличаться от приведенных здесь, поскольку процесс

в целом содержит элементы случайности. Ваш случайный выбор начальных значений весовых коэффициентов не будет совпадать с моим, а потому маршрут градиентного спуска для вашего кода будет другим.

## Улучшение результатов: многократное повторение тренировочных циклов

Нашим следующим усовершенствованием будет многократное повторение циклов тренировки с одним и тем же набором данных. В отношении одного тренировочного цикла иногда используют термин эпоха. Поэтому сеанс тренировки из десяти эпох означает десятикратный прогон всего тренировочного набора данных. А зачем нам это делать? Особенно если для этого компьютеру потребуется 10, 20 или даже 30 минут? Причина заключается в том, что тем самым мы обеспечиваем большее число маршрутов градиентного спуска, оптимизирующих весовые коэффициенты.

Посмотрим, что нам дадут две тренировочные эпохи. Для этого мы должны немного изменить код, предусмотрев в нем дополнительный цикл выполнения кода тренировки. В приведенном ниже коде внешний цикл выделен цветом, чтобы сделать его более заметным.

```
# тренировка нейронной сети

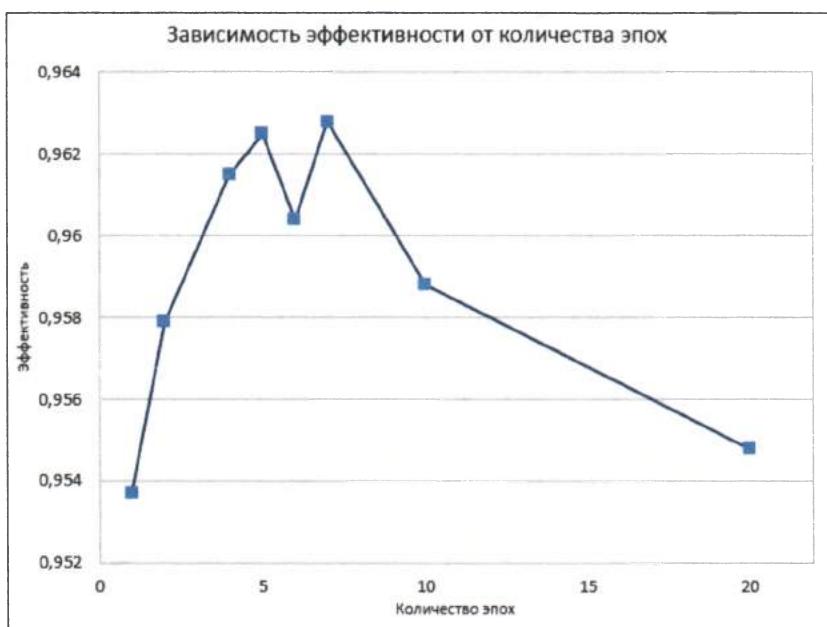
# переменная epochs указывает, сколько раз тренировочный
# набор данных используется для тренировки сети
epochs = 2

for e in range(epochs):
    # перебрать все записи в тренировочном наборе данных
    for record in training_data_list:
        # получить список значений из записи, используя символы
        # запятой (',') в качестве разделителей
        all_values = record.split(',')
        # масштабировать и сместить входные значения
        inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # создать целевые выходные значения (все равны 0,01, за
        # исключением желаемого маркерного значения, равного 0,99)
        targets = numpy.zeros(output_nodes) + 0.01
        # all_values[0] - целевое маркерное значение для
        # данной записи
        targets[int(all_values[0])] = 0.99
        n.train(inputs, targets)
        pass
    pass
```

Результирующий показатель эффективности для двух эпох составляет **0,9579**, что несколько лучше показателя для одной эпохи.

Подобно тому, как мы настраивали коэффициент обучения, проведем эксперимент с использованием различного количества эпох и построим график зависимости показателя эффективности от этого фактора. Интуиция подсказывает нам, что чем больше тренировок, тем выше эффективность. Но можно предположить, что слишком большое количество тренировок чревато ухудшением эффективности из-за так называемого **переобучения** сети на тренировочных данных, снижающего эффективность при работе с незнакомыми данными. Фактора переобучения следует опасаться в любых видах машинного обучения, а не только в нейронных сетях.

В данном случае мы имеем следующие результаты.



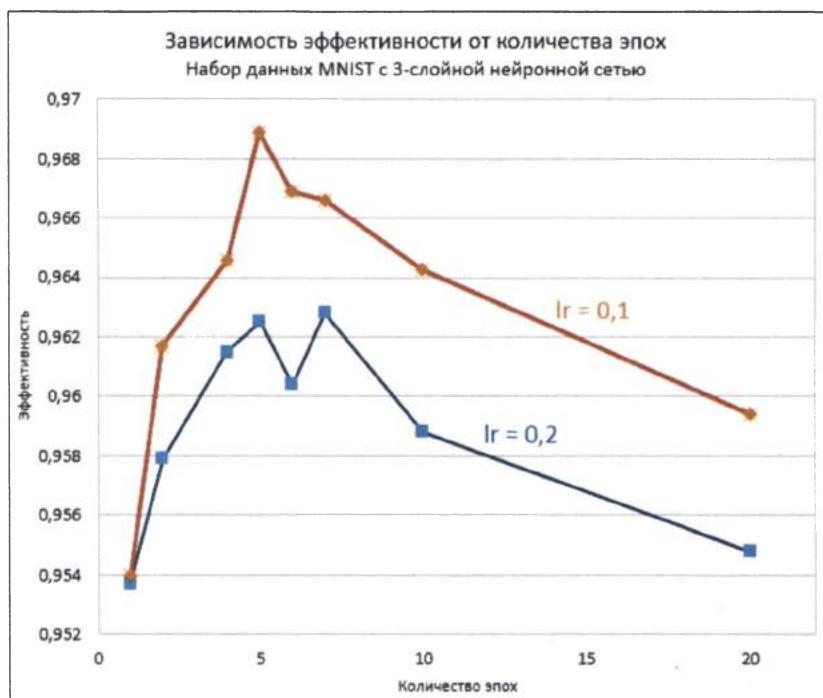
Теперь пиковое значение эффективности составляет **0,9628**, или 96,28%, при семи эпохах.

Как видите, результаты оказались не столь предсказуемыми, как ожидалось. Оптимальное количество эпох — 5 или 7. При больших значениях эффективность падает, что может быть следствием

переобучения. Провал при 6 эпохах, по всей вероятности, обусловлен неудачными параметрами цикла, которые привели градиентный спуск к ложному минимуму. На самом деле можно было ожидать большей вариации результатов, поскольку мы не проводили множества экспериментов для каждой точки данных, чтобы уменьшить вариацию, вызванную случайными факторами. Именно поэтому я оставил эту странную точку, соответствующую шести эпохам, чтобы напомнить вам, что по самой своей сути обучение нейронной сети — это случайный процесс, который иногда может работать не очень хорошо, а иногда и очень плохо.

Другое возможное объяснение — это то, что коэффициент обучения оказался слишком большим для большего количества эпох. Повторим эксперимент, уменьшив коэффициент обучения с 0,2 до 0,1, и посмотрим, что произойдет.

На следующем графике новые значения эффективности при коэффициенте обучения 0,1 представлены вместе с предыдущими результатами, чтобы их было легче сравнить между собой.



Как нетрудно заметить, уменьшение коэффициента обучения действительно привело к улучшению эффективности при большом количестве эпох. Пиковому значению **0,9689** соответствует вероятность ошибок, равная 3%, что сравнимо с эталонными результатами, указанными на сайте Яна Лекуна по адресу <http://yann.lecun.com/exdb/mnist/>.

Интуиция подсказывает нам, что если мы планируем использовать метод градиентного спуска при значительно большем количестве эпох, то уменьшение коэффициента обучения (более короткие шаги) в целом приведет к выбору лучших маршрутов минимизации ошибок. Вероятно, 5 эпох — это оптимальное количество для тестирования нашей нейронной сети с набором данных MNIST. Вновь обращаю ваше внимание на то, что мы сделали это довольно ненаучным способом. Правильно было бы выполнить эксперимент по нескольку раз для каждого сочетания значений коэффициента обучения и количества эпох с целью минимизации влияния фактора случайности, присущего методу градиентного спуска.

## Изменение конфигурации сети

Один из факторов, влияние которых мы еще не исследовали, хотя, возможно, и должны были сделать это раньше, — конфигурация нейронной сети. Необходимо попробовать изменить количество узлов скрытого промежуточного слоя. Давным-давно мы установили его равным 100.

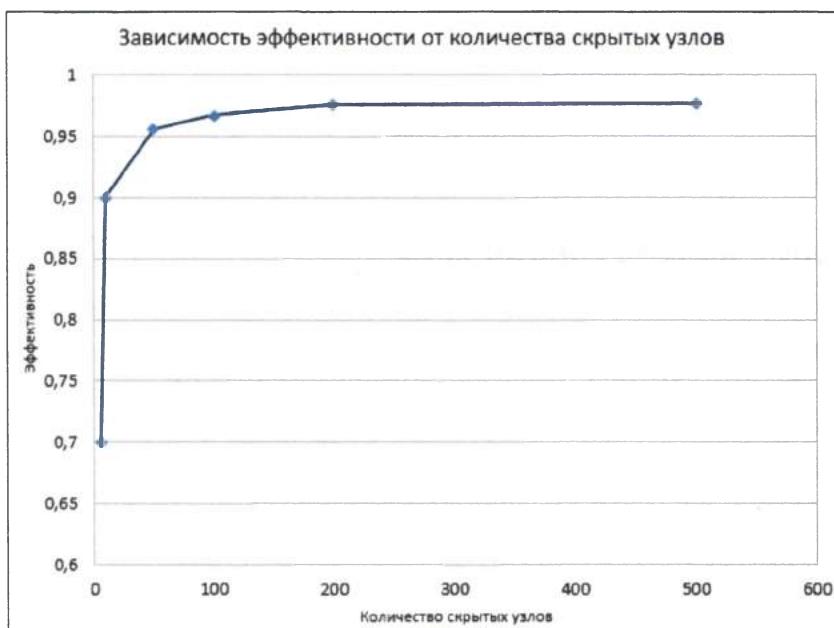
Прежде чем приступить к экспериментам с различными количествами скрытых узлов, давайте подумаем, что при этом может случиться. Скрытый слой как раз и является тем слоем, в котором происходит обучение. Вспомните, что узлы входного слоя принимают входные сигналы, а узлы выходного выдают ответ сети. Собственно, скрытый слой (или слои) должен научить сеть превращать входные сигналы в ответ. Именно в нем происходит обучение. На самом деле все обучение основано на значениях весовых коэффициентов до и после скрытого слоя, но вы понимаете, что я имею в виду.

Согласитесь, что если бы у нас было слишком мало скрытых узлов, скажем, три, то мы не имели бы никаких шансов обучить сеть чему-либо, научить ее преобразовывать все входные сигналы в корректные выходные сигналы. Это все равно что пытаться перевезти десять

человек в автомобиле, рассчитанном на пятерых. Вы просто не смогли бы втиснуть всю входную информацию в эти узлы. Ограничения такого рода называют способностью к обучению. Невозможно обучить большему, чем позволяет способность к обучению, но можно увеличить способность к обучению, изменив конфигурацию сети.

Что, если бы у нас было 10 тысяч скрытых узлов? Ну да, в таком случае способности к обучению вполне хватило бы, но мы могли бы столкнуться с трудностями обучения сети, поскольку число маршрутов обучения было бы непомерно большим. Не исключено, что для тренировки такой сети понадобилось бы 10 тысяч эпох.

Выполним эксперименты и посмотрим, что получится.



Как видите, при небольшом количестве узлов результаты хуже, чем при больших количествах. Это совпадает с нашими ожиданиями. Однако даже для всего лишь пяти скрытых узлов показатель эффективности составил **0,7001**. Это удивительно, поскольку при столь малом количестве узлов, в которых происходит собственно обучение сети, она все равно дает 70% правильных ответов. Вспомните, что до этого мы выполняли тесты, используя сто скрытых узлов. Но

даже десять скрытых узлов обеспечивают точность **0,8998**, или 90%, что тоже впечатляет.

Зафиксируйте в памяти этот факт. Нейронная сеть способна давать хорошие результаты даже при небольшом количестве скрытых узлов, в которых и происходит обучение, что свидетельствует о ее мощных возможностях.

По мере дальнейшего увеличения количества скрытых узлов результаты продолжают улучшаться, однако уже не так резко. При этом значительно растет время, затрачиваемое на тренировку сети, поскольку добавление каждого дополнительного скрытого узла приводит к увеличению количества связей узлов скрытого слоя с узлами предыдущего и следующего слоев, а вместе с этим и объема вычислений! Поэтому необходимо достигать компромисса между количеством скрытых узлов и допустимыми затратами времени. Для моего компьютера оптимальное количество узлов составляет двести. Ваш компьютер может работать быстрее или медленнее.

Мы также установили новый рекорд точности: **0,9751** при 200 скрытых узлах. Длительный расчет с 500 узлами обеспечил точность **0,9762**. Это действительно неплохо по сравнению с результатами эталонных тестов, опубликованными на сайте Лекуна по адресу <http://yann.lecun.com/exdb/mnist/>.

Возвращаясь к графикам, можно заметить, что неподдающийся ранее предел точности 97% был преодолен за счет изменения конфигурации сети.

## Подведем итоги

Давая общую оценку проделанной нами работы, хочу подчеркнуть, что при создании нейронной сети с помощью Python мы использовали лишь самые простые концепции.

И тем не менее с помощью нашей нейронной сети нам, даже без использования каких-либо дополнительных математических ухищрений, удалось получить вполне достойные результаты, сравнимые с теми, которые были достигнуты учеными и профессиональными исследователями.

Дополнительный интересный материал вы найдете в главе 3, но даже если вы не будете применять изложенные там идеи, можете безо всяких колебаний продолжить эксперименты с уже созданной

нами нейронной сетью — используйте другое количество скрытых узлов, задавайте другие коэффициенты масштабирования или даже задействуйте другую функцию активации и анализируйте, что при этом происходит.

