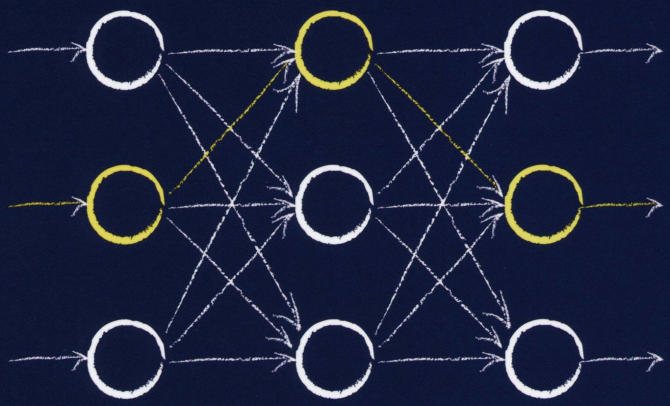
ПОЛНОЦВЕТНОЕ ИЗДАНИЕ

СОЗДАЕМ НЕЙРОННУЮ СЕТЬ



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

ТАРИК РАШИД

СОЗДАЕМ НЕЙРОННУЮ СЕТЬ

MAKE YOUR OWN NEURAL NETWORK

by Tariq Rashid

СОЗДАЕМ НЕЙРОННУЮ СЕТЬ

Тарик Рашид

ББК 32.973.26-018.2.75
С58
УДК 681.3.07

Главный редактор *С.Н. Тригуб*
Зав. редакцией *В.Р. Гинзбург*
Перевод с английского и редакция канд. хим. наук *А.Г. Гузикевича*

Рашид, Тарик.
С58 Создаем нейронную сеть. : Пер. с англ. — СПб. : ООО "Альфа-книга", 2017. — 272 с. : ил. — Парал. тит. англ.
ISBN 978-5-9909445-7-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Тарик Рашид
Создаем нейронную сеть

Литературный редактор *Л.Н. Красножон*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9909445-7-2 (рус.)

ISBN 978-1530826605 (англ.)

Оглавление

Содержание

Об авторе

Тарик Рашид — специалист в области количественного анализа данных и разработки решений на базе продуктов с открытым исходным кодом. Имеет ученую степень по физике и степень магистра по специальности “Machine Learning and Data Mining”.

Тарик — большой поклонник Python, и ему нравится обучать новичков этому языку. Проживая в Лондоне, он возглавляет местную группу разработчиков Python (насчитывающую около 3000 участников), организует многочисленные семинары и часто выступает с докладами на международных конференциях.

Пролог

Попытки создания разумных машин

На протяжении тысячелетий человечество пытается разгадать тайну работы мозга и создать устройства, способные мыслить.

Мы изобрели кремниевые зажигалки, с помощью которых можем в любой момент получить огонь, блоки для поднятия тяжестей и даже калькуляторы, способные выполнять для нас расчеты, но все эти простые механические и электронные устройства, облегчающие нашу жизнь, нас уже не удовлетворяют.

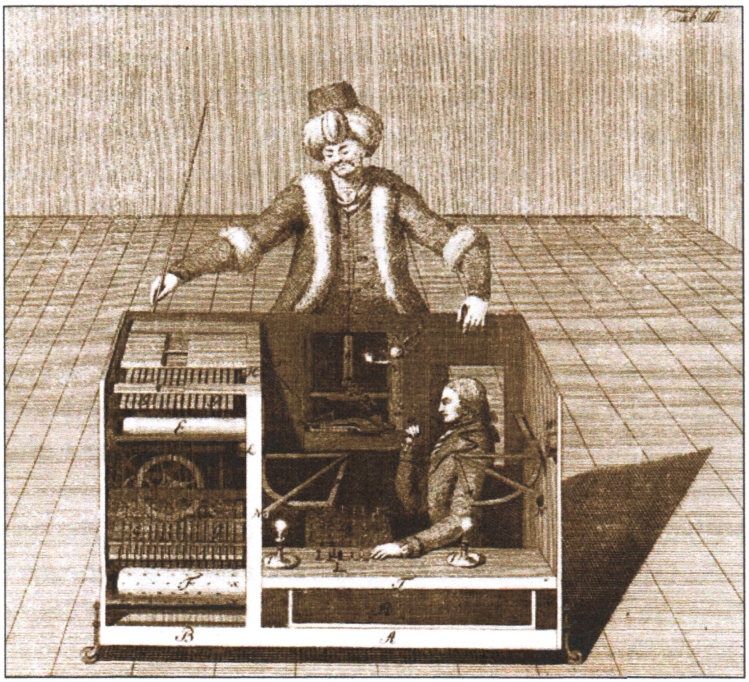
Теперь мы хотим автоматизировать более сложные задачи, такие как группирование схожих фотографий, отделение больных клеток от здоровых и даже игра в шахматы. По-видимому, для решения таких задач требуется человеческий интеллект или по крайней мере некие загадочные возможности человеческой психики, которых вы не найдете в простых устройствах типа калькуляторов.

Идея машин, обладающих интеллектом наподобие человеческого, в равной степени соблазнительная и пугающая, что породило в обществе множество фантазий и страхов на эту тему. Невероятно способный, но и чрезвычайно опасный HAL 9000 из фильма *Космическая одиссея 2001 года* Стэнли Кубрика, боевые роботы в захватывающей кинофраншизе *Терминатор* и говорящий автомобиль KITT с ярко выраженной индивидуальностью в классическом телесериале *Рыцарь дорог* — это лишь некоторые из огромного числа возможных примеров.

Когда в 1997 году чемпион мира по шахматам гроссмейстер Гарри Каспаров потерпел поражение от компьютера IBM Deep Blue, мы не только радовались этому историческому достижению, но и задумались о возможных опасностях со стороны разгулявшегося машинного интеллекта.

Интерес к разумным машинам был настолько велик, что некоторые изобретатели не смогли устоять перед искушением пойти на прямой обман публики, прибегая к всевозможным ухищрениям

и трюкам. Когда секрет шахматной машины *Турок* был раскрыт, оказалось, что внутри ящика скрывался игрок — человек, который и передвигал фигуры посредством системы рычагов.



Природа вдохновила новый золотой век

Оптимизм и амбиции по созданию искусственного интеллекта взмыли до новых высот после формализации этого предмета в 1950-х годах. Начальные успехи ознаменовались разработкой компьютеров, играющих в простые игры и доказывающих теоремы. Кое-кто был убежден, что машины с интеллектом на уровне человеческого появятся в течение ближайших десяти лет.

Однако искусственный интеллект оказался твердым орешком, и дальнейший прогресс застопорился. Осознание теоретических трудностей нанесло сокрушительный удар по амбициям создателей искусственного интеллекта, вслед за чем последовали урезание финансирования и потеря интереса к этой области исследований.

Казалось, что машины с их жесткой аппаратной логикой, состоящей сплошь из единиц и нулей, никогда не смогут соперничать с органической гибкостью, иногда размытой, мыслительных процессов биологического мозга.

По прошествии некоторого периода относительно слабого прогресса возникла невероятно мощная идея о том, как вывести исследования в области искусственного интеллекта из их привычной колеи. Почему бы не попытаться создать искусственный мозг, копируя работу биологического мозга? Реального мозга с нейронами вместо логических вентилей, наделенного способностью делать умозаключения, а не управляемого традиционными жестко закодированными черно-белыми абсолютистскими алгоритмами.

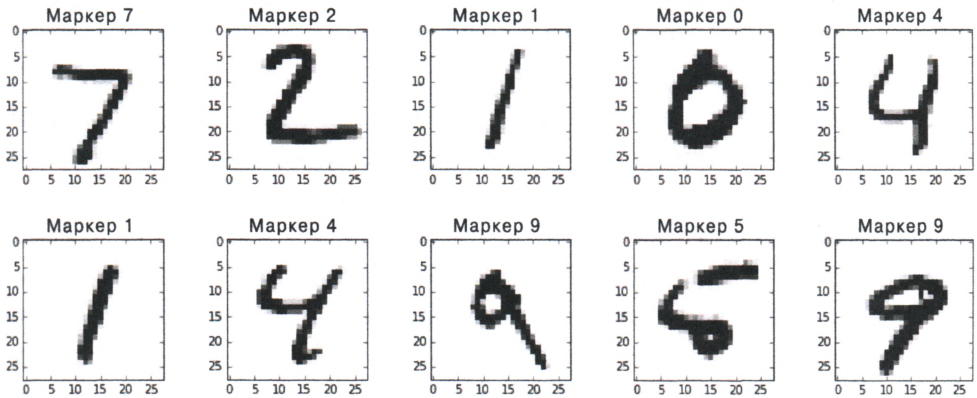
Ученых вдохновляла видимая простота мозга пчел или попугаев по сравнению со сложностью тех задач, которые они могли решать. Мозг весом не более долей грамма демонстрировал способность управлять полетом и адаптироваться к ветру, распознавать пищу и хищников и быстро принимать решения относительно того, стоит ли вступить в схватку или лучше обратиться в бегство. Исследователей интересовало, смогут ли компьютеры с их колоссальными электронными ресурсами имитировать работу такого мозга и даже достичь большего. Если мозг пчелы насчитывает примерно 950 тысяч нейронов, то смогут ли современные компьютеры с их ресурсами памяти, исчисляемыми гигабайтами и терабайтами, превзойти пчел?

Но при использовании традиционных подходов к решению проблем даже суперкомпьютеры с огромным объемом памяти и сверхбыстрыми процессорами не могли обеспечить то, на то способен мозг птицы или пчелы.

Идея проектирования интеллектуальных вычислительных устройств по образу и подобию биологических систем привела к созданию теории **нейронных сетей**, ставшей одним из самых мощных и полезных подходов к разработке искусственного интеллекта. Если говорить о сегодняшних достижениях, то, например, нейронные сети являются основным направлением деятельности компании Deepmind (ныне собственность компании Google), добившейся таких фантастических успехов, как создание нейронной сети, способной учиться играть в видеоигры, и еще одной, которая смогла победить в невероятно сложной игре го гроссмейстера мирового уровня. Нейронные

сети уже составляют саму сердцевину многих повседневных технологий, таких как системы автоматического распознавания автомобильных номеров или системы считывания почтовых индексов, написанных от руки.

В книге я расскажу вам о нейронных сетях, о том, как они работают и как создать собственную нейронную сеть, способную научиться распознавать рукописные символы (задача, решить которую в рамках традиционных компьютерных подходов очень трудно).



Введение

Для кого предназначена эта книга

Эта книга предназначена для всех, кто хочет разобраться в том, что такое нейронные сети. Она адресована тем, кто хочет создать и использовать собственную нейронную сеть, а также по достоинству оценить элегантную простоту математических идей, лежащих в основе работы нейронных сетей.

Это руководство не рассчитано на специалистов в области математики и вычислительной техники. От вас не требуется никаких специальных знаний или владения математикой в объеме, выходящем за пределы школьного курса.

Если вы умеете выполнять простые арифметические операции, то в состоянии создать собственную нейронную сеть. Самое сложное, что мы будем использовать, — градиенты, но и это понятие будет разъяснено так, чтобы у большинства читателей не возникло никаких трудностей по данному поводу.

Для любознательных читателей или студентов книга может послужить стартовой площадкой для дальнейшего путешествия в увлекательный мир искусственного интеллекта. Ухватив суть того, как работают нейронные сети, вы сможете применять базовые идеи для решения самых разнообразных задач.

Преподаватели могут использовать это руководство для того, чтобы доходчиво рассказать о нейронных сетях и способах их реализации студентам с целью заинтересовать их этой темой и вселить в них энтузиазм для создания собственной обучающейся системы искусственного интеллекта с помощью всего лишь нескольких строк программного кода. Приведенный в книге пример кода тестировался на Raspberry Pi — небольшом и недорогом компьютере, пользующемся большой популярностью среди школьников и студентов.

О таком руководстве я мечтал, когда, еще будучи подростком, мучительно пытался понять принципы работы этих загадочных нейронных сетей. О них упоминалось в книгах, журналах и кинофильмах,

но в то время их серьезное обсуждение можно было найти только в научных статьях, предназначенных для математиков и написанных их языком.

Как мне тогда хотелось, чтобы кто-то объяснил мне все это на простом и понятном даже студенту колледжа языке! Именно эту цель и преследую я в книге.

Что мы будем делать

В книге мы создадим нейронную сеть, способную распознавать рукописные цифры.

Мы начнем с очень простых прогнозирующих нейронов и будем постепенно усовершенствовать их по мере достижения предела их текущих возможностей. Время от времени мы будем делать короткие паузы для того, чтобы вы могли изучить очередной ряд математических понятий, которые будут нужны вам для понимания того, каким образом нейронные сети могут обучаться и прогнозировать решения задач.

В нашем путешествии мы охватим такие математические понятия, как функции, простые линейные классификаторы, итеративное улучшение, матричная алгебра, оптимизация методом градиентного спуска и даже геометрические операции вращения. Но все это будет объясняться на достаточно доходчивом уровне, совершенно не требующем никаких дополнительных знаний сверх того, что предлагается в школьном курсе математики.

Успешно создав нашу первую нейронную сеть, мы вооружимся накопленным опытом и исследуем другие возможные направления работы. Например, мы используем распознавание образов для того, чтобы усовершенствовать способность нашей машины к обучению, не прибегая к дополнительным тренировочным данным. Мы даже заглянем в память нейронной сети, чтобы посмотреть, не удастся ли нам найти там нечто сокровенное, причем существует не так-то много руководств, в которых показывается, как это можно сделать!

Кроме того, в ходе поэтапного создания нашей нейронной сети мы изучим Python — простой, полезный и весьма популярный язык программирования. Вновь подчеркну, что никакого предварительного опыта или знакомства с программированием от вас не потребуется.

Как мы будем это делать

Главной целью книги является разъяснение понятий, лежащих в основе нейронных сетей, как можно большему кругу читателей. Это означает, что нашей отправной точкой всегда будет какая-то более или менее понятная для вас идея, с которой вы уже знакомы. Далее, постепенно достраивая конструкцию на уже имеющемся надежном фундаменте, мы будем достигать уровня, достаточного для того, чтобы вы были в состоянии оценить некий новый интересный аспект нейронных сетей.

Чтобы не упускать из виду наиболее важные моменты, автор сознательно опускает обсуждение вопросов, понимание которых не является обязательным для создания собственной нейронной сети. Мы будем иметь дело с множеством любопытных контекстов и соприкасающихся с ними тем, которые могут входить в круг интересов некоторых читателей, и если таким читателем являетесь вы, то у вас есть полная свобода действий для собственных исследований в конкретных направлениях.

В книге вовсе не ставилась задача рассказать обо всех возможных способах оптимизации и улучшения нейронных сетей. Таких способов существует множество, и их анализ отвлек бы нас от основной цели — ознакомления читателей с основными идеями на простом и как можно более доходчивом уровне.

Материал книги разделен на три главы.

- В **главе 1** обсуждаются математические идеи, лежащие в основе работы простых нейронных сетей. Автор намеренно не затрагивает здесь ничего, что связано с программированием, чтобы сфокусировать внимание читателя на базовых положениях теории.
- В **главе 2** вы познакомитесь с языком программирования Python в объеме, достаточном для реализации нашей нейронной сети. Мы научим эту сеть распознавать рукописные цифры и протестируем, насколько эффективно она работает.
- В **главе 3** мы продвинемся немного дальше, чем необходимо для понимания простых нейронных сетей, и просто поработаем в свое удовольствие. Мы опробуем на практике идеи дальнейшего улучшения функционирования своей нейронной сети, а также

заглянем внутрь обученной сети, чтобы увидеть, можем ли мы понять, чему она уже обучилась и как она принимает решения.

Заранее предупреждаю, что беспокоиться по поводу покупки необходимых программных инструментов вам не следует, поскольку мы будем использовать лишь **бесплатное** программное обеспечение с **открытым исходным кодом**, так что вам ни за что не придется платить. Весь приведенный в книге код тестировался на очень дешевом компьютере Raspberry Pi Zero ($5). О том, как запустить этот компьютер, рассказывается в приложении.

Дополнительные замечания

Я буду считать, что не справился со своей задачей, если чтение книги не вызовет у вас чувства подлинного изумления и восхищения успехами математики и компьютерной науки.

Я буду считать, что не справился со своей задачей, если мне не удастся убедительно доказать вам, что даже знания школьного курса математики и простых компьютерных приемов вполне достаточно для реализации самых смелых идей, например для создания искусственного интеллекта, имитирующего способность человеческого мозга к обучению.

Я буду считать, что не справился со своей задачей, если мне не удастся вселить в вас уверенность в своих силах и желание заняться дальнейшими исследованиями в невероятно интересной области искусственного интеллекта.

Каждый из читателей может внести свой вклад в улучшение книги и сообщить автору о своих пожеланиях по электронной почте (`makeyourownneuralnetwork@gmail.com`) или в Твиттере (`@myoneuralnet`).

С обсуждением тем, затронутых в книге, можно ознакомиться по следующему адресу:

`http://makeyourownneuralnetwork.blogspot.co.uk/`

Там же будет публиковаться информация об обнаруженных опечатках, ошибках и исправлениях.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@dialektika.com
WWW: http://www.dialektika.com

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

ГЛАВА 1

Как работают нейронные сети

Черпайте вдохновенье в окружающих вас мелочах.

Что легко одному, трудно другому

Компьютеры, в сущности, — это не более чем калькуляторы, способные выполнять арифметические операции с огромной скоростью.

Эта особенность компьютеров позволяет им отлично справляться с задачами, аналогичными тем, которые решаются с помощью калькуляторов: суммирование чисел с целью определения объемов продаж, применение процентных ставок для начисления налогов или построение графиков на основе существующих данных.

Даже просмотр телевизионных программ или прослушивание потоковой музыки через Интернет с помощью компьютера не требует чего-то большего, чем многократное выполнение простых арифметических операций. Возможно, вы будете удивлены, но реконструкция видеокадра, состоящего сплошь из единиц и нулей, которые поступают на ваш компьютер по сети, осуществляется путем выполнения арифметических действий, лишь ненамного более сложных, чем суммирование чисел, которое вы проходили в школе.

Безусловно, сложение чисел с гигантской скоростью — тысячи или миллионы операций в секунду — это впечатляющий эффект, но его нельзя назвать проявлением искусственного интеллекта. Даже если человеку трудно складывать в уме большие числа, данный процесс вовсе не требует особого интеллекта. Для таких вычислений достаточно способности следовать элементарным инструкциям, и именно это происходит внутри любого компьютера.

А теперь перевернем все вверх тормашками и поставим столы на компьютеры!

Взгляните на приведенные ниже иллюстрации и убедитесь в том, что для вас не составляет труда распознать то, что на них изображено.

Мы с вами, посмотрев на эти фотографии, легко определим, что на них изображены соответственно люди, кот и дерево. Мы способны практически мгновенно и с высокой точностью распознавать объекты, на которые направляем свой взгляд, и при этом очень редко ошибаемся.

В процессе анализа изображений и классификации объектов наш мозг обрабатывает огромные объемы информации. Компьютеру трудно решать подобные задачи, а точнее — невероятно трудно.

Задача	Компьютер	Человек
Быстрое умножение тысяч больших чисел	Легко	Трудно
Распознавание конкретного человека среди толпы на фотографии	Трудно	Легко

Мы догадываемся, что для распознавания образов требуется человеческий интеллект — то, чего недостает машинам, какими бы сложными и мощными мы их ни создавали, а все потому, что они — не люди.

Но это как раз тот тип задач, в отношении которых мы и хотели бы сделать компьютеры более эффективными, поскольку они работают быстрее и никогда не устают. В свою очередь, именно для решения подобных задач и ведутся работы по созданию искусственного интеллекта.

Конечно же, компьютеры всегда будут начиняться электроникой, и потому задачей искусственного интеллекта является поиск предписаний, или **алгоритмов**, основанных на новых подходах к решению трудных задач, о которых идет речь.

Резюме

- Одни задачи, как, например, перемножение миллионов пар чисел, просты для компьютера, но трудны для человека.
- Но есть и такие задачи, как, к примеру, распознавание лиц людей в толпе на фотографии, которые трудны для компьютера, но просты для человека.

Простая прогнозирующая машина

Начнем с простого и будем постепенно усложнять задачу.

Вообразите простую прогнозирующую машину (предиктор), которая получает вопрос, совершает некий "мыслительный" процесс и выдает ответ. Все происходит примерно так, как в приведенном выше примере с распознаванием образов, в котором входная информация воспринималась нашими глазами, далее наш мозг анализировал изображение, после чего мы делали выводы относительно того, какие объекты имеются на данном изображении. Это можно представить с помощью следующей схемы.

Но компьютеры не могут по-настоящему думать (вспомните, что они всего лишь усовершенствованные калькуляторы), поэтому мы используем другую терминологию, более точно соответствующую тому, что происходит на самом деле.

Компьютер получает входную информацию, выполняет некоторые расчеты и выдает результат. Этот процесс схематически представлен на следующей иллюстрации. Входная информация, заданная в виде “3×4”, обрабатывается с возможной заменой операции умножения более простыми операциями сложения, и выдается выходной результат “12”.

Возможно, вы подумали: “Ну и что здесь особенного?” Не переживайте, все нормально. Пока что мы используем простые и хорошо знакомые примеры для введения понятий, которые далее будут применены к более интересным нейронным сетям.

Давайте чуть усложним задачу.

Представьте, что машина должна преобразовывать километры в мили.

Предположим, что формула, преобразующая километры в мили, нам неизвестна. Все, что мы знаем, — это то, что данные единицы измерения связаны между собой **линейной** зависимостью. Это означает, что если мы удвоим количество миль, то количество километров, соответствующее данному расстоянию, также удвоится. Такая зависимость воспринимается нами интуитивно.

Существование линейного соотношения между километрами и милями дает нам ключ к разгадке формулы для вычислений. Она должна иметь следующий вид: мили = километры × **c**, где **c** — константа, величину которой мы пока что не знаем.

Единственными дополнительными подсказками нам могут служить отдельные примеры правильного выражения расстояний в километрах и милях. Эти примеры будут выступать как бы в роли экспериментальных данных, отражающих истинное положение вещей, которые мы используем для проверки своей научной теории.

Истинный пример	Километры	Мили
1	0	0
2	100	62,37

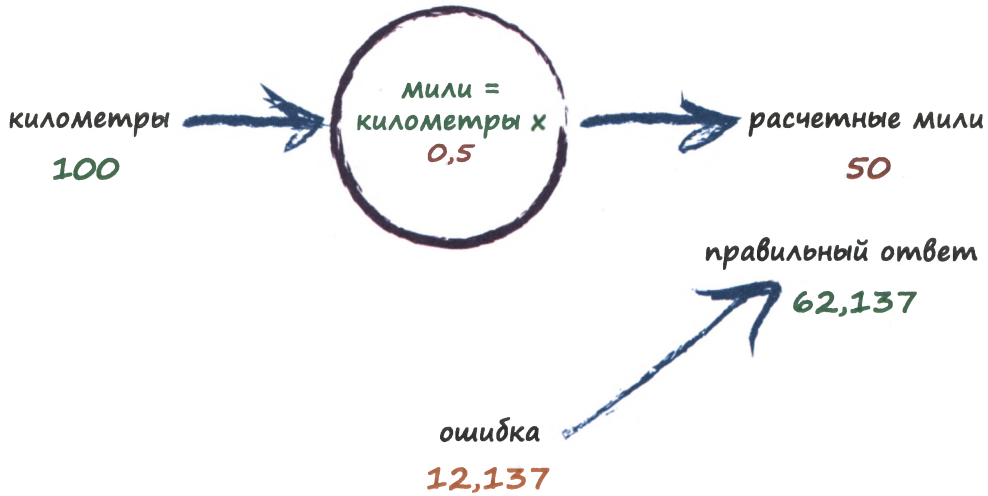
Что нам нужно сделать для того, чтобы определить недостающую величину константы? Давайте просто подставим в формулу какое-либо **случайное** значение! Например, предположим, что **c=0,5**, и посмотрим, что при этом произойдет.

Здесь мы подставляем в формулу мили = километры × **c** значение 100 вместо *километры* и текущее пробное значение 0,5 вместо константы **c**. В результате мы получаем ответ: *50 миль*.

Ну хорошо. Это вовсе неплохо, если учесть, что значение **c=0,5** было выбрано случайным образом! Но мы знаем, что оно не совсем точное, поскольку пример 2 истинного соотношения говорит нам о том, что правильный ответ — 62,137.

Мы ошиблись на 12,137. Это число представляет величину **ошибки**, т.е. разность между истинным значением из нашего списка примеров и расчетным значением.

```
ошибка = истина - расчет
       = 62,137 - 50
       = 12,137
```



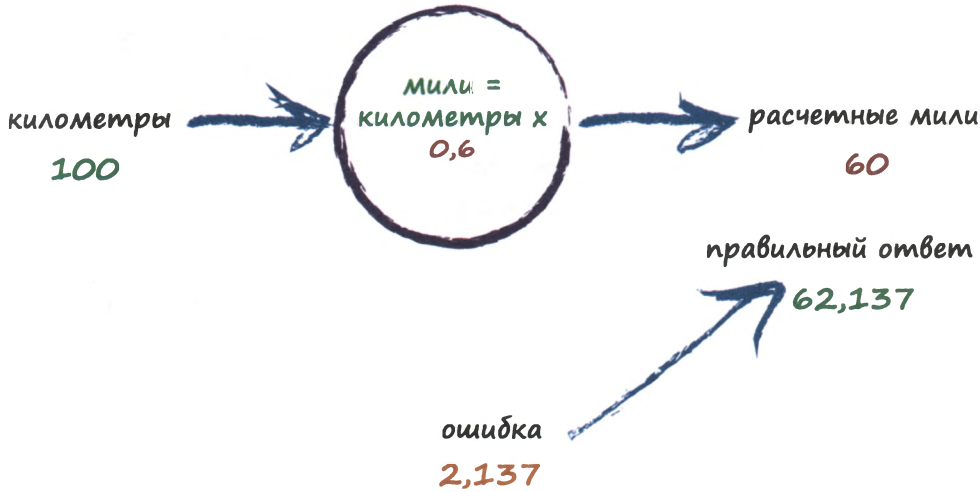
Что дальше? Мы знаем, что ошиблись, и нам известна величина ошибки. Вместо того чтобы видеть в этой ошибке повод для отчаяния, мы используем эту информацию для того, чтобы предложить более удачное пробное значение константы **c**, чем первое.

Вернемся к ошибке. Мы ошиблись на 12,137 в сторону меньших значений. Так как формула для преобразования километров в мили линейная, мили = километры × **c**, мы знаем, что увеличение **c** приведет к увеличению результирующего значения.

Давайте немного подправим **c**, заменив значение 0,5 значением 0,6, и посмотрим, к чему это приведет.

Приняв для **c** значение 0,6, мы получаем `мили = километры × c = = 100 × 0,6 = 60`. Это уже лучше, чем предыдущий ответ — 50. Налицо явный прогресс!

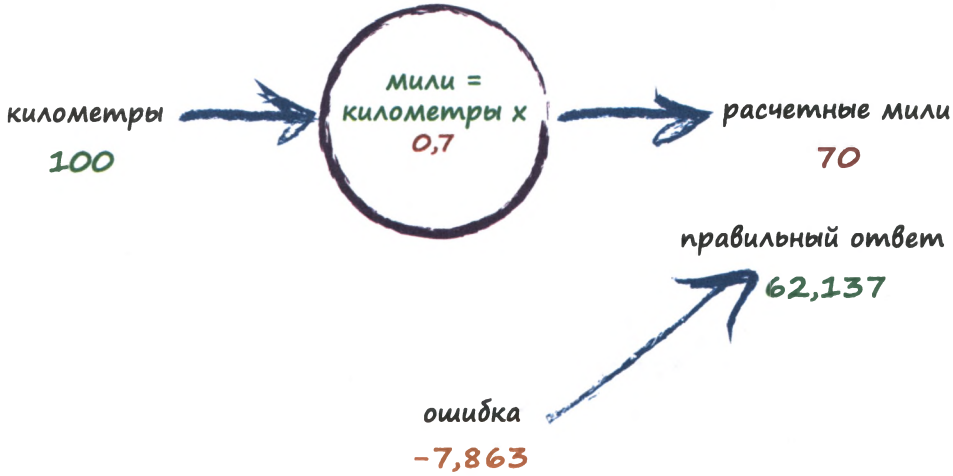
Теперь ошибка уменьшилась до 2,137. Вполне возможно, что с такой ошибкой мы могли бы даже смириться.



Здесь важно то, что величина ошибки подсказала нам, в каком направлении следует откорректировать величину **c**. Мы хотели увеличить выходной результат 50, поэтому немного увеличили **c**.

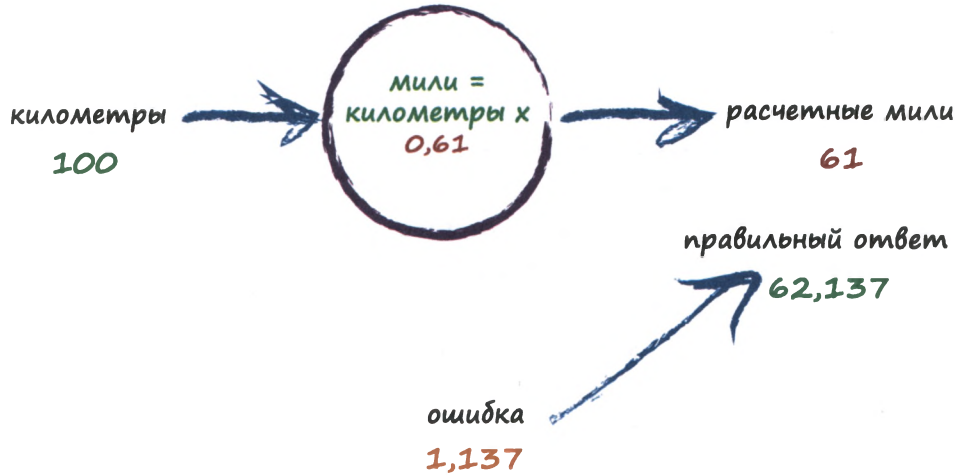
Вместо того чтобы использовать алгебру для нахождения точной величины поправки, на которую следует изменить значение **c**, мы продолжим использовать подход, заключающийся в постепенном уточнении значения этой константы. Если вы не уверены в правильности такого подхода и считаете, что было бы намного проще сразу определить точный ответ, то имейте в виду, что существует множество более интересных задач, для которых не существует простых математических формул, связывающих между собой входные и выходные значения. Именно поэтому нам и нужны более сложные методы наподобие нейронных сетей.

Повторим уже знакомые нам действия. Выходной результат 60 все еще слишком мал. Давайте вновь немного изменим константу **c**, увеличив ее значение с 0,6 до 0,7.



О, нет! Мы перестарались и получили результат, **превышающий** правильный ответ. Предыдущая ошибка была равна 2,137, а теперь она составляет –7,683. Знак "минус" просто свидетельствует о том, что вместо недооценки истинного результата произошла его переоценка (напомню, что величина ошибки определяется выражением *правильное значение минус расчетное значение*).

Итак, `c=0,6` было гораздо лучше, чем `c=0,7`. Сейчас мы могли бы признать величину ошибки при `c=0,6` удовлетворительной и закончить это упражнение. Но мы все-таки продвинемся еще чуть дальше. Почему бы нам не попытаться ввести очень малую поправку и увеличить значение `c` с 0,6 до, скажем, 0,61?



Это дает нам гораздо лучший результат, чем предыдущие, поскольку теперь выходное значение 61 отличается от правильного значения 62,137 всего лишь на 1,137.

Итак, последняя попытка научила нас тому, что величину поправки к величине **c** необходимо каждый раз определять заново. Если выходной результат приближается к правильному ответу, т.е. если ошибка уменьшается, то не следует оставлять величину поправки прежней. Тем самым мы избегаем переоценки значения по сравнению с истинным, как это было ранее.

Опять-таки, не отвлекаясь на поиск точных способов определения величины **c** и по-прежнему фокусируя внимание на идее ее постепенного уточнения, мы можем предположить, что поправка должна выражаться некоторой долей ошибки. Это интуитивно понятно: большая ошибка указывает на необходимость введения большей поправки, тогда как малая ошибка нуждается в незначительной поправке.

Хотите — верьте, хотите — нет, но то, что мы сейчас сделали, передает суть процесса обучения нейронной сети. Мы тренировали машину так, чтобы ее предсказания становились все более и более точными.

Нам стоит сделать небольшую паузу, чтобы поразмышлять над следующим: мы не находили точного решения задачи в один прием, как это часто делается при решении школьных или научных задач. Вместо этого мы предприняли совершенно иной подход, заключающийся в многократных попытках проверки пробного значения и его уточнения. Такие процессы иногда называют **итеративными**, что как раз и означает постепенное, шаг за шагом, улучшение искомого результата.

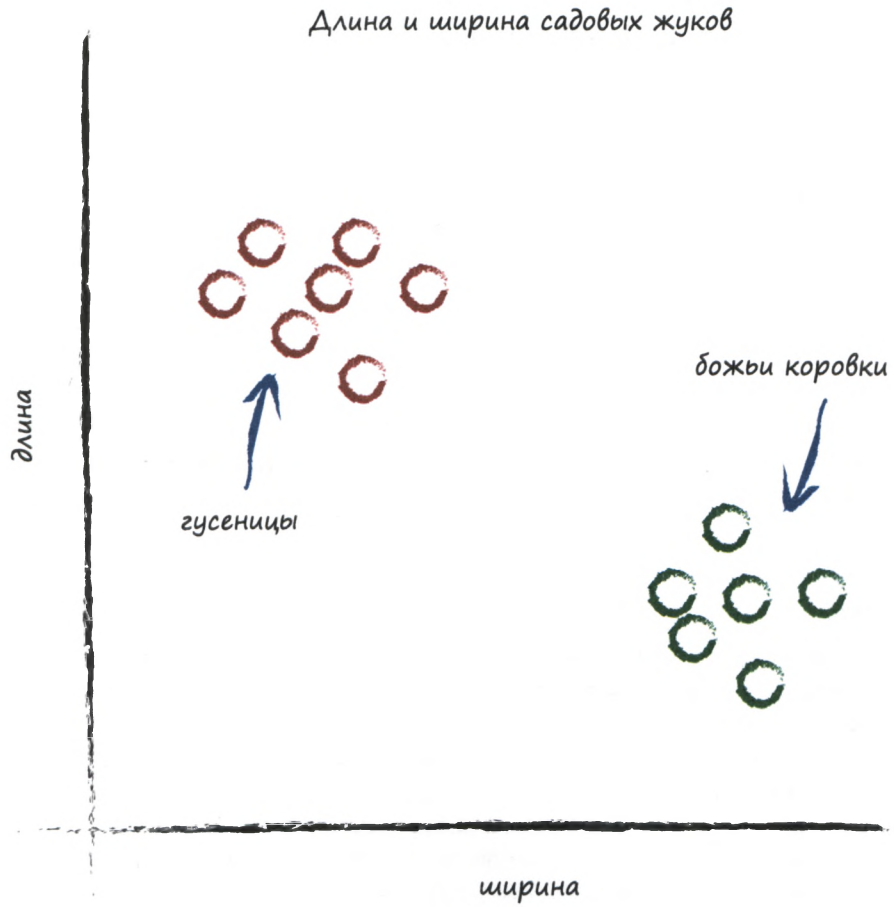
Резюме

- У всех полезных компьютерных систем имеются каналы ввода и вывода, между которыми над данными выполняются некоторые вычисления. В случае нейронных сетей это не так.
- Если точные принципы функционирования какой-либо системы нам неизвестны, то мы пытаемся получить представление о том, как она работает, используя модель с регулируемыми параметрами. Если бы мы не знали, как преобразовать километры в мили, то могли бы использовать для этой цели линейную функцию в качестве модели с регулируемым наклоном.
- Неплохим способом улучшения подобных моделей является настройка параметров на основании сравнения результатов модели с точными результатами в известных примерах.

Задачи классификации и прогнозирования очень близки

Мы назвали описанную ранее простую машину **прогнозирующей**, поскольку она получает входные данные и делает определенный прогноз относительно того, какими должны быть выходные данные. Мы улучшали прогнозы, регулируя внутренний параметр на основании величины ошибки, которую определяли, сравнивая прогноз с известным точным значением.

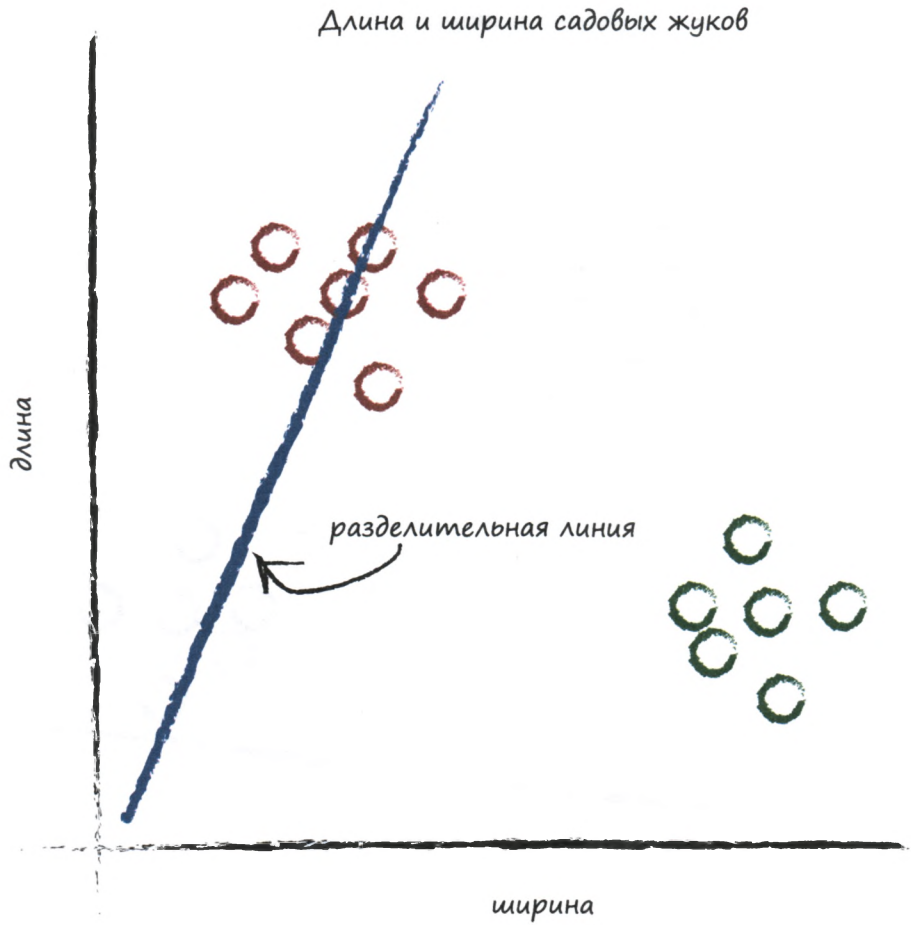
Взгляните на иллюстрацию ниже, на которой приведена диаграмма, представляющая результаты измерения размеров садовых жуков.



На диаграмме отчетливо видны две группы данных. Гусеницы *у*же и длиннее, а божьи коровки шире и короче.

Помните наш предиктор, который пытался правильно вычислить количество миль, соответствующее заданному количеству километров? В основу этого предиктора была положена настраиваемая линейная функция. Надеюсь, вы не забыли, что график зависимости выходных значений линейной функции от ее входных значений представляет собой прямую линию. Изменение настраиваемого параметра **c** приводит к изменению крутизны наклона этой прямой линии.

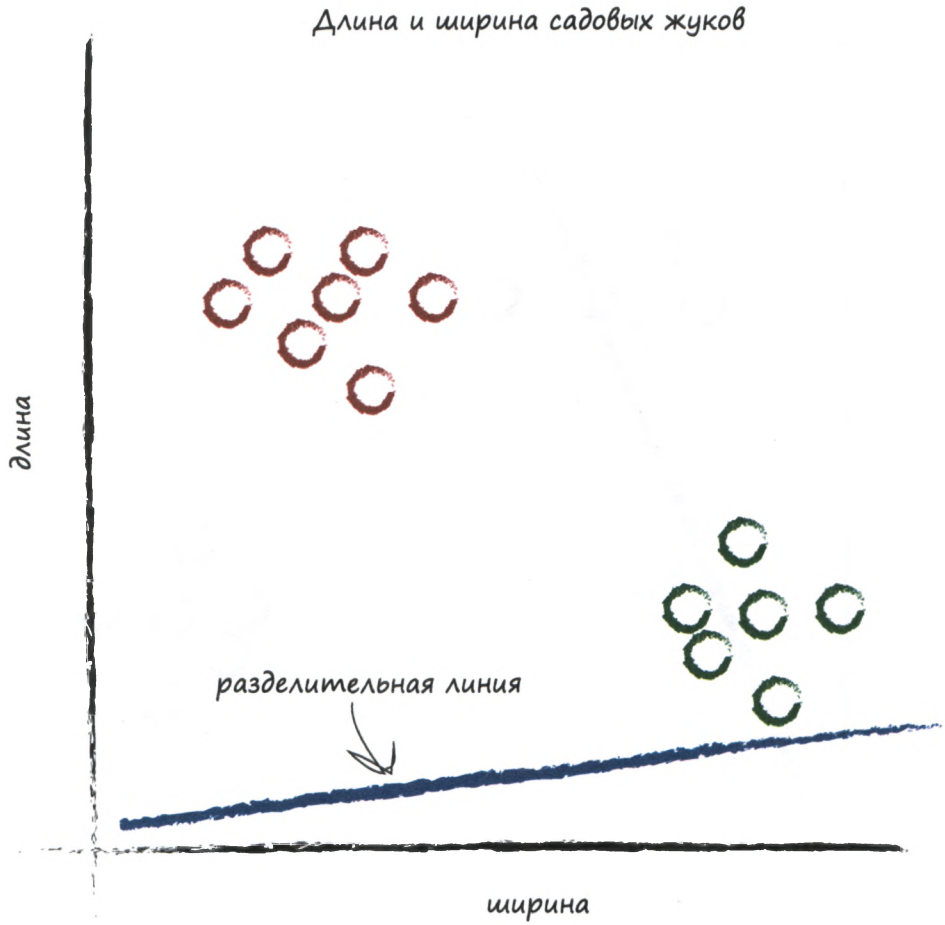
Что получится, если мы наложим на этот график прямую линию?



Мы не можем использовать прямую линию точно так же, как раньше, когда преобразовывали одно число (километры) в другое (мили), но, возможно, в данном случае нам удастся отделить с ее помощью один тип данных от другого.

