МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания
по выполнению лабораторных работ
по дисциплине
«СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА»
для студентов направления подготовки
37.03.02 Конфликтология
направленность (профиль):
Конфликт-менеджмент

Ставрополь, 2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ..... 5

Лабораторная работа 1. Анализ современных программных средств с применением ИИ.....6

Лабораторная работа 2 Знакомство с принципами функционирования искусственных нейронных сетей и с языком программирования Python.....7

Лабораторная работа 3 Принципы классификации образов с помощью искусственных нейронных сетей. Функции, массивы и объекты в Python.....18

Лабораторная работа 4 Решение сложных задач классификации с помощью искусственных нейронных сетей. Распространение сигналов по нейронной сети.....42

Лабораторная работа 5 Решение сложных задач классификации с помощью искусственных нейронных сетей. Метод обратного распространения.....64

Лабораторная работа 6 Обновление весовых коэффициентов. Набор рукописных цифр.....95

Лабораторная работа 7 Тестирование нейронной сети для распознавания рукописных цифр..... 123

цифр MNIST..... 123

ВВЕДЕНИЕ

Цель освоения дисциплины является овладение студентами основными методами теории интеллектуальных систем, приобретение навыков по использованию интеллектуальных систем, изучение основных методов представления знаний и моделирования рассуждений.

Задачи освоения дисциплины – помочь студентам овладеть навыками и знаниями в области искусственного интеллекта

СТРУКТУРА И СОДЕРЖАНИЕ ПРАКТИЧЕСКИХ РАБОТ

Лабораторная работа 1.
Анализ современных программных средств с применением ИИ

Цель занятия – сформировать представление об современных программных средств с применением ИИ.

Задачи занятия:

- изучить литературные источники и материалы из сети Интернет, связанные с нормативно-правовыми основами развития искусственного интеллекта в России и мире;
- проанализировать примеры применения технологий ИИ в образовательной сфере.

Задания для выполнения и методические рекомендации:

Задание 1.

Изучив литературные источники и материалы из сети Интернет, заполните таблицу «Нормативно-правовые основы развития искусственного интеллекта в России»:

№ п/п	Документ	Дата принятия	Ссылка на документ	Сроки реализации (если имеются)	Цели и задачи	Ответственные в реализации	Основные пункты
1.	«Национальная стратегия развития искусственного интеллекта на период до 2030 года»						
2.	…						

Задание 2.

Подготовьте сообщение (текст и презентацию) о любом из перечисленных в таблице Задания 1 документов. Будьте готовы рассказать его группе.

Задание 3.

Изучив литературные источники и материалы из сети Интернет, заполните следующую таблицу «Нормативно-правовые основы развития искусственного интеллекта в мире» (привести не менее 7 стран):

№ п/	Страна	Документ	Дата принятия	Сроки реализации (если	Цели и задачи

п				имеются)	
1.		1.1.			
		1.2.			
		…			
2.	…				

Задание 4.

Изучив литературные источники и материалы из сети Интернет, заполните следующую таблицу «Примеры технологий ИИ в моей профессиональной сфере» (привести 5- 7 технологий):

№ п/п	Название технологии	Разработчик технологии	Время появления технологии	Описание возможностей технологии	Примеры использования технологии в профессиональной сфере
1.					
2.					

Задание 5.

Подготовьте сообщение (текст и презентацию) о любом из перечисленных в таблице Задания 4 технологий. Будьте готовы рассказать его группе.

Сделайте **вывод** о проделанной работе.

Рекомендуемая литература:

1. Пальмов, С.В.
    Интеллектуальные системы и технологии Электронный ресурс : учебное пособие / С.В. Пальмов. - Самара : Поволжский государственный университет телекоммуникаций и информатики, 2017. - 195 с. - Книга находится в базовой версии ЭБС IPRbooks.

2. Сысоев, Д. В. Введение в теорию искусственного интеллекта : учебное пособие / Д. В. Сысоев, О. В. Курипта, Д. К. Проскурин. — Москва : Ай Пи Ар Медиа, 2021. — 170 с. — ISBN 978-5-4497-1092-5. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/108282.html

8.1.2. Перечень дополнительной литературы:

1. Аверченков, В. И. Система формирования знаний в среде Интернет : Монография / Аверченков В. И. - Брянск : Брянский государственный технический университет, 2012. - 181 с. - Книга находится в базовой версии ЭБС IPRbooks. - ISBN 5-89838-328-X

2. Павлов, С. И. Системы искусственного интеллекта : учебное пособие : [16+] / С. И. Павлов. – Томск : Томский государственный университет систем

управления и радиоэлектроники, 2011. – Часть 1. – 175 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=208933

Интернет-ресурсы:
http://www.iprbookshop.ru - ЭБС IPRbooks
http://biblio-online.ru/ - ЭБС «Biblio-online.ru» издательства «Юрайт» ONLINE»
http://www.intuit.ru - Интернет-университет технологий

Лабораторная работа 2
Знакомство с принципами функционирования искусственных нейронных сетей и с языком программирования Python

Цель: рассмотреть принципы функционирования искусственных нейронных сетей и познакомиться с языком программирования Python.

Задание:

1. Изучить теоретическое введение.
2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.
3. Выполнить практические задания с использованием языка программирования Python.
4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчёт предоставляется в электронном виде в текстовом редакторе MS Word. В отчёте указываются:

1) Фамилия студента.
2) Порядковый номер и название лабораторной работы.
3) Цель работы.
4) Ответы на контрольные вопросы.
5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчёту, представленному студентом и с демонстрацией задания на компьютере.

Теоретическое введение

1. Введение
2. Принципы функционирования искусственных нейронных сетей

1. Введение

Компьютеры в общем смысле – это 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. Реконструкция видеокадра, состоящего сплошь из единиц и нулей, которые поступают на ваш компьютер по сети, также осуществляется путем выполнения арифметических действий, лишь ненамного более сложных, чем суммирование чисел.

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.

Посмотрев на какие-либо изображения, человек может легко определить, что на них изображено. Мы способны практически мгновенно и с высокой точностью распознавать объекты, на которые направляем свой взгляд, и при этом очень редко ошибаемся.

Котик	Гитара	Компьютер

В процессе анализа изображений и классификации объектов наш мозг обрабатывает огромные объемы информации. Компьютеру же 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

Принципы функционирования искусственных нейронных сетей

Представьте 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, поэтому будем использовать 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 “3x4” , обрабатывается с возможной заменой операции умножения более простыми операциями сложения, и выдается выходной результат “12”.

Данный простой и хорошо знакомый пример используем для введения понятий, которые далее будут применены к более сложным аспектам нейронных сетей.

Далее 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.

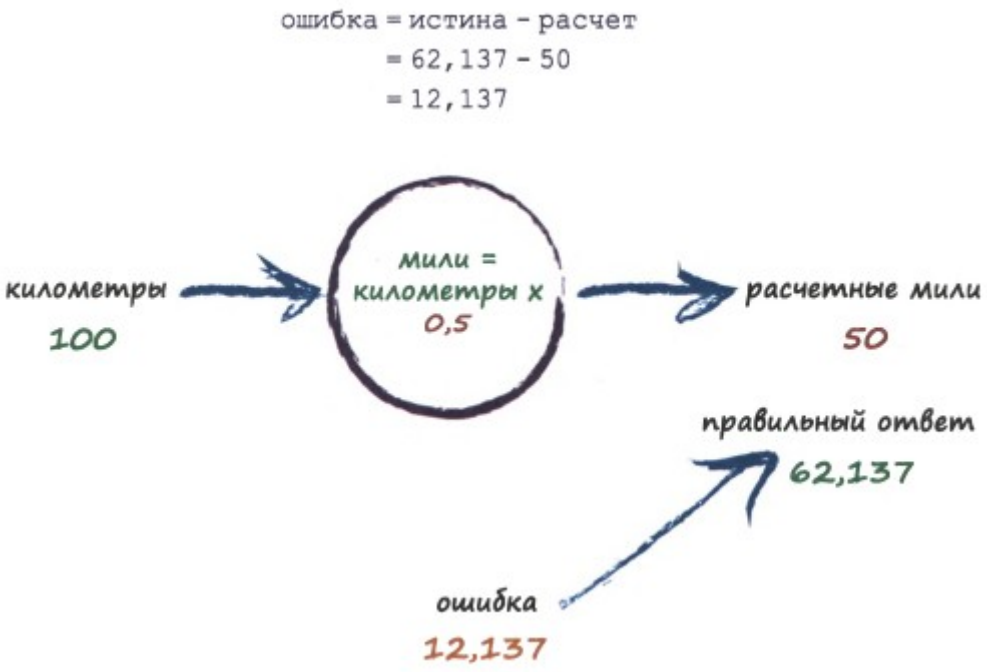
Пример	Километры	Мили
1	0	0
2	100	62,37

Что 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.

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.

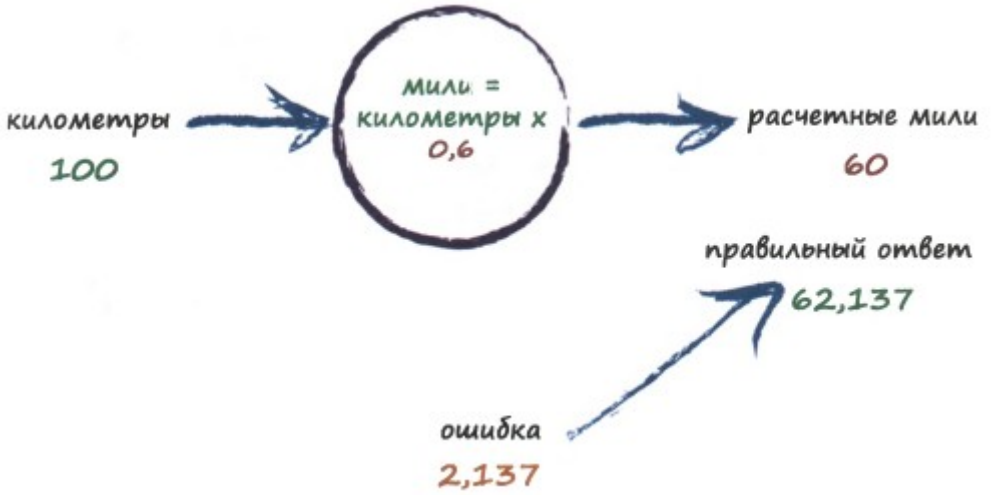


Что дальше? Мы знаем, что ошиблись, и нам известна величина ошибки. Используем эту информацию для того, чтобы предложить более удачное пробное значение константы с, чем первое.

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

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

Приняв для с значение 0,6, мы получаем мили = километры * с = 100 * 0, 6 = 60. Это уже лучше, чем предыдущий ответ — 50. Теперь ошибка уменьшилась до 2,137.

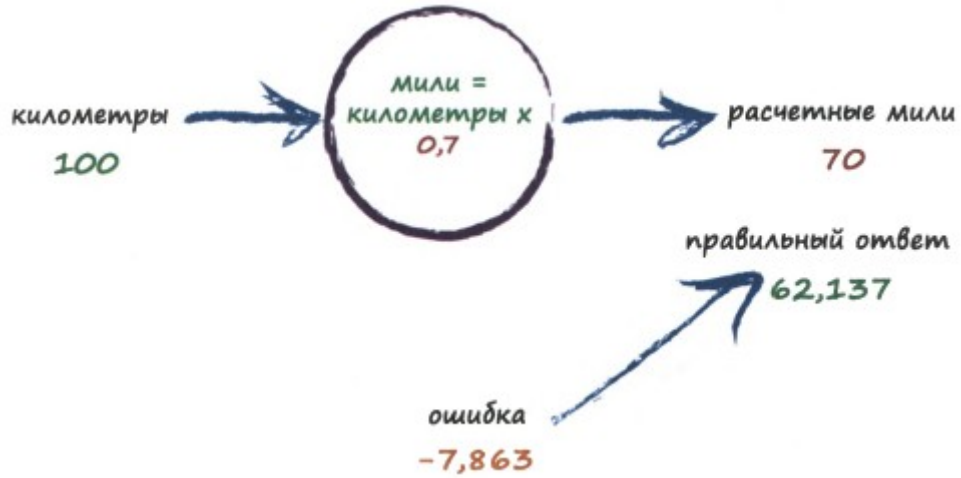


Здесь важно то, что величина ошибки подсказала нам, в каком направлении следует откорректировать величину с. Мы хотели увеличить выходной результат 50, поэтому немного увеличили с.

Вместо того чтобы использовать алгебру для нахождения точной величины поправки, на которую следует изменить значение с, мы продолжим использовать подход, заключающийся в постепенном уточнении значения этой константы. Если

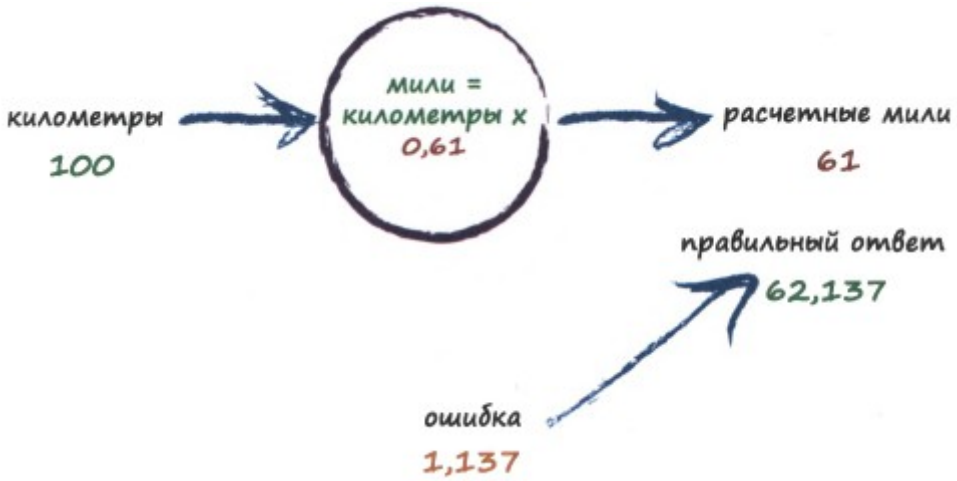
вы не уверены в правильности такого подхода и считаете, что было бы намного проще сразу определить точный ответ, то имейте в виду, что существует множество более интересных задач, для которых не существует простых математических формул, связывающих между собой входные и выходные значения. Именно поэтому 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 с, увеличив ее значение с 0,6 до 0,7.



Данное изменение приводит к результату, превышающему 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 (величина ошибки определяется выражением: правильное значение минус расчетное значение).

Итак, с=0,6 было гораздо лучше, чем с=0,7. Сейчас мы могли бы признать величину ошибки при с=0,6 удовлетворительной и закончить это упражнение. Однако попытаемся ввести очень малую поправку и увеличить значение с с 0,6 до, скажем, 0,61.



Получается 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 с 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.

Не используя поиск точных способов определения величины с 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.

Весь описанный процесс работы в точности передает суть процесса обучения нейронной сети. Мы тренировали машину так, чтобы ее предсказания становились все более и более точными.

Выводы

• У многих компьютерных систем имеются каналы ввода и вывода, между которыми над данными выполняются некоторые вычисления. В случае нейронных сетей имеет место быть другая структура работы.

• 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.

Практические задания

1. Знакомство с Jupyter Notebook
2. Начало работы с Python
3. Многократное выполнение команд
4. Создание комментариев

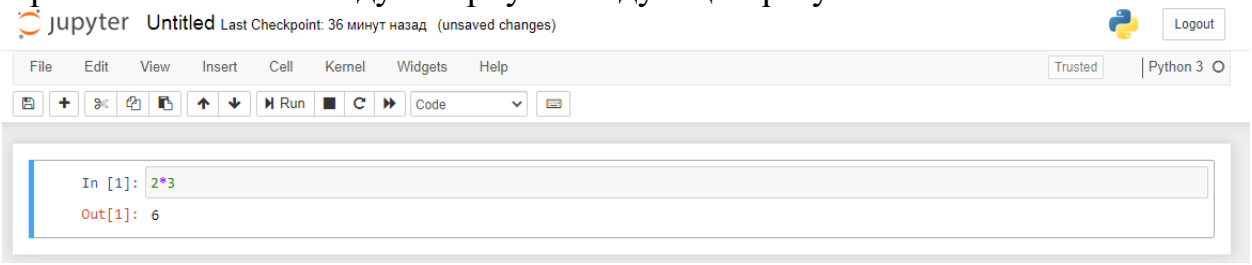
1. Знакомство с Jupyter Notebook

Запустив интерактивную оболочку Jupyter Notebook (Блокнот), щелкните на кнопке New у правого края окна, выберите в открывшемся меню пункт **Folder,** в названии папки укажите **свою фамилию.** Затем перейдите в созданную папку, щелкните на кнопке New у правого края окна и выберите **Python 3**, что приведет к открытию пустого **блокнота.**

Блокнот интерактивен, т.е. ожидает ввода команды, выводит ответ и вновь переходит в состояние ожидания следующей команды или вопроса.

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 команду 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[l]’, которыми в IPython помечаются инструкция и ответ. Так оболочка напоминает 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.

2. Начало работы с Python

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 этот способ более удобен.

print("Hello, World!")

В ответ компьютер должен просто вывести в окне фразу «Hello, World!»

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

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.

Значение «10» остается в хранилище до тех пор, пока в нем не возникнет необходимость. Инструкция print (x), с которой вы уже сталкивались, выводит информации на экран. 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 должна быть понятной, если вспомнить, 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, информирующее о некорректности предпринимаемых действий и пытающееся быть максимально полезным, чтобы можно было устранить ошибку. 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 сообщение об ошибке «name V is not defined» (имя ‘z’ не определено), можно увидеть на следующей иллюстрации.

```
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-e80d55263555> in <module>
              4 y = x+7
              5 print(y)
        ----> 6 print (z)

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

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.

3. Многократное выполнение команд

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

Посмотрим, как вывести квадраты первых десяти натуральных чисел, начиная с нуля: 0 в квадрате, 1 в квадрате, 2 в квадрате и т.д. В результате должен получиться 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. Также нам необходимо создать типовой набор инструкций, позволяющий выводить квадраты чисел в любом заданном диапазоне. Для этого нужно сначала рассмотреть несколько новых особенностей работы данного языка программирования.

Введите в очередную пустую ячейку следующий код:

list(range(10))

В итоге получается список из десяти чисел от 0 до 9.

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

In [ ]:
```

Список содержит числа от 0 до 9, а не от 1 до 10. Это не случайно. 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.

В данном случае мы опустили ключевое слово print, которое использовали при выводе фразы «Hello, World!». Использование ключевого слова print не является обязательным при работе с Python в интерактивном режиме, поскольку оболочка знает, что мы хотим увидеть результат введенной инструкции.

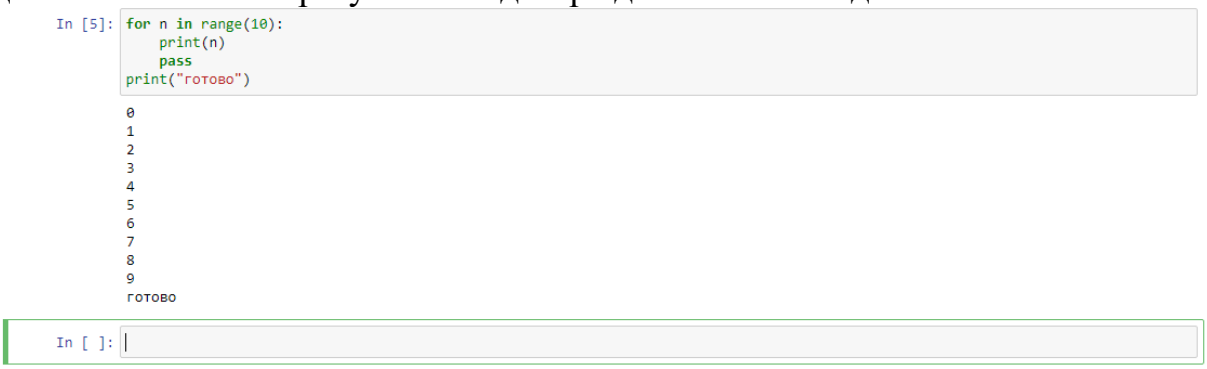
Распространенным способом многократного выполнения вычислений с помощью компьютера является использование так называемых циклов. Слово «цикл» правильно передает суть происходящего, когда некоторые действия повторяются снова и снова, возможно, даже бесконечное число раз. Вместо того чтобы приводить формальное 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 разберем. 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.

Смысл инструкции print (n) в следующей строке, которая просто выводит текущее значение n, уже должен быть понятен и знаком. 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.



Теперь 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 [18]: 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

In [ ]:
```

Точно так же можно увеличить число итераций до пятидесяти или тысячи с помощью выражений range (50) или range (1000). Попробуйте сделать это самостоятельно.

4. Создание комментариев

Рассмотрим, как задать комментарии в Python. Наберите 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.

```
In [19]: # следующая инструкция выводит куб числа 2
         print(2**3)