## Окончательный вариант кода

Для удобства читателей ниже приведен окончательный вариант кода, также доступный на сайте GitHub.

```
# Блокнот Python для книги "Создаем нейронную сеть".
# Код для создания 3-слойной нейронной сети вместе с
# кодом для ее обучения с помощью набора данных MNIST.
# (c) Tariq Rashid, 2016
# лицензия GPLv2

import numpy
# библиотека scipy.special содержит сигмоиду expit()
import scipy.special
# библиотека для графического отображения массивов
import matplotlib.pyplot
# гарантировать размещение графики в данном блокноте,
# а не в отдельном окне
%matplotlib inline

# определение класса нейронной сети
class neuralNetwork:

    # инициализировать нейронную сеть
    def __init__(self, inputnodes, hiddennodes, outputnodes,
     learningrate):
        # задать количество узлов во входном, скрытом и выходном слое
        self.inodes = inputnodes
        self.hnodes = hiddennodes
        self.onodes = outputnodes

        # Матрицы весовых коэффициентов связей wih (между входным
        # и скрытым слоями) и who (между скрытым и выходным слоями).
        # Весовые коэффициенты связей между узлом i и узлом j
        # следующего слоя обозначены как w_i_j:
        # w11 w21
        # w12 w22 и т.д.
        self.wih = numpy.random.normal(0.0, pow(self.hnodes, -0.5),
     (self.hnodes, self.inodes))
        self.who = numpy.random.normal(0.0, pow(self.onodes, -0.5),
     (self.onodes, self.hnodes))
```

```

# коэффициент обучения
self.lr = learningrate

# использование сигмоиды в качестве функции активации
self.activation_function = lambda x: scipy.special.expit(x)

pass

# тренировка нейронной сети
def train(self, inputs_list, targets_list):
    # преобразование списка входных значений
    # в двухмерный массив
    inputs = numpy.array(inputs_list, ndmin=2).T
    targets = numpy.array(targets_list, ndmin=2).T

    # рассчитать входящие сигналы для скрытого слоя
    hidden_inputs = numpy.dot(self.wih, inputs)
    # рассчитать исходящие сигналы для скрытого слоя
    hidden_outputs = self.activation_function(hidden_inputs)

    # рассчитать входящие сигналы для выходного слоя
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # рассчитать исходящие сигналы для выходного слоя
    final_outputs = self.activation_function(final_inputs)

    # ошибки выходного слоя =
    # (целевое значение - фактическое значение)
    output_errors = targets - final_outputs
    # ошибки скрытого слоя - это ошибки output_errors,
    # распределенные пропорционально весовым коэффициентам связей
    # и рекомбинированные на скрытых узлах
    hidden_errors = numpy.dot(self.who.T, output_errors)

    # обновить веса для связей между скрытым и выходным слоями
    self.who += self.lr * numpy.dot((output_errors *
                                    final_outputs * (1.0 - final_outputs)),
                                    numpy.transpose(hidden_outputs))

    # обновить весовые коэффициенты для связей между
    # входным и скрытым слоями
    self.wih += self.lr * numpy.dot((hidden_errors *
                                    hidden_outputs * (1.0 - hidden_outputs)),
                                    numpy.transpose(inputs))

pass

# опрос нейронной сети
def query(self, inputs_list):
    # преобразовать список входных значений

```

```

# в двухмерный массив
inputs = numpy.array(inputs_list, ndmin=2).T

# рассчитать входящие сигналы для скрытого слоя
hidden_inputs = numpy.dot(self.wih, inputs)
# рассчитать исходящие сигналы для скрытого слоя
hidden_outputs = self.activation_function(hidden_inputs)

# рассчитать входящие сигналы для выходного слоя
final_inputs = numpy.dot(self.who, hidden_outputs)
# рассчитать исходящие сигналы для выходного слоя
final_outputs = self.activation_function(final_inputs)

return final_outputs

# количество входных, скрытых и выходных узлов
input_nodes = 784
hidden_nodes = 200
output_nodes = 10

# коэффициент обучения
learning_rate = 0.1

# создать экземпляр нейронной сети
n = neuralNetwork(input_nodes,hidden_nodes,output_nodes,learning_rate)

# загрузить в список тренировочный набор данных
# CSV-файла набора MNIST
training_data_file = open("mnist_dataset/mnist_train.csv", 'r')
training_data_list = training_data_file.readlines()
training_data_file.close()

# тренировка нейронной сети

# переменная epochs указывает, сколько раз тренировочный
# набор данных используется для тренировки сети
epochs = 5

for e in range(epochs):
    # перебрать все записи в тренировочном наборе данных
    for record in training_data_list:
        # получить список значений из записи, используя символы
        # запятой (',') в качестве разделителей
        all_values = record.split(',')
        # масштабировать и сместить входные значения
        inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
        # создать целевые выходные значения (все равны 0,01, за
        # исключением желаемого маркерного значения, равного 0,99)
        targets = numpy.zeros(output_nodes) + 0.01

```

```

# all_values[0] - целевое маркерное значение
# для данной записи
targets[int(all_values[0])] = 0.99
n.train(inputs, targets)
pass
pass

# загрузить в список тестовый набор данных
# CSV-файла набора MNIST
test_data_file = open("mnist_dataset/mnist_test.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()

# тестирование нейронной сети

# журнал оценок работы сети, первоначально пустой
scorecard = []

# перебрать все записи в тестовом наборе данных
for record in test_data_list:
    # получить список значений из записи, используя символы
    # запятой (',') в качестве разделителей
    all_values = record.split(',')
    # правильный ответ - первое значение
    correct_label = int(all_values[0])
    # масштабировать и сместить входные значения
    inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
    # опрос сети
    outputs = n.query(inputs)
    # индекс наибольшего значения является маркерным значением
    label = numpy.argmax(outputs)
    # присоединить оценку ответа сети к концу списка
    if (label == correct_label):
        # в случае правильного ответа сети присоединить
        # к списку значение 1
        scorecard.append(1)
    else:
        # в случае неправильного ответа сети присоединить
        # к списку значение 0
        scorecard.append(0)
    pass

pass

# рассчитать показатель эффективности в виде
# доли правильных ответов
scorecard_array = numpy.asarray(scorecard)
print ("эффективность = ", scorecard_array.sum() /
    scorecard_array.size)

```



## ГЛАВА 3

# Несколько интересных проектов

*Не играя, не научишься.*

В этой главе мы исследуем ряд оригинальных идей. Они не связаны с пониманием основ нейронных сетей, поэтому не смущайтесь, если кое-что будет вам непонятно.

Поскольку глава предназначена больше для развлечения, мы ускорим темп, хотя там, где это потребуется, будут даваться простые пояснения.

## **Собственный рукописный текст**

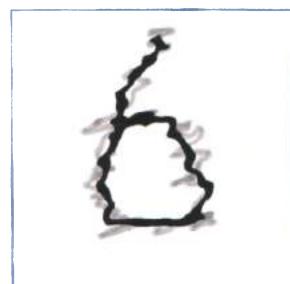
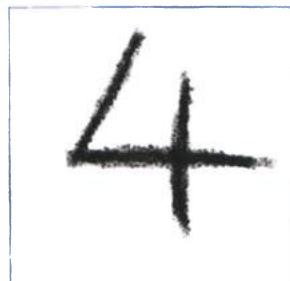
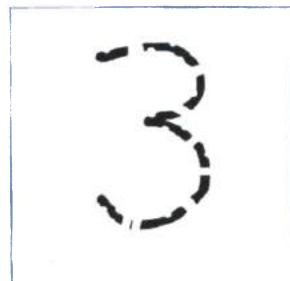
В главе 2 мы распознавали изображения рукописных цифр из базы данных MNIST. А почему бы не использовать собственный рукописный текст?

В этом эксперименте мы создадим свою тестовую базу данных, используя собственноручно написанные цифры. Кроме того, мы попытаемся сымитировать различные виды почерка, а также зашумленные и прерывистые изображения, чтобы посмотреть, насколько хорошо с ними справится наша нейронная сеть.

Для создания изображений можно использовать любой графический редактор. Вы не обязаны делать это с помощью дорогостоящей программы Photoshop. Вполне подойдет альтернативный вариант в виде бесплатной программы с открытым исходным кодом GIMP ([www.gimp.org](http://www.gimp.org)), доступной для компьютеров с операционными системами Windows, Mac и Linux. Вы даже можете написать цифры карандашом на листе бумаги, а затем отсканировать или сфотографировать их с помощью смартфона или фотокамеры. Единственным требованием является то, что изображение цифры должно быть

квадратным (длина равна ширине) и представленным в формате PNG. Этот формат можно выбрать в списке допустимых форматов большинства графических редакторов при выполнении команды меню Файл→Сохранить как или Файл→Экспорт.

Вот как выглядят некоторые из подготовленных мною изображений.



Цифру “5” я написал маркером. Цифра “4” написана мелом. Цифру “3” я написал маркером, но намеренно сделал линию прерывистой. Цифра “2” взята из типичного газетного или книжного шрифта, но немного размыта. Цифра “6” как бы покрыта рябью, словно мы видим ее как отражение на поверхности воды. Последнее изображение совпадает с предыдущим, но в него добавлен шум, чтобы намеренно затруднить распознавание цифры нейронной сетью.

Все это забавно, но здесь есть и серьезный момент. Ученых изумляет поразительная способность человеческого мозга сохранять функциональность даже после того, как он испытал повреждения. Предполагают, что в нейронной сети приобретенные знания распределяются между несколькими связями, а потому даже в условиях повреждения некоторых связей остальные связи могут успешно

функционировать. Это также означает, что они могут достаточно хорошо функционировать и в случае повреждения или неполноты входного изображения. Таковы возможности нашего мозга, и именно это мы хотим протестировать с посеченной цифрой “3”, изображенной на приведенной выше иллюстрации.

Прежде всего, мы должны создать уменьшенные версии PNG-изображений, масштабировав их до размера  $28 \times 28$  пикселей, чтобы привести в соответствие с использовавшимися ранее данными MNIST. Для этого можете использовать свой графический редактор.

Для чтения и декодирования данных из распространенных форматов изображений, включая PNG, мы вновь используем библиотеки Python. Взгляните на следующий простой код.

```
import scipy.misc  
img_array = scipy.misc.imread(image_file_name, flatten=True)  
  
img_data = 255.0 - img_array.reshape(784)  
img_data = (img_data / 255.0 * 0.99) + 0.01
```

Функция `scipy.misc.imread()` поможет нам в получении данных из файлов изображений, таких как PNG- или JPG-файлы. Чтобы использовать библиотеку `scipy.misc`, ее необходимо импортировать. Параметр `flatten=True` превращает изображение в простой массив чисел с плавающей запятой и, если изображение цветное, переводит цветовые коды в шкалу оттенков серого, что нам и надо.

Следующая строка преобразует квадратный массив размерностью  $28 \times 28$  в длинный список значений, который нужен для передачи данных нейронной сети. Ранее мы делали это не раз. Новым здесь является вычитание значений массива из 255,0. Это необходимо сделать, поскольку обычно коду 0 соответствует черный цвет, а коду 255 — белый, но в наборе данных MNIST используется обратная схема, в связи с чем мы должны инвертировать цвета для приведения их в соответствие с соглашениями, принятыми в MNIST.

Последняя строка выполняет уже знакомое вам масштабирование данных, переводя их в диапазон значений от 0,01 до 1,0.

Образец кода, демонстрирующего чтение PNG-файлов, доступен на сайте GitHub по следующему адресу:

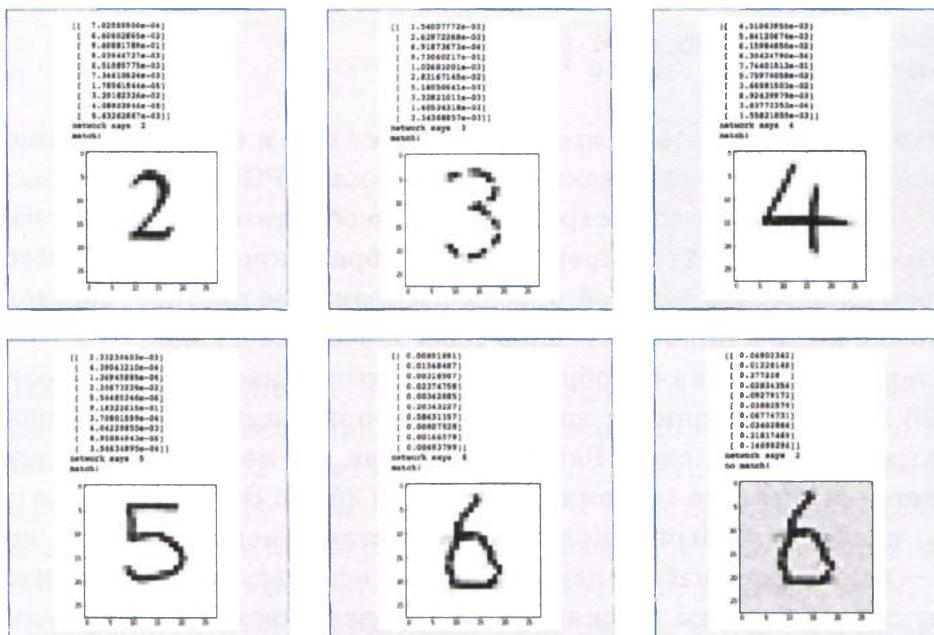
[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3\\_load\\_own\\_images.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_load_own_images.ipynb)

Нам нужно создать версию программы, использовавшейся ранее для создания базовой нейронной сети и ее обучения с помощью набора данных MNIST, но теперь мы будем тестировать программу с использованием набора данных, созданного на основе наших изображений.

Новая программа доступна на сайте GitHub по следующему адресу:

[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3\\_neural\\_network\\_mnist\\_and\\_own\\_data.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_and_own_data.ipynb)

Работает ли она? Конечно, работает! Следующая иллюстрация демонстрирует результаты опроса сети с использованием наших изображений.



Как видите, нейронной сети удалось распознать все изображения, включая намеренно поврежденную цифру “3”. Не удалось распознать лишь зашумленную цифру “6”.

Проведите эксперименты с подготовленными вами изображениями, включая изображения рукописных цифр, и убедитесь в том, что нейронная сеть действительно работает.

Также проверьте, как далеко можно зайти с испорченными или искаженными цифрами. Вы будете поражены гибкостью нашей нейронной сети.

## Проникнем в “мозг” нейронной сети

Нейронные сети полезны при решении тех задач, для которых трудно придумать простой и четкий алгоритм решения с фиксированными правилами. Представьте себе только, каким должен быть набор правил, регламентирующих процедуру определения рукописной цифры, представленной изображением! Вы согласитесь, что создать такие правила довольно нелегко, а вероятность того, что наши попытки окажутся успешными, весьма мала.

### Загадочный черный ящик

Завершив обучение нейронной сети и проверив ее работоспособность на тестовых данных, вы, по сути, получаете **черный ящик**. Фактически вы не знаете, как вырабатывается ответ, но все же он вырабатывается!