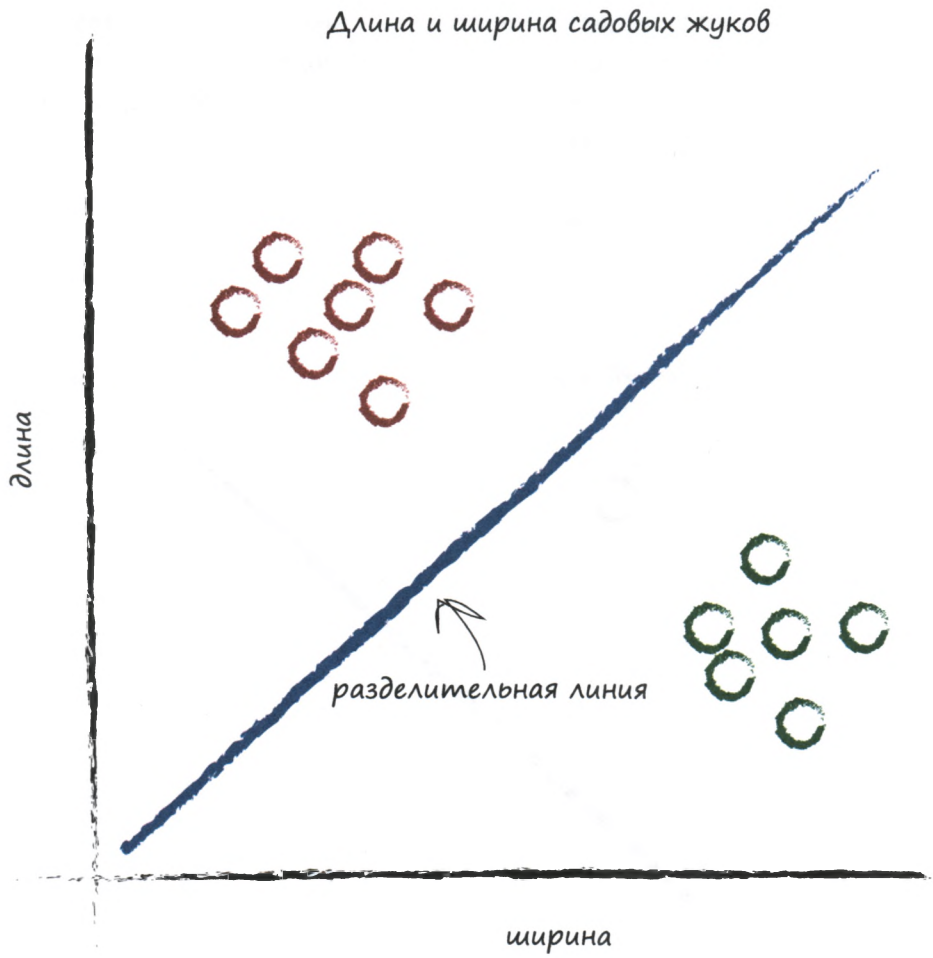
Если бы на показанном выше графике прямая линия отделяла гусениц от божьих коровок, то мы могли бы воспользоваться ею для **классификации** неизвестных жуков, исходя из результатов измерений. Однако имеющаяся линия не справляется с этой задачей, поскольку половина гусениц находится по ту же сторону линии, что и божьи коровки.

Давайте проведем другую линию, изменив наклон, и посмотрим, что при этом произойдет.



На этот раз линия оказалась еще менее полезной, поскольку вообще не отделяет один вид жуков от другого.

Сделаем еще один заход.

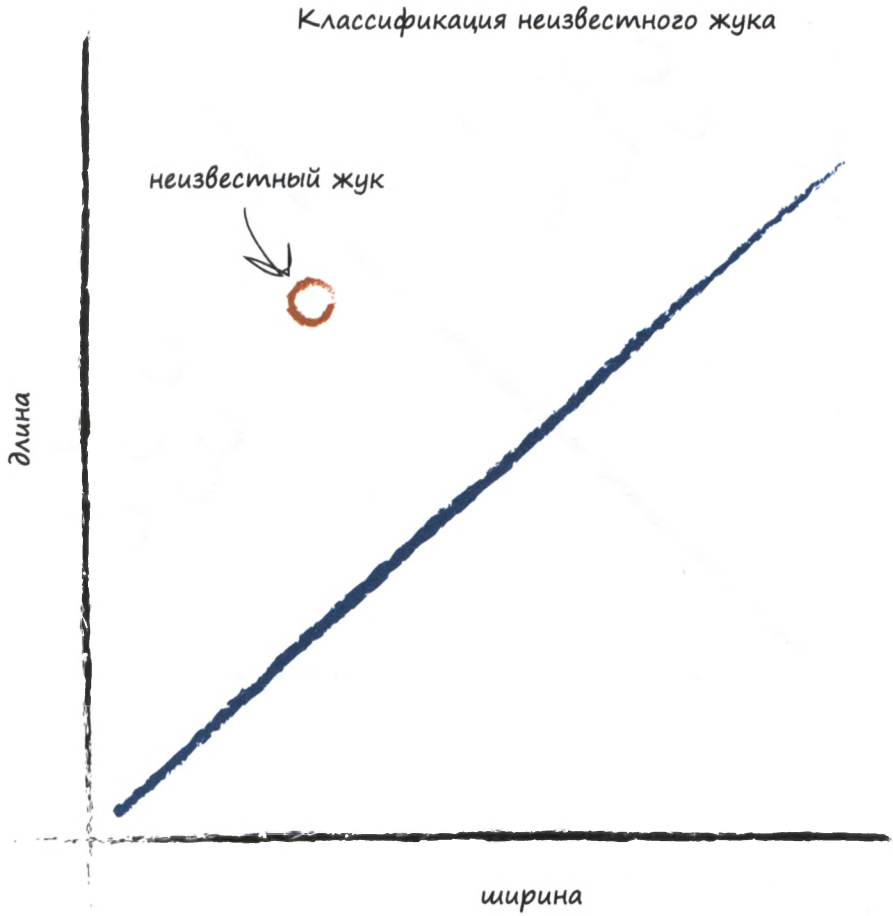


Вот это другое дело! Линия отчетливо отделяет гусениц от божьих коровок. Теперь мы можем использовать ее в качестве **классификатора** жуков.

Мы предполагаем, что не существует никаких других видов жуков, кроме тех, которые показаны на диаграмме, но на данном этапе это не является недостатком подхода — мы просто пытаемся проиллюстрировать суть идеи простого классификатора.

А теперь представьте, что в следующий раз наш компьютер использует робота для того, чтобы тот отобрал очередного жука и выполнил необходимые замеры. Тогда полученную линию можно будет использовать для корректного отнесения жука к семейству гусениц или семейству божьих коровок.

Взглянув на следующий график, вы увидите, что неизвестный жук относится к семейству гусениц, поскольку попадает в область над линией. Несмотря на свою простоту эта классификация уже является довольно мощным инструментом!



Только что вы имели возможность убедиться в том, насколько полезными могут быть предикторы с линейной функцией в качестве инструмента классификации вновь поступающих данных.

Однако мы обошли вниманием один существенный момент. Как нам узнать, какой наклон прямой является подходящим? Как улучшить линию, если она не разделяет должным образом две разновидности жуков?

Ответ на этот вопрос также имеет самое непосредственное отношение к способности нейронной сети обучаться, к рассмотрению чего мы и переходим.

Тренировка простого классификатора

Сейчас мы займемся **тренировкой** (обучением) нашего линейного классификатора и научим его правильно классифицировать жуков, относя их к гусеницам или божьим коровкам. Как вы видели ранее, речь идет об уточнении наклона разграничительной линии, отделяющей на графике одну группу точек данных, соответствующих парам значений длины и ширины, от другой.

Как мы это сделаем?

Вместо того чтобы заранее разработать подходящую научную теорию, мы нащупаем правильный путь методом проб и ошибок. Это позволит нам лучше понять математику, скрытую за этими действиями.

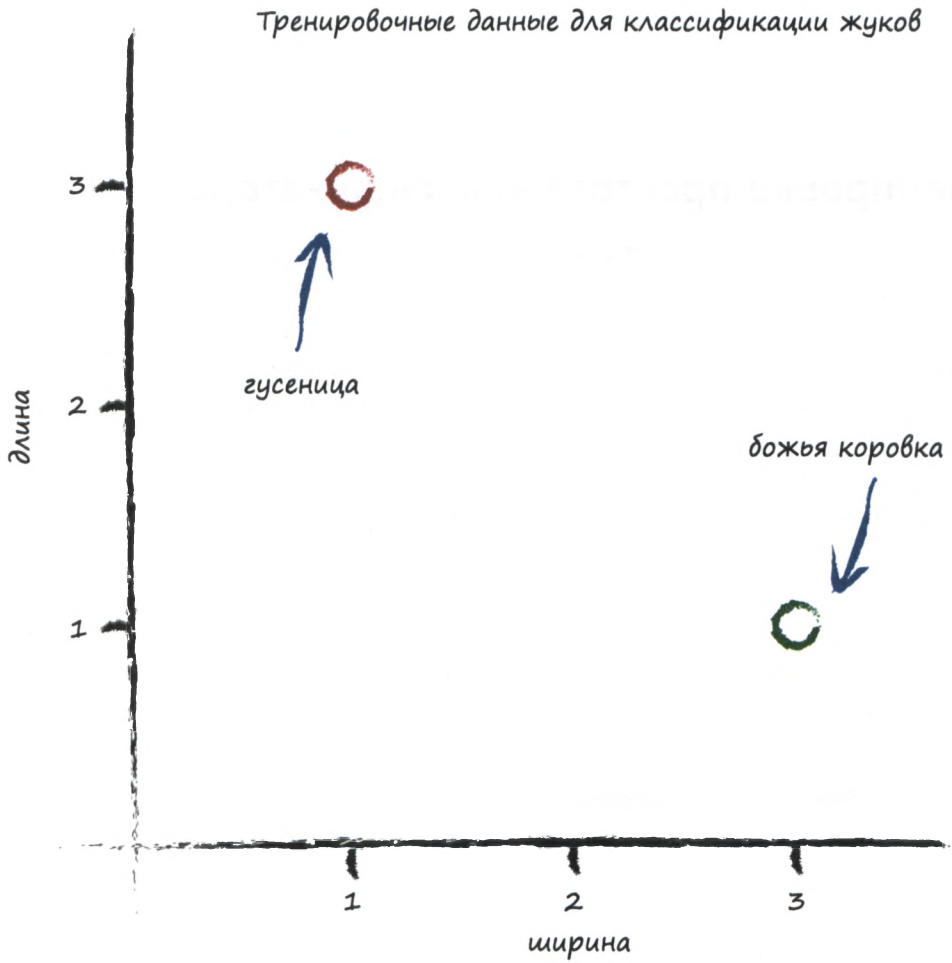
Нам нужны примеры для тренировки классификатора. Чтобы не усложнять себе жизнь, мы ограничимся двумя простыми примерами, приведенными ниже.

Пример	Ширина	Длина	Жук
1	3,0	1,0	Божья коровка
2	1,0	3,0	Гусеница

У нас есть пример жука, имеющего ширину 3,0 и длину 1,0, который, как нам известно, является божьей коровкой. Второй пример относится к жуку, имеющему бо́льшую длину — 3,0 и меньшую ширину — 1,0, которым является гусеница.

Мы знаем, что данные в этом наборе примеров являются истинными. Именно с их помощью будет уточняться значение константы в функции классификатора. Примеры с истинными значениями, которые используются для обучения предиктора или классификатора, называют **тренировочными данными**.

Отобразим эти два примера тренировочных данных на диаграмме. Визуализация данных часто помогает лучше понять их природу, почувствовать их, чего нелегко добиться, просто вглядываясь в список или таблицу, заполненную числами.



Начнем со случайной разделительной линии, потому что нужно ведь с чего-то начинать. Вспомните линейную функцию из примера с преобразованием километров в мили, параметр которой мы настраивали. Мы можем сделать то же самое и сейчас, поскольку разделительная линия в данном случае прямая:

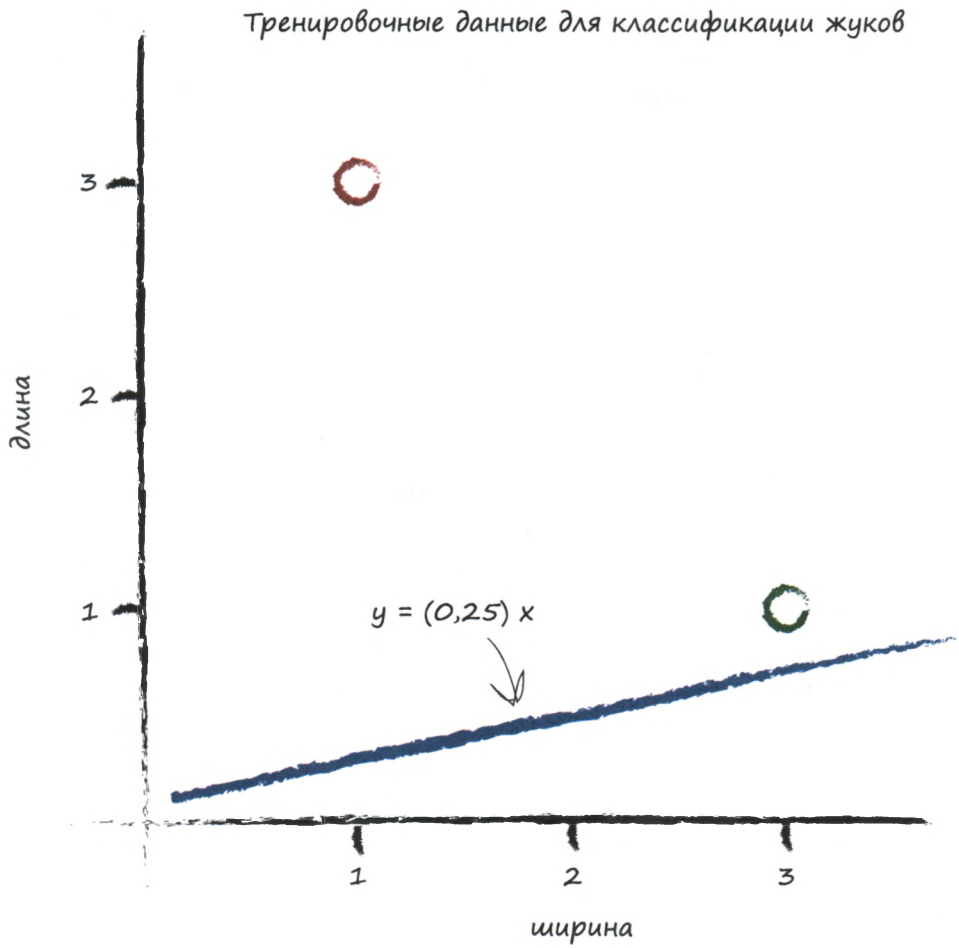
$$y = Ax$$

Мы намеренно используем здесь имена x и y, а не длина и ширина, поскольку, строго говоря, в данном случае линия не служит предиктором. Она не преобразует ширину в длину, как ранее километры переводились в мили. Вместо этого она выступает в качестве разделительной линии, классификатора.

Вероятно, вы обратили внимание на неполную форму уравнения **y=Ax**, поскольку полное уравнение прямой линии имеет следующий вид: **y=Ax+B**. Мы намеренно делаем этот сценарий с садовыми жуками максимально простым. Ненулевое значение **B** просто соответствует линии, которая не проходит через начало координат на диаграмме, что не добавляет в наш сценарий ничего нового.

Ранее было показано, что параметр **A** управляет наклоном линии. Чем больше **A**, тем больше крутизна наклона.

Для начала примем, что **A=0,25**. Тогда разделительная линия описывается уравнением **y=0,25x**. Отобразим эту линию в графическом виде на той же диаграмме, на которой отложены тренировочные данные.



Благодаря графику мы безо всяких вычислений сразу же видим, что линия **y=0,25x** не является хорошим классификатором. Она не отделяет один тип жуков от другого. Например, мы не можем делать утверждения наподобие "Если точка данных располагается над линией, то она соответствует гусенице", поскольку точно там же располагаются и данные божьей коровки.

Интуитивно мы понимаем, что правый край линии следует немного приподнять. Мы устоим перед соблазном сделать это, глядя на диаграмму и проводя подходящую линию вручную. Мы хотим проверить, не удастся ли нам подобрать для этого подходящий повторяемый рецепт, последовательность команд, которую компьютерщики называют **алгоритмом**.

Обратимся к первому тренировочному примеру, соответствующему божьей коровке: ширина — 3,0 и длина — 1,0. Если бы мы тестировали функцию **y=Ax** с этим примером, в котором **x** равен 3,0, то получили бы следующий результат:

```
y = (0,25) * (3,0) = 0,75
```

Функция, в которой для параметра **A** установлено начальное случайно выбранное значение, равное 0,25, сообщает, что для жука шириной 3,0 длина должна быть равна 0,75. Мы знаем, что это слишком мало, поскольку согласно тренировочным данным длина жука равна 1,0.

Итак, налицо расхождение, или **ошибка**. Точно так же, как в примере с предиктором, преобразующим километры в мили, мы можем использовать величину этой ошибки для получения информации о том, каким образом следует корректировать параметр **A**.

Однако сначала давайте подумаем, каким должно быть значение **y**. Если положить его равным 1,0, то линия пройдет через точку с координатами $(x,y) = (3{,}0;\ 1{,}0)$, соответствующую божьей коровке. Само по себе это неплохо, но это не совсем то, что нам нужно. Нам желательно, чтобы линия проходила над этой точкой. Почему? Да потому, что мы хотим, чтобы точки данных божьей коровки лежали под линией, а не на ней. Линия должна служить разделителем между точками данных божьих коровок и гусениц, а не предсказывать длину жука по известной ширине.

В связи с этим попробуем нацелиться на значение **y=1,1** при **x=3,0**. Оно лишь ненамного больше 1,0. Вместо него можно было бы взять значение 1,2 или 1,3, но никак не 10 или 100, поскольку с большой долей вероятности это привет к тому, что прямая будет проходить над всеми точками данных, как божьих коровок, так и гусениц, в результате чего она станет полностью бесполезной в качестве разделителя.

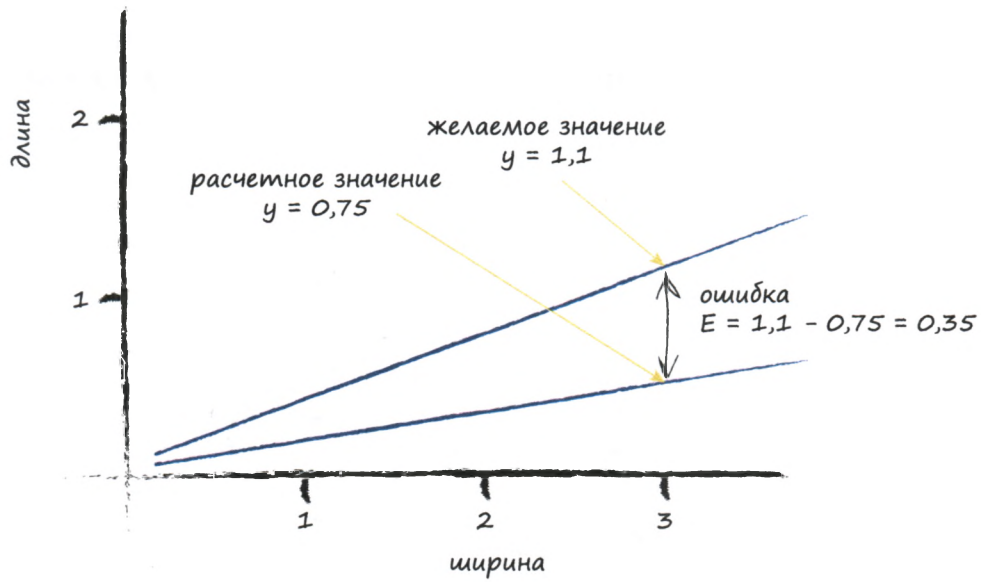
Следовательно, мы выбираем целевое значение 1,1 и определяем ошибку **E** с помощью следующей формулы:

ошибка = желаемое целевое значение - фактический результат

Или (после подстановки значений):

$$E = 1,1 - 0,75 = 0,35$$

Помня о пользе визуализации информации, обратимся к приведенной ниже диаграмме, на которой в графическом виде представлены ошибка, а также целевое и расчетное значения.



Вы спросите: а каким образом наше знание величины ошибки **E** может помочь в нахождении лучшего значения для параметра **A**? Это очень важный вопрос.

Давайте на время отступим от этой задачи и немного порассуждаем. Мы хотим использовать ошибку в значении **y**, которую назвали **E**, для нахождения искомого изменения параметра **A**. Для этого нам нужно знать, как эти две величины связаны между собой. Каково соотношение между **A** и **E**? Если бы это было нам известно, то мы могли бы понять, как изменение одной величины влияет на другую.

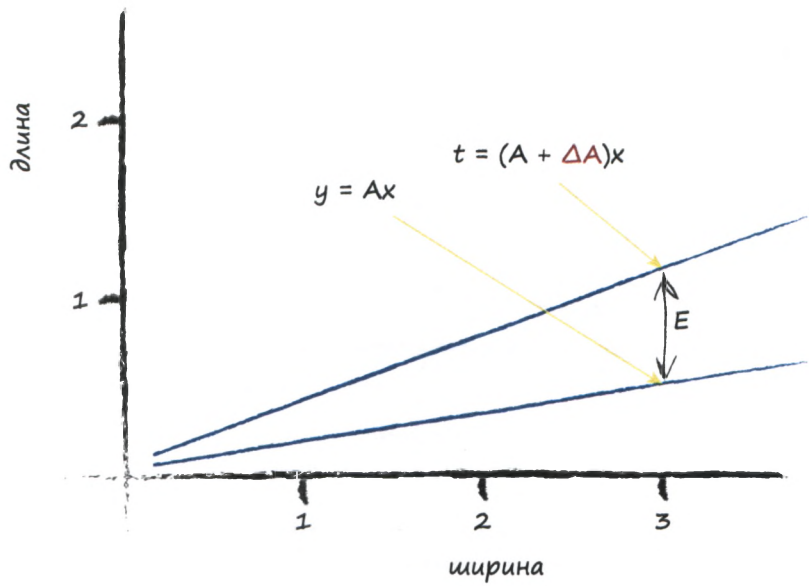
Начнем с линейной функции для классификатора:

$$y = Ax$$

Нам уже известно, что начальные попытки присвоения пробных значений параметру **A** привели к неверным значениям **y**, если ориентироваться на тренировочные данные. Пусть **t** — корректное целевое значение. Чтобы получить его, мы должны ввести в **A** небольшую поправку. Для таких поправок в математике принято использовать символ Δ, означающий "небольшое изменение". Запишем соответствующее уравнение:

$$t = (A + \Delta A)\,x$$

Отобразим это соотношение в графическом виде на диаграмме, на которой показаны линии для двух значений наклона: **A** и **A + ΔA**.



Вспомните, что ошибку **E** мы определили как разность между желаемым корректным значением **y** и расчетным значением, полученным для текущего пробного значения **A**. Таким образом, **E = t - y**.

Запишем это в явном виде:

$$t - y = (A + \Delta A)x - Ax$$

Раскрыв скобки и приведя подобные члены, получаем:

$$E = t - y = Ax + (\Delta A)x - Ax$$
$$E = (\Delta A)x$$

Это просто замечательно! Ошибка **E** связана с **ΔA** очень простым соотношением. Оно настолько простое, что поначалу я даже засомневался, не кроется ли где-то ошибка, но оно оказалось действительно верным. Как бы то ни было, это простое соотношение значительно упрощает нашу работу.

Делая подобного рода выкладки, можно легко забыть о первоначальной задаче. Сформулируем простыми словами то, чего мы хотели добиться.

Мы хотели узнать, каким образом можно использовать информацию об ошибке **E** для определения величины поправки к **A**, которая изменила бы наклон линии таким образом, чтобы классификатор лучше справлялся со своими функциями. Преобразуем последнее уравнение, чтобы найти выражение для **ΔA**:

$$\Delta A = E / x$$

Есть! Это и есть то волшебное выражение, которое мы искали. Теперь мы можем использовать ошибку **E** для изменения наклона классифицирующей линии на величину **ΔA** в нужную сторону.

Примемся за дело — обновим начальный наклон линии.

Когда **x** был равен 3,0, ошибка была равна 0,35. Таким образом, **ΔA = E/x** превращается в 0,35 / 3,0 = 0,1167. Это означает, что текущее значение **A=0,25** необходимо изменить на величину 0,1167. Отсюда следует, что новое, улучшенное значение **A** равно **(A + ΔA)**, т.е. 0,25 + 0,1167 = 0,3667. Не составляет труда убедиться в том, что

расчетное значение **y** при новом значении **A** равно, как и следовало ожидать, 1,1 — желаемому целевому значению.

Ух ты! У нас получилось! Все работает, и мы располагаем методом для улучшения параметра **A**, если известна текущая ошибка.

Давайте поднажмем.

Закончив с первым примером, потренируемся на втором. Он дает нам следующие истинные данные: **x=1,0** и **y=3,0**.

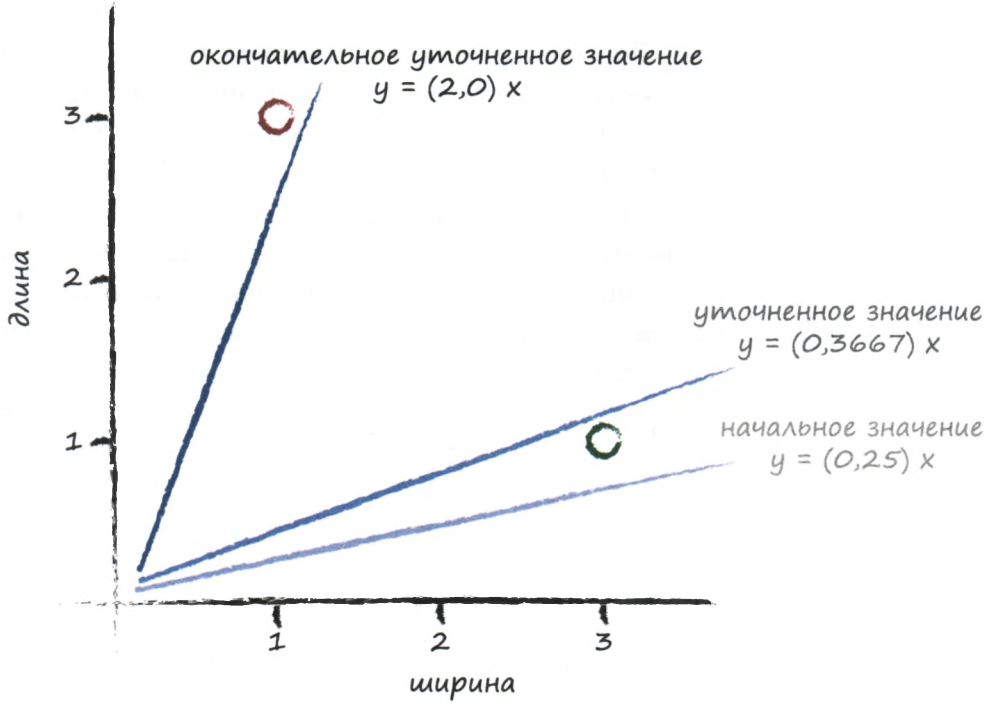
Посмотрим, что получится, если вставить **x=1,0** в линейную функцию, в которой теперь используется обновленное значение **A=0,3667**. Мы получаем **y** = 0,3667 * 1,0 = 0,3667. Это очень далеко от значения **y=3,0** в тренировочном примере.

Используя те же рассуждения, что и перед этим, когда мы нащупывали путь к построению такой линии, которая не пересекала бы тренировочные данные, а проходила над ними или под ними, мы можем задать желаемое целевое значение равным 2,9. При этом данные тренировочного примера, соответствующего гусеницам, находятся над линией, а не на ней. Ошибка **E** равна (2,9 – 0,3667) = 2,5333.

Эта ошибка больше предыдущей, но если хорошо подумать, то у нас ведь был всего лишь один пример для обучения линейной функции, который отчетливо смещал функцию в своем направлении.

Опять обновим **A**, как делали до этого. Соотношение **ΔA = E/x** дает 2,5333 / 1,0 = 2,5333. Это означает, что после очередного обновления параметр **A** принимает значение 0,3667 + 2,5333 = 2,9. Отсюда следует, что для **x=1,0** функция возвращает в качестве ответа значение 2,9, которое и является желаемым целевым значением.

Мы проделали довольно большую работу, поэтому можем снова передохнуть и визуализировать полученные результаты. На следующей диаграмме представлены начальная линия, линия, обновленная после обучения на первом тренировочном примере, и окончательная линия, обновленная на втором тренировочном примере.



Но погодите! Что произошло? Глядя на график, мы видим, что нам не удалось добиться того наклона прямой, которого мы хотели. Она не обеспечивает достаточно надежное разделение областей диаграммы, занимаемых точками данных божьих коровок и гусениц.

Ну что тут сказать? Мы получили то, что просили. Линия обновляется, подстраиваясь под то целевое значение **y**, которое мы задаем.

Что-то здесь не так? А ведь действительно, если мы будем продолжать так и далее, т.е. просто обновлять наклон для очередного примера тренировочных данных, то все, что мы будем каждый раз получать в конечном счете, — это линию, проходящую вблизи точки данных последнего тренировочного примера. В результате этого мы отбрасываем весь предыдущий опыт обучения, который могли бы использовать, и учимся лишь на самом последнем примере.

Как исправить эту ситуацию?

Легко! И эта идея играет ключевую роль в **машинном обучении**. Мы **сглаживаем** обновления, т.е. немного уменьшаем величину поправок. Вместо того чтобы каждый раз с энтузиазмом заменять **A**

новым значением, мы используем лишь некоторую долю поправки **ΔA**, а не всю ее целиком. Благодаря этому мы движемся в том направлении, которое подсказывает тренировочный пример, но делаем это осторожно, сохраняя некоторую часть предыдущего значения, которое было получено в результате, возможно, многих предыдущих тренировочных циклов. Мы уже видели, как работает эта идея сглаживания в примере с преобразованием километров в мили, когда изменяли параметр **c** лишь на некоторую долю фактической ошибки.

У такого сглаживания есть еще один очень мощный и полезный побочный эффект. Если тренировочные данные не являются надежными и могут содержать ошибки или шум (а в реальных измерениях обычно присутствуют оба этих фактора), то сглаживание уменьшает их влияние.

Ну что ж, сделаем перерасчет, на этот раз добавив сглаживание в формулу обновления:

$$\Delta A = L\ (E / x)$$

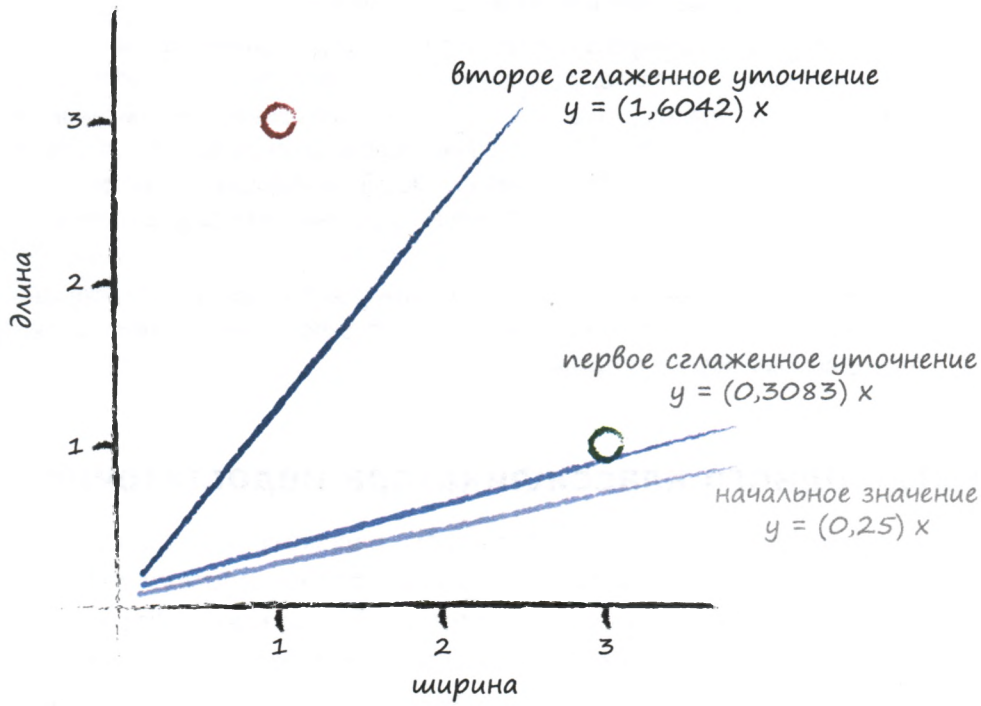
Фактор сглаживания, обозначенный здесь как **L**, часто называют коэффициентом **скорости обучения**. Выберем **L=0,5** в качестве разумного начального приближения. Это означает, что мы собираемся использовать поправку вдвое меньшей величины, чем без сглаживания.

Повторим все расчеты, используя начальное значение **A=0,25**. Первый тренировочный пример дает нам **y** = 0,25 * 3,0 = 0,75. При целевом значении 1,1 ошибка равна 0,35. Поправка равна **ΔA** = **L (E / x)** = 0,5 * 0,35 / 3,0 = 0,0583. Обновленное значение **A** равно 0,25 + + 0,0583 = 0,3083.

Проведение расчетов с этим новым значением **A** для тренировочного примера при **x=3,0** дает **y** = 0,3083 * 3,0 = 0,9250. Как видим, расположение этой линии относительно тренировочных данных оказалось неудачным — она проходит ниже значения 1,1, но этот результат не так уж и плох, если учесть, что это была всего лишь первая попытка. Главное то, что мы движемся в правильном направлении от первоначальной линии.

Перейдем ко второму набору тренировочных данных при **x=1,0**. Используя **A=0,3083**, мы получаем **y** = 0,3083 * 1,0 = 0,3083. Желаемым значением было 2,9, поэтому ошибка составляет (2,9 – 0,3083) = = 2,5917. Поправка **ΔA** = **L (E / x)** = 0,5 * 2,5917 / 1,0 = 1,2958. Теперь обновленное значение **A** равно 0,3083 + 1,2958 = 1,6042.

Вновь отобразим на диаграмме начальный, улучшенный и окончательный варианты линии, чтобы убедиться в том, что сглаживание обновлений приводит к более удовлетворительному расположению разделительной линии между областями данных божьих коровок и гусениц.



Это действительно отличный результат!

Всего лишь с двумя простыми тренировочными примерами и относительно простым методом обновления мы, используя сглаживание **скорости обучения**, смогли очень быстро получить хорошую разделительную линию **y=Ax**, где **A=1,6042**.

Не будем преуменьшать свои достижения. Нам удалось создать автоматизированный метод обучения классификации на примерах, который, несмотря на простоту подхода, продемонстрировал замечательную эффективность.

Великолепно!

Резюме

- Чтобы понять соотношение между выходной ошибкой линейного классификатора и параметром регулируемого наклона, достаточно владеть элементарной математикой. Зная это соотношение, можно определить величину изменения наклона, необходимую для устранения выходной ошибки.
- Недостаток прямолинейного подхода к регулировке параметров заключается в том, что модель обновляется до наилучшего соответствия только последнему тренировочному примеру, тогда как все предыдущие примеры не принимаются во внимание. Одним из неплохих способов устранения этого недостатка является уменьшение величины обновлений с помощью коэффициента скорости обучения, чтобы ни один отдельно взятый тренировочный пример не доминировал в процессе обучения.
- Тренировочные примеры, взятые из реальной практики, могут быть искажены шумом или содержать ошибки. Сглаживание обновлений способствует ограничению влияния подобных ложных примеров.

Иногда одного классификатора недостаточно

Как вы имели возможность убедиться, рассмотренные нами простые предикторы и классификаторы, относящиеся к числу тех, которые получают некоторые входные данные, выполняют соответствующие вычисления и выдают ответ, довольно эффективны. И все же их возможностей недостаточно для решения более интересных задач, к которым мы намереваемся применять методы нейронных сетей.

Ограничения линейного классификатора продемонстрированы ниже на простом, но весьма показательном примере. Но почему мы должны заниматься этим вместо того, чтобы сразу же перейти к обсуждению нейронных сетей? Дело в том, что от понимания сути указанных ограничений зависит один из ключевых элементов проектирования нейронных сетей, так что этот вопрос стоит того, чтобы его обсудить.

Мы отойдем от темы садовых жучков и обратимся к логическим (**булевым**) функциям. Не беспокойтесь, если эта терминология вам ни о чем не говорит. Джордж Буль — математик и философ, с именем которого связаны такие простые функции, как логические И и ИЛИ.

Функции булевой логики — это своего рода "мыслительные" функции со своим языком. Если мы говорим "Ты получишь пудинг только в том случае, если уже съел овощи И все равно голоден", то мы используем булеву функцию И. Результат булевой функции И истинен только тогда, когда истинны (выполняются) оба условия. Он не будет истинным, если истинно только одно из условий. Поэтому, если я голоден, но еще не съел овощи, то не получу свой пудинг.

Аналогично, если мы говорим "Ты можешь погулять в парке, если сегодня выходной ИЛИ у тебя отпуск", то используем булеву функцию ИЛИ. Результат булевой функции ИЛИ истинен, если истинным является хотя бы одно из условий. Вовсе не обязательно, чтобы все условия были истинными, как в случае функции И. Поэтому, если сегодня не выходной день, но у меня отпуск, то я могу погулять в парке.

Ранее мы представляли функцию в виде машины, которая принимает некоторые входные данные, выполняет определенные действия и выдает один ответ.

Значение **истина** часто представляется в компьютерах как число 1, а значение **ложь** — как число 0. В приведенной ниже таблице результаты работы логических функций И и ИЛИ представлены с использованием этой лаконичной нотации для всех комбинаций входных значений А и В.

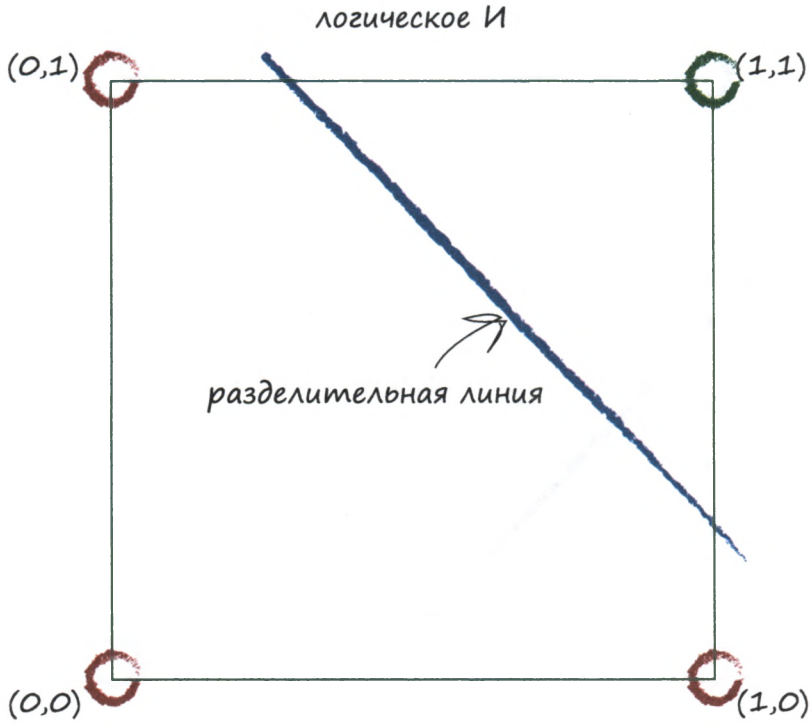
Входное значение А	Входное значение В	Логическое И	Логическое ИЛИ
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Здесь отчетливо видно, что результат функции И будет истинным только тогда, когда истинны и А, и В.

Булевы функции играют очень важную роль в информатике, и первые электронные вычислительные устройства фактически собирались из крохотных электрических схем, выполняющих логические операции. Даже арифметические операции выполнялись с использованием этих схем, которые сами по себе всего лишь выполняли простые булевы операции.

Представьте себе простой линейный классификатор, который должен использовать тренировочный набор данных для выяснения того, управляются ли данные логической функцией. Задачи такого рода естественным образом возникают перед учеными, исследующими причинно-следственные связи или корреляцию наблюдаемых величин. Например, требуется найти ответ на следующий вопрос: "Повышается ли вероятность заболевания малярией в тех местностях, в которых постоянно идут дожди И температура превышает 35 градусов?" Или такой: "Повышается ли вероятность заболевания малярией, если выполняется любое (булева функция ИЛИ) из этих условий?"

Взгляните на приведенную ниже диаграмму, где значения на двух входах логической функции, А и В, представляют координаты точек на графике. Вы видите, что только в том случае, когда оба входных значения истинны, т.е. каждое из них равно 1, выходной результат, выделенный зеленым цветом, является истинным. Ложные выходные значения обозначены красным цветом.

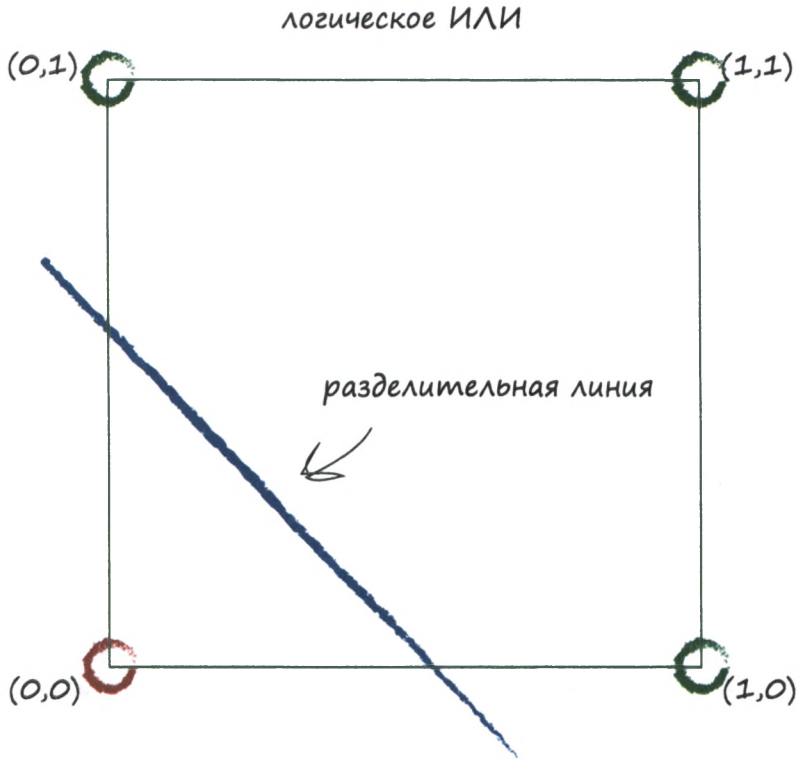


На диаграмме также показана прямая линия, отделяющая красную область от зеленой. Эта линия представляет линейную функцию, которую можно использовать для обучения линейного классификатора, что мы делали ранее.

