

Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Архитектура вычислительных систем (ABC)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту на тему

“Реализация нейронной сети для распознавания рукописных цифр”

Выполнил: студент гр. 953501

Кременевский В.С.

Руководитель: ст. преподаватель
Шиманский В. В.

Минск 2021

Заведующий кафедрой ИИТП
_____ Волорова Н. А.
«___» _____ 2021 г.

ЗАДАНИЕ по курсовому проекту

Группа 953501

Студенту Кременевскому Владиславу Сергеевичу

1. Тема проекта: Реализация нейронной сети для распознавания рукописных цифр

2. Сроки сдачи студентом законченного проекта: 21.12.2021 г.

3. Исходные данные к проекту: Для написания курсового проекта был выбран язык программирования Python. В качестве среды разработки была выбрана интегрированная среда разработки PyCharm, а также интерактивные jupyter notebooks.

4. Содержание расчетно-пояснительной записи (перечень подлежащих разработке вопросов):

Введение

Раздел 1. Постановка задачи

Раздел 2. Проектирование задачи

Раздел 3. Программная реализация

Раздел 4. Демонстрация работы

Заключение. Список использованных источников. Приложение

5. Перечень графического материала (с указанием обязательных чертежей и графиков):

6. Консультанты по проекту: Шиманский В. В.

7. Дата выдачи задания: 15.09.2021 г

8. Календарный график работы над проектом на весь период проектирования (с указанием сроков выполнения и трудоемкости отдельных этапов):

№ п/п	Наименование этапов курсового проекта	Срок выполнения этапов проекта	Примечание
1.	Разработка общего положения	10.10.2021	30%
2.	Разработка программного продукта	08.11.2021	65%
3.	Разработка инструкции по эксплуатации	20.11.2021	100%
4.	Защита курсового проекта	21.12.2021	Согласно графику

Руководитель _____ (Шиманский В. В.)

Задание принял к исполнению 15.09.2021 _____ (_____)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	5
1. Анализ предметной области	6
1.1 Ключевые концепции	6
1.2 Перцептрон	7
1.3 Сигмоидные нейроны	12
1.4 Архитектура нейросетей	17
1.5. Сеть для классификации рукописных цифр	19
2. Алгоритм обучения нейронных сетей	24
2.1 Градиентный спуск	24
2.2 Обучение нейросети. Метод обратного распространения ошибки	30
3. Реализация нейронной сети	38
3.1 Основные методы	38
3.2 Проверка работы	41
4. Оптимизирование нейронной сети	44
4.1 Реализация полностью матричного градиентного спуска	44
4.2 Кросс энтропийная функция потерь	45
4.3 SoftMax функция активации	50
5. Переобучение нейронных сетей	51
5.1 Понятие переобучения	51
5.2 Регуляризация как способ устранения переобучения	54
6. Инициализация весов сети	63
6.1 Проблема рандомной инициализации	63
6.2 Новый подход инициализации весов	64
7. Демонстрация работы	67
Заключение	83
Список использованной литературы	84

ВВЕДЕНИЕ

Нейронные сети – одна из прекраснейших программных парадигм, когда-либо придуманных человеком. При стандартном подходе к программированию мы сообщаем компьютеру, что делать, разбиваем большие задачи на множество малых, точно определяем задачи, которые компьютеру будет легко исполнить. В случае с нейронными сетями мы, наоборот, не говорим компьютеру, как решать задачу. Он сам обучается этому на основе «наблюдений» за данными, «придумывая» собственное решение поставленной задачи.

Автоматическое обучение на основе данных звучит многообещающе. Однако до 2006 года люди не знали, как обучать Нейронные сети (НС) так, чтобы они могли превзойти более традиционные подходы, за исключением нескольких особых случаев. В 2006 были открыты техники обучения глубоких нейросетей (ГНС). Теперь эти техники известны, как глубокое обучение (ГО). Их продолжали разрабатывать, и сегодня ГНС и ГО достигли потрясающих результатов во многих важных задачах, связанных с компьютерным зрением, распознаванием речи и обработки естественного языка. В крупных масштабах их развёртывают такие компании, как Google, Microsoft и Facebook.

Цель данного проекта – овладеть ключевыми концепциями нейросетей, включая и современные техники глубокого обучения, написать код, использующий Нейронные сети и Глубокое Обучение для решения сложных задач распознавания закономерностей, непосредственно для распознавания рукописных цифр.

1. Анализ предметной области

1.1 Ключевые концепции

Зрительная система человека – одно из чудес света. Рассмотрим следующую последовательность рукописных цифр (см. рис 1):

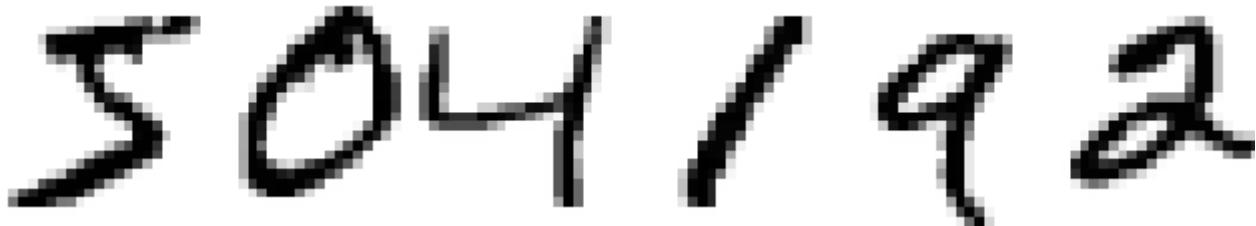


Рис 1. Пример рукописных цифр

Большинство людей без труда прочтут их, как 504192. Но эта простота обманчива. В каждом полушарии мозга у человека есть первичная зрительная кора, также известная, как V1, в которой содержится 140 млн нейронов и десятки миллиардов связей между ними. При этом в зрении человека участвует не только V1, но целая последовательность участков мозга — V2, V3, V4 и V5 — которые занимаются всё более сложной обработкой изображений. Люди носят в своей голове суперкомпьютер, настроенный эволюцией в течение сотен миллионов лет, и прекрасно адаптированный для понимания видимого мира. Распознавать рукописные цифры не так-то легко. Просто мы, люди, потрясающе, удивительно хорошо распознаём то, что показывают нам наши глаза. Но почти вся эта работа проводится бессознательно. И обычно мы не придаём значения тому, какую сложную задачу решают наши зрительные системы.

Сложность распознавания зрительных закономерностей становится очевидной, когда пытаемся написать компьютерную программу для распознавания таких цифр, как приведённые выше. То, что кажется лёгким в нашем исполнении, внезапно оказывается чрезвычайно сложным. Простое понятие о том, как мы распознаём формы — «у девятки сверху петля, и вертикальная чёрточка справа внизу» — оказывается вовсе не таким простым для алгоритмического выражения. Пытаясь сформулировать эти правила чётко, быстро вязнешь в трясине исключений, подводных камней и особых случаев.

НС подходят к решению задачи по-другому. Идея в том, чтобы взять множество рукописных цифр (см.рис 2), известных, как обучающие примеры.

0	4	1	9	2	1	3	1	4	3
5	3	6	1	7	2	8	6	9	4
0	9	1	1	2	4	3	2	7	3
8	6	9	0	5	6	0	7	6	1
8	7	9	3	9	8	5	9	3	3
0	7	4	9	8	0	9	4	1	4
4	6	0	4	5	6	1	0	0	1
7	1	6	3	0	2	1	1	7	9
0	2	6	7	8	3	9	0	4	6
7	4	6	8	0	7	8	3	1	5

Рис 2. Обучающие примеры

А далее разработать систему, способную обучаться на этих примерах. Иначе говоря, НС использует примеры для автоматического построения правил распознавания рукописных цифр. Более того, увеличивая количество обучающих примеров, сеть может больше узнать о рукописных цифрах и улучшить свою точность. Несмотря на то что, сверху приведено всего 100 обучающих примеров, возможно, у нас получится создать более хороший распознаватель рукописных цифр, используя тысячи или даже миллионы и миллиарды обучающих примеров.

1.2 Перцептрон

Что такое нейросеть? Для начала рассмотрим один тип искусственного нейрона, который называется перцептрон. Перцептроны придумал в 1950-60-х учёный Фрэнк Розенблattt, вдохновившись ранней работой Уоррена Мак-Каллока и Уолтера Питтса. Сегодня чаще используются другие модели искусственных нейронов – большинстве современных работ по НС в основном используют сигмоидную модель нейрона. Вскоре мы с ней познакомимся. Но чтобы понять, почему сигмоидные нейроны определяются именно так, стоит потратить время на разбор перцептрана.

Как работают перцептраны? Перцептрон принимает на вход несколько двоичных чисел x_1, x_2 (см. рис 3) и выдаёт одно двоичное число.

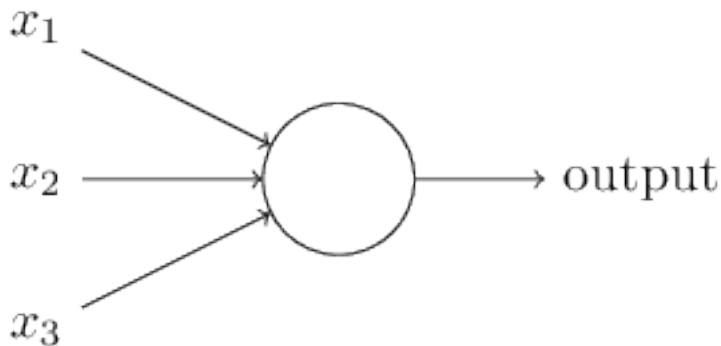


Рис 3. Пример перцептрана

В данном примере у перцептрана есть три числа на входе, x_1 , x_2 , x_3 . В общем случае их может быть больше или меньше. Розенблatt предложил простое правило для вычисления результата. Он ввёл веса, w_1 , w_2 - вещественные числа, выражающие важность соответствующих входных чисел для результатов. Выход нейрона, 0 или 1, определяется тем, меньше или больше некого порога . Как и веса, порог – вещественное число, параметр нейрона. Если математическими терминами:

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq threshold \\ 1 & \text{if } \sum_j w_j x_j > threshold \end{cases}$$

Такова базовая математическая модель. Перцептран можно представить себе, как устройство, принимающее решение, взвешивая свидетельства. Приведем не очень реалистичный, но простой пример. Допустим, наступают выходные, и мы слышали, что в вашем городе пройдёт фестиваль сыра. Мы любим сыр, и пытаемся решить, идти ли на фестиваль, или нет. Мы можем принять решение, взвесив три фактора:

- A. Хорошая ли погода?
- B. Хочет ли идти с вами ваш партнёр?
- C. Далеко ли фестиваль от остановки общественного транспорта?

Три этих фактора можно представить двоичными переменными x_1 , x_2 , x_3 . К примеру, $x_1 = 1$, если погода хорошая, а 0 – если плохая. $x_2 = 1$, если ваш партнёр хочет пойти, и 0 – если нет. То же для x_3 .

Теперь, допустим, мы фанатеем от сыра настолько, что готовы ехать на фестиваль, даже если вашего партнёра это не интересует, и до него трудно добраться. Но, возможно, мы просто ненавидим плохую погоду, и в случае

непогоды на фестиваль не пойдём. Мы можем использовать перцептроны для моделирования такого процесса принятия решения. Один из способов – выбрать вес $w_1 = 6$ для погоды, и $w_2 = 2$, $w_3 = 2$ для других условий. Большее значение w_1 говорит о том, что погода имеет для нас значение гораздо большее, чем то, присоединится ли к нам наш партнёр, или близость фестиваля к остановке. Наконец, допустим, мы выбираем для перцептрана порог 5. С такими вариантами перцептран реализует нужную модель принятия решений, выдавая 1, когда погода хорошая, и 0, когда она плохая. Желание партнёра и близость остановки не влияют на выходное значение.

Изменяя веса и порог, мы можем получить разные модели принятия решений. К примеру, допустим, возьмём порог 3. Тогда перцептран решит, что нам нужно идти на фестиваль, либо когда погода хорошая, либо когда фестиваль находится рядом с остановкой и наш партнёр согласен идти с нами. Иначе говоря, модель получается другой. Понижение порога означает, что нам больше хочется пойти на фестиваль.

Очевидно, перцептран не является полной моделью принятия решений человеком, но этот пример показывает, как перцептран может взвешивать разные виды свидетельств, чтобы принимать решения. Кажется возможным, что сложная сеть (см. рис 4) перцептранов может принимать весьма сложные решения.

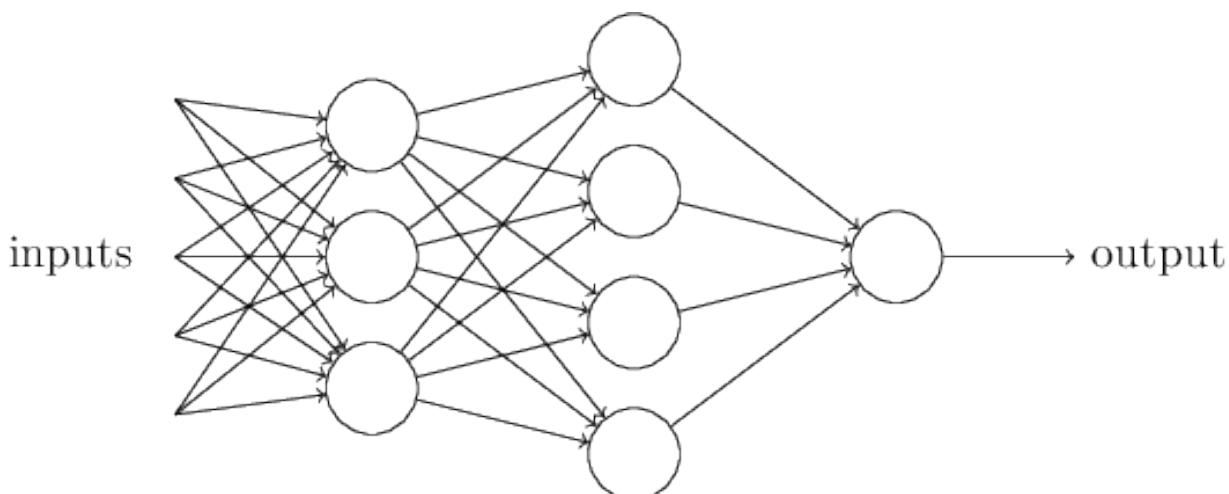


Рис 4. Сеть перцептранов

В этой сети первый столбец перцептранов – то, что мы называем первым слоем перцептранов – принимает три очень простых решения, взвешивая входные свидетельства. Каждый из перцептранов второго слоя принимает решение, взвешивая результаты первого слоя принятия решений. Таким способом перцептран второго слоя может принять решение на более сложном и абстрактном уровне, по сравнению с перцептраном первого слоя. А ещё более сложные решения могут принимать перцептраны на третьем слое. Таким способом многослойная сеть перцептранов может заниматься

принятием сложных решений.

Кстати, когда мы определяли перцептрон, было оговорено, что у него есть только одно выходное значение. Но в сети наверху перцептроны выглядят так, будто у них есть несколько выходных значений. На самом деле, выход у них только один. Множество выходных стрелок – просто удобный способ показать, что выход перцептрана используется как вход нескольких других перцептронов.

Попробуем упростить описание перцептронов.

Условие $\sum w_j x_j > threshold$ неуклюжее, и мы можем договориться о двух изменениях записи для её упрощения. Первое – записывать сумму как скалярное произведение: $\sum w_j x_j = w^* x$, где w и x – векторы, компонентами которых служат веса и входные данные, соответственно. Второе – перенести порог в другую часть неравенства, и заменить его значением, известным, как смещение перцептрана ($b = -threshold$). Используя смещение вместо порога, мы можем переписать правило перцептрана:

$$output = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \quad (1)$$

Смещение можно представлять, как меру того, насколько легко получить от перцептрана значение 1 на выходе. Или, в биологических терминах, смещение – мера того, насколько просто заставить перцептран активироваться. Перцептрану с очень большим смещением крайне легко выдать 1. Но с очень большим отрицательным смещением это сделать трудно. Очевидно, введение смещения – это небольшое изменение в описании перцептронов. Далее мы не будем использовать порог, а всегда будем использовать смещение.

Мы описали перцептраны с точки зрения метода взвешивания свидетельств с целью принятия решения. Ещё один метод их использования – вычисление элементарных логических функций, которые мы обычно считаем основными вычислениями, таких, как AND, OR и NAND. Допустим, к примеру, что у нас есть перцептран с двумя входами (см рис 5), вес каждого из которых равен -2, а его смещение равно 3. Вот он:

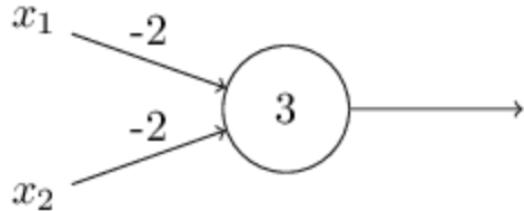


Рис 5. Перцептрон с весами

Входные данные 0 и 0 дают на выходе 1, поскольку $(-2)*0+(-2)*0+3=3$ больше нуля. Те же вычисления говорят, что входные данные 01 и 10 дают 1. Но 11 на входе даёт 0 на выходе, поскольку $(-2)*1+(-2)*1+3=-1$, меньше нуля. Поэтому наш перцептрон реализует функцию NAND!

Этот пример показывает, что можно использовать перцептроны для вычисления базовых логических функций. На самом деле, мы можем использовать сети перцепtronов для вычисления вообще любых логических функций. Дело в том, что логический вентиль NAND универсален для вычислений – на его основе можно строить любые вычисления. К примеру, можно использовать вентили NAND для создания контура (см. рис 6), складывающего два бита, x_1 и x_2 . Для этого нужно вычислить побитовую сумму , а также флаг переноса, который равняется 1, когда оба x_1 и x_2 равны 1 – то есть, флаг переноса является просто результатом побитового умножения x_1x_2 :

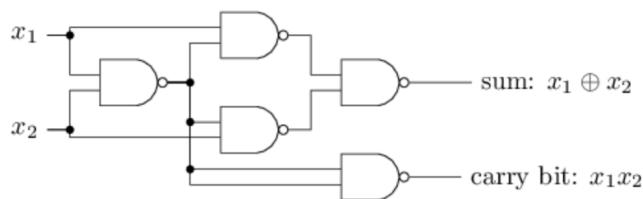


Рис 6. Контур складывающий два бита

Чтобы получить эквивалентную сеть из перцепtronов, мы заменим все вентили NAND перцепtronами с двумя входами, вес каждого из которых равен -2 , и со смещением 3 . Вот получившаяся сеть:

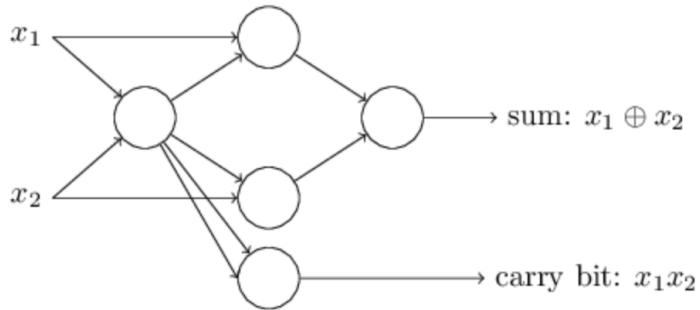


Рис 7. Контур в виде сети

Пример с сумматором (см. рис 7) демонстрирует, как можно использовать сеть из перцептронов для симуляции контура, содержащего множество вентилей NAND. А поскольку эти вентили универсальны для вычислений, следовательно, и перцептроны универсальны для вычислений.

Вычислительная универсальность перцептронов одновременно обнадёживает и разочаровывает. Обнадёживает она, гарантируя, что сеть из перцептронов может быть настолько же мощной, насколько любое другое вычислительное устройство. Разочаровывает, создавая впечатление, что перцептроны – всего лишь новый тип логического вентиля NAND.

Однако на самом деле ситуация лучше. Оказывается, что можно разработать обучающие алгоритмы, способные автоматически подстраивать веса и смещения сети из искусственных нейронов. Эта подстройка происходит в ответ на внешние стимулы, без прямого вмешательства программиста. Эти обучающие алгоритмы позволяют нам использовать искусственные нейроны способом, радикально отличным от обычных логических вентилей. Вместо того, чтобы явно прописывать контур из вентилей NAND и других, наши нейросети могут просто обучиться решать задачи, иногда такие, для которых было бы чрезвычайно сложно напрямую спроектировать обычный контур.

1.3 Сигмоидные нейроны

Обучающие алгоритмы – это прекрасно. Однако как разработать такой алгоритм для нейросети? Допустим, у нас есть сеть перцептронов, которую мы хотим использовать для обучения решения задачи. Допустим, входными данными сети могут быть пиксели отсканированного изображения рукописной цифры. И мы хотим, чтобы сеть узнала веса и смещения, необходимые для правильной классификации цифры. Чтобы понять, как может работать такое обучение, представим, что мы немного меняем некий вес (или смещение) в сети (см. рис 8). Мы хотим, чтобы это небольшое изменение привело к небольшому изменению выходных данных сети. Как мы скоро увидим, это свойство делает возможным обучение. Схематично мы хотим следующего:

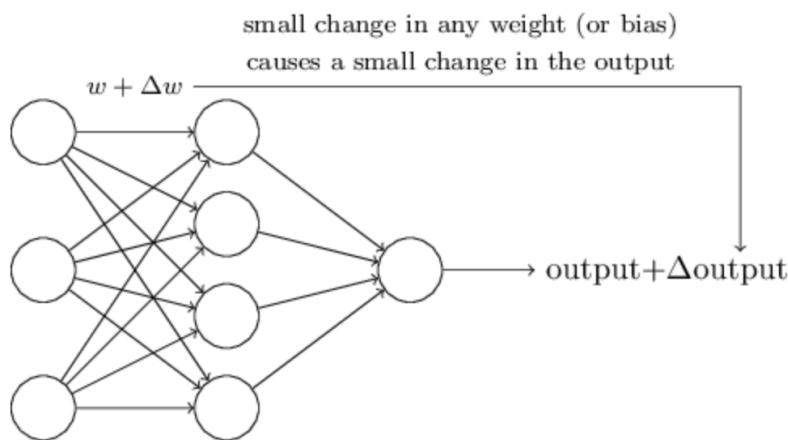


Рис 8. Пример зависимости выхода от весов в сети

Если бы небольшое изменение веса (или смещения) приводило бы к небольшому изменению результата на выходе, мы могли бы изменять веса и смещения, чтобы наша сеть вела себя чуть ближе к желаемому. К примеру, допустим, что сеть неправильно отнесла изображение к «8», хотя должна была к «9». Мы могли бы разобраться, как внести небольшое изменение в веса и смещения, чтобы сеть немного ближе подобралась к классификации изображения, как «9». А потом мы бы повторяли это, изменяя веса и смещения снова и снова, чтобы получать всё лучший и лучший результат. Сеть бы училась.

Проблема в том, что если в сети есть перцептроны, такого не происходит. Небольшое изменение весов или смещения любого перцептрана иногда может привести к изменению его выхода на противоположный, допустим, с 0 на 1. Такой переворот может изменить поведение остальной сети очень сложным образом. И даже если теперь наша «9» и будет правильно распознана, поведение сети со всеми остальными изображениями, вероятно, полностью изменилось таким образом, который сложно контролировать. Из-за этого сложно представить, как можно постепенно подстраивать веса и смещения, чтобы сеть постепенно приближалась к желаемому поведению. Возможно, существует некий хитроумный способ обойти эту проблему. Но нет никакого простого решения задачи обучения сети из перцептронов.