8

In [ ]:
```

Контрольные вопросы

1. В чем заключаются особенности вычислительных операций, выполняемых компьютером?
2. Кратко опишите принципы функционирования искусственных нейронных сетей.
3. С помощью какой команды можно вывести на печать Вашу фамилию на языке Python?
4. Как вывести список чисел от 0 до 20 на языке Python?
5. Как задаются комментарии на языке Python?

Лабораторная работа 3

Принципы классификации образов с помощью искусственных нейронных сетей. Функции, массивы и объекты в Python.

Цель: изучить принципы классификации образов с помощью искусственных нейронных сетей и рассмотреть функции, массивы и объекты в Python.

Задание:

1. Изучить теоретическое введение.

2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.

3. Выполнить практические задания с использованием языка программирования Python.

4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчѐт предоставляется в электронном виде в текстовом редакторе MS Word. В отчѐте указываются:

1) Фамилия студента.

2) Порядковый номер и название лабораторной работы.

3) Цель работы.

4) Ответы на контрольные вопросы.

5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчѐту, представленному студентом и с демонстрацией задания на компьютере.

Теоретическое введение

1. Задача классификации с помощью искусственных нейронных сетей

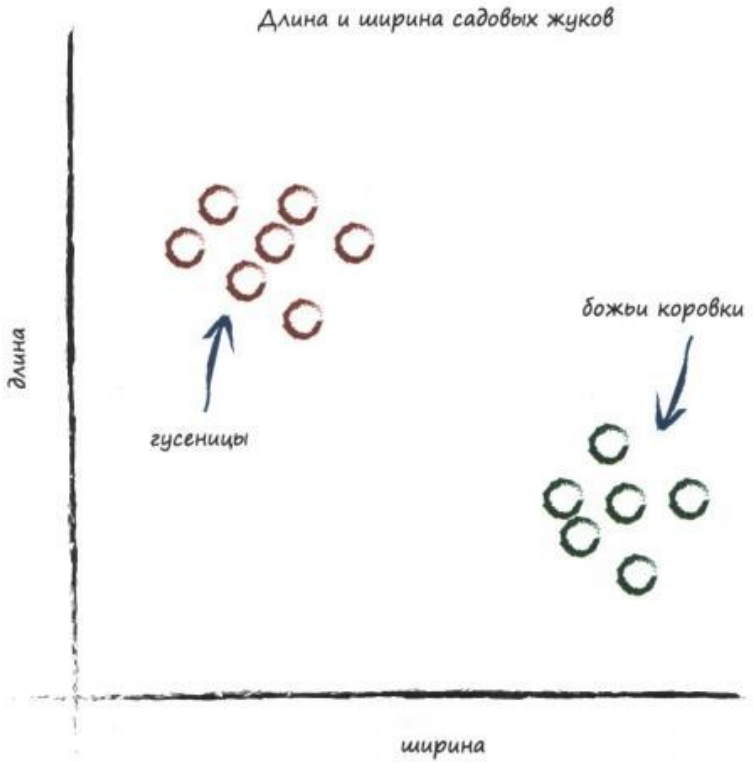
2. Тренировка простого классификатора

Задача классификации с помощью искусственных нейронных сетей

В прошлой лабораторной работе в теоретической части мы описали функционирование простой прогнозирующей машины. 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.

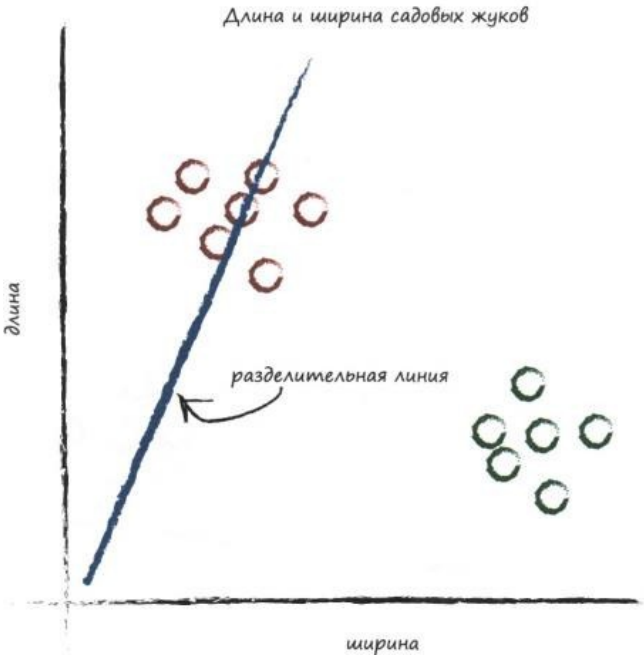
Далее на простом примере рассмотрим **задачу классификации** с помощью искусственных нейронных сетей.

Приведем диаграмму, представляющую результаты измерения размеров садовых жуков.



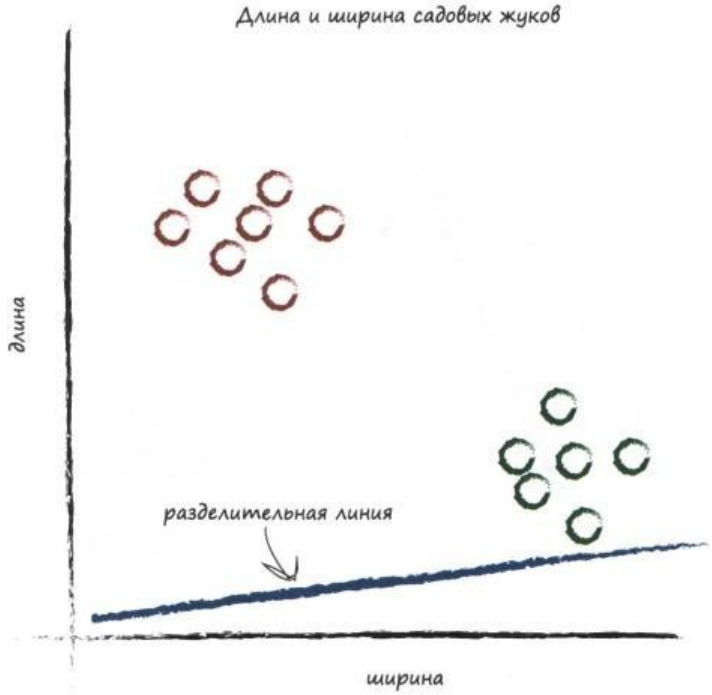
На диаграмме отчетливо видны две группы данных. Гусеницы *уже* и длиннее, а божьи коровки шире и короче. Обратимся к прогнозирующей машине или предиктору, принципы функционирования которого мы рассмотрели в предыдущей работе. В основу этого предиктора была положена настраиваемая линейная функция. График 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?

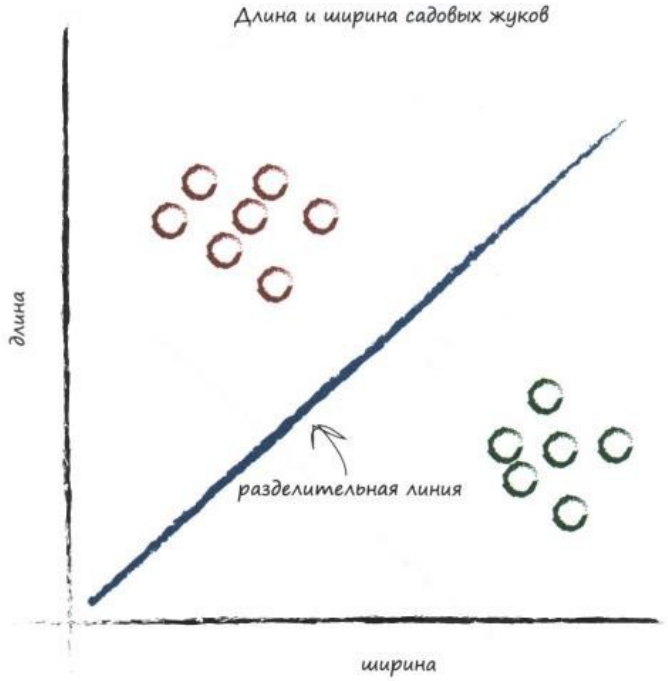


Мы не можем использовать прямую линию точно так же, как раньше, когда преобразовывали одно число (километры) в другое (мили), но,

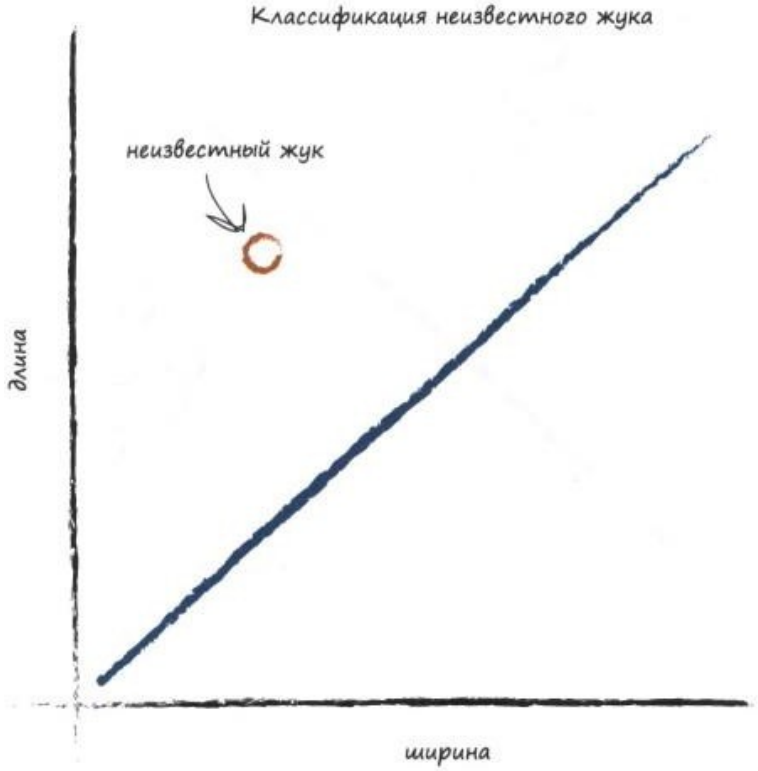
возможно, в данном случае нам удастся отделить с ее помощью один тип данных от другого.
Если бы на показанном выше графике прямая линия отделяла гусениц от божьих коровок, то мы могли бы воспользоваться ею для классификации неизвестных жуков, исходя из результатов измерений. Однако имеющаяся линия не 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 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.
Таким образом мы убедились 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 займемся тренировкой (обучением) 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.

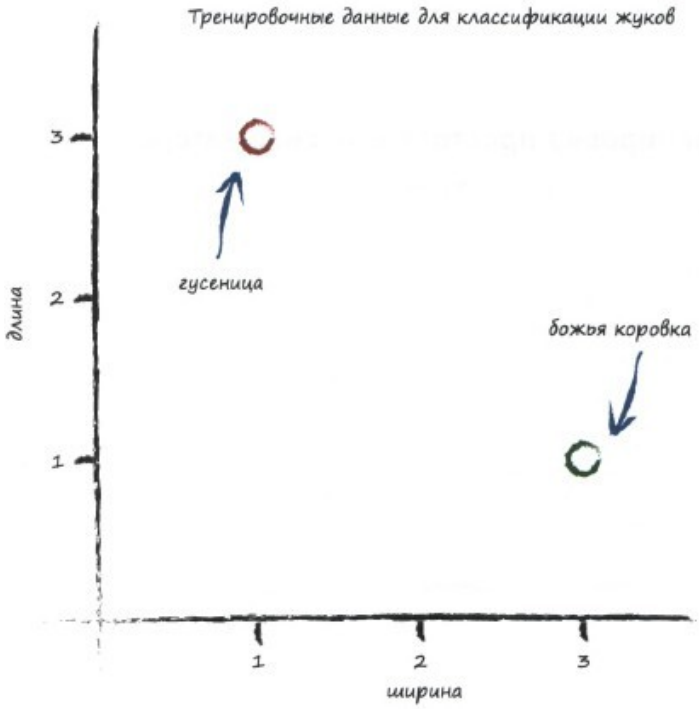
Нам нужны примеры для тренировки классификатора. Чтобы не усложнять себе жизнь, мы ограничимся двумя простыми примерами, приведенными ниже.

Пример	Длина	Ширина	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 большую 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.



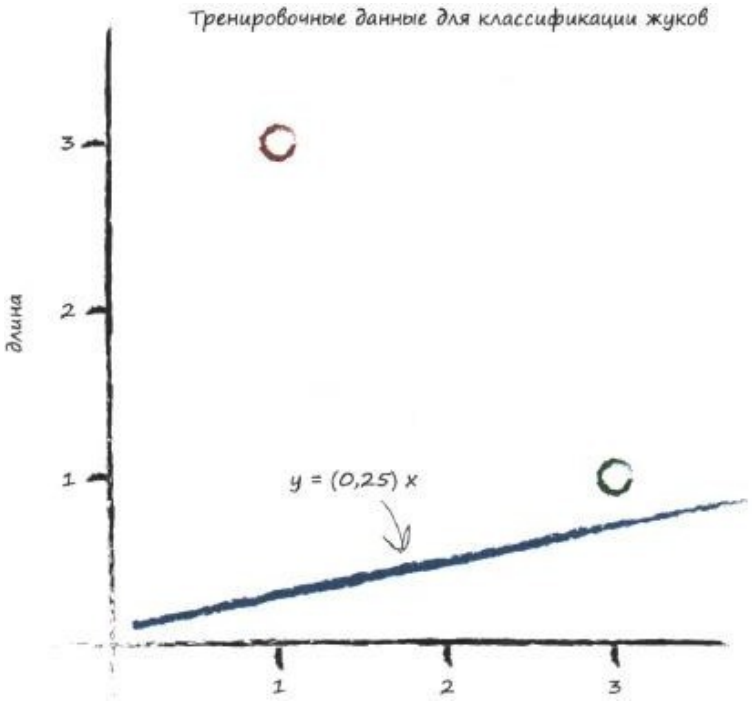
Начнем со случайной разделительной линии, потому что нужно ведь с чего-то начинать. Вспомните линейную функцию из примера с преобразованием километров в мили, параметр которой мы настраивали. Мы можем сделать то же самое и сейчас, поскольку разделительная линия в данном случае прямая:

$y = Ax$

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. Проводить подходящую линию вручную – не самая лучшая идея. Удастся ли нам подобрать для этого подходящее решение с использованием повторяющейся последовательности 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:

У = (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? 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.

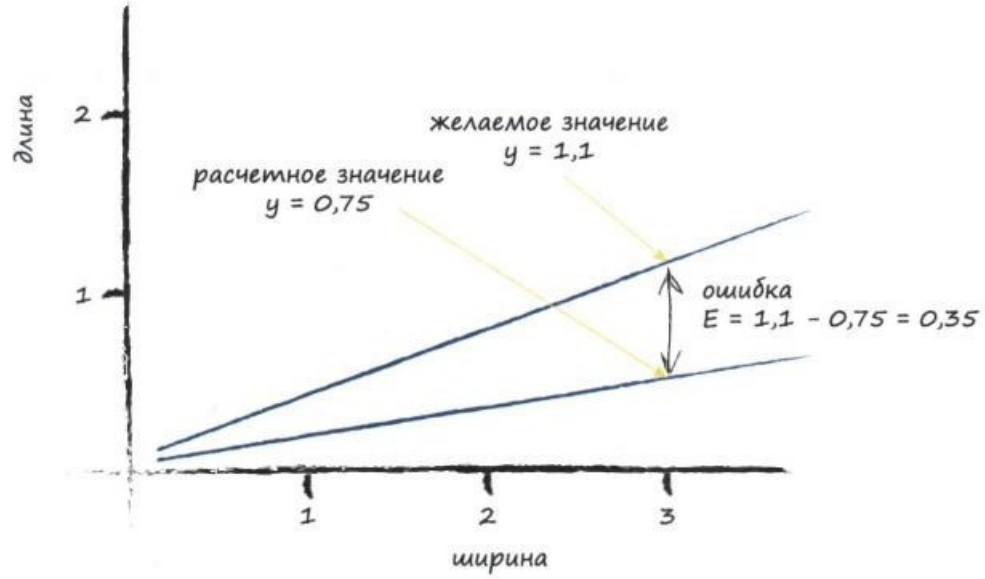
В связи с этим попробуем применить значение y=1,1 при x=3,0. Оно лишь не намного больше 1,0. Вместо него можно было бы взять значение 1,2 или 1,3, но никак не 10 или 100, поскольку с большой долей вероятности это привет к тому, что прямая будет проходить над всеми точками данных, как божьих коровок, 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

Обратимся к приведенной ниже диаграмме, на которой в графическом виде представлены ошибка, а также целевое и расчетное значения.



Каким образом наше знание величины ошибки *E* может помочь в нахождении лучшего значения для параметра *A*?

Давайте на время отступим от этого вопроса 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:

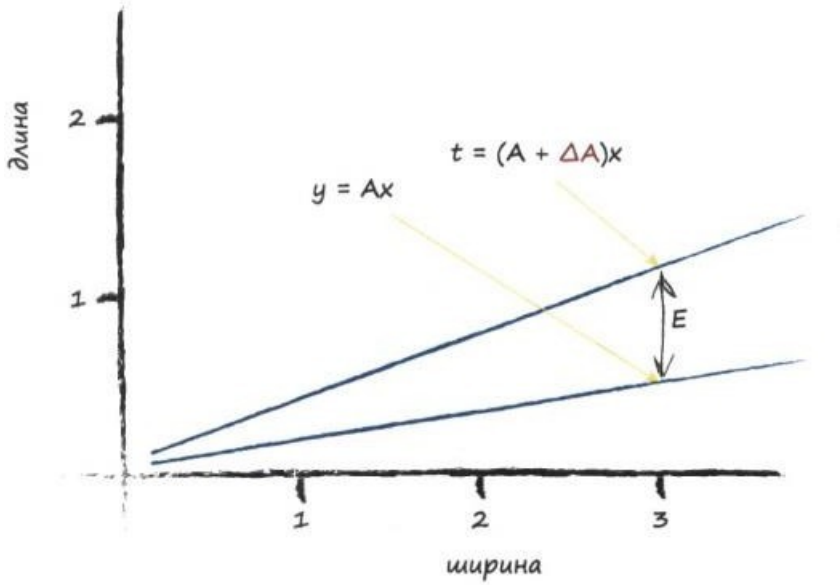
y = Ax

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 + ΔA) x - Ax

Раскрыв скобки и приведя подобные члены, получаем:

E = t - y = A x + (ΔA) x - Ax E = (ΔA)x

Следовательно, ошибка *E* связана с *ΔA* очень простым соотношением.

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*:

$$\Delta A = E / x$$

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

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.

Все работает, и мы располагаем методом для улучшения параметра *A*, если известна текущая ошибка.

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

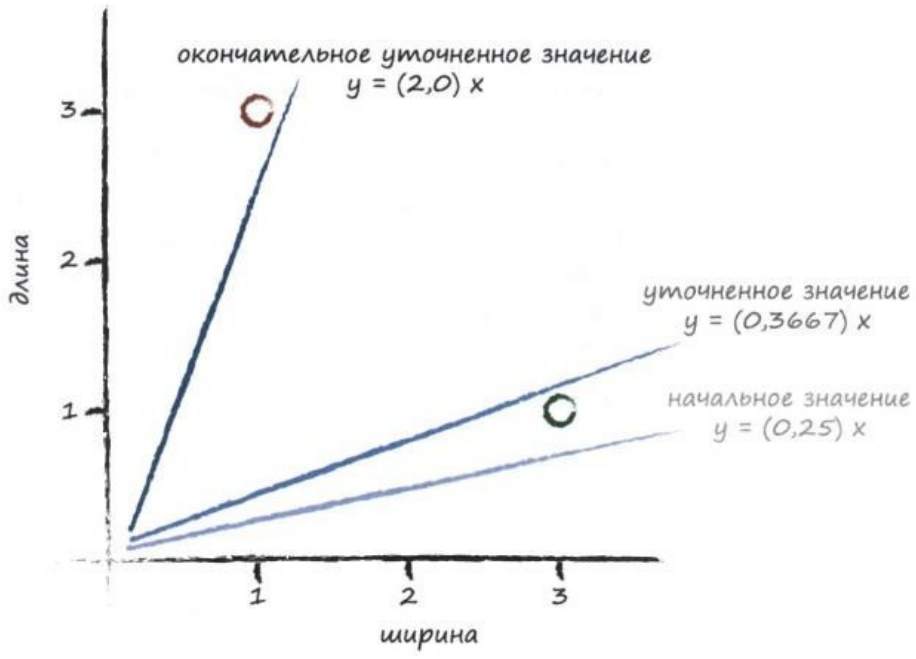
Посмотрим, что получится, если вставить x = 1,0 в линейную функцию, в которой теперь используется обновленное значение A = 0,3667. Мы получаем y = 0,3667 * 1,0 = 0,3667. Это очень далеко от значения *y* = 3,0 в тренировочном примере.

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 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 $\Delta A = E / x$ 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.

Теперь мы можем 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.



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. Однако, линия обновляется, подстраиваясь под то целевое значение *y*, которое мы задаем.

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? Мы сглаживаем обновления, т.е. **немного** уменьшаем величину поправок. Вместо того чтобы каждый раз 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 с 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.

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.

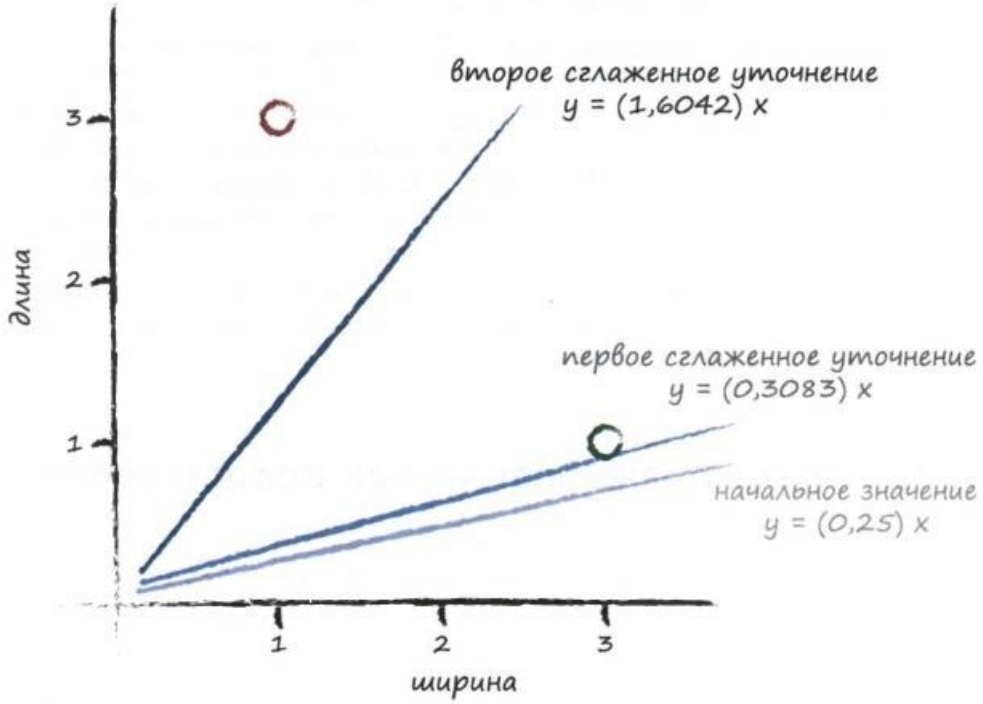
Повторим все расчеты, используя начальное значение A*0,25. Первый тренировочный пример дает нам y = O,25*3,0 = 0,75. При целевом значении 1,1 ошибка равна 0,35. Поправка равна ΔA = L (E / x) =0,5 * 0,35 / 3,0 = 0,0583. Обновленное значение A равно 0,25 ++ 0,0583 = 0,3083.

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 , 5 9 1 7 / 1 , 0 = 1 , 2 9 5 8 . 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.

Таким образом, нам удалось создать автоматизированный метод обучения классификации на примерах, который, несмотря на простоту подхода, продемонстрировал замечательную эффективность.

Выводы:

- 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.

Практические задания

1. Функции
2. Массивы
3. Графическое представление массивов
4. Объекты

1. Функции

Ранее мы работали с математическими функциями. Данные функции 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, гораздо проще, чем каждый раз полностью записывать образующий эту функцию код.

Для задания функции необходимо четко определить, *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.

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 равно', x)
print('второе число равно', 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 напомнить об использовании переменной не по назначению, например, когда вы пытаетесь использовать слово, как будто оно является числом и т.д.

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 иногда забывают о важности отступов.

Понять смысл кода в определении функции среднее (x, y) будет совсем несложно, поскольку в нем используется все то, с чем мы 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.

```
In [7]: # функция, которая принимает два числа в качестве входных данных
        # и выводит их среднее значение
        def среднее(x,y):
            print('первое число равно', x)
            print('второе число равно', y)
            a = (x + y) /2.0
            print('среднее значение равно', a)
            return a
```

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.

```
In [8]: среднее (2, 4)

        первое число равно 2
        второе число равно 4
        среднее значение равно 3.0

Out[8]: 3.0
```

Поэкспериментируйте самостоятельно, вызывая функцию с различными входными значениями.

```
In [9]: среднее (200, 301)

        первое число равно 200
        второе число равно 301
        среднее значение равно 250.5

Out[9]: 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. 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. 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.

Итак, 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. Об этом в следующих работах.

2. Массивы

Массивы – это не более чем таблицы значений. 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 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 3x2, 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 [10]: import numpy

In [11]: 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 [16]: 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 известно, как присваивать значения элементам массива, рассмотрим, каким образом можно узнать значение отдельного элемента, не выводя на экран весь массив. 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 [17]: 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]. Если мы попытаемся, к примеру, обратиться к элементу массива *a[0,2]*, то получим сообщение об ошибке.

```
In [18]: #попытка обращения к несуществующему элементу массива
         a[0,2]

         ---------------------------------------------------------------------------
         IndexError                                Traceback (most recent call last)
         <ipython-input-18-f5ac02a5343a> 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.

3. 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 x 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.

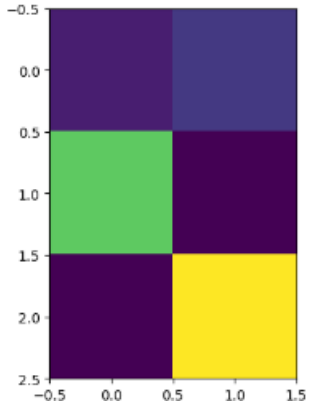
Кроме того, мы должны дополнительно сообщить IPython о том, что любую графику следует отображать в блокноте, а не в отдельном окне. Это делается с помощью следующей директивы:

```
%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")
```

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

[10]: import matplotlib.pyplot
      %matplotlib inline

[11]: matplotlib.pyplot.imshow(a, interpolation="nearest")

[11]: <matplotlib.image.AxesImage at 0x36e71c0>
```



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. Функция imshow () предлагает множество опций графического представления данных, таких как использование различных цветовых палитр.

4. Объекты

Далее 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.

Запишите 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!” 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 к простому, но реальному коду, который оживляет эти понятия.

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

В первой строке создается переменная sizzles, источником которой является вызов функции. Dog () – это особая функция, которая создает экземпляр класса Dog. Таким образом можно 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(). 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!”

```
[13]: # класс объектов Dog (собака)
      class Dog:
      # собаки могут лаять
          def bark(self):
              print("Гав!")
              pass
          pass

[14]: sizzles = Dog()

[15]: sizzles.bark()
      Гав!
```

Возможно, вы обратили внимание на элемент self в определении функции – bark (self). Он здесь для того, чтобы указывать Python, к какому объекту приписывается функция при ее создании.

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

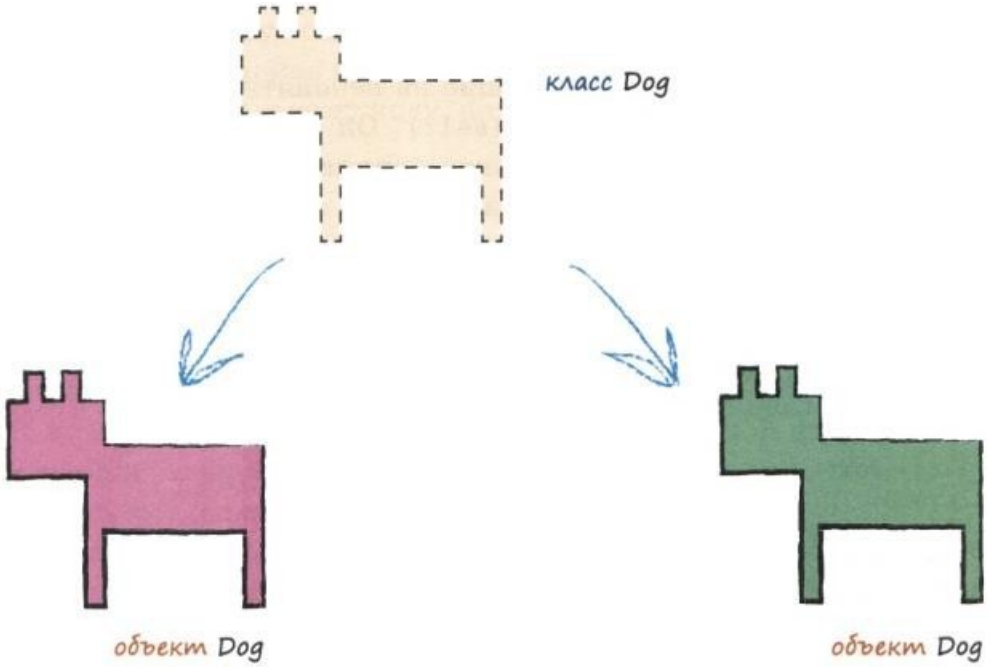
```
sizzles = Dog () mutley =
Dog()

sizzles.bark()
mutley.bark()
```

```
[16]: sizzles = Dog ()
      mutley = Dog()

      sizzles.bark()
      mutley.bark()
```

```
Гав!
Гав!
```

Мы создали два объекта: sizzles и mutley. **Важно понимать, что оба они были созданы на основе одного и того же определения класса Dog. 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. 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.