Мы не будем проводить соответствующие вычисления, как в предыдущих примерах, поскольку ничего принципиально нового это не даст.

В действительности можно было бы предложить много других вариантов проведения разделительной линии, которые работали бы столь же удовлетворительно, но главный вывод из этого примера заключается в том, что простой линейный классификатор вида **y=ax+b** можно обучить работе с булевой функцией И.

А теперь взгляните на аналогичное графическое представление булевой функции ИЛИ.

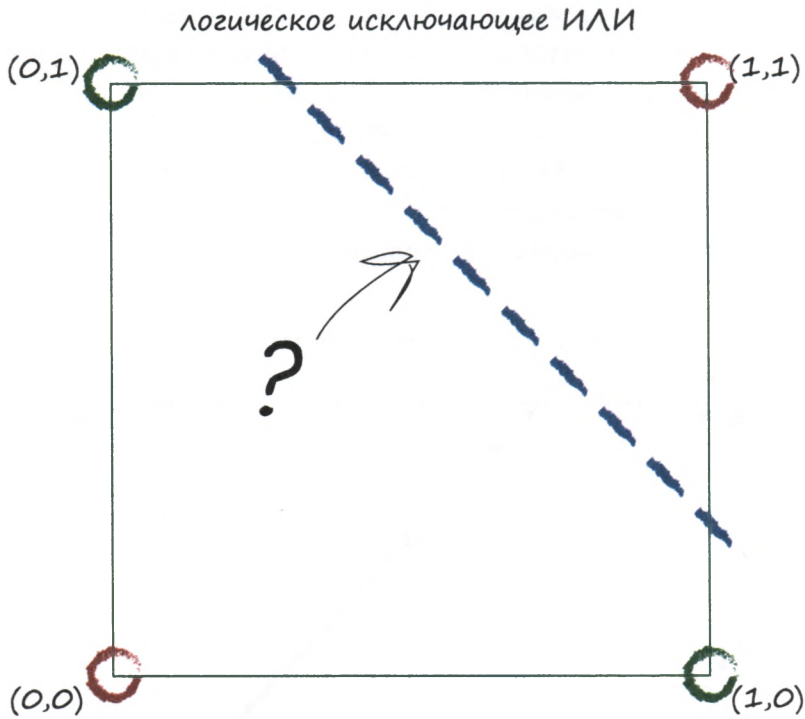


На этот раз красной оказалась лишь точка *(0,0)*, поскольку ей соответствуют ложные значения на обоих входах, A и B. Во всех других комбинациях значений хотя бы одно из них является истинным, и поэтому для них результат является истинным. Вся прелесть этой диаграммы заключается в том, что она наглядно демонстрирует возможность обучения линейного классификатора работе с функцией ИЛИ.

Существует еще одна булева функция, исключающее ИЛИ, которая дает истинный результат только в том случае, если лишь одно из значений на входах A и B истинно, но не оба. Таким образом, если оба входных значения ложны или оба истинны, то результат будет ложным. Все вышесказанное резюмировано в приведенной ниже таблице.

Входное значение A	Входное значение B	Исключающее ИЛИ
0	0	0
0	1	1
1	0	1
1	1	0

Взгляните на диаграмму, соответствующую этой функции.



Вот вам и проблема! Мы не видим способа разделить зеленую и красную области одной прямой линией.

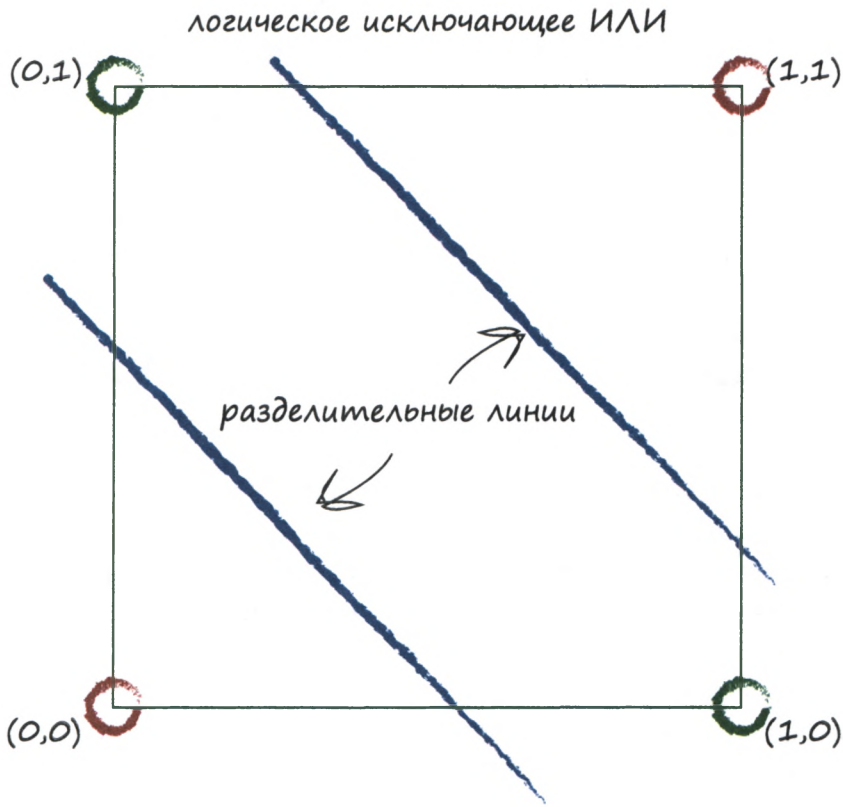
В действительности так оно и есть: невозможно провести одну прямую линию таким образом, чтобы она успешно отделила красные точки данных от зеленых для функции исключающего ИЛИ. Это означает, что простой линейный классификатор не в состоянии обучиться работе с функцией исключающего ИЛИ, если предоставить ему тренировочные данные, управляемые этой функцией.

Только что мы продемонстрировали главное ограничение простого линейного классификатора: такие классификаторы оказываются непригодными, если базовая задача не допускает разделения данных одной прямой линией.

Но ведь мы хотим, чтобы нейронные сети можно было использовать для решения самого широкого круга задач, даже тех, которые не допускают линейного разделения данных.

Следовательно, нам необходимо найти какой-то выход из этой ситуации.

К счастью, такой выход существует: совместное использование нескольких классификаторов. Его иллюстрирует приведенная ниже диаграмма с двумя разделительными линиями. Эта идея занимает центральное место в теории нейронных сетей. Теперь вам должно быть ясно, что с помощью множества прямых линий можно разделить желаемым образом любую сколь угодно сложную конфигурацию областей, подлежащих классификации.



Прежде чем мы займемся созданием нейронных сетей, которые предполагают существование нескольких классификаторов, работающих совместно, снова обратимся к природе и рассмотрим работу мозга животных, аналогии с которым послужили толчком к разработке подходов на основе нейронных сетей.

Резюме

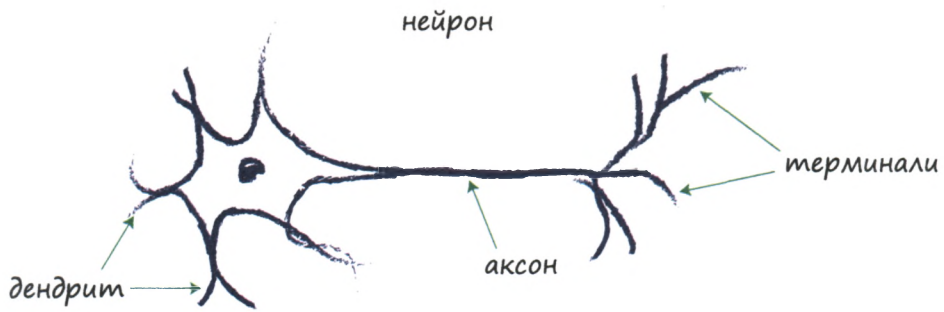
- Простой линейный классификатор не в состоянии разделить нужным образом области данных, если данные не управляются простым линейным процессом. Это было продемонстрировано на примере данных, управляемых логической функцией исключающего ИЛИ.
- Однако эта проблема решается очень просто: для разграничения данных, которые не удается разделить одной прямой линей, следует использовать множество линейных классификаторов.

Нейроны — вычислительные машины, созданные природой

Ранее говорилось о том, что мозг животных ставил ученых в тупик, поскольку даже у столь малых представителей живой природы, как попугаи, он демонстрирует несравненно бо́льшие способности, чем цифровые компьютеры с огромным количеством электронных вычислительных элементов и памятью невероятных объемов, работающих на частотах, недостижимых для живого мозга из плоти и крови.

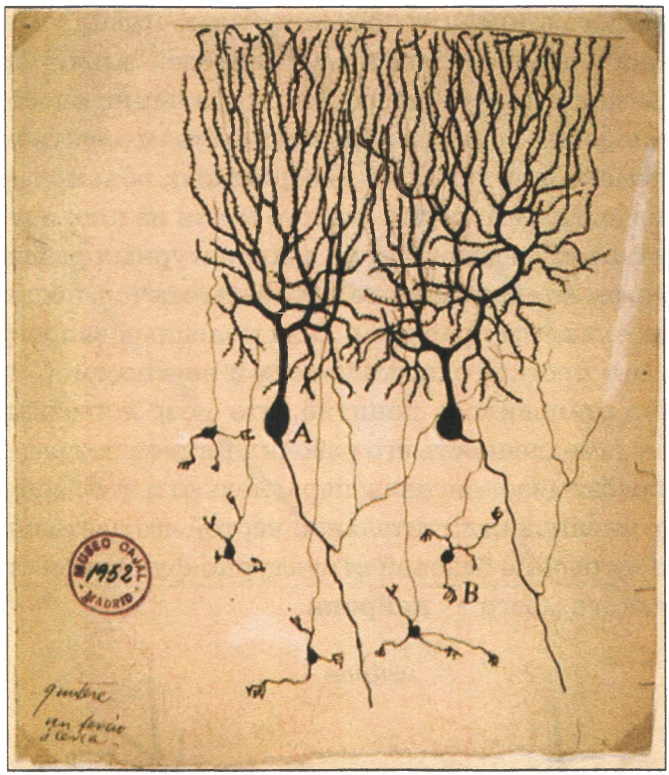
Тогда внимание сосредоточили на архитектурных различиях. В традиционных компьютерах данные обрабатываются последовательно, по четко установленным правилам. В их холодных запрограммированных расчетах нет места неоднозначности и неясности. С другой стороны, постепенно становилось понятно, что мозг животных, несмотря на кажущуюся замедленность его рабочих ритмов по сравнению с компьютерами, обрабатывает сигналы параллельно и что неопределенность является существенной отличительной чертой его деятельности.

Рассмотрим строение базовой структурно-функциональной единицы биологического мозга — **нейрона**.



Несмотря на то что нейроны существуют в различных формах, все они передают электрические сигналы от одного конца нейрона к другому — от дендритов через аксоны до терминалей. Далее эти сигналы передаются от одного нейрона к другому. Именно благодаря такому механизму вы способны воспринимать свет, звук, прикосновение, тепло и т.п. Сигналы от специализированных рецепторных нейронов доставляются по вашей нервной системе до мозга, который в основном также состоит из нейронов.

На приведенной ниже иллюстрации показана схема строения нейронов мозга попугая, представленная одним испанским нейробиологом в 1899 году. На ней отчетливо видны важнейшие компоненты нейрона — дендриты и терминали.



Сколько нейронов нам нужно для выполнения интересных, более сложных задач?

В головном мозге человека, являющемся самым развитым, насчитывается около 100 миллиардов нейронов. Мозг дрозофилы (плодовой мушки) содержит примерно 100 тысяч нейронов, но она способна летать, питаться, избегать опасности, находить пищу и решать множество других довольно сложных задач. Это количество — 100 тысяч — вполне сопоставимо с возможностями современных компьютеров, и поэтому в попытке имитации работы такого мозга есть смысл. Мозг нематоды (круглого червя) насчитывает всего 302 нейрона — ничтожно малая величина по сравнению с ресурсами современных цифровых компьютеров! Но этот червь способен решать такие довольно сложные задачи, которые традиционным компьютерам намного больших размеров пока что не по силам.

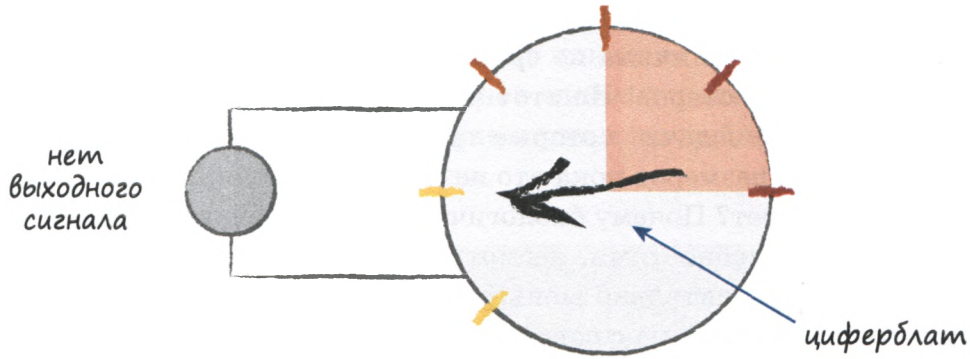
В чем же секрет? Почему биологический мозг обладает столь замечательными способностями, несмотря на то что работает медленнее и состоит из относительно меньшего количества вычислительных элементов по сравнению с современными компьютерами? Сложные механизмы функционирования мозга (например, наличие сознания) все еще остаются загадкой, но мы знаем о нейронах достаточно много для того, чтобы можно было предложить различные способы выполнения вычислений, т.е. различные способы решения задач.

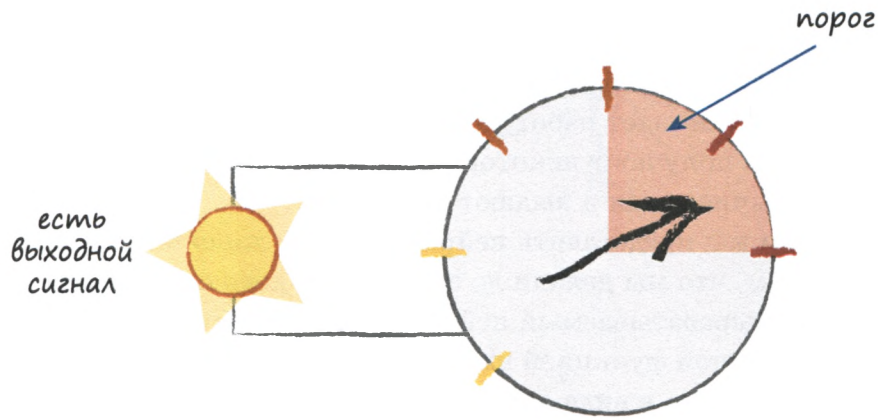
Рассмотрим, как работает нейрон. Он принимает поступающий электрический сигнал и вырабатывает другой электрический сигнал. Это очень напоминает работу моделей классификатора или предиктора, которые получают некоторые входные данные, выполняют определенные вычисления и выдают результат.

Так можем ли мы представить нейроны в виде линейных функций по аналогии с тем, что мы делали до этого? Нет, хотя сама по себе эта идея неплохая. Вырабатываемый нейроном выходной сигнал не является простой линейной функцией входного сигнала, т.е. выходной сигнал нельзя представить в виде *выход* = (*константа* * *вход*) + (*возможная другая константа*).

Согласно результатам наблюдений нейроны не реагируют немедленно, а подавляют входной сигнал до тех пор, пока он не возрастет до такой величины, которая запустит генерацию выходного сигнала. Это можно представить себе как наличие некоего порогового значения, которое должно быть превышено, прежде чем будет сгенерирован выходной сигнал. Сравните поведение воды в чашке: вода не

сможет выплеснуться, пока чашка не наполнится. Интуитивно это понятно — нейроны пропускают лишь сильные сигналы, несущие полезную информацию, но не слабый шум. Идея вырабатывания выходного сигнала лишь в случаях, когда амплитуда входного сигнала достигает достаточной величины, превышающей некоторый **порог**, проиллюстрирована ниже в графическом виде.

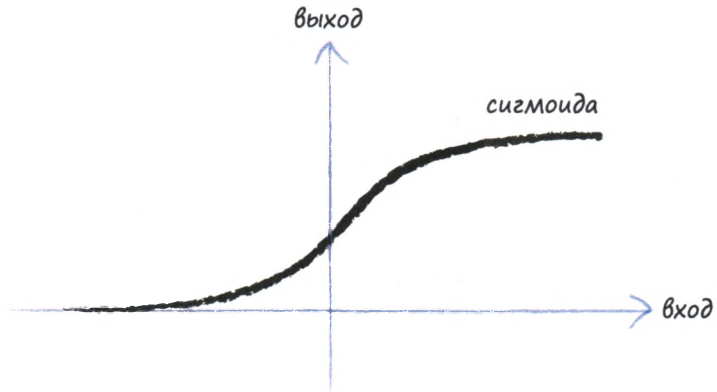




Функция, которая получает входной сигнал, но генерирует выходной сигнал с учетом порогового значения, называется **функцией активации**. С математической точки зрения существует множество таких функций, которые могли бы обеспечить подобный эффект. В качестве примера можно привести **ступенчатую функцию**.

Нетрудно заметить, что для слабых входных значений выходное значение равно нулю. Но стоит превысить входной порог, как на выходе появляется сигнал. Искусственный нейрон с таким поведением напоминал бы настоящий биологический нейрон. Это очень хорошо описывается термином, который используется учеными: они говорят, что нейрон **возбуждается** при достижении входным сигналом порогового значения.

Ступенчатую функцию можно усовершенствовать. Представленную ниже S-образную функцию называют **сигмоидой**, или сигмоидальной функцией. Резкие прямоугольные границы ступенчатой функции в ней сглажены, что делает ее более естественной и реалистичной. Природа не любит острых углов!



Сглаженная S-образная сигмоида — это и есть то, что мы будем использовать для создания собственной нейронной сети. Исследователями в области искусственного интеллекта используются также другие функции аналогичного вида, но сигмоида проста и очень популярна, так что она будет для нас отличным выбором.

Сигмоида, которую иногда называют также **логистической функцией**, описывается следующей формулой:

$$y = \frac{1}{1 + e^{-x}}$$

Это выражение не настолько устрашающее, как поначалу может показаться. Буквой **e** в математике принято обозначать константу, равную 2,71828... Это очень интересное число, которое встречается во многих областях математики и физики, а причина, по которой я использовал в нем многоточие (...), заключается в том, что запись десятичных знаков может быть продолжена до бесконечности. Для подобных чисел существует причудливое название — **трансцендентные числа**. Все это, конечно, интересно, но для наших целей вполне достаточно считать, что это число просто равно 2,71828. Входное значение берется с отрицательным знаком, и **e** возводится в степень **-x**. Результат прибавляется к 1, что дает нам $1+e^{-x}$. Наконец, мы обращаем последнюю сумму, т.е. делим 1 на $1+e^{-x}$. Это и есть то, что делает приведенная выше функция с входным значением **x** для того, чтобы предоставить нам выходное значение **y**. Поэтому на самом деле ничего страшного в ней нет.

Просто ради интереса следует отметить, что при нулевом значении **x** выражение e^{-x} принимает значение 1, поскольку возведение любого числа в нулевую степень всегда дает 1. Поэтому **y** становится равным 1 / (1 + 1) или просто 1/2, т.е. половине. Следовательно, базовая сигмоида пересекает ось *y* при **y=1/2**.

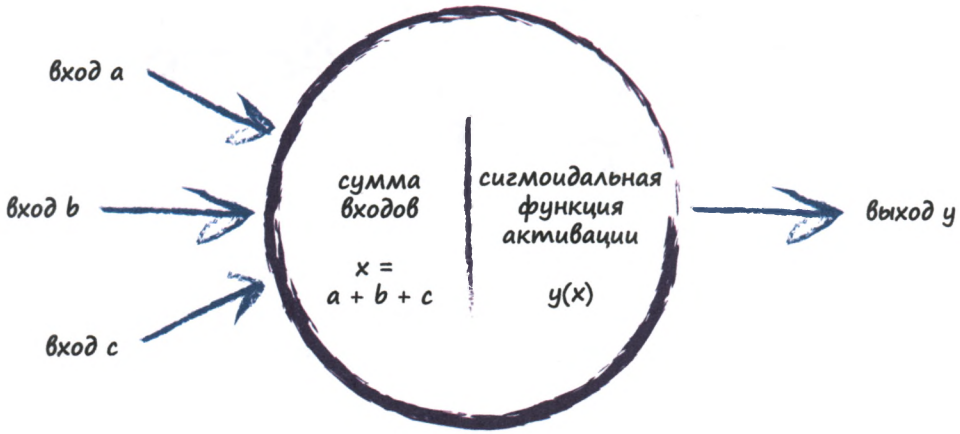
Существует еще одна, очень веская, причина для того, чтобы из множества всех S-образных функций, которые можно было бы использовать для определения выходного значения нейрона, выбрать

именно сигмоиду. Дело в том, что выполнять расчеты с сигмоидой намного проще, чем с любой другой S-образной функцией, и вскоре вы на практике убедитесь в том, что это действительно так.

Вернемся к нейронам и посмотрим, как мы можем смоделировать искусственный нейрон.

Первое, что следует понять, — это то, что реальные биологические нейроны имеют несколько входов, а не только один. Мы уже сталкивались с этим на примере булевой логической машины с двумя входами, поэтому идея более чем одного входа не будет для вас чем-то новым или необычным.

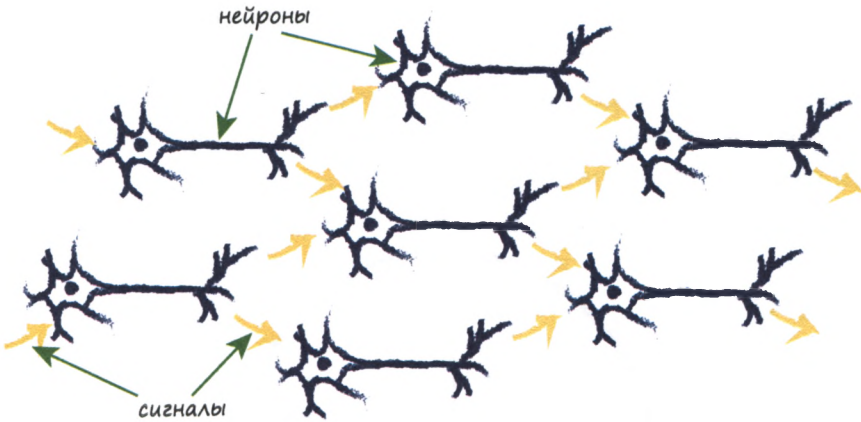
Но что нам делать со всеми этими входами? Мы будем просто комбинировать их, суммируя соответствующие значения, и результирующая сумма будет служить входным значением для сигмоиды, управляющей выходным значением. Такая схема отражает принцип работы нейронной сети. Приведенная ниже диаграмма иллюстрирует идею комбинирования входных значений и сравнения результирующей суммы с пороговым значением.



Если комбинированный сигнал недостаточно сильный, то сигмоида подавляет выходной сигнал. Если же сумма **x** достаточно велика, то функция возбуждает нейрон. Интересно отметить, что даже если только один сигнал достаточно сильный, в то время как все остальные имеют небольшую величину, то и этого может вполне хватить для возбуждения нейрона. Более того, нейрон может возбудиться

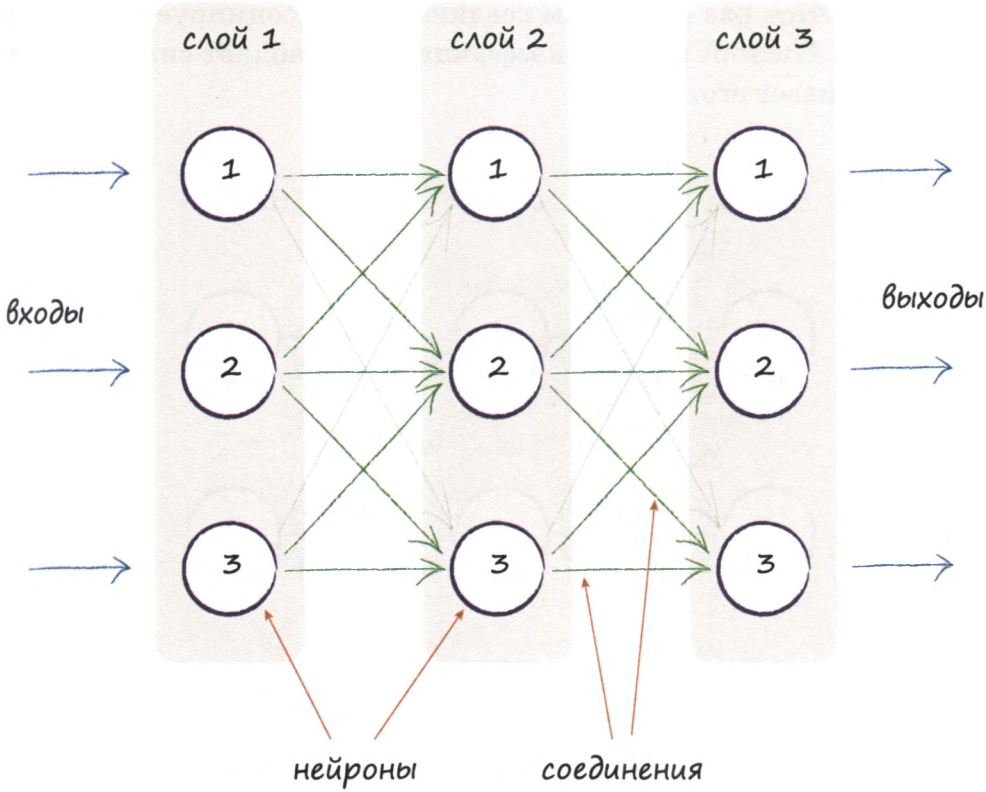
и тогда, когда каждый из сигналов, взятых по отдельности, имеет недостаточную величину, но, будучи взятыми вместе, они обеспечивают превышение порога. В этом уже чувствуется прототип более сложных и в некотором смысле неопределенных вычислений, на которые способны подобные нейроны.

Электрические сигналы воспринимаются дендритами, где они комбинируются, формируя более сильный сигнал. Если этот сигнал превышает порог, нейрон возбуждается, и сигнал передается через аксон к терминалам, откуда он поступает на дендриты следующего нейрона. Связанные таким способом нейроны схематически изображены на приведенной ниже иллюстрации.



На этой схеме бросается в глаза то, что каждый нейрон принимает входной сигнал от нескольких находящихся перед ним нейронов и, в свою очередь, также передает сигнал многим другим в случае возбуждения.

Одним из способов воспроизведения такого поведения нейронов, наблюдаемого в живой природе, в искусственной модели является создание многослойных нейронных структур, в которых каждый нейрон соединен с каждым из нейронов в предшествующем и последующем слоях. Эта идея поясняется на следующей иллюстрации.



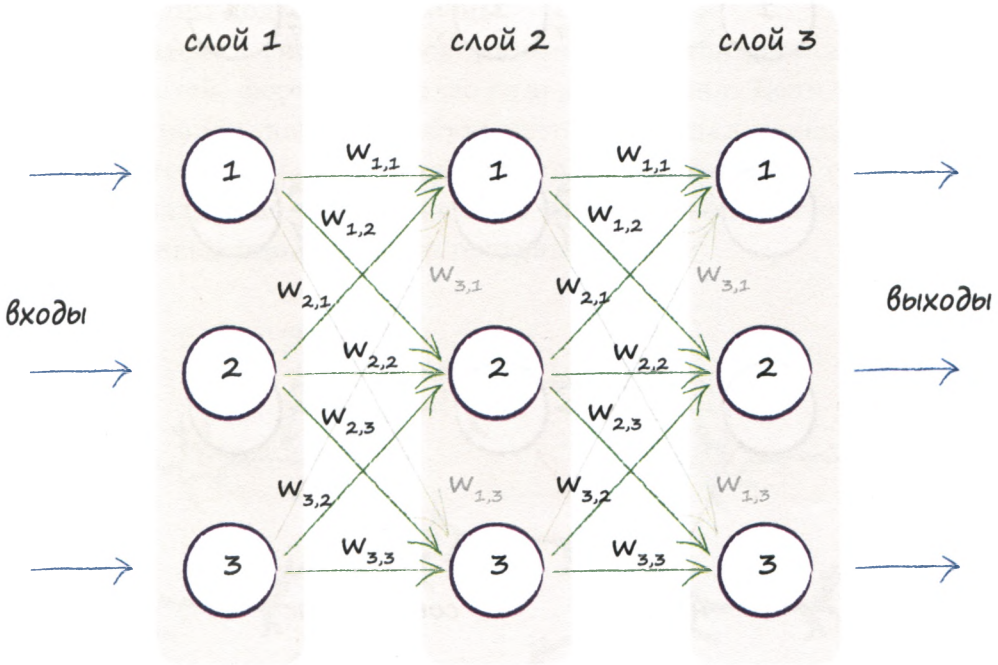
На этой иллюстрации представлены три слоя, каждый из которых включает три искусственных нейрона, или **узла**. Как нетрудно заметить, здесь каждый узел соединен с каждым из узлов предшествующего и последующего слоев.

Прекрасно! Но в какой части этой застывшей структуры заключена способность к обучению? Что мы должны регулировать, реагируя на данные тренировочных примеров? Есть ли здесь параметр, который можно было бы улучшать подобно тому, как ранее мы делали это с наклоном прямой линейного классификатора?

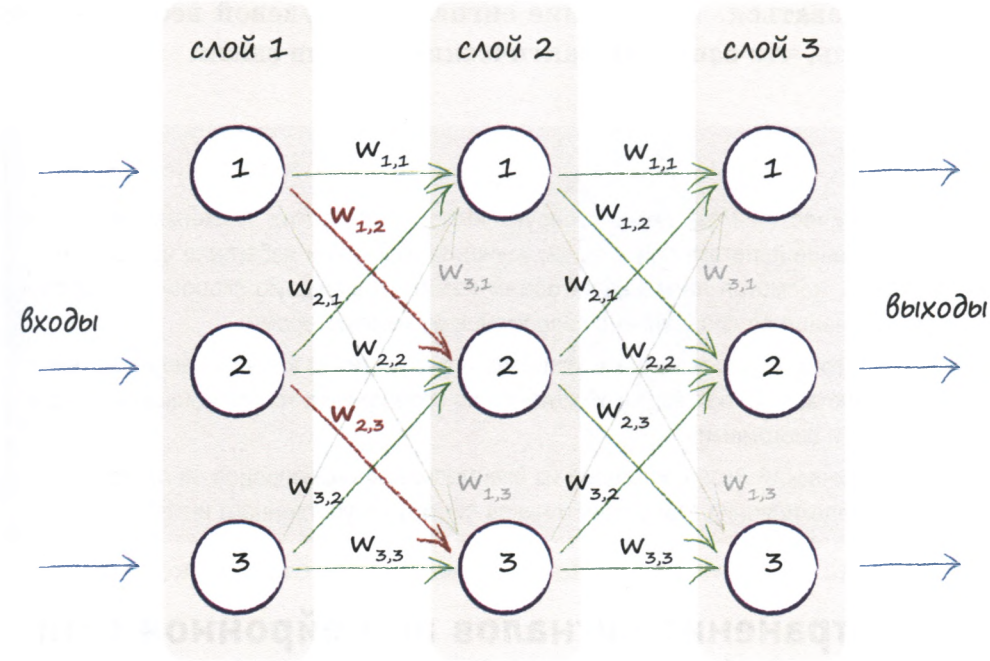
Наиболее очевидной величиной, регулировать которую мы могли бы, является сила связи между узлами. В пределах узла мы могли бы регулировать суммирование входных значений или же форму сигмоиды, но это уже немного сложнее предыдущей регулировки.

Если работает более простой подход, то давайте им и ограничимся! На следующей диаграмме вновь показаны соединенные между собой

узлы, но на этот раз с каждым соединением ассоциируется определенный **вес**. Низкий весовой коэффициент ослабляет сигнал, высокий — усиливает его.



Следует сказать несколько слов о небольших индексах, указанных рядом с коэффициентами. Например, символ $W_{2,3}$ обозначает весовой коэффициент, связанный с сигналом, который передается от узла 2 данного слоя к узлу 3 следующего слоя. Следовательно, $W_{1,2}$ — это весовой коэффициент, который ослабляет или усиливает сигнал, передаваемый от узла 1 к узлу 2 следующего слоя. Чтобы проиллюстрировать эту идею, на следующей диаграмме оба этих соединения между первым и вторым слоями выделены цветом.



Вы могли бы вполне обоснованно подвергнуть сомнению данный замысел и задаться вопросом о том, почему каждый узел слоя должен быть связан с каждым из узлов предыдущего и последующего слоев. Это требование не является обязательным, и слои можно соединять между собой любым мыслимым способом. Мы не рассматриваем здесь другие возможные способы по той причине, что благодаря однородности описанной схемы полного взаимного соединения нейронов закодировать ее в виде компьютерных инструкций на самом деле значительно проще, чем любую другую схему, а также потому, что наличие большего количества соединений, чем тот их обязательный минимум, который может потребоваться для решения определенной задачи, не принесет никакого вреда. Если дополнительные соединения действительно не нужны, то процесс обучения ослабит их влияние.

Что под этим подразумевается? Это означает, что, как только сеть научится улучшать свои выходные значения путем уточнения весовых коэффициентов связей внутри сети, некоторые веса обнулятся или станут близкими к нулю. В свою очередь, это означает, что такие связи не будут оказывать влияния на сеть, поскольку их сигналы не

будут передаваться. Умножение сигнала на нулевой вес дает в результате нуль, что означает фактический разрыв связи.

Резюме

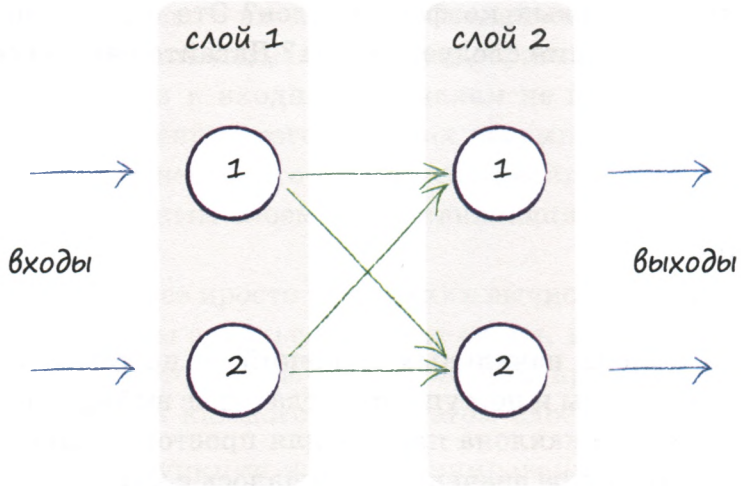
- Биологический мозг демонстрирует выполнение таких сложных задач, как управление полетом, поиск пищи, изучение языка или избегание встреч с хищниками, несмотря на меньший объем памяти и меньшую скорость обработки информации по сравнению с современными компьютерами.
- Кроме того, биологический мозг невероятно устойчив к повреждениям и несовершенству обрабатываемых сигналов по сравнению с традиционными компьютерными системами.
- Биологический мозг, состоящий из взаимосвязанных нейронов, является источником вдохновения для разработчиков систем искусственного интеллекта.

Распространение сигналов по нейронной сети

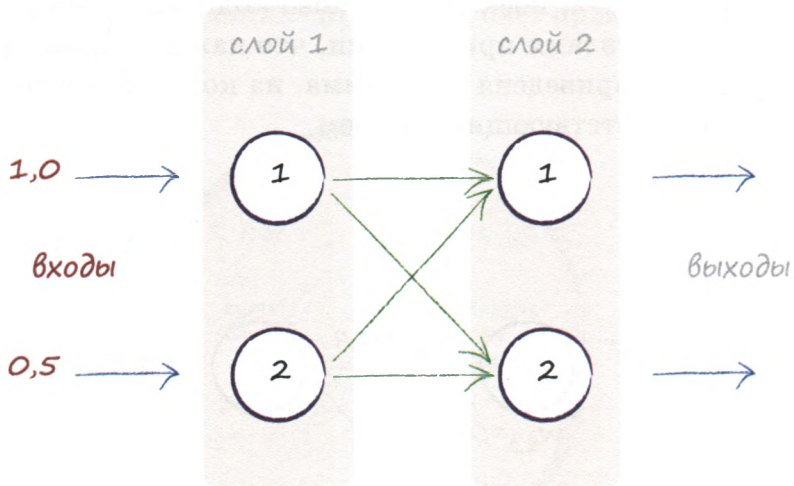
Приведенная выше картинка с тремя слоями нейронов, каждый из которых связан с каждым из нейронов в предыдущем и следующем слоях, выглядит довольно привлекательно.

Однако интуиция подсказывает нам, что рассчитать распространение входных сигналов по всем слоям, пока они не превратятся в выходные сигналы, не так-то просто, и нам, скажем так, придется хорошенько потрудиться!

Я согласен, что эта работа непростая, но также важно показать, как работает этот механизм, чтобы мы всегда четко представляли себе, что происходит в нейронной сети на самом деле, даже если впоследствии мы используем компьютер, который выполнит вместо нас всю работу. Поэтому мы попытаемся проделать необходимые вычисления на примере меньшей нейронной сети, состоящей всего лишь из двух слоев, каждый из которых включает по 2 нейрона, как показано ниже.



Предположим, что сигналам на входе соответствуют значения 1,0 и 0,5.



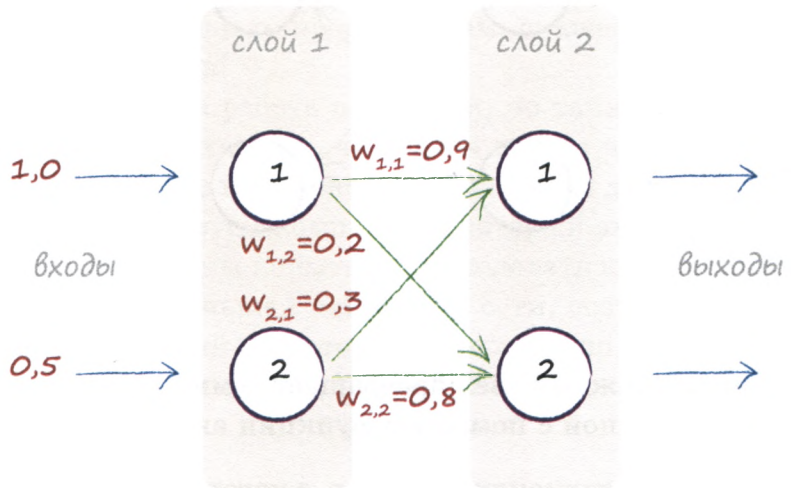
Как и раньше, каждый узел превращает сумму двух входных сигналов в один выходной с помощью функции активации. Мы также будем использовать сигмоиду $y = \frac{1}{1+e^{-x}}$, с которой вы до этого познакомились, где **x** — это сумма сигналов, поступающих в нейрон, а **y** — выходной сигнал этого нейрона.

А что насчет весовых коэффициентов? Это очень хороший вопрос: с какого значения следует начать? Давайте начнем со случайных весов:

- $W_{1,1} = 0,9$
- $W_{1,2} = 0,2$
- $W_{2,1} = 0,3$
- $W_{2,2} = 0,9$

Выбор случайных начальных значений — не такая уж плохая идея, и именно так мы и поступали, когда ранее выбирали случайное начальное значение наклона прямой для простого линейного классификатора. Случайное значение улучшалось с каждым очередным тренировочным примером, используемым для обучения классификатора. То же самое должно быть справедливым и для весовых коэффициентов связей в нейронных сетях.

В данном случае, когда сеть небольшая, мы имеем всего четыре весовых коэффициента, поскольку таково количество всех возможных связей между узлами при условии, что каждый слой содержит по два узла. Ниже приведена диаграмма, на которой все связи промаркированы соответствующим образом.

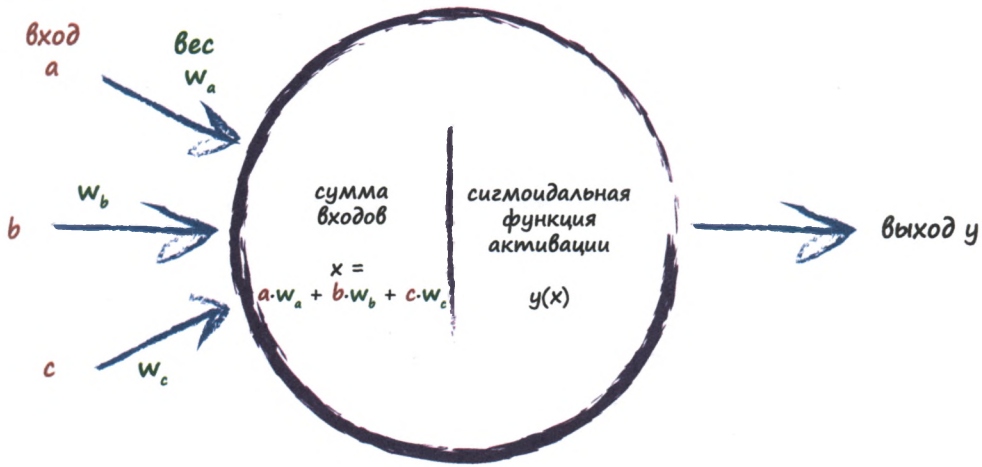


Приступим к вычислениям!

Первый слой узлов — входной, и его единственное назначение — представлять входные сигналы. Таким образом, во входных узлах функция активации к входным сигналам не применяется. Мы не выдвигаем в отношении этого никаких разумных доводов и просто принимаем как данность, что первый слой нейронных сетей является всего лишь входным слоем, представляющим входные сигналы. Вот и все.