Незнание внутреннего механизма не всегда является проблемой, если главное для вас — ответы и вас не интересует, каким образом они получены. Но именно этим недостатком страдают рассматриваемые методы машинного обучения: обучение не всегда означает понимание сути задачи, которую черный ящик научился решать.

Давайте разберемся, можем ли мы заглянуть внутрь нашей простой нейронной сети и увидеть, чему она научилась, чтобы визуализировать знания, приобретенные ею в процессе тренировки.

Мы могли бы проанализировать весовые коэффициенты, в которых, в конце концов, и сосредоточено все, чему учится сеть. Но вряд ли эти данные будут нести в себе полезную для нас информацию, особенно если учесть, что нейронная сеть работает таким образом, чтобы распределить то, чему она учится, по различным связям. Это обеспечивает устойчивость сети к повреждениям, как это свойственно биологическому мозгу. Маловероятно, что удаление одного или

даже большего количества узлов полностью лишит сеть возможности нормально работать.

Рассмотрим одну безумную идею.

## Обратные запросы

Обычно мы задаем тренированной сети вопрос, и она выдает нам ответ. В нашем примере вопросом является изображение рукописной цифры. Ответом является маркер, представляющий число из диапазона значений от 0 до 9.

А что, если развернуть этот процесс наоборот? Что, если мы подадим маркер на выходные узлы и проследим за распространением сигнала по уже натренированной сети в обратном направлении, пока не получим на входных узлах исходное изображение? Следующая диаграмма иллюстрирует процесс распространения обычного запроса и описанный только что процесс распространения обратного запроса.



Мы уже знаем, как распространять сигналы по сети, сглаживая их весовыми коэффициентами и рекомбинируя на узлах, прежде чем применять к ним функцию активации. Весь этот механизм работает также для сигналов, распространяющихся в обратном направлении,

за исключением того, что в этом случае используется обратная функция активации. Если  $y = f(x)$  — функция активации для прямых сигналов, то ее обратная функция —  $x = g(y)$ . Для логистической функции нахождение обратной функции сводится к простой алгебре:

$$\begin{aligned}y &= 1 / (1 + e^{-x}) \\1 + e^{-x} &= 1/y \\e^{-x} &= (1/y) - 1 = (1-y)/y \\-x &= \ln[(1-y)/y] \\x &= \ln[y/(1-y)]\end{aligned}$$

Эта функция называется **logit**, и библиотека Python `scipy.special` предоставляет ее как `scipy.special.logit()`, точно так же, как и логистическую функцию `scipy.special.expit()`.

Прежде чем использовать обратную функцию активации `logit()`, мы должны убедиться в допустимости сигналов. Что это означает? Вы помните, что сигмоида принимает любое значение и возвращает значение из диапазона от 0 до 1, исключая граничные значения. Обратная функция должна принимать значения из того же самого диапазона — от 0 до 1, исключая сами значения 0 и 1, — и возвращать значение, которое может быть любым положительным или отрицательным числом. Для этого мы просто берем все значения в слое, к которым собираемся применить функцию `logit()`, и приводим их к допустимому диапазону. В качестве такового я выбрал диапазон чисел от 0,01 до 0,99.

Соответствующий код доступен на сайте GitHub по следующему адресу:

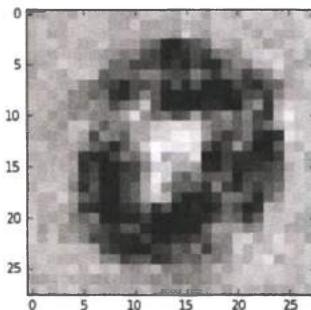
[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3\\_neural\\_network\\_mnist\\_backquery.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_backquery.ipynb)

## Маркер “0”

Посмотрим, что произойдет, если выполнить обратный запрос для маркера “0”, т.е. мы предоставляем выходным узлам значения, равные 0,01, за исключением первого узла, соответствующего маркеру “0”, которому мы предоставляем значение 0,99. Иными словами,

мы передаем на выходные узлы массив чисел  $[0.99, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01]$ .

Ниже показано изображение, полученное на входных узлах.



Это уже интересно! Так “видит” картинку “мозг” нашей нейронной сети.

Как расценивать полученное изображение? Как его следует интерпретировать?

Основное, что сразу же бросается в глаза, — округлая форма изображения. Это соответствует истине, поскольку мы интересовались у нейронной сети, каков тот идеальный вопрос, ответом на который будет “0”.

Кроме того, на изображении можно выделить темные, светлые и серые области.

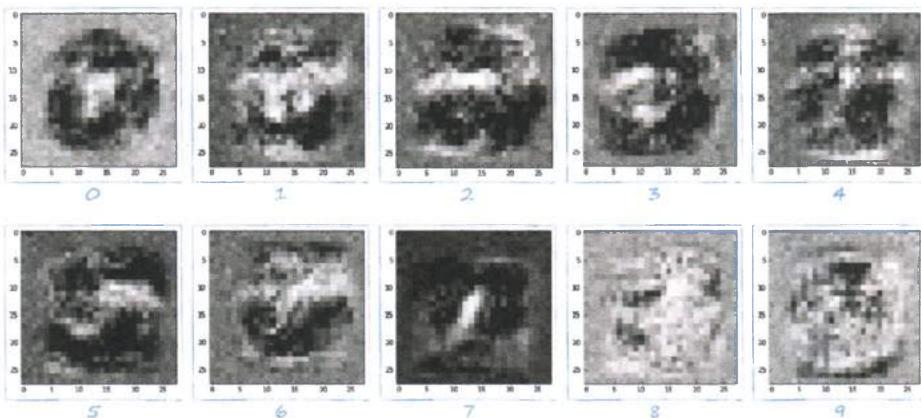
- **Темные области** — это те части искомого изображения, которые, если обвести их маркером, свидетельствуют в пользу того, что ответом следует считать “0”. Форма их контура действительно напоминает цифру “0”.
- **Светлые области** — это те участки искомого изображения, которые следует считать не закрашенными, если остановиться на версии, что ответом является “0”. Это предположение представляется разумным, поскольку эти участки располагаются внутри контура цифры “0”.
- **Серые области** в основном не несут никакой информации для нейронной сети.

Итак, в общих чертах нам удалось достигнуть некоторого понимания того, как именно сформировалось умение сети классифицировать изображения с маркером “0”.

Нам выпала редкостная удача заглянуть внутрь нейронной сети, поскольку в случае сетей с более сложной структурой и большим количеством слоев, а также в случае более сложных задач вряд ли можно рассчитывать на столь же легкую интерпретацию результатов.

## Остальные изображения

Ниже представлены результаты обратного запроса для остальных случаев.



Вот это да! Мы вновь получили довольно-таки интересные изображения. Они словно представляют собой снимки мозга нейронной сети, полученные с помощью компьютерной томографии.

По поводу этих изображений можно сделать некоторые замечания.

- Маркер “7” довольно отчетливо распознается как цифра “7”. Если вы обведете карандашом темные пиксели изображения, то получите достаточно наглядное подтверждение этого. Также заметно выделяется “белая” область, где не должно быть закрашенных элементов. Оба этих фактора вместе указывают на то, что в данном случае мы имеем дело с цифрой “7”.

- То же самое справедливо для маркера “3”, поскольку здесь на лицо те же два фактора: схожесть контура темных пикселей, если обвести его карандашом, с цифрой “3” и наличие белых областей в тех местах, где они и должны быть в цифре “3”.
- Маркеры “2” и “5” интерпретируются аналогичным образом.
- Случай маркера “4” интересен наличием фигуры, напоминающей четыре квадранта, и белых областей.
- Изображение, полученное для маркера “8”, очень напоминает снеговика, “голова и туловище” которого сформированы двумя белыми областями, что в целом соответствует цифре “8”.
- Изображение для маркера “1” ставит нас в тупик. Создается впечатление, что сеть уделила больше внимания тем областям, которые должны оставаться белыми, чем тем, которые должны быть закрашены. Ну что ж, это то, чему научилась сеть на примерах.
- Изображение для маркера “9” вообще неразборчиво. В нем нет четко выделенных темных областей и каких-либо фигур, образованных белыми областями. Это результат того, чему сеть научилась на предоставленных ей примерах, обеспечивая в целом точность на уровне 97,5%. Глядя на данное изображение, напрашивается вывод, что, возможно, сеть нуждается в дополнительных тренировочных примерах, которые помогли бы ей лучше распознавать образцы цифры “9”.

Итак, вам была предоставлена увлекательная возможность заглянуть во внутренний мир нейронной сети и, что называется, увидеть, как работает ее мозг.

## Создание новых тренировочных данных: вращения

Оценивая тренировочные данные MNIST, следует сказать, что они содержат довольно богатый набор рукописных начертаний цифр. В нем встречается множество видов почерка и стилей, причем как хороших, так и плохих.

Нейронная сеть должна учиться на как можно большем количестве всевозможных вариантов написания цифр. Удачно то, что в этот

набор входит также множество форм написания цифры “4”. Одни из них искусственно сжаты, другие растянуты, некоторые повернуты, в одних верхушка цифры разомкнута, в других сомкнута.

Разве не будет полезно создать дополнительные варианты написания и использовать их в качестве тренировочных примеров? Как это сделать? Нам трудно собрать коллекцию из тысячи дополнительных примеров рукописного написания той или иной цифры. Вернее, мы могли бы попытаться, но это стоило бы нам огромных усилий.

Но ведь ничто не мешает нам взять существующие примеры и создать на их основе новые, повернув цифры по часовой стрелке или против, скажем, на 10 градусов. В этом случае мы могли бы создать по два дополнительных примера для каждого из уже имеющихся. Мы могли бы создать гораздо больше примеров, врачаая образцы на разные углы, но мы ограничимся углами +10 и -10 градусов, чтобы посмотреть, как работает эта идея.

И вновь большую помощь в этом нам окажут расширения и библиотеки Python. Функция `scipy.ndimage.interpolation.rotate()` поворачивает изображение, представленное массивом, на заданный угол, а это именно то, что нам нужно. Описание функции можно найти здесь:

<https://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.ndimage.interpolation.rotate.html>

Вспомните, что наши входные значения представляют собой одномерный список длиной 784 элемента, поскольку мы спроектировали нашу нейронную сеть таким образом, чтобы она принимала длинный список входных сигналов. Мы должны реорганизовать этот список в массив размерностью  $28 \times 28$ , чтобы повернуть изображение, а затем проделать обратную операцию по преобразованию массива в список из 784 входных сигналов, которые мы подадим на вход нейронной сети.

В приведенном ниже коде показан пример использования функции `scipy.ndimage.interpolation.rotate()` в предположении, что у нас уже имеется массив `scaled_input`, о котором шла речь ранее.

```
# создание повернутых на некоторый угол вариантов изображений
```

```
# повернуть на 10 градусов против часовой стрелки
inputs_plus10_img =
```

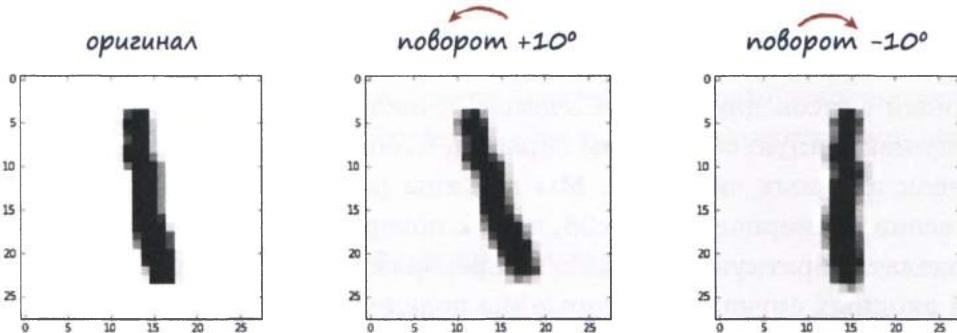
```

scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28),
10, cval=0.01, reshape=False)
# повернуть на 10 градусов по часовой стрелке
inputs_minus10_img =
scipy.ndimage.interpolation.rotate(scaled_input.reshape(28,28),
-10, cval=0.01, reshape=False)

```

В этом коде первоначальный массив `scaled_input` преобразуется в массив размерностью  $28 \times 28$ . Параметр `reshape=False` сдерживает излишнюю рьяность библиотеки в ее желании быть как можно более полезной и сжать изображение таким образом, чтобы после вращения все оно уместилось в массиве и ни один его пиксель не был отсечен. Параметр `cval` — это значение, используемое для заполнения элементов массива, которые не существовали в исходном массиве, но теперь появились. Мы откажемся от используемого по умолчанию значения 0,0 и заменим его значением 0,01, поскольку во избежание подачи на вход нейронной сети нулей мы используем смещенный диапазон входных значений.

Запись 6 (седьмая по счету) малого тренировочного набора MNIST содержит рукописное начертание цифры “1”. Вот как выглядят ее исходное изображение и две его повернутые вариации, полученные с помощью нашего кода.



Результаты очевидны. Версия исходного изображения, повернутая на  $+10$  градусов, является примером почерка человека, “заливающего” текст влево. Еще более интересна версия оригинала, повернутая на  $-10$  градусов, т.е. по часовой стрелке. Эта версия

располагается даже ровнее по сравнению с оригиналом и в некотором смысле более представительна в качестве изображения для обучения.

Создадим новый блокнот Python с имеющимся кодом нейронной сети, но с дополнительными тренировочными примерами, созданными путем поворота исходных изображений на 10 градусов в обе стороны. Этот код доступен на сайте GitHub по следующему адресу:

[https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3\\_neural\\_network\\_mnist\\_data\\_with\\_rotations.ipynb](https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_data_with_rotations.ipynb)

Запуск этого кода с использованием коэффициента обучения 0,1 и всего лишь одной тренировочной эпохи дает показатель эффективности, равный **0,9669**. Это значительное улучшение по сравнению со значением 0,954, полученным без дополнительных повернутых изображений. Такой показатель уже попадает в число лучших из тех, которые опубликованы на сайте Яна Лекуна (<http://yann.lecun.com/exdb/mnist/>).

Запустим серию экспериментов, изменяя количество эпох, чтобы проверить, можно ли еще больше улучшить полученный показатель эффективности обучения. Кроме того, уменьшим коэффициент обучения до **0,01**, поскольку, предоставив гораздо больше тренировочных данных и тем самым увеличив общее время обучения, мы можем позволить себе более осторожные шаги обучения меньшей величины.