Рассмотрим ниже обновленное определение класса Dog. Оно включает ряд новых элементов, которые мы представим 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, обратим 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.
Рассмотрим переменные, указанные в круглых скобках после имен новых функций.,например, функция *setTemperature* фактически определена как *setTemperature (self, temp)*. Эти переменные, получения значений которых функции ожидают во время вызова, называются **параметрами**.
Вспомним 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 названием *__init_()*. 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. Что же именно происходит в 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. Поначалу это может показаться слишком сложным, однако, все значительно упростится, когда мы приступим к рассмотрению конкретного примера.
Следующая – 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. По замыслу разработчиков это должно напоминать Python, что функция, которую вы собираетесь определить, принадлежит объекту, ссылкой на который служит *self* .
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. 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.

```
[19]: # 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
```

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

```
[22]: lassie.status()
имя собаки:  Lassie
температура собаки:  37
```

Попытаемся изменить температуру объекта и проверим, действительно ли она изменилась, введя следующий код.
lassie.setTemperature(40)
lassie.status()
Результат представлен ниже.

```
[22]: lassie.status()
имя собаки:  Lassie
температура собаки:  37

[24]: lassie.setTemperature(40)
lassie.status()
имя собаки:  Lassie
температура собаки:  40
```

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

Контрольные вопросы

1. В чем заключается задача классификации с помощью искусственных нейронных сетей?
2. Что представляют собой тренировочные данные?
3. Что понимают под сглаживанием при обучении искусственных нейронных сетей? С какой целью оно применяется?
4. Приведите этапы обучения сети в задаче классификации.
5. Что такое функция? Как задаются функции на языке Python?
6. Что включает и для чего используется пакет *numpy*?
7. Для каких задач используется пакет *matplotlib.pyplot*?
6. Как задать массив и вывести его графическое представление на языке Python?
7. Что такое класс, объект, метод?
8. Как задаются объекты на языке Python?

Лабораторная работа 4
Решение сложных задач классификации с помощью искусственных нейронных сетей. Распространение сигналов по нейронной сети

Цель: изучить возможности искусственных нейронных сетей для решения сложных задач классификации образов.

Задание:

1. Изучить теоретическое введение.
2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.
3. Выполнить практические задания с использованием языка программирования Python.
4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчѐт предоставляется в электронном виде в текстовом редакторе MS Word. В отчѐте указываются:

1) Фамилия студента.
2) Порядковый номер и название лабораторной работы.
3) Цель работы.
4) Ответы на контрольные вопросы.
5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчѐту, представленному студентом и с демонстрацией задания на компьютере.

Теоретическое введение

1. Применение нескольких классификаторов
2. Нейроны – вычислительные машины, созданные природой
3. Распространение сигналов по нейронной сети

1. Применение нескольких классификаторов.

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 сложных и интересных задач, к которым относятся и задачи, решаемые на основе методов нейронных сетей.
Ограничения линейного классификатора продемонстрированы ниже на простом примере. От понимания сути указанных ограничений зависит один из ключевых элементов проектирования нейронных сетей, который нам позже пригодится.
Обратимся к логическим (*булевым, по имени Джорджа Буля*), функциям. Джордж Буль – математик, логик 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 десерт 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 десерт.

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

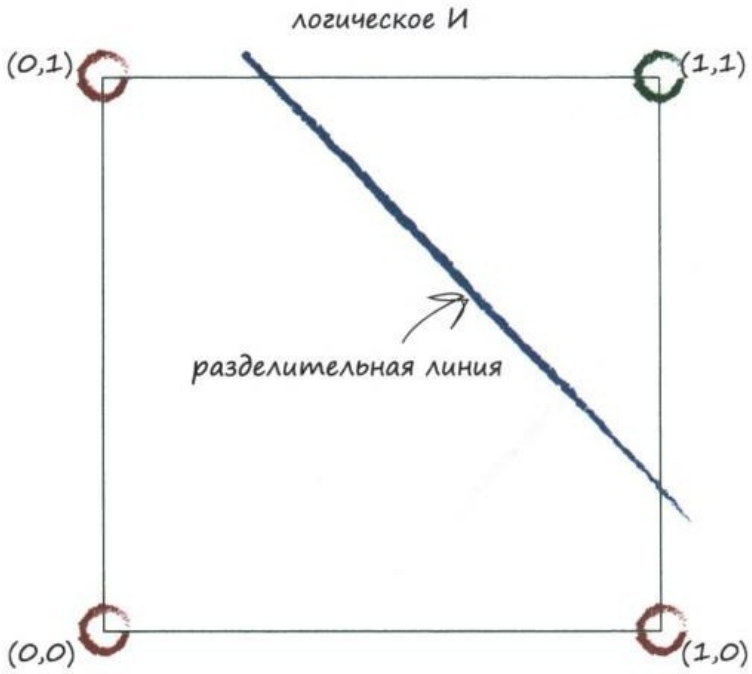
ложь – как число 0. В приведенной ниже таблице результаты работы логических функций И и ИЛИ представлены с использованием этой лаконичной нотации для всех комбинаций входных значений A и B.

Входное значение A	Входное значение B	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 A, и B.

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.

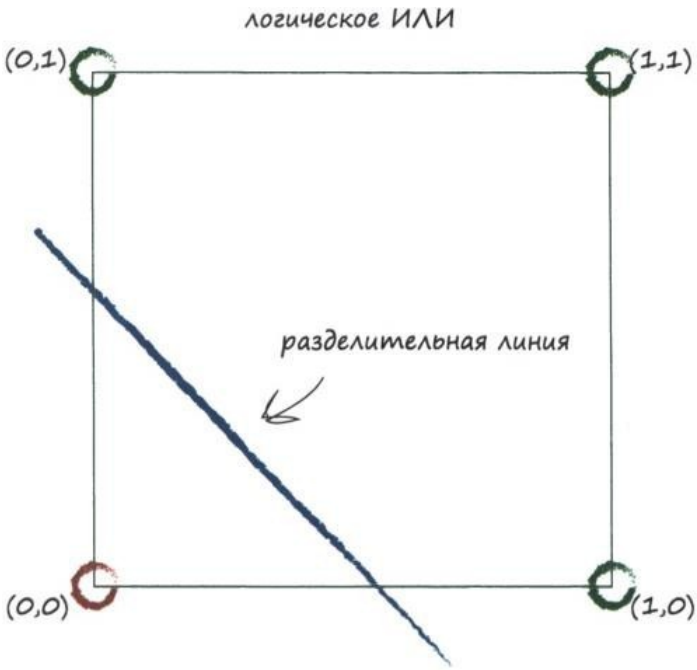
Представьте себе простой линейный классификатор, который должен использовать тренировочный набор данных для выяснения того, управляются ли данные логической функцией. Задачи такого рода естественным образом возникают перед учеными, исследующими причинно-следственные связи или корреляцию наблюдаемых величин. 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, A и B, 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 зеленымцветом, 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. Смысл 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

Диаграмма, соответствующая этой функции, будет иметь вид:

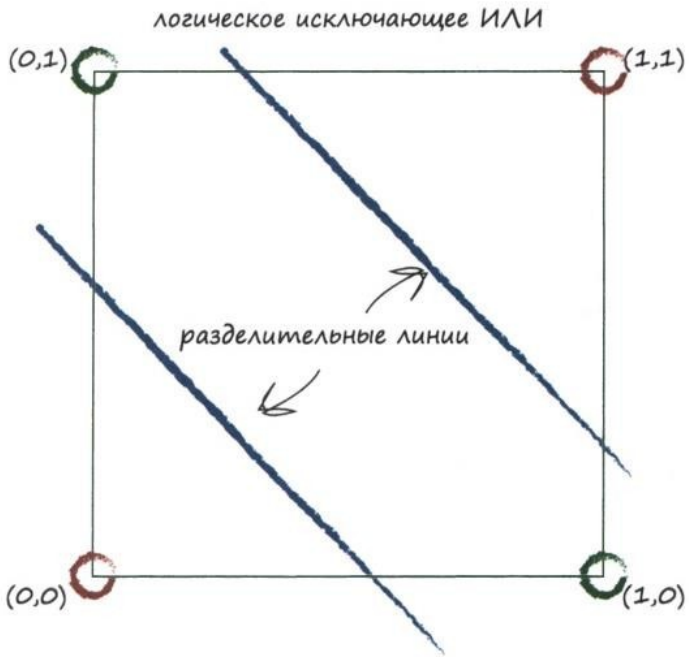
Что важно, здесь нет способа разделить зеленую и красную области одной прямой линией. Невозможно провести одну прямую линию таким образом, чтобы она успешно отделила красные точки данных от зеленых для функции *исключающего ИЛИ*. Это означает, что простой линейный классификатор не в состоянии обучиться 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 другой способ работы с данным вопросом.

Такой способ 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.

Выводы по первому пункту:

• 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.

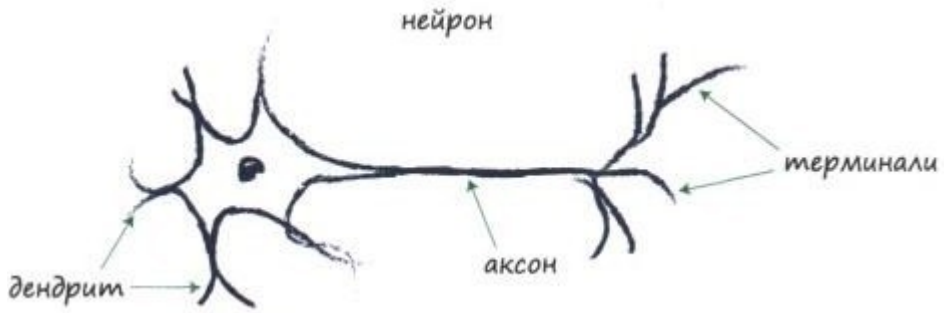
• Однако эта проблема решается очень просто: для разграничения данных, которые не удается разделить одной прямой линей, **следует использовать множество линейных классификаторов.**

2. Нейроны – вычислительные машины, созданные природой.

Мозг живых организмов, являясь сложной системой, ставит ученых в тупик, поскольку даже у столь малых представителей живой природы, как, например, попугаи, он демонстрирует несравненно большие 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.

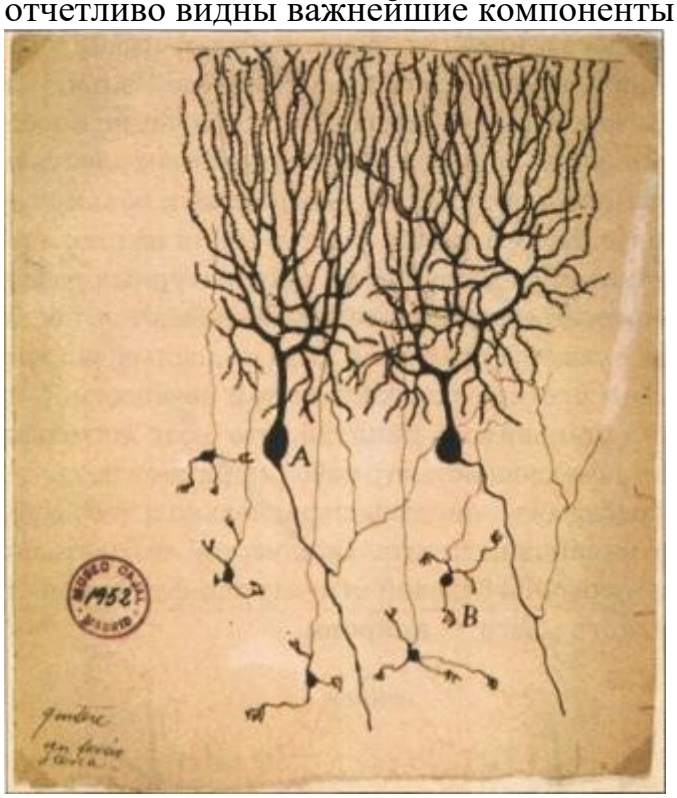
Тогда ученые сосредоточили внимание на архитектурных различиях. В традиционных компьютерах данные обрабатываются последовательно, по четко установленным правилам. В их запрограммированных расчетах нет места неоднозначности и неясности. С другой стороны, постепенно становилось понятно, что мозг животных, 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 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.

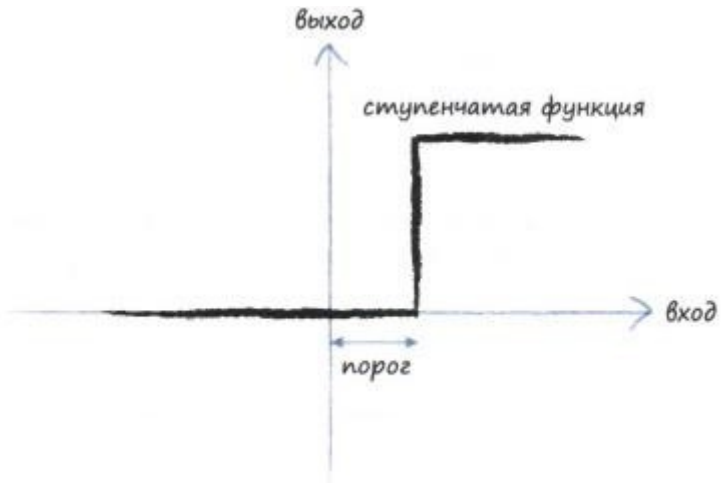
Можем ли мы представить нейроны в виде линейных функций? Нет, хотя сама по себе эта идея неплохая. Вырабатываемый нейроном выходной сигнал не является простой линейной функцией входного сигнала, т.е. выходной сигнал нельзя представить в виде *выход = (константа * вход) + (возможная другая константа).*

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 в том, что выходной сигнал вырабатывается лишь в случаях, когда амплитуда входного сигнала достигает достаточной величины, превышающей некоторый порог.

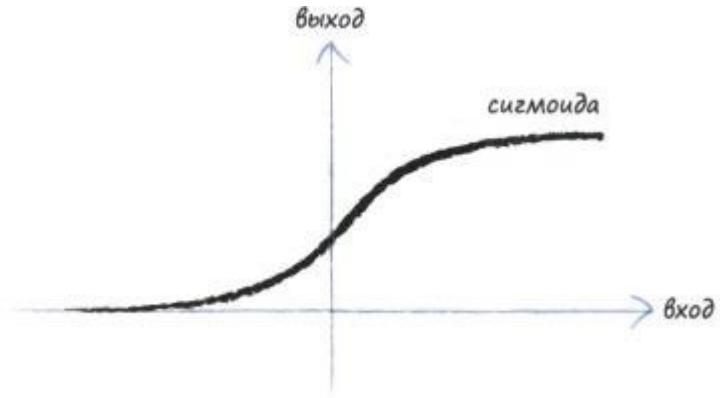
Функция, которая получает входной сигнал, но генерирует выходной сигнал с учетом порогового значения, называется функцией активации. С математической точки зрения существует множество таких функций, которые могли бы обеспечить подобный эффект.

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:

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

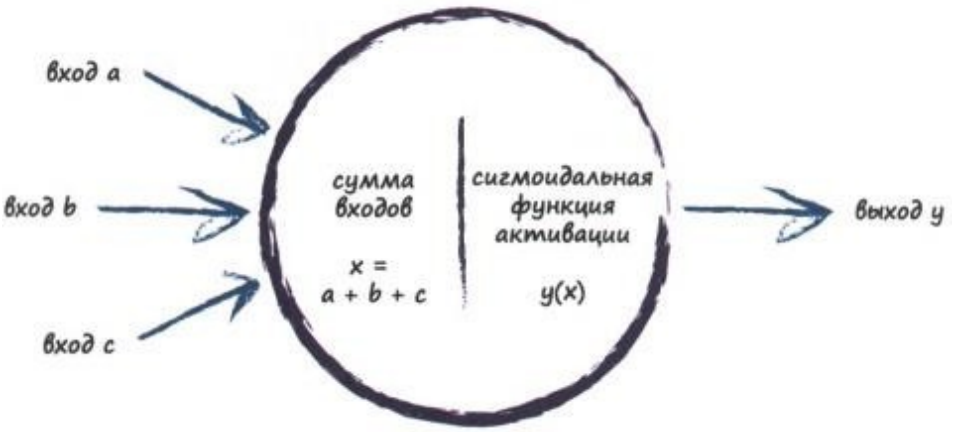
Это выражение не настолько сложное, как поначалу может показаться. Буквой e в математике принято обозначать константу, равную 2,71828... Это очень интересное число, которое встречается во многих областях математики и физики, а причина, по которой в данном числе мы использовали многоточие (...), заключается 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*.

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 на практике можно будет убедиться в том, что это действительно так.

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

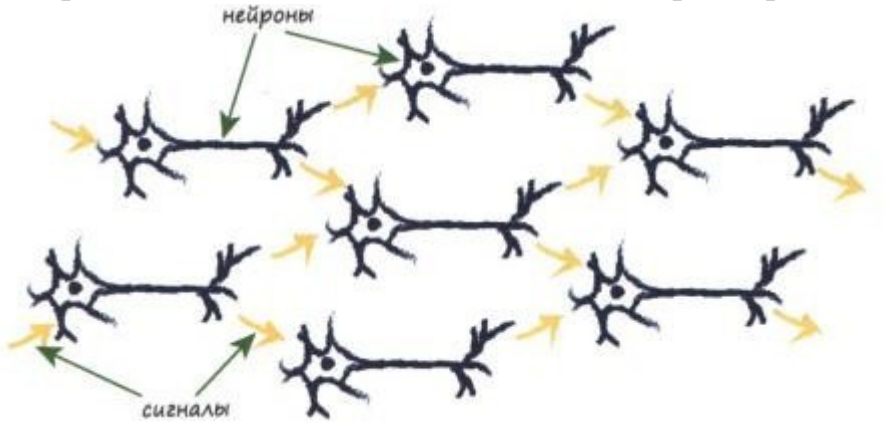
Первое, что следует понять, – это то, что реальные биологические нейроны имеют несколько входов, а не только один. Но что нам делать со всеми этими входами? Мы будем просто комбинировать их, суммируя соответствующие значения, и результирующая сумма будет служить входным значением для сигмоиды, управляющей 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 активироваться 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.

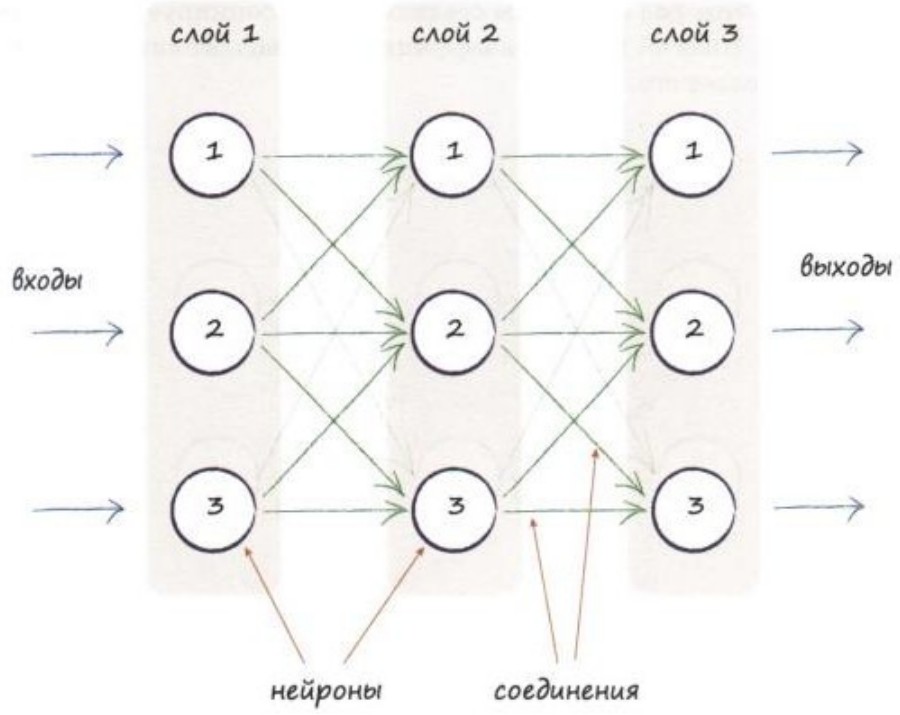
Электрические сигналы воспринимаются дендритами, где они комбинируются, формируя более сильный сигнал. Если этот сигнал превышает порог, нейрон активируется, и сигнал передается через аксон к терминалам, откуда он поступает на дендриты следующего нейрона. Связанные таким способом нейроны схематически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



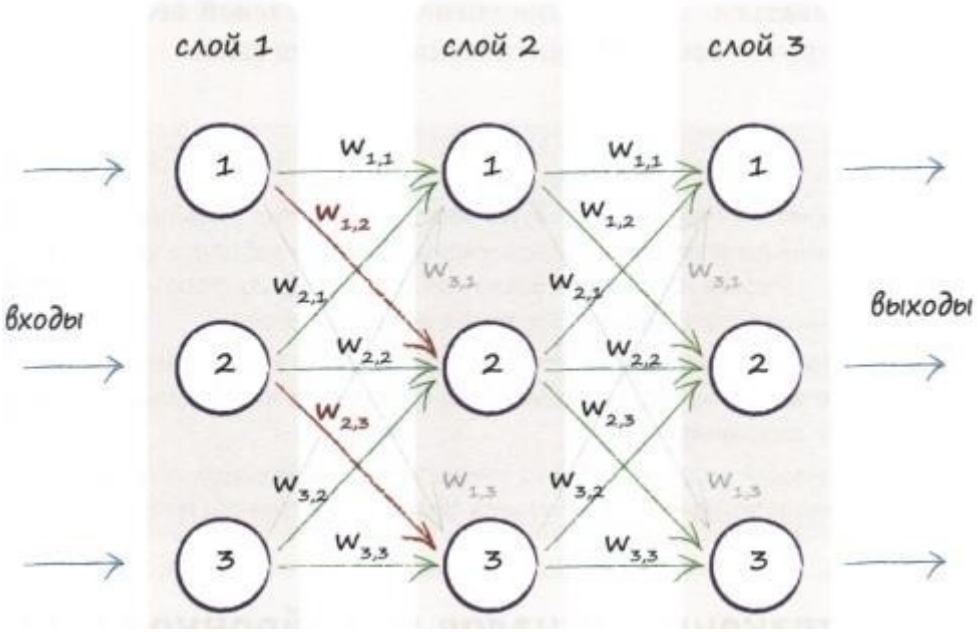
На этой иллюстрации представлены три слоя, каждый из которых включает три искусственных нейрона, или узла. Как нетрудно заметить, здесь каждый узел соединен с каждым из узлов предшествующего и последующего слоев.

Но возникает вопрос: в какой части этой застывшей структуры заключена способность к обучению? Что мы должны регулировать, реагируя на данные тренировочных примеров?

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.



Может возникнуть вопрос, 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.

Выводы по второму пункту:

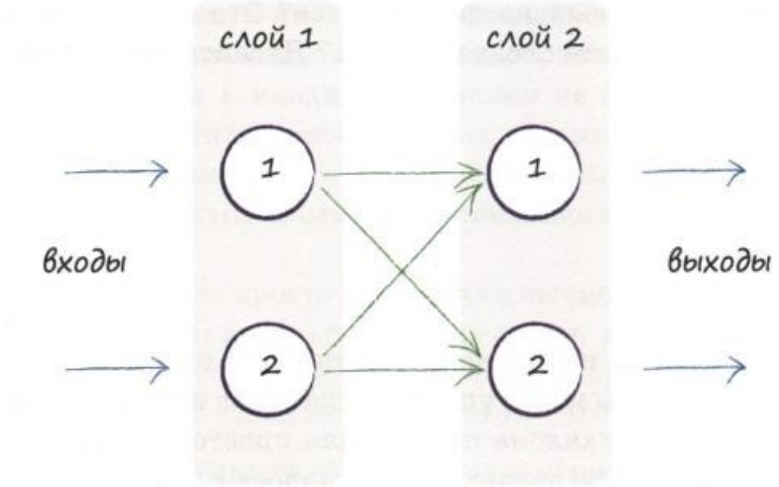
- Биологический мозг демонстрирует выполнение таких сложных задач, как управление полетом, поиск пищи, изучение языка или избегание встреч с хищниками, несмотря на меньший объем памяти и меньшую скорость обработки информации по сравнению с современными компьютерами.
- Кроме того, биологический мозг невероятно 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.