С первым слоем все просто — никаких вычислений.

Далее мы должны заняться вторым слоем, в котором потребуется выполнить некоторые вычисления. Нам предстоит определить входной сигнал для каждого узла в этом слое. Помните сигмоиду $y=\frac{1}{1+e^{-x}}$? В этой функции **x** — комбинированный сигнал на входе узла. Данная комбинация образуется из необработанных выходных сигналов связанных узлов предыдущего слоя, сглаженных весовыми коэффициентами связей. Приведенная ниже диаграмма аналогична тем, с которыми вы уже сталкивались, но теперь на ней указано сглаживание поступающих сигналов за счет применения весовых коэффициентов связей.



Для начала сосредоточим внимание на узле 1 слоя 2. С ним связаны оба узла первого, входного слоя. Исходные значения на этих входных узлах равны 1,0 и 0,5. Связи первого узла назначен весовой

коэффициент 0,9, связи второго — 0,3. Поэтому сглаженный входной сигнал вычисляется с помощью следующего выражения:

```
x = (выход первого узла * вес связи) +
  + (выход второго узла * вес связи)
x = (1,0 * 0,9) + (0,5 * 0,3)
x = 0,9 + 0,15
x = 1,05
```

Без сглаживания сигналов мы просто получили бы их сумму 1,0 + 0,5, но мы этого не хотим. Именно с весовыми коэффициентами будет связан процесс обучения нейронной сети по мере того, как они будут итеративно уточняться для получения все лучшего и лучшего результата.

Итак, мы уже имеем значение **x=1,05** для комбинированного сглаженного входного сигнала первого узла второго слоя и теперь располагаем всеми необходимыми данными, чтобы рассчитать для этого узла выходной сигнал с помощью функции активации $y = \frac{1}{1+e^{-x}}$. Попробуйте справиться с этим самостоятельно, используя калькулятор. Вот правильный ответ: **y** = `1 / (1 + 0,3499) = 1 / 1,3499`. Таким образом, **y=0,7408**.

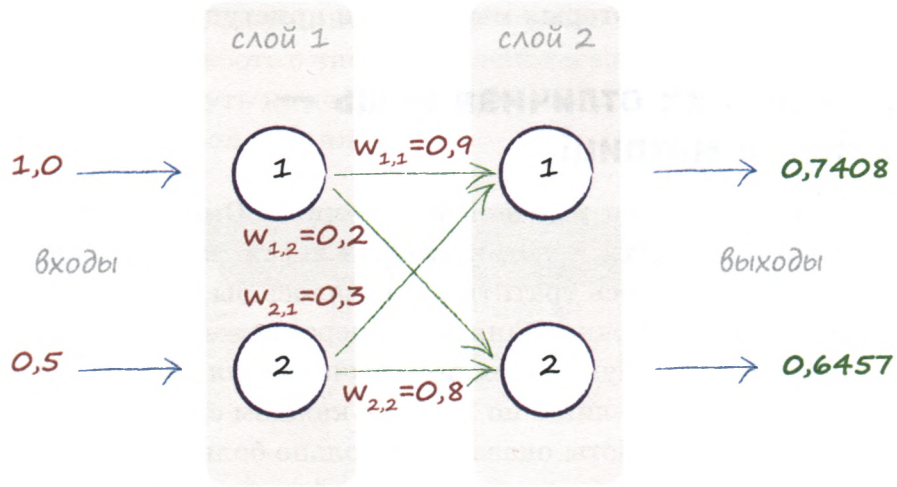
Отличная работа! Мы рассчитали фактический выходной сигнал для одного из двух выходных узлов сети.

Повторим те же вычисления для оставшегося узла — узла 2 второго слоя, т.е. вновь вычислим сглаженный входной сигнал с помощью следующего выражения:

```
x = (выход первого узла * вес связи) +
  + (выход второго узла * вес связи)
x = (1,0 * 0,2) + (0,5 * 0,8)
x = 0,2 + 0,4
x = 0,6
```

Располагая значением **x**, можно рассчитать выходной сигнал узла с помощью функции активации: **y** = `1 / (1 + 0,5488) = 1 / 1,5488`. Таким образом, **y=0,6457**.

Рассчитанные нами выходные сигналы сети представлены на приведенной ниже диаграмме.



Как видите, чтобы получить всего лишь два выходных сигнала для весьма простой нейронной сети, нам пришлось хорошо потрудиться. Лично я не был бы в восторге от необходимости проведения аналогичных расчетов вручную в случае более крупных сетей! К счастью, для выполнения подобных вычислений идеально подходят компьютеры, которые могут быстро справиться с этой работой, избавив вас от лишних хлопот.

Однако даже с учетом этого я не горел бы желанием выписывать соответствующие программные инструкции для сети, насчитывающей более двух слоев, каждый из которых может включать 4, 8 или даже 100 узлов. Стоит ли говорить о том, насколько это утомительное занятие, к тому же оно чревато большим количеством ошибок и опечаток, что вообще может сделать выполнение расчетов практически невозможным.

Но нам повезло. Математики разработали чрезвычайно компактный способ записи операций того типа, которые приходится выполнять для вычисления выходного сигнала нейронной сети, даже если она содержит множество слоев и узлов. Эта компактность благоприятна не только для людей, которые выписывают или читают соответствующие

формулы, но и для компьютеров, поскольку программные инструкции получаются более короткими и выполняются намного быстрее.

В этом компактном подходе предполагается использование **матриц**, к рассмотрению которых мы сейчас и приступим.

Какая все-таки отличная вещь — умножение матриц!

Матрицы пользуются ужасной репутацией. Они вызывают воспоминания о тех долгих и утомительных часах, которые в студенческие годы приходилось тратить на, казалось бы, совершенно бесцельное и никому не нужное занятие — перемножение матриц.

Перед этим мы вручную выполнили вычисления для двухслойной сети, содержащей всего лишь по 2 узла в каждом слое. Однако даже в этом случае объем работы оказался довольно большим, а что тогда можно сказать, например, о сети, состоящей из пяти слоев по 100 узлов? Уже сама по себе запись всех необходимых выражений была бы сложной задачей. Составление комбинаций сигналов, умножение на подходящие весовые коэффициенты, применение сигмоиды для каждого узла... Ух!

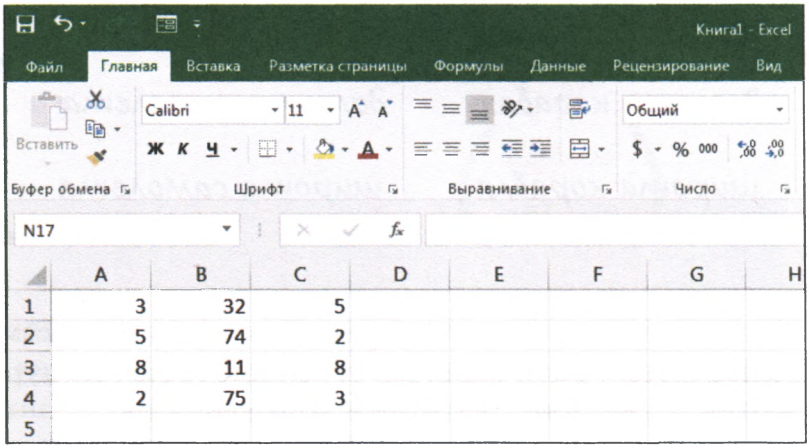
Так чем же нам могут помочь матрицы? Они будут нам полезны в двух отношениях. Во-первых, они обеспечивают сжатую запись операций, придавая им чрезвычайно компактную форму. Это большой плюс для нас, поскольку такая работа утомительна, требует концентрации внимания и чревата ошибками. Во-вторых, в большинстве компьютерных языков программирования предусмотрена работа с матрицами, а поскольку многие операции при этом повторяются, компьютеры распознают это и выполняют их очень быстро.

Резюмируя, можно сказать, что матрицы позволяют формулировать задачи в более простой и компактной форме и обеспечивают их более быстрое и эффективное выполнение.

Теперь, когда вам известны причины, по которым мы собираемся воспользоваться матрицами несмотря на возможный горький опыт знакомства с ними в студенческие годы, давайте демистифицируем матрицы, начав работать с ними.

Матрица — это всего лишь таблица, прямоугольная сетка, образованная числами. Вот и все. Все, что связано с матрицами, ничуть не сложнее этого.

Если вы уже пользовались электронными таблицами, то никаких сложностей при работе с числами, располагающимися в ячейках сетки, у вас возникнуть не должно. Ниже показан пример рабочего листа Excel с числовой таблицей.



Это и есть матрица — таблица или сетка, образованная числами, точно так же, как и следующий пример матрицы размером 2×3.

$$\begin{pmatrix} 23 & 43 & 22 \\ 43 & 12 & 54 \end{pmatrix}$$

При обозначении матриц принято указывать сначала количество строк, а затем количество столбцов. Поэтому размерность данной матрицы не 3×2, а 2×3.

Кроме того, иногда матрицы заключают в квадратные скобки, иногда — в круглые, как это сделали мы.

На самом деле матрицы не обязательно должны состоять из чисел. Они могут включать величины, которым мы дали имена, но не присвоили фактических значений. Поэтому на следующей иллюстрации также показана матрица с элементами в виде **переменных**, каждая из которых имеет определенный смысл и может иметь значение, просто мы пока не указали, что это за значения.

$$\begin{pmatrix} \text{долгота корабля} & \text{долгота самолета} \\ \text{широта корабля} & \text{широта самолета} \end{pmatrix}$$

Чем полезны матрицы, вам станет понятно, когда мы рассмотрим, как они умножаются. Возможно, вы это помните еще из курса высшей математики, а если нет, то освежите свою память.

Вот пример умножения одной простой матрицы на другую.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Как видите, эта операция не сводится к простому перемножению соответствующих элементов. Левый верхний элемент не равен 1*5, как и правый нижний — не 4*8.

Вместо этого матрицы умножаются по другим правилам. Возможно, вы и сами догадаетесь, по каким именно, если внимательно присмотритесь к примеру. Если нет, то взгляните на приведенную ниже иллюстрацию, где показано, как получается ответ для левого верхнего элемента.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5)+(2*7) & (1*6)+(2*8) \\ (3*5)+(4*7) & (3*6)+(4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Вы видите, что верхний левый элемент вычисляется с использованием верхней строки первой матрицы и левого столбца второй. Мысленно перебирая обе последовательности элементов, перемножьте их попарно и сложите полученные результаты. Таким образом, чтобы вычислить левый верхний элемент результирующей матрицы, мы начинаем перемещаться вдоль верхней строки первой матрицы, где находим число 1, а начиная перемещение вниз по левому столбцу второй матрицы, находим число 5. Мы перемножаем их и запоминаем результат, который равен 5. Мы продолжаем перемещаться вдоль ряда и вниз по столбцу и находим элементы 2 и 7. Результат их перемножения, равный 14, мы также запоминаем. При этом мы уже достигли конца строки и столбца, поэтому суммируем числа, которые перед этим запомнили, и получаем в качестве результата число 19. Это и есть левый верхний элемент результирующей матрицы.

Описание получилось длинным, но выполнение самих действий не отнимает много времени. Продолжите далее самостоятельно. На приведенной ниже иллюстрации показано, как вычисляется правый нижний элемент результирующей матрицы.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} = \begin{pmatrix} (1*5) + (2*7) & (1*6) + (2*8) \\ (3*5) + (4*7) & (3*6) + (4*8) \end{pmatrix}$$

$$= \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

Вы вновь можете видеть, как, используя строку и столбец, которые соответствуют искомому элементу (в данном случае вторая строка и второй столбец), мы получаем два произведения, *(3*6)* и *(4*8)*, сложение которых дает 18 + 32 = 50.

Продолжая действовать в таком же духе, находим, что левый нижний элемент равен (3*5) + (4*7) = 15 + 28 = 43, а правый верхний — (1*6) + (2*8) = 6 + 16 = 22.

На следующей иллюстрации продемонстрировано использование переменных вместо чисел:

$$\begin{pmatrix} a & b & .. \\ c & d & .. \end{pmatrix} \begin{pmatrix} e & f \\ g & h \\ .. & .. \end{pmatrix} = \begin{pmatrix} (a*e) + (b*g) + \ldots & (a*f) + (b*h) + \ldots \\ (c*e) + (d*g) + \ldots & (c*f) + (d*h) + \ldots \end{pmatrix}$$

$$= \begin{pmatrix} ae+bg+\ldots & af+bh+\ldots \\ ce+dg+\ldots & cf+dh+\ldots \end{pmatrix}$$

Это всего лишь другой способ описания подхода, с помощью которого мы умножаем матрицы. Используя буквы, вместо которых могут быть подставлены любые числа, мы даем общее описание этого подхода. Такое описание действительно является более общим, чем предыдущее, поскольку применимо к матрицам различных размеров.

Говоря о применимости описанного подхода к матрицам любого размера, следует сделать одну важную оговорку: умножаемые матрицы должны быть совместимыми. Вы поймете, в чем здесь дело, вспомнив, как отбирали попарно элементы из строк первой матрицы и столбцов второй: этот метод не сможет работать, если количество элементов в строке не будет совпадать с количеством элементов в столбце. Поэтому вы не сможете умножить матрицу 2×2 на матрицу 5×5. Попытайтесь это сделать, и сами увидите, почему это невозможно. Чтобы матрицы можно было умножить, количество столбцов в первой из них должно совпадать с количеством строк во второй.

В разных руководствах этот тип матричного умножения может встречаться под разными названиями: **скалярное произведение**, **точечное произведение** или **внутреннее произведение**. В математике возможны и другие типы умножения матриц, такие как перекрестное произведение, но мы будем работать с точечным произведением матриц.

Но почему мы должны углубляться в дебри этого ужасного матричного умножения и отвратительной алгебры? На то есть серьезная причина... Крепитесь!

Посмотрим, что произойдет, если заменить буквы словами, имеющими более непосредственное отношение к нашей нейронной сети. На приведенной ниже иллюстрации вторая матрица имеет размерность 2×1, но матричная алгебра остается прежней.

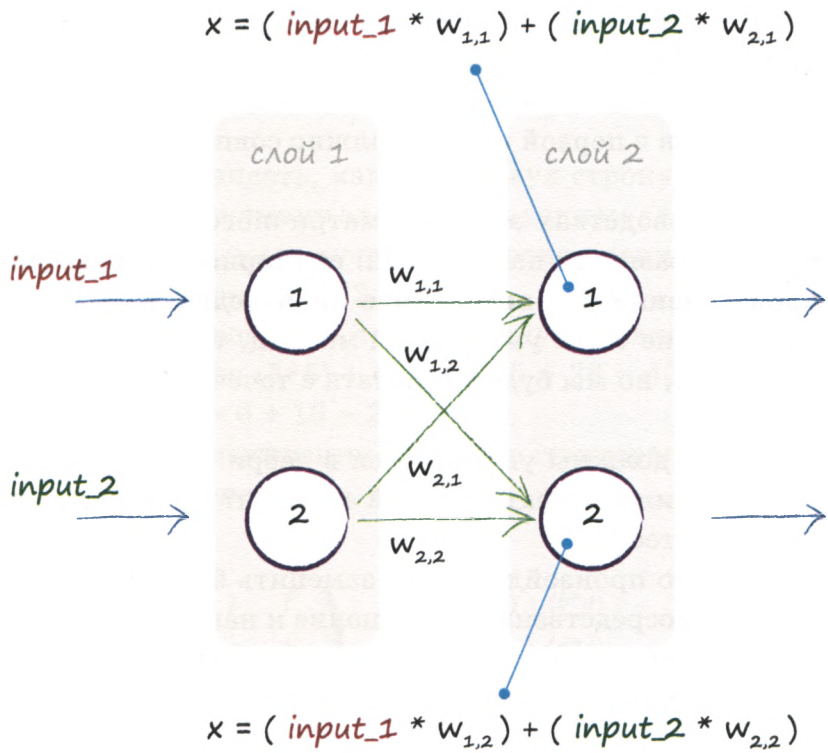
$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input\_1 \\ input\_2 \end{pmatrix} = \begin{pmatrix} (input\_1 * w_{1,1}) + (input\_2 * w_{2,1}) \\ (input\_1 * w_{1,2}) + (input\_2 * w_{2,2}) \end{pmatrix}$$

Иначе как магией это никак не назовешь!

Первая матрица содержит весовые коэффициенты для связей между узлами двух слоев, вторая — сигналы первого, входного слоя. Результатом умножения этих двух матриц является объединенный сглаженный сигнал, поступающий на узлы второго слоя. Присмотритесь повнимательнее, и вы это поймете. На первый узел поступает сумма двух сигналов — сигнала `input_1`, умноженного

на весовой коэффициент $w_{1,1}$, и сигнала input_2, умноженного на весовой коэффициент $w_{2,1}$. Это значения **x** до применения к ним функции активации.

Представим все это в более наглядной форме с помощью иллюстрации.



Описанный подход чрезвычайно полезен.

Почему? Потому что в его рамках все вычисления, которые необходимы для расчета входных сигналов, поступающих на каждый из узлов второго слоя, могут быть выражены с использованием матричного умножения, обеспечивающего чрезвычайно компактную форму записи соответствующих операций:

$$\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$$

Здесь **W** — матрица весов, **I** — матрица входных сигналов, а **X** — результирующая матрица комбинированных сглаженных сигналов, поступающих в слой 2. Символы матриц часто записывают с использованием **полужирного** шрифта, чтобы подчеркнуть, что данные символы представляют матрицы, а не просто одиночные числа.

Теперь нам не нужно заботиться о том, сколько узлов входит в каждый слой. Увеличение количества слоев приводит лишь к увеличению размера матриц. Но количество символов в записи при этом не увеличивается. Она остается по-прежнему компактной, и мы просто записываем произведение матриц в виде **W**·**I**, независимо от количества элементов в каждой из них, будь это 2 или же 200!

Если используемый язык программирования распознает матричную нотацию, то компьютер выполнит все трудоемкие расчеты, связанные с вычислением выражения **X** = **W** · **I**, без предоставления ему отдельных инструкций для каждого узла в каждом слое.

Это просто фантастика! Затратив немного времени на то, чтобы понять суть матричного умножения, мы получили в свое распоряжение мощнейший инструмент для реализации нейронных сетей без каких-либо особых усилий с нашей стороны.

А что насчет функции активации? Здесь все просто и не требует применения матричной алгебры. Все, что нам нужно сделать, — это применить сигмоиду $y = \frac{1}{1+e^{-x}}$ к каждому отдельному элементу матрицы **X**.

Это звучит слишком просто, но так оно и есть, поскольку нам не приходится комбинировать сигналы от разных узлов: это уже сделано, и вся необходимая информация уже содержится в **X**. Как мы уже видели, роль функции активации заключается в применении пороговых значений и подавлении ненужных сигналов с целью имитации поведения биологических нейронов. Поэтому результирующий выходной сигнал второго слоя можно записать в таком виде:

O = сигмоида (**X**)

Здесь символом **O**, выделенным полужирным шрифтом, обозначена матрица, которая содержит все выходные сигналы последнего слоя нейронной сети.

Выражение $X = W \cdot I$ применяется для вычисления сигналов, проходящих от одного слоя к следующему слою. Например, при наличии трех слоев мы просто вновь выполним операцию умножения матриц, использовав выходные сигналы второго слоя в качестве входных для третьего, но, разумеется, предварительно скомбинировав их и сгладив с помощью дополнительных весовых коэффициентов.

На данном этапе мы закончили с теорией и теперь должны посмотреть, как она работает, обратившись к конкретному примеру, который на этот раз будет представлять нейронную сеть несколько большего размера, с тремя слоями по три узла.

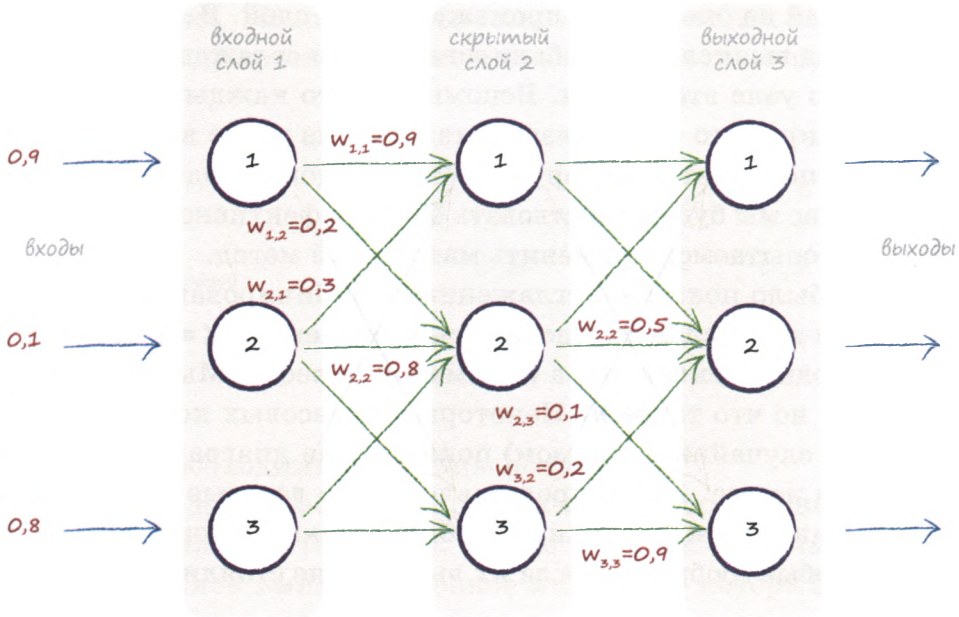
Резюме

- Многие вычисления, связанные с распространением сигналов по нейронной сети, могут быть выполнены с использованием операции **матричного умножения**.
- Использование матричного умножения обеспечивает компактную запись выражений, размер которых не зависит от размеров нейронной сети.
- Что немаловажно, операции матричной алгебры изначально предусмотрены в ряде языков программирования, благодаря чему могут выполняться быстро и эффективно.

Пример использования матричного умножения в сети с тремя слоями

Мы пока еще не использовали матрицы для выполнения расчетов, связанных с вычислением сигналов на выходе нейронной сети. Кроме того, мы пока что не обсуждали примеры с более чем двумя слоями, что было бы гораздо интереснее, поскольку мы должны посмотреть, как обрабатываются выходные сигналы промежуточного слоя в качестве входных сигналов последнего, третьего слоя.

На следующей диаграмме представлен пример нейронной сети, включающей три слоя по три узла. Чтобы избежать загромождения диаграммы лишними надписями, некоторые веса на ней не указаны.



Мы начнем со знакомства с общепринятой терминологией. Как мы знаем, первый слой — **входной**, а второй — **выходной**. Промежуточный слой называется **скрытым слоем**. Это звучит загадочно и таинственно, но, к сожалению, здесь нет ничего загадочного и таинственного. Такое название закрепилось за промежуточным слоем из-за того, что его выходные сигналы не обязательно проявляются как таковые, отсюда и термин “скрытый”.

Приступим к работе над примером, представленным на этой диаграмме. Входными сигналами нейронной сети являются следующие: 0,9; 0,1 и 0,8. Поэтому входная матрица `I` имеет следующий вид.

$$I = \begin{pmatrix} 0,9 \\ 0,1 \\ 0,8 \end{pmatrix}$$

Это было просто, т.е. с первым слоем мы закончили, поскольку его единственная задача — просто представлять входной сигнал.

Следующий на очереди — промежуточный слой. В данном случае нам придется вычислить комбинированные (и сглаженные) сигналы для каждого узла этого слоя. Вспомните, что каждый узел промежуточного скрытого слоя связан с каждым из узлов входного слоя, поэтому он получает некоторую часть каждого входного сигнала. Однако сейчас мы будем действовать более эффективно, чем раньше, поскольку попытаемся применить матричный метод.

Как уже было показано, сглаженные комбинированные входные сигналы для этого слоя определяются выражением $X = W \cdot I$, где I — матрица входных сигналов, а W — матрица весов. Мы располагаем матрицей I, но что такое W? Некоторые из весовых коэффициентов (выбранные случайным образом) показаны на диаграмме для этого примера, но не все. Ниже представлены все весовые коэффициенты (опять-таки, каждый из них выбирался как случайное число). Никакие особые соображения за их выбором не стояли.

$$W_{входной\_скрытый} = \begin{pmatrix} 0{,}9 & 0{,}3 & 0{,}4 \\ 0{,}2 & 0{,}8 & 0{,}2 \\ 0{,}1 & 0{,}5 & 0{,}6 \end{pmatrix}$$

Как нетрудно заметить, весовой коэффициент для связи между первым входным узлом и первым узлом промежуточного скрытого слоя $w_{1,1}=0{,}9$, как и на приведенной выше диаграмме. Точно так же весовой коэффициент для связи между вторым входным узлом и вторым узлом скрытого слоя $w_{2,2}=0{,}8$. На диаграмме не показан весовой коэффициент для связи между третьим входным узлом и первым узлом скрытого слоя $w_{3,1}=0{,}4$.

Постойте-ка, а почему мы снабдили матрицу W индексом “входной_скрытый”? Да потому, что матрица $W_{входной\_скрытый}$ содержит весовые коэффициенты для связей между входным и скрытым слоями. Коэффициенты для связей между скрытым и выходным слоями будут содержаться в другой матрице, которую мы обозначим как $W_{скрытый\_выходной}$.

Эта вторая матрица $W_{скрытый\_выходной}$, элементы которой, как и элементы предыдущей матрицы, представляют собой случайные числа,

приведена ниже. Например, вы видите, что весовой коэффициент для связи между третьим скрытым узлом и третьим выходным узлом $w_{3,3}=0,9$.

$$W_{скрытый\_выходной} = \begin{pmatrix} 0,3 & 0,7 & 0,5 \\ 0,6 & 0,5 & 0,2 \\ 0,8 & 0,1 & 0,9 \end{pmatrix}$$

Отлично, необходимые матрицы получены.

Продолжим нашу работу и определим комбинированный сглаженный сигнал для скрытого слоя. Кроме того, мы должны присвоить ему описательное имя, снабженное индексом, который указывает на то, что это входной сигнал для скрытого, а не последнего слоя. Назовем соответствующую матрицу $X_{скрытый}$:

$$X_{скрытый} = W_{входной\_скрытый} \cdot I$$

Мы не собираемся выполнять вручную все действия, связанные с перемножением матриц. Выполнение этой трудоемкой работы мы будем поручать компьютеру, ведь именно с этой целью мы используем матрицы. В данном случае готовый ответ выглядит так.

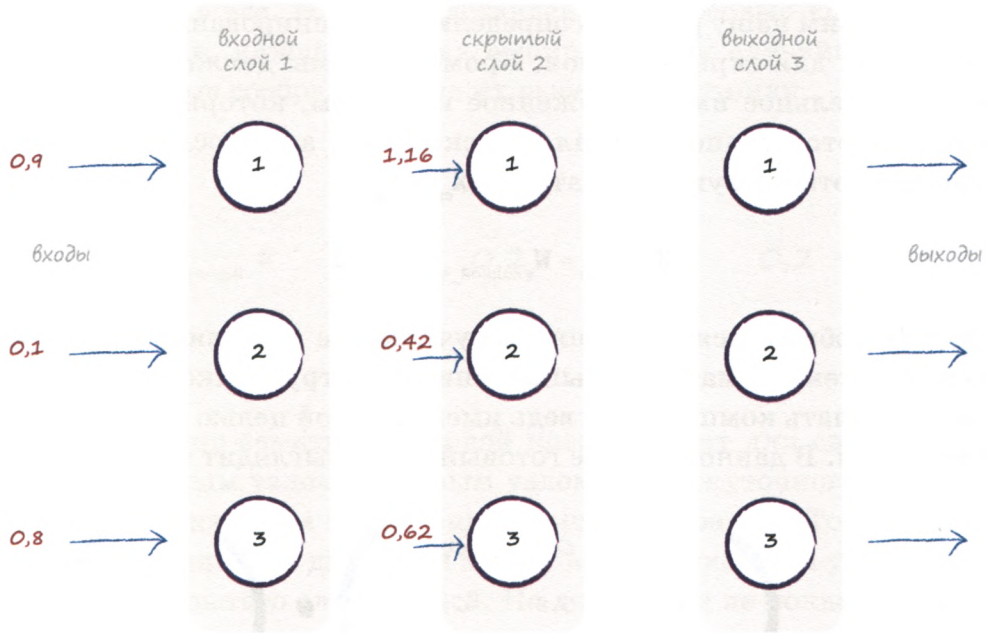
$$X_{скрытый} = \begin{pmatrix} 0,9 & 0,3 & 0,4 \\ 0,2 & 0,8 & 0,2 \\ 0,1 & 0,5 & 0,6 \end{pmatrix} \cdot \begin{pmatrix} 0,9 \\ 0,1 \\ 0,8 \end{pmatrix}$$

$$X_{скрытый} = \begin{pmatrix} 1,16 \\ 0,42 \\ 0,62 \end{pmatrix}$$

Я получил этот ответ с помощью компьютера, и вы научитесь делать это, используя язык программирования Python, о чем мы поговорим далее. Мы не будем заниматься этим сейчас, чтобы пока что не отвлекаться на обсуждение программного обеспечения.

Итак, мы располагаем комбинированными сглаженными входными сигналами скрытого промежуточного слоя, значения которых равны 1,16; 0,42 и 0,62. Для выполнения всех необходимых вычислений были использованы матрицы. Это достижение, которым мы вправе гордиться!

Отобразим входные сигналы для скрытого второго слоя на диаграмме.



Пока что все хорошо, но нам еще остается кое-что сделать. Как вы помните, чтобы отклик слоя на входной сигнал как можно лучше имитировал аналогичный реальный процесс, мы должны применить к узлам функцию активации. Так и поступим:

$$\mathbf{O}_{скрытый} = сигмоида\,(\mathbf{X}_{скрытый})$$

Применяя сигмоиду к каждому элементу матрицы $\mathbf{X}_{скрытый}$, мы получаем матрицу выходных сигналов скрытого промежуточного слоя.

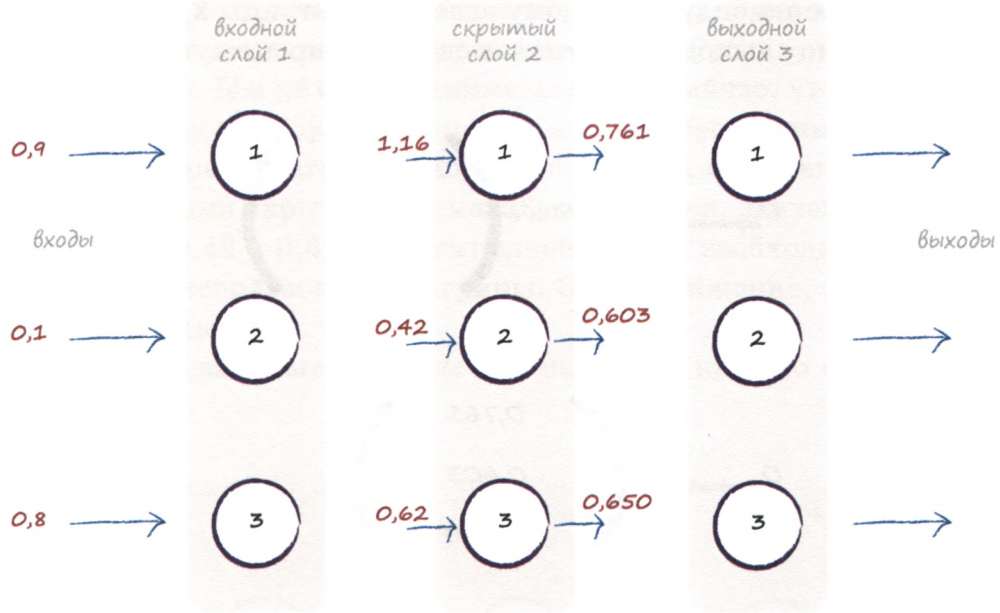
$$O_{скрытый} = сигмоида\begin{pmatrix} 1{,}16 \\ 0{,}42 \\ 0{,}62 \end{pmatrix}$$

$$O_{скрытый} = \begin{pmatrix} 0{,}761 \\ 0{,}603 \\ 0{,}650 \end{pmatrix}$$

Для уверенности давайте проверим, правильно ли вычислен первый элемент. Наша сигмоида имеет вид $y = \frac{1}{1+e^{-x}}$. Полагая **x=1,16**, получаем $e^{1,16} = 0{,}3135$. Это означает, что **y** = `1 / (1 + 0,3135) = 0,761`.

Нетрудно заметить, что все возможные значения этой функции находятся в пределах от 0 до 1. Вернитесь немного назад и взгляните на график логистической функции, если хотите убедиться в том, что это действительно так.

Ух! Давайте немного передохнем и подытожим, что мы к этому времени успели сделать. Мы рассчитали прохождение сигнала через промежуточный слой, т.е. определили значения сигналов на его выходе. Для полной ясности уточним, что эти значения были получены путем применения функции активации к комбинированным входным сигналам промежуточного слоя. Обновим диаграмму в соответствии с этой новой информацией



Если бы это была двухслойная нейронная сеть, то мы сейчас остановились бы, поскольку получили выходные сигналы второго слоя. Однако мы продолжим, поскольку есть еще и третий слой.

Как рассчитать прохождение сигнала для третьего слоя? Точно так же, как и для второго, поскольку эта задача на самом деле ничем не отличается от предыдущей. Нам известна величина входных сигналов, получаемых третьим слоем, как ранее была известна величина входных сигналов, получаемых вторым слоем. У нас также есть весовые коэффициенты для связей между узлами, ослабляющие сигналы. Наконец, чтобы отклик сети максимально правдоподобно имитировал естественный процесс, мы по-прежнему можем применить функцию активации. Поэтому вам стоит запомнить следующее: независимо от количества слоев в нейронной сети, вычислительная процедура для каждого из них одинакова — комбинирование входных сигналов, сглаживание сигналов для каждой связи между узлами с помощью весовых коэффициентов и получение выходного сигнала с помощью функции активации. Нам безразлично, сколько слоев образуют нейронную сеть — 3, 53 или 103, ведь к любому из них применяется один и тот же подход.

Итак, продолжим вычисления и рассчитаем сглаженный комбинированный входной сигнал $\mathbf{X} = \mathbf{W} \cdot \mathbf{I}$ для третьего слоя.

Входными сигналами для третьего слоя служат уже рассчитанные нами выходные сигналы второго слоя $\mathbf{O}_{скрытый}$. При этом мы должны использовать весовые коэффициенты для связей между узлами второго и третьего слоев $\mathbf{W}_{скрытый\_выходной}$, а не те, которые мы уже использовали для первого и второго слоев. Следовательно, мы имеем:

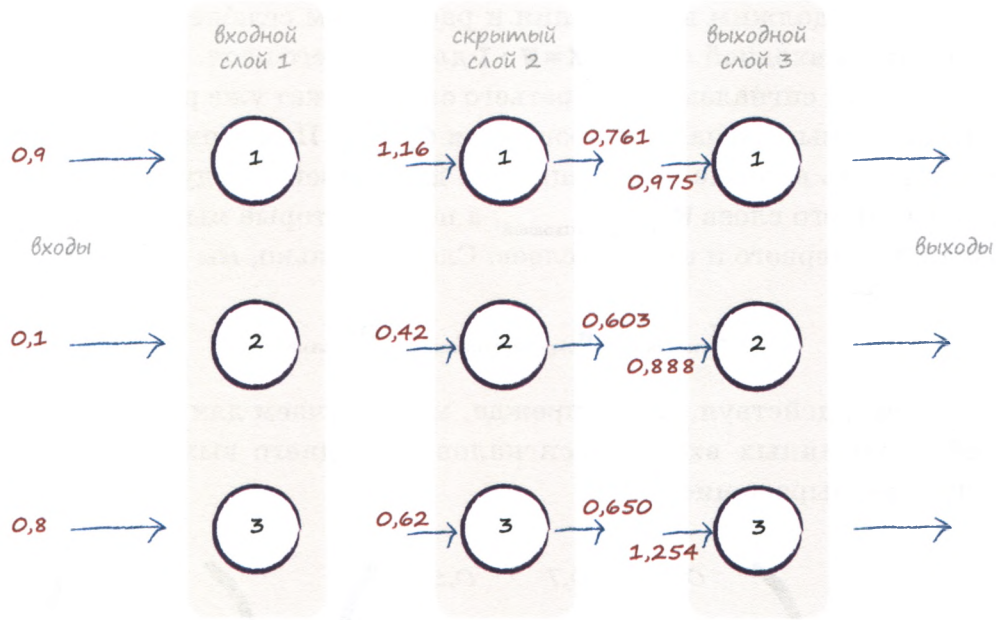
$$\mathbf{X}_{выходной} = \mathbf{W}_{скрытый\_выходной} \cdot \mathbf{O}_{скрытый}$$

Поэтому, действуя, как и прежде, мы получаем для сглаженных комбинированных входных сигналов последнего выходного слоя следующее выражение:

$$X_{выходной} = \begin{pmatrix} 0{,}3 & 0{,}7 & 0{,}5 \\ 0{,}6 & 0{,}5 & 0{,}2 \\ 0{,}8 & 0{,}1 & 0{,}9 \end{pmatrix} \cdot \begin{pmatrix} 0{,}761 \\ 0{,}603 \\ 0{,}650 \end{pmatrix}$$

$$X_{выходной} = \begin{pmatrix} 0{,}975 \\ 0{,}888 \\ 1{,}254 \end{pmatrix}$$

Обновленная диаграмма отражает наш прогресс в расчете преобразования начальных сигналов, поступающих на узлы первого слоя, в сглаженные комбинированные сигналы, поступающие на узлы последнего слоя, в процессе их распространения по нейронной сети.

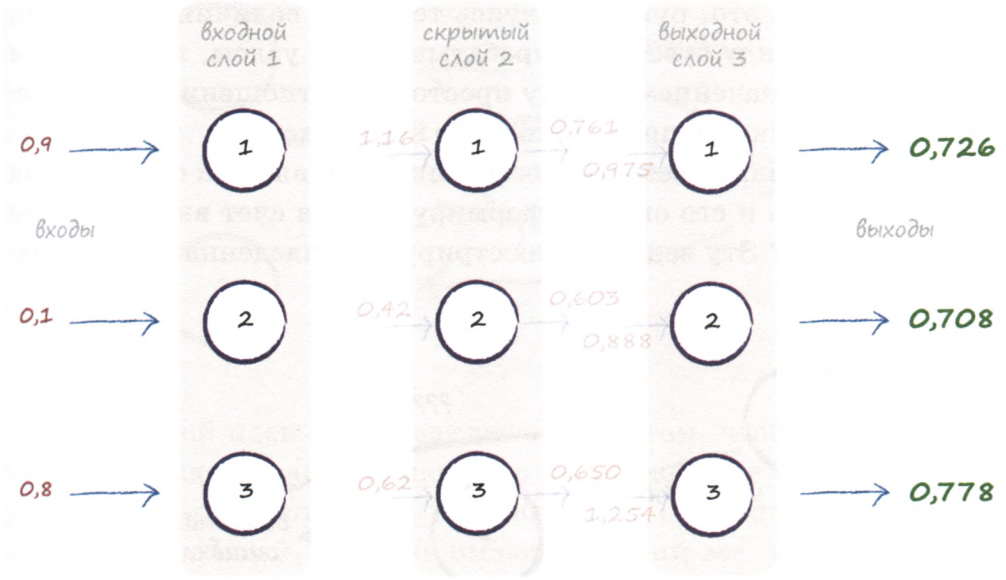


Все, что нам остается, — это применить сигмоиду, и сделать это не составляет труда.

$$O_{выходной} = сигмоида\begin{pmatrix} 0{,}975 \\ 0{,}888 \\ 1{,}254 \end{pmatrix}$$

$$O_{выходной} = \begin{pmatrix} 0{,}726 \\ 0{,}708 \\ 0{,}778 \end{pmatrix}$$

Есть! Мы получили сигналы на выходе нейронной сети. Опять-таки, отобразим текущую ситуацию на обновленной диаграмме.



Таким образом, в нашем примере нейронной сети с тремя слоями выходные сигналы имеют следующую величину: 0,726;ы 0,708 и 0,778.

Итак, нам удалось успешно описать распространение сигналов по нейронной сети, т.е. определить величину выходных сигналов при заданных величинах входных сигналов.

Что дальше?

Наш следующий шаг заключается в сравнении выходных сигналов нейронной сети с данными тренировочного примера для определения ошибки. Нам необходимо знать величину этой ошибки, чтобы можно было улучшить выходные результаты путем изменения параметров сети.

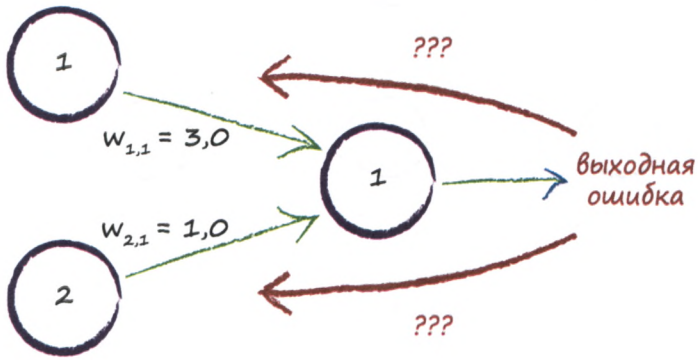
Пожалуй, эта часть работы наиболее трудна для понимания, поэтому мы будем продвигаться не спеша, постепенно знакомясь с основными идеями в процессе работы.

Корректировка весовых коэффициентов в процессе обучения нейронной сети

Ранее мы улучшали поведение простого линейного классификатора путем регулирования параметра наклона линейной функции

узла. Мы делали это, руководствуясь текущей величиной ошибки, т.е. разности между ответом, вырабатываемым узлом, и известным нам истинным значением. Ввиду простоты соотношения между величинами ошибки и поправки это было несложно.

Как нам обновлять весовые коэффициенты связей в случае, если выходной сигнал и его ошибка формируются за счет вкладов более чем одного узла? Эту задачу иллюстрирует приведенная ниже диаграмма.

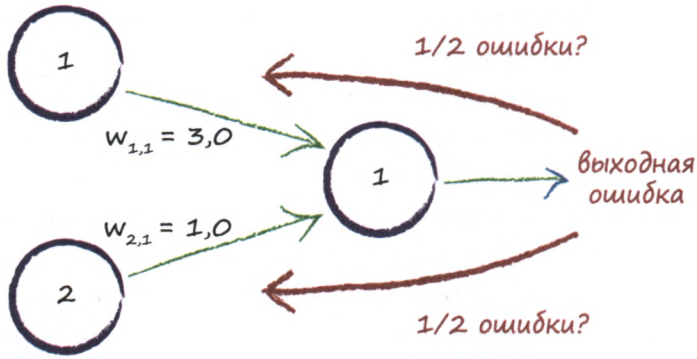


Когда на выходной узел поступал сигнал только от одного узла, все было намного проще. Но как использовать ошибку выходного сигнала при наличии двух входных узлов?