Не забывайте о том, что мы не ожидаем получить точность распознавания 100%, поскольку, вероятно, существует естественный предел точности, обусловленный спецификой архитектуры нейронной сети или полнотой тренировочных данных, в связи с чем мы вряд ли можем получить точность выше примерно 98%. Под “спецификой архитектуры нейронной сети” здесь подразумевается выбор количества узлов в каждом слое, количества скрытых слоев, функции активации и т.п.

Ниже представлены графики, отражающие зависимость эффективности нейронной сети от угла поворота дополнительных тренировочных изображений. Для сравнения показана также точка данных, соответствующая отсутствию дополнительных примеров.



Как видите, для пяти эпох наилучший результат равен **0,9745**, или 97,5% точности. Это явное улучшение по сравнению с предыдущим примером.

Также следует отметить, что с увеличением угла поворота изображения точность падает. Это вполне объяснимо, поскольку при больших углах поворота результирующие изображения фактически вообще не представляют цифры. Представьте цифру “3”, повернутую на 90 градусов, т.е. положенную на бок. Это будет вовсе не тройка. Поэтому, добавляя тренировочные примеры с чрезмерно большими углами поворота, мы снижаем качество тренировки, потому что добавляемые примеры являются ложными. Пожалуй, оптимальным углом поворота для дополнительных тренировочных изображений является угол 10 градусов.

Для десяти эпох рекордное пиковое значение точности составляет **0,9787**, или почти **98%**! Это поистине ошеломляющий результат для простой нейронной сети такого рода. Не забывайте о том, что мы не использовали никаких изощренных математических трюков в отношении нейронной сети или данных, как это делают некоторые

люди. Мы придерживались предельной простоты и тем не менее достигли результатов, которыми по праву можем гордиться.



**Отличная работа!**

## ПРИЛОЖЕНИЕ А

# Краткое введение в дифференциальное исчисление

Представьте, что вы движетесь в автомобиле с постоянной скоростью 30 миль в час. Затем вы нажимаете на педаль акселератора. Если вы будете держать ее постоянно нажатой, скорость движения постепенно увеличится до 35, 40, 50, 60 миль в час и т.д.

**Скорость движения автомобиля изменяется.**

В этом разделе мы исследуем природу изменения различных величин и обсудим способы математического описания этих изменений. А что подразумевается под “математическим описанием”? Это означает установление соотношений между различными величинами, что позволит нам точно определить степень изменения одной величины при изменении другой. Например, речь может идти об изменении скорости автомобиля с течением времени, отслеживаемого по наручным часам. Другими возможными примерами могут служить зависимость высоты растений от уровня выпавших осадков или изменение длины пружины в зависимости от приложенного к ее концам усилия.

Математики называют это **дифференциальным исчислением**. Я долго сомневался, прежде чем решился вынести этот термин в название приложения. Мне казалось, что многих людей он отпугнет, поскольку они посчитают изложенный ниже материал слишком сложным для того, чтобы на него тратить время. Однако такое отношение к данному предмету обсуждения могли привить только плохие учителя или плохие школьные учебники.

Прочитав все приложение до конца, вы убедитесь в том, что математическое описание изменений — а именно для этого и предназначено дифференциальное исчисление — оказывается совсем не сложным во многих полезных сценариях.

Даже если вы уже знакомы с дифференциальным исчислением, возможно, еще со школы, вам все равно стоит прочитать это приложение, поскольку вы узнаете много интересного об истории математики. Вам будет полезно взять на вооружение идеи и инструменты математического анализа, чтобы использовать их в будущем при решении различных задач.

Если вам нравятся исторические эссе, почитайте книгу “Великие противостояния в науке” (ИД “Вильямс”, 2007), где рассказывается о драме, разыгравшейся между Лейбницем и Ньютона, каждый из которых приписывал открытие дифференциального исчисления себе!



Готфрид Лейбниц



Сэр Исаак Ньютона

## Прямая линия

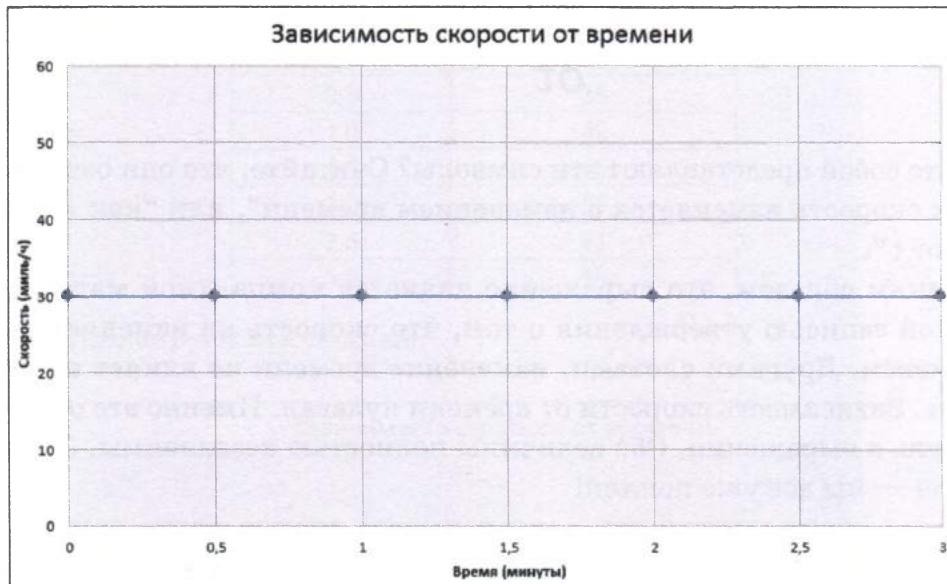
Чтобы настроиться на работу, начнем для разминки с очень простого сценария.

Вновь представьте себе автомобиль, движущийся с постоянной скоростью 30 миль в час. Не больше и не меньше, а ровно 30 миль в час.

В приведенной ниже таблице указана скорость автомобиля в различные моменты времени с интервалом в полминуты.

Время (мин.)	Скорость (миль/ч)
0,0	30
0,5	30
1,0	30
1,5	30
2,0	30
2,5	30
3,0	30

Следующий график представляет значения скорости в соответствующие моменты времени.



Как видите, скорость не изменяется с течением времени, и поэтому точки выстраиваются в прямую линию. Эта линия не идет ни вверх (увеличение скорости), ни вниз (уменьшение скорости), а остается на уровне 30 миль в час.

В данном случае математическое выражение для скорости, которую мы обозначим как  $s$ , имеет следующий вид:

$$s = 30$$

Если бы кто-то спросил нас о том, как изменяется скорость со временем, мы бы ответили, что она не изменяется. Скорость ее изменения равна нулю. Иными словами, скорость не зависит от времени. Зависимость в данном случае нулевая.

Мы только что выполнили одну из операций дифференциального исчисления! Я не шучу!

Дифференциальное исчисление сводится к нахождению изменения одной величины в результате изменения другой. В данном случае нас интересует, как скорость изменяется со временем.

Вышеизложенное можно записать в следующей математической форме:

$$\frac{\delta s}{\delta t} = 0$$

Что собой представляют эти символы? Считайте, что они означают “как скорость изменяется с изменением времени”, или “как  $s$  зависит от  $t$ ”.

Таким образом, это выражение является компактной математической записью утверждения о том, что скорость не изменяется со временем. Другими словами, изменение времени не влияет на скорость. Зависимость скорости от времени нулевая. Именно это означает нуль в выражении. Обе величины полностью независимы. Ладно, ладно — мы все уже поняли!

В действительности эту независимость можно легко заметить, если вновь взглянуть на выражение для скорости  $s = 30$ . В нем вообще нет даже намека на время, т.е. в нем отсутствует любое проявление символа  $t$ . Поэтому, чтобы сказать, что  $\partial s / \partial t = 0$ , нам не потребуется никакое дифференциальное исчисление. Говоря языком математиков, “это следует из самого вида выражения”.

Выражения вида  $\partial s / \partial t$ , определяющие степень изменения одной величины при изменении другой, называют производными. Для наших целей знание этого термина не является обязательным, но он вам может встретиться где-нибудь еще.

А теперь посмотрим, что произойдет, если надавить на педаль акселератора. Поехали!

## Наклонная прямая линия

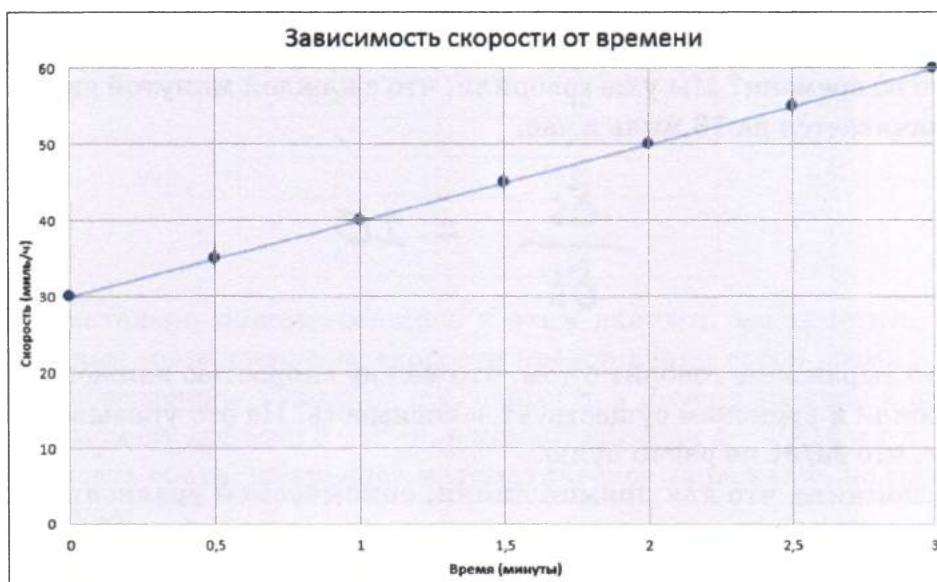
Представьте себе все тот же автомобиль, движущийся со скоростью 30 миль в час. Вы надавили на педаль акселератора, и автомобиль набирает скорость. Вы удерживаете педаль нажатой, смотрите на показания спидометра и записываете их через каждые 30 секунд.

По прошествии 30 секунд скорость автомобиля составила 35 миль в час. Через минуту она возрастает до 40 миль в час. Через 90 секунд автомобиль ускоряется до 45 миль в час, а после двух минут автомобиль разгоняется до 50 миль в час. С каждой минутой скорость автомобиля увеличивается на 10 миль в час.

Эта информация сведена в представленной ниже таблице.

Время (мин.)	Скорость (миль/ч)
0,0	30
0,5	35
1,0	40
1,5	45
2,0	50
2,5	55
3,0	60

Визуализируем эти данные.



Как видите, увеличение скорости движения автомобиля с 30 до 60 миль в час происходит с **постоянной скоростью изменения**. Она действительно постоянна, поскольку приращение скорости за каждые полминуты остается одним и тем же, что приводит к **прямолинейному** графику скорости.

А что собой представляет выражение для скорости? В нулевой момент времени скорость равна 30 миль в час. Далее мы добавляем по 10 миль в час за каждую минуту. Таким образом, искомое выражение должно иметь следующий вид:

$$\text{скорость} = 30 + (10 * \text{время})$$

Перепишем его, используя символические обозначения:

$$s = 30 + 10t$$

В этом выражении имеется константа 30. В нем также есть член **(10 \* время)**, который каждую минуту увеличивает скорость на 10 миль в час. Вы быстро сообразите, что 10 — это **угловой коэффициент линии**, график которой мы построили. Вспомните, что общая форма уравнения прямой линии имеет вид  $y = ax + b$ , где  $a$  — угловой коэффициент, или **наклон**, линии.

А как будет выглядеть выражение, описывающее изменение скорости во времени? Мы уже говорили, что с каждой минутой скорость увеличивается на 10 миль в час.

$$\frac{\delta s}{\delta t} = 10$$

Это выражение говорит о том, что между скоростью движения автомобиля и временем существует зависимость. На это указывает тот факт, что  $\partial s / \partial t$  не равно нулю.

Вспомните, что для прямой линии, описываемой уравнением  $y = ax + b$ , наклон равен  $a$ , и поэтому наклон для прямой линии  $s = 30 + 10t$  должен быть равным 10.

Отличная работа! Мы уже успели охватить довольно много базовых элементов дифференциального исчисления, и это оказалось совсем несложно.

А теперь надавим на акселератор еще сильнее!

## Кривая линия

Представьте, что я начинаю поездку в автомобиле, нажав педаль акселератора почти до упора и удерживая ее в этом положении. Совершенно очевидно, что начальная скорость равна нулю, поскольку в первый момент автомобиль вообще не двигался.

Поскольку педаль акселератора нажата почти до упора, скорость движения будет увеличиваться не равномерно, а быстрее. Это означает, что скорость автомобиля уже не будет возрастать только на 10 миль в час каждую минуту. Само ежеминутное приращение скорости будет с каждой минутой увеличиваться, если педаль акселератора по-прежнему удерживается нажатой.

Предположим, что скорость автомобиля принимает в каждую минуту значения, приведенные в следующей таблице.

Время (мин.)	Скорость (миль/ч)
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64

Внимательно присмотревшись к этим данным, вы заметите, что выбранные мною значения скорости представляют собой время в минутах, возведенное в квадрат, т.е. скорость в момент времени 2 равна  $2^2=4$ , в момент времени 3 –  $3^2=9$ , в момент времени 4 –  $4^2=16$  и т.д.

Записать соответствующее математическое выражение не составляет труда:

$$s = t^2$$

Да, я отдаю себе отчет в том, что пример со скоростью автомобиля довольно искусственный, но он позволяет наглядно продемонстрировать, как мы можем использовать дифференциальное исчисление.

Представим табличные данные в виде графика, который позволит нам лучше понять характер изменения скорости автомобиля во времени.



Как видите, со временем скорость автомобиля растет все интенсивнее. График уже не является прямой линией. Нетрудно сделать вывод, что вскоре скорость должна достигнуть очень больших значений. На 20-й минуте она должна была бы составить 400 миль в час, а на 100-й — целых 10000 миль в час!

Возникает интересный вопрос: как быстро изменяется скорость с течением времени? Другими словами, каково приращение скорости в каждый момент времени?

Это не то же самое, что спросить: а какова фактическая скорость в каждый момент времени? Ответ на этот вопрос нам уже известен, поскольку для него у нас есть соответствующее выражение:  $s = t^2$ .

Мы же спрашиваем следующее: какова **скорость изменения скорости** автомобиля в каждый момент времени? Но что вообще это означает в нашем примере, в котором график оказался криволинейным?