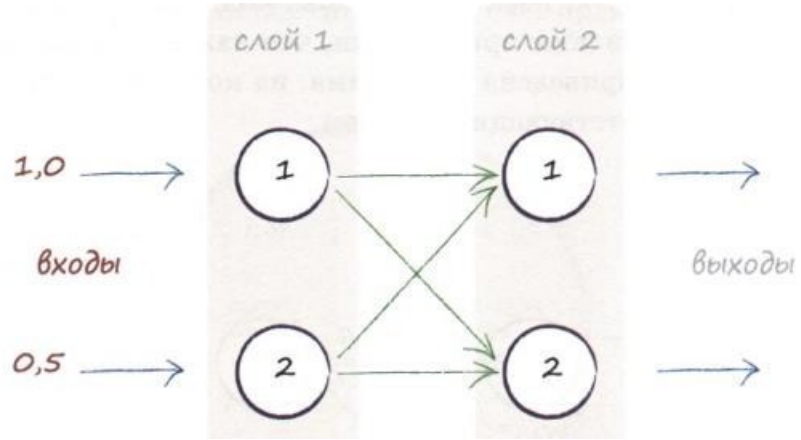
3.Распространение сигналов по нейронной сети.

Приведенная выше картинка с тремя слоями нейронов, каждый из которых связан с каждым из нейронов в предыдущем и следующем слоях, выглядит довольно просто и понятно. Однако рассчитать распространение входных сигналов по всем слоям, пока они не превратятся в выходные сигналы, не очень простая задача.

Как работает этот механизм, что происходит в нейронной сети на самом деле? Попытаемся проделать необходимые вычисления на примере меньшей нейронной сети, состоящей 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.

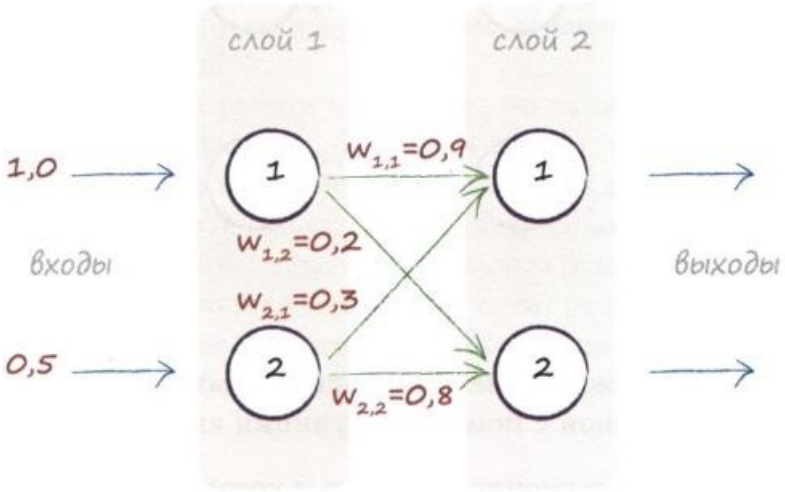


Как и раньше, каждый узел превращает сумму двух входных сигналов в один выходной с помощью функции активации. Мы также будем использовать сигмоиду $\frac{1}{1+e^{-x}}$, с которой вы до этого познакомились, где x – это сумма сигналов, поступающих в нейрон, а y – выходной сигнал этого нейрона.

А что насчет весовых коэффициентов? 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.

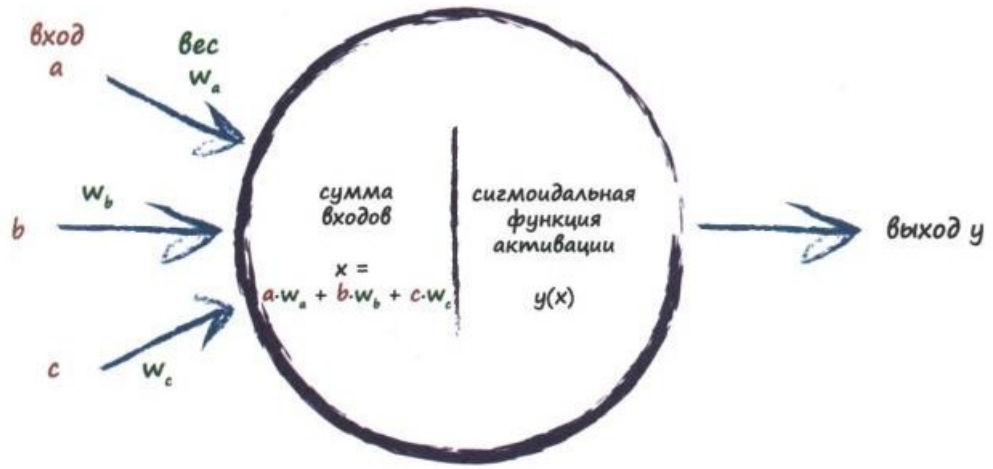


Приступим к вычислениям.

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.

С первым слоем все просто – никаких вычислений.

Далее мы должны заняться вторым слоем, в котором потребуется выполнить некоторые вычисления. В сигмоиде $\frac{1}{1+e^{-x}}$ 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 $\frac{1}{1+e^{-x}}$.

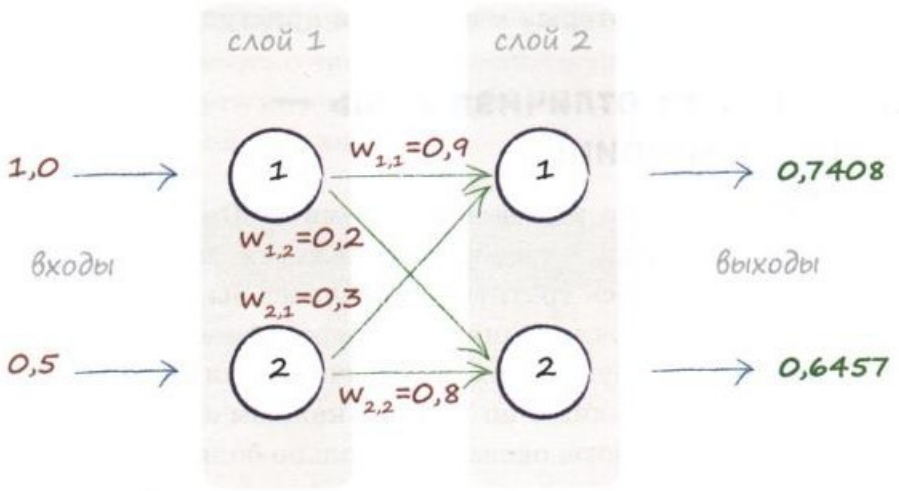
Попробуйте справиться с этим самостоятельно, используя калькулятор. Вот правильный ответ: y = 1 / (1 + 0,3499) = 1 / 1,3499. Таким образом, y=0,7408.

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.



Таким образом, чтобы получить всего лишь два выходных сигнала для весьма простой нейронной сети, нам пришлось хорошо потрудиться. К счастью, для вычислений крупных сетей идеально подходят компьютеры.
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 приступим в следующей работе.

Практические задания

1. Создание класса нейронной сети
2. Инициализация сети
3. Весовые коэффициенты – основная часть сети
4. Улучшенный вариант инициализации весовых коэффициентов

Проект нейронной сети на Python. Часть 1.

1. Создание класса нейронной сети.

Класс нейронной сети 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.

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
```

2. Инициализация сети.

Начнем с инициализации. 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 при каждом 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 и **добавьте нужный код**.

```
# 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
```

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

[3]:
```
# 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
```

[4]:
```
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)
```

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.

Это нормальная практика – начинать с малого и постепенно наращивать код, попутно находя и устраняя ошибки.

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

3. Весовые коэффициенты – основная часть сети.

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.
Удобна компактная запись весов в виде матриц. Следовательно, мы можем создать следующие матрицы:

• матрицу весов для связей между входным и скрытым слоями,

W входной_скрытый, размерностью **hidden_nodes x input_nodes**;

• другую матрицу для связей между скрытым и выходным слоями,

W скрытый_выходной, размерностью **output_nodes x hidden_nodes**.

При необходимости вернитесь к теоретическим материалам лабораторных работ, чтобы понять, почему первая матрица имеет размерность **hidden_nodes x input_nodes**, а не **input_nodes x hidden_nodes**.
Вспомним, 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 x columns**:
numpy.random.rand(rows, columns)
Все хорошие программисты ищут в Интернете онлайновую документацию по функциям Python и находят полезные функции. При желании 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 3x3. Как видите, матрица заполнилась случайными значениями в диапазоне от 0 до 1.

```
[5]: import numpy

[6]: numpy.random.rand(3,3)

[6]: array([[0.96043689, 0.17240669, 0.04986505],
            [0.29109845, 0.73728517, 0.24248404],
            [0.68053887, 0.05742121, 0.43125263]])
```

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 подхода, которые демонстрируют, что некоторые весовые коэффициенты теперь имеют отрицательные значения.

```
[7]: numpy.random.rand(3,3) -0.5

[7]: array([[ 0.21210893, -0.20506655,  0.29219392],
            [-0.38341736,  0.31064623, -0.33060099],
            [-0.00113976, -0.01018176,  0.39300294]])
```

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.

```
1]: # 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)
```

4. 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 теоретических материалах лабораторных работ мы рассматривали различные способы подготовки данных и инициализации коэффициентов. Можно применить и другой, 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*. 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 так. Центр нормального распределения установлен здесь в 0,0. Стандартное