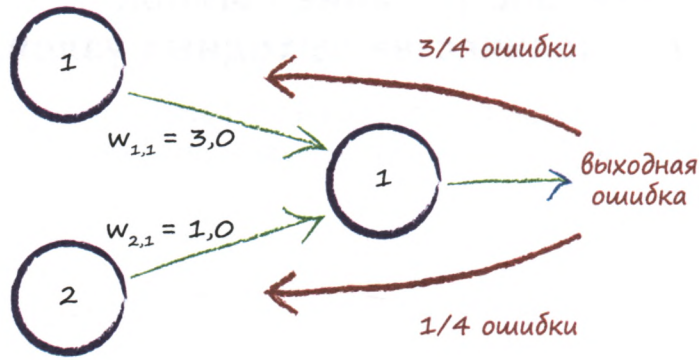
Использовать всю величину ошибки для обновления только одного весового коэффициента не представляется разумным, поскольку при этом полностью игнорируются другая связь и ее вес. Но ведь величина ошибки определяется вкладами всех связей, а не только одной.

Правда, существует небольшая вероятность того, что за ошибку ответственна только одна связь, но эта вероятность пренебрежимо мала. Если мы изменили весовой коэффициент, уже имеющий "правильное" значение, и тем самым ухудшили его, то при выполнении нескольких последующих итераций он должен улучшиться, так что не все потеряно.

Один из возможных способов состоит в том, чтобы распределить ошибку поровну между всеми узлами, вносящими вклад, как показано на следующей диаграмме.



Суть другой идеи также заключается в том, чтобы распределять ошибку между узлами, однако это распределение не обязано быть равномерным. Вместо этого бо́льшая доля ошибки приписывается вкладам тех связей, которые имеют больший вес. Почему? Потому что они оказывают большее влияние на величину ошибки. Эту идею иллюстрирует следующая диаграмма.



В данном случае сигнал, поступающий на выходной узел, формируется за счет двух узлов. Весовые коэффициенты связей равны 3,0 и 1,0. Распределив ошибку между двумя узлами пропорционально их весам, вы увидите, что для обновления значения первого, большего веса следует использовать 3/4 величины ошибки, тогда как для обновления значения второго, меньшего веса — 1/4.

Мы можем расширить эту идею на случаи с намного бо́льшим количеством узлов. Если бы выходной узел был связан со ста входными узлами, мы распределили бы выходную ошибку между всеми ста

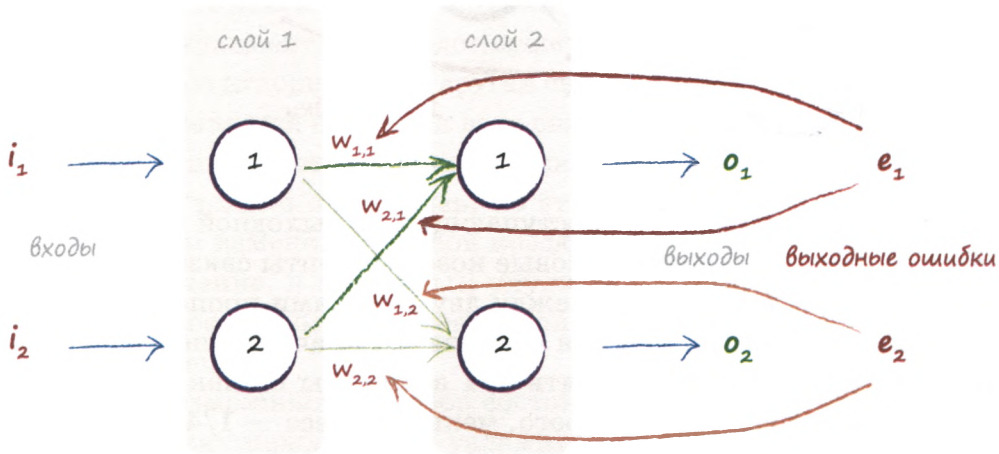
связями **пропорционально** их вкладам, размер которых определяется весами соответствующих связей.

Как вы могли заметить, мы используем весовые коэффициенты в двух целях. Во-первых, они учитываются при расчете распространения сигналов по нейронной сети от входного слоя до выходного. Мы интенсивно использовали их ранее именно в таком качестве. Во-вторых, мы используем веса для распространения ошибки в обратном направлении — от выходного слоя вглубь сети. Думаю, вы не будете удивлены, узнав, что этот метод называется **обратным распространением ошибки** (обратной связью) в процессе обучения нейронной сети.

Если бы выходной слой имел два узла, мы повторили бы те же действия и для второго узла. Второй выходной узел будет характеризоваться собственной ошибкой, распределяемой аналогичным образом между соответствующим количеством входных узлов. Теперь продолжим наше рассмотрение.

Обратное распространение ошибок от большего количества выходных узлов

На следующей диаграмме отображена простая сеть с двумя входными узлами, но на этот раз с двумя выходными узлами.



Ошибка может формироваться на обоих узлах — фактически эта ситуация очень похожа на ту, которая возникает, когда сеть еще не обучалась. Вы видите, что для коррекции весов внутренних связей нужна информация об ошибках в обоих узлах. Мы можем использовать прежний подход и распределять ошибку выходного узла между связанными с ним узлами пропорционально весовым коэффициентам соответствующих связей.

В действительности тот факт, что сейчас имеется более чем один выходной узел, ничего не меняет. Мы просто повторяем для второго узла те же действия, которые уже выполняли для первого. Почему все так просто? Эта простота объясняется тем, что связи одного выходного узла не зависят от связей другого. Между этими двумя наборами связей отсутствует какая-либо зависимость.

Вернемся к диаграмме, на которой ошибка на первом выходном узле обозначена как e_1. Не забывайте, что это разность между желаемым значением, предоставляемым тренировочными данными t_1, и фактическим выходным значением o_1. Таким образом, $e_1 = (t_1 - o_1)$. Ошибка на втором выходном узле обозначена как e_2.

На диаграмме видно, что ошибка e_1 распределяется пропорционально весам связей, обозначенным как w_{11} и w_{21}. Точно так же ошибка e_2 должна распределяться пропорционально весам w_{12} и w_{22}.

Чтобы у вас не возникало никаких сомнений в правильности получаемых результатов, запишем эти доли в явном виде. Ошибка e_1 информирует о величинах поправок для весов w_{11} и w_{21}. При ее распределении между узлами доля e_1, информация о которой используется для обновления w_{11}, определяется следующим выражением:

$$\frac{w_{11}}{w_{11} + w_{21}}$$

Доля e_1, используемая для обновления w_{21}, определяется аналогичным выражением:

$$\frac{w_{21}}{w_{11} + w_{21}}$$

Возможно, эти выражения несколько смущают вас, поэтому рассмотрим их более подробно. За всеми этими символами стоит очень простая идея, которая заключается в том, что узлы, сделавшие больший вклад в ошибочный ответ, получают больший сигнал об ошибке, тогда как узлы, сделавшие меньший вклад, получают меньший сигнал.

Если w_{11} в два раза превышает w_{21} (скажем, $w_{11}=6$, а $w_{21}=3$), то доля e_1, используемая для обновления w_{11}, составляет $6/(6+3) = 6/9 = 2/3$. Тогда для другого, меньшего веса w_{21} должно остаться $1/3$ e_1, что можно подтвердить с помощью выражения $3/(6+3) = 3/9$, результат которого действительно равен $1/3$.

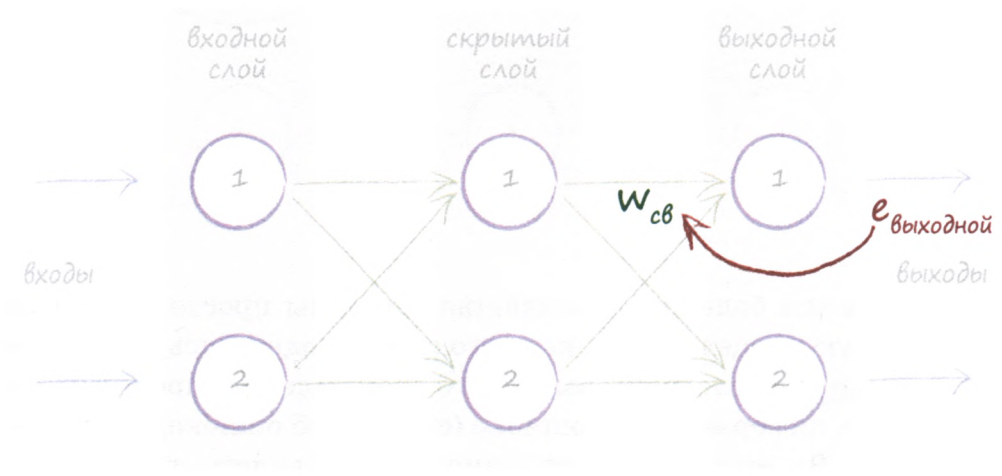
Как и следовало ожидать, при равных весах будут равны и соответствующие доли. Давайте в этом убедимся. Пусть $w_{11}=4$ и $w_{21}=4$, тогда в обоих случаях доля будет составлять $4/(4+4) = 4/8 = 1/2$.

Прежде чем продолжить, сделаем паузу и вернемся на шаг назад, чтобы оценить то, что мы уже успели сделать. Мы знали, что при определении величины поправок для некоторых внутренних параметров сети, в данном случае — весов связей, необходимо использовать величину ошибки. Вы видели, как это делается для весов, которые сглаживают входные сигналы последнего, выходного слоя нейронной сети. Вы также видели, что увеличение количества выходных узлов не усложняет задачу, поскольку мы повторяем одни и те же действия для каждого выходного узла. Замечательно!

Но как быть, если количество слоев превышает два? Как обновлять веса связей для слоев, далеко отстоящих от последнего, выходного слоя?

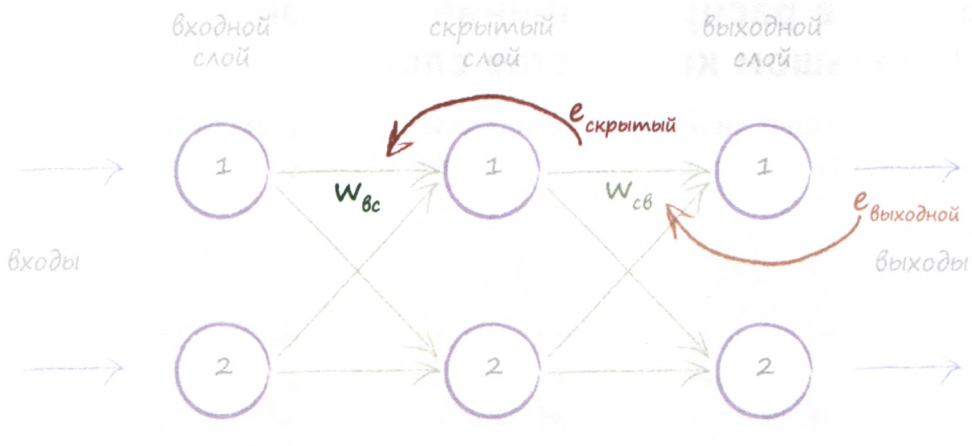
Обратное распространение ошибок при большом количестве слоев

На следующей диаграмме представлен пример простой нейронной сети с тремя слоями: входным, скрытым и выходным.



Продвигаясь в обратном направлении от последнего, выходного слоя (крайнего справа), мы видим, как информация об ошибке в выходном слое используется для определения величины поправок к весовым коэффициентам связей, со стороны которых к нему поступают сигналы. Здесь использованы более общие обозначения $e_{выходной}$ для выходных ошибок и $w_{св}$ для весов связей между скрытым и выходным слоями. Мы вычисляем конкретные ошибки, ассоциируемые с каждой связью, путем распределения ошибки пропорционально соответствующим весам.

Графическое представление позволяет лучше понять, какие вычисления следует выполнить для дополнительного слоя. Мы просто берем ошибки $e_{скрытый}$ на выходе скрытого слоя и вновь распределяем их по предшествующим связям между входным и скрытым слоями пропорционально весовым коэффициентам $w_{вс}$. Следующая диаграмма иллюстрирует эту логику.

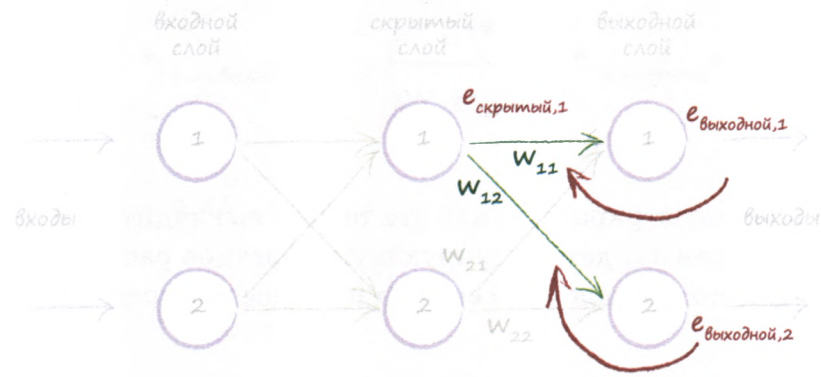


При наличии большего количества слоев мы просто повторили бы описанную процедуру для каждого слоя, продвигаясь в обратном направлении от последнего, выходного слоя. Идея распространения этого потока информации об ошибке (сигнала об ошибке) интуитивно понятна. Вы еще раз имели возможность увидеть, почему этот процесс описывается термином **обратное распространение ошибок**.

Если мы сначала использовали ошибку $\mathbf{e}_{\text{выходной}}$ выходного сигнала узлов выходного слоя, то какую ошибку $\mathbf{e}_{\text{скрытый}}$ мы собираемся использовать для узлов скрытого слоя? Это хороший вопрос, поскольку мы не можем указать очевидную ошибку для узла в таком слое. Из расчетов, связанных с распространением входных сигналов в прямом направлении, мы знаем, что у каждого узла скрытого слоя в действительности имеется только один выход. Вспомните, как мы применяли функцию активации к взвешенной сумме всех входных сигналов данного узла. Но как определить ошибку для такого узла?

У нас нет целевых или желаемых выходных значений для скрытых узлов. Мы располагаем лишь целевыми значениями узлов последнего, выходного слоя, и эти значения происходят из тренировочных примеров. Попытаемся найти вдохновение, взглянув еще раз на приведенную выше диаграмму. С первым узлом скрытого слоя ассоциированы две исходящие из него связи, ведущие к двум узлам выходного слоя. Мы знаем, что можем распределить выходную ошибку между этими связями, поступая точно так же, как и прежде.

Это означает, что с каждой из двух связей, исходящих из узла промежуточного слоя, ассоциируется некоторая ошибка. Мы могли бы воссоединить ошибки этих двух связей, чтобы получить ошибку для этого узла в качестве второго наилучшего подхода, поскольку мы не располагаем фактическим целевым значением для узла промежуточного слоя. Следующая диаграмма иллюстрирует эту идею:



На диаграмме отчетливо видно, что именно происходит, однако для уверенности давайте разберем ее более детально. Нам необходимы величины ошибок для узлов скрытого слоя, чтобы использовать их для обновления весовых коэффициентов связей с предыдущим слоем. Обозначим эти ошибки как $\mathbf{e}_{скрытый}$. Но у нас нет очевидного ответа на вопрос о том, какова их величина. Мы не можем сказать, что ошибка — это разность между желаемым или целевым выходным значением этого узла и его фактическим выходным значением, поскольку данные тренировочного примера предоставляют лишь целевые значения для узлов последнего, выходного слоя. Они не говорят абсолютно ничего о том, какими должны быть выходные сигналы узлов любого другого слоя. В этом и заключается суть сложившейся головоломки.

Мы можем воссоединить ошибки, распределенные по связям, используя обратное распространение ошибок, с которым вы уже познакомились. Поэтому ошибка на первом скрытом узле представляет собой сумму ошибок, распределенных по всем связям, исходящим из

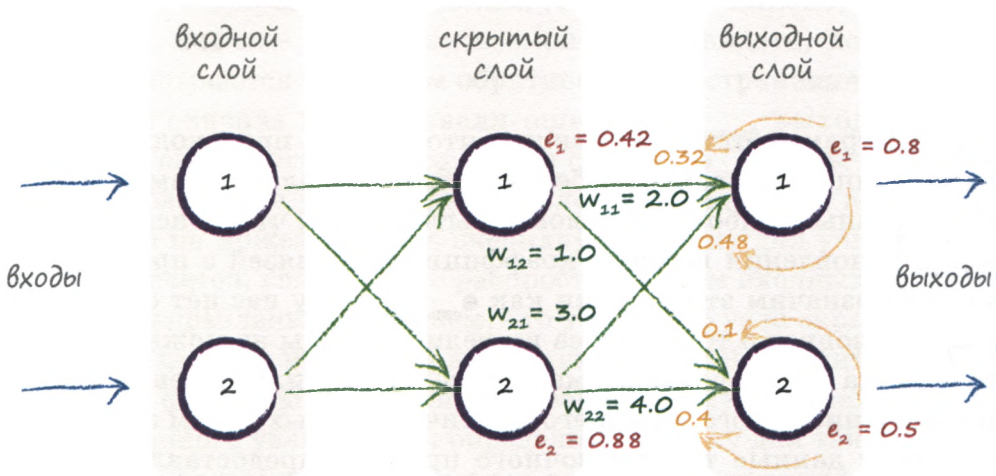
этого узла в прямом направлении. На приведенной выше диаграмме показано, что имеется некоторая доля выходной ошибки $e_{выходной,1}$, приписываемая связи с весом w_{11}, и некоторая доля выходной ошибки $e_{выходной,2}$, приписываемая связи с весом w_{12}.

Вышесказанное можно записать в виде следующего выражения:

$$e_{скрытый,1} = \text{сумма ошибок, распределенных по связям } w_{11} \text{ и } w_{12}$$

$$= e_{выходной,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{выходной,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

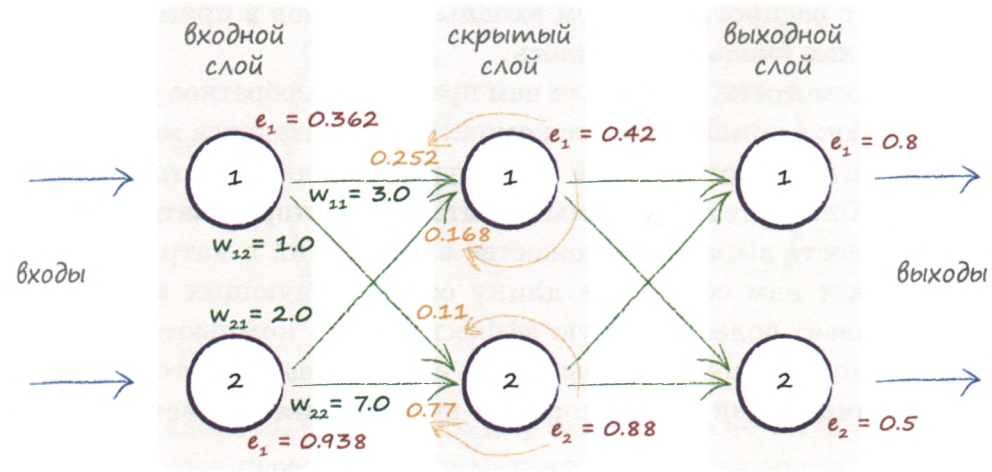
Чтобы проиллюстрировать, как эта теория выглядит на практике, приведем диаграмму, демонстрирующую обратное распространение ошибок в простой трехслойной сети на примере конкретных данных.



Проследим за обратным распространением одной из ошибок. Вы видите, что после распределения ошибки 0,5 на втором узле выходного слоя между двумя связями с весами 1,0 и 4,0 мы получаем доли, равные 0,1 и 0,4 соответственно. Также можно видеть, что объединенная ошибка на втором узле скрытого слоя представляет

собой сумму распределенных ошибок, в данном случае равных 0,48 и 0,4, сложение которых дает 0,88.

На следующей диаграмме демонстрируется применение той же методики к слою, который предшествует скрытому.



Резюме

- Нейронные сети обучаются посредством уточнения весовых коэффициентов своих связей. Этот процесс управляется **ошибкой** — разностью между правильным ответом, предоставляемым тренировочными данными, и фактическим выходным значением.
- Ошибка на выходных узлах определяется простой разностью между желаемым и фактическим выходными значениями.
- В то же время величина ошибки, связанной с внутренними узлами, не столь очевидна. Одним из способов решения этой проблемы является распределение ошибок выходного слоя между соответствующими связями пропорционально весу каждой связи с последующим объединением соответствующих разрозненных частей ошибки на каждом внутреннем узле.

Описание обратного распространения ошибок с помощью матричной алгебры

Можем ли мы упростить трудоемкие расчеты, используя возможности матричного умножения? Ранее, когда мы проводили расчеты, связанные с распространением входных сигналов в прямом направлении, это нам очень пригодилось.

Чтобы посмотреть, удастся ли нам представить обратное распространение ошибок с помощью более компактного синтаксиса матриц, опишем все шаги вычислительной процедуры, используя матричные обозначения. Кстати, тем самым мы попытаемся **векторизовать** процесс.

Возможность выразить множество вычислений в матричной форме позволяет нам сократить длину соответствующих выражений и обеспечивает более высокую эффективность компьютерных расчетов, поскольку компьютеры могут использовать повторяющийся шаблон вычислений для ускорения выполнения соответствующих операций.

Отправной точкой нам послужат ошибки, возникающие на выходе нейронной сети в последнем, выходном слое. В данном случае выходной слой содержит только два узла с ошибками e_1 и e_2:

$$ошибка_{выходной} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Далее нам нужно построить матрицу для ошибок скрытого слоя. Эта задача может показаться сложной, поэтому мы будем выполнять ее по частям. Первую часть задачи представляет первый узел скрытого слоя. Взглянув еще раз на приведенные выше диаграммы, вы увидите, что ошибка на первом узле скрытого слоя формируется за счет двух вкладов со стороны выходного слоя. Этими двумя сигналами ошибок являются $e_1 * w_{11} / (w_{11} + w_{21})$ и $e_2 * w_{12} / (w_{12} + w_{22})$. А теперь обратите внимание на второй узел скрытого слоя, и вы вновь увидите, что ошибка на нем также формируется за счет двух вкладов:

$e_1 * w_{21} / (w_{21} + w_{11})$ и $e_2 * w_{22} / (w_{22} + w_{12})$. Ранее мы уже видели, как работают эти выражения.

Итак, для скрытого слоя мы имеем следующую матрицу, которая выглядит немного сложнее, чем мне хотелось бы.

$$\text{ошибка}_{\text{скрытый}} = \begin{pmatrix} \dfrac{w_{11}}{w_{11} + w_{21}} & \dfrac{w_{12}}{w_{12} + w_{22}} \\ \dfrac{w_{21}}{w_{21} + w_{11}} & \dfrac{w_{22}}{w_{22} + w_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Было бы здорово, если бы это выражение можно было переписать в виде простого перемножения матриц, которыми мы уже располагаем. Это матрицы весовых коэффициентов, прямого сигнала и выходных ошибок. Преимущества, которые можем при этом получить, огромны.

К сожалению, легкого способа превратить это выражение в сверхпростое перемножение матриц, как в случае распространения сигналов в прямом направлении, не существует. Распутать все эти доли, из которых образованы элементы большой матрицы, непросто. Было бы замечательно, если бы мы смогли представить эту матрицу в виде комбинации имеющихся матриц.

Что можно сделать? Нам позарез нужен способ, обеспечивающий возможность использования матричного умножения, чтобы повысить эффективность вычислений.

Ну что ж, дерзнем!

Взгляните еще раз на приведенное выше выражение. Вы видите, что наиболее важная для нас вещь — это умножение выходных ошибок e_n на связанные с ними веса w_{ij}. Чем больше вес, тем бо́льшая доля ошибки передается обратно в скрытый слой. Это важный момент. В дробях, являющихся элементами матрицы, нижняя часть играет роль нормирующего множителя. Если пренебречь этим фактором, можно потерять лишь масштабирование ошибок, передаваемых по меха-

низму обратной связи. Таким образом, выражение $e_1 * w_{11} / (w_{11} + w_{21})$ упростится до $e_1 * w_{11}$.

Сделав это, мы получим следующее уравнение.

$$\text{ошибка}_{\text{скрытый}} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} \bullet \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

Эта матрица весов напоминает ту, которую мы строили ранее, но она повернута вокруг диагонали, так что правый верхний элемент теперь стал левым нижним, а левый нижний — правым верхним. Такая матрица называется **транспонированной** и обозначается как w^T.

Ниже приведены два примера транспонирования числовых матриц, которые помогут вам лучше понять смысл операции транспонирования. Вы видите, что она применима даже в тех случаях, когда количество столбцов в матрице отличается от количества строк.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}^T = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Итак, мы достигли того, чего хотели, — применили матричный подход к описанию обратного распространения ошибок:

$$ошибка_{скрытый} = w^{T}_{скрытый\_выходной} \cdot ошибка_{выходной}$$

Конечно, это просто замечательно, но правильно ли мы поступили, отбросив нормирующий множитель? Оказывается, что эта упрощенная модель обратного распространения сигналов ошибок работает ничуть не хуже, чем более сложная, которую мы разработали перед этим. В блоге, посвященном данной книге, вы найдете публикацию, в которой представлены результаты расчетов обратного распространения ошибки, выполненных несколькими различными способами:

http://makeyourownneuralnetwork.blogspot.co.uk/2016/07/error-backpropagation-revisted.html

Если наш простой подход действительно хорошо работает, мы оставим его!

Немного подумав, можно прийти к выводу, что даже в тех случаях, когда в обратном направлении распространяются слишком большие или слишком малые ошибки, сеть сама все исправит при выполнении последующих итераций обучения. Важно то, что при обратном распространении ошибок учитываются весовые коэффициенты связей, и это наилучший показатель того, что мы пытаемся справедливо распределить ответственность за возникающие ошибки.

Мы проделали большую работу, очень большую!

Резюме

- Обратное распространение ошибок можно описать с помощью матричного умножения.
- Это позволяет нам записывать выражения в более компактной форме, независимо от размеров нейронной сети, и обеспечивает более эффективное и быстрое выполнение вычислений компьютерами, если в языке программирования предусмотрен синтаксис матричных операций.
- Отсюда следует, что использование матриц обеспечивает повышение эффективности расчетов как для распространения сигналов в прямом направлении, так и для распространения ошибок в обратном направлении.

Сделайте вполне заслуженный перерыв, поскольку следующий теоретический раздел потребует от вас концентрации внимания и приложения умственных усилий.

Как мы фактически обновляем весовые коэффициенты

Однако мы пока что не приступили к решению главной задачи — обновлению весов связей в нейронной сети. Мы работали в этом направлении и уже почти достигли намеченной цели. Нам осталось разобрать лишь одну ключевую идею, чтобы больше не было никаких неясностей.

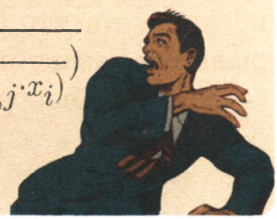
К этому моменту мы научились рассчитывать обратное распространение ошибок до каждого слоя сети. Зачем нам это нужно? Затем, что ошибки подсказывают нам, как должны быть изменены веса связей, чтобы улучшить результирующий общий ответ на выходе нейронной сети. В основном это то, что мы делали с линейным классификатором еще в начале главы.

Однако узлы — это не простые линейные классификаторы. В узлах сигналы суммируются с учетом весов, после чего к ним применяется сигмоида. Но как нам все-таки справиться с обновлением весов для связей, соединяющих эти более сложные узлы? Почему бы не использовать алгебру для непосредственного вычисления весов?

Последний путь нам не подходит ввиду громоздкости соответствующих выкладок. Существует слишком много комбинаций весов и слишком много функций, зависящих от функций, зависящих от других функций и т.д., которые мы должны комбинировать в ходе анализа распространения сигнала по сети. Представьте себе хотя бы небольшую нейронную сеть с тремя слоями и тремя нейронами в каждом слое, подобную той, с которой мы перед этим работали. Как отрегулировать весовой коэффициент для связи между первым входным узлом и вторым узлом скрытого слоя, чтобы сигнал на выходе третьего узла увеличился, скажем, на 0,5? Даже если бы вам повезло и вы сделали это, достигнутый результат мог бы быть разрушен настройкой другого весового коэффициента, улучшающего сигнал другого выходного узла. Как видите, эти расчеты далеко не тривиальны.

Чтобы убедиться в том, насколько они не тривиальны, достаточно взглянуть на приведенное ниже устрашающее выражение, которое представляет выходной сигнал узла выходного слоя как функцию входных сигналов и весовых коэффициентов связей для простой нейронной сети с тремя слоями по три узла. Входной сигнал на узле **i** равен x_i, весовой коэффициент для связи, соединяющей входной узел **i** с узлом **j** скрытого слоя, равен $w_{i,j}$. Аналогичным образом выходной сигнал узла **j** скрытого слоя равен x_j, а весовой коэффициент для связи, соединяющей узел **j** скрытого слоя с выходным узлом **k**, равен $w_{j,k}$. Необычный символ Σ_a^b означает суммирование следующего за ним выражения по всем значениям между **a** и **b**.

$$o_k = \frac{1}{1 + e^{-\sum_{j=1}^{3}\left(w_{j,k} \cdot \frac{1}{1+e^{-\sum_{i=1}^{3}(w_{i,j} \cdot x_i)}}\right)}}$$



Ничего себе! Лучше держаться от этого подальше.

А не могли бы мы вместо того, чтобы пытаться выглядеть очень умными, просто перебирать случайные сочетания весовых коэффициентов, пока не будет получен устраивающий нас результат?

Этот вопрос не столь уж наивен, как могло бы показаться, особенно когда задача действительно трудная. Такой подход называется **методом грубой силы**. Некоторые люди пытаются использовать методы грубой силы для того, чтобы взламывать пароли, и это может сработать, если паролем является какое-либо осмысленное слово, причем не очень длинное, а не просто набор символов. Такая задача вполне по силам достаточно мощному домашнему компьютеру. Но представьте, что каждый весовой коэффициент может иметь 1000 возможных значений в диапазоне от −1 до +1, например 0,501, −0,203 или 0,999. Тогда в случае нейронной сети с тремя слоями

по три узла, насчитывающей 18 весовых коэффициентов, мы должны были бы протестировать 18 тысяч возможностей. Если бы у нас была более типичная нейронная сеть с 500 узлами в каждом слое, то нам пришлось бы протестировать 500 миллионов различных значений весов. Если бы для расчета каждого набора комбинаций требовалась одна секунда, то для обновления весов с помощью всего лишь одного тренировочного примера понадобилось бы примерно 16 лет. Тысяча тренировочных примеров — и мы имели бы 16 тысяч лет!

Как видите, подход, основанный на методе грубой силы, практически нереализуем. В действительности по мере добавления в сеть новых слоев, узлов или вариантов перебора весов ситуация очень быстро ухудшается.

Эта головоломка не поддавалась математикам на протяжении многих лет и получила реальное практическое разрешение лишь в 1960–1970-х годах. Существуют различные мнения относительно того, кто сделал это впервые или совершил ключевой прорыв, но для нас важно лишь то, что это запоздалое открытие послужило толчком к взрывному развитию современной теории нейронных сетей, которая сейчас способна решать некоторые весьма впечатляющие задачи.

Но все же, как нам разрешить эту явно трудную проблему? Хотите верьте, хотите нет, но вы уже владеете средствами, с помощью которых сможете справиться с этим самостоятельно. Все, что вам для этого нужно знать, мы уже обсуждали. Теперь можем продолжить.

Самое главное, что требуется от нас, — это не бояться пессимизма.

Математические выражения, позволяющие определить выходной сигнал нейронной сети при известных весовых коэффициентах, необычайно сложны, и в них нелегко разобраться. Количество различных комбинаций слишком велико для того, чтобы пытаться тестировать их поочередно.

Для пессимизма есть и ряд других причин. Тренировочных данных может оказаться недостаточно для эффективного обучения сети. В тренировочных данных могут быть ошибки, в связи с чем справедливость нашего предположения о том, что они истинны и на них можно учиться, оказывается под вопросом. Количество слоев или узлов в самой сети может быть недостаточным для того, чтобы правильно моделировать решение задачи.

Это означает, что предпринимаемый нами подход должен быть реалистичным и учитывать указанные ограничения. Если мы будем

следовать этим принципам, то, возможно, найдем решение, которое, даже не будучи идеальным с математической точки зрения, даст нам лучшие результаты, поскольку не будет основано на ложных идеалистических допущениях.

Проиллюстрируем суть наших рассуждений на следующем примере. Представьте себе ландшафт с очень сложным рельефом, имеющим возвышения и впадины, а также холмы с предательскими буграми и ямами. Вокруг так темно, что ни зги не видно. Вы знаете, что находитесь на склоне холма, и вам нужно добраться до его подножия. Точной карты местности у вас нет. Но у вас есть фонарь. Что вы будете делать? Пожалуй, вы воспользуетесь фонарем и осмотритесь вокруг себя. Света фонаря не хватит для дальнего обзора, и вы наверняка не сможете осмотреть весь ландшафт целиком. Но вы сможете увидеть, по какому участку проще всего начать спуск к подножию холма, и сделаете несколько небольших шагов в этом направлении. Действуя подобным образом, вы будете медленно, шаг за шагом, продвигаться вниз, не располагая общей картой и заблаговременно проложенным маршрутом.

Математическая версия этого подхода называется методом **градиентного спуска**, и причину этого нетрудно понять. После того как вы сделали шаг в выбранном направлении, вы вновь осматриваетесь, чтобы увидеть, какой путь ведет вас к цели, и делаете очередной шаг в этом направлении. Вы продолжаете действовать точно так же до тех пор, пока благополучно не спуститесь к подножию холма.

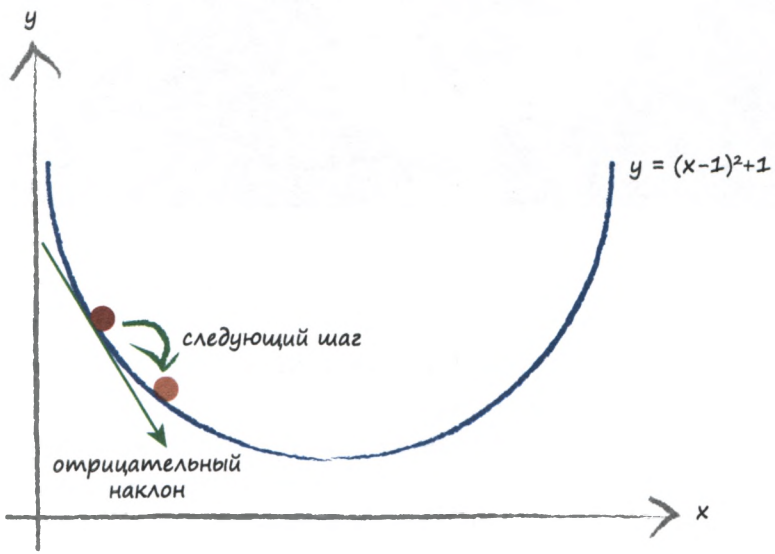
А теперь представьте, что этим сложным ландшафтом является математическая функция. Метод градиентного спуска позволяет

находить минимум, даже не располагая знаниями свойств этой функции, достаточными для нахождения минимума математическими методами. Если функция настолько сложна, что простого способа нахождения минимума алгебраическими методами не существует, то мы можем вместо этого применить метод градиентного спуска. Ясное дело, он может не дать нам точный ответ, поскольку мы приближаемся к ответу шаг за шагом, постепенно улучшая нашу позицию. Но это лучше, чем вообще не иметь никакого ответа. Во всяком случае, мы можем продолжить уточнение ответа еще более мелкими шагами по направлению к минимуму, пока не достигнем желаемой точности.

А какое отношение имеет этот действительно эффективный метод градиентного спуска к нейронным сетям? Если упомянутой сложной функцией является ошибка сети, то спуск по склону для нахождения минимума означает, что мы минимизируем ошибку. Мы улучшаем выходной сигнал сети. Это именно то, чего мы хотим!

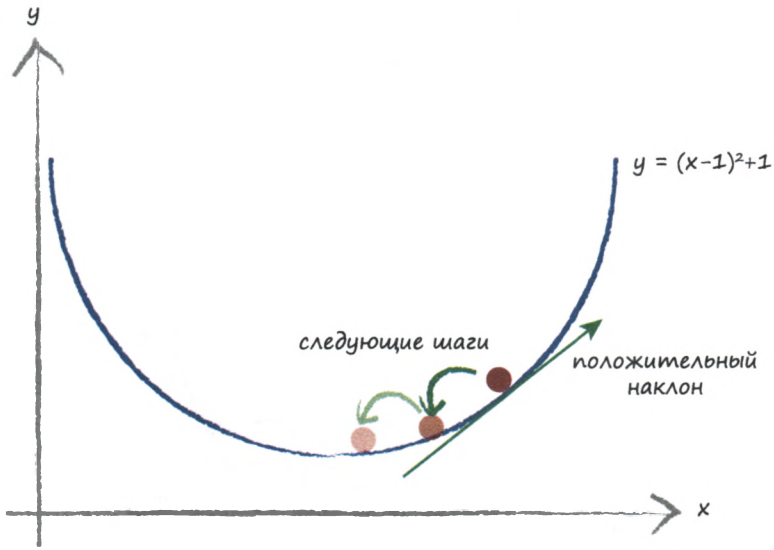
Чтобы вы все наглядно усвоили, рассмотрим использование метода градиентного спуска на простейшем примере.

Ниже приведен график простой функции $y = (x - 1)^2 + 1$. Если бы это была функция, описывающая ошибку, то мы должны были бы найти значение **x**, которое минимизирует эту функцию. Представим на минуту, что мы имеем дело не со столь простой функцией, а с гораздо более сложной.



Градиентный спуск должен с чего-то начинаться. На графике показана случайно выбранная начальная точка. Подобно покорителю гор, мы исследуем место, в котором находимся, и видим, в каком направлении идет спуск. Наклон кривой также обозначен на графике, и в данном случае ему соответствует отрицательный градиент. Мы хотим следовать в направлении вниз, поэтому движемся вдоль оси x вправо. Таким образом, мы немного увеличиваем **x**. Это первый шаг нашего альпиниста. Вы видите, что мы улучшили нашу позицию и продвинулись ближе к фактическому минимуму.

Представим, что мы начали спуск с какого-то другого места, как показано на следующем графике.

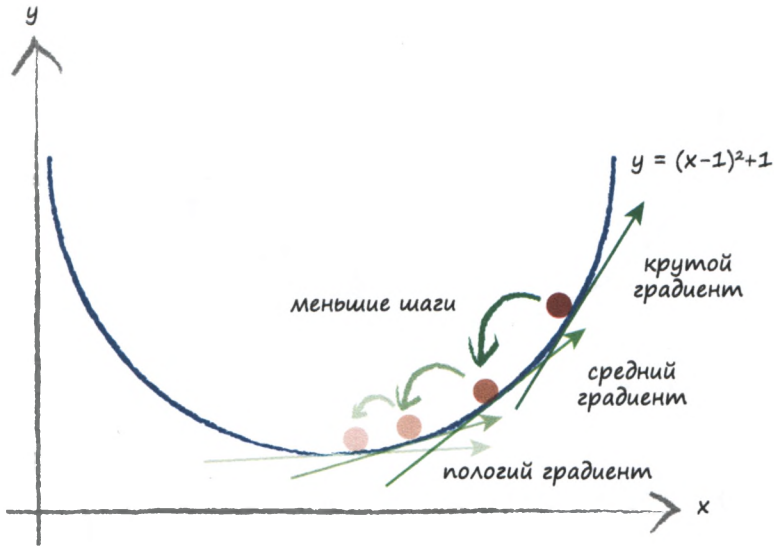


На этот раз наклон почвы под нашими ногами положителен, поэтому мы движемся влево, т.е. немного уменьшаем **x**. И вновь вы видите, что мы улучшили наше положение, приблизившись к фактическому минимуму. Мы можем продолжать действовать в том же духе до тех пор, пока изменения не станут настолько малыми, что мы будем считать, что достигли минимума.

Необходимым усовершенствованием этого метода должно быть изменение величины шагов во избежание перескока через минимум, что приведет к бесконечным прыжкам вокруг него. Вообразите, что мы оказались на расстоянии 0,5 метра от истинного минимума, но

длина нашего шага всегда равна 2 метра. Тогда мы будем постоянно пропускать минимум, поскольку на каждом шаге в его направлении мы будем перескакивать через него. Если мы будем уменьшать величину шага пропорционально величине градиента, то по мере приближения к минимуму будем совершать все более мелкие шаги. При этом мы предполагаем, что чем ближе к минимуму, тем меньше наклон. Для большинства гладких (непрерывно дифференцируемых) функций такое предположение вполне приемлемо. Оно не будет справедливым лишь по отношению к экзотическим зигзагообразным функциям со взлетами и провалами в точках, которые математики называют **точками разрыва**.

Идея уменьшения величины шага по мере уменьшения величины градиента, являющейся хорошим индикатором близости к минимуму, иллюстрируется следующим графиком.