Эту проблему можно обойти, введя новый тип искусственного нейрона под названием сигмоидный нейрон(СН). Они похожи на перцептроны, но изменены так, чтобы небольшие изменения весов и смещений приводили только к небольшим изменениям выходных данных. Это основной факт, который позволит сети из сигмоидных нейронов обучаться.

Опишем сигмоидный нейрон (см. рис 9). Нарисуем его так же, как перцептроны:

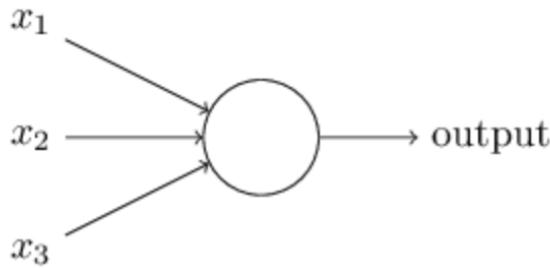


Рис 9. Простейший сигмоидный нейрон

У него точно так же есть входные данные x_1, x_2 . Но вместо того, чтобы приравниваться к 0 или 1, эти входы могут иметь любое значение в промежутке от 0 до 1. К примеру, величина 0,638 будет допустимым значением входных данных для сигмоидного нейрона (СН). Так же, как у перцептрана, у СН есть веса для каждого входа, w_1, w_2, \dots и общее смещение b . Но его выходным значением будет не 0 или 1. Это будет $\sigma(w \cdot x + b)$, где σ — это сигмоида (2).

Иногда σ называют логистической функцией, а этот класс нейронов — логистическими нейронами. Эту терминологию используют многие люди, работающие с нейросетями. Однако будем придерживаться сигмоидной терминологии.

Определяется функция так:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \quad (2)$$

В нашем случае выходное значение сигмоидного нейрона с входными данными x_1, x_2 весами w_1, w_2 и смещением b будет считаться, как:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)} \quad (3)$$

На первый взгляд, СН кажутся совсем не похожими на нейроны. Алгебраический вид сигмоиды может показаться запутанным и малопонятным. На самом деле между перцептранами и СН есть много общего, и алгебраическая форма сигмоиды оказывается больше технической подробностью, нежели серьёзным барьером к пониманию.

Чтобы понять сходство с моделью перцептрона, допустим, что $z \equiv w \cdot x + b$ – большое положительное число. Тогда $e^{-z} \approx 0$, поэтому $\sigma(z) \approx 1$. Иначе говоря, когда $z = w \cdot x + b$ большое и положительное, выход СН примерно равен 1, как у перцептрона. Допустим, что $z = w \cdot x + b$ большое со знаком минус. Тогда $e^{-z} \rightarrow \infty$, а $\sigma(z) \approx 0$. Так что при больших z со знаком минус поведение СН тоже приближается к перцептрону. И только когда $w \cdot x + b$ имеет средний размер, наблюдаются серьёзные отклонения от модели перцептрона.

Как понять вид функции $\sigma(z)$? На самом деле, точная форма σ не так уж важна – важна форма функции (см. рис 10). Вот она:

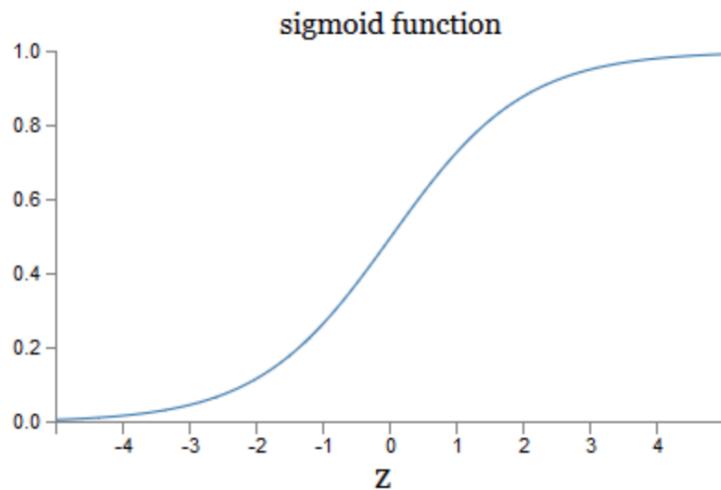


Рис 10. Функция сигмоиды

Это сглаженный вариант ступенчатой функции (см. рис 11):

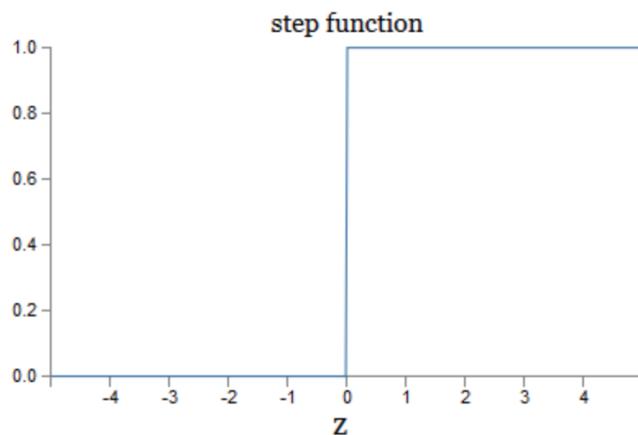


Рис 11. Ступенчатая функция активации

Если бы σ была ступенчатой, тогда СН был бы перцептроном, поскольку у него на выходе наблюдались бы 0 или 1 в зависимости от знака

$w \cdot x + b$ (ну, на самом деле при $z = 0$ перцептрон выдаёт 0, а ступенчатая функция – 1, так что в одной этой точке функцию пришлось бы поменять).

Используя реальную функцию σ , мы получаем сглаженный перцептрон. И главным тут является гладкость функции, а не её точная форма. Гладкость означает, что небольшие изменения Δw_j весов и Δb смещений дадут небольшие изменения $\Delta output$ выходных данных. Алгебра говорит нам, что $\Delta output$ хорошо аппроксимируется так:

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b \quad (4)$$

Где суммирование идёт по всем весам w_j , а $\partial output / \partial w_j$ и $\partial output / \partial b$ обозначают частные производные выходных данных по w_j и b соответственно. Формула выглядит сложной, со всеми этими частными производными, но на самом деле она говорит нечто совсем простое (и полезное): $\Delta output$ – это линейная функция (4), зависящая от Δw_j и Δb весов и смещения. Её линейность облегчает выбор небольших изменений весов и смещений для достижения любого желаемого небольшого смещения выходных данных. Так что, хотя по качественному поведению СН похожи на перцептроны, они облегчают понимание того, как можно изменить выход, меняя веса и смещения.

Вместо функции сигмоиды, мы можем брать другие функции активации. Главное, что меняется при смене функции – значения частных производных в уравнении. Оказывается, что когда мы потом подсчитываем эти частные производные, использование σ сильно упрощает алгебру, поскольку у экспонент есть очень приятные свойства при дифференцировании.

Как интерпретировать результат работы СН? Очевидно, главным различием между перцептрами и СН будет то, что СН не выдают только 0 или 1. Их выходными данными может быть любое вещественное число от 0 до 1, так что значения типа 0,173 или 0,689 являются допустимыми. Это может быть полезно, к примеру, если нам нужно, чтобы выходное значение обозначало, к примеру, среднюю яркость пикселей изображения, поступившего на вход НС. Но иногда это может быть неудобно. Допустим, мы хотим, чтобы выход сети говорил о том, что «на вход поступило изображение 9» или «входящее изображение не 9». Очевидно, проще было бы, если бы выходные значения были 0 или 1, как у перцептрана. Но на практике мы можем договориться, что любое выходное значение не меньше 0,5 обозначало бы «9» на входе, а любое значение меньше 0,5, обозначало бы, что это «не 9». Будем явно указывать на наличие подобных договорённостей.

1.4 Архитектура нейросетей

В следующем разделе будет представлена нейросеть, способная на неплохую классификацию рукописных цифр. Но для начала немного терминологии. Допустим, у нас есть следующая сеть:

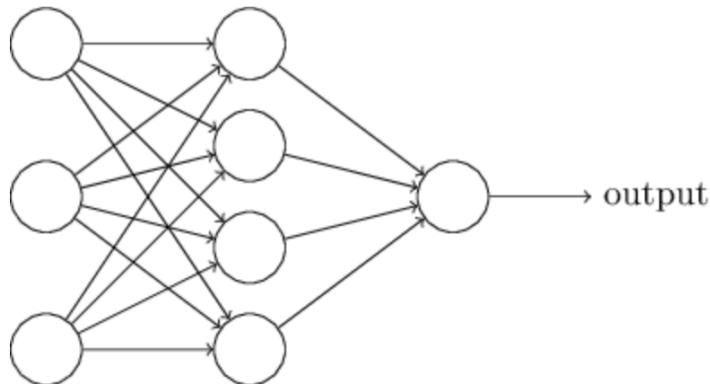


Рис 12. Пример сети с сигмоидными нейронами

Как я уже упоминал, самый левый слой в сети называется входным слоем, а его нейроны – входными нейронами. Самый правый, или выходной слой, содержит выходные нейроны, или, как в нашем случае, один выходной нейрон. Средний слой называется скрытым, поскольку его нейроны не являются ни входными, ни выходными. У сети выше есть только один скрытый слой, но у некоторых сетей есть по несколько скрытых слоёв. К примеру, в следующей четырёхслойной сети есть два скрытых слоя:

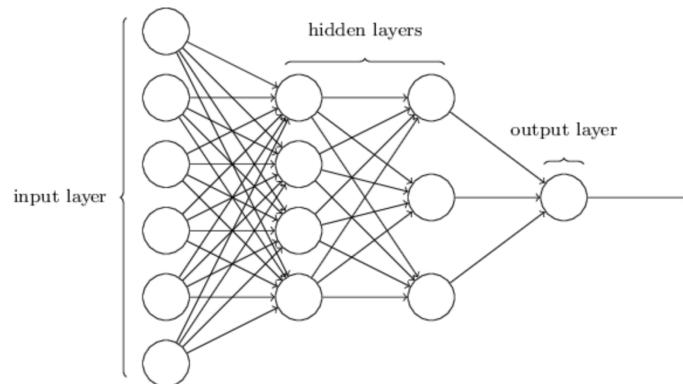


Рис 13. Демонстрация слоев НС

Проектирование входных и выходных слоёв иногда является простой

задачей. К примеру, допустим, мы пытаемся определить, означает ли рукописная цифра «9», или нет. Естественной схемой сети будет кодировать яркость пикселей изображения во входных нейронах. Если изображение будет чёрно-белым, размером 64x64 пикселя, тогда у нас будет $64 \times 64 = 4096$ входных нейронов, с яркостью в промежутке от 0 до 1. Выходной слой будет содержать всего один нейрон, чьё значение менее 0,5 будет означать, что «на входе была не 9», а значения более будут означать, что «на входе была 9».

И если проектирование входных и выходных слоёв часто является простой задачей, то проектирование скрытых слоёв может оказаться сложным искусством. В частности, невозможно описать процесс разработки скрытых слоёв при помощи нескольких простых практических правил. Исследователи НС разработали множество эвристических правил проектирования скрытых слоёв, помогающих получать нужное поведение нейросетей. К примеру, такую эвристику можно использовать, чтобы понять, как достичь компромисса между количеством скрытых слоёв и временем, доступным для обучения сети.

Пока что мы обсуждали НС, в которых выходные данные одного слоя используются в качестве входных для следующего. Такие сети называются нейросетями прямого распространения. Это значит, что в сети нет петель – информация всегда проходит вперёд, и никогда не скармливается назад. Если бы у нас были петли, мы бы встречали ситуации, в которых входные данные сигмоиды зависели бы от выходных. Это было бы тяжело осмыслить, и таких петель мы не допускаем.

Однако существуют и другие модели искусственных НС, в которых возможно использовать петли обратной связи. Эти модели называются рекуррентными нейронными сетями (РНС). Идея этих сетей в том, что их нейроны активируются на ограниченные промежутки времени. Эта активация может стимулировать другие нейроны, которые могут активироваться чуть позже, также на ограниченное время. Это приводит к активации следующих нейронов, и со временем мы получаем каскад активированных нейронов. Петли в таких моделях не представляют проблем, поскольку выход нейрона влияет на его вход в некий более поздний момент, а не сразу.

РНС были не такими влиятельными, как НС прямого распространения, в частности потому, что обучающие алгоритмы для РНС пока что обладают меньшими возможностями. Однако РНС всё равно остаются чрезвычайно интересными. По духу работы они гораздо ближе к мозгу, чем НС прямого распространения. Возможно, что РНС смогут решить важные задачи, которые при помощи НС прямого распространения решаются с большими сложностями.

1.5. Сеть для классификации рукописных цифр

Определив нейросети, вернёмся к распознаванию рукописного текста. Задачу распознавания рукописных цифр мы можем разделить на две подзадачи. Сначала мы хотим найти способ разбиения изображения, содержащего много цифр, на последовательность отдельных изображений, каждое из которых содержит одну цифру. К примеру, мы хотели бы разбить изображение (см. рис 14)



Рис 14. Изображение с рукописными цифрами

на шесть отдельных (см. рис 15)

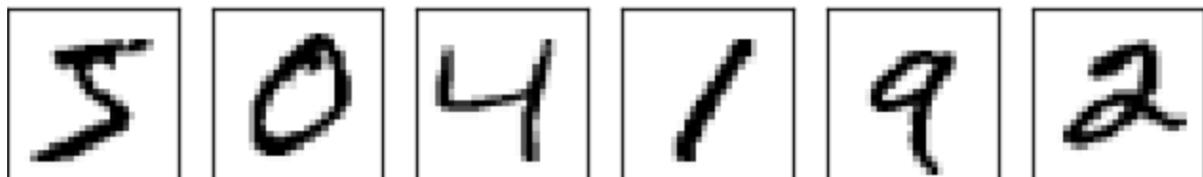


Рис 15. Декомпозированное изображение

Люди, с лёгкостью решают эту задачу сегментации, однако для компьютерной программы сложно правильно разбить изображение. После сегментации программе нужно классифицировать каждую отдельную цифру. Так что, к примеру, мы хотим, чтобы наша программа распознала (см. рис 16), что первая цифра



Рис 16. Отдельная цифра 5

это 5.

Сконцентрируемся на создании программы для решения второй задачи, классификации отдельных цифр. Оказывается, что задачу сегментации решить не так сложно, как только мы найдём хороший способ классификации отдельных цифр. Для решения задачи сегментации есть множество подходов. Один из них – попробовать множество разных способов сегментации изображения с использованием классификатора отдельных цифр, оценивая каждую попытку. Пробная сегментация получает высокую оценку, если классификатор отдельных цифр уверен в классификации всех сегментов, и низкую, если у него есть проблемы в одном или нескольких сегментах. Идея в том, что если у классификатора где-то есть проблемы, это, скорее всего, означает, что сегментация проведена некорректно. Этую идею и другие варианты можно использовать для неплохого решения задачи сегментации. Так что, вместо того, чтобы беспокоиться по поводу сегментации, мы сконцентрируемся на разработке НС, способной решать более интересную и сложную задачу, а именно, распознавать отдельные рукописные цифры.

Для распознавания отдельных цифр мы будем использовать НС из трёх слоёв(см. рис 17):

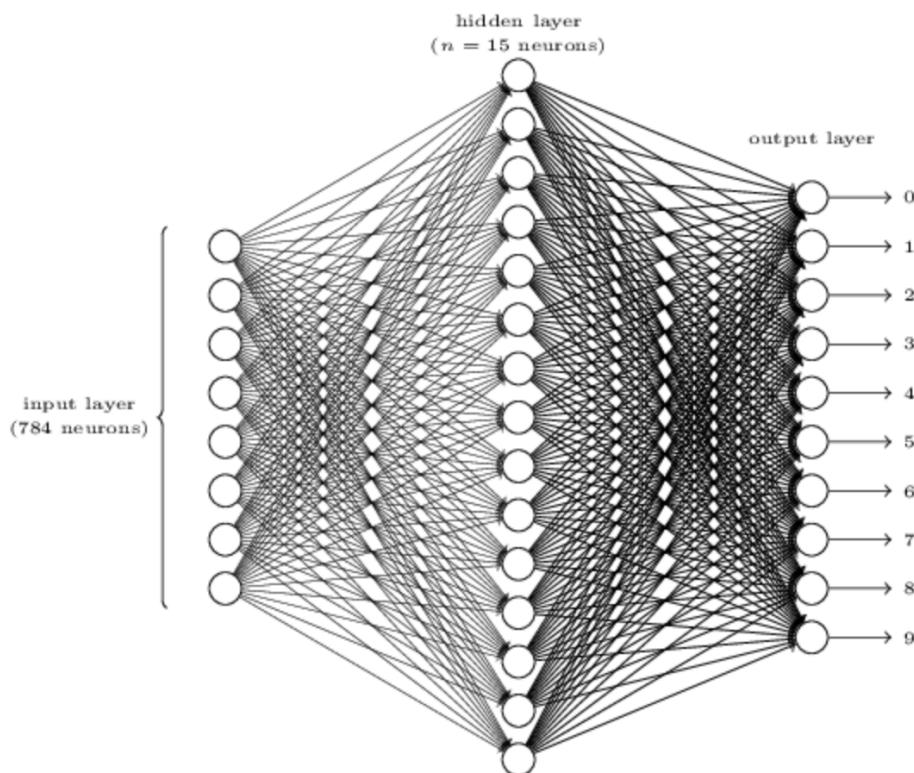


Рис 17. НС из трех слоев

Входной слой сети содержит нейроны, кодирующие различные значения входных пикселей. Как будет указано в следующем разделе, наши обучающие данные будут состоять из множества изображений

отсканированных рукописных цифр размера 28x28 пикселей, поэтому входной слой содержит $28 \times 28 = 784$ нейрона. Для простоты не будем указывать большую часть из 784 нейронов на диаграмме. Входящие пиксели чёрно-белые, при этом значение 0.0 обозначает белый цвет, 1.0 обозначает чёрный, а промежуточные значения обозначают всё более тёмные оттенки серого.

Второй слой сети – скрытый. Обозначим количество нейронов в этом слое n , и будем экспериментировать с различными значениями n . На примере вверху показан небольшой скрытый слой, содержащий всего $n=15$ нейронов.

В выходном слое сети 10 нейронов. Если активируется первый нейрон, то есть, его выходное значение ≈ 1 , это говорит о том, что сеть считает, что на входе был 0. Если активируется второй нейрон, сеть считает, что на входе был 1. И так далее. Строго говоря, мы нумеруем выходные нейроны от 0 до 9, и смотрим, у какого из них значение активации было максимальным. Если это, допустим, нейрон №6, тогда наша сеть считает, что на входе была цифра 6. И так далее.

Можно задуматься над тем, зачем нам использовать десять нейронов. Ведь мы же хотим узнать, какой цифре от 0 до 9 соответствует входное изображение. Естественно было бы использовать всего 4 выходных нейрона, каждый из которых принимал бы двоичное значение в зависимости от того, ближе его выходное значение к 0 или 1. Четырёх нейронов будет достаточно, поскольку $2^4=16$, больше, чем 10 возможных значений. Зачем нашей сети использовать 10 нейронов? Это ведь неэффективно? Основание для этого эмпирическое; мы можем попробовать оба варианта сети, и окажется, что для данной задачи сеть с 10-ю выходными нейронами лучше обучается распознавать цифры, чем сеть с 4-мя. Однако остаётся вопрос, почему же 10 выходных нейронов лучше. Есть ли какая-то эвристика, которая заранее сказала бы нам, что следует использовать 10 выходных нейронов вместо 4?

Чтобы понять, почему, полезно подумать о том, что делает нейросеть. Рассмотрим сначала вариант с 10 выходными нейронами. Сконцентрируемся на первом выходном нейроне, который пытается решить, является ли входящее изображение нулём. Он делает это, взвешивая свидетельства, полученные из скрытого слоя. А что делают скрытые нейроны? Допустим, первый нейрон в скрытом слое определяет, есть ли на картинке что-то вроде такого (см. рис 18):

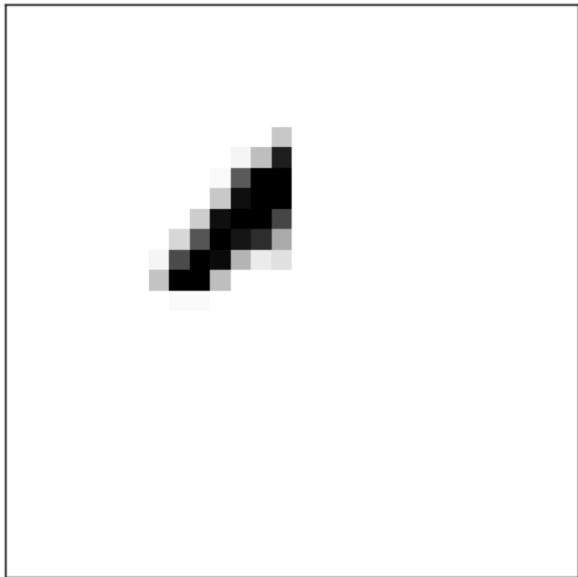


Рис 18. Изображение в скрытом слое

Делать это он может, назначая большие веса пикселям, совпадающим с этим изображением, и малые веса остальным. Точно так же допустим, что второй, третий и четвёртый нейроны в скрытом слое ищут, есть ли на изображении подобные фрагменты (см. рис 19):

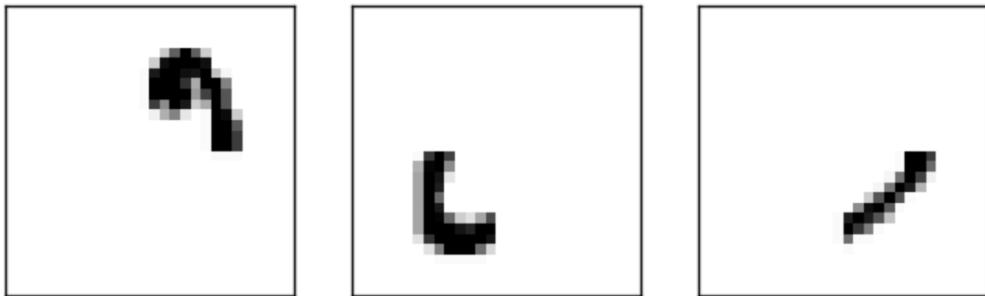


Рис 19. Примеры фрагментов в скрытых слоях

Как мы можем догадаться, все вместе эти четыре фрагмента дают изображение 0 (см. рис 20), которое мы видели ранее:

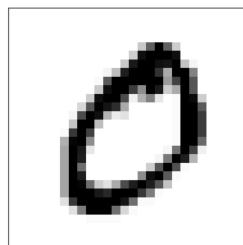


Рис 20. Распознанная цифра 0

Итак, если четыре скрытых нейрона активируются, мы можем заключить, что цифра равна 0. Конечно, это не единственное свидетельство того, что там был изображён 0 – мы можем получить 0 и множеством других способов (немного сдвигая указанные изображения или слегка искажая их). Однако можно точно сказать, что, по крайней мере, в этом случае мы можем заключить, что на входе был 0.