Если вновь обратиться к двум предыдущим примерам, то в них скорость изменения скорости определялась наклоном графика зависимости скорости от времени. Когда автомобиль двигался с постоянной скоростью 30 миль в час, его скорость не изменялась, и поэтому скорость ее изменения была равна 0. Когда автомобиль равномерно набирал скорость, скорость ее изменения составляла 10 миль в час за минуту. Данный показатель имел одно и то же значение в любой момент времени. Он был равен 10 миль в час на второй, четвертой и даже на сотой минуте.

Можем ли мы применить те же рассуждения к криволинейному графику? Можем, но с этого момента нам следует немного сбавить темп, чтобы не спеша обсудить этот вопрос.

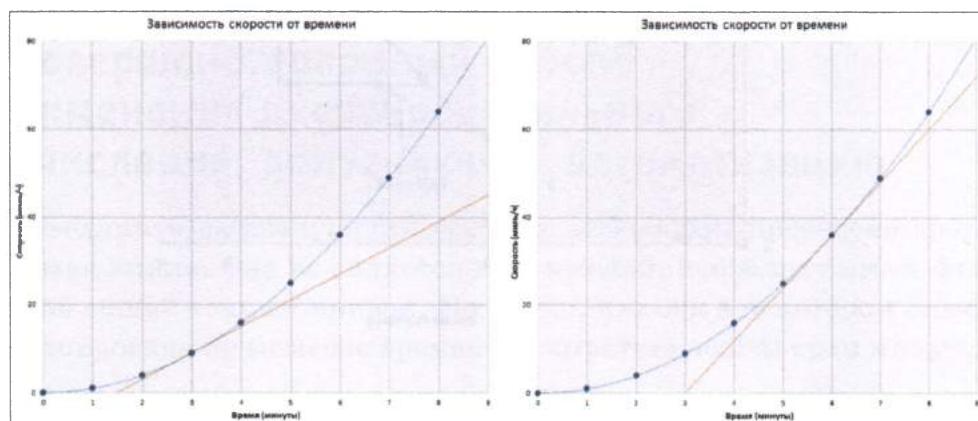
## Применение дифференциального исчисления вручную

Присмотримся повнимательнее к тому, что происходит в конце третьей минуты движения.

По прошествии трех минут с момента начала движения ( $t=3$ ) скорость ( $s$ ) составит 9 миль в час. Сравним это с тем, что будет в конце шестой минуты движения. В этот момент скорость составит 36 миль в час, но мы знаем, что после этого автомобиль будет двигаться еще быстрее.

Мы также знаем, что в любой момент времени вслед за шестой минутой скорость будет увеличиваться быстрее, чем в эквивалентный момент времени, следующий за третьей минутой. Существует реальное различие в том, что происходит в моменты времени, соответствующие трем и шести минутам движения.

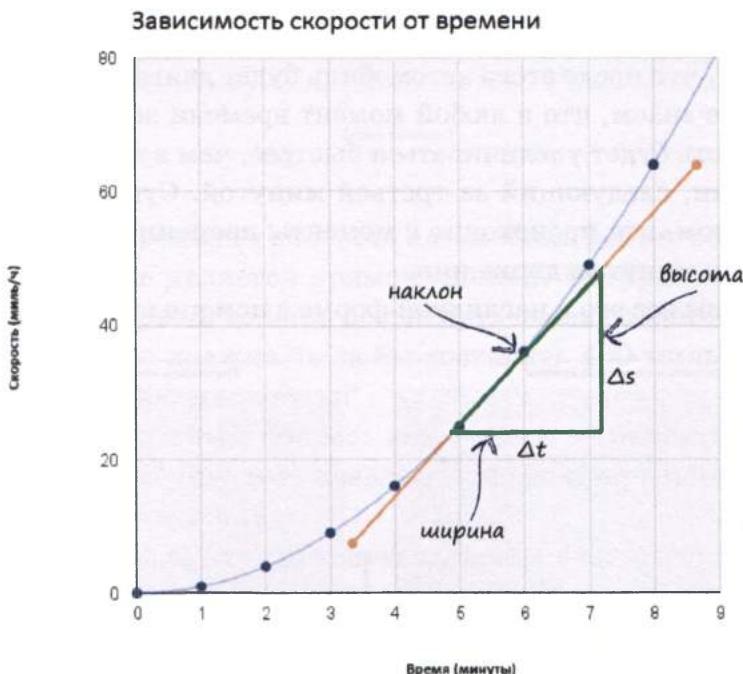
Представим все это в наглядной форме с помощью графика.



Вы видите, что в момент времени 6 минут наклон кривой круче, чем в момент времени 3 минуты. Оба наклона представляют иско-мую скорость изменения скорости движения. Очень важно, чтобы вы это поняли, потому повторим мысль в следующей формулировке: скорость изменения кривой в любой точке определяется ее наклоном в этой точке.

Но как измерить наклон линии, которая искривлена? С прямыми линиями все просто, а вот как быть с кривыми? Мы можем попытаться оценить наклон, прочертив прямую линию, так называемую касательную, которая лишь касается кривой (имеет с ней только одну общую точку) таким образом, чтобы иметь тот же наклон, что и кривая в данной точке. Именно так в действительности люди и поступали, пока не были изобретены другие способы.

Давайте испытаем этот приближенный простой способ хотя бы для того, чтобы лучше понять, к чему он приводит. На следующей иллюстрации представлен график скорости с касательной к кривой в точке, соответствующей шести минутам движения.



Из школьного курса математики нам известно, что для определения наклона, или углового коэффициента, следует разделить приращение вертикальной координаты на приращение горизонтальной координаты. На диаграмме приращение по вертикали (скорость) обозначено как  $\Delta s$ , а приращение по горизонтали (время) — как  $\Delta t$ . Символ  $\Delta$  (читается “дельта”) просто означает небольшое изменение. Поэтому  $\Delta t$  — это небольшое изменение  $t$ .

Наклон определяется отношением  $\Delta s / \Delta t$ . Мы можем выбрать для определения наклона любой треугольник и измерить длину его катетов с помощью линейки. В выбранном мною треугольнике приращение  $\Delta s$  оказалось равным 9,6, а приращение  $\Delta t$  — 0,8. Это дает следующую величину наклона.

скорость изменения = наклон в данной точке

$$\begin{aligned} &= \frac{\Delta s}{\Delta t} \\ &= 9,6 / 0,8 \\ &= 12,0 \end{aligned}$$

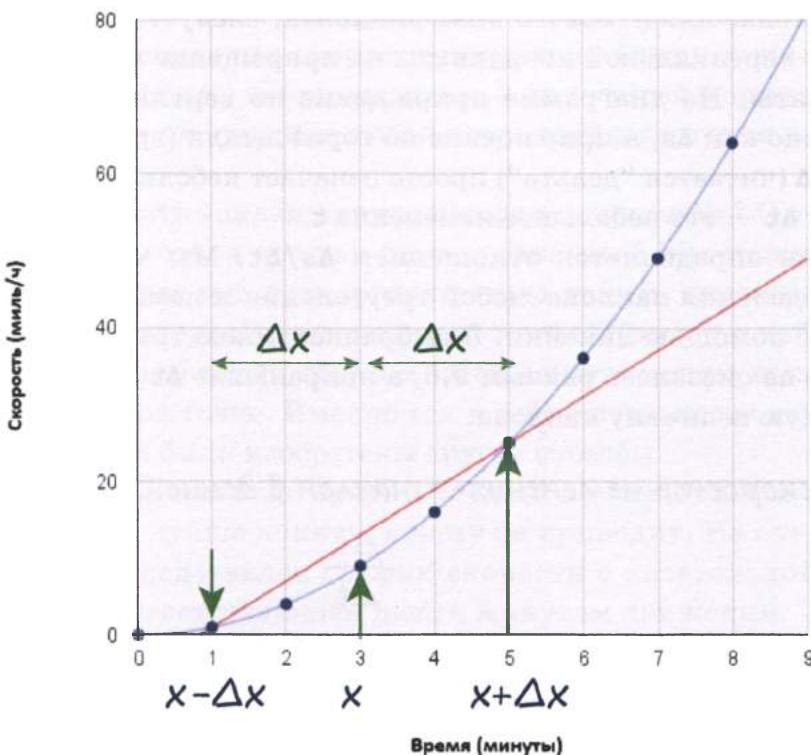
Мы получили очень важный результат! Скорость изменения скорости в момент времени 6 минут составляет 12,0 миль в час за минуту.

Нетрудно заметить, что от способа, основанного на проведении касательной вручную и выполнении измерений с помощью линейки, вряд ли можно ожидать высокой точности. Поэтому мы должны обратиться к более совершенным методам.

## Усовершенствованный способ применения дифференциального исчисления, допускающий автоматизацию

Взгляните на следующий график, на котором проведена другая прямая линия. Она не является касательной, поскольку имеет более одной общей точки с кривой. Но видно, что она в некотором смысле центрирована на моменте времени, соответствующем трем минутам.

### Зависимость скорости от времени



Связь этой прямой с моментом времени 3 минуты действительно существует. Для ее проведения были выбраны моменты времени до и после интересующего нас момента времени  $t=3$ . В данном случае были выбраны точки, отстоящие от точки  $t=3$  на две минуты в большую и меньшую стороны, т.е. этим моментам времени соответствуют точки  $t=1$  и  $t=5$ .

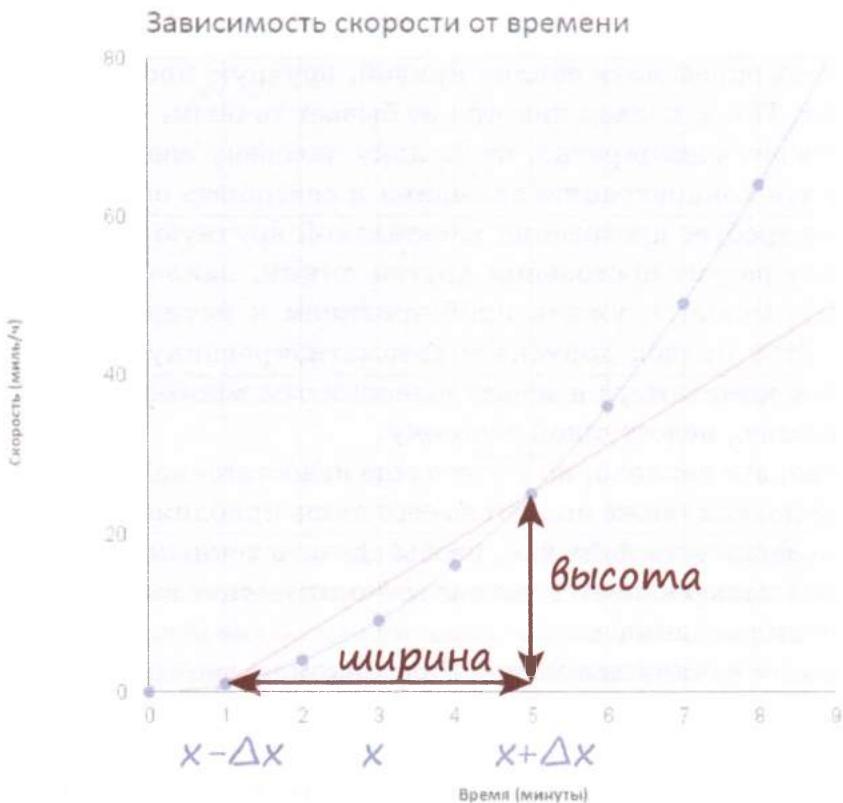
Используя математические обозначения, можно сказать, что  $\Delta x$  равно 2 минуты. Тогда выбранным нам точкам соответствуют координаты  $x - \Delta x$  и  $x + \Delta x$ . Вспомните, что символ  $\Delta$  означает “небольшое изменение”, поэтому  $\Delta x$  — это небольшое изменение  $x$ .

Зачем мы это делаем? Потерпите минутку — очень скоро все прояснится.

Если мы возьмем значения скорости в моменты времени  $x - \Delta x$  и  $x + \Delta x$  и проведем через соответствующие две точки прямую линию, то ее наклон будет примерно равен наклону касательной в средней точке  $x$ .

Вернитесь к предыдущей иллюстрации и взгляните на эту прямую линию. Несомненно, ее наклон не совпадает в точности с наклоном истинной касательной в точке  $x$ , но мы исправим этот недостаток.

Вычислим наклон этой прямой. Мы используем прежний подход и рассчитаем наклон как отношение смещения точки по вертикали к смещению по горизонтали. Следующая иллюстрация проясняет, что в данном случае представляют собой эти смещения.



Смещение по вертикали — это разность между значениями скорости в точках  $x + \Delta x$  и  $x - \Delta x$ , соответствующих пяти минутам и одной минуте движения. Эти скорости нам известны:  $5^2 = 25$  и  $1^2 = 1$ , поэтому разность составляет 24. Смещение по горизонтали — это просто расстояние между точками  $x + \Delta x$  и  $x - \Delta x$ , т.е.  $5 - 1 = 4$ . Следовательно, имеем:

$$\text{наклон} = \frac{\text{высота}}{\text{ширина}}$$

$$= 24 / 4$$

$$= 6$$

Таким образом, наклон прямой линии, являющейся приближением к касательной в точке  $t=3$  минуты, составляет 6 миль в час за минуту.