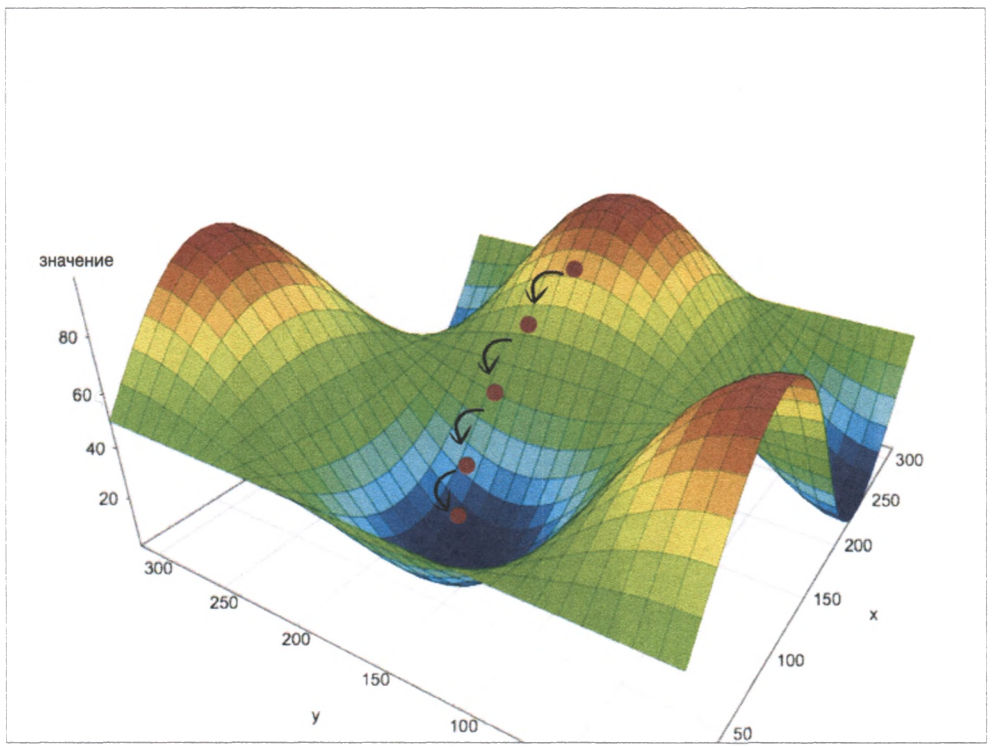


Кстати, обратили ли вы внимание на то, что мы изменяем **x** в направлении, противоположном направлению градиента? Положительный градиент означает, что мы должны уменьшить **x**. Отрицательный градиент требует увеличения **x**. Все это отчетливо видно на графике, но это легко упустить из виду и пойти совершенно неправильным путем.

Используя метод градиентного спуска, мы не пытались находить истинный минимум алгебраическим методом, поскольку сделали вид, будто функция $y = (x - 1)^2 + 1$ для этого слишком сложна. Даже если бы мы не могли определить наклон кривой с математической точностью, мы могли бы оценить его, и, как нетрудно заметить, это все равно вело бы нас в правильном общем направлении.

Вся мощь этого метода по-настоящему проявляется в случае функций, зависящих от многих параметров. Например, вместо зависимости **y** от **x** мы можем иметь зависимость от **a**, **b**, **c**, **d**, **e** и **f**. Вспомните, что функция выходного сигнала, а вместе с ней и функция ошибки зависят от множества весовых коэффициентов, которые часто исчисляются сотнями!

Следующий график вновь иллюстрирует метод градиентного спуска, но на этот раз применительно к более сложной функции, зависящей от двух параметров. График такой функции можно представить в трех измерениях, где высота представляет значение функции.



Возможно, глядя на эту трехмерную поверхность, вы задумались над тем, может ли метод градиентного спуска привести в другую долину, которая расположена справа. В действительности этот вопрос можно обобщить: не приводит ли иногда метод градиентного спуска в ложную долину, поскольку некоторые сложные функции имеют множество долин?

Что такое ложная долина? Это долина, которая не является самой глубокой. Тогда на поставленный вопрос следует дать утвердительный ответ: да, такое может происходить.

На следующей иллюстрации показаны три варианта градиентного спуска, один из которых приводит к ложному минимуму.

Давайте немного передохнем и соберемся с мыслями.

Резюме

- **Метод градиентного спуска** — это действительно хороший способ нахождения минимума функции, и он прекрасно работает, когда функция настолько сложна, что ее математическая обработка алгебраическими методами сопряжена с большими трудностями.
- Более того, этот метод хорошо работает в случае функций многих переменных, когда другие методы не срабатывают либо их невозможно реализовать на практике.
- Данный метод также **устойчив** к наличию дефектных данных и не заведет вас далеко в неправильную сторону, если функция не описывается идеально или же время от времени мы совершаем неверные шаги.

Выходной сигнал нейронной сети представляет собой сложную, трудно поддающуюся описанию функцию со многими параметрами, весовыми коэффициентами связей, которые влияют на выходной

сигнал. Так можем ли мы использовать метод градиентного спуска для определения подходящих значений весов? Можем, если правильно выберем функцию ошибки.

Функция выходного сигнала сама по себе не является функцией ошибки. Но мы знаем, что можем легко превратить ее в таковую, поскольку ошибка — это разность между целевыми тренировочными значениями и фактическими выходными значениями.

Однако здесь есть кое-что, чего следует остерегаться. Взгляните на приведенную ниже таблицу с тренировочными данными и фактическими значениями для трех выходных узлов вместе с кандидатами на роль функции ошибок.

Выход сети	Целевой результат	Ошибка (целевое–фактическое)	Ошибка |целевое–фактическое|	Ошибка (целевое–фактическое)^2
0,4	0,5	0,1	0,1	0,01
0,8	0,7	–0,1	0,1	0,01
1,0	1,0	0	0	0
Сумма		0	0,2	0,02

Нашим первым кандидатом на роль функции ошибки является простая разность значений **(целевое – фактическое)**. Это кажется вполне разумным, не так ли? Но если вы решите использовать сумму ошибок по всем узлам в качестве общего показателя того, насколько хорошо обучена сеть, то эта сумма равна нулю!

Что случилось? Ясно, что сеть еще недостаточно натренирована, поскольку выходные значения двух узлов отличаются от целевых значений. Но нулевая сумма означает отсутствие ошибки. Это объясняется тем, что положительная и отрицательная ошибки взаимно сократились. Отсюда следует, что простая разность значений, даже если бы их взаимное сокращение было неполным, не годится для использования в качестве меры величины ошибки.

Пойдем другим путем, взяв **абсолютную** величину разности. Формально это записывается как **|целевое–фактическое|** и означает, что знак результата вычитания игнорируется. Это могло бы сработать, поскольку в данном случае ничто ни с чем не может сократиться. Причина, по которой данный метод не получил популярности,

связана с тем, что при этом наклон не является непрерывной функцией вблизи минимума, что затрудняет использование метода градиентного спуска, поскольку мы будем постоянно совершать скачки вокруг V-образной долины, характерной для функции ошибки подобного рода. При приближении к минимуму наклон, а вместе с ним и величина шага изменения переменной, не уменьшается, а это означает риск перескока.

Третий вариант заключается в том, чтобы использовать в качестве меры ошибки квадрат разности: **(целевое–фактическое)**^{2}. Существует несколько причин, по которым третий вариант предпочтительнее второго, включая следующие:

- он упрощает вычисления, с помощью которых определяется величина наклона для метода градиентного спуска;
- функция ошибки является непрерывно гладкой, что обеспечивает нормальную работу метода градиентного спуска ввиду отсутствия провалов и скачков значений функции;
- по мере приближения к минимуму градиент уменьшается, что означает снижение риска перескока через минимум, если используется уменьшение величины шагов.

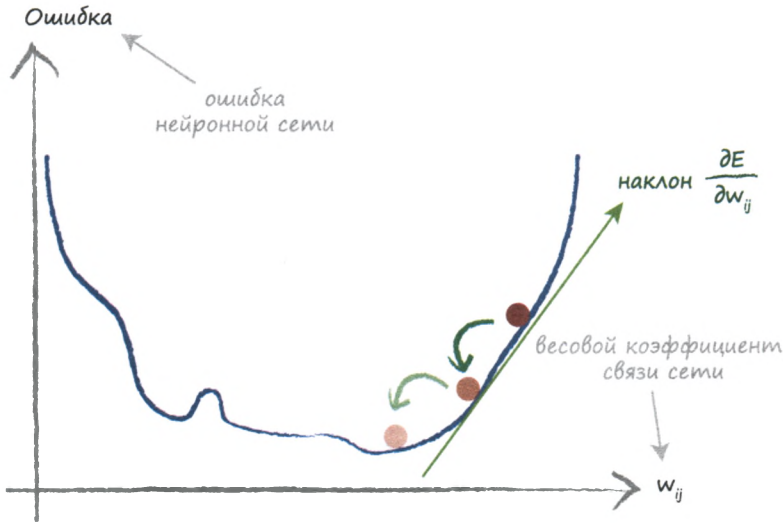
А возможны ли другие варианты? Да, вы вправе сконструировать любую сложную функцию, которую считаете нужной. Одни из них могут вообще не работать, другие могут хорошо работать для определенного круга задач, а третьи могут работать действительно хорошо, но их чрезмерная сложность приводит к неоправданным затратам ресурсов.

А тем временем мы вышли на финишную прямую!

Чтобы воспользоваться методом градиентного спуска, нам нужно определить наклон функции ошибки по отношению к весовым коэффициентам. Это требует применения **дифференциального исчисления**. Возможно, вы уже знакомы с ним, а если не знакомы или вам требуется освежить свою память, то обратитесь к приложению А. Дифференциальное исчисление — это просто математически строгий подход к определению величины изменения одних величин при изменении других. Например, оно позволяет ответить на вопрос о том, как изменяется длина пружины в зависимости от величины усилия,

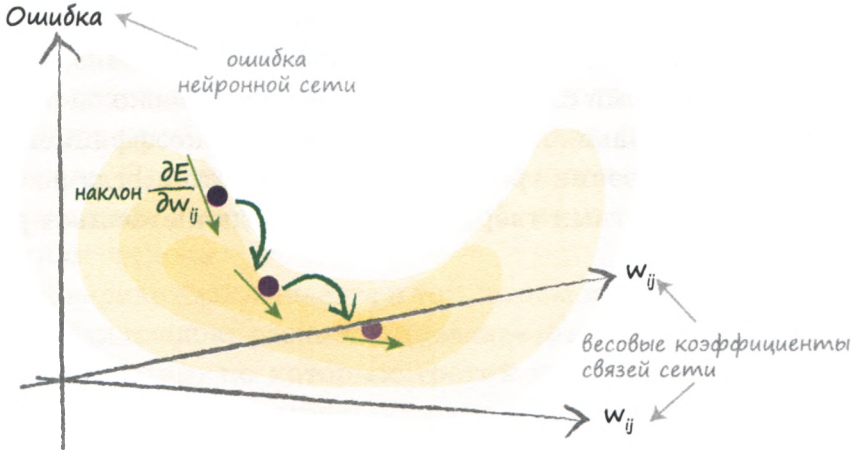
приложенного к ее концам. В данном случае нас интересует зависимость функции ошибки от весовых коэффициентов связей внутри нейронной сети. Иными словами, нас интересует, насколько величина ошибки чувствительна к изменениям весовых коэффициентов.

Начнем с рассмотрения графика, поскольку это всегда позволяет почувствовать под ногами твердую почву, когда пытаешься решать трудную задачу.



Этот график в точности повторяет один из тех, которые приводились ранее, чтобы подчеркнуть, что мы не делаем ничего принципиально нового. На этот раз функцией, которую мы пытаемся минимизировать, является ошибка на выходе нейронной сети. В этом простом примере показан лишь один весовой коэффициент, но мы знаем, что в нейронных сетях их будет намного больше.

На следующей диаграмме отображаются два весовых коэффициента, и поэтому функция ошибки графически отображается в виде трехмерной поверхности, высота расположения точек которой изменяется с изменением весовых коэффициентов связей. Как видите, теперь процесс минимизации ошибки больше напоминает спуск в долину по рельефной местности.



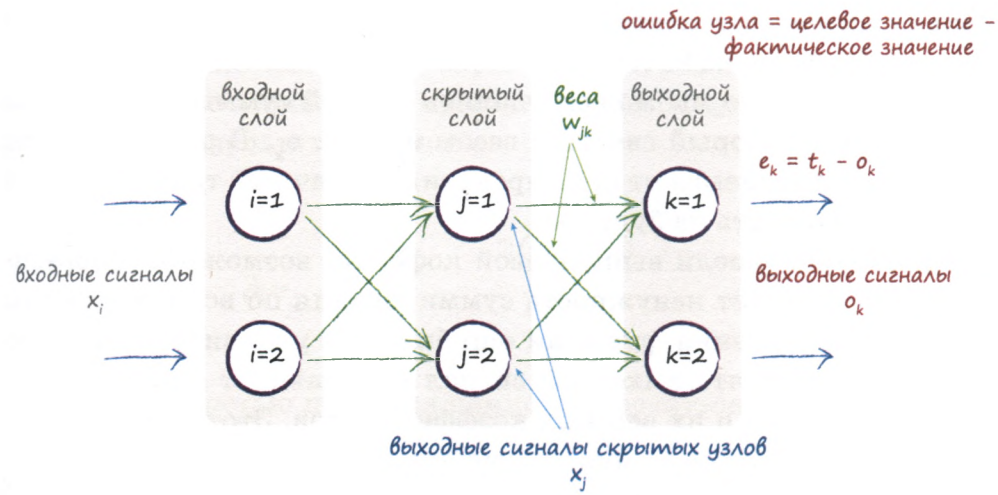
Визуализировать многомерную поверхность ошибки как функции намного большего количества параметров значительно труднее, но идея нахождения минимума методом градиентного спуска остается той же.

Сформулируем на языке математики, чего мы хотим:

$$\frac{\partial E}{\partial w_{jk}}$$

Это выражение представляет изменение ошибки **E** при изменении веса w_{jk}. Это и есть тот наклон функции ошибки, который нам нужно знать, чтобы начать градиентный спуск к минимуму.

Прежде чем мы развернем это выражение, временно сосредоточим внимание только на весовых коэффициентах связей между скрытым слоем и последним выходным слоем. Интересующая нас область выделена на приведенной ниже диаграмме. К связям между входным и скрытым слоями мы вернемся позже.



Мы будем постоянно ссылаться на эту диаграмму, дабы в процессе вычислений не забыть о том, что в действительности означает каждый символ. Пусть вас это не смущает, поскольку шаги вычислительной процедуры не являются сложными и будут объясняться, а все необходимые понятия ранее уже обсуждались.

Прежде всего запишем в явном виде функцию ошибки, которая представляет собой сумму возведенных в квадрат разностей между целевым и фактическим значениями, где суммирование осуществляется по всем **n** выходным узлам.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} \Sigma_n (t_n - o_n)^2$$

Здесь мы всего лишь записали, что на самом деле представляет собой функция ошибки **E**.

Мы можем сразу же упростить это выражение, заметив, что выходной сигнал o_n на узле **n** зависит лишь от связей, которые с ним соединены. Для узла **k** это означает, что выходной сигнал o_k зависит лишь от весов w_{jk}, поскольку эти веса относятся к связям, ведущим к узлу **k**.

Это можно рассматривать еще и как то, что выходной сигнал узла **k** не зависит от весов w_{jb}, где **b** не равно **k**, поскольку связь между

этими узлами отсутствует. Вес w_{jb} относится к связи, ведущей к выходному узлу **b**, но не **k**.

Это означает, что мы можем удалить из этой суммы все сигналы o_n кроме того, который связан с весом w_{jk}, т.е. o_k. В результате мы полностью избавляемся от суммирования! Отличный трюк, который вам стоит запомнить на будущее.

Если вы уже успели выпить свой кофе, то, возможно, сообразили, что это означает ненужность суммирования по всем выходным узлам для нахождения функции ошибки. Мы уже видели, чем это объясняется: тем, что выходной сигнал узла зависит лишь от ведущих к нему связей и их весовых коэффициентов. Этот момент часто остается нераскрытым во многих учебниках, которые просто приводят выражение для функции без каких-либо пояснений.

Как бы то ни было, теперь мы имеем более простое выражение:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}} (t_k - o_k)^2$$

А сейчас мы используем некоторые средства дифференциального исчисления. Помните, что при необходимости восстановить в памяти необходимые знания вы всегда можете заглянуть в приложение А.

Член t_k — константа и поэтому не изменяется при изменении w_{jk}, т.е. не является функцией w_{jk}. Было бы очень странно, если бы истинные примеры, предоставляющие целевые значения, изменялись в зависимости от весовых коэффициентов! В результате у нас остается член o_k, который, как мы знаем, зависит от w_{jk}, поскольку весовые коэффициенты влияют на распространение в прямом направлении сигналов, которые затем превращаются в выходные сигналы o_k.

Чтобы разбить эту задачу дифференцирования на более простые части, мы воспользуемся цепным правилом (правило дифференцирования сложных функций). Прочитать о нем можно в приложении А.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} \cdot \frac{\partial o_k}{\partial w_{jk}}$$

Теперь мы можем разделаться с каждой из этих частей по отдельности. С первой частью мы справимся легко, поскольку для этого нужно всего лишь взять простую производную от квадратичной функции. В результате получаем:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial o_k}{\partial w_{jk}}$$

Со второй частью придется немного повозиться, но и это не вызовет больших затруднений. Здесь o_k — это выходной сигнал узла **k**, который, как вы помните, получается в результате применения сигмоиды к сигналам, поступающим на данный узел. Для большей ясности запишем это в явном виде:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \frac{\partial}{\partial w_{jk}} \text{сигмоида}\left(\Sigma_j w_{jk} \cdot o_j\right)$$

Здесь o_j — выходной сигнал узла предыдущего скрытого слоя, а не выходной сигнал узла последнего слоя.

Как продифференцировать сигмоиду? Мы могли бы это сделать самостоятельно, проведя сложные и трудоемкие вычисления в соответствии с изложенными в приложении А фундаментальными идеями, однако эта работа уже проделана другими людьми. Поэтому мы просто воспользуемся уже известным ответом, как это ежедневно делают математики по всему миру.

$$\frac{\partial}{\partial x} \text{сигмоида}(x) = \text{сигмоида}(x)\left(1 - \text{сигмоида}(x)\right)$$

Дифференцирование некоторых функций приводит к выражениям устрашающего вида. В случае же сигмоиды результат получается очень простым. Это одна из причин широкого применения сигмоиды в качестве функции активации в нейронных сетях.

Используя этот впечатляющий результат, получаем следующее выражение:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) \cdot \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)) \cdot \frac{\partial}{\partial w_{jk}}(\Sigma_j w_{jk} \cdot o_j)$$

$$= -2(t_k - o_k) \cdot \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

А откуда взялся последний сомножитель? Это результат применения цепного правила к производной сигмоиды, поскольку выражение под знаком функции `сигмоида()` также должно быть продифференцировано по переменной w_{jk}. Это делается очень просто и дает в результате o_j.

Прежде чем записать окончательный ответ, избавимся от множителя 2 в начале выражения. Мы вправе это сделать, поскольку нас интересует только направление градиента функции ошибки, так что этот множитель можно безболезненно отбросить. Нам совершенно безразлично, какой множитель будет стоять в начале этого выражения, 2, 3 или даже 100, коль скоро мы всегда будем его игнорировать. Поэтому для простоты избавимся от него.

Окончательное выражение, которое мы будем использовать для изменения веса w_{jk}, выглядит так.

$$\boxed{\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j}$$

Ух ты! У нас все получилось!

Это и есть то магическое выражение, которое мы искали. Оно является ключом к тренировке нейронных сетей.

Проанализируем вкратце это выражение, отдельные части которого выделены цветом. Первая часть, с которой вы уже хорошо знакомы, — это ошибка (целевое значение минус фактическое значение).

Сумма, являющаяся аргументом сигмоиды, — это сигнал, поступающий на узел выходного слоя, и для упрощения вида выражения мы могли бы обозначить этот сигнал просто как i_k. Он выступает в качестве входного сигнала узла **k** до применения функции активации. Последняя часть — это выходной сигнал узла **j** предыдущего скрытого слоя. Рассмотрение полученного выражения именно в таком ракурсе позволяет лучше понять связь физической картины происходящего с наклоном функции и в конечном счете с уточнением весовых коэффициентов.

Это поистине фантастический результат, и мы можем заслуженно гордиться собой. Путь к этому результату многим людям дается с большим трудом.

Нам осталось сделать совсем немного, и мы достигнем цели. Выражение, с которым мы совладали, предназначено для уточнения весовых коэффициентов связей между скрытым и выходным слоями. Мы должны завершить работу и найти наклон аналогичной функции ошибки для коэффициентов связей между входным и скрытым слоями.

Можно было бы вновь произвести полностью все математические выкладки, но мы не будем этого делать. Мы воспользуемся описанной перед этим физической интерпретацией составляющих выражения для производной и реконструируем его для интересующего нас нового набора коэффициентов. При этом необходимо учесть следующие изменения.

- Первая часть выражения для производной, которая ранее была выходной ошибкой (целевое значение минус фактическое значение), теперь становится рекомбинированной выходной ошибкой скрытых узлов, рассчитываемой в соответствии с механизмом обратного распространения ошибок, с чем вы уже знакомы. Назовем ее e_j.
- Вторая часть выражения, включающая сигмоиду, остается той же, но выражение с суммой, передаваемое функции, теперь относится к предыдущим слоям, и поэтому суммирование осуществляется по всем входным сигналам скрытого слоя, сглаженным весами связей, ведущих к скрытому узлу **j**. Мы могли бы назвать эту сумму i_j.

- Последняя часть выражения приобретает смысл выходных сигналов o_i узлов первого слоя, и в данном случае эти сигналы являются входными.

Тем самым нам удалось изящно избежать излишних трудоемких вычислений, в полной мере воспользовавшись всеми преимуществами симметрии задачи для конструирования нового выражения. Несмотря на всю ее простоту, это очень мощная методика, взятая на вооружение выдающимися математиками и учеными. Овладев ею, вы, несомненно, произведете на коллег большое впечатление!

Итак, вторая часть окончательного ответа, к получению которого мы стремимся (градиент функции ошибки по весовым коэффициентам связей между входным и скрытым слоями), приобретает следующий вид:

$$\frac{\partial E}{\partial w_{ij}} = -(e_j) \cdot \text{сигмоида}(\Sigma_i w_{ij} \cdot o_i)(1 - \text{сигмоида}(\Sigma_i w_{ij} \cdot o_i)) \cdot o_i$$

На данном этапе нами получены все ключевые магические выражения, необходимые для вычисления искомого градиента, который мы используем для обновления весовых коэффициентов по результатам обучения на каждом тренировочном примере, чем мы сейчас и займемся.

Не забывайте о том, что направление изменения коэффициентов противоположно направлению градиента, что неоднократно демонстрировалось на предыдущих диаграммах. Кроме того, мы сглаживаем интересующие нас изменения параметров посредством коэффициента обучения, который можно настраивать с учетом особенностей конкретной задачи. С этим подходом вы также уже сталкивались, когда при разработке линейных классификаторов мы использовали его для уменьшения негативного влияния неудачных примеров на эффективность обучения, а при минимизации функции ошибки — для того, чтобы избежать постоянных перескоков через минимум. Выразим это на языке математики:

$$\text{новый } w_{jk} = \text{старый } w_{jk} - \alpha \cdot \frac{\partial E}{\partial w_{jk}}$$

Обновленный вес w_{jk} — это старый вес с учетом отрицательной поправки, величина которой пропорциональна производной функции ошибки. Поправка записана со знаком “минус”, поскольку мы хотим, чтобы вес увеличивался при отрицательной производной и уменьшался при положительной, о чем ранее уже говорилось. Символ α (альфа) — это множитель, сглаживающий величину изменений во избежание перескоков через минимум. Этот коэффициент часто называют **коэффициентом обучения**.

Данное выражение применяется к весовым коэффициентам связей не только между скрытым и выходным, но и между входным и скрытым слоями. Эти два случая различаются градиентами функции ошибки, выражения для которых приводились выше.

Прежде чем закончить с этим примером, посмотрим, как будут выглядеть те же вычисления в матричной записи. Для этого сделаем то, что уже делали раньше, — запишем, что собой представляет каждый элемент матрицы изменений весов.

$$\begin{pmatrix} \Delta w_{1,1} & \Delta w_{2,1} & \Delta w_{3,1} & \dots \\ \Delta w_{1,2} & \Delta w_{2,2} & \Delta w_{3,2} & \dots \\ \Delta w_{1,3} & \Delta w_{2,3} & \Delta w_{j,k} & \dots \\ \dots & \dots & \dots & \dots \end{pmatrix} = \begin{pmatrix} E_1 * S_1 (1-S_1) \\ E_2 * S_2 (1-S_2) \\ E_k * S_k (1-S_k) \\ \dots \end{pmatrix} \cdot \begin{pmatrix} O_1 & O_2 & O_j & \dots \end{pmatrix}$$

значения из следующего слоя

значения из предыдущего слоя

Я опустил коэффициент обучения α, поскольку это всего лишь константа, которая никак не влияет на то, как мы организуем матричное умножение.

Матрица изменений весов содержит значения поправок к весовым коэффициентам w_{jk} для связей между узлом j одного слоя и узлом k следующего слоя. Вы видите, что в первой части выражения справа

от знака равенства используются значения из следующего слоя (узел **k**), а во второй — из предыдущего слоя (узел **j**).

Возможно, глядя на приведенную выше формулу, вы заметили, что горизонтальная матрица, представленная одной строкой, — это транспонированная матрица сигналов o_j на выходе предыдущего слоя. Цветовое выделение элементов матриц поможет вам понять, что скалярное произведение матриц отлично работает и в этом случае.

Используя символическую запись матриц, мы можем привести эту формулу к следующему виду, хорошо приспособленному для реализации в программном коде на языке, обеспечивающем эффективную работу с матрицами.

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

Фактически это выражение совсем не сложное. Сигмоиды исчезли из поля зрения, поскольку они скрыты в матрицах выходных сигналов o_k узлов.

Вот и все! Работа сделана.

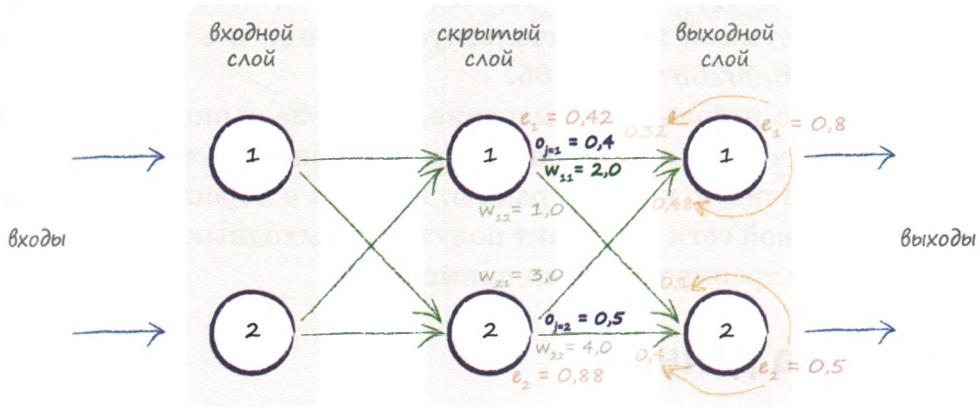
Резюме

- Ошибка нейронной сети является функцией весов внутренних связей.
- Улучшение нейронной сети означает уменьшение этой ошибки посредством изменения указанных весов.
- Непосредственный подбор подходящих весов наталкивается на значительные трудности. Альтернативный подход заключается в итеративном улучшении весовых коэффициентов путем уменьшения функции ошибки небольшими шагами. Каждый шаг совершается в направлении скорейшего спуска из текущей позиции. Этот подход называется **градиентным спуском**.
- Градиент ошибки можно без особых трудностей рассчитать, используя дифференциальное исчисление.

Пример обновления весовых коэффициентов

Проиллюстрируем применение описанного метода обновления весовых коэффициентов на конкретном примере с использованием числовых данных.

С представленной ниже сетью мы уже работали, но в этот раз будут использованы заданные значения выходных сигналов первого и второго узлов скрытого слоя, $o_{j=1}$ и $o_{j=2}$. Эти значения лишь иллюстрируют применение методики и выбраны произвольно, а не вычислены, как это следовало бы сделать, по известным выходным сигналам входного слоя.



Мы хотим обновить весовой коэффициент w_{11} для связи между скрытым и выходным слоями, текущее значение которого равно 2,0.

Запишем еще раз выражение для градиента ошибки.

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) \cdot \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)(1 - \text{сигмоида}(\Sigma_j w_{jk} \cdot o_j)) \cdot o_j$$

Разберем это выражение по частям.

- Первая часть $(t_k - o_k)$ — это уже известная вам по предыдущим диаграммам ошибка e_1=0,8.

- Сумма $\Sigma_j w_{jk} o_j$, передаваемая сигмоидам, равна (2,0 * 0,4) + (3,0 * * 0,5) = 2,3.
- Тогда сигмоида $1/(1 + e^{-2,3})$ равна 0,909. Следовательно, промежуточное выражение равно 0,909 * (1 – 0,909) = 0,083.
- Последняя часть — это просто сигнал o_j, которым в данном случае является сигнал o_{j-1}, так как нас интересует вес w_{11}, где j=1. Поэтому данная часть просто равна 0,4.

Перемножив все три части этого выражения и не забыв при этом о начальном знаке "минус", получаем значение –0,0265.

При коэффициенте обучения, равном 0,1, изменение веса составит –0,1 * (–0,0265) = +0,002650. Следовательно, новое значение w_{11}, определяемое суммой первоначального значения и его изменения, составит 2,0 + 0,00265 = 2,00265.

Это довольно небольшое изменение, но после выполнения сотен или даже тысяч итераций весовые коэффициенты в конечном счете образуют устойчивую конфигурацию, которая в хорошо натренированной нейронной сети обеспечит получение выходных сигналов, согласующихся с тренировочными примерами.

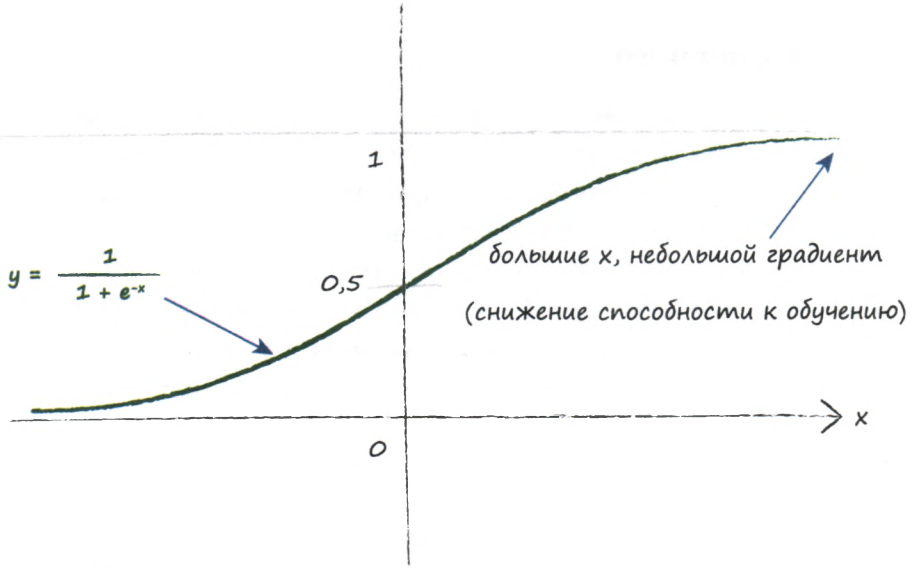
Подготовка данных

В этом разделе мы обсудим, как лучше всего подготовить тренировочные данные и случайные начальные значения весовых коэффициентов и даже спланировать выходные значения таким образом, чтобы процесс обучения имел все шансы на успех.

Совершенно верно, вы все правильно прочитали! Не все попытки использования нейронных сетей заканчиваются успехом, и на то есть множество причин. Некоторые из них можно устранить за счет продуманного отбора тренировочных данных и начальных значений весовых коэффициентов, а также тщательного планирования схемы выходных сигналов. Рассмотрим все эти факторы поочередно.

Входные значения

Взгляните на приведенный ниже график сигмоиды. Как нетрудно заметить, при больших значениях входного сигнала кривая функции заметно спрямляется.



Значительное спрямление функции активации служит источником проблем, поскольку для усваивания сетью новых значений весов используется градиент. Посмотрите еще раз на выражение для изменений весовых коэффициентов. Оно зависит от градиента функции активации. Использование небольших значений градиента равносильно ограничению способности сети к обучению. Это называется **насыщением** нейронной сети. Отсюда следует, что для входных сигналов лучше задавать небольшие значения.

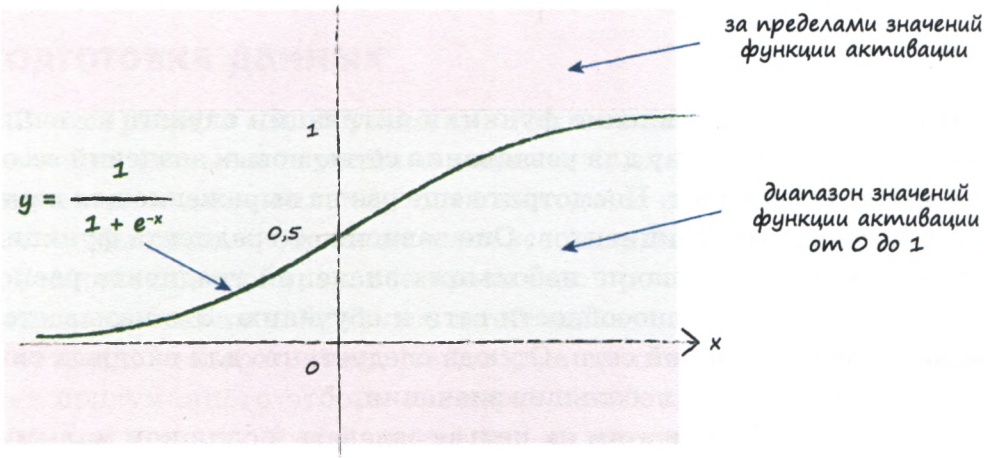
Любопытно, что при этом их нельзя задавать и слишком малыми, поскольку они тоже входят в указанное выражение для изменений весовых коэффициентов. Слишком малые значения входных сигналов могут быть проблематичными еще и потому, что при обработке очень больших или очень малых значений точность компьютерных вычислений значительно снижается.

Неплохим решением этой проблемы является масштабирование входных сигналов до значений в интервале от 0,0 до 1,0. Иногда для входных сигналов вводят небольшое смещение, скажем, 0,01, с целью недопущения нулевых входных сигналов, которые неприятны тем, что при $o_j=0$ выражение для поправки к весам обнуляется, тем самым лишая сеть способности к обучению.

Выходные значения

Выходные значения нейронной сети — это сигналы, появляющиеся на узлах последнего слоя. Если мы используем функцию активации, которая не обеспечивает получение значений свыше 1,0, то было бы глупо пытаться устанавливать значения большей величины в качестве целевых. Вспомните о том, что логистическая функция не дотягивает даже до значения 1,0 — она только приближается к нему. В математике это называется **асимптотическим** стремлением к 1,0.

Следующий график наглядно демонстрирует, почему логистическая функция активации не может обеспечить получение значений, превышающих 1,0 или меньших нуля.



Если мы все же установим целевые значения в этих недостижимых, запрещенных диапазонах, то тренировка сети приведет к еще бо́льшим весовым коэффициентам в попытке добиться все бо́льших и бо́льших значений выходных сигналов, которые фактически никогда не могут быть достигнуты вследствие использования

функции активации. Мы понимаем, что это так же плохо, как и насыщение сети.

Поэтому мы должны масштабировать наши целевые выходные значения таким образом, чтобы они были допустимыми при данной функции активации, одновременно заботясь о том, чтобы избежать значений, которые в действительности никогда не могут быть достигнуты.

Общепринято использовать диапазон значений от 0,0 до 1,0, но некоторые разработчики используют диапазон от 0,01 до 0,99, поскольку значения 0,0 и 1,0, с одной стороны, не являются подходящими целевыми значениями, а с другой — могут приводить к чрезмерно большим значениям весовых коэффициентов.

Случайные начальные значения весовых коэффициентов

Здесь применима та же аргументация, что и в случае входных и выходных сигналов. Мы должны избегать больших начальных значений весовых коэффициентов, поскольку использование функции активации в этой области значений может приводить к насыщению сети, о котором мы только что говорили, и снижению способности сети обучаться на лучших значениях.

Один из возможных вариантов — прибегнуть к выбору значений из случайного равномерного распределения чисел в диапазоне от −1,0 до +1,0. Это гораздо лучше, чем использовать, скажем, диапазон чисел от −1000 до +1000.

Возможен ли лучший вариант? Вполне вероятно.

Математики и ученые-компьютерщики разработали подходы, позволяющие определять эмпирические правила для задания случайных начальных значений весовых коэффициентов в зависимости от конкретной конфигурации сети и используемой функции активации. Соответствующие рецепты в высшей степени специфичны, но, невзирая на это, мы рискнем подступиться к ним!

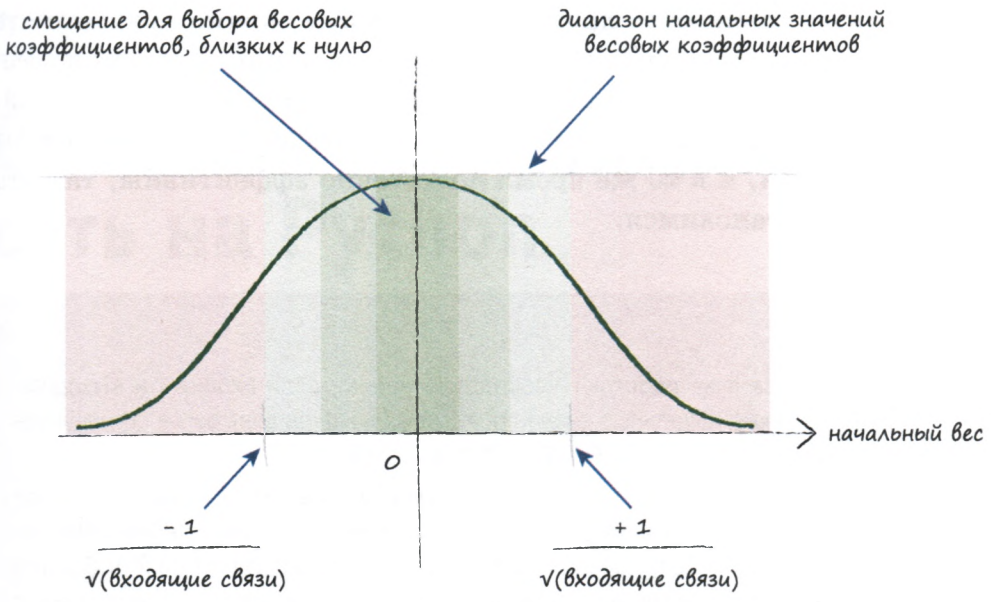
Мы не будем вдаваться в детали математических выкладок, но центральная идея заключается в том, что если на узел нейронной сети поступает множество сигналов, причем поведение этих сигналов хорошо определено, они не достигают слишком больших значений и не распределены каким-то невероятным образом, то весовые коэффициенты не должны нарушать такое состояние

сигналов в процессе их объединения и обработки функцией активации. Иными словами, мы не должны использовать весовые коэффициенты, разрушающие результаты наших попыток тщательно масштабировать входные сигналы. Суть эмпирического правила, к которому пришли математики, заключается в том, что весовые коэффициенты инициализируются числами, случайно выбираемыми из диапазона, грубая оценка которого определяется обратной величиной квадратного корня из количества связей, ведущих к узлу. Таким образом, если к каждому узлу ведут три связи, то начальные значения весов не должны превышать значение $1/(\sqrt{3}) = 0{,}577$. Если же каждый узел имеет 100 входящих связей, то веса должны находиться в диапазоне, ограниченном значением $1/(\sqrt{100}) = 0{,}1$.

Интуитивно это понятно. Некоторые слишком большие веса сместили бы функцию активации в область больших значений, что привело бы к ее **насыщению**. И чем больше связей приходится на узел, тем больше складывается весовых коэффициентов. Поэтому эмпирическое правило, которое уменьшает диапазон значений весовых коэффициентов с увеличением количества связей на узел, находит логическое объяснение.

Если вы уже знакомы с идеей выборочных значений, извлекаемых из распределений вероятностей, то поймете, что это правило фактически относится к нормальному распределению, для которого среднее значение равно нулю, а стандартное отклонение равно обратной величине корня из количества связей, ведущих к узлу. Однако не будем слишком строго придерживаться этой рекомендации, поскольку она предполагает выполнение довольно большого количества условий, которые не всегда соблюдаются, таких как альтернативное использование гиперболического тангенса `th()` в качестве функции активации и специфическое распределение входных сигналов.

Следующий график иллюстрирует в наглядной форме как простой подход, так и более сложный, в котором используется нормальное распределение.



В любом случае никогда не задавайте для начальных весов равные значения, особенно нулевые. Это был бы крайне неудачный вариант!

Этот вариант был бы неудачным по той причине, что в таком случае все узлы сети получили бы одинаковые сигналы, и сигналы на выходе каждого узла были бы одинаковыми. Если затем приступить к обновлению весов с использованием механизма обратного распространения ошибки, то ошибка распределится равномерно. Вы ведь не забыли, что ошибка распределяется между узлами пропорционально весам. Это приведет к одинаковым поправкам для всех весовых коэффициентов, что, в свою очередь, вновь приведет к весам, имеющим одинаковые значения. Подобная симметрия играет крайне отрицательную роль, ведь если правильно натренированная сеть должна иметь неодинаковые значения весовых коэффициентов (что характерно для большинства задач), то вы никогда не достигнете этого состояния.

Еще худший выбор — нулевые значения, поскольку они полностью "убивают" входной сигнал. В этом случае функция обновления весов, которая зависит от входных сигналов, обнуляется, тем самым полностью исключая саму возможность обновления весов.

Существует множество других мер, которые можно предпринять для улучшения процедур подготовки входных данных, задания весовых коэффициентов и организации желаемых выходных значений. С учетом целей книги изложенные идеи достаточно просты, чтобы быть понятными, и в то же время достаточно эффективны, так что на этом мы и остановимся.

Резюме

- Нейронные сети не работают удовлетворительно, если входные и выходные данные, а также начальные значения весовых коэффициентов не согласуются со структурой сети и спецификой конкретной задачи.
- Распространенной проблемой является **насыщение** сети — ситуация, когда большие значения сигналов, часто обусловленные большими значениями весовых коэффициентов, приводят к сигналам, попадающим в область близких к нулю значений градиента функции активации. Результатом этого является снижение способности сети обучаться на лучших значениях весовых коэффициентов.
- Другую проблему представляют **нулевые** значения сигналов или весов. Эти значения также полностью лишают сеть возможности обучаться на лучших значениях весовых коэффициентов.
- Значения весовых коэффициентов внутренних связей должны быть **случайными** и **небольшими**, но не нулевыми. Иногда используют более сложные правила, включающие, например, уменьшение значений весовых коэффициентов с увеличением количества связей, ведущих к узлу.
- **Входные сигналы** должны масштабироваться до небольших, но ненулевых значений. Обычно используют диапазоны значений от 0,01 до 0,99 и от –1.0 до +1,0 в зависимости от того, какой из них лучше соответствует специфике задачи.
- **Выходные сигналы** должны находиться в пределах диапазона, который способна обеспечить функция активации. Значения, меньшие или равные 0 и большие или равные 1, не совместимы с логистической сигмоидой. Установка тренировочных целевых значений за пределами допустимого диапазона приведет к еще большим значениям весов и в конечном счете к насыщению сети. Неплохим диапазоном является диапазон значений от 0,01 до 0,99.