```
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))
```

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.

Контрольные вопросы

1. Для каких задач применим простой линейный классификатор?
2. Каким образом произвести классификацию, если линейный классификатор не применим к задаче?
3. При каких значениях аргументов функция И принимает значение 1?
4. При каких значениях аргументов функция ИЛИ является истинной.

5. При каких значениях аргументов функция *исключающее ИЛИ* принимает значение 1?

6. Опишите строение нейрона.

7. Что представляет собой функция активации?

8. Какая функция чаще всего используется в качестве функции активации в искусственных нейронных сетях? По каким причинам?

9. Что такое весовые коэффициенты сети?

10. Каким образом происходит распространение сигнала по искусственной нейронной сети?

11. Какие функции может содержать класс нейронной сети, созданный на языке Python?

Лабораторная работа 5
Решение сложных задач классификации с помощью искусственных нейронных сетей. Метод обратного распространения

Цель: рассмотреть метод обратного распространения ошибки при разработке проекта нейронной сети.

Задание:

1. Изучить теоретическое введение.
2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.
3. Выполнить практические задания с использованием языка программирования Python.
4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчѐт предоставляется в электронном виде в текстовом редакторе MS Word. В отчѐте указываются:

1) Фамилия студента.
2) Порядковый номер и название лабораторной работы.
3) Цель работы.
4) Ответы на контрольные вопросы.
5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчѐту, представленному студентом и с демонстрацией задания на компьютере.

Теоретическое введение

1. Умножение матриц
2. Пример использования матричного умножения в сети с тремя слоями
3. Корректировка весовых коэффициентов в процессе обучения нейронной сети
4. Обратное распространение ошибок при большом количестве слоев
5. Описание обратного распространения ошибок с помощью матричной алгебры

1. Умножение матриц

В предыдущей работе для перемножения матриц вручную мы 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...

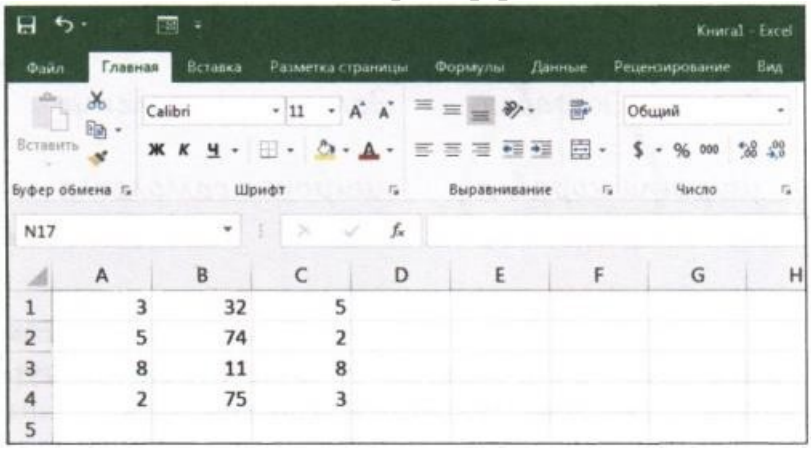
Так чем же нам могут помочь матрицы? Они будут нам полезны в двух случаях. ***Во-первых***, они обеспечивают сжатую запись операций, придавая

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.*

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.



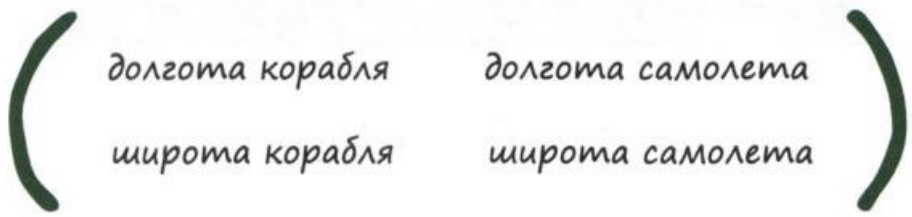
	A	B	C	D	E	F	G	H
1	3	32	5					
2	5	74	2					
3	8	11	8					
4	2	75	3					
5								

Это и есть матрица – таблица или сетка, образованная числами, точно так же, как и следующий пример матрицы размером 2x3.

При обозначении матриц принято указывать сначала количество строк, а затем количество столбцов. Поэтому размерность данной матрицы не 3x2, а 2x3.

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.



Чем полезны матрицы, вам станет понятно, когда мы рассмотрим, как они умножаются. Вот пример умножения одной простой матрицы на другую.

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

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

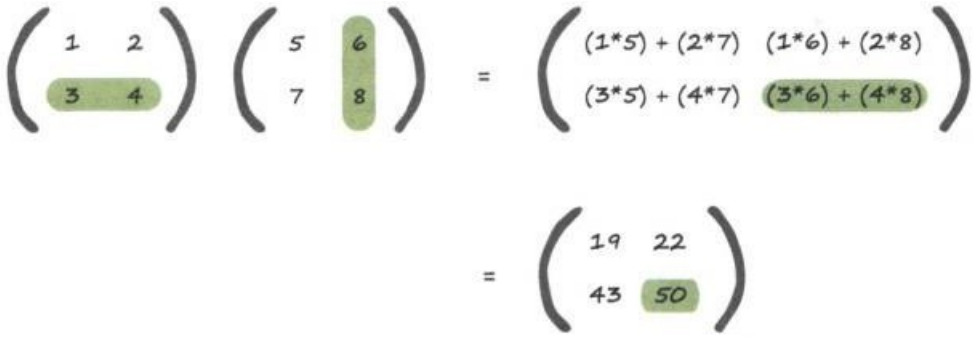
Эта операция не сводится к простому перемножению соответствующих элементов. Левый верхний элемент не равен 1*5, как и правый нижний – не 4*8. Вместо этого матрицы умножаются по другим правилам. Возможно, вы
и сами догадаетесь, по каким именно, если внимательно присмотритесь к примеру. Если нет, то взгляните на приведенную ниже иллюстрацию, где показано, как получается ответ для левого верхнего элемента.

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

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

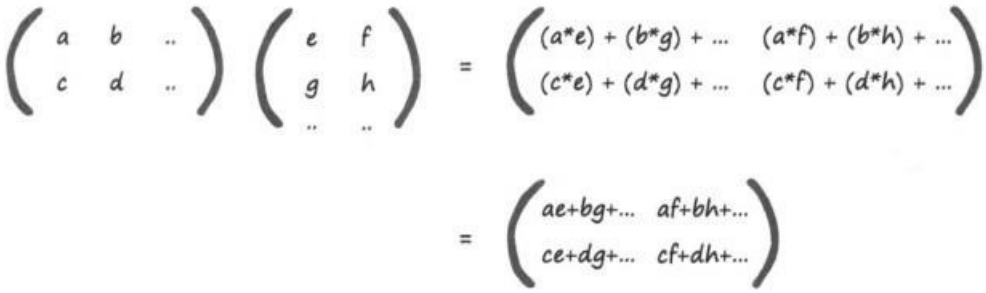
Вы видите, что верхний левый элемент вычисляется с использованием верхней строки первой матрицы и левого столбца второй.
Мысленно перебирая обе последовательности элементов, перемножьте их попарно и сложите полученные результаты. Таким образом, чтобы вычислить левый верхний элемент результирующей матрицы, 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.



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:



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 2x2 на матрицу 5x5. 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.

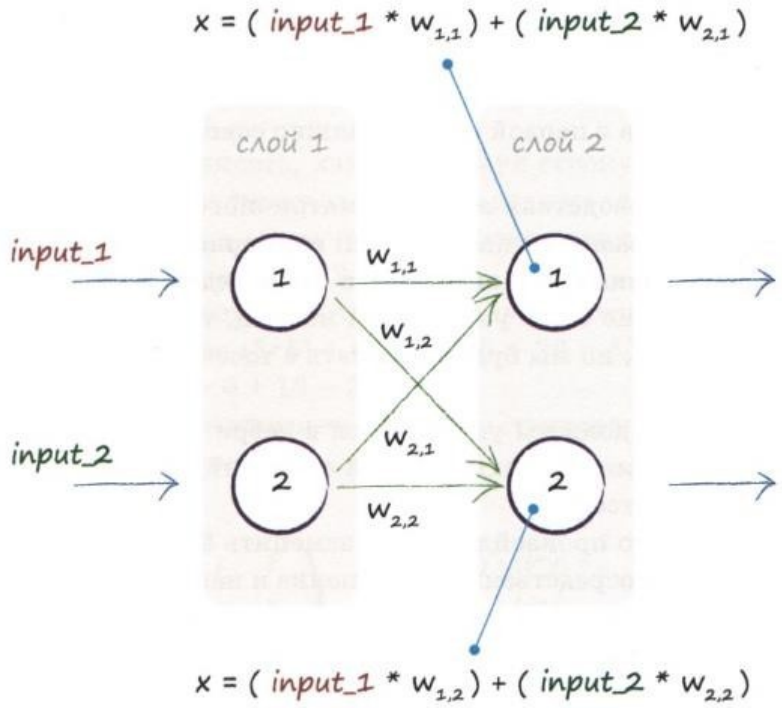
Посмотрим, что произойдет, если заменить буквы словами, имеющими более непосредственное отношение к нашей нейронной сети. На приведенной ниже иллюстрации вторая матрица имеет размерность 2x1, но матричная алгебра остается прежней.

$$\begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix} \begin{pmatrix} input_1 \\ input_2 \end{pmatrix} = \begin{pmatrix} (input_1 * w_{1,1}) + (input_2 * w_{2,1}) \\ (input_1 * w_{1,2}) + (input_2 * w_{2,2}) \end{pmatrix}$$

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 $\mathbf{w}_{1,1}$, и сигнала *input_2*, умноженного на весовой коэффициент $\mathbf{w}_{2,1}$. 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:

X = W 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 подумать 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.

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.

Что насчет функции активации? Здесь все просто и не требует применения матричной алгебры. Все, что нам нужно сделать, – это применить сигмоиду $\frac{1}{1+e^{-x}}$ 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 * 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.

Резюме

- Многие вычисления, связанные с распространением сигналов по нейронной сети, могут быть выполнены с использованием операции матричного умножения.

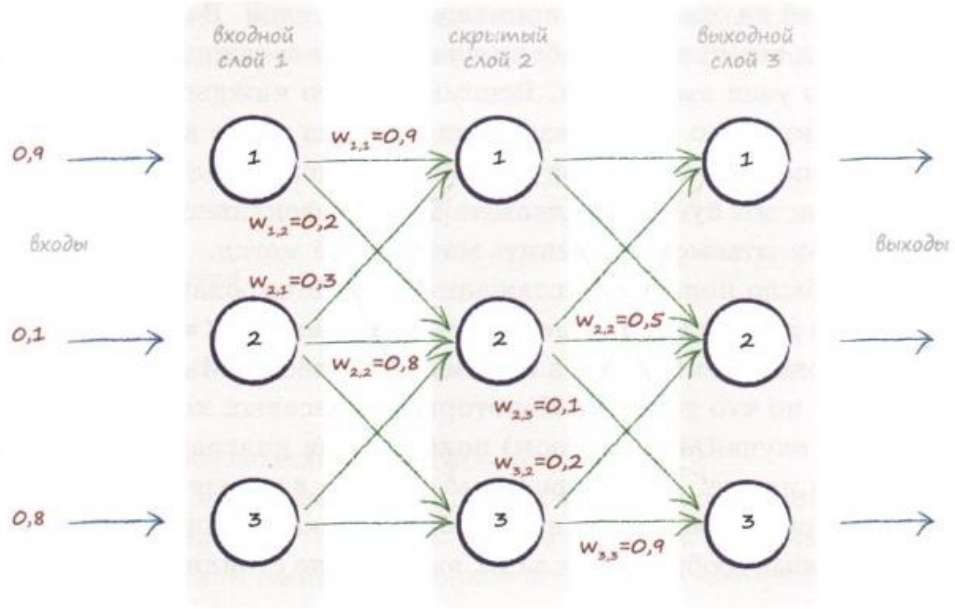
• Использование матричного умножения обеспечивает компактную запись выражений, размер которых не зависит от размеров нейронной сети.

• Что немаловажно, операции матричной алгебры изначально предусмотрены в ряде языков программирования, благодаря чему могут выполняться быстро и эффективно.

2. 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**. 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 * 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).

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

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

Рассмотрим, почему мы снабдили матрицу **W** индексом «входной_скрытый»? Потому, что матрица $\mathbf{W}_{входной_скрытый}$ 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 $\mathbf{W}_{скрытый_выходной}$.

Эта вторая матрица $\mathbf{W}_{скрытый_выходной}$, 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 $\mathbf{X}_{скрытый}$:

$$\mathbf{X}_{скрытый} = \mathbf{W}_{входной_скрытый} * \mathbf{I}$$

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.

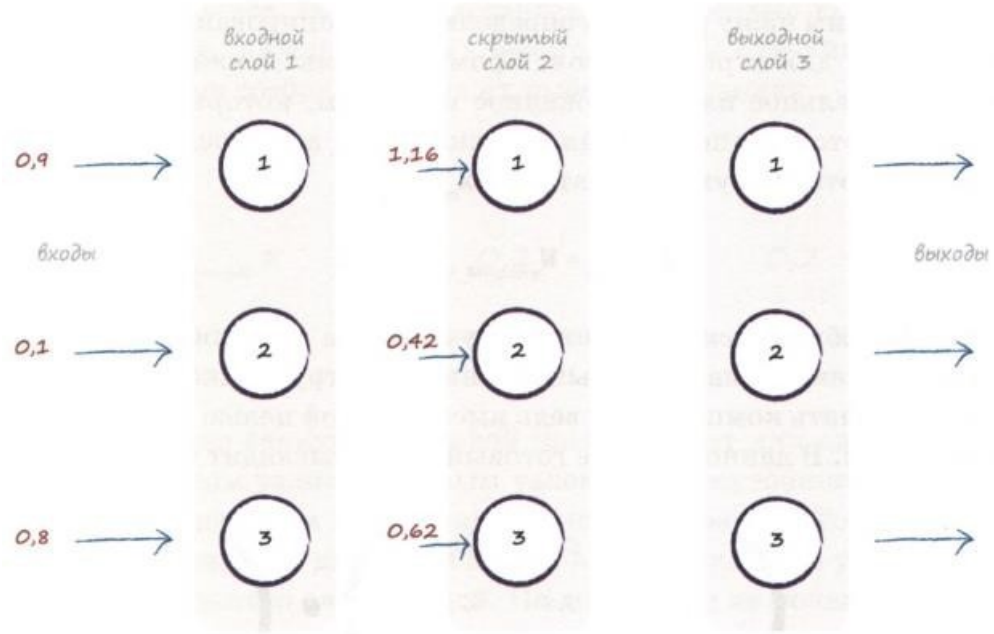
$$X_{скрытый} = \begin{pmatrix} 0,9 & 0,3 & 0,4 \\ 0,2 & 0,8 & 0,2 \\ 0,1 & 0,5 & 0,6 \end{pmatrix} \cdot \begin{pmatrix} 0,9 \\ 0,1 \\ 0,8 \end{pmatrix}$$

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

Вы научитесь получать готовый ответ, используя язык программирования Python, о чем мы поговорим позже.

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.

Отобразим входные сигналы для скрытого второго слоя на диаграмме.



Нам остается еще 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:

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

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

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

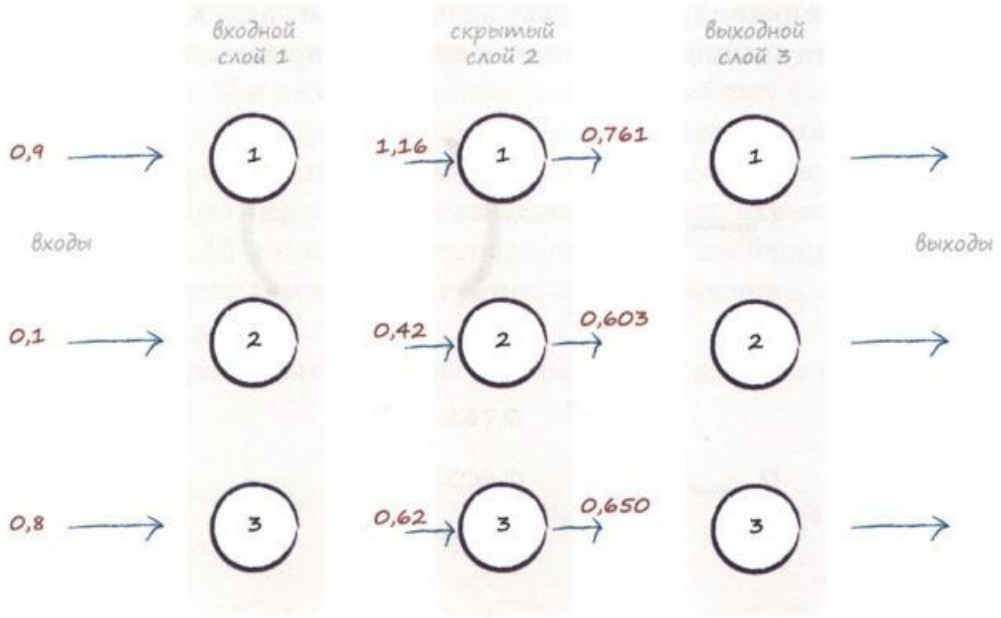
$$O_{\text{скрытый}} = \begin{pmatrix} 0{,}761 \\ 0{,}603 \\ 0{,}650 \end{pmatrix}$$

Для уверенности давайте проверим, правильно ли вычислен первый элемент. Наша сигмоида имеет вид $\frac{1}{1+e^{-x}}$. 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.

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 ни чем 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 $\mathbf{X = W * I}$ для третьего слоя.

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

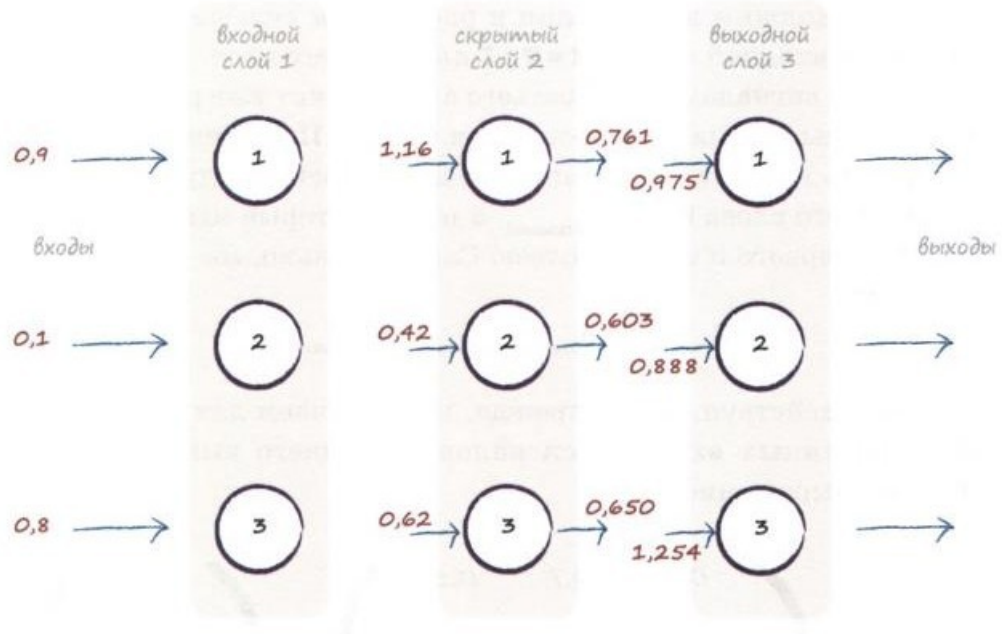
$$\mathbf{X}_{\text{выходной}} = \mathbf{W}_{\text{скрытый_выходной}} * \mathbf{O}_{\text{скрытый}}$$

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

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

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

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.

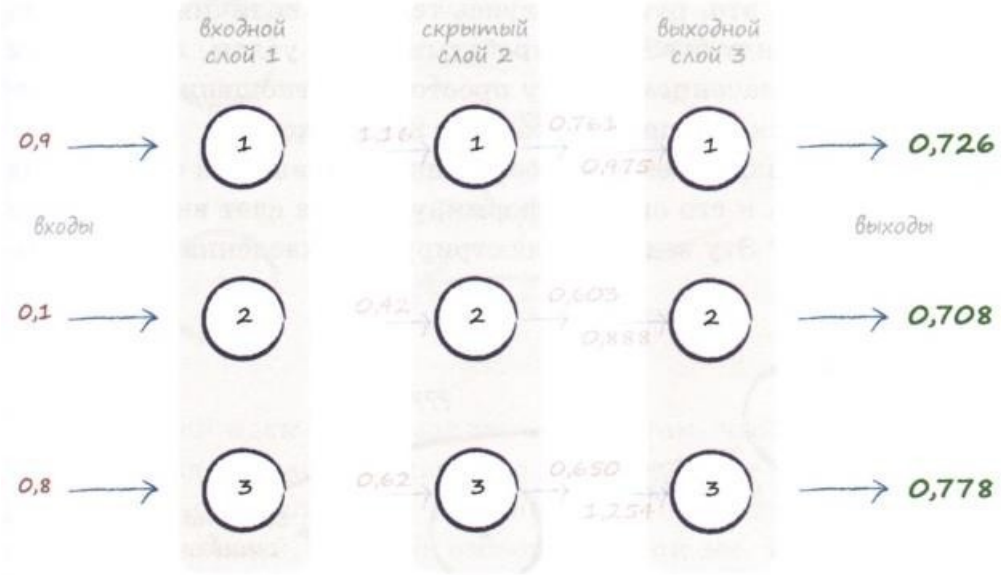


Все, что нам остается, – это применить сигмоиду.

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

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

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; 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?

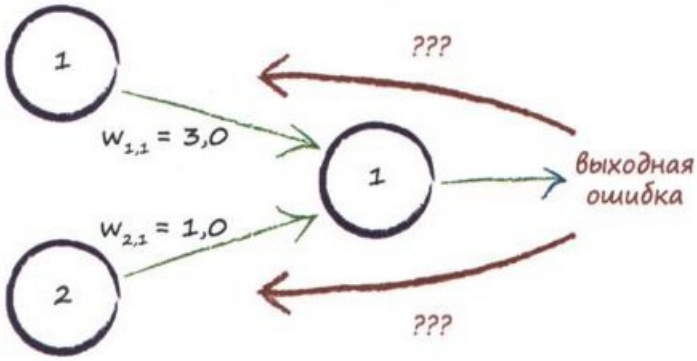
Наш следующий шаг заключается в сравнении выходных сигналов нейронной сети с данными тренировочного примера для определения ошибки. Нам необходимо знать величину этой ошибки, чтобы можно было улучшить выходные результаты путем изменения параметров сети.

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

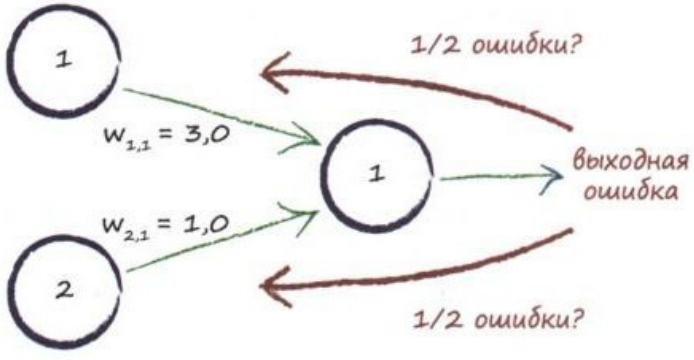
3. Корректировка весовых коэффициентов в процессе обучения нейронной сети

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.

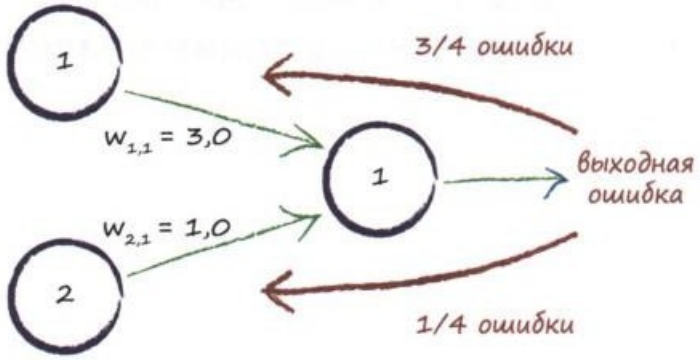


Когда на выходной узел поступал сигнал только от одного узла, все было намного проще. Но как использовать ошибку выходного сигнала при наличии двух входных узлов?
Использовать всю величину ошибки для обновления только одного весового коэффициента не представляется разумным, поскольку при этом полностью 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 большая 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.

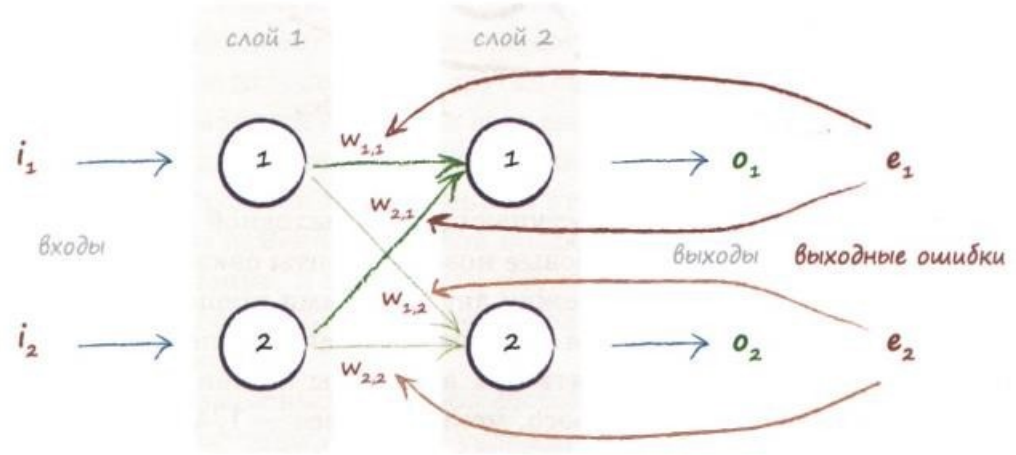
Мы можем расширить эту идею на случаи с намного большим количеством узлов. Если бы выходной узел был связан со ста входными узлами, мы распределили бы выходную ошибку между всеми ста связями **пропорционально** их вкладам, размер которых определяется весами соответствующих связей.

Мы используем весовые коэффициенты в двух целях. *Во-первых*, они учитываются при расчете распространения сигналов по нейронной сети от входного слоя до выходного. Мы использовали их ранее именно в таком качестве. *Во-вторых*, мы используем веса для распространения ошибки в обратном направлении – от выходного слоя вглубь сети. 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.

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. 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 вы□одного 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 $\mathbf{e}_1$. Не забывайте, что это разность между желаемым значением, предоставляемым тренировочными данными $\mathbf{t}_1$, и фактическим выходным значением $\mathbf{o}_1$. Таким образом, $\mathbf{e}_1 = (\mathbf{t}_1 - \mathbf{o}_1)$. Ошибка на втором выходном узле обозначена как $\mathbf{e}_2$.

На диаграмме видно, что ошибка $\mathbf{e}_1$ распределяется пропорционально весам связей, обозначенным как $\mathbf{w}_{11}$ и $\mathbf{w}_{21}$. Точно так же ошибка $\mathbf{e}_2$ должна распределяться пропорционально весам $\mathbf{w}_{12}$ и $\mathbf{w}_{22}$.

Запишем эти доли в явном виде. Ошибка $\mathbf{e}_1$ информирует о величинах поправок для весов $\mathbf{w}_{11}$ и $\mathbf{w}_{21}$. При ее распределении между узлами доля $\mathbf{e}_1$, информация о которой используется для обновления $\mathbf{w}_{11}$, определяется следующим выражением:

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

Доля $\mathbf{e}_1$, используемая для обновления $\mathbf{w}_{21}$, определяется аналогичным выражением:

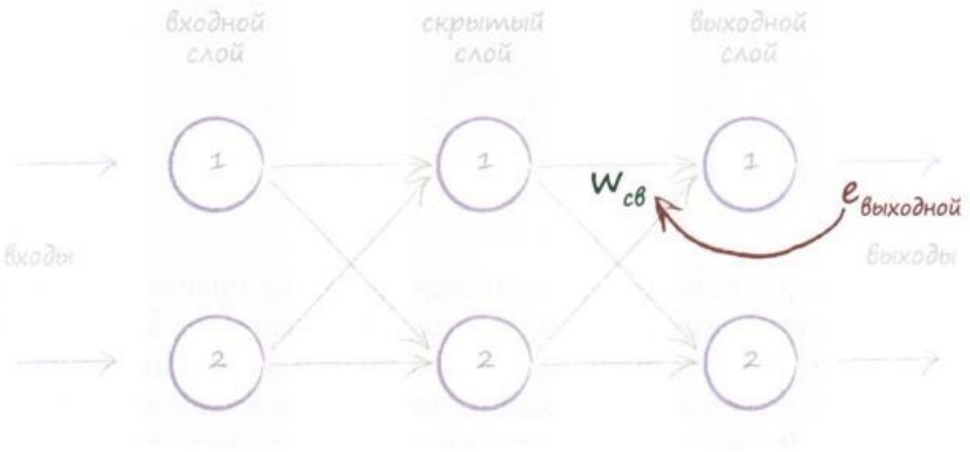
$$\frac{w_{21}}{w_{11} + w_{21}}$$

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.

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

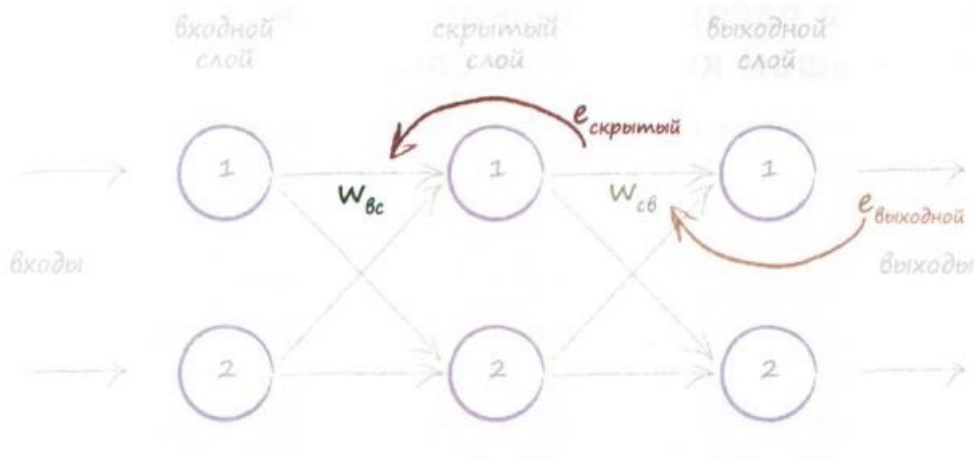
4. Обратное распространение ошибок при большом количестве слоев

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 $\mathbf{e}_{\text{выходной}}$ для выходных ошибок и $\mathbf{w}_{\text{св}}$ для весов связей между скрытым и выходным слоями. Мы вычисляем конкретные ошибки, ассоциируемые с каждой связью, путем распределения ошибки пропорционально соответствующим весам.

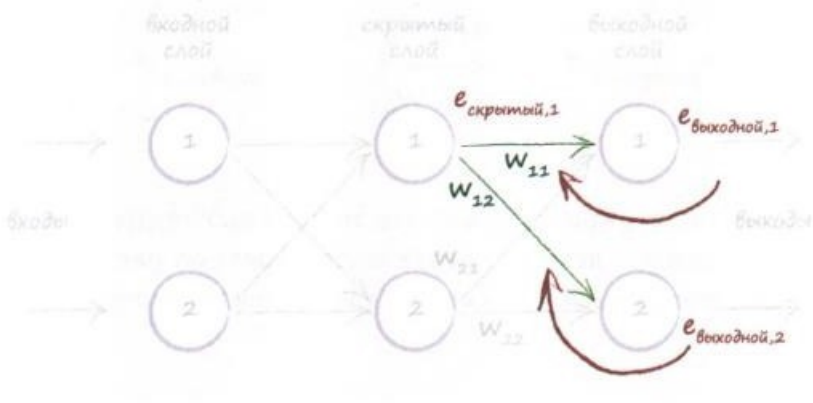
Графическое представление позволяет лучше понять, какие вычисления следует выполнить для дополнительного слоя. Мы просто берем ошибки $e_{\text{скрытый}}$ на выходе скрытого слоя и вновь распределяем их по предшествующим связям между входным и скрытым слоями пропорционально весовым коэффициентам $w_{\text{вс}}$. 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. Мы еще раз имели возможность увидеть, почему этот процесс описывается термином **обратное распространение ошибок**.

Если мы сначала использовали ошибку **e** $_{\text{выходной}}$ выходного сигнала узлов выходного слоя, то какую ошибку **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 ответ, 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:



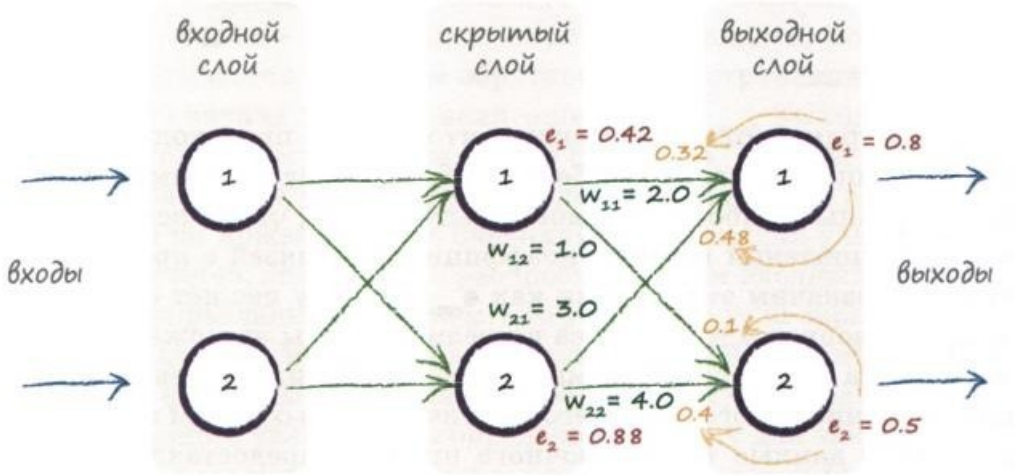
На диаграмме видно, что именно происходит. Нам необходимы величины ошибок для узлов скрытого слоя, чтобы использовать их для обновления весовых коэффициентов связей с предыдущим слоем. Обозначим эти ошибки как **e** скрытый. Но у нас нет очевидного ответа на вопрос о том, какова их величина. Мы не можем 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 задачи.

Мы можем воссоединить ошибки, распределенные по связям, используя обратное распространение ошибок, с которым 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:

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

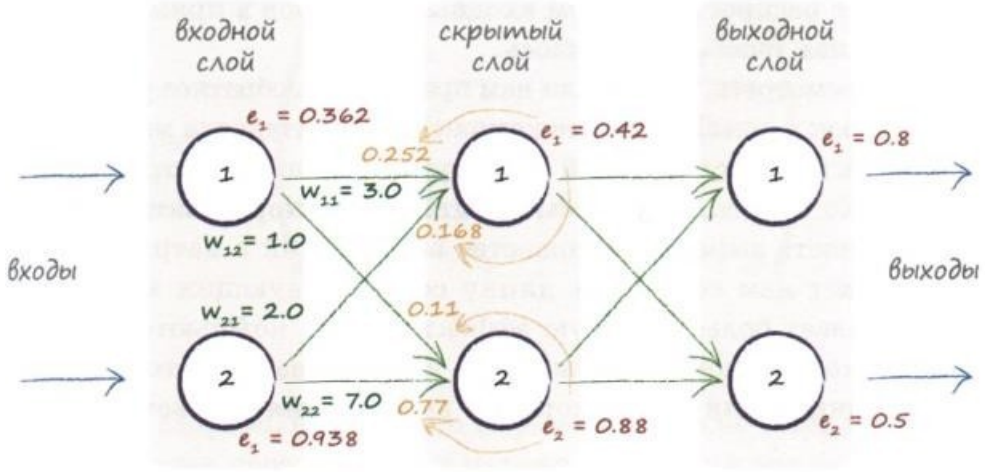
$$= e_{выходной,1} * \frac{w_{11}}{w_{11} + w_{21}} + e_{выходной,2} * \frac{w_{12}}{w_{12} + w_{22}}$$

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. 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.

5. 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 $\mathbf{e}_1$ и $\mathbf{e}_2$:

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

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, можно увидеть, что ошибка на первом узле скрытого слоя формируется за счет двух вкладов со стороны выходного слоя. Этими двумя сигналами ошибок являются $\mathbf{e}_1 * \mathbf{w}_{11} / (\mathbf{w}_{11} + \mathbf{w}_{21})$ и $\mathbf{e}_2 * \mathbf{w}_{12} / (\mathbf{w}_{12} + \mathbf{w}_{22})$. Обратите внимание на второй узел скрытого слоя, можно увидеть, что ошибка на нем также формируется за счет двух вкладов: $\mathbf{e}_1 * \mathbf{w}_{21} / (\mathbf{w}_{21} + \mathbf{w}_{11})$ и $\mathbf{e}_2 * \mathbf{w}_{22} / (\mathbf{w}_{22} + \mathbf{w}_{12})$. Ранее мы уже видели, как работают эти выражения.

Итак, для скрытого слоя мы имеем следующую матрицу, которая выглядит немного сложнее, чем матрицы, рассматриваемые ранее.

$$ошибка_{скрытый} = \begin{pmatrix} \frac{w_{11}}{w_{11} + w_{21}} & \frac{w_{12}}{w_{12} + w_{22}} \\ \frac{w_{21}}{w_{21} + w_{11}} & \frac{w_{22}}{w_{22} + w_{12}} \end{pmatrix} \cdot \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

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 нужен способ, обеспечивающий возможность использования матричного умножения, чтобы повысить эффективность вычислений.

Посмотрите еще раз на приведенное выше выражение. Вы видите, что наиболее важная для нас вещь – это умножение выходных ошибок $\mathbf{e}_n$ на связанные с ними веса $\mathbf{w}_{ij}$. Чем больше вес, тем большая 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 механизму обратной связи. Таким образом, выражение $\mathbf{e}_1 * \mathbf{w}_{11} / (\mathbf{w}_{11} + \mathbf{w}_{21})$ упростится до $\mathbf{e}_1 * \mathbf{w}_{11}$.

Получим следующее уравнение.

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

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 $\mathbf{w}^T$.

Ниже приведены два примера транспонирования числовых матриц, которые помогут лучше понять смысл операции транспонирования. 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 применили матричный подход к описанию обратного распространения ошибок:

$$ошибка_{скрытый} = w^{T}_{скрытый_выходной} \bullet ошибка_{выходной}$$

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

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.

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.

Практические задания

1. Опрос сети
2. Текущее состояние кода
3. Тренировка сети

Проект нейронной сети на Python. Часть 2.

1.Опрос сети

Далее зададим функцию *query ()*, которая обеспечивает опрос сети – получение значений сигналов с выходных узлов после предоставления значений входящих сигналов. Т.е. функция *query ()* принимает в качестве аргумента входные данные нейронной сети и возвращает ее выходные данные. 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 так□же 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:

$$\mathbf{X}_{\text{скрытый}} = \mathbf{W}_{\text{входной_скрытый}} * \mathbf{I}$$

Компьютерные языки, в том числе Python, распознают матрицы и эффективно выполняют все расчеты, поскольку им известно об однотипности всех стоящих за этим вычислений.
Код на языке Python для этой операции достаточно простой. Ниже представлена инструкция, которая показывает, как применить функцию скалярного произведения библиотеки *numpy* к матрицам весов и входных сигналов:
Эта короткая строка кода Python выполняет всю работу по объединению всех

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

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!
Для получения выходных сигналов скрытого слоя мы просто при□меняем 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.
Библиотека *scipy* в Python содержит набор специальных функций, в том числе сигмоиду, которая называется в данной библиотеке *expit ()*. 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 создает функцию наподобие любой другой, только с использованием более короткого способа записи, называемого ***лямбда-выражением***. Вместо привычного определения функции в форме *def имя()* мы использовали 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, вы□деленные полужирным шрифтом, которые выполняют все необходимые расчеты: две – для скрытого слоя и две – для выходного слоя.

2. Текущее состояние кода

Посмотрим, как выглядит в целом код, который мы к этому времени создали.

```
7]: # 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:
    # wll w21
    # wl2 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*.

Функции *query ()* в качестве входных данных потребуются только входные

```
In [8]: import numpy
        # библиотека scipy.special с сигмоидой expit ()
        import scipy.special
```

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)*.

```
[8]: 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)
```

```
[9]: n.query([1.0, 0.5, -1.5])
```

```
[9]: array([[0.638757861,
            [0.656137261,
            [0.436785441]])
```

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.