Если предположить, что сеть работает так, можно дать правдоподобное объяснение тому, почему лучше использовать 10 выходных нейронов вместо 4. Если бы у нас было 4 выходных нейрона, тогда первый нейрон пытался бы решить, каков самый старший бит у входящей цифры. И нет простого способа связать самый старший бит с простыми формами, приведёнными выше. Сложно представить какие-то исторические причины, по которым части формы цифры были бы как-то связаны с самым старшим битом выходных данных.

Однако всё вышесказанное подкрепляется только эвристикой. Ничто не говорит в пользу того, что трёхслойная сеть должна работать так, как было описано выше, а скрытые нейроны должны находить простые компоненты форм. Возможно, хитрый алгоритм обучения найдёт какие-нибудь значения весов, которые позволят нам использовать только 4 выходных нейрона. Однако в качестве эвристики данный способ работает неплохо, и может сэкономить значительное время при разработке хорошей архитектуры НС.

2. Алгоритм обучения нейронных сетей

2.1 Градиентный спуск

Функция стоимости, которую мы будем минимизировать:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \quad (5)$$

Если присмотреться к С, видно, что она не отрицательна, поскольку все члены суммы (5) неотрицательные. Кроме того, стоимость $C(w, b)$ становится малой, то есть, $C(w, b) \approx 0$, именно тогда, когда $y(x)$ примерно равна выходному вектору a у всех обучающих входных данных x . Так что наш алгоритм сработал хорошо, если сумел найти веса и смещения такие, что $C(w, b) \approx 0$. И наоборот, сработал плохо, когда $C(w, b)$ большая – это означает, что $y(x)$ не совпадает с выходом для большого количества входных данных. Получается, цель обучающего алгоритма – минимизация стоимости $C(w, b)$ как функции весов и смещений. Иначе говоря, нам нужно найти набор весов и смещений, минимизирующих значение стоимости. Мы будем делать это при помощи алгоритма под названием градиентный спуск.

Если же мы будем использовать гладкую функцию стоимости, нам будет легко понять, как вносить небольшие изменения в веса и смещения, чтобы улучшать стоимость.

Хорошо, допустим, мы пытаемся минимизировать некую функцию $C(v)$. Это может быть любая функция с вещественными значениями от многих переменных $v = v_1, v_2, \dots$. Полезно представлять себе, что у функции С есть только две переменных — v_1 и v_2 :

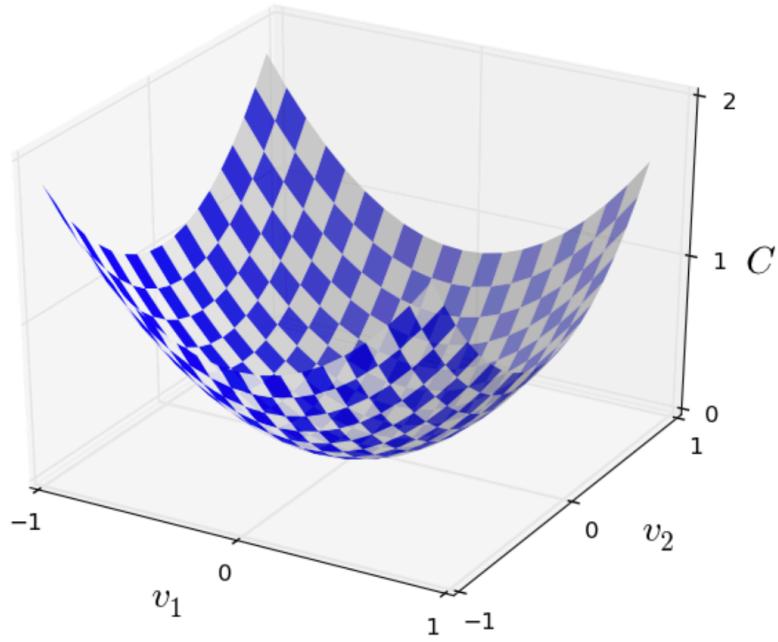


Рис 21. График функции С

Мы бы хотели найти, где С достигает глобального минимума. Конечно, у нарисованной выше функции (см. рис 21) мы можем изучить график и найти минимум.

Мы представляем себе нашу функцию чем-то вроде долины. С последним графиком это будет не так сложно сделать. И мы представляем себе шар, катящийся по склону долины. Наш опыт говорит нам, что шар в итоге скатится на самый низ. Мы случайным образом выберем начальную точку для воображаемого шара, а потом симулируем движение шара, как будто он скатывается на дно долины. Эту симуляцию мы можем использовать просто подсчитывая производные (и, возможно, вторые производные) С – они скажут нам всё о локальной форме долины, и, следовательно, о том, как наш шарик будет катиться.

Чтобы уточнить вопрос, подумаем, что произойдёт, если мы передвинем шар на небольшое расстояние Δv_1 в направлении v_1 , и на небольшое расстояние Δv_2 в направлении v_2 . Алгебра говорит нам, что С меняется следующим образом:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \quad (6)$$

Мы найдем способ выбора таких Δv_1 и Δv_2 , чтобы ΔC была меньше нуля, то есть, мы будем выбирать их так, чтобы шар катился вниз.

Данное уравнение можно переписать в векторном виде:

$$\Delta C \approx \nabla C \cdot \Delta v \quad (7)$$

Где ΔC - градиент вектор:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T \quad (8)$$

Это уравнение позволяет нам увидеть, как выбрать Δv так, чтобы ΔC было отрицательным. Допустим, мы выберем

$$\Delta v = -\eta \nabla C \quad (9)$$

Где η — небольшой положительный параметр (скорость обучения). Тогда уравнение, говорит нам о том, что $\Delta C \approx -\eta * \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$. Поскольку $\|\nabla C\|^2 \geq 0$, это гарантирует, что $\Delta C \leq 0$, то есть C будет всё время уменьшаться, если мы будем менять v , как прописано в верхнем уравнении. Именно это нам и надо! Поэтому мы возьмём уравнение для определения «закона движения» шара в нашем алгоритме градиентного спуска. То есть, мы будем использовать уравнение для вычисления значения Δv , а потом будем двигать шар на это значение:

$$v \rightarrow v' = v - \eta \nabla C \quad (10)$$

Потом мы снова применим это правило, для следующего хода. Продолжая повторение, мы будем понижать C , пока, как мы надеемся, не достигнем глобального минимума.

Подытоживая, градиентный спуск работает через последовательное вычисление градиента ∇C , и последующее смещение в противоположном направлении, что приводит к «падению» по склону долины. Визуализировать это можно так (см. рис 22):

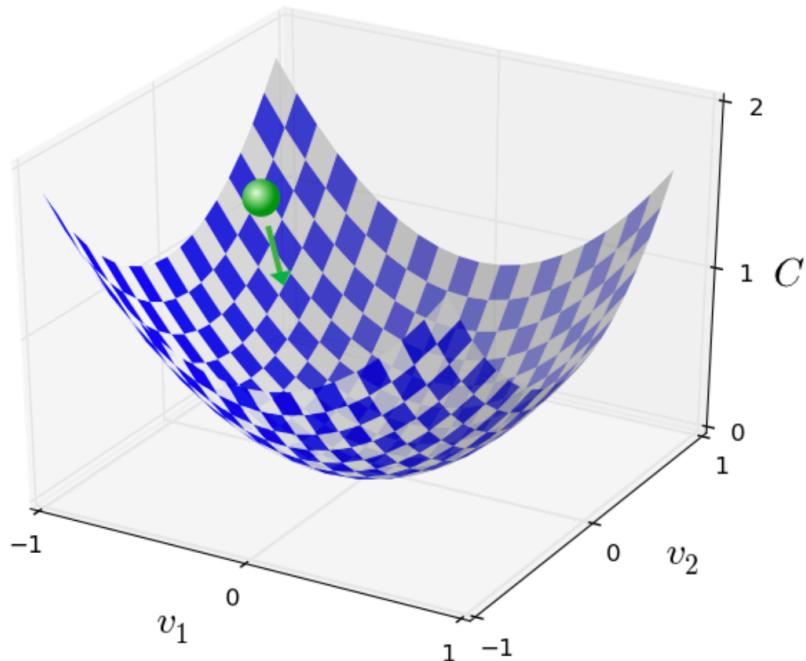


Рис 22. Визуализация градиентного спуска

Чтобы градиентный спуск работал правильно, нам нужно выбрать достаточно малое значение скорости обучения η , чтобы уравнение (10) было хорошей аппроксимацией. В противном случае может получиться, что $\Delta C > 0$.

В то же время, не нужно, чтобы η была слишком маленькой, поскольку тогда изменения Δv будут крохотными, и алгоритм будет работать слишком медленно. На практике η меняется так, чтобы уравнение (10) давало неплохую аппроксимацию, и при этом алгоритм работал не слишком медленно. Градиентный спуск работает так же, если C будет функцией от многих переменных.

Как нам применить градиентный спуск к обучению НС? Нам нужно использовать его для поиска весов w_k и смещений b_i , минимизирующих уравнение стоимости (5). Перезапишем правило обновления градиентного спуска, заменив переменные v_j весами и смещениями. Иначе говоря, теперь у

нашей «позиции» есть компоненты w_k и b_l , а у градиентного вектора ∇C есть соответствующие компоненты $\partial C / \partial w_k$ и $\partial C / \partial b_l$. Записав наше правило обновления с новыми компонентами, мы получим:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \quad (11)$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \quad (12)$$

Повторно применяя это правило обновления (11, 12), мы можем «катиться под горку», и, если повезёт, найти минимум функции стоимости. Иначе говоря, это правило можно использовать для обучения НС.

Применению правила градиентного спуска есть несколько препятствий. Но есть одна проблема. Если вернуться к квадратичной стоимости в уравнении (5). Заметим, что эта функция стоимости выглядит, как $C = 1/n \sum_x C_x$, то есть это среднее по стоимости $C_x \equiv (\|y(x) - a\|^2)/2$ для отдельных обучающих примеров. На практике для вычисления градиента ∇C нам нужно вычислять градиенты ∇C_x отдельно для каждого обучающего входа x , а потом усреднять их, $\nabla C = 1/n \sum_x \nabla C_x$. К сожалению, когда количество входных данных будет очень большим, это займет очень много времени, и такое обучение будет проходить медленно.

Для ускорения обучения можно использовать стохастический градиентный спуск. Идея в том, чтобы приблизительно вычислить градиент ∇C , вычислив ∇C_x для небольшой случайной выборки обучающих входных данных. Посчитав их среднее, мы можем быстро получить хорошую оценку истинного градиента ∇C , и это помогает ускорить градиентный спуск, и, следовательно, обучение.

Формулируя более точно, стохастический градиентный спуск работает через случайную выборку небольшого количества m обучающих входных данных. Мы назовём эти случайные данные X_1, X_2, \dots, X_m , и назовём их минипакетом. Если размер выборки m будет достаточно большим, среднее значение ∇C_{X_j} будет достаточно близким к среднему по всем ∇C_x , то есть

$$\frac{\sum_{j=1}^m \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \quad (13)$$

где вторая сумма идёт по всему набору обучающих данных. Поменяв части местами, мы получим

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{X_j} \quad (14)$$

Чтобы связать это непосредственно с обучением НС, допустим, что w_k и b_l обозначают веса и смещения нашей НС. Тогда стохастический градиентный спуск выбирает случайный мини-пакет входных данных, и обучается на них

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \quad (15)$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l} \quad (16)$$

Где идёт суммирование по всем обучающим примерам X_j в текущем мини-пакете. Затем мы выбираем ещё один случайный мини-пакет и обучаемся на нём. И так далее, пока мы не исчерпаем все обучающие данные, что называется окончанием обучающей эпохи. В этот момент мы начинаем заново новую эпоху обучения.

Стохастический градиентный спуск можно представлять себе, как политическое голосование: гораздо проще взять выборку в виде мини-пакета, чем применить градиентный спуск к полной выборке – точно так же, как опрос на выходе из участка провести проще, чем провести полноценные выборы. К примеру, если наш обучающий набор имеет размер $n = 60\,000$, как MNIST, и мы сделаем выборку мини-пакета размером $m = 10$, то в 6000 раз ускорит оценку градиента! Конечно, оценка не будет идеальной – в ней будет статистическая флуктуация – но ей и не надо быть идеальной: нам лишь надо двигаться в примерно том направлении, которое уменьшает C , а это значит,

что нам не нужно точно вычислять градиент. На практике стохастический градиентный спуск – распространённая и мощная техника обучения НС, и база большинства обучающих технологий, которые мы разработаем в рамках книги.

2.2 Обучение нейросети. Метод обратного распространения ошибки

Итак, у нас есть схема НС – как ей обучиться распознавать цифры? Первое, что нам понадобится – это обучающие данные, т.н. набор обучающих данных. Мы будем использовать набор MNIST, содержащий десятки тысяч отсканированных изображений рукописных цифр (см. рис 23), и их правильную классификацию.

Несколько изображений MNIST:

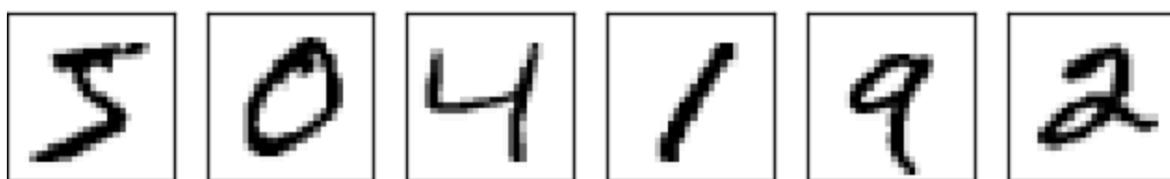


Рис 23. Изображения MNIST

Данные MNIST состоят из двух частей. Первая содержит 60 000 изображений, предназначенных для обучения. Это отсканированные рукописные записи 250 человек, половина из которых были сотрудниками Бюро переписи населения США, а вторая половина – старшеклассниками. Изображения чёрно-белые, размером 28x28 пикселей. Вторая часть набора данных MNIST – 10 000 изображений для проверки сети. Это тоже чёрно-белые изображения 28x28 пикселей. Мы будем использовать эти данные, чтобы оценить, насколько хорошо сеть научилась распознавать цифры. Чтобы улучшить качество оценки, эти цифры были взяты у других 250 людей, не участвовавших в записи обучающего набора (хотя это тоже были сотрудники Бюро и старшеклассники). Это помогает нам убедиться в том, что наша система может распознавать рукописный ввод людей, который она не встречала при обучении.

Обучающие входные данные мы обозначим через x . Будет удобно относиться к каждому входному изображению x как к вектору с $28 \times 28 = 784$ измерениями. Каждая величина внутри вектора обозначает яркость одного пикселя изображения. Выходное значение мы будем обозначать, как $y=y(x)$, где y – десятимерный вектор. К примеру, если определённое обучающее изображение x содержит 6, тогда $y(x)=(0,0,0,0,0,0,1,0,0,0)$ будет нужным нам вектором.

Нам хочется найти алгоритм, позволяющим нам искать такие веса и смещения, чтобы выход сети приближался к $y(x)$ для всех обучающих входных x . Чтобы количественно оценить приближение к этой цели, определим функцию стоимости(потерь):

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2. \quad (17)$$

где: n – общее количество обучающих примеров; сумма идёт по всем примерам x ; $y=y(x)$ – необходимые выходные данные.

Данную функцию (17) также часто называют MSE - Mean Squared Error или среднее значение ошибки.

Функция стоимости - показатель того, насколько хорошо работает наш алгоритм, чем больше функция стоимости, тем больше ошибок мы совершаляем в процессе обучения.

Для того, чтобы вычислить это выражение с использованием градиентного спуска требуется найти все частные производные в сети. Для этого используется алгоритм обратного распространения ошибки (backpropagation).

В основе обратного распространения лежит выражение частной производной $\partial C / \partial w$ функции стоимости C по весу w (или смещению b) сети. Выражение показывает, насколько быстро меняется стоимость при изменении весов и смещений.

Функцию стоимости можно записать как среднее $C = 1/n * \sum_x C_x$ функций стоимости C_x для отдельных обучающих примеров x . Это выполняется в случае квадратичной функции стоимости, где стоимость одного обучающего примера $C_x = 1/2 \|y - a_L\|^2$. Это предположение нужно нам потому, что на самом деле обратное распространение позволяет нам вычислять частные производные $\partial C / \partial w$ и $\partial C / \partial b$, усредняя по обучающим примерам.

Обратное распространение связано с пониманием того, как изменение весов и смещений сети меняет функцию стоимости. По сути, это означает подсчёт частных производных $\partial C / \partial w_{ljk}$ и $\partial C / \partial b_{lj}$. Но для их вычисления сначала мы вычисляем промежуточное значение δ_{lj} , которую мы называем ошибкой в нейроне № j в слое № l . Обратное распространение даст нам процедуру для вычисления ошибки δ_{lj} , а потом свяжет δ_{lj} с $\partial C / \partial w_{ljk}$ и $\partial C / \partial b_{lj}$.

Определим ошибку δ_{lj} нейрона j в слое l , как

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \quad (18)$$

Обратное распространение даст нам способ подсчитать δ_l для любого слоя, а потом соотнести эти ошибки с теми величинами, которые нас реально интересуют, $\partial C / \partial w_{lj}$ и $\partial C / \partial b_{lj}$.

Обратное распространение основано на четырёх фундаментальных уравнениях (33). Совместно они дают нам способ вычислить как ошибку δ_l , так и градиент функции стоимости.

Уравнение ошибки выходного слоя, δ_L :

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (19)$$

Если функция стоимости известна, то не должно быть проблем с вычислением $\partial C / \partial a_{lj}$. К примеру, если мы используем квадратичную функцию стоимости, тогда $C = 1/2 \sum (y_j - a_{lj})^2$, поэтому $\partial C / \partial a_{lj} = (a_{lj} - y_j)$, что легко подсчитать.

Вывод: применяя цепное правило, перепишем частные производные через частные производные по выходным активациям:

$$\delta_j^L = \sum_k \frac{\partial C}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_j^L} \quad (20)$$

Вспомнив, что $a_{lj} = \sigma(z_{lj})$, мы можем переписать второй член справа, как $\sigma'(z_{lj})$, и уравнение превращается в

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (21)$$

Данное уравнение можно переписать в матричной форме:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad (22)$$

В случае с квадратичной стоимостью, у нас будет $\nabla_a C = (aL - y)$, поэтому полной матричной формой будет

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \quad (23)$$

Выражение ошибки δ^l через ошибку в следующем слое, δ^{l+1} :

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \cdot \sigma'(z^l) \quad (24)$$

Разберемся откуда оно получилось. С использованием цепного правила получим:

$$\begin{aligned} \delta_j^l &= \frac{\partial C}{\partial z_j^l} \\ &= \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} \\ &= \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \end{aligned} \quad (25)$$

Чтобы вычислить первый член:

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \quad (26)$$

Продифференцировав, получим

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l). \quad (27)$$

Подставив это в начальное уравнение (26) получим:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l). \quad (28)$$

В частности, рассмотрим член $\sigma'(z_l)$ (22). Это значит, что δ_l , скорее всего, будет малой при приближении нейрона к насыщению. А это, в свою очередь, означает, что любые веса на входе насыщенного нейрона будут обучаться медленно.

Уравнение скорости изменения стоимости по отношению к любому смещению в сети:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (29)$$

В матричном виде:

$$\frac{\partial C}{\partial b} = \delta \quad (30)$$

Уравнение для скорости изменения стоимости по отношению к любому весу в сети:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (31)$$

В матричном виде:

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}} \quad (32)$$

Интересное свойство, которые выходят из данного уравнения: Когда активация a_{in} мала, $a_{\text{in}} \approx 0$, член градиента $\partial C / \partial w$ тоже стремится к нулю. В таком случае мы говорим, что вес обучается медленно, то есть, не сильно меняется во время градиентного спуска. Иначе говоря, одним из следствий последнего уравнения будет то, что весовой выход нейронов с низкой активацией обучается медленно.

Рассмотрим член $\sigma'(zLj)$ в (22). Вспомним из графика сигмоиды, что она становится плоской, когда $\sigma(zLj)$ приближается к 0 или 1. В данных случаях $\sigma'(zLj) \approx 0$. Поэтому вес в последнем слое будет обучаться медленно, если активация выходного нейрона мала (≈ 0) или велика (≈ 1). В таком случае обычно говорят, что выходной нейрон насыщен, и в итоге вес перестал обучаться (или обучается медленно). Те же замечания справедливы и для смещений выходного нейрона.

Еще раз приведем все основные уравнения лежащие в основе алгоритма обратного распространения ошибки

$$\begin{aligned} \delta^L &= \nabla_a C \odot \sigma'(z^L) \\ \delta^l &= ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \\ \frac{\partial C}{\partial b_j^l} &= \delta_j^l \\ \frac{\partial C}{\partial w_{jk}^l} &= a_k^{l-1} \delta_j^l \end{aligned} \quad (33)$$

Тогда с учетом всего вышесказанного получим следующий алгоритм обратного распространения ошибки, который дает нам метод подсчета градиента функции стоимости:

A. Вход x : назначить соответствующую активацию a_1 для входного слоя.

- B. Прямое распространение: вычислить $z = w * a + b$ и $a = \sigma(z)$.
- C. Выходная ошибка δ_L : вычислить вектор $\delta_L = \nabla_a C \odot \sigma'(zL)$.
- D. Обратное распространение ошибки: вычислить $\delta = ((w)^*\delta) \odot \sigma'(z)$.
- E. Выход: градиент функции стоимости задаётся:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \text{ и } \frac{\partial C}{\partial b_j^l} = \delta_j^l. \quad (34)$$

Для проверки градиента или того правильно ли мы высчитываем частные производные можно попробовать использовать приближение:

$$\frac{\partial C}{\partial w_j} \approx \frac{C(w + \epsilon e_j) - C(w)}{\epsilon} \quad (35)$$

Где $\epsilon > 0$ — небольшое положительное число, а e_j — единичный вектор направления j . Иначе говоря, мы можем приблизительно оценить $\partial C / \partial w_j$, вычислив стоимость C для двух немного различных значений w_j , а потом применить уравнение (15). Та же идея позволит нам подсчитать частные производные $\partial C / \partial b$ по смещениям.

Хотя такой подход выглядит многообещающим, при его реализации в коде оказывается, что работает он крайне медленно. Чтобы понять, почему, представьте, что у нас в сети миллион весов. Тогда для каждого веса w_j нам нужно вычислить $C(w + \epsilon e_j)$, чтобы подсчитать $\partial C / \partial w_j$. А это значит, что для вычисления градиента нам нужно вычислить функцию стоимости миллион раз, что потребует миллион прямых проходов по сети (на каждый обучающий пример). А ещё нам надо подсчитать $C(w)$, так что получается миллион и один проход по сети.

Однако для проверки того правильно ли мы высчитываем частные производные через метод обратного распространения, можно использовать этот метод для нескольких обучающих примеров, а потом отключать его, так как он правда является довольно долгим.

Хитрость обратного распространения в том, что она позволяет нам одновременно вычислять все частные производные $\partial C / \partial w_j$, используя только один прямой проход по сети, за которым следует один обратный проход. Грубо говоря, вычислительная стоимость обратного прохода примерно такая же, как у прямого.