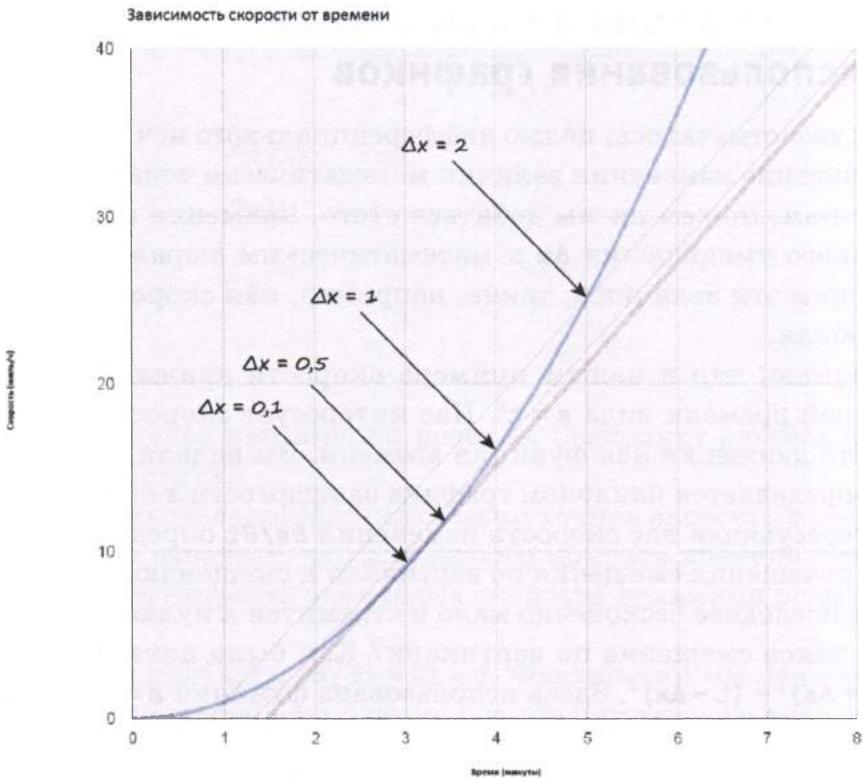
Сделаем паузу и осмыслим полученные результаты. Сначала мы попытались определить наклон кривой, вручную проведя касательную к ней. Такой подход никогда не бывает точным, и мы не можем повторять его многократно, поскольку человеку свойственно уставать, терять концентрацию внимания и совершать ошибки. Второй подход не требует проведения касательной вручную и вместо этого предлагает рецепт построения другой линии, наклон которой, по-видимому, может служить приближением к истинному наклону кривой. Этот подход допускает автоматизированную реализацию с помощью компьютера и может выполняться многократно с огромной скоростью, недоступной человеку.

Конечно, это неплохо, но и этого еще недостаточно!

Второй подход также является всего лишь приближенным. Можно ли его усовершенствовать так, чтобы сделать точным? В конце концов, нашей целью является точное математическое описание скорости изменения величин.

Вот здесь и начинается магия! Я познакомлю вас с одним из самых элегантных инструментов, разработанных математиками.

Что случится, если мы уменьшим величину смещения. Иными словами, что произойдет, если уменьшить  $\Delta x$ ? Следующая иллюстрация демонстрирует несколько приближений в виде наклонных прямых, соответствующих уменьшению  $\Delta x$ .



Мы провели линии для  $\Delta x=2,0$ ,  $\Delta x=1,0$ ,  $\Delta x=0,5$  и  $\Delta x=0,1$ . Вы видите, что эти линии постепенно приближаются к интересующей нас точке  $x=3$ . Нетрудно сообразить, что по мере уменьшения  $\Delta x$  прямые линии будут все более и более приближаться к истинной касательной.

При бесконечно малой величине  $\Delta x$  линия приблизится к истинной касательной на бесконечно малое расстояние. Это очень круто!

Идея постепенного улучшения первоначального приближенного решения путем уменьшения отклонений необычайно плодотворна. Она позволяет математикам решать задачи, непосредственное решение которых наталкивается на значительные трудности. При таком подходе к решению приближаются постепенно, словно крадучись, вместо того чтобы атаковать его прямо в лоб!

# Дифференциальное исчисление без использования графиков

Как уже отмечалось, целью дифференциального исчисления является описание изменения величин математически точным способом. Посмотрим, можем ли мы добиться этого, применяя идею последовательного уменьшения  $\Delta x$  к математическим выражениям, определяющим эти величины, такие, например, как скорость движения автомобиля.

Напомню, что в нашем примере скорость движения является функцией времени вида  $s = t^2$ . Нас интересует скорость изменения скорости движения как функция времени. Вы видели, что эта величина определяется наклоном графика зависимости  $s$  от  $t$ .

Интересующая нас скорость изменения  $\frac{\partial s}{\partial t}$  определяется величиной отношения смещения по вертикали к смещению по горизонтали, где последнее бесконечно мало и стремится к нулю.

Что такое смещение по вертикали? Как было показано раньше, это  $(t + \Delta x)^2 - (t - \Delta x)^2$ . Здесь использована функция  $s = t^2$ , где  $t$  принимает значения, немного меньшие и немного большие, чем в интересующей нас точке. Это “немного” равно  $\Delta x$ .

Что такое смещение по горизонтали? Как вы видели раньше, это просто расстояние между точками  $(t + \Delta x)$  и  $(t - \Delta x)$ , которое равно  $2\Delta x$ .

Мы уже почти у цели:

$$\begin{aligned}\frac{\delta s}{\delta t} &= \frac{\text{высота}}{\text{ширина}} \\ &= \frac{(t + \Delta x)^2 - (t - \Delta x)^2}{2\Delta x}\end{aligned}$$

Раскроем и упростим это выражение:

$$\begin{aligned}\frac{\delta s}{\delta t} &= \frac{t^2 + \Delta x^2 + 2t\Delta x - t^2 - \Delta x^2 + 2t\Delta x}{2\Delta x} \\ &= \frac{4t\Delta x}{2\Delta x} \\ \frac{\delta s}{\delta t} &= 2t\end{aligned}$$

В данном случае нам крупно повезло, поскольку алгебра необычайно все упростила.

Итак, мы это сделали! Математически точная скорость изменения  $\frac{\partial s}{\partial t} = 2t$ . Это означает, что для любого момента времени  $t$  мы можем вычислить скорость изменения скорости движения по формуле  $\frac{\partial s}{\partial t} = 2t$ .

При  $t=3$  мы получаем  $\frac{\partial s}{\partial t} = 2t = 6$ . Фактически мы подтвердили этот результат еще раньше с помощью приближенного метода. Для  $t=6$  получаем  $\frac{\partial s}{\partial t} = 2t = 12$ , что также согласуется с найденным ранее результатом.

А что насчет ста минут? В этом случае  $\frac{\partial s}{\partial t} = 2t = 200$  миль в час за минуту. Это означает, что спустя сто минут от начала движения машина будет ускоряться со скоростью 200 миль в час за минуту.

Сделаем небольшую паузу и немного поразмышляем над величием и красотой того, что нам удалось сделать. Мы получили математическое выражение, с помощью которого можем точно определить скорость изменения скорости движения автомобиля в любой момент времени. При этом, в полном соответствии с нашими рассуждениями в начале приложения, мы видим, что эти изменения  $s$  действительно зависят от времени.

Нам повезло, что алгебра все упростила, но в силу простоты выражения  $s = t^2$  нам не представилась возможность проверить на практике идею устремления  $\Delta x$  к нулю. В связи с этим полезно рассмотреть другой пример, в котором скорость движения автомобиля описывается несколько более сложной формулой:

$$s = t^2 + 2t$$

$$\frac{\delta s}{\delta t} = \frac{\text{высота}}{\text{ширина}}$$

А что теперь означает вертикальное смещение? Это разность между скоростью  $s$ , рассчитанной в момент времени  $t + \Delta x$ , и скоростью  $s$ , рассчитанной в момент времени  $t - \Delta x$ . Найдем эту разность путем несложных вычислений:  $(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)$ .

А что насчет горизонтального смещения? Это просто расстояние между точками  $(t + \Delta x)$  и  $(t - \Delta x)$ , которое по-прежнему равно  $2\Delta x$ :

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^2 + 2(t + \Delta x) - (t - \Delta x)^2 - 2(t - \Delta x)}{2\Delta x}$$

Раскроем и упростим это выражение:

$$\frac{\delta s}{\delta t} = \frac{t^2 + \Delta x^2 + 2t\Delta x + 2t + 2\Delta x - t^2 - \Delta x^2 + 2t\Delta x - 2t + 2\Delta x}{2\Delta x}$$

$$= \frac{4t\Delta x + 4\Delta x}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 2t + 2$$

Это замечательный результат! К сожалению, алгебра и в этот раз немного перестаралась, упростив нам работу. Но этот пример не был напрасным, поскольку здесь уже вырисовывается закономерность, к которой мы еще вернемся.

Попробуем рассмотреть еще один, чуть более сложный, пример. Предположим, что скорость движения автомобиля описывается кубической функцией времени:

$$s = t^3$$

$$\frac{\delta s}{\delta t} = \frac{\text{высота}}{\text{ширина}}$$

$$\frac{\delta s}{\delta t} = \frac{(t + \Delta x)^3 - (t - \Delta x)^3}{2\Delta x}$$

Раскроем и упростим это выражение:

$$\frac{\delta s}{\delta t} = \frac{t^3 + 3t^2\Delta x + 3t\Delta x^2 + \Delta x^3 - t^3 + 3t^2\Delta x - 3t\Delta x^2 + \Delta x^3}{2\Delta x}$$

$$= \frac{6t^2\Delta x + 2\Delta x^3}{2\Delta x}$$

$$\frac{\delta s}{\delta t} = 3t^2 + \Delta x^2$$

Это уже намного интереснее! Мы получили результат, содержащий  $\Delta x$ , тогда как раньше эти члены взаимно сокращались.

Вспомните, что корректное значение наклона получается лишь в том случае, если величина  $\Delta x$  уменьшается, становясь бесконечно малой.

А сейчас смотрите! Что произойдет с величиной  $\Delta x$  в выражении  $\frac{\delta s}{\delta t} = 3t^2 + \Delta x^2$ , если она становится все меньшей и меньшей? Она исчезнет! Если для вас это неожиданность, то представьте, что величина  $\Delta x$  очень мала. Напрягитесь и представьте, что она еще меньше. Потом представьте, что на самом деле она еще меньше... И этот процесс мысленного уменьшения  $\Delta x$  вы могли бы продолжать до бесконечности, устремляя ее к нулю. Поэтому давайте проявим решимость и сразу же перейдем к нулю без лишней сути.

В результате мы получаем искомый математически точный ответ:

$$\frac{\delta s}{\delta t} = 3t^2$$

Это фантастический результат, и на этот раз он был получен с использованием мощного математического инструмента, что оказалось совсем несложным.

## Закономерности

Конечно, это весьма интересное занятие — находить производные, используя приращения наподобие  $\Delta x$ , и смотреть, что произойдет, если делать их все меньшими и меньшими. Но зачастую можно получить результат, не выполняя всю эту работу.

Посмотрите на приведенные ниже формулы и постарайтесь увидеть в них закономерность.

$$s = t^2 \quad \xrightarrow{\hspace{1cm}} \quad \frac{\delta s}{\delta t} = 2t$$

$$s = t^2 + 2t \quad \xrightarrow{\hspace{1cm}} \quad \frac{\delta s}{\delta t} = 2t + 2$$

$$s = t^3 \quad \xrightarrow{\hspace{1cm}} \quad \frac{\delta s}{\delta t} = 3t^2$$

Вы видите, что производная функции  $t$  представляет собой ту же функцию, но с понижением на единицу каждой степени  $t$ . Поэтому  $t^4$  превращается в  $t^3$ , а  $t^7$  превратилось бы в  $t^6$  и т.д. Это очень просто! А если вы вспомните, что  $t$  — это  $t^1$ , то в производной это превращается в  $t^0$ , т.е. в 1.

Свободные постоянные члены, такие как 3, 4 или 5, просто исчезают. Постоянные переменные, не являющиеся коэффициентами, такие как  $a$ ,  $b$  или  $c$ , также исчезают, поскольку скорость их изменения также нулевая. Именно поэтому их называют константами.

Но погодите, ведь  $t^2$  превращается в  $2t$ , а не просто в  $t$ , а  $t^3$  превращается в  $3t^2$ , а не просто в  $t^2$ . Это общее правило: степень переменной, прежде чем уменьшиться на единицу, становится коэффициентом.

Поэтому 5 в  $2t^5$  используется в качестве дополнительного коэффициента перед уменьшением степени на единицу:  $5 \cdot 2t^4 = 10t^4$ .

Приведенная ниже формула суммирует все, что было сказано о дифференцировании степеней, в виде следующего правила:

$$y = ax^n \rightarrow \frac{\delta y}{\delta n} = nax^{n-1}$$

Испытаем эту формулу на дополнительных примерах только ради того, чтобы набить руку в использовании этого нового приема:

$$s = t^5 \rightarrow \frac{\delta s}{\delta t} = 5t^4$$

$$s = 6t^6 + 9t + 4 \rightarrow \frac{\delta s}{\delta t} = 36t^5 + 9$$

$$s = t^3 + c \rightarrow \frac{\delta s}{\delta t} = 3t^2$$

Это правило пригодится вам во многих случаях, а зачастую ничего другого вам и не потребуется. Верно и то, что правило применимо только к полиномам, т.е. к выражениям, состоящим из переменных в различных степенях, как, например, выражение  $y = ax^3 + bx^2 + cx + d$ , но не к функциям вида  $\sin(x)$  или  $\cos(x)$ . Это не является существенным недостатком, поскольку в огромном количестве случаев вам вполне хватит правила дифференцирования степеней.

Однако для нейронных сетей нам понадобится еще один инструмент, о котором сейчас пойдет речь.

# Функции функций

Представьте, что в функции

$$f = y^2$$

переменная  $y$  сама является функцией:

$$y = x^3 + x$$

При желании можно переписать эту формулу в виде  $f = (x^3 + x)^2$ .

Как  $f$  изменяется с изменением  $y$ ? То есть что собой представляет производная  $\partial f / \partial y$ ? Получить ответ на этот вопрос не составляет труда, поскольку для этого достаточно применить только что полученное нами правило дифференцирования степенных выражений, поэтому  $\partial f / \partial y = 2y$ .

Но возникает более интересный вопрос: как изменяется  $f$  при изменении  $x$ ? Ну хорошо, мы могли бы раскрыть выражение  $f = (x^3 + x)^2$  и применить уже знакомый подход. Только ни в коем случае не считайте наивно, что производная от  $(x^3 + x)^2$  — это  $2(x^3 + x)$ .

Если бы мы проделали множество подобных вычислений прежним трудоемким способом, предполагающим устремление приращений к нулю в результирующих выражениях, то рано или поздно мы подметили бы еще одну закономерность. Я сразу же дам вам готовый рецепт.

Вот как выглядит новая закономерность.

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} \cdot \frac{\delta y}{\delta x}$$

Это очень мощный результат, который называется **цепным правилом**.

В соответствии с этим правилом нахождение производной в подобных случаях осуществляется поэтапно. Может оказаться так, что для нахождения производной  $\frac{\partial f}{\partial x}$  проще найти производные  $\frac{\partial f}{\partial y}$  и  $\frac{\partial y}{\partial x}$ . Если последние две производные действительно вычисляются очень просто, то с помощью этого приема удается находить производные, определить которые другими способами практически невозможно. Цепное правило позволяет разбивать трудные задачи на более легкие.