3. 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) :
  # 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)
```

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.

```
    # ошибки выходного слоя =
    # (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:

$$\textbf{ошибки}_{\text{скрытый}} = \textbf{веса}^{T}_{\text{скрытый_выходной}} * \textbf{ошибки}_{\text{выходной}}$$

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 ***a*** – 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 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))
```

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*** 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 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))
```

self.who += self.lr * numpy .dot ((output_errors * final_outputs * (1.0 - final_outputs)), numpy.transpose (hidden_outputs))

self.wih += self.lr * numpy.dot((hidden_errors * hidden_outputs * (1.0 - hidden_outputs)), numpy.transpose(inputs))

Вот и все!

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.

Приведем полный код функции train. **Его необходимо добавить в программу.**

```
# 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
```

Полный код нейронной сети 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.

Контрольные вопросы

1. Что такое матрица?
2. Как получить скалярное произведение матриц? Приведите пример.
3. Что означает транспонирование матриц?
4. Что представляют собой входной, выходной и скрытый слои нейронной сети?
5. Опишите кратко процесс корректировки весовых коэффициентов в процессе обучения нейронной сети.
6. Что такое метод обратного распространения ошибки?

Лабораторная работа 6
Обновление весовых коэффициентов. Набор рукописных цифр MNIST.

Цель: рассмотреть процесс обновления весовых коэффициентов; подготовить тренировочные данные MNIST для проекта нейронной сети по распознаванию рукописных цифр.

Задание:

1. Изучить теоретическое введение.
2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.
3. Выполнить практические задания с использованием языка программирования Python.
4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчѐт предоставляется в электронном виде в текстовом редакторе MS Word. В отчѐте указываются:

1) Фамилия студента.
2) Порядковый номер и название лабораторной работы.
3) Цель работы.
4) Ответы на контрольные вопросы.
5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчѐту, представленному студентом и с демонстрацией задания на компьютере.

Теоретическое введение

Рассмотрение процесса обновления весовых коэффициентов

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 в первой лабораторной работе.

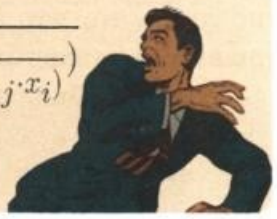
Однако узлы – это не простые линейные классификаторы. В узлах сигналы суммируются с учетом весов, после чего к ним применяется сигмоида. Но как нам все-таки справиться с обновлением весов для связей, соединяющих эти более сложные узлы? Почему бы не использовать алгебру для непосредственного вычисления 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 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 $\mathbf{x}_i$, весовой коэффициент для связи, соединяющей входной узел **i** с узлом **j** скрытого слоя, равен $\mathbf{w}_{i,j}$. Аналогичным образом выходной сигнал узла **j** скрытого слоя равен $\mathbf{x}_j$, а весовой коэффициент для связи, соединяющей узел **j** скрытого слоя с выходным узлом **k**, равен $\mathbf{w}_{j,k}$. Символ $\sum_a$ означает суммирование следующего за ним выражения по всем значениям между **a** и **b**.

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



А не могли бы мы вместо данных сложных расчетов просто перебирать случайные сочетания весовых коэффициентов, пока не будет получен устраивающий нас результат?

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 получилось 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.

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.

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.

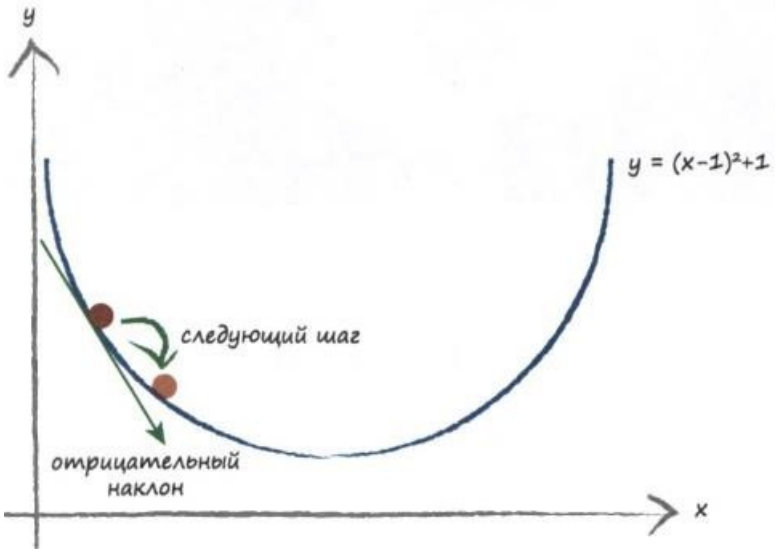
Проиллюстрируем суть наших рассуждений на следующем примере. Представьте себе ландшафт с очень сложным рельефом, имеющим возвышения и впадины, а также холмы с буграми и ямами. Вокруг так темно, что ни зги не видно. Вы знаете, что находитесь на склоне холма, и вам нужно добраться до его подножия. Точной карты местности у вас нет. Но у вас есть фонарь. 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.

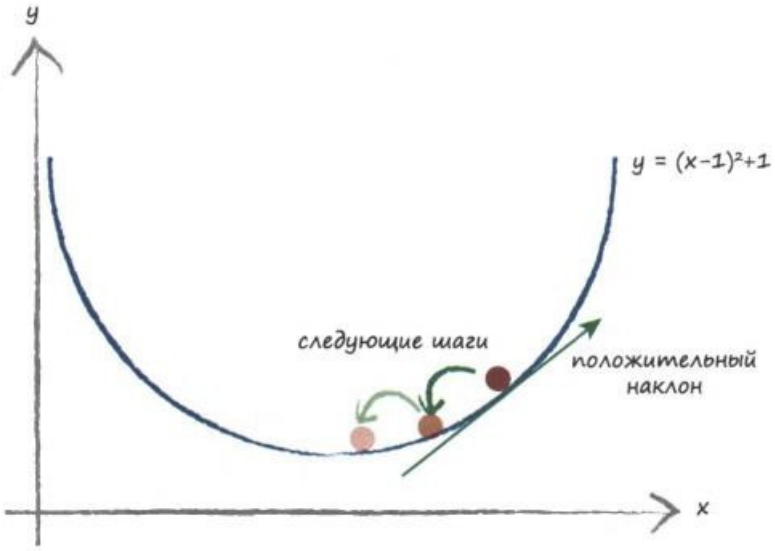
Рассмотрим использование метода градиентного спуска на простейшем примере.

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, что мы имеем дело не со столь простой функцией, а с гораздо более сложной.



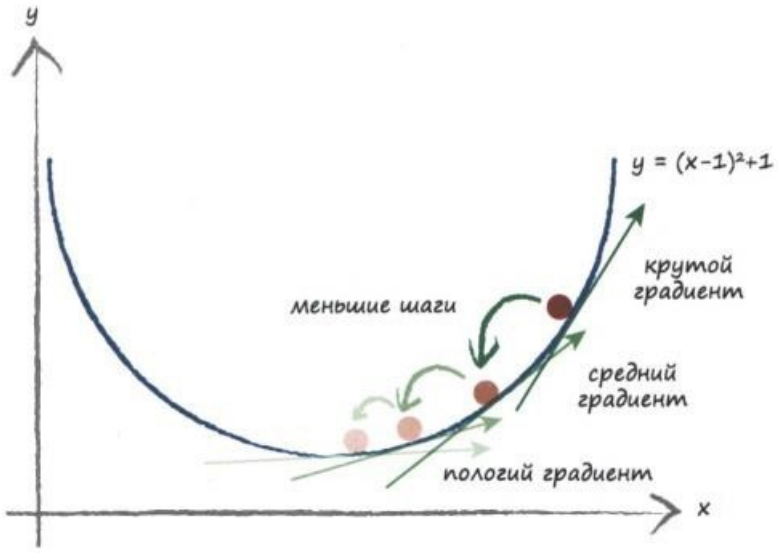
Градиентный спуск должен с чего-то начинаться. На графике показана случайно выбранная начальная точка. Мы исследуем место, в котором находимся, и видим, в каком на□правлении идет спуск. Наклон кривой также обозначен на графике, и в данном случае ему соответствует отрицательный градиент. Мы хотим следовать в направлении вниз, поэтому движемся вдоль *оси x* вправо. Таким образом, мы немного увеличиваем **x**. Это первый шаг. 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 видно, 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 сложным редко встречающимся 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.

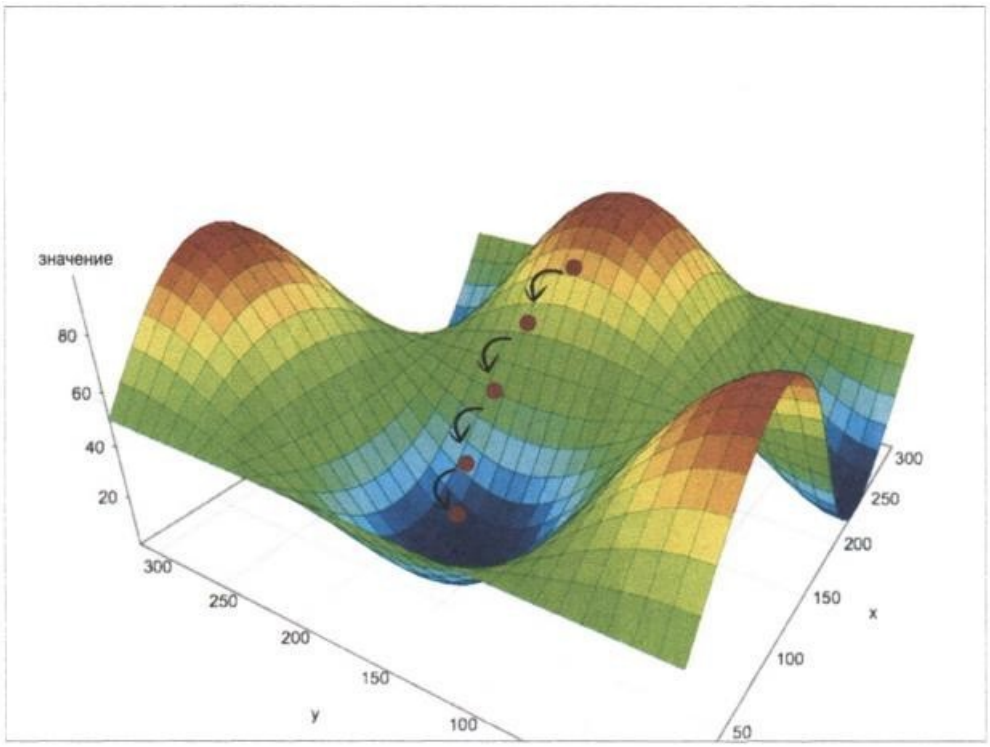


Мы изменяем **x** в направлении, противоположном направлению градиента. Положительный градиент означает, что мы должны уменьшить **x**. Отрицательный градиент требует увеличения **x**.

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 в правильном направлении.

Преимущества 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.

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. Но если вы решите использовать сумму ошибок по всем узлам в качестве общего показателя того, насколько хорошо обучена сеть, то эта сумма равна нулю.

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;

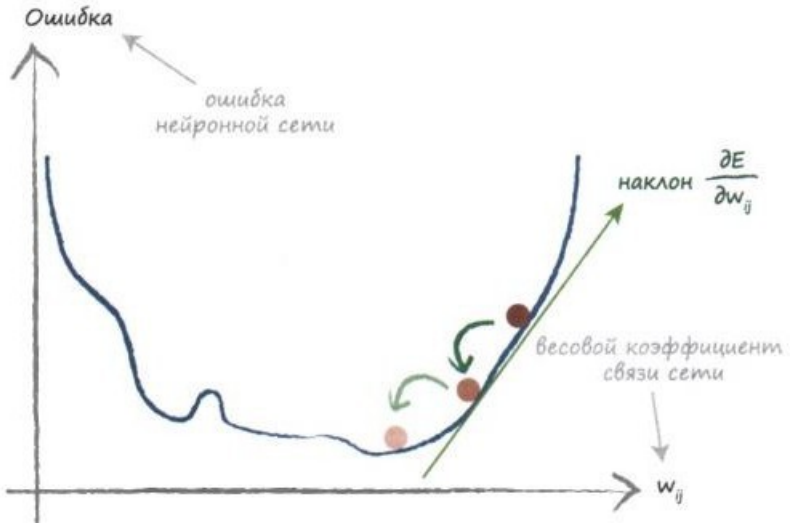
• функция ошибки является непрерывно гладкой, что обеспечивает нормальную работу метода градиентного спуска ввиду отсутствия провалов и скачков значений функции;

• 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, можно 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.

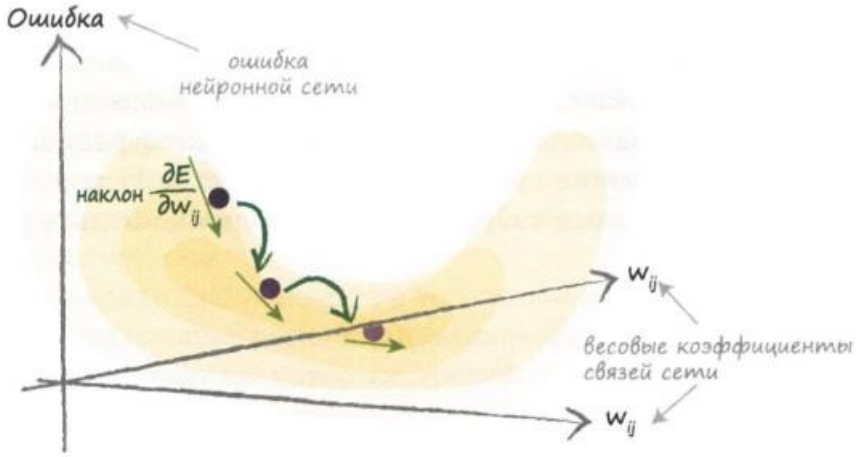
Чтобы воспользоваться методом градиентного спуска, нужно определить наклон функции ошибки по отношению к весовым коэффициентам. Это требует применения дифференциального исчисления. Дифференциальное исчисление – это математически строгий подход к определению величины изменения одних величин при изменении 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.



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

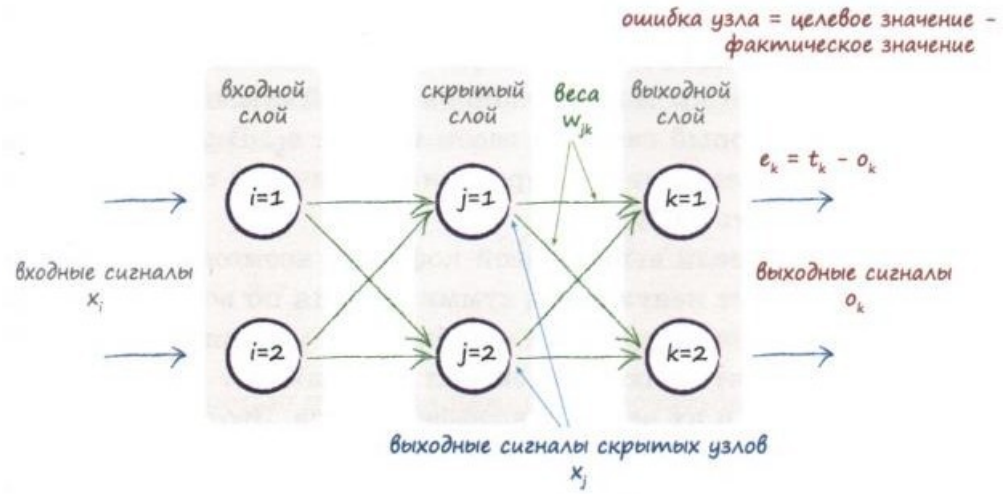
один весовой коэффициент, но мы знаем, что в нейронных сетях их будет намного больше.
На следующей диаграмме отображаются два весовых коэффициента, и поэтому функция ошибки графически отображается в виде трехмерной поверхности, высота расположения точек которой изменяется с изменением весовых коэффициентов 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 нужное выражение:

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

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. 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.

Здесь мы всего лишь записали, что на самом деле представляет собой функция

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

ошибки **E**.

Мы можем сразу же упростить это выражение, заметив, что выходной сигнал $\mathbf{o_n}$ на узле **n** зависит лишь от связей, которые с ним соединены. Для узла **k** это означает, что выходной сигнал $\mathbf{o_k}$ зависит лишь от весов $\mathbf{w_{jk}}$, 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 $\mathbf{w_{jb}}$, где **b** не равно **k**, поскольку связь между этими узлами отсутствует. Вес $\mathbf{w_{jb}}$ относится к связи, ведущей к выходному узлу **b**, но не **k**.

Это означает, что мы можем удалить из этой суммы все сигналы $\mathbf{o_n}$ кроме того, который связан с весом $\mathbf{w_{jk}}$, т.е. $\mathbf{o_k}$. В результате мы полностью избавляемся от суммирования.

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:

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

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

Чтобы разбить эту задачу дифференцирования на более простые части, мы воспользуемся цепным правилом (правило дифференцирования сложных функций).

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

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 $\mathbf{o_k}$ – это выходной сигнал узла **k**, который, получается в результате применения сигмоиды к сигналам, поступающим на данный узел. Для большей ясности запишем это в явном виде:

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

Здесь $\mathbf{o_j}$ – 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, однако эта работа уже проделана другими людьми. Поэтому мы просто воспользуемся уже известным ответом, как это ежедневно делают математики по всему миру.

Дифференцирование некоторых функций приводит к выражениям сложнейшего

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

вида. В случае же сигмоиды результат получается очень

простым. Это одна из причин широкого применения сигмоиды в качестве функции активации в нейронных сетях.
Получаем следующее выражение:

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

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

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 $\mathbf{w}_{jk}$. Это делается очень просто и дает в результате $\mathbf{o}_j$.
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.
Окончательное выражение, которое мы будем использовать для изменения веса $\mathbf{w}_{jk}$, выглядит так.

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

Это выражение 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 $\mathbf{i}_k$. 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.
Выражение 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 $\mathbf{e_j}$.

• 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 $\mathbf{i_j}$.

• Последняя часть выражения приобретает смысл выходных сигналов $\mathbf{o_i}$ узлов первого слоя, и в данном случае эти сигналы являются входными.

Тем самым нам удалось избежать излишних трудоемких вычислений, в полной мере воспользовавшись всеми преимуществами симметрии задачи для конструирования нового выражения.

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 мы 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 $\mathbf{w_{jk}}$ – 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. 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.

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.

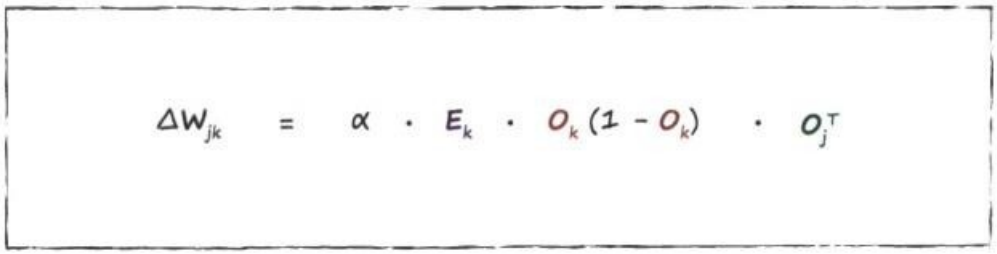
Коэффициент обучения **α** опущен, поскольку это всего лишь константа, которая никак не влияет на то, как мы организуем матричное умножение.

Матрица изменений весов содержит значения поправок к весовым коэффициентам $\mathbf{w}_{jk}$ 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 $\mathbf{o}_j$ 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.



Фактически это выражение совсем не сложное. Сигмоиды исчезли из поля зрения, поскольку они скрыты в матрицах выходных сигналов $\mathbf{o}_k$ узлов.

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.

Практические задания

1. Набор рукописных цифр MNIST
2. Подготовка тренировочных данных MNIST

1. 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.

Существует коллекция изображений рукописных цифр, используемых исследователями искусственного интеллекта в качестве популярного набора для тестирования идей и алгоритмов.

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.

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 CSVфайлы, 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);

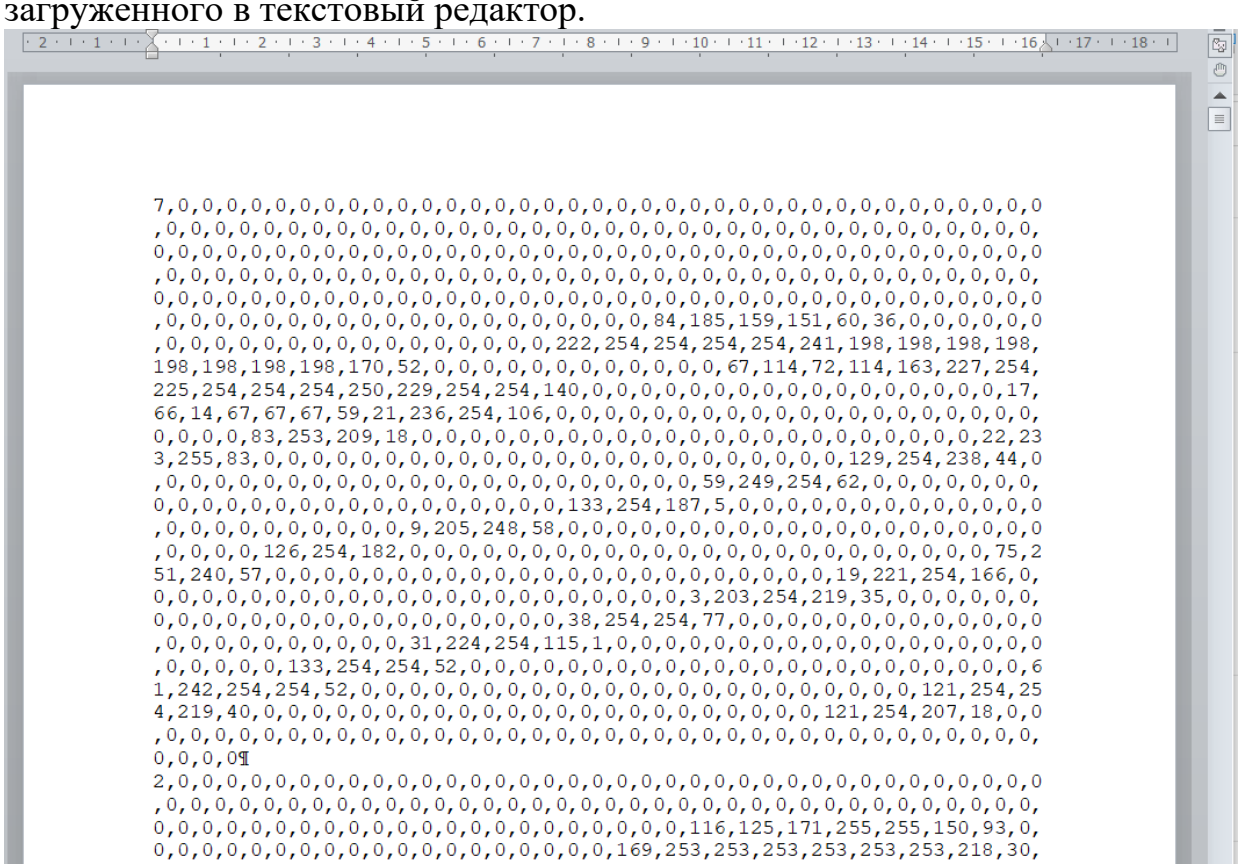
• тестовый набор (http://www.pjreddie.com/media/files/mnist_test.csv).

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 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.



```
7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,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
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,84,185,159,151,60,36,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,222,254,254,254,254,241,198,198,198,198,
198,198,198,198,170,52,0,0,0,0,0,0,0,0,0,0,67,114,72,114,163,227,254,
225,254,254,254,250,229,254,254,140,0,0,0,0,0,0,0,0,0,0,0,0,0,0,17,
66,14,67,67,67,59,21,236,254,106,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,83,253,209,18,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,22,23
3,255,83,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,129,254,238,44,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,59,249,254,62,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,133,254,187,5,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,9,205,248,58,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,126,254,182,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,75,2
51,240,57,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,19,221,254,166,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,3,203,254,219,35,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,38,254,254,77,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,31,224,254,115,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,133,254,254,52,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6
1,242,254,254,52,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,121,254,25
4,219,40,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,121,254,207,18,0,0
,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¶
2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,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,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,116,125,171,255,255,150,93,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,169,253,253,253,253,253,253,218,30,
```

В окне текстового редактора отображаются длинные строки текста (в конце каждой строки отображается знак абзаца), которые содержат числа,

разделенные запятыми. Это легко можно увидеть. Текстовые строки такие длинные, потому что каждая из них занимает несколько строк на экране.
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 28x28, поэтому за каждым маркером следуют 784 пикселя.

Таким образом, первая запись представляет цифру «7» , о чем говорит первое значение, тогда как остальная часть текста в этой строке – это пиксельные значения написанной кем-то от руки цифры «7» . Вторая строка представляет рукописную цифру «2» , 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 «7» , 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 мы отработаем алгоритм, можно будет вернуться к полному набору данных.
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_train.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, а путь доступа к нему. 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.

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 на данном этапе очень важны.