Алгоритм обратного распространения вычисляет градиент функции стоимости для одного обучающего примера, $C = Cx$. На практике часто

комбинируют обратное распространение с алгоритмом обучения, например, со стохастическим градиентным спуском, когда мы подсчитываем градиент для многих обучающих примеров. В частности, при заданном мини-пакете m обучающих примеров, следующий алгоритм применяет градиентный спуск на основе этого мини-пакета:

- A. Вход: набор обучающих примеров.
- B. Для каждого обучающего примера x назначить соответствующую входную активацию $a_{x,1}$ и выполнить следующие шаги:
 - Прямое распространение. Вычислить $z = w^*a + b$ и $a = \sigma(z)$.
 - Выходная ошибка δ : вычислить вектор $\delta = \nabla_a C \cdot \sigma'(z)$.
 - Обратное распространение ошибки: для каждого вычислить $\delta = ((w)^*\delta) \cdot \sigma'(z)$.
- C. Градиентный спуск: для каждого $l=L, L-1, \dots, 2$ обновить веса согласно правилу

$$w^l \rightarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T, \quad (36)$$

и смещения согласно правилу

$$b^l \rightarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}. \quad (37)$$

3. Реализация нейронной сети

3.1 Основные методы

Центральное место занимает класс Network, который мы используем для представления НС.

Код инициализации объекта Network (см. рис 24):

```
class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(x, y) for x, y in zip(sizes[:-1], sizes[1:])]
```

Рис 24. Код инициализации сети

Массив sizes содержит количество нейронов в соответствующих слоях.

Смещения(biases) и веса(weights) в объекте Network инициализируются случайным образом с использованием функции np.random.randn из Numpy, которая генерирует распределение Гаусса с математическим ожиданием 0 и среднеквадратичным отклонением 1. Такая случайная инициализация даёт нашему алгоритму стохастического градиентного спуска отправную точку.

Метод прямого распространения (см. рис 25) в класс Network, который принимает на вход a от сети и возвращает соответствующие выходные данные.

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = self.sigmoid(np.dot(w, a) + b)
    return a
```

Рис 25. Код прямого прохода

Для обучения нейросети используется метод SGD (Stochastic Gradient Descent) (см. рис 25):

```

def SGD(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    if test_data:
        n_test = len(test_data)
    n = len(training_data)
    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size] for k in range(0, n, mini_batch_size)
        ]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print(f"Epoch {j} - val: {self.evaluate(test_data)} / {n_test}")
        else:
            print(f"Epoch {j} complete")
    print(f"Epoch {j} - train: {self.evaluate(training_data)} / {n}\n")

```

Рис 26. Код метода стохастического градиентного спуска

`training_data` – список кортежей "(x, y)", обозначающих обучающие входные данные и желаемые выходные. Переменные `epochs` и `mini_batch_size` – это количество эпох для обучения и размер мини-пакетов для использования. `eta` – скорость обучения, η . Если `test_data` задан, тогда сеть будет оцениваться относительно проверочных данных после каждой эпохи, и будет выводиться текущий прогресс. Это полезно для отслеживания прогресса, однако существенно замедляет работу.

Код работает так. В каждую эпоху он начинает с того, что случайно перемешивает обучающие данные, а потом разбивает их на мини-пакеты нужного размера. Это простой способ создания выборки из обучающих данных. Затем для каждого `mini_batch` мы применяем один шаг градиентного спуска. Это делает код `self.update_mini_batch(mini_batch, eta)`, обновляющий веса и смещения сети в соответствии с одной итерацией градиентного спуска, используя только обучающие данные в `mini_batch`. Вот код для метода `update_mini_batch`:

```

def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w - (eta/len(mini_batch)) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch)) * nb for b, nb in zip(self.biases, nabla_b)]

```

Рис 27. Код метода обновления в каждом мини-батче

Большую часть работы делает строчка

```
delta_nabla_b, delta_nabla_w = self.backprop(x, y)
```

Рис 28. Вызов метода обратного распространения

Она вызывает алгоритм обратного распространения – это быстрый способ вычислить градиент функции стоимости. Так что update_mini_batch просто вычисляет эти градиенты для каждого обучающего примера из mini_batch, а потом обновляет self.weights и self.biases.

Метод обратного распространения ошибки (см. рис 28), которые вычисляет частные производные по всем весам и смещениям для отдельного обучающего примера для дальнейшего применения шага градиентного спуска:

```
def backprop(self, x, y):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # feed forward
    activation = x
    activations = [x]

    zs = []
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = self.sigmoid(z)
        activations.append(activation)

    # backpropagation
    delta = self.cost_derivative_mse(activations[-1], y) * self.sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].T)
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = self.sigmoid_prime(z)
        delta = (np.dot(self.weights[-l+1].T, delta)) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].T)
    return nabla_b, nabla_w
```

Рис 28. Код метода обратного распространения ошибки

Метод evaluate (см. рис 29) оценивает долю правильных ответов нашего алгоритма

```
def evaluate(self, test_data):
    test_result = [(np.argmax(self.feedforward(x)), np.argmax(y)) for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_result)
```

Рис 29. Метод для оценки доли правильных ответов

Полный код программы будет доступен на гитхабе:

<https://github.com/kremenevskiy/NeuralNetwork>

3.2 Проверка работы

Сейчас проверим насколько хорошо наша программы распознает цифры.

Создадим сеть из 3 слоев (см. рис 30): первый - входные данные(пиксели изображения), второй слой - 30 нейронов, третий слой - 10 нейронов, так как мы будем обучать нейросеть для классификации на 10 разных классов, соответствующих каждой цифре:

```
net = Net([784, 30, 10])
net.SGD(training_data=training_data, epochs=30, mini_batch_size=10, eta=3, test_data=test_data)
```

Рис 30. Инициализация и запуск нейронной сети

Результат нейросети (см. рис 31):

```
Epoch 1 - val: 9276 / 10000
Epoch 1 - train: 55678 / 60000
```

Рис 31. Вывод результатов обучения первых эпох

Уже после первой эпохи наша нейросеть показывает верно классифицирует 9276 объектов из тестовой выборки.

Если посмотреть результат сети после завершения работы, то есть

после 30 проходов по выборки с использованием стохастического градиентного спуска, получим:

```
Epoch 27 - val: 9504 / 10000
Epoch 27 - train: 58238 / 60000

Epoch 28 - val: 9519 / 10000
Epoch 28 - train: 58273 / 60000

Epoch 29 - val: 9493 / 10000
Epoch 29 - train: 58173 / 60000
```

Рис 32. Вывод результатов обучения последних эпох

То есть нейросеть в процессе обучения подбрала такие параметры, что смогла научиться верно классифицировать 9493 объектов из 10000, то есть точность классификации достигла 94.93% (см. рис 32). И это без настройки ее гиперпараметров, к которым относятся скорость обучения сети, количество скрытых слоев, количества нейронов в скрытых слоях.

Попробуем увеличить количество нейронов в скрытом слое до 100 и обучить ее (см. рис 33), что должно дать на выходе более сложную нейросеть, которая способна генерировать и находить еще более сложные зависимости в данных

```
net = Net([784, 100, 10])
net.SGD(training_data=training_data, epochs=30, mini_batch_size=10, eta=3, test_data=test_data)
```

Рис 33. Запуск обучения со 100 нейронами в скрытом слое

Результат:

```
Epoch 27 - val: 9656 / 10000
Epoch 27 - train: 59175 / 60000

Epoch 28 - val: 9647 / 10000
Epoch 28 - train: 59188 / 60000

Epoch 29 - val: 9661 / 10000
Epoch 29 - train: 59210 / 60000
```

Рис 34. Результат обучения нейросети со 100 нейронами в скрытом слое

Как можно заметить при усложнении структуры нейронной сети
качество классификации увеличилось с 95% до 96.6%

4. Оптимизация нейронной сети

4.1 Реализация полностью матричного градиентного спуска

Ускоренный стохастический градиентный спуск. Наша реализация стохастического градиентного спуска использует цикл по обучающим примерам из мини-пакета. Алгоритм обратного распространения можно поменять так, чтобы он вычислял градиенты для всех обучающих примерах мини-пакета одновременно. Вместо того, чтобы начинать с одного вектора x , мы можем начать с матрицы $X = [x_1 \ x_2 \dots \ x_m]$, чьими столбцами будут векторы мини-пакета. Прямое распространение идёт через произведение весовых матриц, добавление подходящей матрицы для смещений и повсеместного применения сигмоиды. Обратное распространение идёт по той же схеме.

Поэтому имеется возможность реализовать полностью основанный на матрицах подход к обратному распространению на мини-пакете. Преимуществом такого подхода будет использование всех преимуществ современных библиотек для линейной алгебры. В итоге он может работать быстрее цикла по мини-пакеты.

Время обучения без матричного подхода (см. рис 35):

```
Epoch 28 - val: 8720 / 10000
Epoch 28 - train: 53391 / 60000

Epoch 29 - val: 8717 / 10000
Epoch 29 - train: 53381 / 60000

CPU times: user 40min 54s, sys: 36min 23s, total: 1h 17min 17s
Wall time: 19min 43s
```

Рис 35. Время обучения сети без матричного подхода в обратном распространении

Время обучения после реализации матричного подхода (см. рис 36):

```
Epoch 28 - val: 8752 / 10000
Epoch 28 - train: 53391 / 60000

Epoch 29 - val: 8744 / 10000
Epoch 29 - train: 53411 / 60000

CPU times: user 12min 49s, sys: 9min 14s, total: 22min 3s
Wall time: 5min 39s
```

Рис 36. Время обучения сети с матричным подходом в обратном распространении

Как можно заметить время обучения нейронной сети ускорилось в 4 раза.

4.2 Кросс энтропийная функция потерь

В идеале мы ожидаем, что наши нейросети будут обучаться быстро на своих ошибках. Происходит ли это на практике? Для ответа на этот вопрос посмотрим на надуманный пример. В нём участвует нейрон всего с одним входом (см. рис 37):

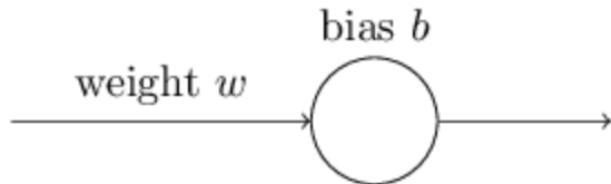


Рис 37. Нейрон с одним входом

Мы обучаем этот нейрон делать нечто простое: принять на вход 1 и выдать 0 с использованием градиентного спуска.

Допустим, что мы вместо этого выберем начальные вес и смещение 2,0. В данном случае изначальный выход будет равен 0,98, что совсем неверно. Давайте посмотрим, как в этом случае нейрон будет учиться выдавать 0.

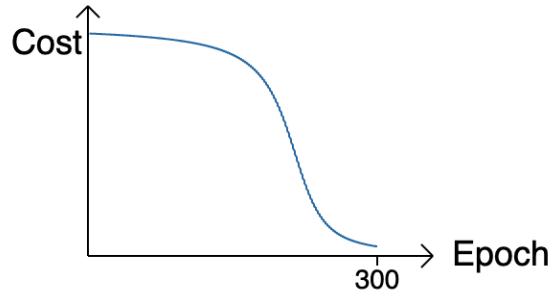


Рис 38. Изменения функции стоимости с MSE стоимостью

Как видно, обучение проходит очень медленно (см. рис 38). Порядка 150 первых эпох веса и смещения почти не меняются. Затем обучение разгоняется, однако несмотря на большие ошибки вначале, учиться наш нейрон достаточно долго.

Это происходит потому что градиент квадратичной функции стоимости очень мал на первых эпохах:

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \quad (38)$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z) \quad (39)$$

Ошибка большая потому что изначальный выход нейрона равен 1 и как следствие член $\sigma'(z)$ почти равен нулю.

и как результат $\partial C / \partial w$ и $\partial C / \partial b$ становятся очень маленькими. Отсюда и растёт замедление обучения.

Эту проблему можно частично решить введя новую функцию стоимости, а именно стоимость с перекрестной кросс энтропией, которая выглядит следующим образом:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \quad (40)$$

Из формулы видно, что вклад стоимости будет небольшим, если реальный выход будет близким к желаемому, то есть при $y=1$ и $a \approx 1$ ошибка

будет $C \approx 0$, то же верно и для квадратичной функции стоимости.

Однако у функции стоимости с перекрёстной энтропией есть преимущество, поскольку, в отличии от квадратичной стоимости, она избегает проблемы замедления обучения. Чтобы увидеть это, подсчитаем частную производную стоимости с перекрёстной энтропией по весам.

$$\begin{aligned} \frac{\partial C}{\partial w_j} &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \frac{\partial \sigma}{\partial w_j} \\ &= -\frac{1}{n} \sum_x \left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)} \right) \sigma'(z) x_j \end{aligned} \quad (41)$$

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x \frac{\sigma'(z) x_j}{\sigma(z)(1-\sigma(z))} (\sigma(z) - y). \quad (42)$$

Используя определение сигмоиды, $\sigma(z)=1/(1+e^{-z})$ и немножко алгебры, можно показать, что $\sigma'(z) = \sigma(z)(1-\sigma(z))$. Члены $\sigma'(z)$ и $\sigma(z)(1-\sigma(z))$ сокращаются, и это приводит к

$$\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_x x_j (\sigma(z) - y). \quad (43)$$

Из него следует, что скорость, с которой обучаются веса, контролируется $\sigma(z)-y$, то есть, ошибкой на выходе. Чем больше ошибка, тем быстрее обучается нейрон.

Этот вариант избегает замедления обучения, вызванного членом $\sigma'(z)$ в аналогичном уравнении для квадратичной стоимости. Когда мы используем перекрёстную энтропию, член $\sigma'(z)$ сокращается, и нам уже не приходится волноваться о его малости. Это сокращение – особое чудо, гарантируемое функцией стоимости с перекрёстной энтропией.

Сходным образом можно вычислить частную производную для смещения.

$$\frac{\partial C}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y). \quad (44)$$

Теперь посмотрим на случай (см. рис 39), в котором наш нейрон застревал с весом и смещением, начинающимися с величины 2,0.

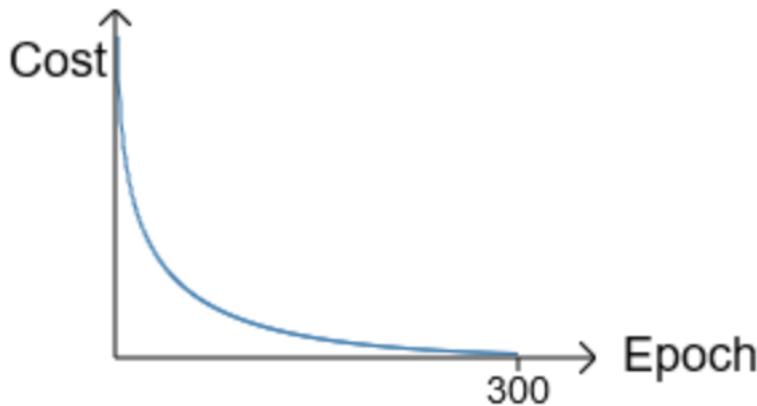


Рис 39. Изменения функции стоимости с кросс-энтропийной стоимостью

На этот раз нейрон обучился быстро, как мы и хотели. Если наблюдать пристально, можно увидеть, что наклон кривой стоимости изначально более крутой, по сравнению с плоским регионом соответствующей кривой квадратичной стоимости. Эту крутость даёт нам перекрестная энтропия, и не даёт застрять там, где мы ожидаем наискорейшее обучение нейрона, когда он начинает с очень больших ошибок.

Перекрестную энтропию легко обобщить на сети со многими слоями и многими нейронами.

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] \quad (45)$$

Проверим, насколько хорошо наша новая программа классифицирует цифры MNIST. Будем использовать сеть с 30 скрытыми нейронами, и минипакетом размером в 10. Установим скорость обучения $\eta=0,5$ и будем обучаться 30 эпох.

```
Epoch 28 - val: 9575 / 10000
Epoch 28 - train: 58695 / 60000

Epoch 29 - val: 9568 / 10000
Epoch 29 - train: 58785 / 60000
```

Рис 40. Результат обучения сети с кросс-энтропийной стоимостью

Получим сеть, работающую с точностью в 95,68%. Это уже лучше чем результат с квадратичной функцией стоимостью, где точность была 94.93%.

Посмотрим на случай, где мы используем 100 скрытых нейронов (см. рис 41) и перекрёстную энтропию, а всё остальное оставляем таким же.

```
net = Net([784, 100, 10], cost=network.CrossEntropyCost)
net.SGD(training_data=training_data, epochs=30, mini_batch_size=10, eta=0.5, test_data=test_data, verbose=True)
```

Рис 41. Запуск сети с 100 нейронами в скрытом слое и кросс-энтропийной стоимостью

Результат:

```
Epoch 27 - val: 9718 / 10000
Epoch 27 - train: 59945 / 60000

Epoch 28 - val: 9717 / 10000
Epoch 28 - train: 59955 / 60000

Epoch 29 - val: 9714 / 10000
Epoch 29 - train: 59961 / 60000
```

Рис 42. Результат обучения сети с 100 нейронами в скрытом слое и кросс-энтропийной стоимостью

В этом случае точность получается 97.14% (см. рис 42). Это серьёзное улучшение по сравнению с результатами из первой главы, где мы достигли точности в 96.61%, используя квадратичную стоимость.

4.3 SoftMax функция активации

Идея Softmax состоит в том, чтобы определить новый тип выходного слоя для НС. Он начинается так же, как сигмоидный слой, с формирования взвешенных входов z . Однако мы не применяем сигмоиду для получения ответа. В Softmax-слое мы применяем Softmax-функцию к z^L_j . Согласно ей, активация a^L_j выходного нейрона № j равна:

$$a^L_j = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}} \quad (46)$$

Преимуществом softmax выходного слоя можно отнести то, что он выдает распределение вероятности, потому что сумма всех выходных активаций будет равной 1

$$\sum_j a^L_j = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1 \quad (47)$$

Во многих задачах удобно иметь возможность интерпретировать выходные активации a_j как оценку сетью вероятности того, что правильным вариантом будет j . Так что, к примеру в задаче классификации MNIST мы можем интерпретировать a_j как оценку сетью вероятности того, что правильным вариантом классификации цифры будет j .

5. Переобучение нейронных сетей

5.1 Понятие переобучения

У нашей НС с 30 скрытыми нейронами для классификации цифр MNIST есть почти 24 000 параметров. Это довольно много параметров. У нашей НС с 100 скрытыми нейронами есть почти 80 000 параметров, а у передовых глубоких НС этих параметров иногда миллионы или даже миллиарды. Встает сразу вопрос можем ли мы доверять результатам их работы?

усложним эту проблему, создав ситуацию, в которой наша сеть плохо обобщает новую для неё ситуацию. Мы будем использовать НС с 30 скрытыми нейронами и 23 860 параметрами. Но мы не будем обучать сеть при помощи всех 50 000 изображений MNIST. Вместо этого используем только первые 1000. Использование ограниченного набора сделает проблему обобщения более очевидной. Однако мы будем обучаться 400 эпох (см. рис 43), что немного больше, чем было раньше, поскольку обучающих примеров у нас не так много.

```
net = Net([784, 30, 10], cost=network.CrossEntropyCost)
data = \
    net.SGD(training_data=training_data[:1000], epochs=400, mini_batch_size=10, eta=0.5,
             validation_data=test_data, monitor_validation_cost=True, monitor_validation_accuracy=True,
             monitor_training_cost=True, monitor_training_accuracy=True)