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 много 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.

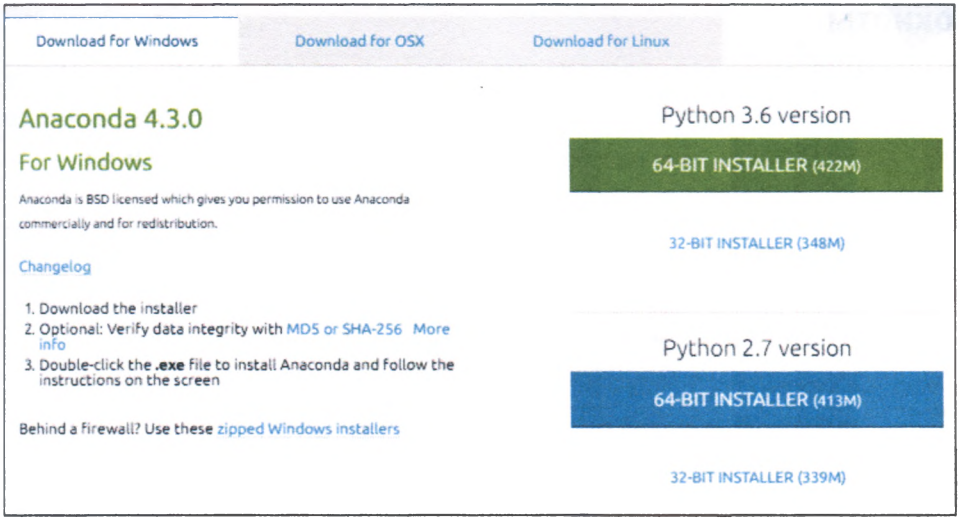
Интерактивный Python = IPython

Мы не будем самостоятельно устанавливать Python вместе со всевозможными 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)[1], 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`, на котором предлагаются различные варианты установки IPython. Я использую пакет **Anaconda**, который можно загрузить на сайте `http://www.continuum.io/downloads`.

[1] 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.*



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.

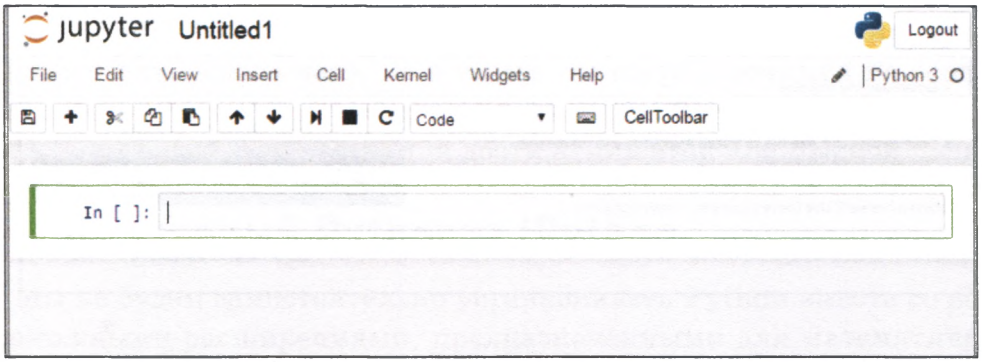
Установите IPython, следуя приведенным на сайте инструкциям. Этот процесс не должен вызвать у вас никаких затруднений.

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

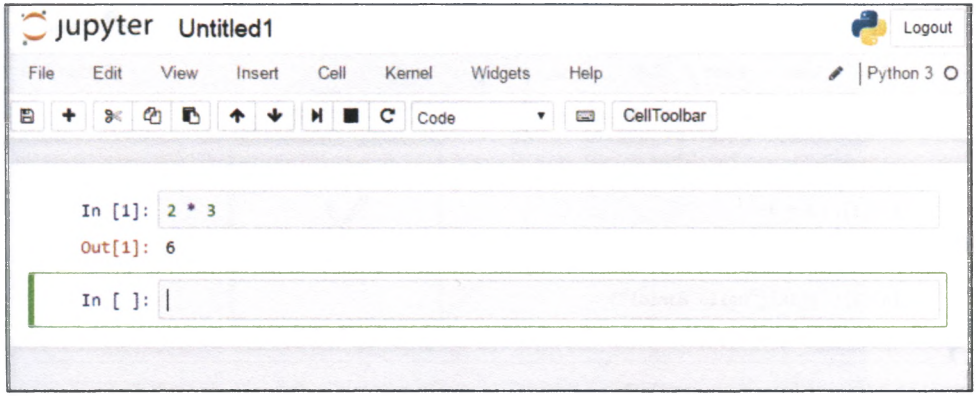
Запустив интерактивную оболочку Jupyter Notebook (Блокнот), щелкните на кнопке New у правого края окна и выберите в открывшемся меню пункт Python 3, что приведет к открытию пустого **блокнота**.



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!

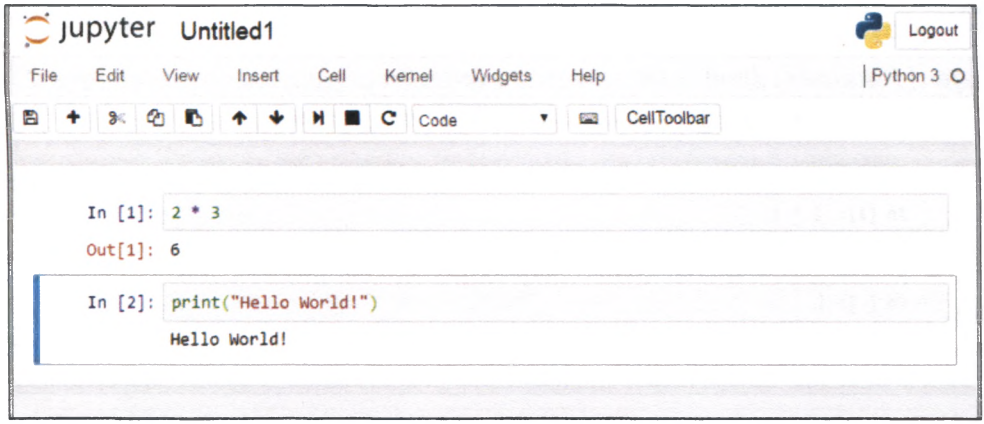
Не обращайте внимания на надписи “In [1]” и “Out[1]”, которыми в IPython помечаются инструкция и ответ. Так оболочка напоминает вам о том, что именно вы просили сделать (input) и что вы получаете в ответ (output). Числа в скобках указывают на последовательность вопросов и ответов, что весьма удобно,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

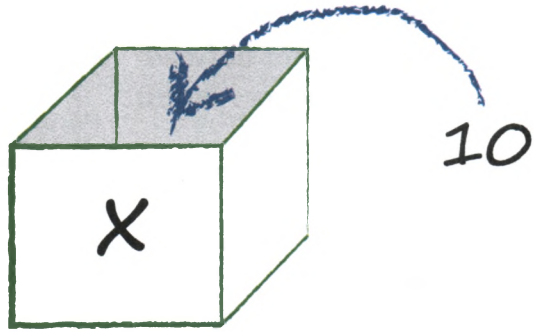


Как видите, ввод инструкции, предписывающей вывод фразы “Hello World!”, не привел к удалению предыдущей ячейки с содержащимися в ней собственной инструкцией и собственным выводом. Это средство оказывается очень полезным при поэтапном создании решений из нескольких частей.

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 “x”? Потому что Python всегда старается вычислить все, что только можно, а `x` 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 успешно справляются 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.

```
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

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-3-5797bafab021> in <module>()
              6 print(y)
              7
        ----> 8 print(z)

        NameError: name 'z' is not defined

In [ ]:
```

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 несколькими 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*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 мы ожидаем, 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*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 неким 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 вовсе 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.

Введите 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 сеансах 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
```

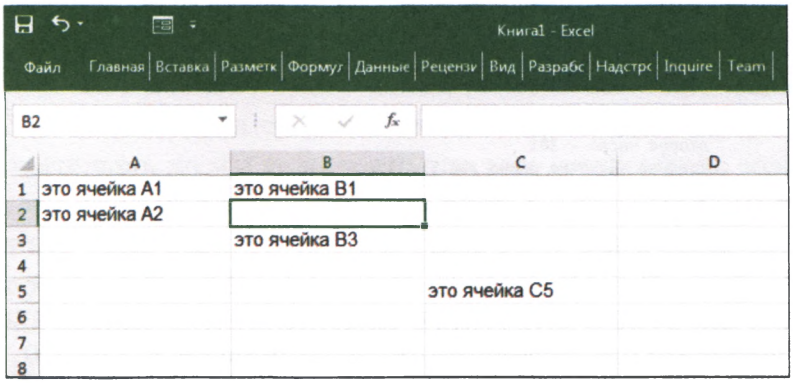
Возможно, вы обратили внимание на то, что в коде функции среднее двух значений находится путем деления не на 2, а на 2,0. Почему мы так поступили? Это особенность Python, которая мне самому не нравится. Если бы мы делили на 2, то результат был бы округлен до ближайшего целого в меньшую сторону, поскольку 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).



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 3×2, 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)
        <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 позволят 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 3×2.

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 одалживаем. 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!

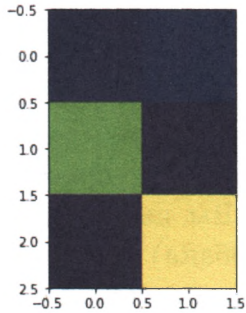
Кроме того, мы должны дополнительно сообщить IPython о том, что любую графику следует отображать в блокноте, а не в отдельном окне. Это делается с помощью следующей директивы:

```
%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.

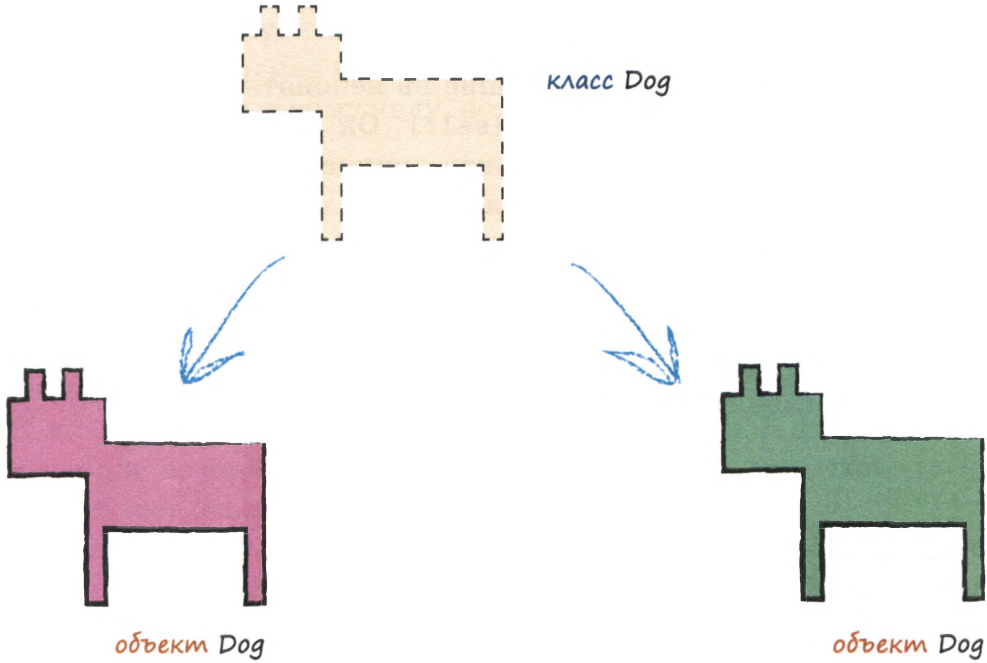
Рассмотрим примеры более полезного использования объектов и классов. Взгляните на следующий код.

```
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, получения 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 магической 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
        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 изменялась.

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 обрастать 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 устраняя 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, $\mathbf{W}_{\text{входной\_скрытый}}$, размерностью **hidden_nodes** × **input_nodes**;
- другую матрицу для связей между скрытым и выходным слоями, $\mathbf{W}_{\text{скрытый\_выходной}}$, 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 3×3. 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, -0.1606345 ]])
```

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 временным 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:

$$\mathbf{X}_{\text{скрытый}} = \mathbf{W}_{\text{входной\_скрытый}} \cdot \mathbf{I}$$

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:

$$\mathbf{O}_{\text{скрытый}} = \text{сигмоида}\,(\mathbf{X}_{\text{скрытый}})$$

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:

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

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 **α** — 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

```
# 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.

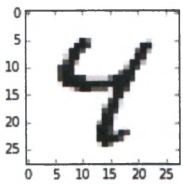
Далее мы займемся решением одной конкретной задачи: обучение нейронной сети распознаванию рукописных цифр.

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 предоставлены следующие два файла:

- тренировочный набор (`http://www.pjreddie.com/media/files/mnist_train.csv`);
- тестовый набор (`http://www.pjreddie.com/media/files/mnist_test.csv`).

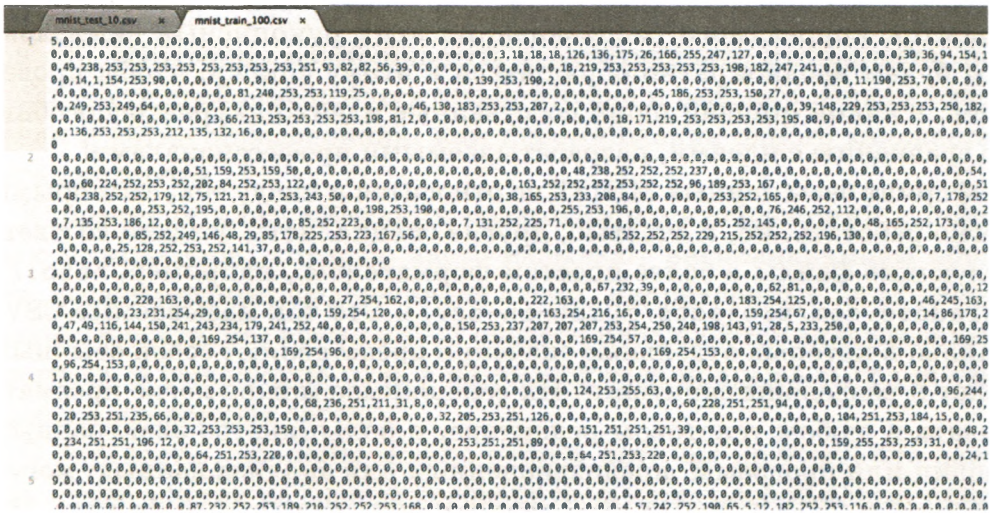
Тренировочный набор содержит **60 000** промаркированных экземпляров, используемых для тренировки нейронной сети. Слово “промаркированные” означает, что для каждого экземпляра указан соответствующий правильный ответ.

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.

Идея разделения тренировочных и тестовых данных распространена среди специалистов по машинному обучению.

Присмотримся к этим файлам. Ниже показана часть тестового набора MNIST, загруженного в текстовый редактор.



Ого! Создается впечатление, будто здесь что-то не так. Напоминает кадры из кинофильмов 1980-х годов, содержащих сцены взлома компьютерных систем хакерами.

На самом деле все хорошо. В окне текстового редактора отображаются длинные строки текста, которые содержат числа, разделенные запятыми. Это легко можно увидеть. Текстовые строки такие длинные, потому что каждая из них занимает несколько строк на экране. Очень удобно то, что текстовый редактор отображает на полях реальные номера строк, и мы видим четыре целые строки и часть пятой.

Содержимое этих записей, т.е. строк текста, легко понять.

- Первое значение — это **маркер**, т.е. фактическая цифра, например “7” или “9”, которую должен представлять данный рукописный экземпляр. Это ответ, правильному получению которого должна обучиться нейронная сеть.
- Последующие значения, разделенные запятыми, — это значения пикселей рукописной цифры. Пиксельный массив имеет размерность 28×28, поэтому за каждым маркером следуют 784 пикселя. Если у вас есть такое желание, можете пересчитать!

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.

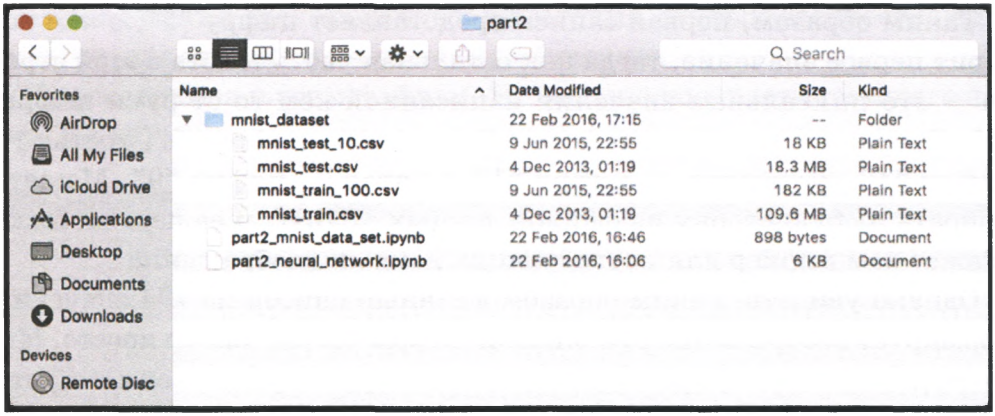
Прежде чем углубиться в рассмотрение деталей и приступить к написанию кода, загрузим небольшое подмножество набора данных, содержащихся в базе MNIST. Файлы данных MNIST имеют очень большой размер, тогда как работать с небольшими наборами значительно удобнее, поскольку это позволит нам экспериментировать, 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/makeyourownneuralnetwork/makeyourownneuralnetwork/master/mnist_dataset/mnist_test_10.csv`
- 100 записей из тренировочного набора данных MNIST:
  `https://raw.githubusercontent.com/makeyourownneuralnetwork/makeyourownneuralnetwork/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.

```
In [4]: data_file = open("mnist_dataset/mnist_train_100.csv", 'r')
        data_list = data_file.readlines()
        data_file.close()

In [5]: len(data_list)

Out[5]: 100

In [6]: data_list[0]

Out[6]: '5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,18,1
        8,18,126,136,175,26,166,255,247,127,0,0,0,0,0,0,0,0,0,0,0,0,30,36,94,154,170,25
        3,253,253,253,253,225,172,253,242,195,64,0,0,0,0,0,0,0,0,0,0,0,49,238,253,253,2
        53,253,253,253,253,253,251,93,82,82,56,39,0,0,0,0,0,0,0,0,0,0,0,0,18,219,253,25
        3,253,253,253,198,182,247,241,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,80,156,107,25
        3,253,205,11,0,43,154,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,14,1,154,253,90,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,139,253,190,2,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,11,190,253,70,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,35,241,225,160,108,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,81,240,253,253,119,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,45,18
        6,253,253,150,27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,93,252,253,18
        7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,249,253,249,64,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,46,130,183,253,253,207,2,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,39,148,229,253,253,253,250,182,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,24,114,221,253,253,253,253,201,78,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,23,66,21
        3,253,253,253,253,198,81,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,18,171,219,253,253,2
        53,253,195,80,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,55,172,226,253,253,253,253,244,
        133,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,136,253,253,253,212,135,132,16,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\n'
```

Вы видите, что длина списка равна 100. Функция Python `len()` сообщает о размере списка. Вы также можете видеть содержимое первой записи, **`data_list[0]`**. Первое число, 5, — это маркер, тогда как остальные 784 числа — это цветовые коды пикселей, из которых состоит изображение. Если вы внимательно присмотритесь, то заметите, что их значения не выходят за пределы диапазона 0 до 255. Вы можете взглянуть на другие записи, чтобы проверить, выполняется ли это условие и для них. Вы убедитесь в том, что так оно и есть: значения кодов всех цветов попадают в диапазон чисел от 0 до 255.

Ранее приводился пример графического отображения прямоугольного массива чисел с использованием функции `imshow()`. То же самое мы хотим сделать и в данном случае, но для этого нам нужно преобразовать список чисел, разделенных запятыми, в подходящий массив. Это можно сделать в соответствии со следующей процедурой:

- разбить длинную текстовую строку значений, разделенных запятыми, на отдельные значения, используя символ запятой в качестве разделителя;

- проигнорировать первое значение, являющееся маркером, извлечь оставшиеся 28*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.asfarray(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.

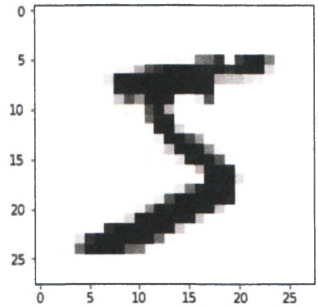
Следующая строка кода выглядит чуть более сложной, поскольку в ней происходит сразу несколько вещей. Начнем с середины. Запись списка в виде **all_values[1:]** указывает на то, что берутся все элементы списка за исключением первого. Тем самым мы игнорируем первое значение, играющее роль маркера, и берем лишь остальные 784 элемента. `numpy.asfarray()` — 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 28×28. 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.

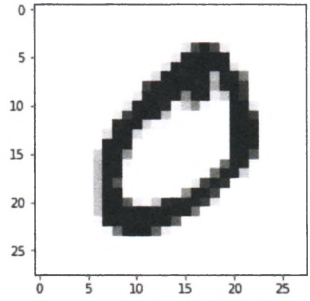
Результат работы кода представлен ниже.

```
In [8]: all_values = data_list[0].split(',')
        image_array = numpy.asfarray(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.asfarray(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.asfarray(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.asfarray(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        0.208       0.62729412  0.99223529  0.62729412  0.20411765
```

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 28×28=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

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

```
# 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.asfarray(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 28×28, 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 бо́льшим, 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.

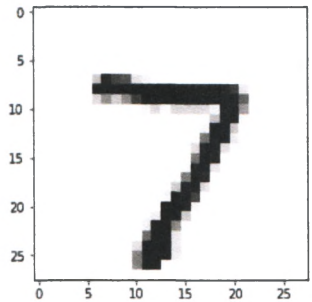
Прежде всего, необходимо получить тестовые записи. Соответствующий код очень похож на тот, который мы использовали для получения тренировочных данных.

```
# 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(',')
        # вывести маркер
        print(all_values[0])

        7

In [4]: image_array = numpy.asfarray(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.asfarray(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 обучаться, 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.asfarray(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, 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

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

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.

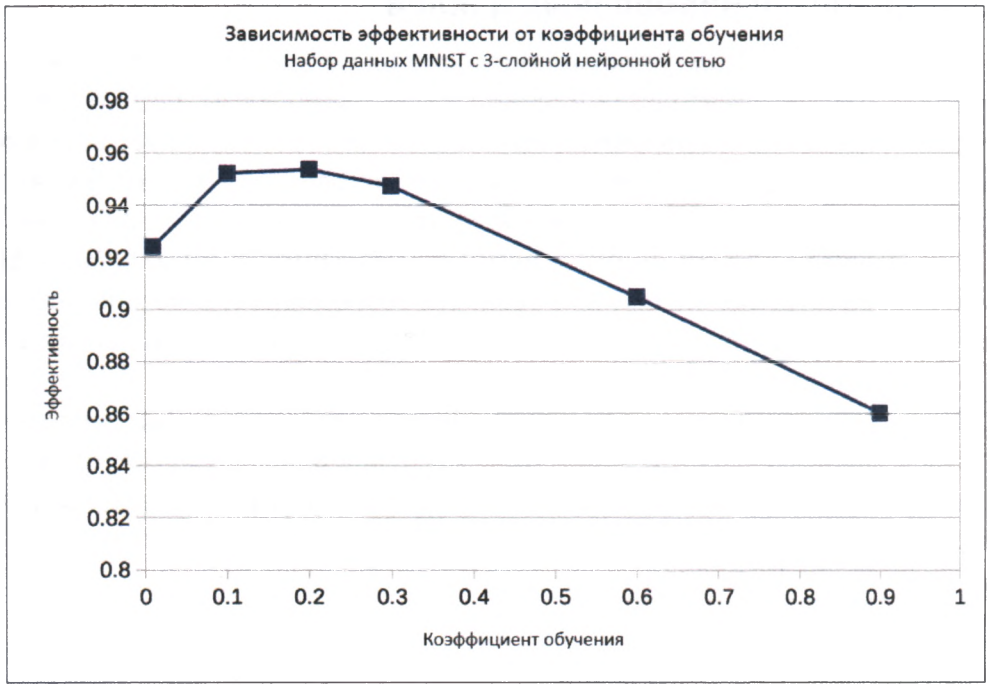
Давайте удвоим это значение до величины **0,6** и посмотрим, как это скажется на способности нейронной сети обучаться. Выполнение кода с таким значением коэффициента обучения дает показатель эффективности **0,9047**. Этот результат хуже прежнего. По-видимому, увеличение коэффициента обучения нарушает 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.

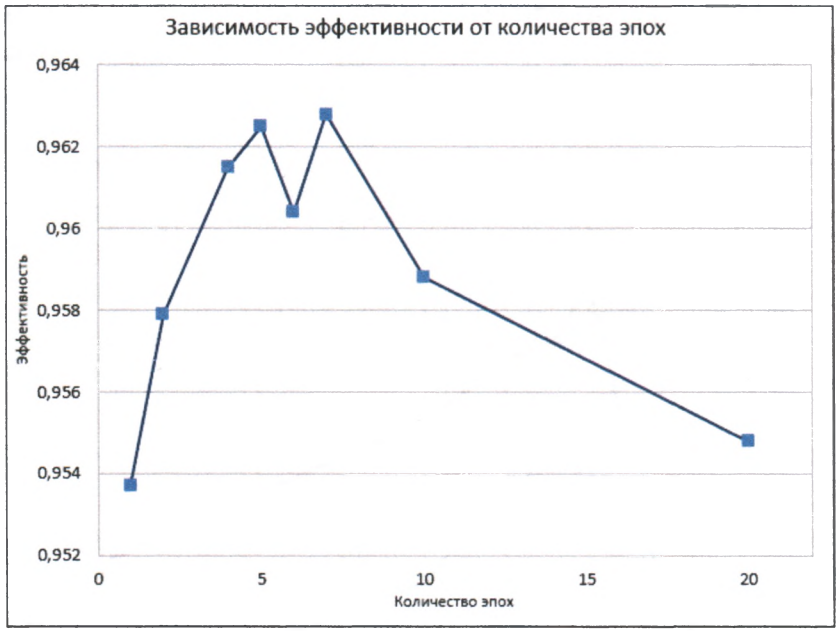
Посмотрим, что нам дадут две тренировочные эпохи. Для этого мы должны немного изменить код, предусмотрев в нем дополнительный цикл выполнения кода тренировки. В приведенном ниже коде внешний цикл выделен цветом, чтобы сделать его более заметным.

```
# 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.asfarray(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.



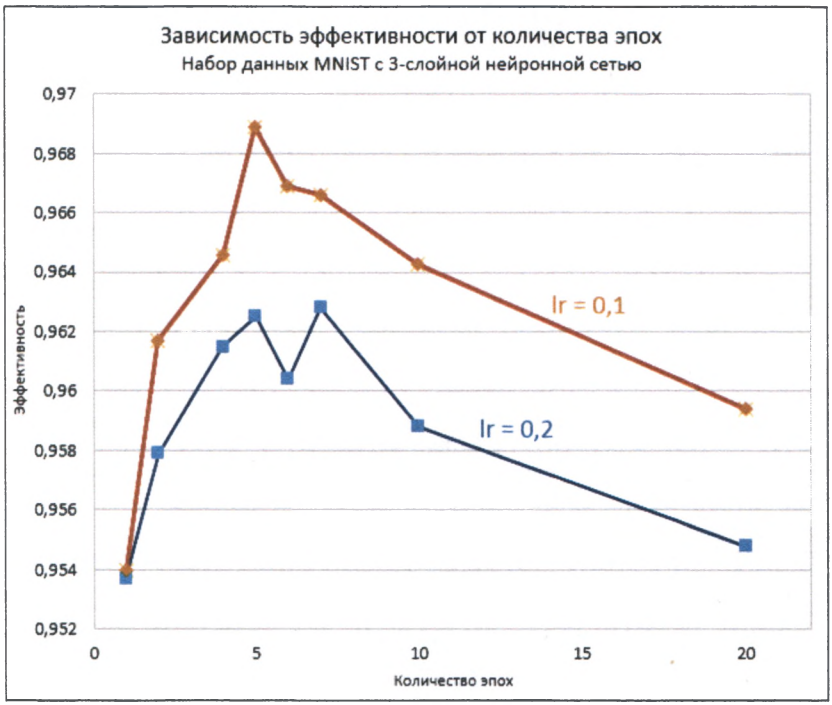
Теперь пиковое значение эффективности составляет **0,9628**, или 96,28%, при семи эпохах.

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 обращу 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.

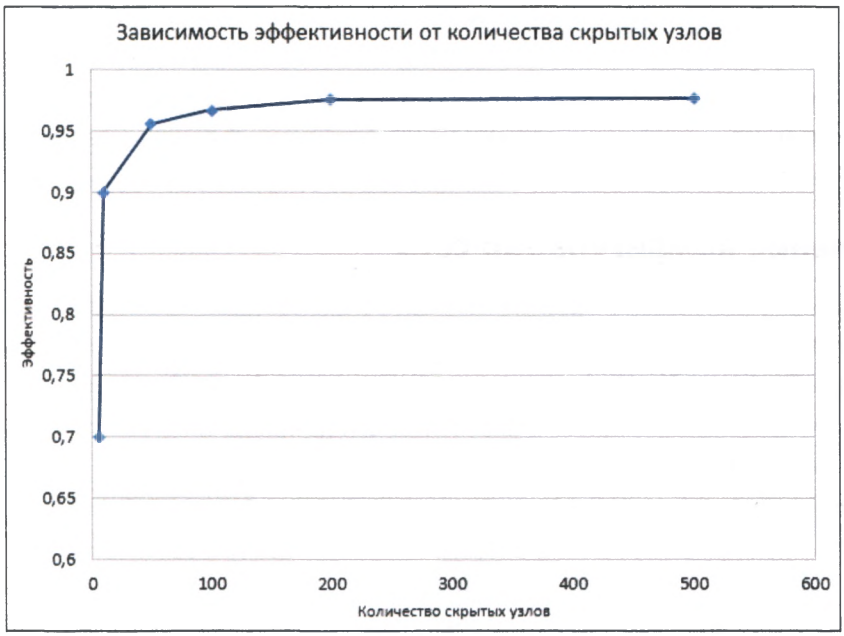
Прежде чем приступить к экспериментам с различными количествами скрытых узлов, давайте подумаем, что при этом может случиться. Скрытый слой как раз и является тем слоем, в котором происходит обучение. Вспомните, что узлы входного слоя принимают входные сигналы, а узлы выходного выдают ответ сети. Собственно, 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.asfarray(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.asfarray(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

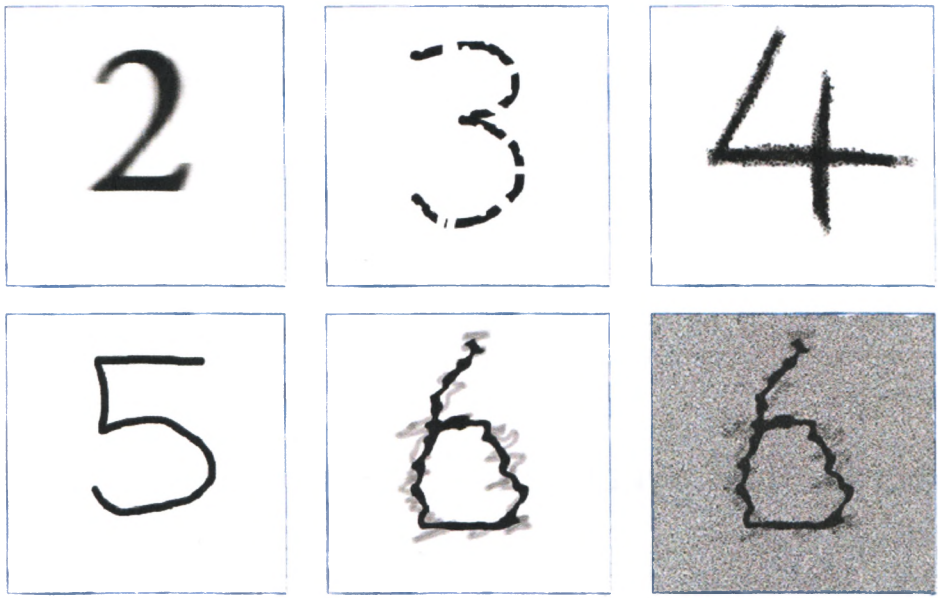
В главе 2 мы распознавали изображения рукописных цифр из базы данных MNIST. А почему бы не использовать собственный рукописный текст?

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`), 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 28×28 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 28×28 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:

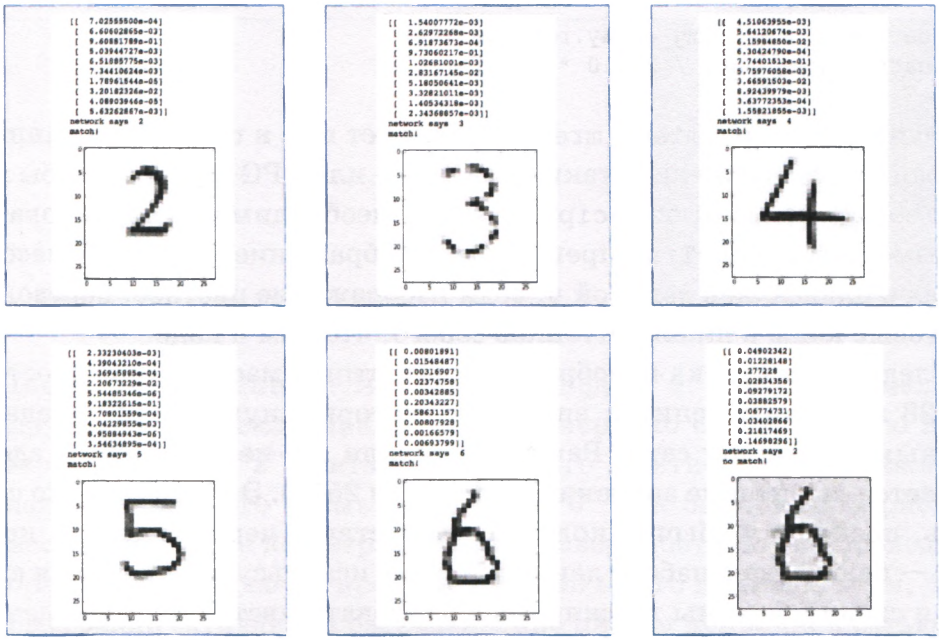
https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_load_own_images.ipynb

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

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

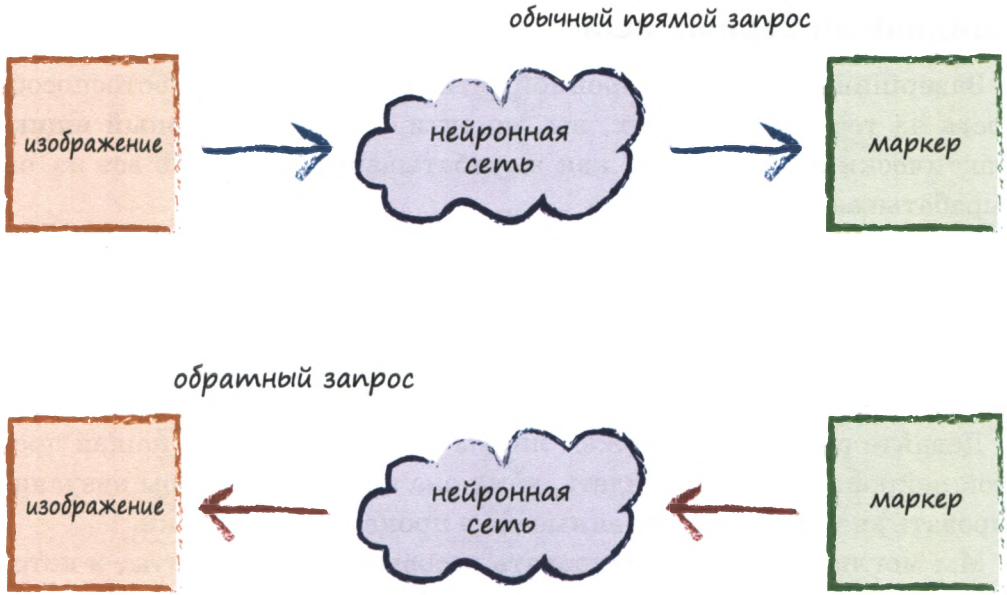
даже большего количества узлов полностью лишит сеть возможности нормально работать.

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:

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

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:

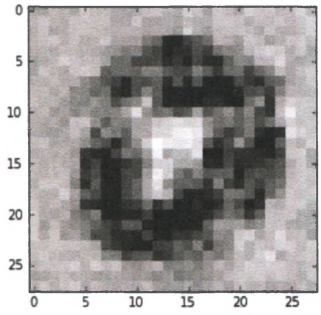
https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part3_neural_network_mnist_backquery.ipynb

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, 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.

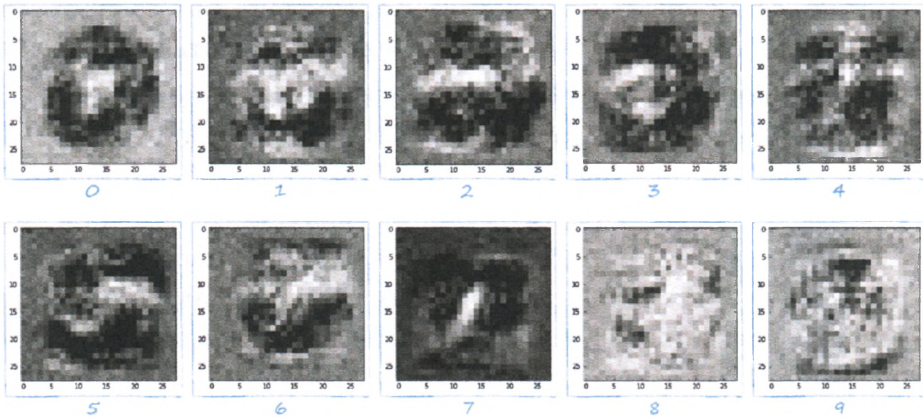
- **Темные области** — это те части искомого изображения, которые, если обвести их маркером, свидетельствуют в пользу того, что ответом следует считать "0". Форма их контура действительно напоминает цифру "0".
- **Светлые области** — это те участки искомого изображения, которые следует считать не закрашенными, 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 бо́льшим 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 налицо 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, вращая 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 28×28, чтобы повернуть изображение, а затем проделать обратную операцию по преобразованию массива в список из 784 входных сигналов, которые мы подадим на вход нейронной сети.