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. 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 [11]: data_file = open("mnist_train.csv", 'r')
         data_list = data_file.readlines()
         data_file.close()

In [12]: len(data_list)

Out[12]: 60000

In [13]: data_list[0]

Out[13]: '5,0,0,0
```

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.
Посмотрите на следующие три строки кода. Переменные окрашены различными

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

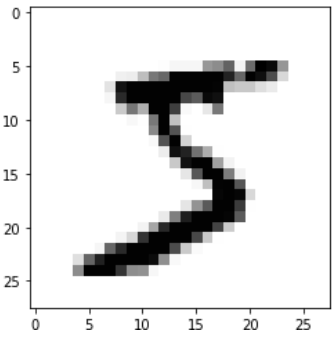
цветами таким образом, чтобы было понятно, где и какие данные используются.

```
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,

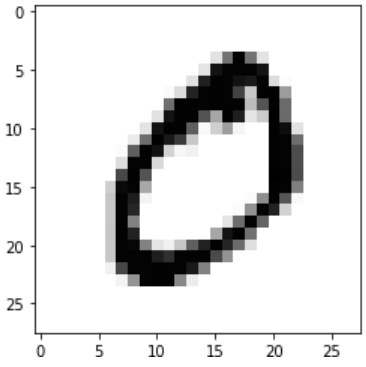
numpy.asfarray() – это функция библиотеки numpy, преобразующая текстовые строки в реальные числа и создающая массив этих чисел. Но почему текстовые строки преобразуются в числа? Если файл был прочитан как текстовый, то каждая строка или запись все еще остается текстом. Извлечение из них отдельных элементов, 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 28x28. Результирующий массив такой же размерности получает название image_array.
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 [18]: 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[18]: <matplotlib.image.AxesImage at 0x2aa4a9c1340>
```



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 [19]: 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[19]: <matplotlib.image.AxesImage at 0x2aa4b12c130>
```



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

2. Подготовка тренировочных данных MNIST

Когда мы уже знаем, 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.

```
In [24]: # 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    0.208    0.62729412 0.99223529 0.62729412 0.20411765
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    0.19635294 0.934
0.98835294 0.98835294 0.98835294 0.93011765 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    0.21964706 0.89129412 0.99223529 0.98835294 0.93788235
0.91458824 0.98835294 0.23129412 0.03329412 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    0.04882353 0.24294118 0.87964706
0.98835294 0.99223529 0.98835294 0.79423529 0.33611765 0.98835294
0.99223529 0.48364706 0.01     0.01     0.01     0.01
```

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

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 что именно мы должны получить на выходе нейронной сети? Должно ли это быть изображение ответа? В таком случае нам нужно иметь 28x28=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 [23]: # количество выходных узлов - 10 (пример)
         onodes = 10
         targets = numpy.zeros(onodes) + 0.01
         targets[int(all_values[0])] =0.99

In [25]: print(targets)

         [0.99 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01]
```

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.

```
In [1]: import numpy
        # библиотека scipy.special с сигмоидой expit ()
        import scipy.special
        # гарантировать размещение графики в данном блокноте,
        # а не в отдельном окне
        %matplotlib inline
```

In [2]:

```
# 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:
        # wil w21
        # wl2 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
```

In [3]:
```
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)
```

In [5]:
```
# загрузить в список тестовый набор данных CSV-файла набора MNIST
training_data_file = open("mnist_train.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 28x28, представляющему количество пикселей, из которых состоит изображение рукописной цифры.

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

В связи с этим следует сделать одно важное замечание: идеального общего метода для выбора количества скрытых узлов не существует. В действительности не существует и идеального метода выбора количества скрытых слоев. В настоящее время наилучшим подходом является

проведение экспериментов до тех пор, пока не будет получена конфигурация сети, оптимальная для задачи, которую вы пытаетесь решить.

Контрольные вопросы

1. В чем заключается метод градиентного спуска?
2. Назовите преимущества метода градиентного спуска?
3. Что означает функция ошибки (ошибка)? Как подсчитать ошибку?
4. Что представляют собой коэффициентом обучения нейронной сети?
5. Каким образом происходит обновление весовых коэффициентов в процессе обучения нейронной сети.
6. Что представляют собой набор рукописных цифр MNIST?
7. Что такое распознавание образов?
8. Что такое тренировочный набор?
9. Что такое тестовый набор?

Лабораторная работа 7
Тестирование нейронной сети для распознавания рукописных цифр MNIST.

Цель: провести тестирование разработанной нейронной сети для распознавания рукописных цифр MNIST.

Задание:

1. Изучить теоретическое введение.
2. Ответить на контрольные вопросы, приведенные в конце лабораторной работы.
3. Выполнить практические задания с использованием языка программирования Python.

4. Составить отчет по лабораторной работе. *Требования к отчету:*

Отчѐт предоставляется в электронном виде в текстовом редакторе MS Word. В отчѐте указываются:

1) Фамилия студента.
2) Порядковый номер и название лабораторной работы.
3) Цель работы.
4) Ответы на контрольные вопросы.
5) Описание решения выполненных заданий лабораторной работы, содержащее: *а) формулировку задания; в) скриншот выполненного задания, содержащий фамилию студента.*

Защита лабораторной работы осуществляется по отчѐту, представленному студентом и с демонстрацией задания на компьютере.

Тестирование нейронной сети

В данной работе проверим, как работает, и сделаем нейросеть на тестовом наборе 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_test.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.

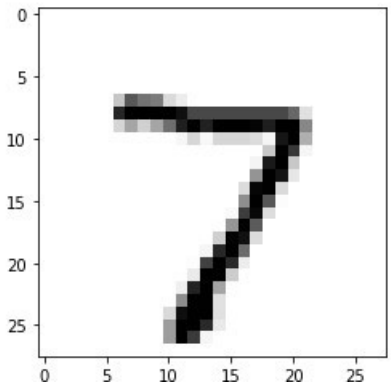
```
# загрузить в список тестовый набор данных CSV-файла набора MNIST
test_data_file = open("mnist_test.csv", 'r')
test_data_list = test_data_file.readlines()
test_data_file.close()
```

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

```
7
```

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

```
29]: <matplotlib.image.AxesImage at 0x251210f4b20>
```



Как видите, в качестве маркера первой записи тестового набора сеть определила символ “ 7” .

Не забудьте подключить библиотеку *matplotlib.pyplot*.

```
import numpy
# библиотека scipy.special с сигмоидой expit ()
import scipy.special
# гарантировать размещение графики в данном блокноте,
# а не в отдельном окне
import matplotlib.pyplot
%matplotlib inline
```

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! 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.

Далее 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.
Ознакомьтесь с приведенным ниже кодом.

```
# 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
```

Разберем данный код. 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” .

```
7 истинный маркер
7 ответ сети
2 истинный маркер
2 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
9 ответ сети
5 истинный маркер
5 ответ сети
```

Кроме того, в некоторых местах кода можно включить 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.

```
print (scorecard)
```

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

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)
```

```
эффективность =  0.9466
```

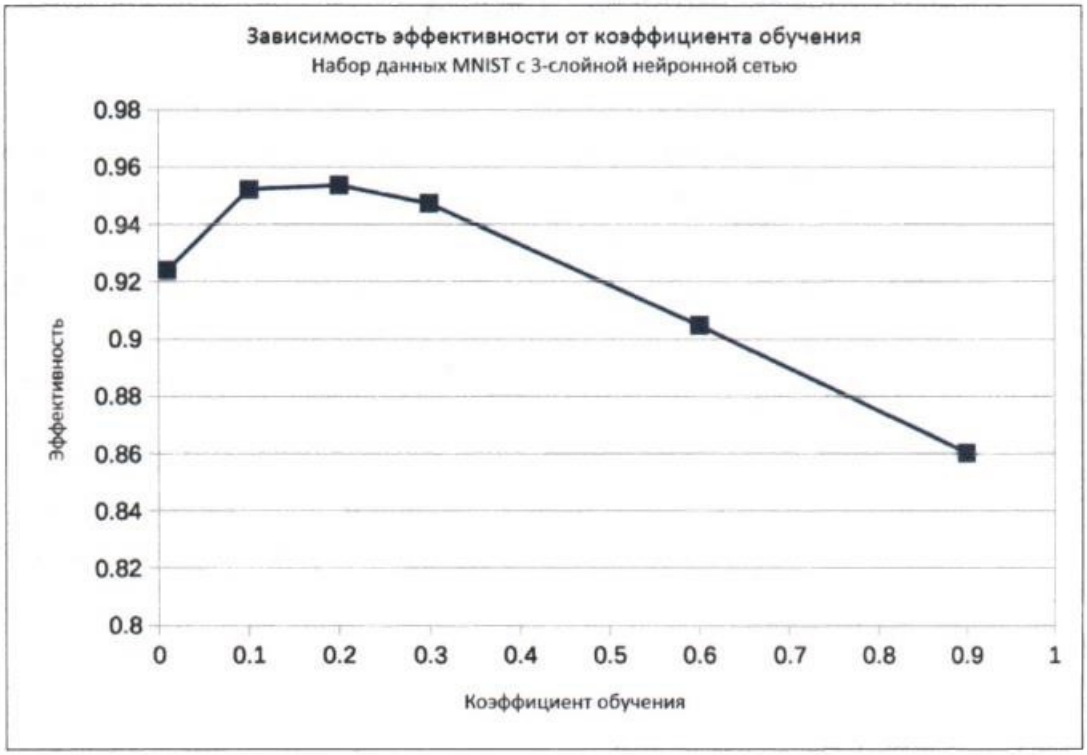
Эта доля рассчитывается как количество всех записей в журнале, содержащих “1” , деленное на общее количество записей (размер журнала). Вот каким получается результат.
В соответствии с результатами обучения нашей простой трехслойной нейронной сети с использованием полного набора, включающего 60 тысяч

примеров, и последующего тестирования на 10 тысячах записей показатель общей эффективности сети составляет 0,9466.
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.
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.

Улучшение результатов: настройка коэффициента обучения

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 эталонного теста, который проводился с использованием тысячи скрытых 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. Для получения более точных научных результатов 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.

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

При выполнении этого кода ваши оценки будут немного отличаться от приведенных, поскольку процесс в целом содержит элементы случайности. Ваш случайный выбор начальных значений весовых коэффициентов не будет совпадать с приведенным, а потому маршрут градиентного спуска для вашего кода будет другим.

Улучшение результатов: многократное повторение тренировочных циклов

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.

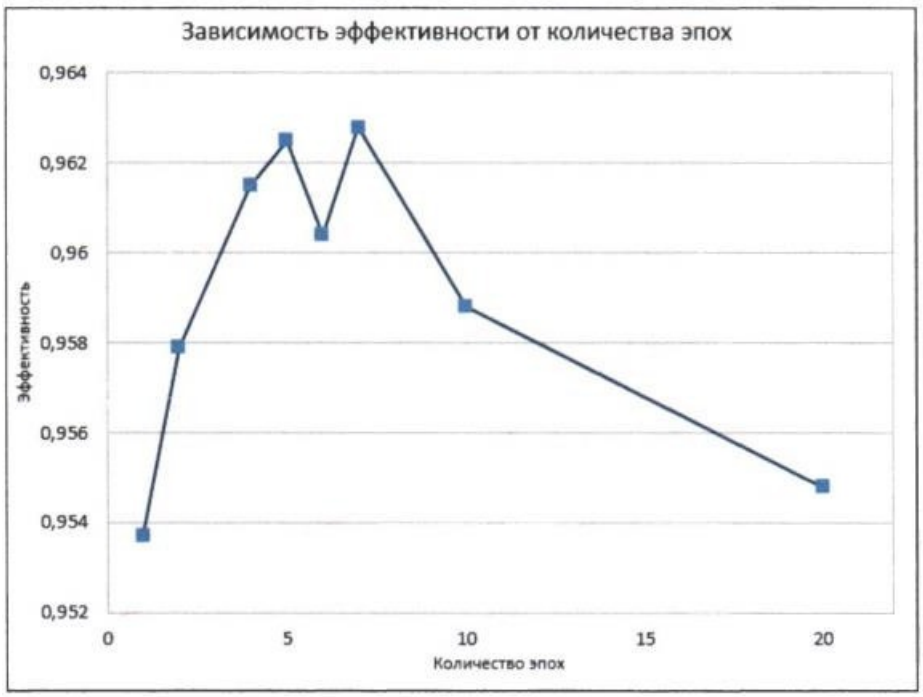
Посмотрим, что нам дадут две тренировочные эпохи. Для этого мы должны немного изменить код, предусмотрев в нем дополнительный цикл выполнения кода тренировки.

```
# 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.

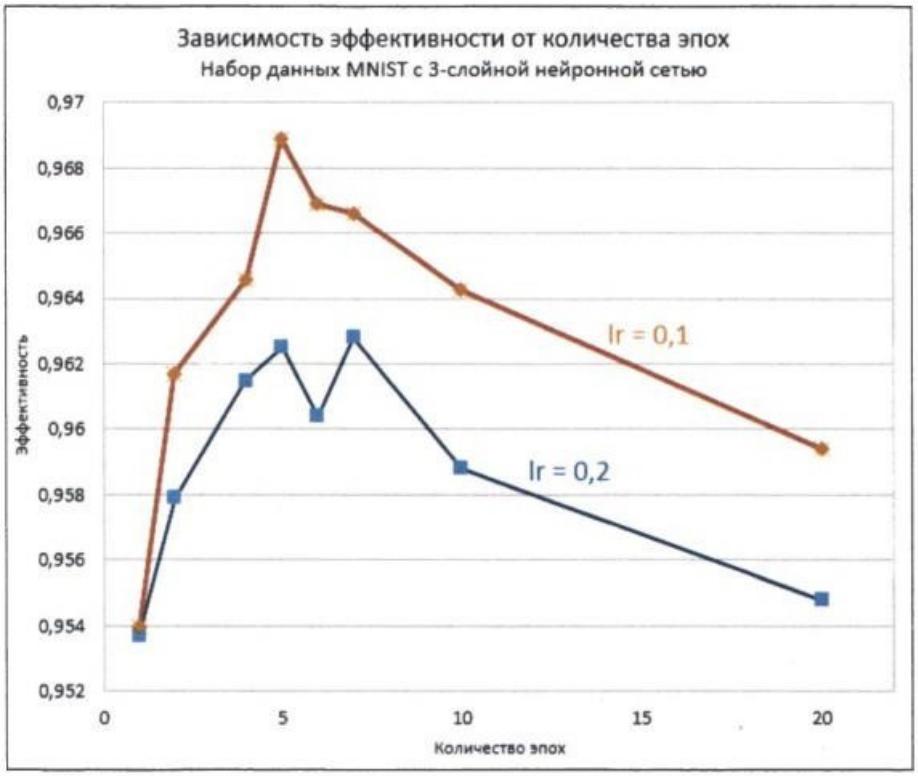


Теперь пиковое значение эффективности составляет 0,9628, или 96,28%, при семи эпохах.

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 мы оставили 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.

Изменение конфигурации сети

Один из факторов, влияние которых мы еще не исследовали, – конфигурация нейронной сети. Необходимо попробовать изменить количество узлов скрытого промежуточного слоя. Мы установили его равным 100.

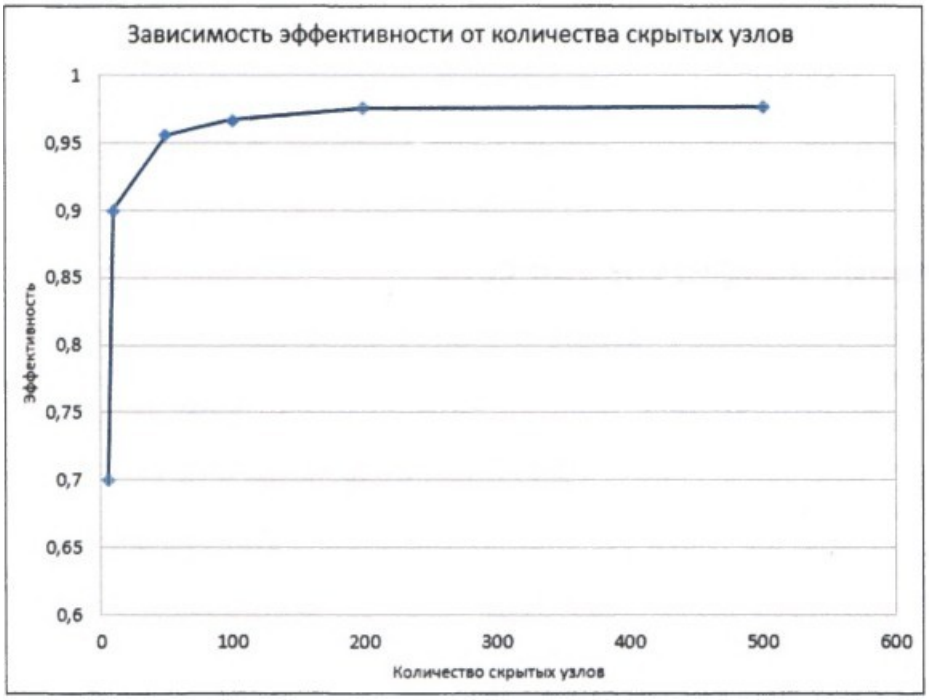
Прежде чем приступить к экспериментам с различными количествами скрытых узлов, давайте подумаем, что при этом может случиться. Скрытый слой как раз и является тем слоем, в котором происходит обучение. Вспомните, что узлы входного слоя принимают входные сигналы, а узлы выходного выдают ответ сети. Собственно,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

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. 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.

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

Итоги

Давая общую оценку проделанной нами работы, необходимо подчеркнуть, что при создании нейронной сети с помощью Python мы использовали лишь самые простые концепции.
И тем не менее с помощью нейронной сети нам, даже без использования каких-либо дополнительных математических ухищрений, удалось получить вполне достойные результаты.
Далее можно 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.**

Рекомендуемая литература:

1. Интеллектуальные информационные системы и технологии / Ю.Ю. Громов. - Тамбов : Издательство ФГБОУ ВПО «ТГТУ», 2013. - 244 с. - ISBN 978-5-8265-1178-7

2. Матвеев, М. Г. Модели и методы искусственного интеллекта. Применение в экономике / М.Г. Матвеев ; А.С. Свиридов ; Н.А. Алейникова. - Москва : Финансы и статистика, 2011. - 448 с. - ISBN 978-5-279-03279-2

3. Пальмов, С.В.
    Интеллектуальные системы и технологии Электронный ресурс : учебное пособие / С.В. Пальмов. - Самара : Поволжский государственный университет телекоммуникаций и информатики, 2017. - 195 с. - Книга находится в базовой версии ЭБС IPRbooks.

Дополнительная литература:

1. Аверченков, В. И. Система формирования знаний в среде Интернет : Монография / Аверченков В. И. - Брянск : Брянский государственный технический университет, 2012. - 181 с. - Книга находится в базовой версии ЭБС IPRbooks. - ISBN 5-89838-328-X

2. Богомолова, М.А.
    Экспертные системы (техника и технология проектирования) Электронный ресурс : учебно-методическое пособие / М.А. Богомолова. - Самара : Поволжский

государственный университет телекоммуникаций и информатики, 2015. - 47 с. - Книга находится в базовой версии ЭБС IPRbooks.

3. Сотник, С.Л.
    Проектирование систем искусственного интеллекта Электронный ресурс : учебное пособие / С.Л. Сотник. - Проектирование систем искусственного интеллекта,2021-01-23. - Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 228 c. - Книга находится в базовой версии ЭБС IPRbooks.

4. Ясницкий, Л. Н. Введение в искусственный интеллект : учебное пособие для студентов вузов, обучающихся по мат. направлениям и специальностям / Л.Н. Ясницкий. - 3-е изд., стер. - М. : Академия, 2010. - 176 с. : ил. - (Высшее профессиональное образование. Информатика и вычислительная техника). - Библиогр.: с. 170-173. - ISBN 978-5-7695-7042-1

Интернет-ресурсы:

1. Воройский Ф. С. Информатика. Энциклопедический словарь-справочник: введение в современные информационные и телекоммуникационные технологии в терминах и фактах. - М.: ФИЗМАТЛИТ, 2006. - 768 с. – Доступно: http://physics-for-students.ru/bookpc/informatika/slovar.zip

2. Иванов В. Основы искусственного интеллекта – https://libtime.ru/expertsystems/osnovy-iskusstvennogo-intellekta.html

3. Романов П.С. Основы искусственного интеллекта; Учебно-метод. пособие. – http://www.studfiles.ru/preview/2264160/

4. Сайт Основы ИИ – https://sites.google.com/site/osnovyiskusstvennogointellekta/

5. Соболь Б.В. Информатика: учебник/ Б.В. Соболь [и др.] – Изд. 3-е, дополн. и перераб. – Ростов н/Д: Феникс, 2007. – 446 с. – Доступно: http://physics-for-students.ru/bookpc/informatika/Sobol.rar

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания

по организации и проведению самостоятельной работы
по дисциплине
«СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА»
для студентов направления подготовки
37.03.02 «Конфликтология»
Направленность (профиль):
«Конфликт-менеджмент»

Ставрополь, 2026

СОДЕРЖАНИЕ

1. Общие положения.....5
2. Цель и задачи самостоятельной работы.....5
3. Порядок выполнения самостоятельной работы.....6
3.1. Методические рекомендации по работе с учебной литературой.....6
3.2. Методические рекомендации по подготовке к лабораторным занятиям.....7
3.3. Методические рекомендации по самопроверке знаний.....9
3.4. Методические рекомендации по написанию научных текстов (докладов, рефератов, эссе, научных статей и т.д.).....9
3.5. Методические рекомендации по выполнению исследовательских проектов.....12
4. Контроль самостоятельной работы.....15
5. Список литературы для выполнения СРС.....16

1. Общие положения

Самостоятельная работа – планируемая учебная, учебно-исследовательская, научно-исследовательская работа студентов, выполняемая во внеаудиторное (аудиторное) время по заданию и при методическом руководстве преподавателя, но без его непосредственного участия (при частичном непосредственном участии преподавателя, оставляющем ведущую роль за работой студентов).

Самостоятельная работа студентов (СРС) в ВУЗе является важным видом учебной и научной деятельности студента.

К основным видам самостоятельной работы студентов относятся:

– формирование и усвоение содержания конспекта лекций на базе рекомендованной лектором учебной литературы, включая информационные образовательные ресурсы (электронные учебники, электронные библиотеки и др.);

– написание докладов;

– подготовка к семинарам, практическим и лабораторным работам, их оформление;

– составление аннотированного списка статей из соответствующих журналов по отраслям знаний (педагогических, психологических, методических и др.);

– выполнение учебно-исследовательских работ, проектная деятельность;

– подготовка практических разработок и рекомендаций по решению проблемной ситуации;

– выполнение домашних заданий в виде решения отдельных задач, проведения типовых расчетов, расчетно-компьютерных и индивидуальных работ по отдельным разделам содержания дисциплин и т.д.;

– компьютерный текущий самоконтроль и контроль успеваемости на базе электронных обучающих и аттестующих тестов;

– выполнение курсовых работ (проектов) в рамках дисциплин;

– выполнение выпускной квалификационной работы и др.

Методика организации самостоятельной работы студентов зависит от структуры, характера и особенностей изучаемой дисциплины, объема часов на ее изучение, вида заданий для самостоятельной работы студентов, индивидуальных качеств студентов и условий учебной деятельности.

Процесс организации самостоятельной работы студентов включает в себя следующие этапы:

- подготовительный (определение целей, составление программы, подготовка методического обеспечения, подготовка оборудования);
- основной (реализация программы, использование приемов поиска информации, усвоения, переработки, применения, передачи знаний, фиксирование результатов, самоорганизация процесса работы);
- заключительный (оценка значимости и анализ результатов, их систематизация, оценка эффективности программы и приемов работы, выводы о направлениях оптимизации труда).

2. Цель и задачи самостоятельной работы

При организации СРС важным и необходимым условием становятся формирование умения самостоятельной работы для приобретения знаний, навыков и возможности организации учебной и научной деятельности. Целью самостоятельной работы студентов является овладение фундаментальными знаниями, профессиональными умениями и навыками деятельности по профилю, опытом творческой, исследовательской деятельности. Самостоятельная работа студентов способствует развитию самостоятельности,

ответственности и организованности, творческого подхода к решению проблем учебного и профессионального уровня.

Задачами СРС являются:

- систематизация и закрепление полученных теоретических знаний и практических умений студентов;
- углубление и расширение теоретических знаний;
- формирование умений использовать нормативную, правовую, справочную документацию и специальную литературу;
- развитие познавательных способностей и активности студентов: творческой инициативы, самостоятельности, ответственности и организованности;
- формирование самостоятельности мышления, способностей к саморазвитию, самосовершенствованию и самореализации;
- развитие исследовательских умений;
- использование материала, собранного и полученного в ходе самостоятельных занятий на семинарах, на практических и лабораторных занятиях, при написании курсовых и выпускной квалификационной работ, для эффективной подготовки к итоговым зачетам и экзаменам.

3. Порядок выполнения самостоятельной работы

3.1. Методические рекомендации по работе с учебной литературой

При работе с книгой необходимо подобрать литературу, научиться правильно ее читать, вести записи. Для подбора литературы в библиотеке используются алфавитный и систематический каталоги.