make_plots(data, 400, training_cost_xmin=0, validation_cost_xmin=0,
           train_accuracy_xmin=0, validation_accuracy_xmin=0)
```

Рис 43. Запуск обучения сети с 1000 обучающих примерами

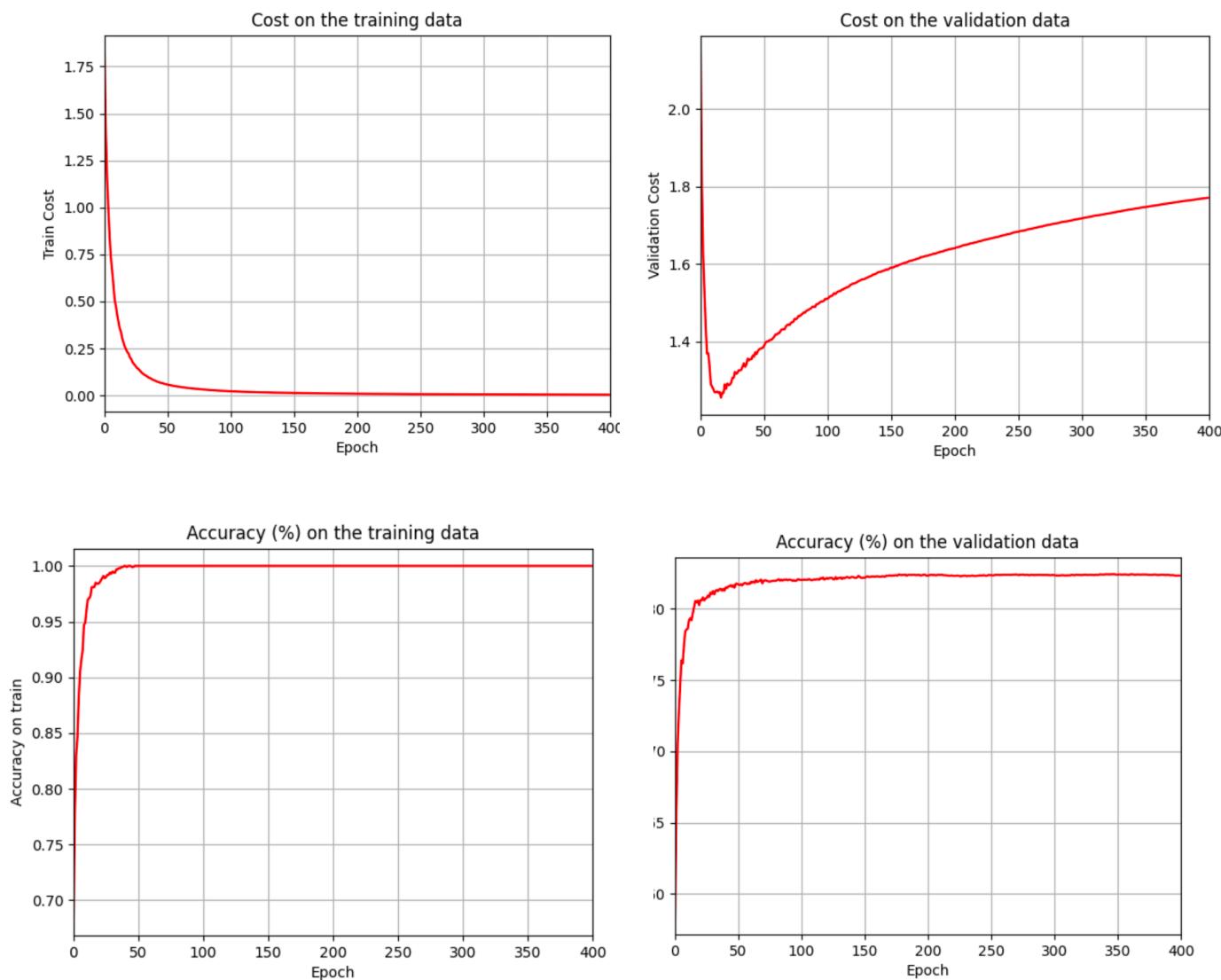


Рис 44. Графики изменения точности и функции стоимости с тестовой и валидационной выборкой в процессе обучения

Посмотрим как меняется точность классификации на тренировочных и валидационных данных, а также значение функции стоимости для для каждой из эпох (см. рис 44)

Как видно, значения функции стоимости на обучении стабильно уменьшаются, так что она доходит почти до нуля.

Точно также как и точность классификации тестовых изображений, которая доходит до 100%.

Однако если посмотреть на точность классификации тестовых картинок, видно что она выходит на плато со значением 84%. Первые 100 эпох точность растет до 82%, затем обучение постепенно замедляется и после 200 эпохи вообще перестает улучшаться. о есть, наша сеть лишь изучает особенности обучающего набора, а не учится распознавать цифры вообще. Похоже на то, что сеть просто запоминает обучающий набор, недостаточно хорошо поняв цифры для того, чтобы обобщить это на проверочный набор.

Интересно то, что если изучать только эту стоимость тренировочных данных, то будет казаться, что модель улучшается. Однако результаты работы

с проверочными данными говорят нам, что это улучшение – лишь иллюзия.

Поэтому это обучение перестаёт быть полезным. Мы говорим, что после 280-й эпохи сеть переобучается, или переподгоняется

Переобучение – серьёзная проблема НС. Особенно это верно для современных НС, в которых обычно есть огромное количество весов и смещений. Для эффективного обучения нам нужен способ определять, когда возникает переобучение, чтобы не переобучать. А ещё нам хотелось бы уметь уменьшать эффекты переобучения.

Пока что мы рассматривали переобучение с использованием 1000 обучающих изображений. Проверим что будет, если мы используем полный обучающий набор из всех 50 000 изображений. Все остальные параметры мы оставим без изменений (30 скрытых нейронов, скорость обучения 0,5, размер мини-пакета 10), но будем обучаться 30 эпох с использованием всех 50 000 картинок.

График, на котором показана точность классификации на обучающих данных и проверочных данных:

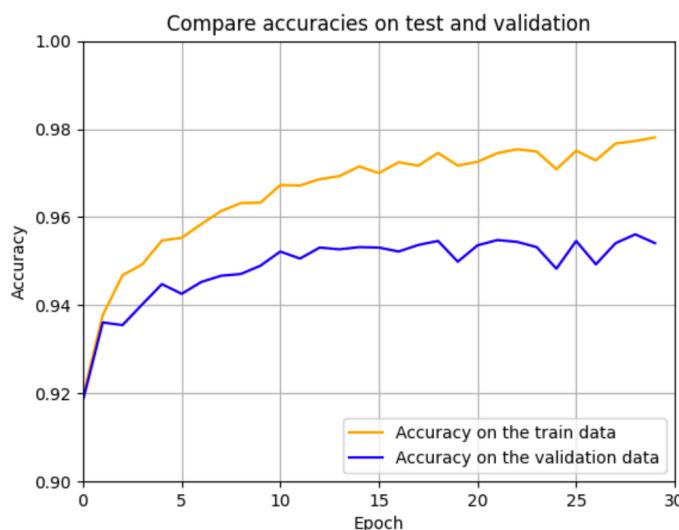


Рис 45. Сравнение точности с тестовым набором и валидационным в процессе обучения

Видно, что показатели точности на проверочных и обучающих данных остаются ближе друг к другу, чем при использовании 1000 обучающих примеров. В частности, наилучшая точность классификации, 97.81%, всего на 2.2% выше, чем 95.61% проверочных данных. Сравним с ранним разрывом в 13%. Переобучение происходит, но сильно уменьшилось. Наша сеть гораздо лучше обобщает информацию, переходя с обучающих на проверочные данные. В целом, один из лучших способов уменьшения переобучения — увеличение объёма обучающих данных. Взяв достаточно обучающих данных, сложно переобучить даже очень крупную сеть. К сожалению, получить обучающие данные бывает дорого или сложно, поэтому такой вариант не

всегда оказывается практичным.

5.2 Регуляризация как способ устранения переобучения

Один из возможных подходов уменьшить переобучение – уменьшение размера сети. Правда, у больших сетей возможностей потенциально больше чем у малых, поэтому к такому варианту прибегают неохотно.

Существуют и другие техники, способные уменьшить переобучение, даже когда у нас фиксированы размер сети и обучающих данных. Они известны, как техники регуляризации.

Например регуляризация L2. Её идея в том, чтобы добавить к функции стоимости дополнительный член под названием член регуляризации. Вот перекрёстная энтропия с регуляризацией:

$$C = -\frac{1}{n} \sum_{xj} [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] + \frac{\lambda}{2n} \sum_w w^2 \quad (48)$$

Возможно регуляризовать и другие функции стоимости, например, квадратичную. Это можно сделать схожим образом:

$$C = \frac{1}{2n} \sum_x \|y - a^L\|^2 + \frac{\lambda}{2n} \sum_w w^2 \quad (49)$$

Мы добавили второй член к функции стоимости (49), а именно, сумму квадратов всех весов сети. Он масштабируется множителем λ , где $\lambda > 0$ – это параметр регуляризации

В общем случае можно записать регуляризованную функцию стоимости, как

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \quad (50)$$

где C_0 — оригинальная функция стоимости без регуляризации.

Смысл регуляризации склонить сеть к предпочтению более малых весов, при прочих равных. Крупные веса будут возможны, только если они значительно улучшают первую часть функции стоимости. Иначе говоря, регуляризация – это способ выбора компромисса между нахождением малых весов и минимизацией изначальной функции стоимости. Важно, что эти два элемента компромисса зависят от значения λ : когда λ мала, мы предпочитаем минимизировать оригинальную функцию стоимости, а когда λ велика, то предпочитаем малые веса.

В частности, нам надо знать, как подсчитывать частные производные, $\partial C / \partial w$ и $\partial C / \partial b$ для всех весов и смещений в сети. После взятия частных производных получим следующие формулы:

$$\begin{aligned}\frac{\partial C}{\partial w} &= \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w \\ \frac{\partial C}{\partial b} &= \frac{\partial C_0}{\partial b}\end{aligned}\tag{51}$$

Частные производные по смещениям не меняются, поэтому правило обучения градиентным спуском для смещений не отличается от обычного:

$$b \rightarrow b - \eta \frac{\partial C_0}{\partial b}\tag{52}$$

Правило обучения для весов превращается в:

$$\begin{aligned}w &\rightarrow w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \\ &= \left(1 - \frac{\eta \lambda}{n}\right) w - \eta \frac{\partial C_0}{\partial w}\end{aligned}\tag{53}$$

Регуляризованное правило обучения для стохастического градиентного спуска превращается в

$$w \rightarrow \left(1 - \frac{\eta\lambda}{n}\right) w - \frac{\eta}{m} \sum_x \frac{\partial C_x}{\partial w} \quad (54)$$

Посмотрим, как регуляризация меняет эффективность нашей НС. Мы будем использовать сеть с 30 скрытыми нейронами, мини-пакет размера 10, скорость обучения 0,5, и функцию стоимости с перекрёстной энтропией. Однако на этот раз мы используем параметр регуляризации $\lambda=0,1$.

Но снова возьмем лишь 1000 обучающих примеров (см. рис 46):

```
net = Net([784, 30, 10], cost=network.CrossEntropyCost)
epochs = 400
data = \
    net.SGD(training_data=training_data[:1000], epochs=epochs, mini_batch_size=10, eta=0.5, lmbda=0.1,
            validation_data=test_data, monitor_validation_cost=True, monitor_validation_accuracy=True,
            monitor_training_cost=True, monitor_training_accuracy=True)

make_plots(data, epochs, training_cost_xmin=0, validation_cost_xmin=0,
           train_accuracy_xmin=0, validation_accuracy_xmin=0, overlay_ylim=0.6)
```

Рис 46. Запуск обучения сети с регуляризацией на неполном наборе данных

Но на этот раз ошибка стабильно уменьшается, а также точность валидационных данных продолжает увеличиваться в течение всех 400 эпох

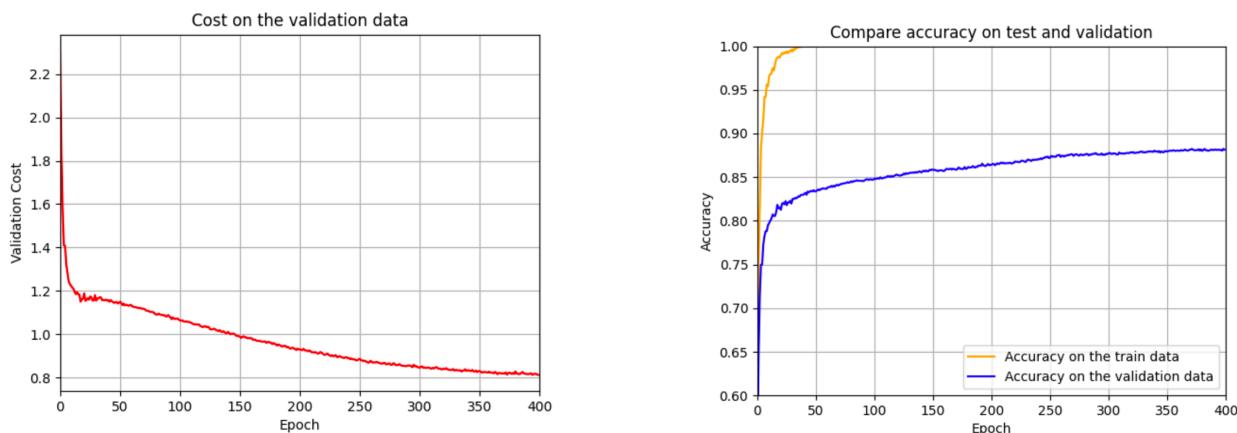


Рис 47. Графики процесса обучения регуляризованной сети

Очевидно, регуляризация подавила переобучение. Более того, точность значительно возросла, и пиковая точность классификации достигает 88,2% (см. рис 47), по сравнению с пиком 84%, достигнутым в случае без регуляризации. Мы почти наверняка достигаем лучших результатов, продолжая обучение после 400 эпох. Судя по всему, эмпирически, регуляризация заставляет нашу сеть лучше обобщать знания, и значительно уменьшает эффекты переобучения.

Вернёмся к полному набору из 50 000 изображений. Мы уже увидели, что переобучение представляет куда как меньшую проблему с полным набором из 50 000 изображений. Оставим прежние значения гиперпараметров – 30 эпох, скорость 0.5, размер мини-пакета 10. Однако, необходимо поменять параметр регуляризации. Дело в том, что размер n обучающего набора скакнул от 1000 до 50 000, а это меняет фактор ослабления весов $1 - \eta * \lambda / n$. Если мы продолжим использовать $\lambda=0,1$, это означало бы, что веса ослабляются куда меньше, и в итоге эффект от регуляризации уменьшается. Мы компенсируем это, приняв $\lambda=5,0$.

```
net = Net([784, 30, 10], cost=network.CrossEntropyCost)
epochs = 30
data = \
    net.SGD(training_data=training_data, epochs=epochs, mini_batch_size=10, eta=0.5, lmbda=5,
            validation_data=test_data, monitor_validation_cost=True, monitor_validation_accuracy=True,
            monitor_training_cost=True, monitor_training_accuracy=True)

make_plots(data, epochs, training_cost_xmin=0, validation_cost_xmin=0,
           train_accuracy_xmin=0, validation_accuracy_xmin=0, overlay_ylim=0.6)
```

Рис 48. Запуск обучения сети с регуляризацией на полном наборе данных

Мы получаем результаты (см. рис 49):

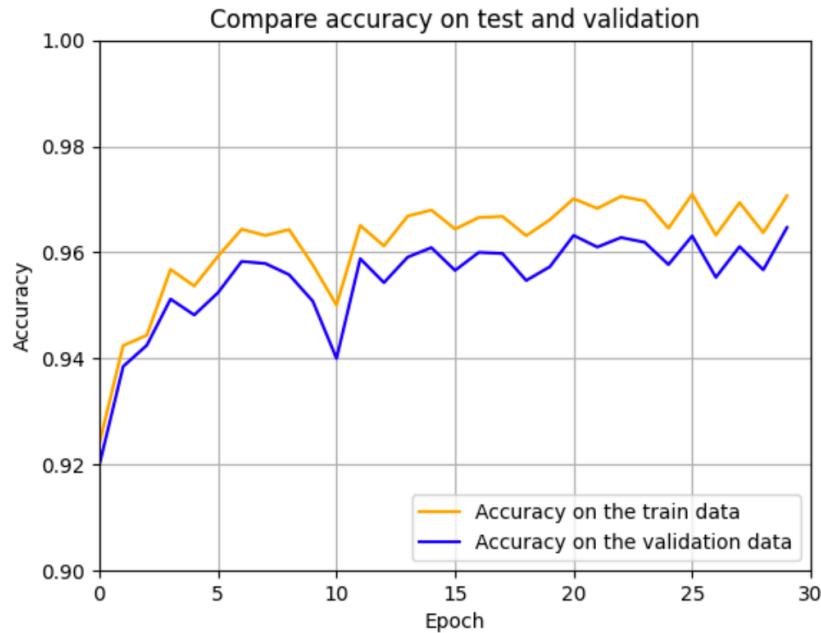


Рис 49. График изменения точности классификации сети с регуляризацией

```

Epoch 26: Cost_train[0.115209]..Acc_train[0.987250]..Cost_val[0.173716]..Acc_val[0.976100]
Epoch 27: Cost_train[0.117431]..Acc_train[0.987333]..Cost_val[0.174582]..Acc_val[0.975700]
Epoch 28: Cost_train[0.105029]..Acc_train[0.988867]..Cost_val[0.153103]..Acc_val[0.979500]
Epoch 29: Cost_train[0.105604]..Acc_train[0.988467]..Cost_val[0.165873]..Acc_val[0.976100]

```

Рис 50. Метрики обучения сети на последних эпохах

Точность классификации на проверочных данных подросла, с 95,61% без регуляризации до 96,47% с регуляризацией. Во-вторых, можно видеть, что разрыв между результатами работы на обучающем и проверочном наборах гораздо ниже, чем раньше, менее 1%. Разрыв всё равно приличный, но мы, очевидно, достигли значительного прогресса в уменьшении переобучения.

Посмотрим, какую точность классификации мы получим при использовании 100 скрытых нейронов и параметра регуляризации $\lambda=5.0$

Получаем точность классификации в 97,9% (см. рис 51) на подтверждающих данных. Большой скачок по сравнению со случаем с 30 скрытыми нейронами, где было 96.47%

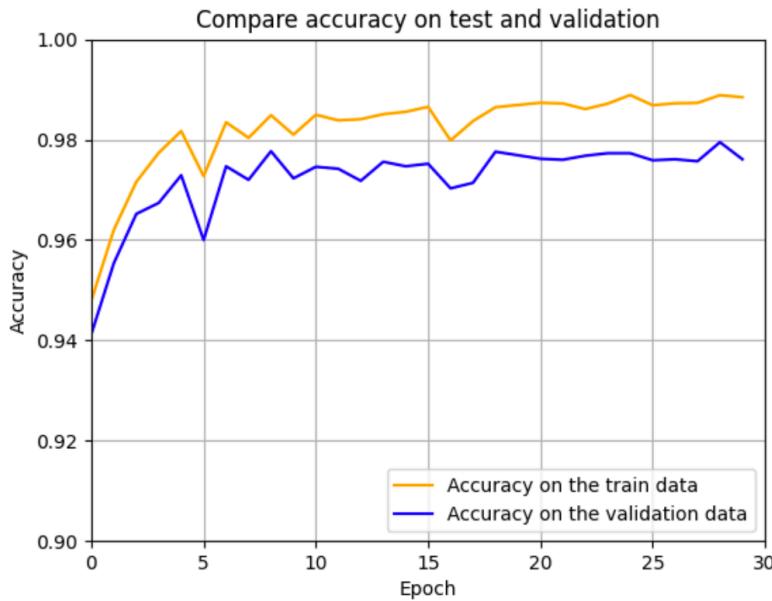


Рис 51. Графики процесса обучения регуляризованной сети со 100 нейронами в скрытом слое

Попробуем запустить процесс снова с 100 нейронами в скрытом слое на 60 эпох с $\eta=0,1$ и $\lambda=5,0$. Это должно увеличить качество классификации, так как мы уменьшили шаг градиентного спуска и сможем зайти еще дальше в минимум функции стоимости не перепрыгивая его.

Проверим догадку (см. рис 52):

```
net = Net([784, 100, 10], cost=network.CrossEntropyCost)
epochs = 60
data = \
    net.SGD(training_data=training_data, epochs=epochs, mini_batch_size=10, eta=0.1, lmbda=5,
            validation_data=test_data, monitor_validation_cost=True, monitor_validation_accuracy=True,
            monitor_training_cost=True, monitor_training_accuracy=True)

make_plots(data, epochs, training_cost_xmin=0, validation_cost_xmin=0,
           train_accuracy_xmin=0, validation_accuracy_xmin=0, overlay_ylim=0.9)
```

Рис 52. Запуск обучения сети с регуляризацией на полном наборе данных с шагом 0,1, 100 нейронами в скрытом слое и обучении на 60 эпохах

Графики процесса обучения регуляризованной сети со 100 нейронами в скрытом слое на 60 эпох

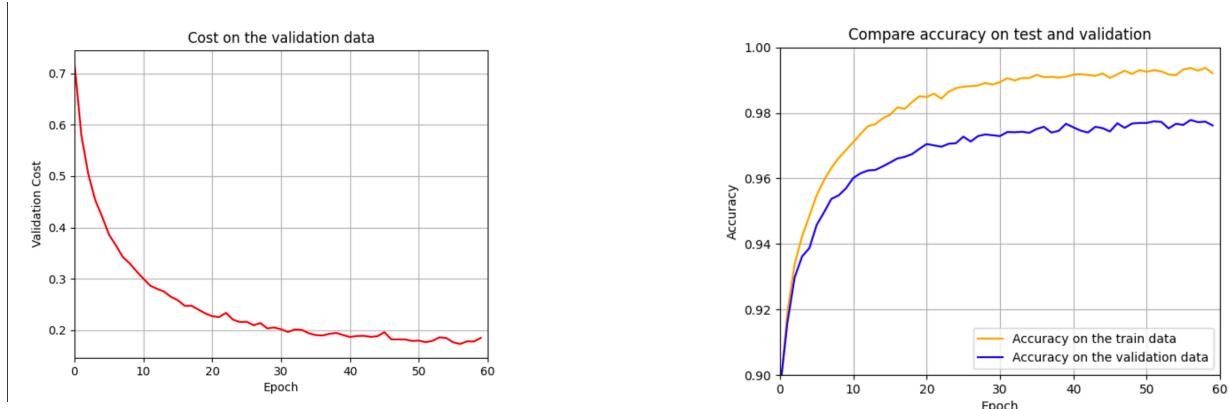


Рис 53. Графики процесса обучения регуляризованной сети со 100 нейронами в скрытом слое

Как можно заметить, ошибка на проверочных данных постепенно уменьшается (см. рис 53), а точность растет и не проседает, что говорит о том, что если мы продолжим обучение модели, скорее всего, она станет еще лучше и еще более точная.

Как можно заметить, пиковая точность достигла 98% на подтверждающих данных