Рассмотрим следующий пример и применим к нему цепное правило:

$$f = y^2 \quad \text{и} \quad y = x^3 + x$$

$$\frac{\delta f}{\delta x} = \frac{\delta f}{\delta y} \cdot \frac{\delta y}{\delta x}$$

Мы разбили задачу на две простые части. Первая часть дает  $(\frac{\partial f}{\partial y}) = 2y$ , вторая —  $(\frac{\partial y}{\partial x}) = 3x^2 + 1$ . Объединяя эти части с помощью цепного правила, получаем

$$\frac{\delta f}{\delta x} = (2y) * (3x^2 + 1)$$

Мы знаем, что  $y = x^3 + x$ , поэтому можем получить выражение, содержащее только  $x$ :

$$\frac{\delta f}{\delta x} = (2(x^3 + x)) * (3x^2 + 1)$$

$$\frac{\delta f}{\delta x} = (2x^3 + 2x)(3x^2 + 1)$$

Магия!

Возможно, вас так и подмывает спросить: а почему бы не представить  $f$  в виде функции, зависящей только от  $x$ , и применить простое правило дифференцирования степеней к результирующему полиному? Мы могли бы это сделать, но тогда я не продемонстрировал бы вам, как работает цепное правило, которое позволяет разгрызать более твердые орешки.

Рассмотрим еще один пример, на этот раз последний, который демонстрирует, как обращаться с переменными, не зависящими от других переменных.

Предположим, имеется функция

$$f = 2xy + 3x^2z + 4z$$

В ней переменные  $x$ ,  $y$  и  $z$  не зависят одна от другой. Что мы подразумеваем под независимостью переменных? Под этим подразумевается, что каждая из переменных  $x$ ,  $y$  и  $z$  может принимать любые значения, какими бы ни были значения остальных переменных — их изменения на нее не влияют. В предыдущем примере это было не так, поскольку значение  $y$  определялось значением выражения  $x^3 + x$ , а значит, переменная  $y$  зависела от  $x$ .

Что такое  $\frac{\partial f}{\partial x}$ ? Рассмотрим каждый член длинного полинома по отдельности. Первый член — это  $2xy$ , поэтому его производная равна  $2y$ . Почему так просто? Да потому, что  $y$  не зависит от  $x$ . Когда мы интересуемся величиной  $\frac{\partial f}{\partial x}$ , нас интересует, как изменяется  $f$  при изменении  $x$ . Если переменная  $y$  не зависит от  $x$ , то с ней можно обращаться как с константой. На ее месте могло бы быть любое другое число, например 2, 3 или 10.

Идем дальше. Следующий член выражения —  $3x^2z$ . Применяя правило понижения степеней, получаем  $2 \cdot 3xz$  или  $6xz$ . Мы рассматриваем  $z$  как обычную константу, значением которой может быть 2, 4 или 100, поскольку  $x$  и  $z$  не зависят друг от друга. Изменение  $z$  не влияет на  $x$ .

Последний член,  $4z$ , вообще не содержит  $x$ . Поэтому он полностью исчезает, так как мы рассматриваем его как постоянное число, которым, например, могло бы быть 2 или 4.

Вот как выглядит окончательный ответ:

$$\frac{\delta f}{\delta x} = 2y + 6xz$$

В этом примере была важна возможность уверенно игнорировать переменные, о которых известно, что они являются независимыми. Это значительно упрощает дифференцирование довольно сложных выражений, в чем часто возникает необходимость в ходе анализа нейронных сетей.

## Вы освоили дифференциальное исчисление!

Если вам удалось одолеть материал, изложенный в этом приложении, примите мои поздравления!

Я постарался донести до вас, в чем состоит суть дифференциального исчисления и как оно возникло на основе постепенного улучшения приближенных решений. Всегда пытайтесь применять описанные в данном приложении методы, если другие способы решения задачи не приводят к успеху.

Используя метод дифференцирования выражений путем понижения степеней переменных и применения цепного правила для нахождения производных сложных функций, вы сможете многое узнать о природе и механизмах функционирования нейронных сетей.

Желаю вам максимально эффективно использовать этот мощный инструмент, которым вы теперь владеете!



## ПРИЛОЖЕНИЕ Б

# Нейронная сеть на Raspberry Pi

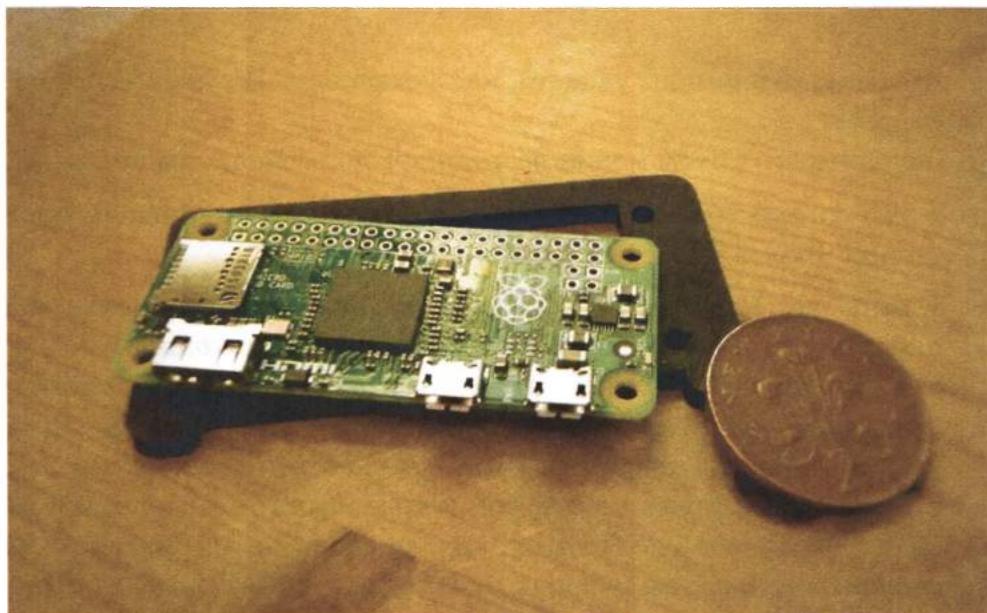
В этом приложении вы узнаете, как установить IPython на устройстве Raspberry Pi.

Необходимость в этом может возникнуть по нескольким причинам.

- Компьютеры Raspberry Pi очень дешевые и доступные по сравнению с более дорогими ноутбуками.
- Компьютер Raspberry Pi работает под управлением **бесплатной** операционной системы Linux с **открытым исходным кодом**, для которой доступно множество всевозможных бесплатных программ, включая Python. Открытость исходного кода чрезвычайно важна, поскольку это позволяет разобраться в том, как функционирует та или иная программа, и поделиться результатами своей работы с другими людьми, которые смогут воспользоваться ими в своих проектах. Это важно и для образовательных целей, ведь, в отличие от коммерческого программного обеспечения, открытый исходный код свободно доступен для изучения.
- В силу описанных, а также множества других причин устройства Raspberry Pi получили широкую популярность в качестве школьных и домашних компьютеров для детей, увлекающихся созданием компьютерных программ или программно-аппаратных систем.
- Компьютеры Raspberry Pi не такие мощные, как более дорогие компьютеры и ноутбуки. Поэтому интересно продемонстрировать, что даже этот фактор не мешает реализовать на Raspberry Pi нейронную сеть с помощью Python.

Я буду использовать модель Raspberry Pi Zero, поскольку она еще дешевле и миниатюрнее, чем обычные устройства Raspberry Pi, и это делает задачу развертывания на ней нейронной сети еще более интересной. Стоит эта модель около 5 долларов. Это не опечатка!

Ниже представлена фотография моего устройства с двухпенсововой монетой для сравнения.



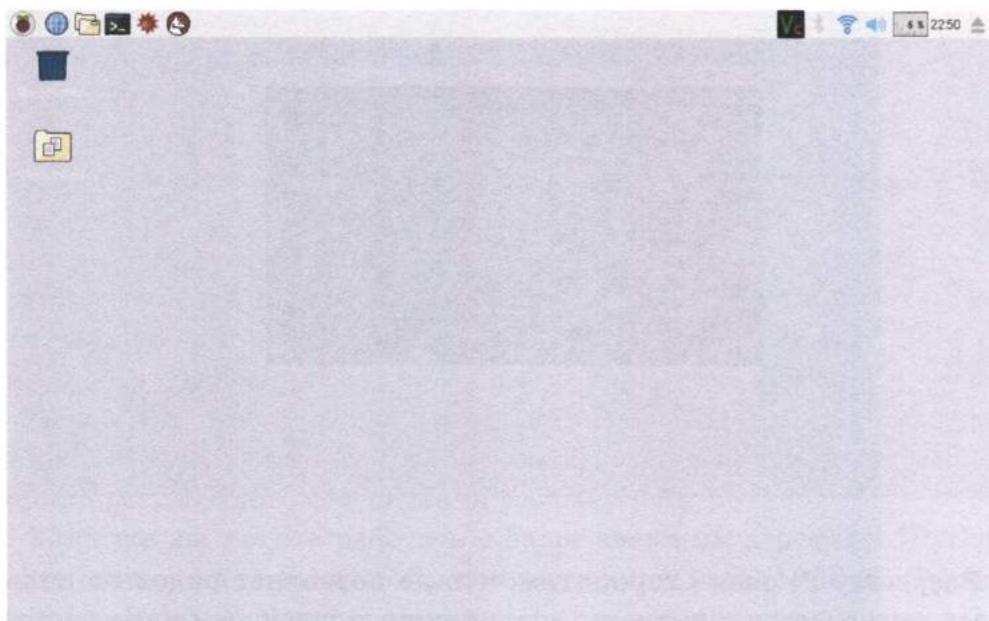
## Установка IPython

Далее предполагается, что питание вашего Raspberry Pi включено, а клавиатура, мышь, дисплей и подключение к Интернету работают нормально.

Существует несколько дистрибутивов операционных систем для Raspberry Pi, но мы будем ориентироваться на Raspbian — версию популярного дистрибутива Debian Linux, оптимизированную для аппаратных возможностей Raspberry Pi и доступную для загрузки по следующему адресу:

<https://www.raspberrypi.org/downloads/raspbian/>

Ниже показан вид рабочего стола после запуска Raspberry Pi. Я убрал фоновое изображение, чтобы оно не отвлекало внимание.

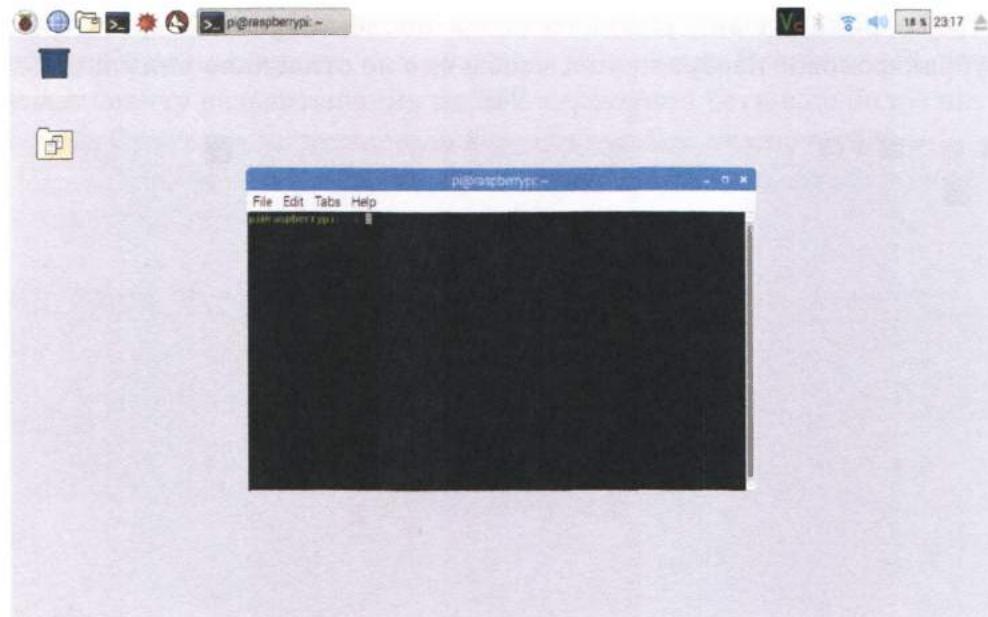


В левом верхнем углу видны кнопки меню, а также другие значки.

Мы собираемся установить IPython, чтобы иметь возможность работать с помощью дружественного интерфейса блокнотов через веб-браузер, а не в режиме командной строки.

Для установки IPython нам все-таки придется использовать командную строку, но сама эта процедура очень простая, и ее придется выполнить только один раз.

Откройте приложение Terminal, представленное в верхней части окна значком с изображением черного экрана. При наведении на него указателя мыши появляется подсказка “Terminal”. Когда вы запустите это приложение, откроется окно для ввода команд.



Raspberry Pi очень хорош тем, что не позволяет рядовым пользователям вводить команды, приводящие к глубоким изменениям в системе. Для этого необходимо иметь специальные привилегии. Введите в окне терминала следующую команду:

```
sudo su -
```

Вместо символа доллара (\$), которым до этого заканчивалась подсказка для ввода команд, должен появиться символ решетки (#). Это свидетельствует о том, что теперь вы обладаете административными привилегиями и должны внимательно относиться к выбору вводимых команд.

Следующие команды актуализируют список текущего программного обеспечения Raspberry Pi, а затем обновляют установленные вами программы, загружая дополнительные программные компоненты.

```
apt-get update  
apt-get dist-upgrade
```

Если ваше программное обеспечение в последнее время не обновлялось, то, вероятно, эта процедура потребуется некоторым программам. В таком случае вы увидите, как на экране промелькнет множество текстовых строк. На них можно не обращать внимания. От вас может потребоваться подтвердить обновление нажатием клавиши <Y>.

Обновив состояние системы, введите команду для получения IPython. Имейте в виду, что на момент написания книги программные пакеты Raspbian не содержали версий IPython, позволяющих работать с размещенными на сайте GitHub для всеобщего доступа блокнотами, которые мы ранее создали. Если бы они их содержали, то достаточно было бы просто ввести команду apt-get install ipython3 ipython3-notebook или аналогичную ей.

Если вы не хотите запускать блокноты из GitHub, можете спокойно использовать более старые версии IPython и блокнота из репозитория программного обеспечения Raspberry Pi.