Важно помнить, что рациональные навыки работы с книгой - это всегда большая экономия времени и сил.

Правильный подбор учебников рекомендуется преподавателем, читающим лекционный курс. Необходимая литература может быть также указана в методических разработках по данному курсу.

Изучая материал по учебнику, следует переходить к следующему вопросу только после правильного уяснения предыдущего, описывая на бумаге все выкладки и вычисления (в том числе те, которые в учебнике опущены или на лекции даны для самостоятельного вывода).

При изучении любой дисциплины большую и важную роль играет самостоятельная индивидуальная работа.

Особое внимание следует обратить на определение основных понятий курса. Студент должен подробно разбирать примеры, которые поясняют такие определения, и уметь строить аналогичные примеры самостоятельно. Нужно добиваться точного представления о том, что изучаешь. Полезно составлять опорные конспекты. При изучении материала по учебнику полезно в тетради (на специально отведенных полях) дополнять конспект лекций. Там же следует отмечать вопросы, выделенные студентом для консультации с преподавателем.

Выводы, полученные в результате изучения, рекомендуется в конспекте выделять, чтобы они при перечитывании записей лучше запоминались.

Опыт показывает, что многим студентам помогает составление листа опорных сигналов, содержащего важнейшие и наиболее часто употребляемые формулы и понятия. Такой лист помогает запомнить формулы, основные положения лекции, а также может служить постоянным справочником для студента.

Чтение научного текста является частью познавательной деятельности. Ее цель – извлечение из текста необходимой информации. От того на сколько осознанна читающим собственная внутренняя установка при обращении к печатному слову (найти нужные

сведения, усвоить информацию полностью или частично, критически проанализировать материал и т.п.) во многом зависит эффективность осуществляемого действия.

Выделяют ***четыре основные установки в чтении научного текста***:

информационно-поисковый (задача – найти, выделить искомую информацию)

усваивающая (усилия читателя направлены на то, чтобы как можно полнее осознать и запомнить как сами сведения излагаемые автором, так и всю логику его рассуждений)

аналитико-критическая (читатель стремится критически осмыслить материал, проанализировав его, определив свое отношение к нему)

творческая (создает у читателя готовность в том или ином виде – как отправной пункт для своих рассуждений, как образ для действия по аналогии и т.п. – использовать суждения автора, ход его мыслей, результат наблюдения, разработанную методику, дополнить их, подвергнуть новой проверке).

Основные виды систематизированной записи прочитанного:

Аннотирование – предельно краткое связное описание просмотренной или прочитанной книги (статьи), ее содержания, источников, характера и назначения;

Планирование – краткая логическая организация текста, раскрывающая содержание и структуру изучаемого материала;

Тезирование – лаконичное воспроизведение основных утверждений автора без привлечения фактического материала;

Цитирование – дословное выписывание из текста выдержек, извлечений, наиболее существенно отражающих ту или иную мысль автора;

Конспектирование – краткое и последовательное изложение содержания прочитанного.

Конспект – сложный способ изложения содержания книги или статьи в логической последовательности. Конспект аккумулирует в себе предыдущие виды записи, позволяет всесторонне охватить содержание книги, статьи. Поэтому умение составлять план, тезисы, делать выписки и другие записи определяет и технологию составления конспекта.

Методические рекомендации по составлению конспекта:

1. Внимательно прочитайте текст. Уточните в справочной литературе непонятные слова. При записи не забудьте вынести справочные данные на поля конспекта;
2. Выделите главное, составьте план;
3. Кратко сформулируйте основные положения текста, отметьте аргументацию автора;
4. Законспектируйте материал, четко следуя пунктам плана. При конспектировании старайтесь выразить мысль своими словами. Записи следует вести четко, ясно.
5. Грамотно записывайте цитаты. Цитируя, учитывайте лаконичность, значимость мысли.

В тексте конспекта желательно приводить не только тезисные положения, но и их доказательства. При оформлении конспекта необходимо стремиться к емкости каждого предложения. Мысли автора книги следует излагать кратко, заботясь о стиле и выразительности написанного. Число дополнительных элементов конспекта должно быть логически обоснованным, записи должны распределяться в определенной последовательности, отвечающей логической структуре произведения. Для уточнения и дополнения необходимо оставлять поля.

Овладение навыками конспектирования требует от студента целеустремленности, повседневной самостоятельной работы.

Вопросы для собеседования по самостоятельно изученной литературе

1. Краткая история искусственного интеллекта (ИИ), машины и интеллект. Основные направления исследований в области ИИ.

2. Интеллектуальные системы общения: системы обработки текстов естественного языка, системы обучения с базами данных, диалоговые системы решения задач, системы речевого общения.

3. Разработка естественно-языковых интерфейсов и машинный перевод с одного языка на другой.

4. Интеллектуальные роботы.

5. Обучение и самообучение, искусственные нейронные сети.

6. Задачи распознавания образов.

7. Интеллектуальные игры и машинное творчество.

8. Представление задач и стратегии поиска их решения в пространстве состояний (поиск в глубину и ширину, слепой и эвристический поиск, поиск на игровых деревьях, минимаксный алгоритм, альфа-бета алгоритм и др.).

9. Представление знаний - центральная проблема ИИ. Процедурная и декларативная информация. Переход от обработки данных к оперированию со знаниями.

10. Отличительные особенности знаний от данных: внутренняя интерпретируемость, структурированность, связность, активность. Базы знаний. Нечеткие и неточные знания.

11. Открытость знаний в системах ИИ. Основные методы приобретения знаний. Инженерия знаний.

12. Основные модели представления знаний.

13. Формальные логические модели представления знаний. Проблема понимания естественного языка.

14. Сетевые модели представления знаний (семантические сети). Отображение множества информационных единиц во множество типов связей между ними. Отношения типа «абстрактное-конкретное» и «целое-часть». Иерархия наследования.

15. Фреймовые модели представления знаний. Понятия фрейма, терминального слота, протофрейма. Имя фрейма и имя слота, значение слота и тип данных слота. Фреймы-экземпляры. Механизм наследования.

16. Продукционная модель представления знаний. Системы продукций. База знаний (база фактов и правил).

17. Рабочая память. Механизм вывода: прямая и обратная цепочка рассуждений.

18. Достоинства и недостатки продукционной модели представления знаний.

19. Интегрированные модели представления знаний. Языки представления знаний.

20. Общая характеристика программных средств для разработки и реализации систем ИИ. Требования к программному обеспечению систем ИИ.

21. Инструментальные средства для создания систем ИИ.

22. Понятие экспертной системы (ЭС). Назначение, принципы построения и области применения ЭС. Организация знаний в ЭС.

23. Виды ЭС и типы решаемых ими задач. Инструментальные средства разработки ЭС. Оболочковые средства создания прототипов ЭС.

24. Гибридные логические и моделирующие ЭС, ЭС на базе нечеткой логики. Интеллектуальные информационные ЭС.

25. Структурная схема, основные компоненты, архитектура и режимы использования ЭС продукционного типа.

26. Основные этапы разработки ЭС. Жизненный цикл ЭС.

27. Языки программирования для решения задач ИИ.

3.2. Методические рекомендации по подготовке к практическим и лабораторным занятиям

Для того чтобы практические и лабораторные занятия приносили максимальную пользу, необходимо помнить, что упражнение и решение задач проводятся по вычитанному

на лекциях материалу и связаны, как правило, с детальным разбором отдельных вопросов лекционного курса. Следует подчеркнуть, что только после усвоения лекционного материала с определенной точки зрения (а именно с той, с которой он излагается на лекциях) он будет закрепляться на практических занятиях как в результате обсуждения и анализа лекционного материала, так и с помощью решения проблемных ситуаций, задач. При этих условиях студент не только хорошо усвоит материал, но и научится применять его на практике, а также получит дополнительный стимул (и это очень важно) для активной проработки лекции.

При самостоятельном решении задач нужно обосновывать каждый этап решения, исходя из теоретических положений курса. Если студент видит несколько путей решения проблемы (задачи), то нужно сравнить их и выбрать самый рациональный. Полезно до начала вычислений составить краткий план решения проблемы (задачи). Решение проблемных задач или примеров следует излагать подробно, вычисления располагать в строгом порядке, отделяя вспомогательные вычисления от основных. Решения при необходимости нужно сопровождать комментариями, схемами, чертежами и рисунками.

Следует помнить, что решение каждой учебной задачи должно доводиться до окончательного логического ответа, которого требует условие, и по возможности с выводом. Полученный ответ следует проверить способами, вытекающими из существа данной задачи. Полезно также (если возможно) решать несколькими способами и сравнить полученные результаты. Решение задач данного типа нужно продолжать до приобретения твердых навыков в их решении.

4.3. Методические рекомендации по самопроверке знаний

После изучения определенной темы по записям в конспекте и учебнику, а также решения достаточного количества соответствующих задач на практических занятиях и самостоятельно студенту рекомендуется, провести самопроверку усвоенных знаний, ответив на контрольные вопросы по изученной теме.

В случае необходимости нужно еще раз внимательно разобраться в материале.

Иногда недостаточность усвоения того или иного вопроса выясняется только при изучении дальнейшего материала. В этом случае надо вернуться назад и повторить плохо усвоенный материал. Важный критерий усвоения теоретического материала - умение решать задачи или пройти тестирование по пройденному материалу. Однако следует помнить, что правильное решение задачи может получиться в результате применения механически заученных формул без понимания сущности теоретических положений.

3.4. Методические рекомендации по написанию научных текстов (докладов, рефератов, эссе, научных статей и т.д.)

Перед тем, как приступить к написанию научного текста, важно разобраться, какова истинная цель вашего научного текста - это поможет вам разумно распределить свои силы и время.

Во-первых, сначала нужно определиться с идеей научного текста, а для этого необходимо научиться либо относиться к разным явлениям и фактам несколько критически (своя идея – как иная точка зрения), либо научиться увлекаться какими-то известными идеями, которые нуждаются в доработке (идея – как оптимистическая позиция и направленность на дальнейшее совершенствование уже известного). Во-вторых, научиться организовывать свое время, ведь, как известно, свободное (от всяких глупостей) время – важнейшее условие настоящего творчества, для него наконец-то появляется время. Иногда именно на организацию такого времени уходит немалая часть сил и талантов.

Писать следует ясно и понятно, стараясь основные положения формулировать четко и недвусмысленно (чтобы и самому понятно было), а также стремясь структурировать свой текст. Каждый раз надо представлять, что ваш текст будет кто-то читать и ему захочется сориентироваться в нем, быстро находить ответы на интересующие вопросы (заодно представьте себя на месте такого человека). Понятно, что работа, написанная «сплошным текстом» (без заголовков, без выделения крупным шрифтом наиболее важным мест и т, п.),

у культурного читателя должна вызывать брезгливость и даже жалость к автору (исключения составляют некоторые древние тексты, когда и жанр был иной и к текстам относились иначе, да и самих текстов было гораздо меньше – не то, что в эпоху «информационного взрыва» и соответствующего «информационного мусора»).

Объем текста и различные оформительские требования во многом зависят от принятых в конкретном учебном заведении порядков.

Реферат (доклад) - это самостоятельное исследование студентом определенной проблемы, комплекса взаимосвязанных вопросов.

Реферат не должна составляться из фрагментов статей, монографий, пособий. Кроме простого изложения фактов и цитат, в реферате должно проявляться авторское видение проблемы и ее решения.

Рассмотрим основные этапы подготовки реферата студентом.

Выполнение реферата начинается с выбора темы.

Затем студент приходит на первую консультацию к руководителю, которая предусматривает:

- обсуждение цели и задач работы, основных моментов избранной темы;
- консультирование по вопросам подбора литературы;
- составление предварительного плана.

Следующим этапом является работа с литературой. Необходимая литература подбирается студентом самостоятельно.

После подбора литературы целесообразно сделать рабочий вариант плана работы. В нем нужно выделить основные вопросы темы и параграфы, раскрывающие их содержание.

Составленный список литературы и предварительный вариант плана уточняются, согласуются на очередной консультации с руководителем.

Затем начинается следующий этап работы - изучение литературы. Только внимательно читая и конспектируя литературу, можно разобраться в основных вопросах темы и подготовиться к самостоятельному (авторскому) изложению содержания реферата. Конспектируя первоисточники, необходимо отразить основную идею автора и его позицию по исследуемому вопросу, выявить проблемы и наметить задачи для дальнейшего изучения данных проблем.

Систематизация и анализ изученной литературы по проблеме исследования позволяют студенту написать работу.

Рабочий вариант текста реферата предоставляется руководителю на проверку. На основе рабочего варианта текста руководитель вместе со студентом обсуждает возможности доработки текста, его оформление. После доработки реферат сдается на кафедру для его оценивания руководителем.

Требования к написанию реферата

Написание 1 реферата является обязательным условием выполнения плана СРС по любой дисциплине профессионального цикла.

Тема реферата может быть выбрана студентом из предложенных в рабочей программе или фонде оценочных средств дисциплины, либо определена самостоятельно, исходя из интересов студента (в рамках изучаемой дисциплины). Выбранную тему необходимо согласоваться с преподавателем.

Реферат должен быть написан научным языком.

Объем реферата должен составлять 20-25 стр.

Структура реферата:

- Введение (не более 3-4 страниц). Во введении необходимо обосновать выбор темы, ее актуальность, очертить область исследования, объект исследования, основные цели и задачи исследования.
- Основная часть состоит из 2-3 разделов. В них раскрывается суть исследуемой проблемы, проводится обзор мировой литературы и источников Интернет по предмету исследования, в котором дается характеристика степени разработанности проблемы и

авторская аналитическая оценка основных теоретических подходов к ее решению. Изложение материала не должно ограничиваться лишь описательным подходом к раскрытию выбранной темы. Оно также должно содержать собственное видение рассматриваемой проблемы и изложение собственной точки зрения на возможные пути ее решения.

- Заключение (1-2 страницы). В заключении кратко излагаются достигнутые при изучении проблемы цели, перспективы развития исследуемого вопроса
- Список использованной литературы (не меньше 10 источников), в алфавитном порядке, оформленный в соответствии с принятыми правилами. В список использованной литературы рекомендуется включать работы отечественных и зарубежных авторов, в том числе статьи, опубликованные в научных журналах в течение последних 3-х лет и ссылки на ресурсы сети Интернет.
- Приложение (при необходимости).

Требования к оформлению:

- текст с одной стороны листа;
- шрифт TimesNewRoman;
- кегль шрифта 14;
- межстрочное расстояние 1,5;
- поля: сверху 2,5 см, снизу – 2,5 см, слева - 3 см, справа 1,5 см;
- реферат должен быть представлен в сброшюрованном виде.

Порядок защиты реферата:

Защита реферата проводится на практических занятиях, после окончания работы студента над ним и исправления всех недочетов, выявленных преподавателем в ходе консультаций. На защиту реферата отводится 5-7 минут времени, в ходе которого студент должен показать свободное владение материалом по заявленной теме. При защите реферата приветствуется использование мультимедиа-презентации.

Оценка реферата

Реферат оценивается по следующим критериям:

• соблюдение требований к его оформлению;

• необходимость и достаточность для раскрытия темы приведенной в тексте реферата информации;

• умение студента свободно излагать основные идеи, отраженные в реферате;

• способность студента понять суть задаваемых преподавателем и сокурсниками вопросов и сформулировать точные ответы на них.

Критерии оценки:

Оценка «отлично» выставляется студенту, если в докладе студент исчерпывающе, последовательно, четко и логически стройно излагает материал; свободно справляется с задачами, вопросами и другими видами применения знаний; использует для написания доклада современные научные материалы; анализирует полученную информацию; проявляет самостоятельность при написании доклада.

Оценка «хорошо» выставляется студенту, если качество выполнения доклада достаточно высокое. Студент твердо знает материал, грамотно и по существу излагает его, не допуская существенных неточностей в ответе на вопросы по теме доклада.

Оценка «удовлетворительно» выставляется студенту, если материал доклада излагается частично, но пробелы не носят существенного характера, студент допускает неточности и ошибки при защите доклада, дает недостаточно правильные формулировки, наблюдаются нарушения логической последовательности в изложении материала.

Оценка «неудовлетворительно» выставляется студенту, если он не подготовил доклад или допустил существенные ошибки. Студент неуверенно излагает материал доклада, не отвечает на вопросы преподавателя.

Темы докладов

1. Моделирование биологических систем.
2. Автоматический компьютерный синтез речи по тексту. Методы синтеза речи.
3. Примеры систем обработки естественного языка.
4. Классификация инструментальных средств ЭС и организация знаний в ЭС.
5. Модели представления знаний: логическая, сетевая, фреймовая, продукционная.
6. Методы озвучивания речи.
7. История возникновения и современные направления исследований в области ИИ.
8. Классификация систем распознавания речи.
9. Распознавание символов. Шаблонные системы. Структурные системы. Признаковые системы.
10. Речевой вывод информации.
11. Машинный интеллект и робототехника.
12. Типы задач решаемые в ЭС.
13. Основные направления исследований в области искусственного интеллекта.
14. Предпосылки возникновения систем понимания естественного языка. Понимание в диалоге.
15. Распознавание рукописных текстов.

3.5. Методические рекомендации по выполнению исследовательских проектов

Исследовательская проектная работа – это групповая работа, для выполнения которой необходим выбор и приложение научной методики к поставленной задаче, получение собственного теоретического или экспериментального материала, на основании которого необходимо провести анализ и сделать выводы об исследуемом явлении. Выполнение проекта – это всегда коллективная, творческая Лабораторная работа, предназначенная для получения определенного продукта или научно-технического результата. Такая работа подразумевает четкое, однозначное формирование поставленной задачи, определение сроков выполнения намеченного, определение требований к разрабатываемому объекту.

Выполнение 1 группового проекта является обязательным условием выполнения самостоятельной работы по любой дисциплине профессионального цикла. Тема проектного задания может быть выбрана студентом из предложенных в рабочей программе или фонде оценочных средств дисциплины, либо определена самостоятельно, исходя из интересов студента (в рамках изучаемой дисциплины). Выбранную тему необходимо согласоваться с преподавателем.

Требования по выполнению и оформлению проекта

При выполнении проекта приветствуется работа в группе (2-3 человека). Проект – это исследовательская работа, в ходе которой студенты должны продемонстрировать владение навыками научного исследования, умения проводить анализ, обобщать информацию, делать выводы, предлагать свои решения проблемы, рассматриваемой в проекте.

При подготовке материалов проекта студенты должны продемонстрировать владение современными методами компьютерной обработки данных.

Критерии оценки работы участника проекта.

Для каждого из участников проекта оцениваются:

• профессиональные теоретические знания в соответствующей области;

• умение работать со справочной и научной литературой, осуществлять поиск необходимой информации в Интернет;

• умение работать с техническими средствами;

• умение пользоваться соответствующими выполняемому проекту информационными технологиями;

• умение готовить материалы проекта для презентации: составлять и редактировать тексты, формировать презентацию проекта;
• умение работать в команде;
• умение публично представлять результаты собственной деятельности;
• коммуникабельность, инициативность, творческие способности.

Критерии выставления оценки участникам проекта

Оценка	Профессиональные компетенции	Компетенции, связанные с использованием соответствующих выполняемому проекту технических средств и информационных технологий	Иные универсальные компетенции (коммуникабельность, инициативность, умение работать в «команде», управленческие навыки и т.д.)	Отчетность
«Отлично»	Работа выполнена на высоком профессиональном уровне. Представленный материал в основном фактически верен, допускаются негрубые фактические неточности. Студент свободно отвечает на вопросы, связанные с проектом.	Технические средства и информационные технологии освоены и использованы для реализации проекта полностью	Студент проявил инициативу, творческий подход, способность к выполнению сложных заданий, навыки работы в коллективе, организационные способности.	Проект представлен полностью и в срок.
«Хорошо»	Работа выполнена на достаточно высоком профессиональном уровне. Допущено до 4–5 фактических ошибок. Студент отвечает на вопросы, связанные с проектом, но недостаточно полно.	Обнаруживаются некоторые ошибки в использовании соответствующих технических средств и информационных технологий	Студент достаточно полно, но без инициативы и творческих находок выполнил возложенные на него задачи.	Проект представлен достаточно полно и в срок, но с некоторыми недоработками.
«Удовлетворительно»	Уровень недостаточно высок. Допущено до 8 фактических ошибок. Студент может ответить лишь на некоторые из заданных вопросов, связанных с	Обнаруживает недостаточное владение навыками работы с техническими средствами и соответствующими информационным	Студент выполнил большую часть возложенной на него работы.	Проект сдан со значительным опозданием (более недели) и не полностью

Оценка	Профессиональные компетенции	Компетенции, связанные с использованием соответствующих выполняемому проекту технических средств и информационных технологий	Иные универсальные компетенции (коммуникабельность, инициативность, умение работать в «команде», управленческие навыки и т.д.)	Отчетность
	проектом.	и технологиями		
«Неудовлетворительно»	Работа не выполнена или выполнена на низком уровне. Допущено более 8 фактических ошибок. Ответы на связанные с проектом вопросы обнаруживают непонимание предмета и отсутствие ориентации в материале проекта.	Навыков работы с техническими средствами нет, информационные технологии не освоены	Студент практически не работал, не выполнил свои задачи или выполнил лишь отдельные не существенные поручения в групповом проекте.	Проект не сдан.

Студенты должны: защитить проект в режиме презентации, предъявить файлы выполненного проекта, уметь рассказать о технологиях, использованных ими при выполнении проекта, дать оценку работы каждого члена группы (*если проект групповой*).

Примерные темы индивидуальных творческих проектов

1. Проект «Электронный словарь»;
2. Проект «Экспертная система для диагностики здоровья»;
3. Проект «Лингвистическая экспертная система»;
4. Проект «Экспертная система для диагностики устройства»;
5. Проект «Обучающая экспертная система».

4. Контроль самостоятельной работы

Контроль самостоятельной работы проводится преподавателем в аудитории.

Предусмотрены следующие виды контроля: собеседование, оценка реферата, оценка презентации, оценка участия в круглом столе, оценка выполнения проекта.

Подробные критерии оценивания компетенций приведены в Фонде оценочных средств для проведения текущей и промежуточной аттестации.

5. Список литературы для выполнения СРС

Рекомендуемая литература:

1. Пальмов, С.В.
    Интеллектуальные системы и технологии Электронный ресурс : учебное пособие / С.В. Пальмов. - Самара : Поволжский

государственный университет телекоммуникаций и информатики, 2017. - 195 с. - Книга находится в базовой версии ЭБС IPRbooks.

2. Сысоев, Д. В. Введение в теорию искусственного интеллекта : учебное пособие / Д. В. Сысоев, О. В. Курипта, Д. К. Проскурин. — Москва : Ай Пи Ар Медиа, 2021. — 170 с. — ISBN 978-5-4497-1092-5. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/108282.html

8.1.2. Перечень дополнительной литературы:

1. Аверченков, В. И. Система формирования знаний в среде Интернет : Монография / Аверченков В. И. - Брянск : Брянский государственный технический университет, 2012. - 181 с. - Книга находится в базовой версии ЭБС IPRbooks. - ISBN 5-89838-328-X

2. Павлов, С. И. Системы искусственного интеллекта : учебное пособие : [16+] / С. И. Павлов. – Томск : Томский государственный университет систем управления и радиоэлектроники, 2011. – Часть 1. – 175 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=208933

3. Павлов, С. И. Системы искусственного интеллекта : учебное пособие / С. И. Павлов. – Томск : Томский государственный университет систем управления и радиоэлектроники, 2011. – Часть 2. – 194 с. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=208939

4. Сергеев, Н. Е. Системы искусственного интеллекта : учебное пособие : [16+] / Н. Е. Сергеев. – Таганрог : Южный федеральный университет, 2016. – Часть 1. – 123 с. : схем., ил., табл. – Режим доступа: по подписке. – URL: https://biblioclub.ru/index.php?page=book&id=493307

5. Сотник, С. Л. Проектирование систем искусственного интеллекта : учебное пособие / С. Л. Сотник. — 3-е изд. — Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), Ай Пи Ар Медиа, 2021. — 228 с. — ISBN 978-5-4497-0868-7. — Текст : электронный // Цифровой образовательный ресурс IPR SMART : [сайт]. — URL: https://www.iprbookshop.ru/102054.html.

Интернет-ресурсы:

1. Воройский Ф. С. Информатика. Энциклопедический словарь-справочник: введение в современные информационные и телекоммуникационные технологии в терминах и фактах. - М.: ФИЗМАТЛИТ, 2006. - 768 с. – Доступно: http://physics-for-students.ru/bookpc/informatika/slovar.zip

2. Иванов В. Основы искусственного интеллекта – https://libtime.ru/expertsystems/osnovy-iskusstvennogo-intellekta.html

3. Романов П.С. Основы искусственного интеллекта; Учебно-метод. пособие. – http://www.studfiles.ru/preview/2264160/

4. Сайт Основы ИИ – https://sites.google.com/site/osnovyiskusstvennogointellekta/

5. Соболь Б.В. Информатика: учебник/ Б.В. Соболь [и др.] – Изд. 3-е, дополн. и перераб. – Ростов н/Д: Феникс, 2007. – 446 с. – Доступно: http://physics-for-students.ru/bookpc/informatika/Sobol.rar