```
Epoch 55: Cost_train[0.087514]..Acc_train[0.992667]..Cost_val[0.150839]..Acc_val[0.980100]
Epoch 56: Cost_train[0.091992]..Acc_train[0.992317]..Cost_val[0.156638]..Acc_val[0.977900]
Epoch 57: Cost_train[0.091112]..Acc_train[0.992433]..Cost_val[0.155631]..Acc_val[0.980300]
Epoch 58: Cost_train[0.087358]..Acc_train[0.992950]..Cost_val[0.153377]..Acc_val[0.978900]
Epoch 59: Cost_train[0.087040]..Acc_train[0.992967]..Cost_val[0.150959]..Acc_val[0.979100]
```

Рис 54. Метрики обучения сети на последних эпохах с достижением точности 98%

Искусственное расширение набора обучающих данных

Ранее мы видели, что наша точность классификации MNIST упала до 80 с чем-то процентов, когда мы использовали всего 1000 обучающих изображений.

попробуем обучить нашу сеть из 30 скрытых нейронов, используя разные объёмы обучающего набора, чтобы посмотреть на изменение эффективности. Мы обучаем, используя размер мини-пакета в 10, скорость обучения $\eta = 0.5$, параметр регуляризации $\lambda=5.0$, и функцию стоимости с перекрёстной энтропией. Мы будем обучать сеть 30 эпох с использованием полного набора данных, и увеличивать количество эпох пропорционально уменьшению объёма обучающих данных. Чтобы гарантировать одинаковый фактор уменьшения весов для разных наборов обучающих данных, мы будем

использовать параметр регуляризации $\lambda=5,0$ с полным обучающим набором, и пропорционально уменьшать его с уменьшением объёмов данных.

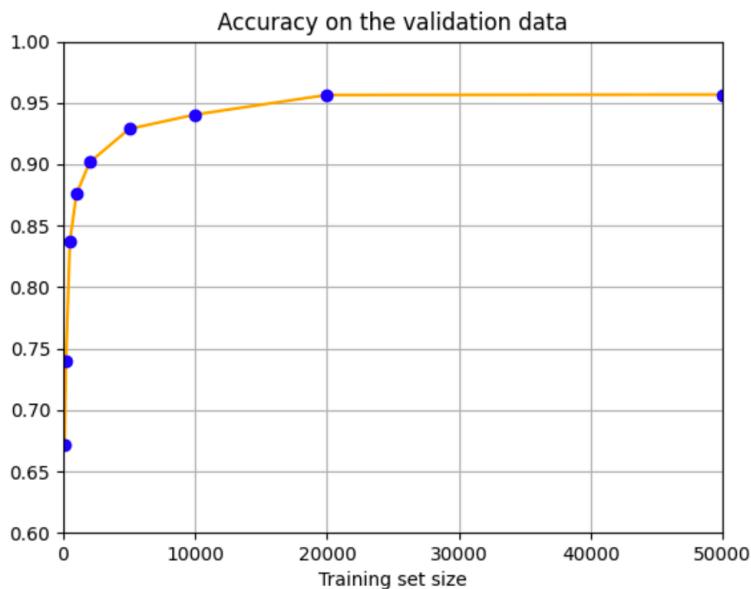


Рис 55. Результат обучения сети на наборах данных с разными размерами

Видно, что точность классификации значительно подрастает с увеличением объёмов обучающих данных (см. рис 55). Вероятно, этот рост будет продолжаться с дальнейшим увеличением объёмов. Конечно, судя по графику выше, мы приближаемся к насыщению. Однако, допустим, что мы переделаем этот график на логарифмическую зависимость от объёма обучающих данных:

Видно, что в конце график всё равно стремится вверх. Это говорит о том, что если мы возьмём гораздо более массивный объём данных – допустим, миллионы или даже миллиарды рукописных примеров, а не 50 000 – тогда мы, вероятно, получим гораздо лучшую сеть даже такого небольшого размера.

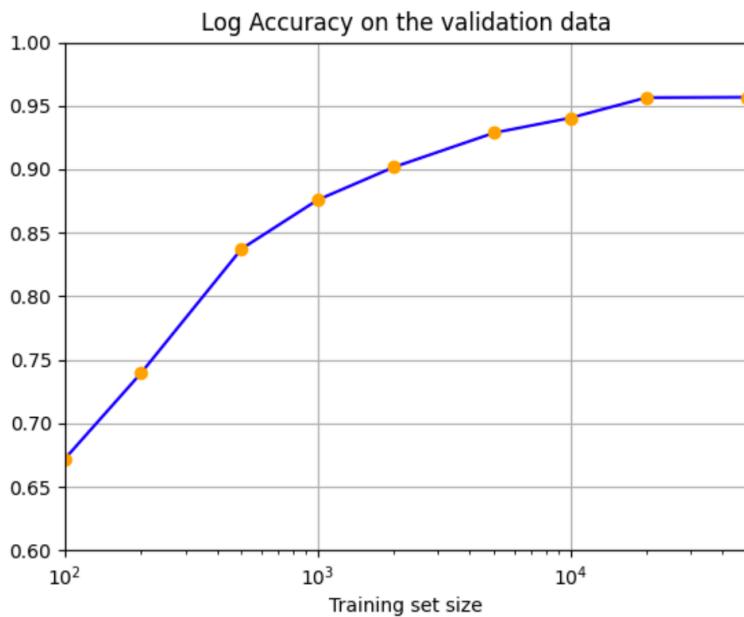


Рис 56. Логарифмический график результата обучения сети на наборах данных с разными размерами

Достать больше обучающих данных – прекрасная идея. К сожалению, это может обойтись дорого, поэтому на практике не всегда возможно. Однако есть и другая идея, способная сработать почти так же хорошо – искусственно увеличить набор данных. К примеру, допустим, добавить к существующему набору данных повернутые на 2-15 градусов случайные картинки, или добавить шума в рядом стоящих пикселях

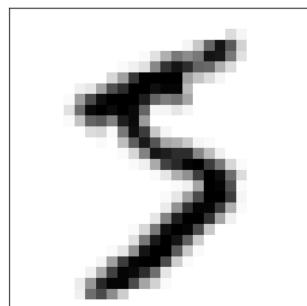


Рис 57. Пример как можно изменять обучающие данные для увеличения их количества

6. Инициализация весов сети

6.1 Проблема рандомной инициализации

Мы выбирали веса и смещения на основе независимого распределения Гаусса с математическим ожиданием 0 и среднеквадратичным отклонением 1. Этот подход хорошо сработал, однако он кажется довольно произвольным

представим, что мы пытаемся обучать сеть входом x , в котором половина входных нейронов включены, то есть, имеют значение 1, а половина – выключены, то есть, имеют значение 0. Следующий аргумент работает и в более общем случае, но вам проще будет понять его на этом особом примере. Рассмотрим взвешенную сумму $z = \sum_j w_j x_j + b$ входов для скрытого нейрона. 500 членов суммы исчезают, поскольку соответствующие x_j равны 0. Поэтому z – это сумма 501 нормализованных гауссовых случайных переменных, 500 весов и 1 дополнительное смещение. Поэтому и само значение z имеет гауссово распределение с математическим ожиданием 0 и среднеквадратичным отклонением $\sqrt{501} \approx 22.4$. То есть, у z довольно широкое гауссово распределение, без острых пиков (см. рис 58):

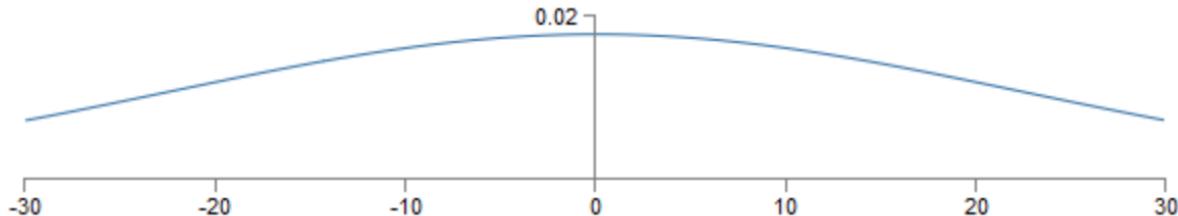


Рис 58. Распределение z при инициализации обычными весами

В частности, из этого графика видно, что $|z|$, скорее всего, будет довольно крупным, то есть, $z \gg 1$ или $z \gg -1$. В таком случае выход скрытых нейронов $\sigma(z)$ будет очень близок к 1 или 0. Это значит, что наш скрытый нейрон насытится. И когда это произойдёт, как нам уже известно, небольшие изменения весов будут давать крохотные изменения в активации скрытого нейрона. Эти крохотные изменения, в свою очередь, практически не затронут остальные нейроны в сети, и мы увидим соответствующие крохотные изменения в функции стоимости. В итоге эти веса будут обучаться очень медленно, когда мы используем алгоритм градиентного спуска.

6.2 Новый подход инициализации весов

Допустим, у нас будет нейрон с количеством входящих весов n_{in} . Тогда нам надо инициализировать эти веса случайными гауссовыми распределениями с математическим ожиданием 0 и среднеквадратичным отклонением $1/\sqrt{n_{in}}$. То есть, мы сжимаем гауссианы, и уменьшаем вероятность насыщения нейрона. Затем мы выберем гауссово распределение для смещений с математическим ожиданием 0 и среднеквадратичным отклонением 1. Сделав такой выбор, мы вновь получим, что $z = \sum_j w_j x_j + b$ будет случайной переменной с гауссовым распределением с математическим ожиданием 0, однако с гораздо более выраженным пиком, чем раньше. Допустим, как и раньше, что 500 входов равны 0, и 500 равны 1. Тогда легко показать, что z имеет гауссово распределение с математическим ожиданием 0 и среднеквадратичным отклонением $\sqrt{(3/2)} = 1,22\dots$ Этот график с гораздо более острым пиком (см. рис 59):

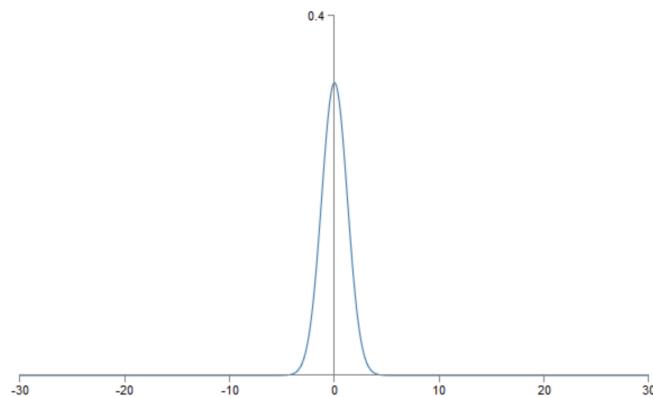


Рис 59. Распределение z при инициализации “сжатыми” весами

Такой нейрон насытится с гораздо меньшей вероятностью, и, соответственно, с меньшей вероятностью столкнётся с замедлением обучения.

Перепишем начальную инициализацию весов в соответствии с новым подходом (см. рис 60):

```
def __init__(self, sizes, cost=CrossEntropyCost):
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases, self.weights = None, None
    self.default_weight_initializer()
    self.cost = cost

def default_weight_initializer(self):
    self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
    self.weights = [np.random.randn(x, y) / np.sqrt(y) for x, y in zip(self.sizes[1:], self.sizes[:-1])]

def large_weight_initializer(self):
    self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
    self.weights = [np.random.randn(x, y) for x, y in zip(self.sizes[1:], self.sizes[:-1])]
```

Рис 60. Код для инициализации весов обычным способом и специальным для предотвращения насыщения первого скрытого слоя

Сравним результаты старого и нового подходов инициализации весов с использованием задачи по классификации цифр из MNIST. Как и ранее, мы будем использовать 30 скрытых нейронов, мини-пакет размером в 10, параметр регуляризации $\lambda=5.0$, и функцию стоимости с перекрёстной энтропией.

Строим график:

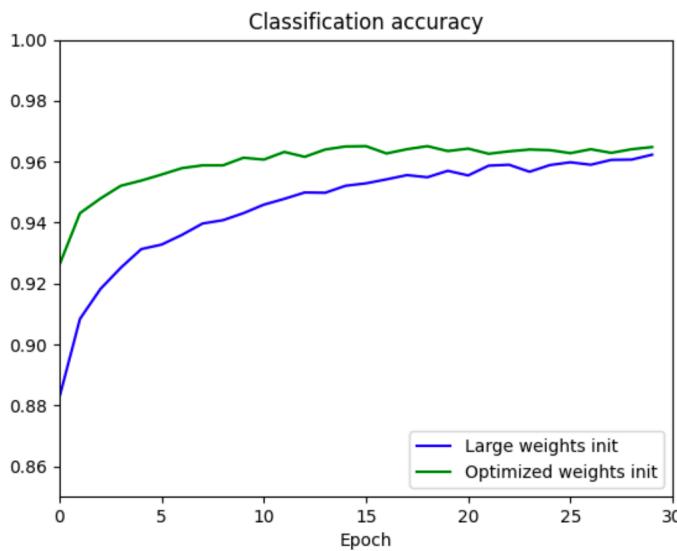


Рис 61. Изменения точности при инициализации весов новым и старым способом

В обоих случаях получается точность классификации в районе 96% (см. рис 61). Итоговая точность почти совпадает в обоих случаях. Но новая техника инициализации доходит до этой точки гораздо, гораздо быстрее. В конце первой эпохи обучения старый подход к инициализации весов

достигает точности в 89%, а новый подход уже подходит к 93%. Судя по всему, новый подход к инициализации весов начинает с гораздо лучшей позиции, благодаря чему мы получаем хорошие результаты гораздо быстрее. То же явление наблюдается, если построить результаты для сети с 100 нейронами:

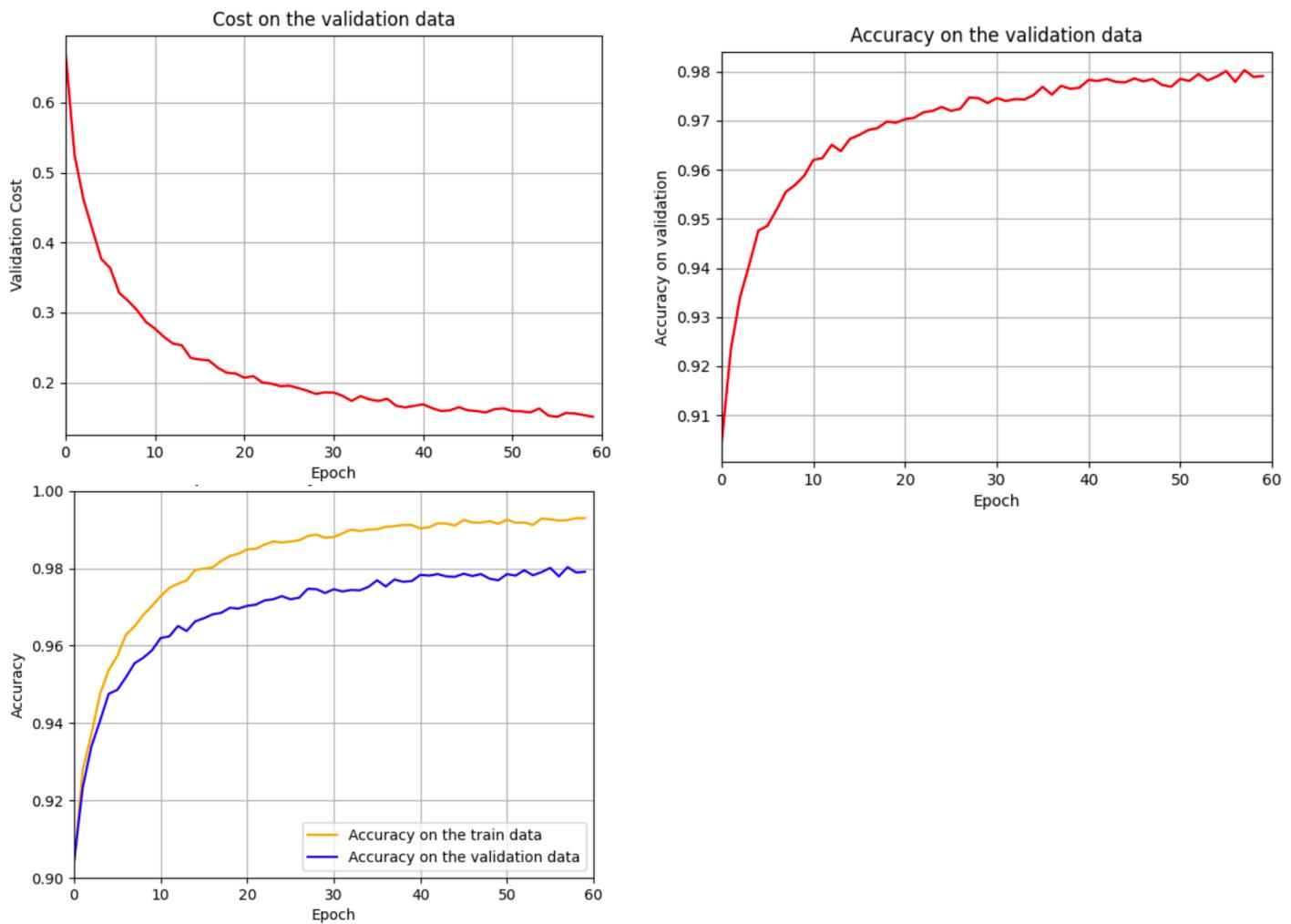


Рис 61. Графики процесса обучения регуляризованной сети со 100 нейронами в скрытом слое с новым способом инициализации весов

7. Демонстрация работы

Создадим сеть:

Make a network

```
1 net = Net([784, 30, 10], cost=network.CrossEntropyCost)
```

Рис 62. Инициализация сети для обучения

Обучим ее и сохраним ее параметры в файл, для дальнейшего использования (см. рис 63)

Train

```
1 %%time
2 epochs = 30
3 data =
4     net.SGD(training_data=training_data, epochs=epochs, mini_batch_size=10, eta=0.5, lmbda=5,
5           validation_data=validation_data, monitor_validation_cost=True, monitor_validation_accuracy=True,
6           monitor_training_cost=True, monitor_training_accuracy=True)
7
8 net.save('neural_init.json')
```

Рис 63. Обучение сети и сохранение результатов

Процесс ее обучения (см. рис 64):

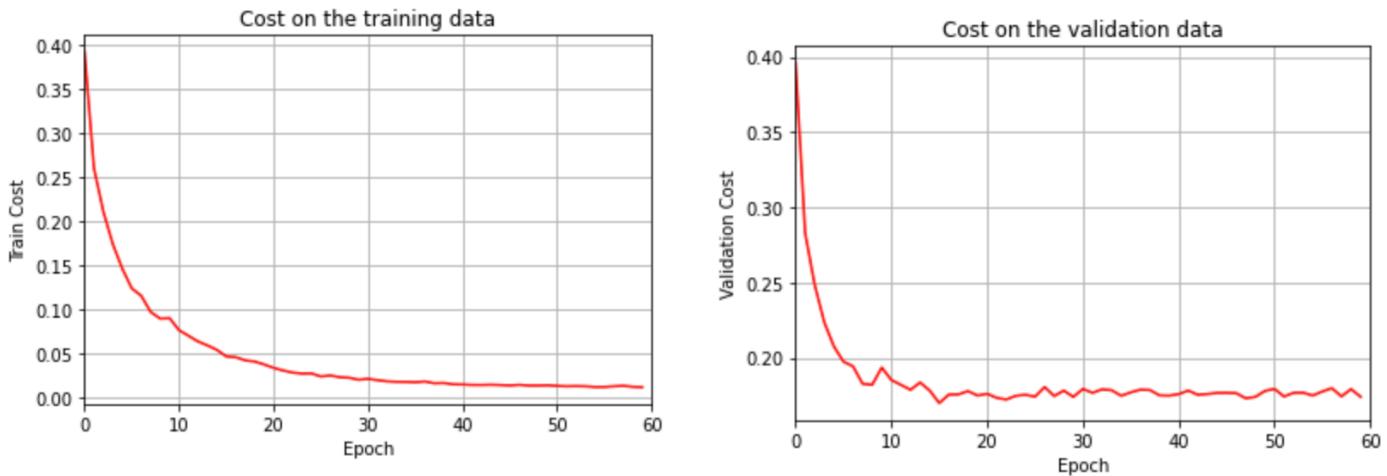


Рис 64. Графики отслеживания процесса обучения нейронной сети по эпохам

Последние 10 метрик (см. рис 65):

```
Metrics [50:60]:  
Epoch 50: Cost_train[0.013533]..Acc_train[0.999958]..Cost_val[0.179313]..Acc_val[0.977833]  
Epoch 51: Cost_train[0.013108]..Acc_train[0.999979]..Cost_val[0.174236]..Acc_val[0.978583]  
Epoch 52: Cost_train[0.013338]..Acc_train[0.999979]..Cost_val[0.176658]..Acc_val[0.977167]  
Epoch 53: Cost_train[0.013097]..Acc_train[0.999958]..Cost_val[0.176755]..Acc_val[0.977667]  
Epoch 54: Cost_train[0.012329]..Acc_train[0.999979]..Cost_val[0.174879]..Acc_val[0.977917]  
Epoch 55: Cost_train[0.012365]..Acc_train[0.999979]..Cost_val[0.177541]..Acc_val[0.977500]  
Epoch 56: Cost_train[0.013089]..Acc_train[0.999979]..Cost_val[0.179759]..Acc_val[0.977250]  
Epoch 57: Cost_train[0.013690]..Acc_train[0.999979]..Cost_val[0.174303]..Acc_val[0.977833]  
Epoch 58: Cost_train[0.012540]..Acc_train[0.999979]..Cost_val[0.179128]..Acc_val[0.977750]  
Epoch 59: Cost_train[0.012123]..Acc_train[0.999979]..Cost_val[0.174033]..Acc_val[0.978000]
```

Рис 65. Метрики обучения сети

Точность на тестовой выборке (см. рис 66):

Test accuracy

```
: 1 print(f'Accuracy on test: {net.accuracy(test_data)}')
```

Accuracy on test: 0.9782

Рис 66. Точность сети на тестовых данных

Посмотрим как справляется наша сеть с некоторыми изображениями из тестовых данных MNIST (см. рис 67)

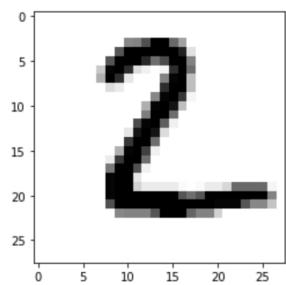
Try to predict some images

```

1 n = 1
2 x, y = test_data[n][0], test_data[n][1]
3
4 res = net.feedforward(x)
5 # print(res)
6 print(f'Predicted number: {np.argmax(res)}')
7 print(f'Real number: {y}')
8 plt.imshow(x.reshape(28,28), cmap=plt.cm.binary)
9 plt.show()

```

Predicted number: 2
Real number: 2

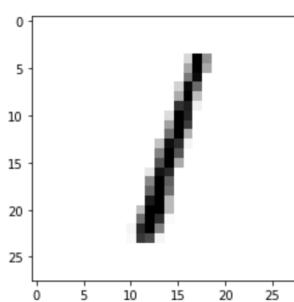


```

1 n = 2
2 x, y = test_data[n][0], test_data[n][1]
3
4 res = net.feedforward(x)
5 # print(res)
6 print(f'Predicted number: {np.argmax(res)}')
7 print(f'Real number: {y}')
8 plt.imshow(x.reshape(28,28), cmap=plt.cm.binary)
9 plt.show()

```

Predicted number: 1
Real number: 1

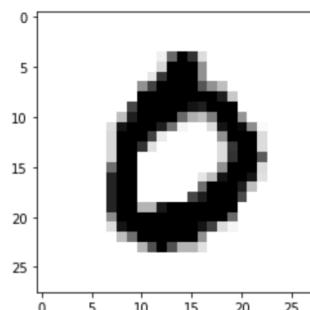


```

1 n = 3
2 x, y = test_data[n][0], test_data[n][1]
3
4 res = net.feedforward(x)
5 # print(res)
6 print(f'Predicted number: {np.argmax(res)}')
7 print(f'Real number: {y}')
8 plt.imshow(x.reshape(28,28), cmap=plt.cm.binary)
9 plt.show()

```

Predicted number: 0
Real number: 0



```

1 n = 4
2 x, y = test_data[n][0], test_data[n][1]
3
4 res = net.feedforward(x)
5 # print(res)
6 print(f'Predicted number: {np.argmax(res)}')
7 print(f'Real number: {y}')
8 plt.imshow(x.reshape(28,28), cmap=plt.cm.binary)
9 plt.show()

```

Predicted number: 4
Real number: 4

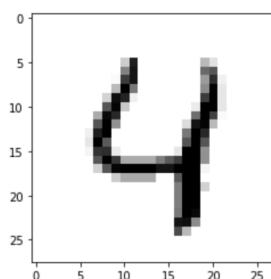


Рис 67. Результат распознавания сети изображений из тестовых данных

Посмотрим как наша сеть справляется с первыми 100 изображениями, красным цветом отмечены те, которые распознаны не верно (см. рис 68)

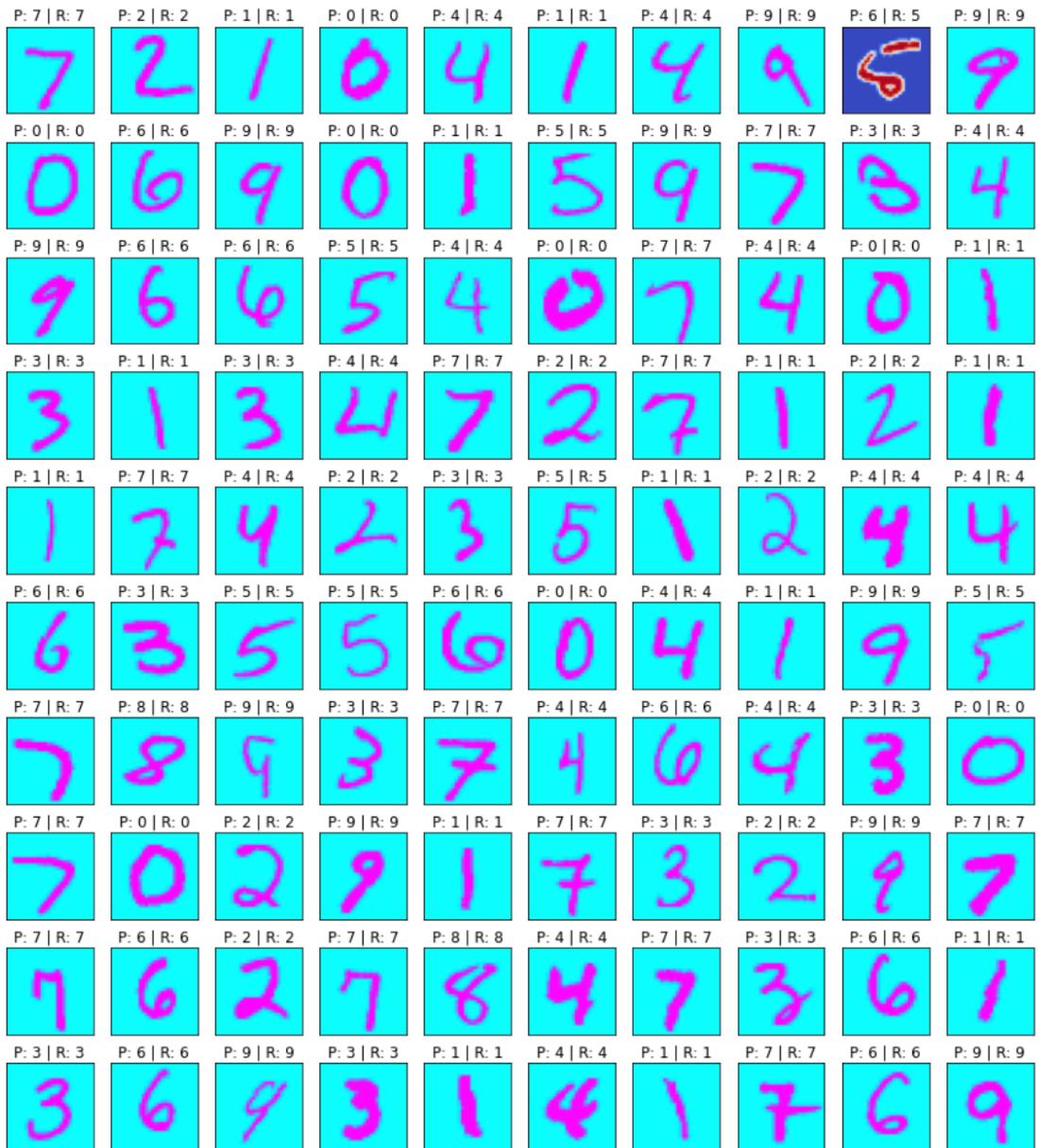


Рис 68. Результат распознавания сети первых 100 изображений

Найдем цифры, которые наша сеть распознает неверно

Numbers hard to predict