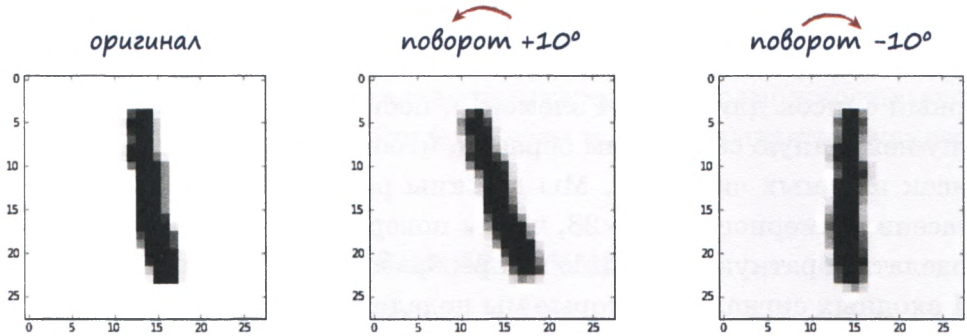
В приведенном ниже коде показан пример использования функции `scipy.ndimage.interpolation.rotate()` в предположении, что у нас уже имеется массив **`scaled_input`**, о котором шла речь ранее.

```
# 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 28×28. 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, “заваливающего” 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:

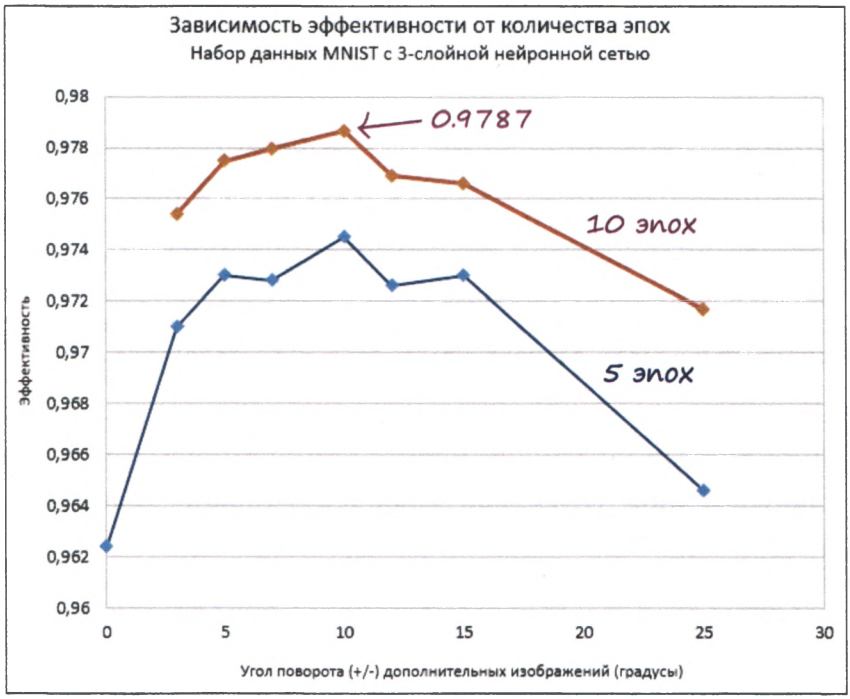
```
https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/
blob/master/part3_neural_network_mnist_data_with_rotations.ipynb
```

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!

Эпилог

Надеюсь, мне удалось продемонстрировать, что в случае использования традиционных подходов задачи, которые легко решает человек, для компьютеров оказываются крепким орешком. Одним из вызовов, брошенных так называемому “искусственному интеллекту”, является задача распознавания образов (изображений).

Значительный прогресс в области распознавания изображений, как, впрочем, и в ряде других проблемных областей, был достигнут благодаря использованию нейронных сетей. Толчком к развитию теории нейронных сетей послужило стремление разгадать тайну биологического мозга, который, обладая, на первый взгляд, меньшими быстродействием и ресурсами по сравнению с современными суперкомпьютерами, даже в случае таких живых существ, как попугаи или насекомые, способен решать сложные задачи, например управление полетом, прием пищи или строительство жилья. Кроме того, биологический мозг необычайно устойчив к повреждениям и способен различать даже несовершенные сигналы. Цифровым компьютерам и традиционным вычислительным подходам до этого пока что далеко.

На сегодняшний день нейронные сети — ключевой фактор успешности самых фантастических проектов в области искусственного интеллекта. Не ослабевает интерес к применению нейронных сетей в области машинного обучения, особенно **глубокого обучения**, предполагающего наличие иерархии обучающих методов. В начале 2016 года компьютерная система компании DeepMind, ныне принадлежащей компании Google, победила профессионального игрока в го. Это был поворотный момент в развитии искусственного интеллекта, поскольку игра го требует гораздо более глубокого продумывания стратегии по сравнению с шахматами, и исследователи полагали, что наступления столь знаменательного события придется ожидать еще долгие годы. Ключевую роль в этом успехе сыграли нейронные сети.

Хочется верить, что мне удалось продемонстрировать вам, насколько простые идеи лежат в основе нейронных сетей. Я также надеюсь, что эксперименты с нейронными сетями доставили вам удовольствие. Возможно, это пробудит в вас интерес к изучению других разновидностей машинного обучения и искусственного интеллекта.

Если хотя бы одно из моих предположений оказалось верным, моя цель достигнута.

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 Ньютоном, каждый из которых приписывал открытие дифференциального исчисления себе!

Готфрид Лейбниц

Сэр Исаак Ньютон

Прямая линия

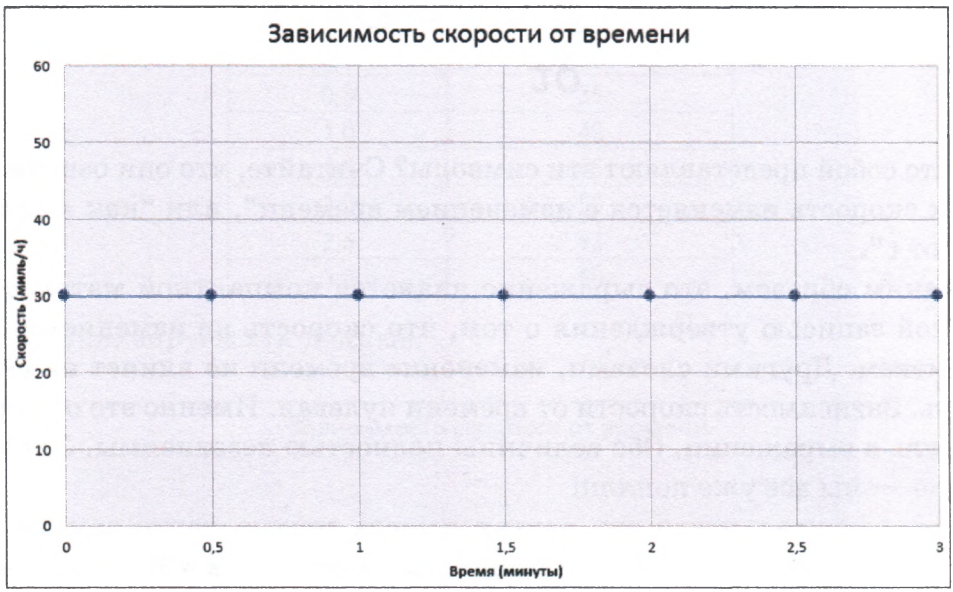
Чтобы настроиться на работу, начнем для разминки с очень простого сценария.

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 **∂s/∂t = 0**, нам не потребуется никакое дифференциальное исчисление. Говоря языком математиков, "это следует из самого вида выражения".

Выражения вида **∂s/∂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

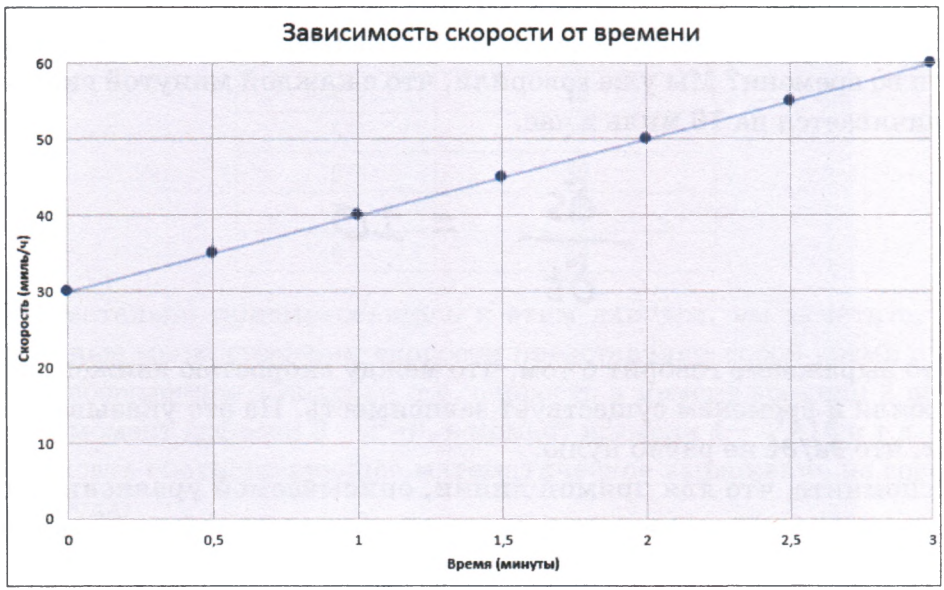
Представьте себе все тот же автомобиль, движущийся со скоростью 30 миль в час. Вы надавили на педаль акселератора, и автомобиль набирает скорость. Вы удерживаете педаль нажатой, смотрите на показания спидометра и записываете их через каждые 30 секунд.

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 символьные 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 `∂s/∂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.

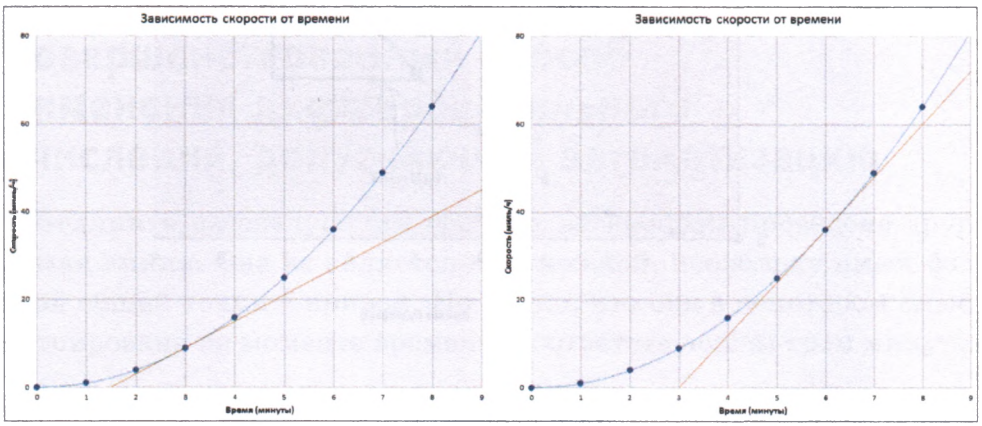
Применение дифференциального исчисления вручную

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 искомую 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 **Δs**, а приращение по горизонтали (время) — как **Δt**. Символ **Δ** (читается "дельта") просто означает небольшое изменение. Поэтому **Δt** — это небольшое изменение **t**.

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

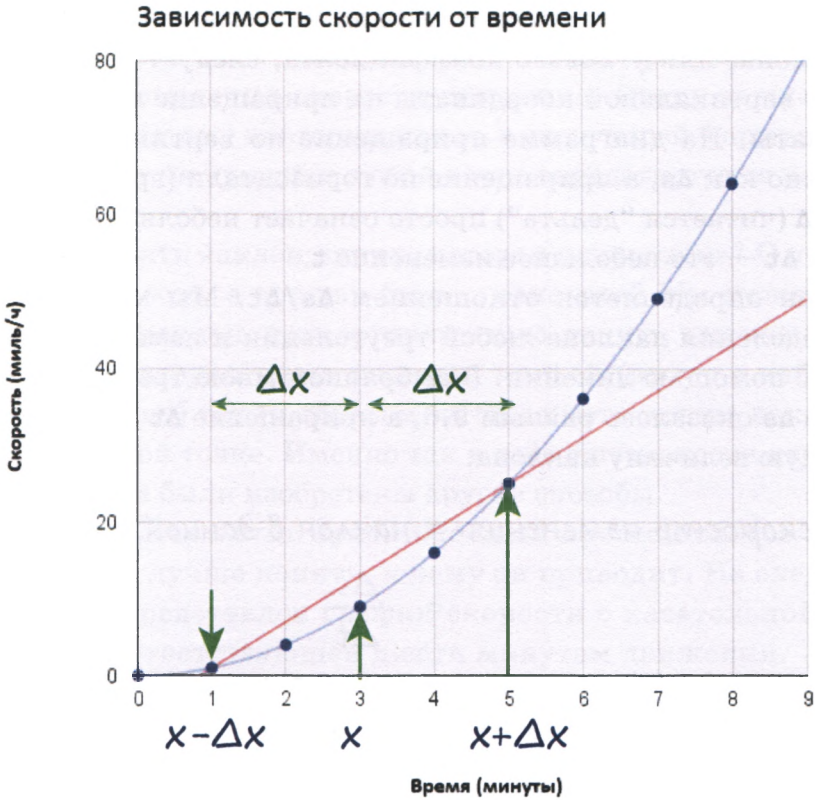
$$\begin{aligned} \text{скорость изменения} &= \text{наклон в данной точке} \\ &= \frac{\Delta s}{\Delta t} \\ &= 9{,}6 \,/\, 0{,}8 \\ &= 12{,}0 \end{aligned}$$

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.



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**.

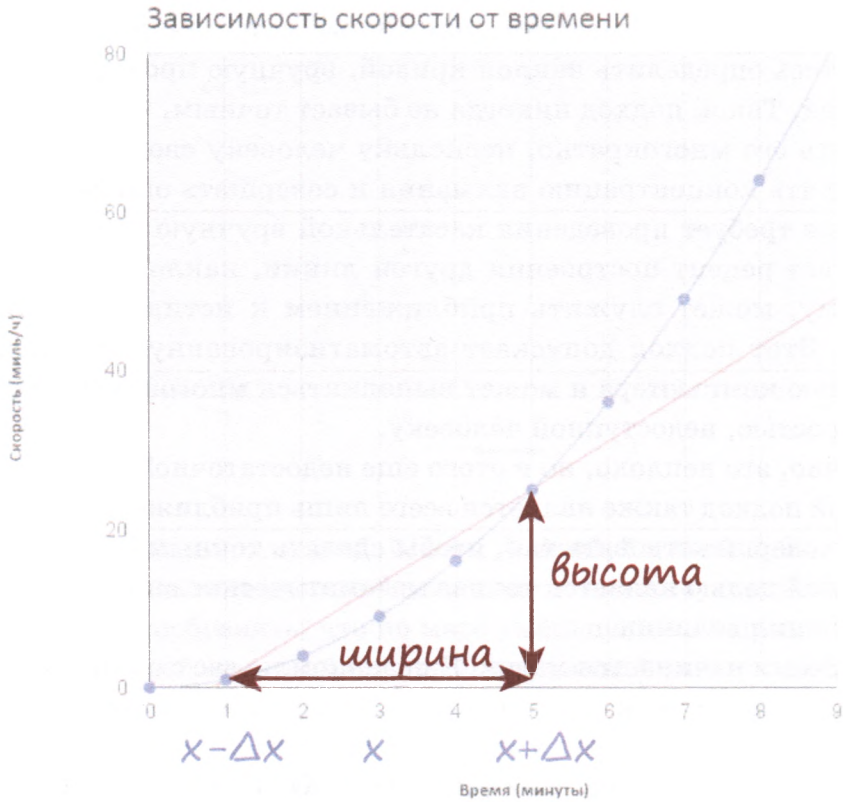
Используя математические обозначения, можно сказать, что **Δx** равно 2 минуты. Тогда выбранным нам точкам соответствуют координаты **x - Δx** и **x + Δx**. Вспомните, что символ **Δ** означает "небольшое изменение", поэтому **Δx** — это небольшое изменение **x**.

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

Если мы возьмем значения скорости в моменты времени **x - Δx** и **x + Δx** 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 + Δx** и **x − Δx**, 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 + Δx** и **x − Δx**. , 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.

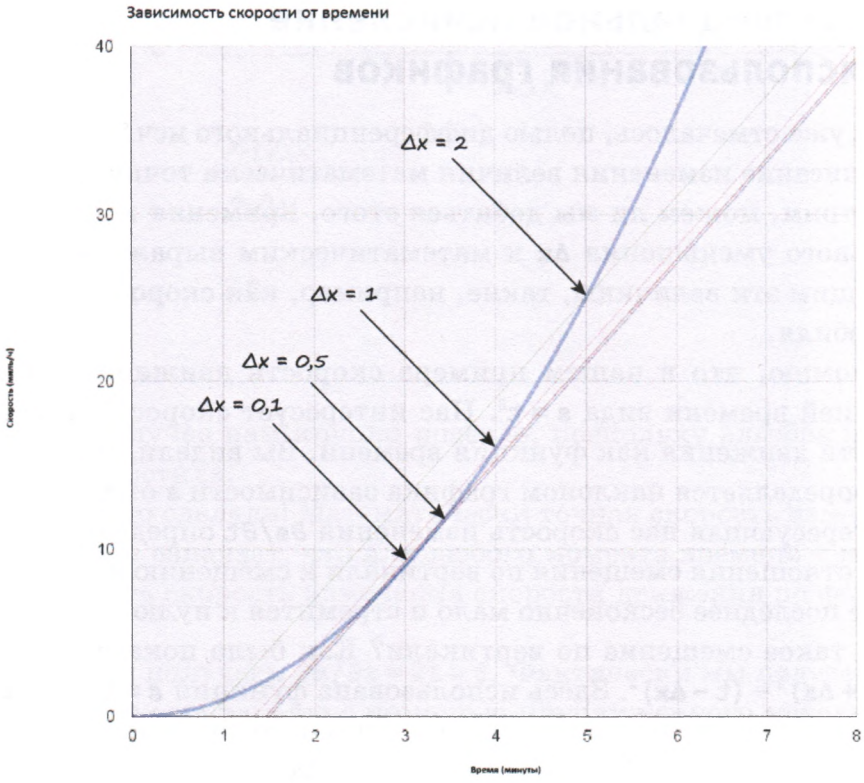
Сделаем паузу и осмыслим полученные результаты. Сначала мы попытались определить наклон кривой, вручную проведя касательную к ней. Такой подход никогда не бывает точным, и мы не можем повторять его многократно, поскольку человеку свойственно уставать, терять концентрацию внимания и совершать ошибки. 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 **Δx**? Следующая иллюстрация демонстрирует несколько приближений в виде наклонных прямых, соответствующих уменьшению **Δx**.



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

При бесконечно малой величине **Δx** 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 Δx 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 $\partial s/\partial t$ 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 Δx.

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:

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

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

Раскроем и упростим это выражение:

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

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

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

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

Итак, мы это сделали! Математически точная скорость изменения **∂s/∂t = 2t**. Это означает, что для любого момента времени **t** мы можем вычислить скорость изменения скорости движения по формуле **∂s/∂t = 2t**.

При **t=3** мы получаем **∂s/∂t = 2t = 6**. Фактически мы подтвердили этот результат еще раньше с помощью приближенного метода. Для **t=6** получаем **∂s/∂t = 2t = 12**, что также согласуется с найденным ранее результатом.

А что насчет ста минут? В этом случае **∂s/∂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 **Δx** 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 + Δx**, и скоростью **s**, рассчитанной в момент времени **t - Δx**. 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 + Δx)** и **(t - Δx)**, которое по-прежнему равно **2Δx**:

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

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 **Δx**, тогда как раньше эти члены взаимно сокращались.

Вспомните, что корректное значение наклона получается лишь в том случае, если величина **Δx** уменьшается, становясь бесконечно малой.

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

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 **Δx**, 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.

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

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

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

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: $5*2t^4 = 10t^4$.

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

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

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

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

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

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

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.

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

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 **∂f/∂x** проще найти производные **∂f/∂y** и **∂y/∂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:

$$f = y^2 \quad и \quad y = x^3 + x$$

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

Мы разбили задачу на две простые части. Первая часть дает **(∂f/∂y) = 2y**, вторая — **(∂y/∂x) = 3x² + 1**. Объединяя эти части с помощью цепного правила, получаем

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

Мы знаем, что **y = x³ + 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 $\partial f/\partial x$? 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 $\partial f/\partial x$, 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*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

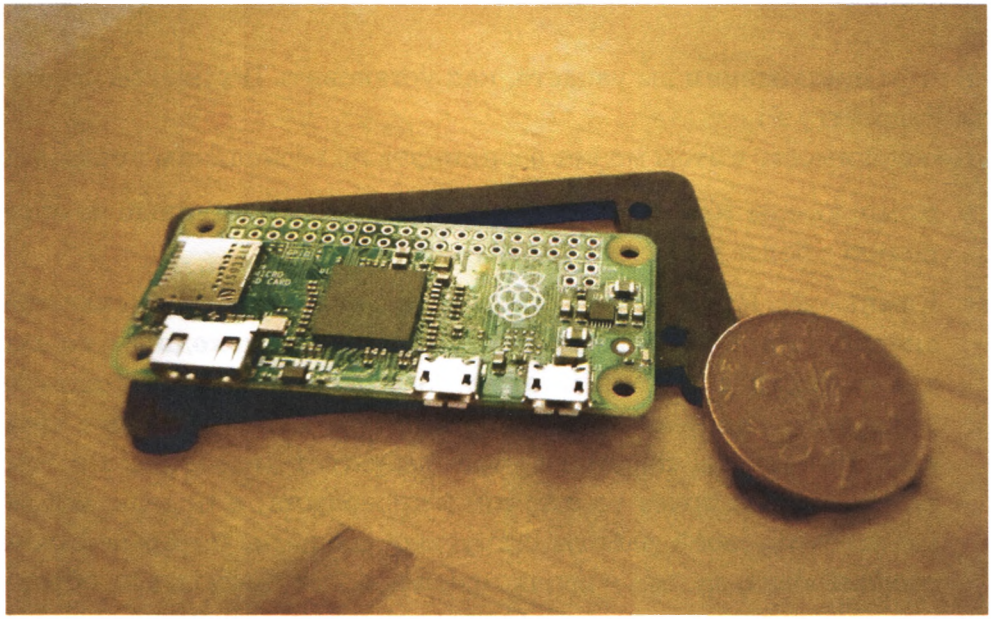
В этом приложении вы узнаете, как установить IPython на устройстве Raspberry Pi.

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 двухпенсовой монетой для сравнения.



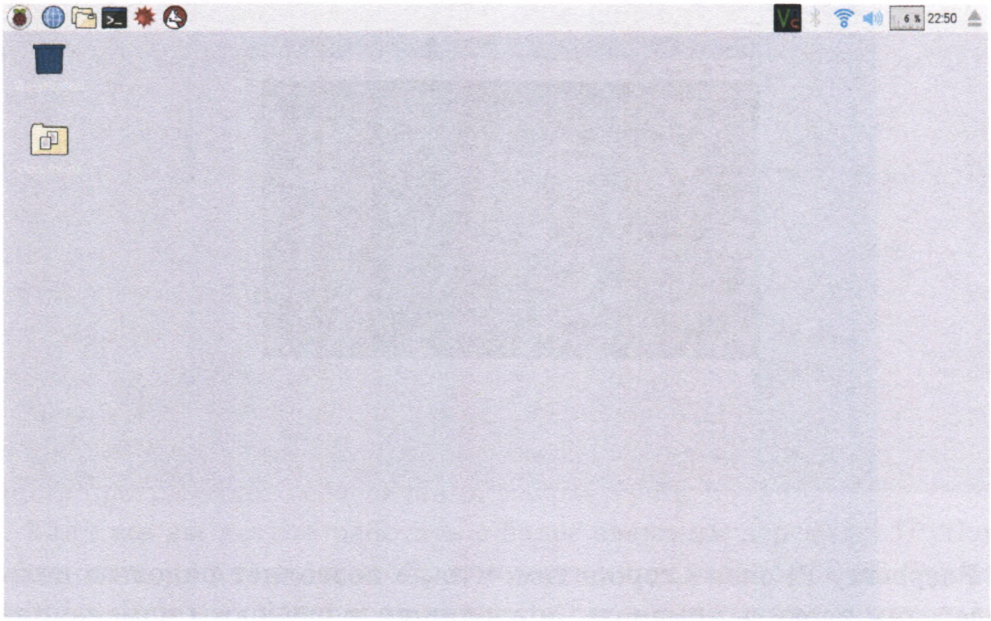
Установка IPython

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.

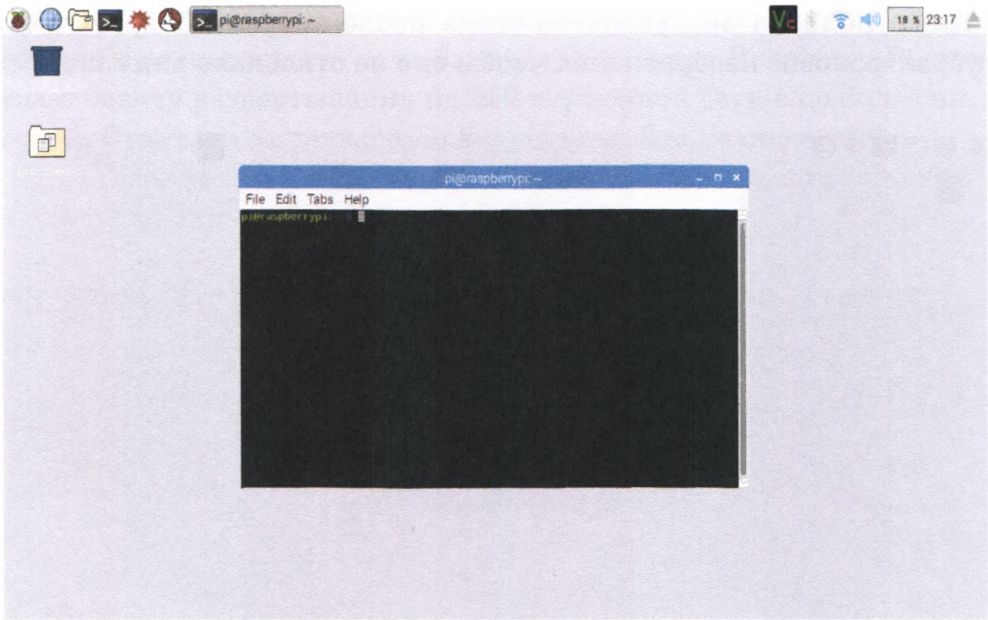


В левом верхнем углу видны кнопки меню, а также другие значки.

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>.

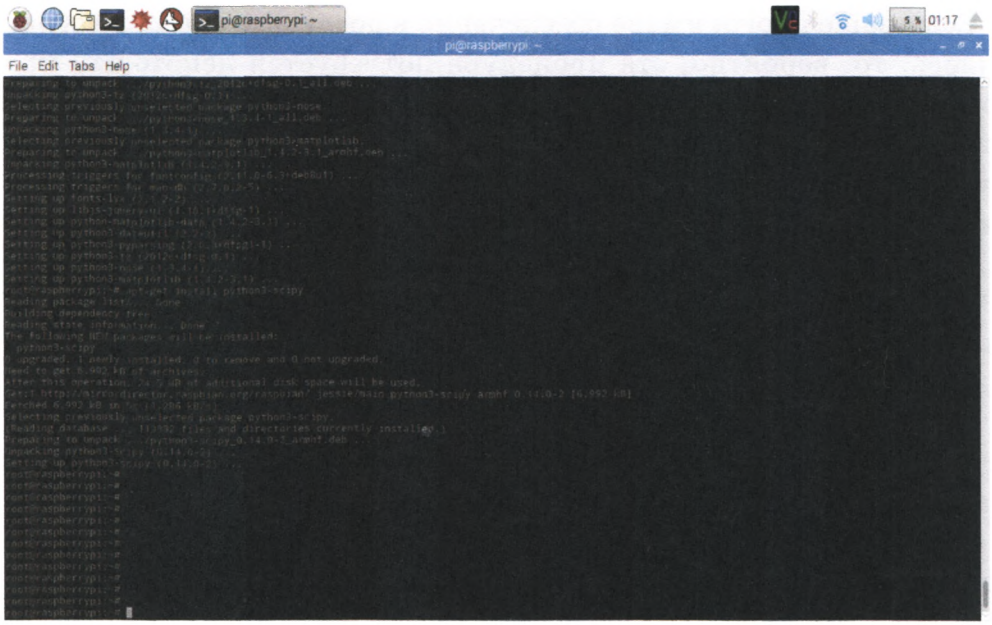
Обновив состояние системы, введите команду для получения IPython. Имейте в виду, что на момент написания книги программные пакеты Raspbian не содержали версий IPython, позволяющих работать с размещенными на сайте GitHub для всеобщего доступа блокнотами, которые мы ранее создали. Если бы они их содержали, 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 (**Py**thon **P**ackage **I**ndex — 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
```

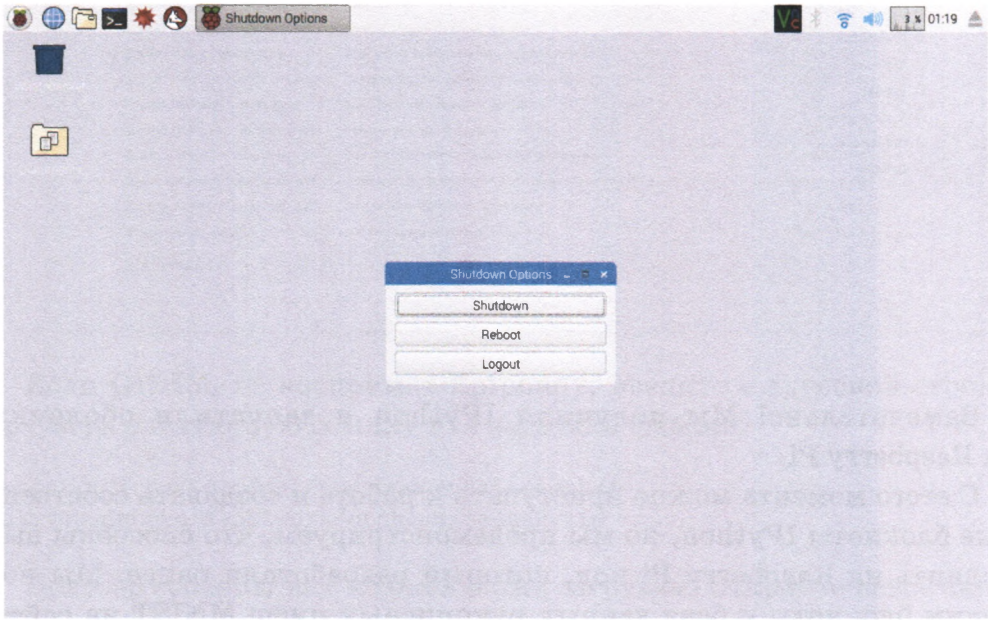
В последних версиях Raspbian браузер Epiphany заменен браузером Chromium (версия с открытым исходным кодом популярного браузера Chrome). Но Epiphany гораздо легче тяжеловесного Chromium и работает лучше с крохотным Raspberry Pi Zero. Чтобы установить его в качестве браузера по умолчанию для блокнотов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

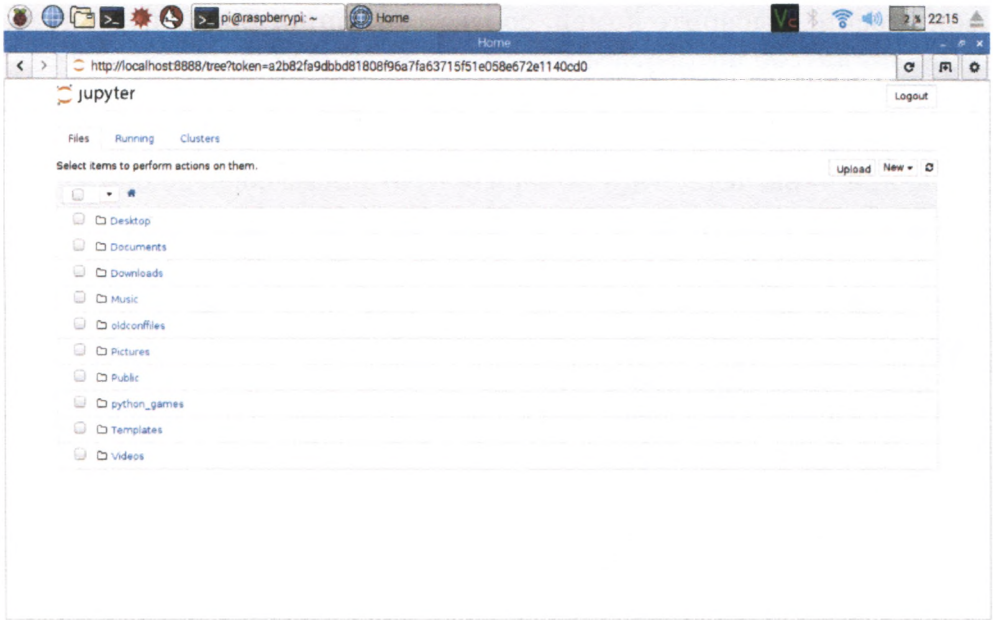
Теперь все готово. Перезапустите Raspberry Pi в том случае, если обновление повлекло за собой некие глубокие изменения, такие как обновление ядра. Для этого выберите пункт Shutdown основного меню в левом верхнем углу, а затем пункт Reboot, как показано ниже.



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.

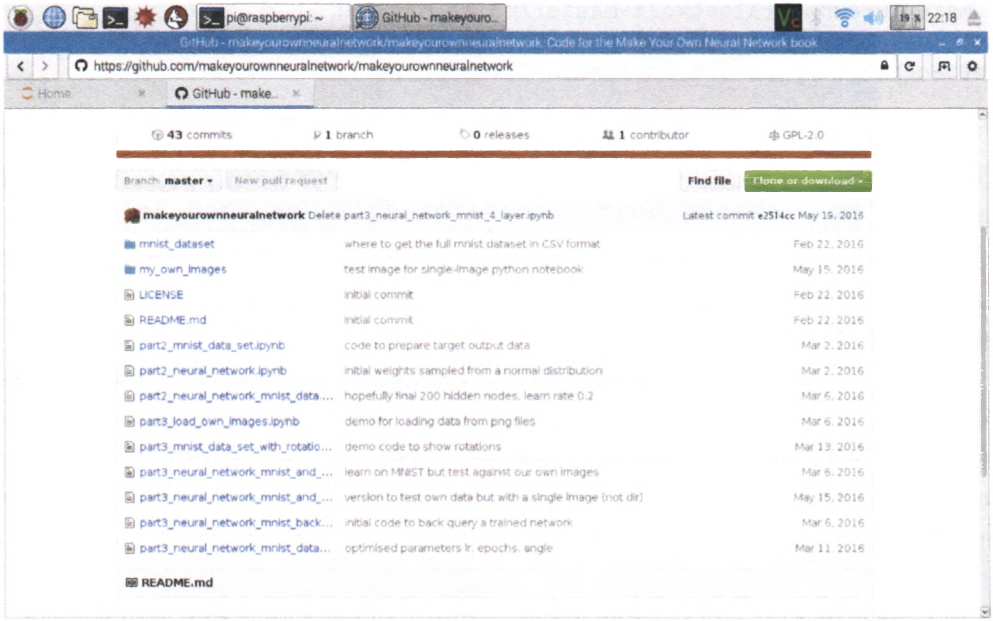


Замечательно! Мы получили IPython и запустили оболочку на Raspberry Pi.

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.



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.

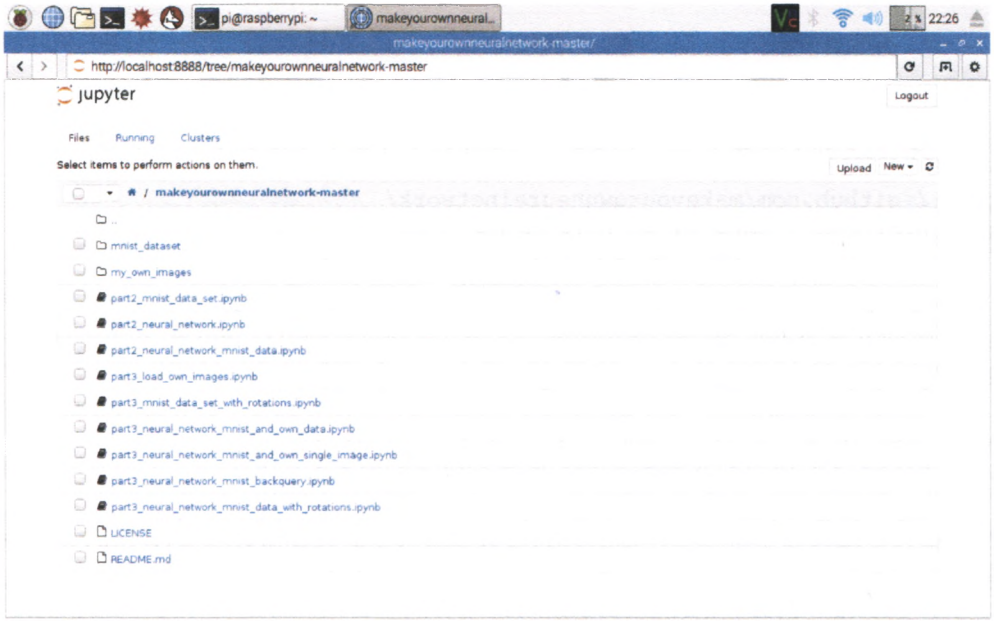
На сайте GitHub содержатся лишь сокращенные версии наборов данных MNIST, поскольку сайт не позволил бы мне разместить файлы большего размера. Чтобы получить полный набор данных, введите в том же окне терминала следующие команды для перехода в каталог `mnist_dataset` и получения тренировочного и тестового 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.

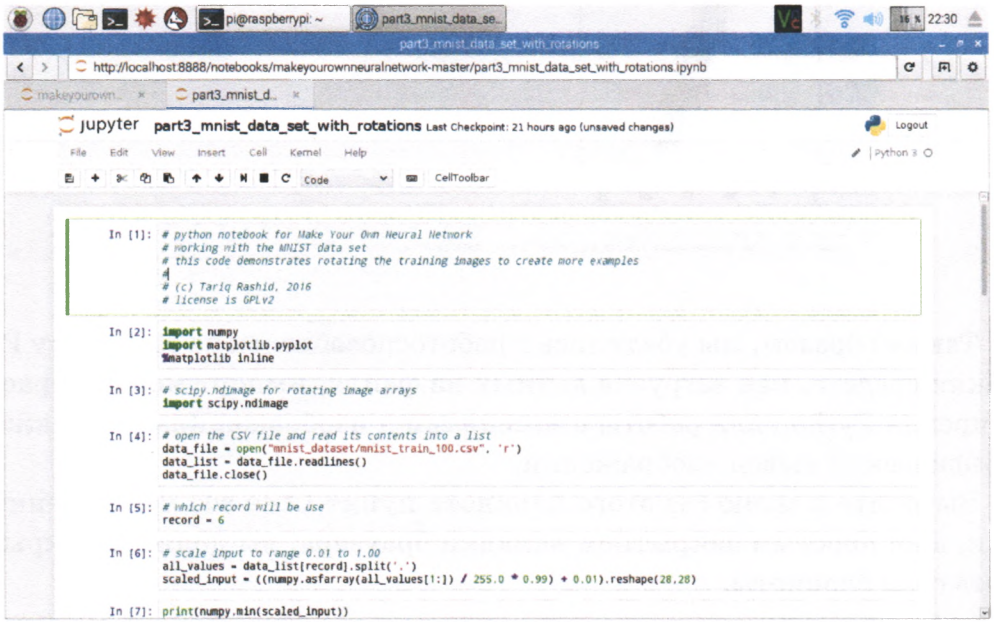
Теперь вы располагаете всеми необходимыми блокнотами и данными MNIST. Закройте это окно терминала, но не то, из которого запускали IPython.

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.

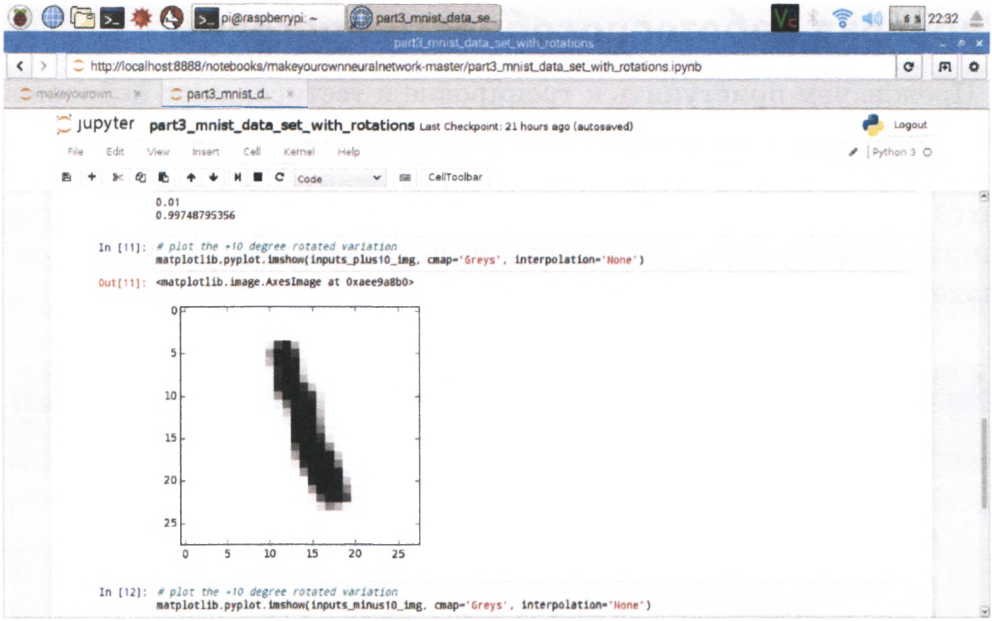


Проверка работоспособности программ

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.



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.



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.

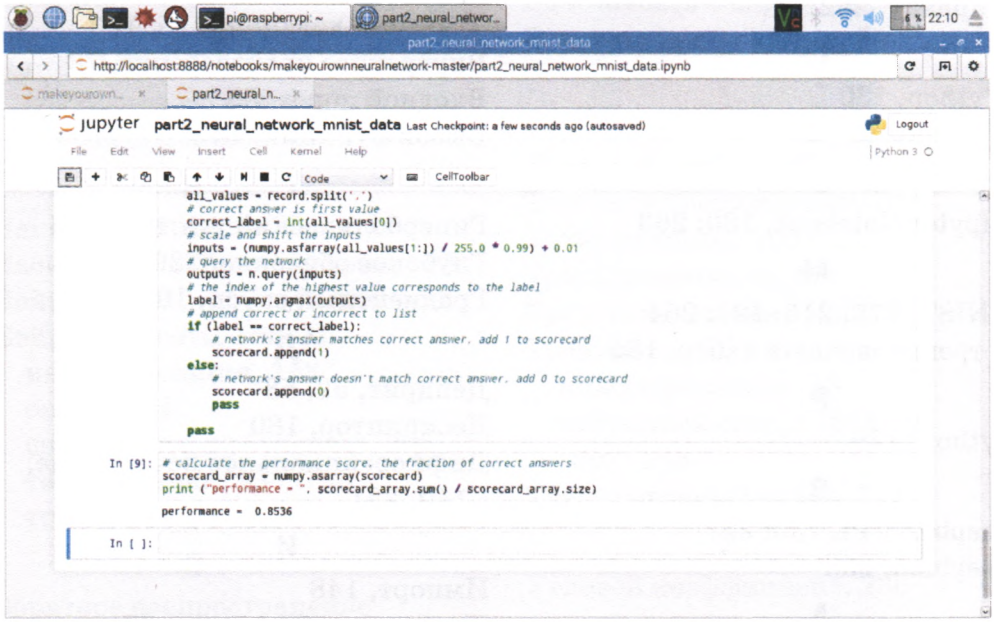
Тренировка и тестирование нейронной сети

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.



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!

Предметный указатель

Ц

Ч

Э

Я

ИСПОЛЬЗУЙТЕ МОЩНЫЕ СРЕДСТВА ЯЗЫКА PYTHON ДЛЯ СОЗДАНИЯ НЕЙРОННЫХ СЕТЕЙ

Эта книга представляет собой введение в теорию и практику создания нейронных сетей. Она предназначена для тех, кто хочет узнать, что такое нейронные сети, где они применяются и как самому создать такую сеть, не имея опыта работы в данной области. Автор простым и понятным языком объясняет теоретические аспекты, знание которых необходимо для понимания принципов функционирования нейронных сетей и написания соответствующих программных инструкций. Изложение материала сопровождается подробным описанием процедуры поэтапного создания полностью функционального кода, который реализует нейронную сеть на языке Python и способен выполняться даже на таком миниатюрном компьютере, как Raspberry Pi Zero.

Основные темы книги:

- нейронные сети и системы искусственного интеллекта
- структура нейронных сетей
- сглаживание сигналов, распространяющихся по нейронной сети, с помощью функции активации
- тренировка и тестирование нейронных сетей
- интерактивная среда программирования IPython
- использование нейронных сетей в качестве классификаторов объектов
- распознавание образов с помощью нейронных сетей

Об авторе

Тарик Рашид — специалист в области количественного анализа данных и разработки решений на базе продуктов с открытым исходным кодом. Имеет ученую степень по физике и степень магистра по специальности "Machine Learning and Data Mining". Проживая в Лондоне, он возглавляет местную группу разработчиков Python (насчитывающую около 3000 участников), организует многочисленные семинары и часто выступает с докладами на международных конференциях.