Если же вы хотите работать с более свежими версиями IPython и блокнота, то для того, чтобы получить их из каталога PyPi (Python Package Index — каталог пакетов Python), вам придется использовать некоторые команды “pip” в дополнение к командам apt-get. В этом случае программное обеспечение будет управляться Python, а не менеджером программного обеспечения операционной системы. Следующие команды обеспечат вас всем необходимым.

```
apt-get install python3-matplotlib  
apt-get install python3-scipy  
pip3 install jupyter
```

Работа будет выполнена, как только на экране промелькнут последние строки текста. Скорость выполнения зависит от конкретной модели Raspberry Pi и скорости вашего интернет-соединения. Вот как выглядел мой экран после выполнения этих команд.



Как правило, в Raspberry Pi используются карты памяти, так называемые *SD-карты*, наподобие тех, которые вы, возможно, используете в своей цифровой камере. Они обеспечивают меньший объем памяти, чем обычные компьютеры. Удалите программные пакеты, которые были загружены для обновления вашего Raspberry Pi, использовав для этого следующую команду:

```
apt-get clean
```

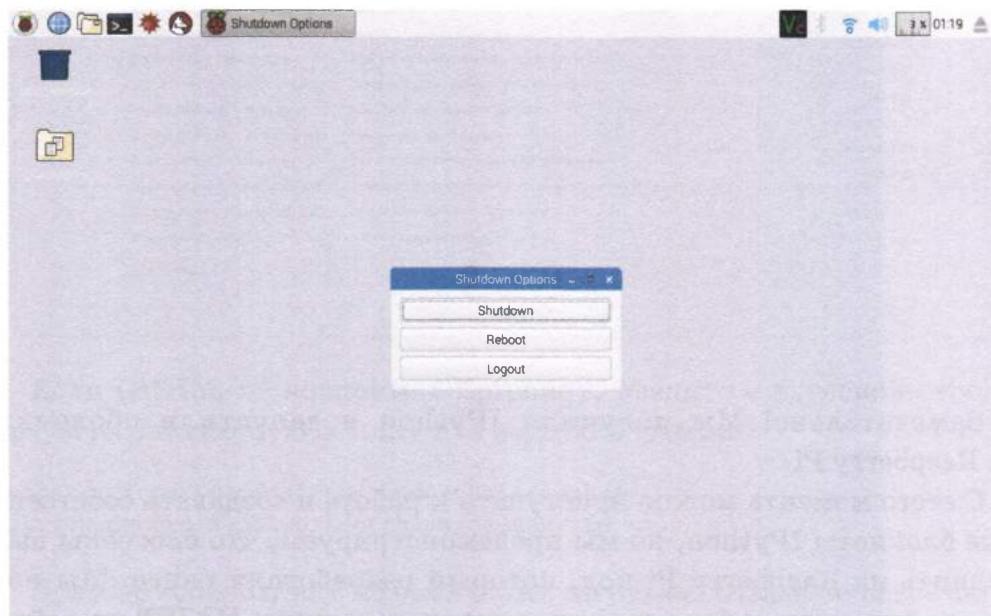
В последних версиях Raspbian браузер Epiphany заменен браузером Chromium (версия с открытым исходным кодом популярного браузера Chrome). Но Epiphany гораздо легче тяжеловесного Chromium и работает лучше с крохотным Raspberry Pi Zero. Чтобы установить его в качестве браузера по умолчанию для блокнотов IPython, используйте следующую команду:

```
update-alternatives --config x-www-browser
```

Команда уведомит вас о текущей установленной версии браузера, используемой по умолчанию, и запросит разрешение на установку

новой версии. Выберите номер, соответствующий Epiphany, и от вас больше ничего не потребуется.

Теперь все готово. Перезапустите Raspberry Pi в том случае, если обновление повлекло за собой некие глубокие изменения, такие как обновление ядра. Для этого выберите пункт Shutdown основного меню в левом верхнем углу, а затем пункт Reboot, как показано ниже.



После повторного запуска Raspberry Pi запустите Python, выполнив в окне терминала следующую команду:

```
jupyter-notebook
```

Это приведет к автоматическому запуску веб-браузера с открытой основной страницей IPython, на которой можно создавать новые блокноты IPython. Jupyter Notebook — это новое программное обеспечение для выполнения блокнотов. Ранее приходилось выполнять команду `ipython3 notebook`, которая будет еще работать в течение переходного периода. Ниже показана основная начальная страница IPython.



Замечательно! Мы получили IPython и запустили оболочку на Raspberry Pi.

С этого момента можно приступать к работе и создавать собственные блокноты IPython, но мы продемонстрируем, что способны выполнить на Raspberry Pi код, который разработали ранее. Мы получим блокноты и базу данных рукописных цифр MNIST на сайте GitHub. Откройте в браузере новую вкладку и перейдите по следующей ссылке:

<https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork>

Вы увидите страницу проекта GitHub (см. ниже). Загрузите файлы, щелкнув на кнопке **Clone or Download** и выбрав вариант **Download ZIP**.

The screenshot shows a terminal window and a web browser. The terminal window is titled 'pi@raspberrypi ~' and contains some command-line text. The web browser is displaying a GitHub repository page for 'makeyourownneuralnetwork/makeyourownneuralnetwork'. The repository has 43 commits, 1 branch, 0 releases, and 1 contributor. The license is GPL-2.0. The commit list includes files such as 'mnist\_dataset', 'my\_own\_images', 'LICENSE', 'README.md', and various Python scripts like 'part1\_mnist\_data\_set.py', 'part2\_neural\_network.py', etc. Each commit has a timestamp and a brief description.

Если GitHub не воспримет Epiphany, введите в адресной строке браузера следующую ссылку для загрузки файлов:

<https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/archive/master.zip>

Браузер сообщит вам об окончании загрузки. Откройте новое окно терминала и введите следующие команды, чтобы распаковать файлы, а затем освободить место, удалив zip-файл.

```
unzip Downloads/makeyourownneuralnetwork-master.zip  
rm -f Downloads/makeyourownneuralnetwork-master.zip
```

Файлы будут распакованы в каталог makeyourownneuralnetwork-master. При желании можете присвоить ему более короткое имя, но делать это вовсе необязательно.

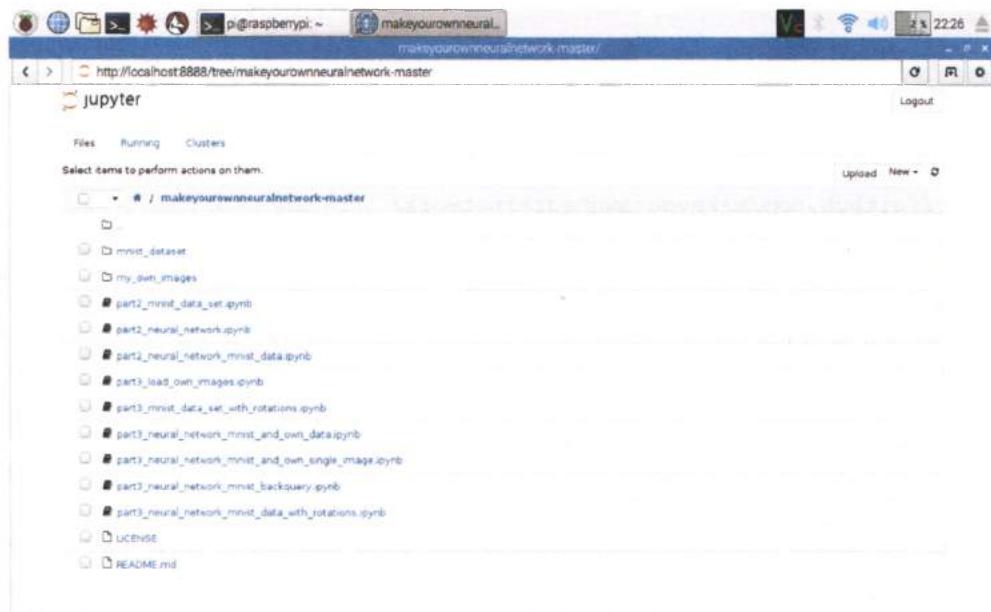
На сайте GitHub содержатся лишь сокращенные версии наборов данных MNIST, поскольку сайт не позволил бы мне разместить файлы большего размера. Чтобы получить полный набор данных, введите в том же окне терминала следующие команды для перехода в каталог mnist\_dataset и получения тренировочного и тестового наборов данных в CSV-формате.

```
cd makeyourownneuralnetwork-master/mnist_dataset  
wget -c http://pjreddie.com/media/files/mnist_train.csv  
wget -c http://pjreddie.com/media/files/mnist_test.csv
```

Загрузка файлов может занять некоторое время, которое зависит от скорости вашего интернет-соединения и конкретной модели Raspberry Pi.

Теперь вы располагаете всеми необходимыми блокнотами и данными MNIST. Закройте это окно терминала, но не то, из которого запускали IPython.

Вернувшись в окно браузера с начальной страницей IPython, вы увидите в списке новую папку `makeyourownneuralnetwork-master`. Откройте эту папку, щелкнув на ней. Теперь вы сможете открыть любой блокнот, как смогли бы это сделать на любом другом компьютере. Ниже показаны блокноты, хранящиеся в указанной папке.



# Проверка работоспособности программ

Прежде чем приступить к тренировке и тестированию нейронной сети, убедимся в работоспособности кода, выполняющего такие операции, как чтение файлов и вывод изображений. Откройте блокнот part3\_mnist\_data\_set\_with\_rotations.ipynb, реализующий эти операции. Открывшийся и готовый к выполнению блокнот должен выглядеть так.

The screenshot shows a Jupyter Notebook window titled "part3\_mnist\_data\_set\_with\_rotations". The code in cell [1] is a multi-line comment explaining the purpose of the script, which is to work with the MNIST dataset by rotating training images to create more examples. The code in cell [2] imports numpy and matplotlib.pyplot, and enables inline plotting. Cell [3] imports scipy.ndimage. Cells [4] through [6] demonstrate reading a CSV file named "mnist\_train\_100.csv" into a list, splitting its contents into individual records, and scaling the input values from 0.01 to 1.00. Cell [7] prints the scaled input array. The notebook interface includes a toolbar at the top and a status bar at the bottom indicating the last checkpoint was 21 hours ago.

```
In [1]: # python notebook for Make Your Own Neural Network
# working with the MNIST data set
# this code demonstrates rotating the training images to create more examples
#
# (c) Tariq Rashid, 2016
# license is GPLv2

In [2]: import numpy
import matplotlib.pyplot
%matplotlib inline

In [3]: import scipy.ndimage

In [4]: # open the CSV file and read its contents into a list
data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
data_list = data_file.readlines()
data_file.close()

In [5]: # which record will be use
record = 6

In [6]: # scale input to range 0.01 to 1.00
all_values = data_list[record].split(',')
scaled_input = ((numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01).reshape(28,28)

In [7]: print(numpy.min(scaled_input))
```

Выполните содержащиеся в блокноте инструкции, выбрав в меню Cell пункт Run All. Спустя некоторое время, превышающее то, которое потребовалось бы в случае современного ноутбука, вы должны увидеть изображения повернутых цифр.

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with various icons. Below it, the title bar says 'http://localhost:8888/notebooks/makeyourownneuralnetwork-master/part3\_mnist\_data\_set\_with\_rotations.ipynb'. The main area has a header 'jupyter part3\_mnist\_data\_set\_with\_rotations Last Checkpoint: 21 hours ago (autosaved)'. On the right, there are buttons for 'Logout' and 'Python 3'. The notebook contains two code cells and their outputs:

```
In [11]: # plot the +10 degree rotated variation
matplotlib.pyplot.imshow(inputs_plus10_img, cmap='Greys', interpolation='None')
Out[11]: <matplotlib.image.AxesImage at 0xaea9a8b0>
```

```
In [12]: # plot the -10 degree rotated variation
matplotlib.pyplot.imshow(inputs_minus10_img, cmap='Greys', interpolation='None')
```

Таким образом, мы убедились в работоспособности на Raspberry Pi таких средств, как загрузка данных из файла, импорт модулей расширения Python для работы с массивами и изображениями, а также графический вывод изображений.

Выберите в меню File этого блокнота пункт Close and Halt. Именно так, а не простым закрытием вкладки браузера, вы должны закрывать свои блокноты.

## Тренировка и тестирование нейронной сети

Приступим к тренировке нейронной сети. Откройте блокнот `part2_neural_network_mnist_data.ipynb`. В нем представлена упрощенная версия нашей программы, лишенная таких возможностей, как вращение изображений. Поскольку наш Raspberry Pi работает гораздо медленнее типичного ноутбука, мы уменьшим некоторые параметры для снижения объема вычислений, чтобы можно было сразу же убедиться в работоспособности кода, а не потратить несколько часов лишь для того, чтобы обнаружить, что он не работает.

Я уменьшил количество скрытых узлов до 10, а количество эпох — до 1. Я по-прежнему использовал полные тренировочные и тестовые

наборы данных MNIST, а не ранее созданные уменьшенные подмножества. Чтобы запустить программу, выберите в меню Cell пункт Run All. Теперь вам остается только ждать.

Обычно на выполнение подобных вычислений на моем ноутбуке уходит около минуты, но в данном случае для завершения работы программе потребовалось около **25 минут**. В целом это не так уж и плохо, если учесть, что Raspberry Pi Zero стоит примерно в 400 раз дешевле, чем мой ноутбук. Я был готов к тому, что программа завершится только к утру.

The screenshot shows a Jupyter Notebook interface running on a Raspberry Pi. The title bar indicates the notebook is titled 'part2\_neural\_network\_mnist\_data'. The code cell contains the following Python script:

```
all_values = record.split(',')
# correct answer is first value
correct_label = int(all_values[0])
# scale and shift the inputs
inputs = (numpy.asarray(all_values[1:]) / 255.0 * 0.99) + 0.01
# query the network
outputs = n.query(inputs)
# the index of the highest value corresponds to the label
label = numpy.argmax(outputs)
# append correct or incorrect to list
if (label == correct_label):
    # network's answer matches correct answer. add 1 to scorecard
    scorecard.append(1)
else:
    # network's answer doesn't match correct answer. add 0 to scorecard
    scorecard.append(0)
pass

In [9]: # calculate the performance score, the fraction of correct answers
scorecard_array = numpy.asarray(scorecard)
print ("performance = ", scorecard_array.sum() / scorecard_array.size)
performance =  0.8536
```

The output cell below the code cell shows the result of the print statement: 'performance = 0.8536'.

## Успех Raspberry Pi

Только что мы продемонстрировали, что функциональных возможностей даже такого миниатюрного компьютера, как Raspberry Pi Zero стоимостью всего 5 долларов, достаточно для полноценной работы с блокнотами IPython и создания кода, позволяющего тренировать и тестировать нейронные сети. Просто все выполняется немного медленнее!