```
1 x_test = np.array([_x for _x, _ in test_data])
2 y_test = np.array([_y for _, _y in test_data])
3 pred = np.array([np.argmax(net.feedforward(x)) for x in x_test])
4 mask = pred == y_test
5 x_invalid = x_test[~mask]
6 pred_invalid = pred[~mask]
7 y_invalid = y_test[~mask]
8 print(f'Total test numbers: {len(x_test)}')
9 print(f'Total invalid predictions: {len(x_invalid)}')
```

```
Total test numbers: 10000
Total invalid predictions: 218
```

Рис 69. Код для нахождения изображений, которые классифицируются не верно

На картинках ниже (см. рис 70) значение P - результат, который выдает наша обученная нейронная сеть, R - реальное значение цифры

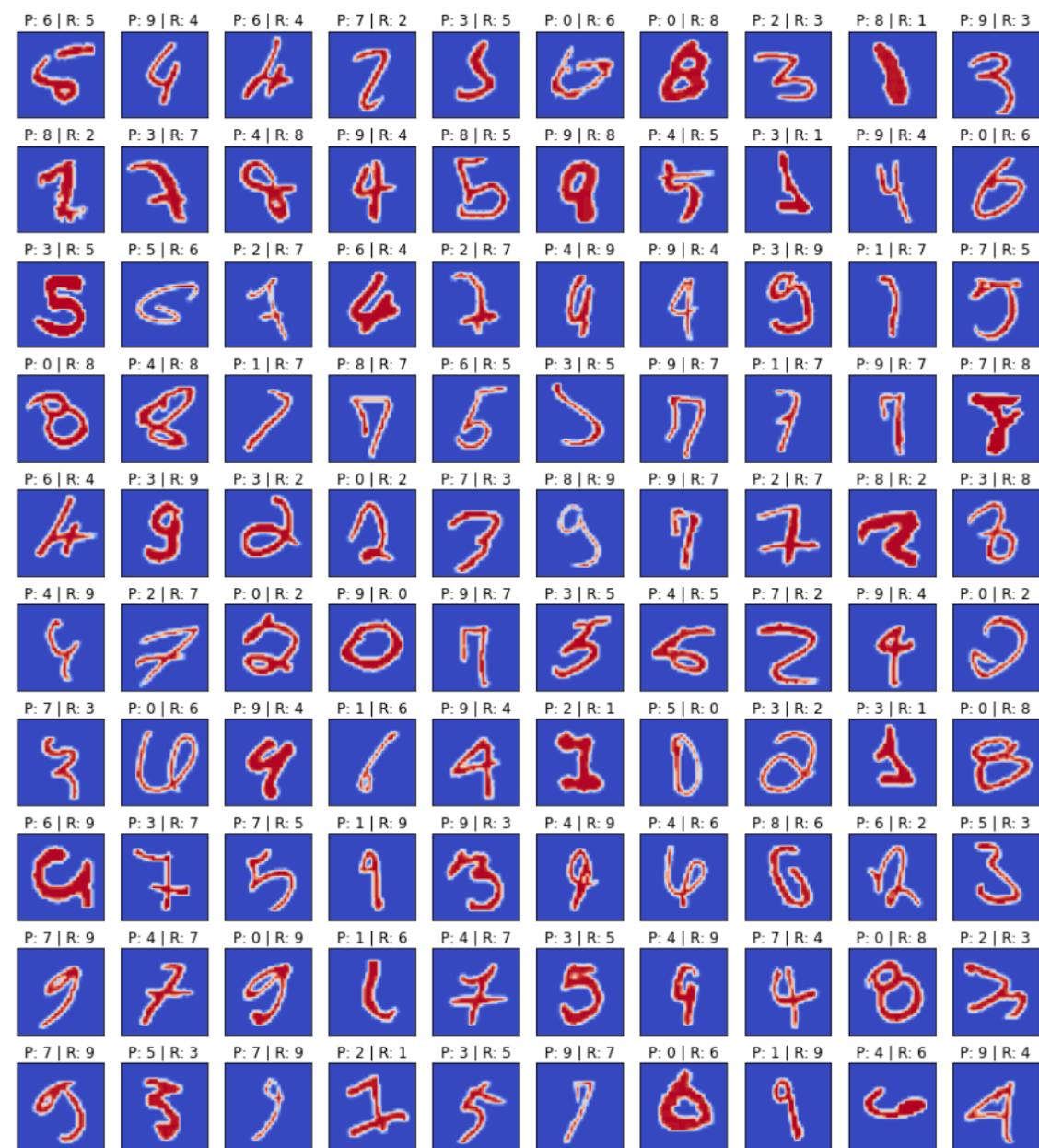


Рис 70. Первые 100 неверно классифицированных изображений

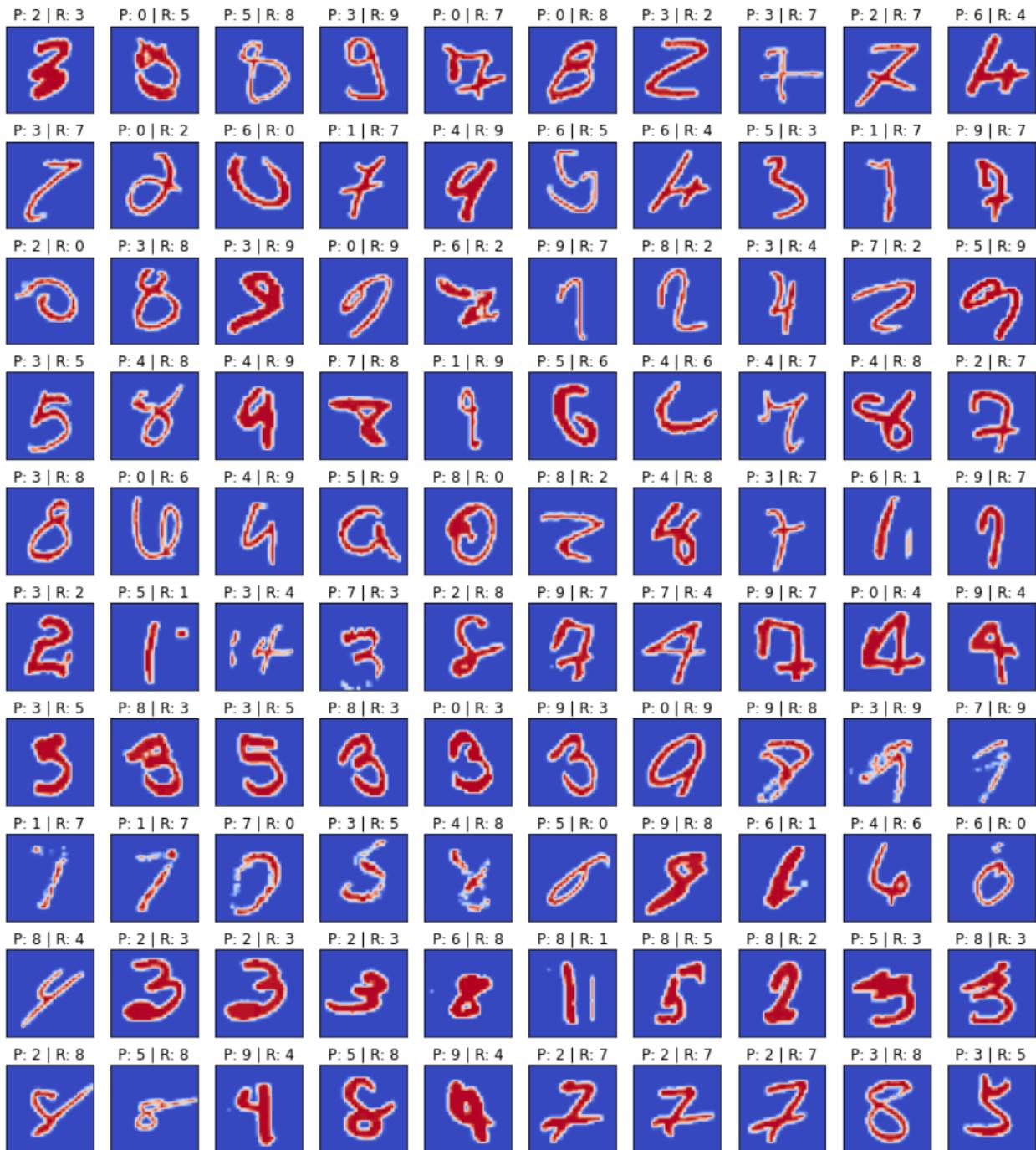


Рис 71. Вторая сотня неверно классифицированных изображений

Как видно некоторые из этих изображений даже человек вряд ли бы смог распознать (см. рис 72)

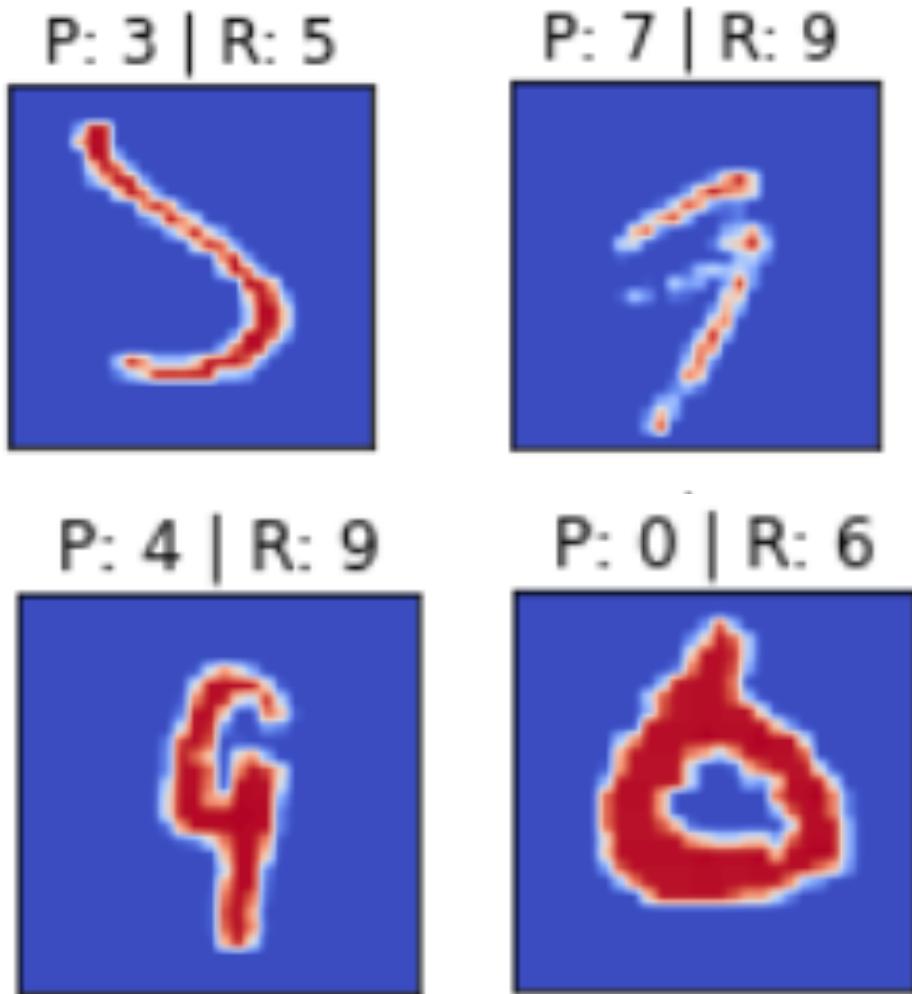


Рис 72. Примеры сложных для распознавания изображений

Попробуем распознать несколько картинок загруженных из интернета (см рис 73-75):

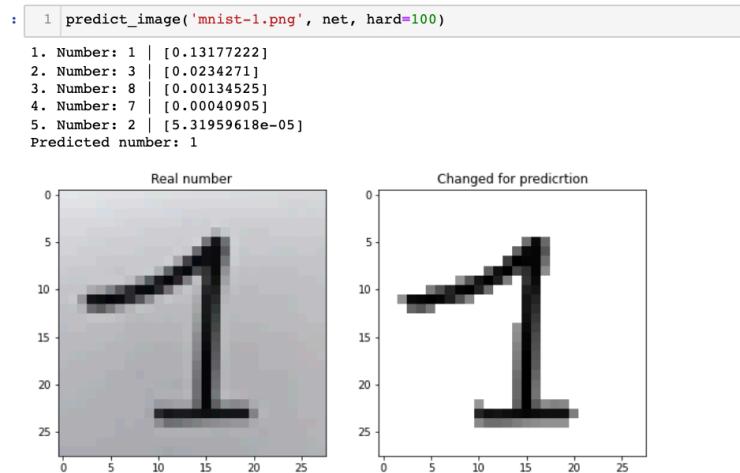


Рис 73. Распознавание изображения с единицей

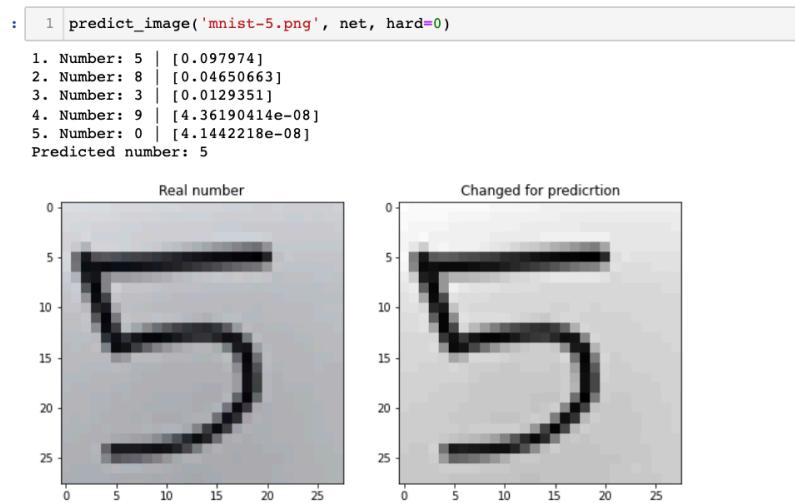


Рис 74. Распознавание изображения с пятеркой

Еще несколько:

```
: 1 predict_image('mnist-0.png', net, hard=0)
1. Number: 0 | [0.92020041]
2. Number: 9 | [0.0004023]
3. Number: 2 | [0.00036712]
4. Number: 4 | [2.090329e-06]
5. Number: 8 | [1.67763505e-06]
Predicted number: 0

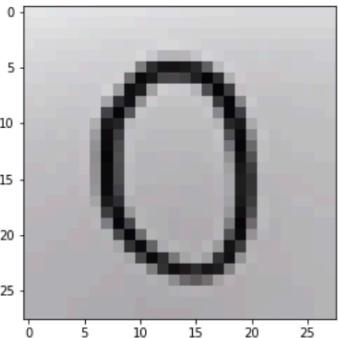
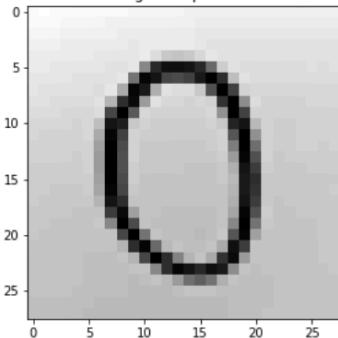
Real number

Changed for prediction

```

Рис 75. Распознавание изображения с нулем

Recognize digits from web

```
: 1 def predict_image(imagepath, net, hard=30, axes=1):
2     plt.figure(figsize=(10, 10))
3     imagepath = 'test_nums/' + imagepath
4
5     image_file = Image.open(imagepath)
6
7     # to show
8     real_file = image_file.copy()
9     real_file = real_file.resize((28, 28))
10    ax_real = plt.subplot(axes, 2, 1)
11    ax_real.imshow(real_file, cmap=plt.cm.binary)
12    ax_real.set_title('Real number')
13
14
15    # to grey scale
16    image_file = image_file.convert('L')
17
18    # revert black -> white, white->black
19    image_file = ImageOps.invert(image_file)
20
21    # resize
22    image_file = image_file.resize((28, 28))
23
24    im_arr = np.array(image_file.getdata()).reshape((784, 1))
25
26    # remove background to 0
27    im_arr[im_arr < hard] = 0
28
29    predicted_seq = net.feedforward(im_arr)
30
31    pred_num = [(index, i) for index, i in enumerate(predicted_seq)]
32    pred_best = sorted(pred_num, key=lambda x: x[1], reverse=True)
33
34    top_n = 5
35    for i in range(top_n):
36        print(f'{i+1}. Number: {pred_best[i][0]} | {pred_best[i][1]}')
37
38    predicted = pred_best[0][0]
39    print(f'Predicted number: {predicted}')
40    ax_changed = plt.subplot(axes, 2, 2)
41    ax_changed.imshow(im_arr.reshape((28, 28)), cmap=plt.cm.binary)
42    ax_changed.set_title('Changed for prediction')
```

Рис 76. Код который делает предобработку изображения и последующее ее распознавание

Как видно сеть неплохо справляется.

Попробуем нарисовать несколько цифр от руки, обработать их и подать на вход нашей сети. Слева изображена написанная цифра, справа обработанная версия для ее подачи в нейронную сеть

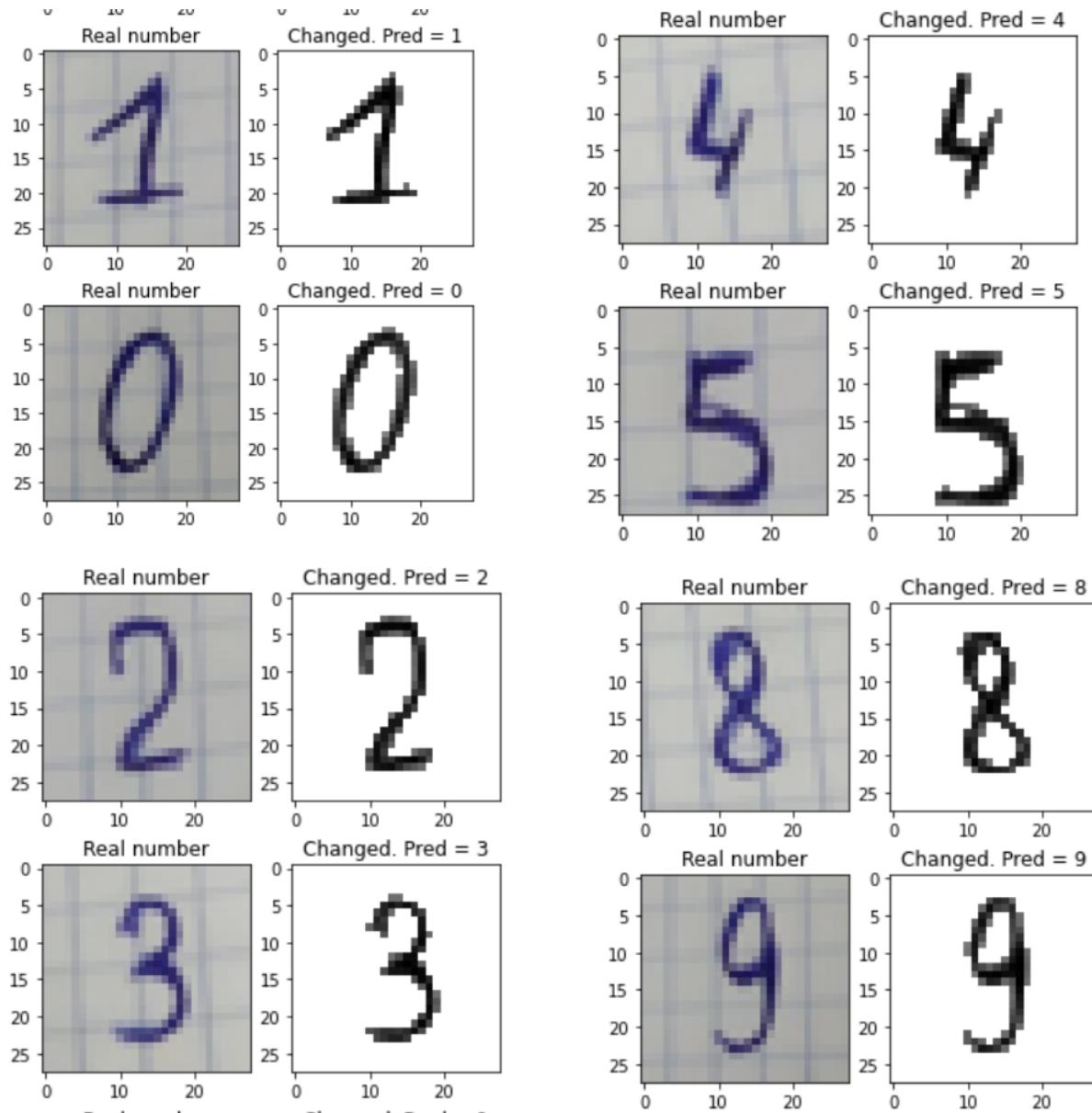


Рис 77. Результат распознавание цифр написанных от руки

Как видно наша нейронная сеть верно распознала все цифры кроме шестерки, для которой выдала результат 5, ведь цифры семантические на самом деле очень похожи по форме.

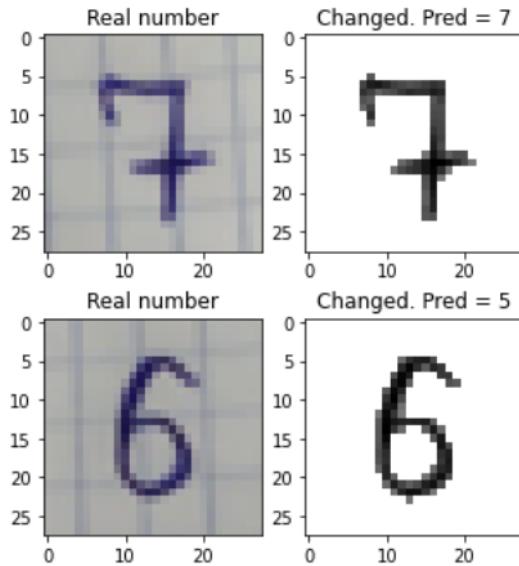


Рис 77. Результат распознавание цифр написанных от руки

Так же можно попробовать распознавать нарисованные цифры онлайн (см. рис 78-84). Справа показываются наиболее вероятные варианты того, что нейронная сеть видит на изображении. Первое значение - ее ответ, остальные два - значения, следующие по вероятности.

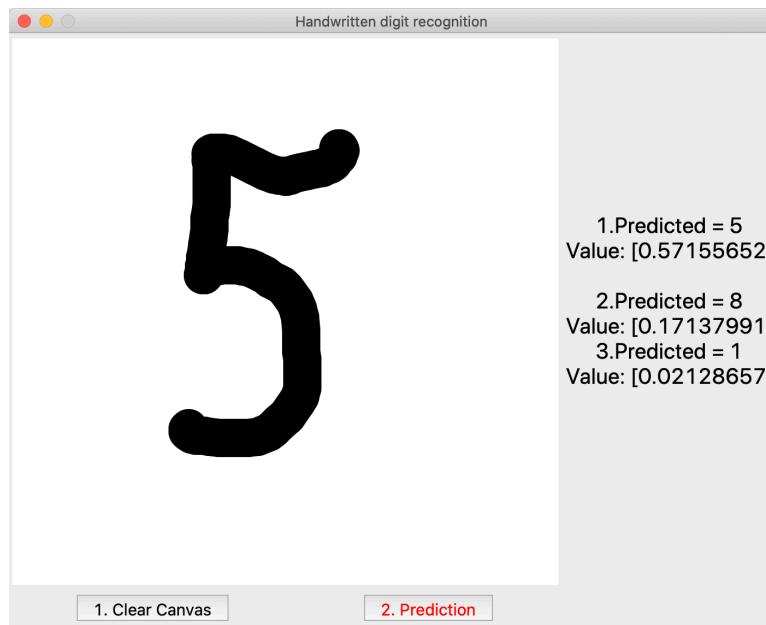


Рис 78. Результат распознавание цифры 5

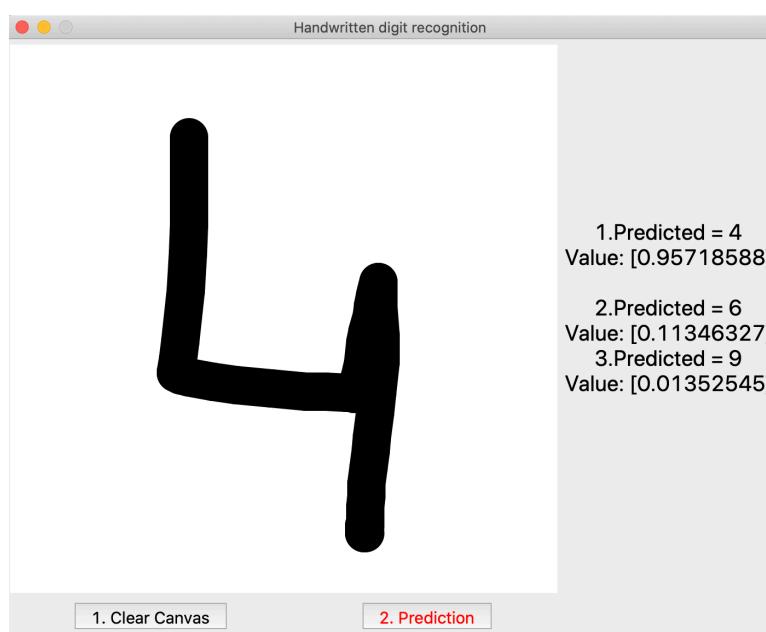


Рис 79. Результат распознавание цифры 4

Еще 2:

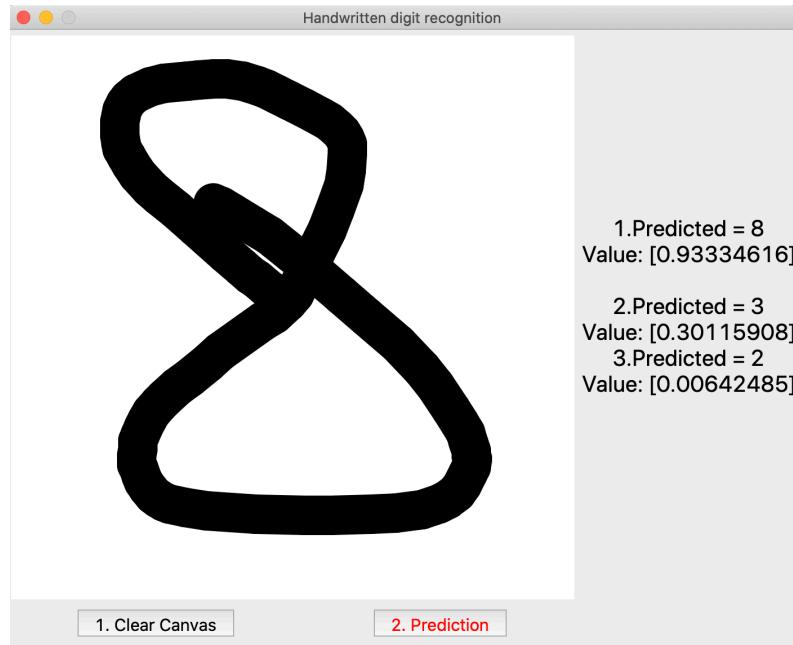


Рис 80. Результат распознавание цифры 8

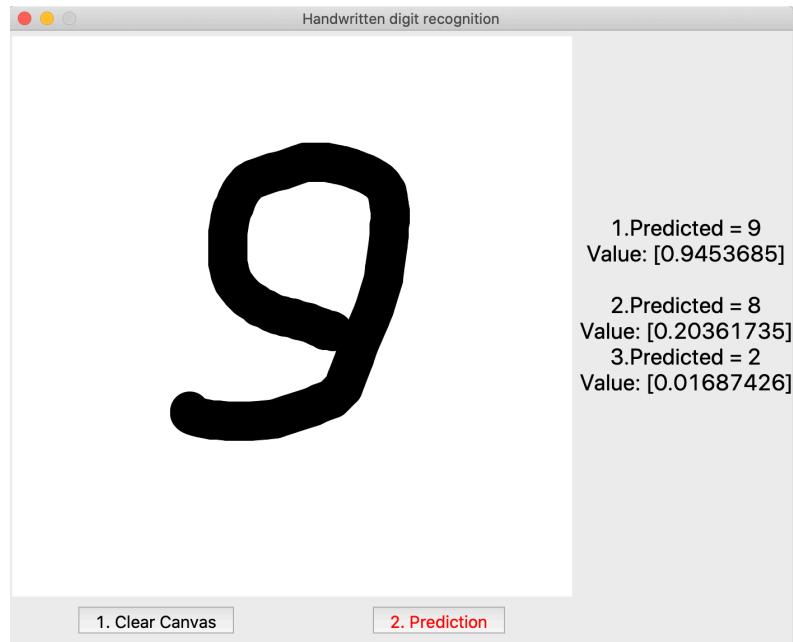


Рис 81. Результат распознавание цифры 9

Еще 2:

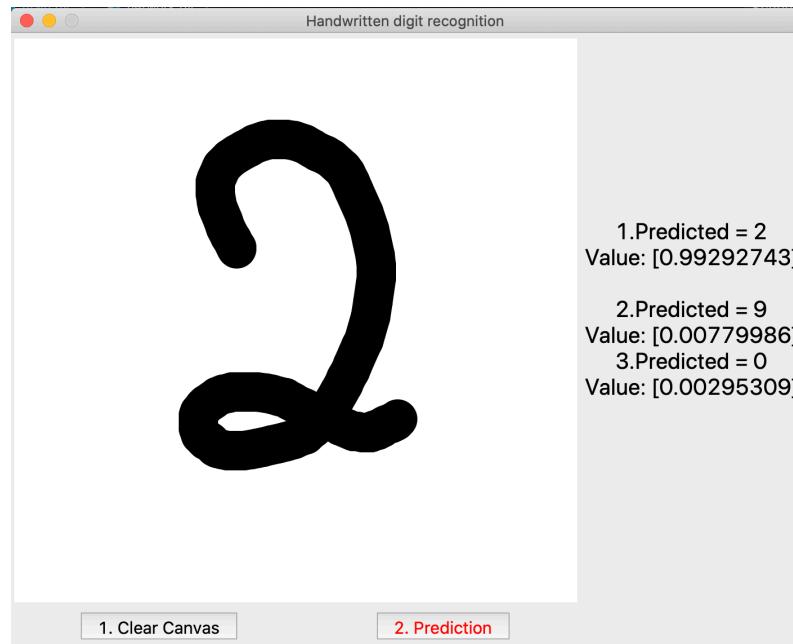


Рис 82. Результат распознавание цифры 2

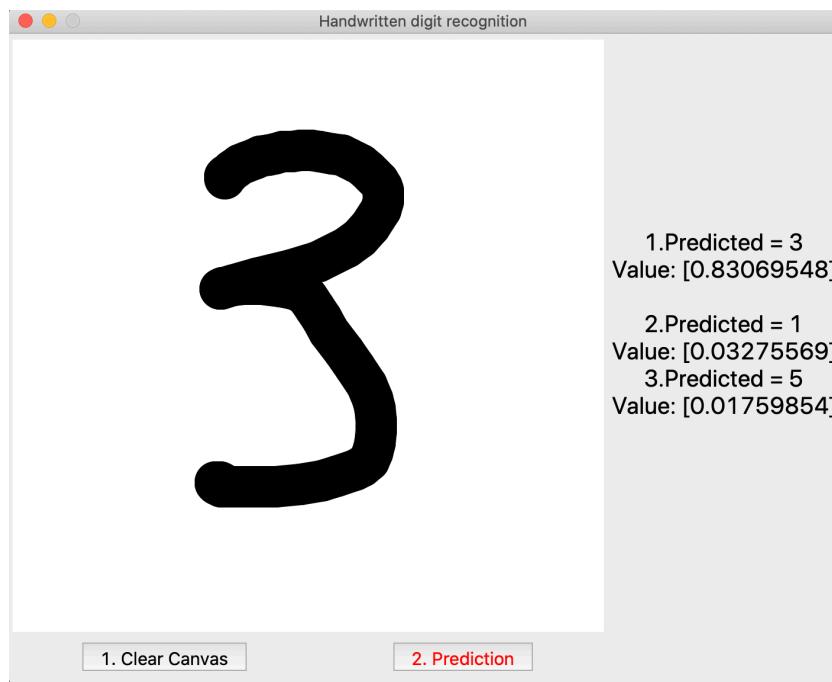


Рис 83. Результат распознавание цифры 3

И наконец единицу:

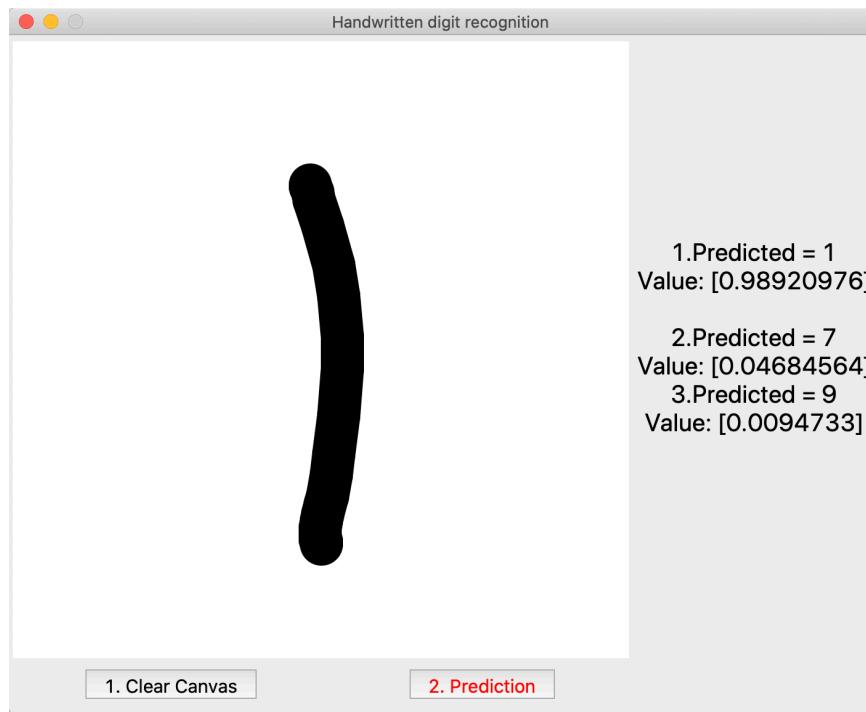


Рис 84. Результат распознавание цифры 1

Заключение

В результате курсового проекта было разработана нейронная сеть с нуля для распознавания рукописных цифр, а также многократно улучшена для получения лучших результатов.

Сеть успешно протестирована на данных MNIST, где показала точность 98% на тестовых данных.

Сеть успешно протестирована на случайных картинках из интернета, которые без проблем распознала.

Сеть успешно протестирована на цифрах написанных от руки человека (своей), снятых на камеру и преобразованных для распознавания.

Также успешна протестирована на написанной программе, которая позволяет рисовать в канвасе цифру и отправлять картинку на распознавание нашей нейронной сети.

Данная сеть неплохо подходит для распознавания простых рукописных цифрах, однако если двинуться дальше в этой области и перейти к распознаванию изображение следует начать с рассмотрения сверточных нейронных сетей.

Исходный код нейронной сети, а также всех используемых в отчете программ приведен на гитхабе: <https://github.com/kremenevskiy/NeuralNetwork>

Список использованной литературы

- Основы статистики, машинного обучения и нейронных сетей. [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://statquest.org/>
- Neural Networks and Deep Learning. Michael Nielson 2019.
- <https://numpy.org/> - Документация NumPy

Приложение 1. Код программы network.py для реализации нейронной сети

```
import numpy as np

import random

import sys

import json


def sigmoid(z):

    return 1.0 / (1.0 + np.exp(-z))



def sigmoid_prime(z):

    return sigmoid(z) * (1 - sigmoid(z))



def vectorized_result(j):

    res = np.zeros((10, 1))

    res[j] = 1.0

    return res



class QuadraticCost(object):



    @staticmethod

    def fn(a, y):

        return 0.5 * np.linalg.norm(a-y) ** 2



    @staticmethod

    def delta(a, y, z):



        return (a - y) * sigmoid_prime(z)
```

```

class CrossEntropyCost(object):

    @staticmethod
    def fn(a, y):
        return np.sum(np.nan_to_num(-y * np.log(a) - (1-y) * np.log(1 - a)))

    @staticmethod
    def delta(a, y, z=0):
        return a - y

class Network(object):

    def __init__(self, sizes, cost=CrossEntropyCost):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases, self.weights = None, None
        self.default_weight_initializer()
        self.cost = cost

    def default_weight_initializer(self):
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(x, y) / np.sqrt(y) for x, y in zip(self.sizes[1:], self.sizes[:-1])]

    def large_weight_initializer(self):
        self.biases = [np.random.randn(y, 1) for y in self.sizes[1:]]
        self.weights = [np.random.randn(x, y) for x, y in zip(self.sizes[1:], self.sizes[:-1])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a) + b)
        return a

```

```

def SGD(self, training_data, epochs, mini_batch_size, eta, lmbda=0.0,
       validation_data=None, verbose=True,
       monitor_validation_cost=False, monitor_validation_accuracy=False,
       monitor_training_cost=False, monitor_training_accuracy=False):

    n = len(training_data)
    if validation_data:
        n_val = len(validation_data)

    # Metrics
    validation_cost, validation_accuracy = [], []
    training_cost, training_accuracy = [], []

    # train
    for j in range(epochs):
        np.random.shuffle(training_data)
        # optimize with numpy
        mini_batches = [training_data[k:k + mini_batch_size] for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta, lmbda, n)

        if verbose and \
                (monitor_training_cost or monitor_validation_cost or
                 monitor_training_accuracy or monitor_validation_accuracy):
            print(f'Epoch {j}: ', end='')

        if monitor_training_cost:
            cost = self.total_cost(training_data)
            training_cost.append(cost)
            if verbose:
                print(f'Cost_train[{cost:.6f}]', end='..')

        if monitor_training_accuracy:
            accuracy = self.accuracy(training_data, convert=True)

```

```

    training_accuracy.append(accuracy)

    if verbose:
        print(f'Acc_train[{accuracy:.6f}]', end='..')

    if monitor_validation_cost:
        cost = self.total_cost(validation_data)
        validation_cost.append(cost)

        if verbose:
            print(f'Cost_val[{cost:.6f}]', end='..')

        if monitor_validation_accuracy:
            accuracy = self.accuracy(validation_data, convert=True)
            validation_accuracy.append(accuracy)

        if verbose:
            print(f'Acc_val[{accuracy:.6f}]')

    return validation_cost, validation_accuracy, training_cost, training_accuracy


def update_mini_batch(self, mini_batch, eta, lmbda, n):
    n_batch = len(mini_batch)

    x = np.asarray([_x.ravel() for _x, _y in mini_batch]).T
    y = np.asarray([_y.ravel() for _x, _y in mini_batch]).T

    nabla_b, nabla_w = self.backprop(x, y)

    # update params

    self.weights = [(1-eta*(lmbda/n)) * w - (eta/n_batch) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b - (eta/n_batch) * nb for b, nb in zip(self.biases, nabla_b)]


def backprop(self, x, y):
    nabla_b = [0 for b in self.biases]
    nabla_w = [0 for w in self.weights]

    # feed forward

    activation = x
    activations = [x]

```

```

zs = []

for b, w in zip(self.biases, self.weights):
    z = np.dot(w, activation) + b
    zs.append(z)
    activation = sigmoid(z)
    activations.append(activation)

# backward pass
delta = self.cost.delta(activations[-1], y, zs[-1])
nabla_b[-1] = delta.sum(1).reshape([len(delta), 1])
nabla_w[-1] = np.dot(delta, activations[-2].T)
for l in range(2, self.num_layers):
    z = zs[-l]
    sp = sigmoid_prime(z)
    delta = np.dot(self.weights[-l+1].T, delta) * sp
    nabla_b[-l] = delta.sum(1).reshape([len(delta), 1])
    nabla_w[-l] = np.dot(delta, activations[-l-1].T)
return nabla_b, nabla_w

def total_cost(self, data, convert=False):
    cost = 0.0
    for x, y in data:
        a = self.feedforward(x)
        if convert:
            y = vectorized_result(y)
        cost += self.cost.fn(a, y) / len(data)
    return cost

def accuracy(self, data, convert=False):
    if convert:
        results = [(np.argmax(self.feedforward(x)), np.argmax(y)) for (x, y) in data]
    else:

```

```
    results = [(np.argmax(self.feedforward(x)), y) for (x, y) in data]
    return sum(int(x == y) for (x, y) in results) / len(data)
```

```
def save(self, filename):
    data = {
        "sizes": list(self.sizes),
        "weights": [w.tolist() for w in self.weights],
        "biases": [b.tolist() for b in self.biases],
        "cost": str(self.cost.__name__)
    }
```

```
    with open(filename, 'w') as f:
```

```
        json.dump(data, f)
```

```
@staticmethod
```

```
def load(filename):
    with open(filename, 'r') as f:
        data = json.load(f)
        cost = getattr(sys.modules['__main__'], data['cost'])
        sizes = data['sizes']
        net = Network(sizes, cost=cost)
        net.weights = [np.array(w) for w in data['weights']]
        net.biases = [np.array(b) for b in data['biases']]
    return net
```

```
@staticmethod
```

```
def get_metrics(data, start=-10, end=0):
    val_cost, val_acc, train_cost, train_acc = data[0], data[1], data[2], data[3]
    n = len(val_cost)
    if abs(start) > n:
        start = -n
    print(f'\tMetrics [{n+start}:{n+end}]')
    for i in range(start, end):
```

```
print(f'Epoch {n+i}: ', end='')

print(f'Cost_train[{train_cost[i]:.6f}]', end='..')
print(f'Acc_train[{train_acc[i]:.6f}]', end='..')
print(f'Cost_val[{val_cost[i]:.6f}]', end='..')
print(f'Acc_val[{val_acc[i]:.6f}]')
```

Приложение 2. Код программы mnist_loader.py для подгрузки данных

```
import numpy as np

from keras.datasets import mnist

data_mnist = mnist.load_data()

def vectorized_result(j):
    res = np.zeros((10, 1))
    res[j] = 1
    return res

train, test = data_mnist

x_train = [np.reshape(x / 255, (784, 1)) for x in train[0]]
y_train = [vectorized_result(j) for j in train[1]]
x_test = [np.reshape(x / 255, (784, 1)) for x in test[0]]
y_test = [j for j in test[1]]

training_data = list(zip(x_train, y_train))

training_data, validation_data = training_data[:int(0.8*len(training_data))], \
                                training_data[int(0.8*len(training_data)): len(training_data)]

test_data = list(zip(x_test, y_test))
```

```
mnist_loader = (training_data, validation_data, test_data)

if __name__ == '__main__':
    pass
```

Приложение 3. Код программы plotting.py для отрисовки графиков

```
import json
import random
import sys

import matplotlib.pyplot as plt
import numpy as np

def plot_training_cost(training_cost, num_epochs, training_cost_xmin=0):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(training_cost_xmin, num_epochs), training_cost[training_cost_xmin:num_epochs],
            color='red')
    ax.set_xlim([training_cost_xmin, num_epochs])
    ax.grid(True)
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Train Cost')
    ax.set_title('Cost on the training data')
    plt.show()

def plot_val_cost(validation_cost, num_epochs, validation_cost_xmin=0):
    fig = plt.figure()
```

```

ax = fig.add_subplot(111)

ax.plot(np.arange(validation_cost_xmin, num_epochs),
validation_cost[validation_cost_xmin:num_epochs], color='red')

ax.set_xlim([validation_cost_xmin, num_epochs])

ax.grid(True)

ax.set_xlabel('Epoch')

ax.set_ylabel('Validation Cost')

ax.set_title('Cost on the validation data')

plt.show()

def plot_training_accuracy(training_accuracy, num_epochs, training_accuracy_xmin=0):
    fig = plt.figure()

    ax = fig.add_subplot(111)

    ax.plot(np.arange(training_accuracy_xmin, num_epochs),
            training_accuracy[training_accuracy_xmin:num_epochs], color='red')

    ax.set_xlim([training_accuracy_xmin, num_epochs])

    ax.grid(True)

    ax.set_xlabel('Epoch')

    ax.set_ylabel('Accuracy on train')

    ax.set_title('Accuracy on the training data')

    plt.show()

def plot_validation_accuracy(validation_accuracy, num_epochs, test_accuracy_xmin=0):
    fig = plt.figure()

    ax = fig.add_subplot(111)

    ax.plot(np.arange(test_accuracy_xmin, num_epochs),
            validation_accuracy[test_accuracy_xmin:num_epochs], color='red')

    ax.set_xlim([test_accuracy_xmin, num_epochs])

    ax.grid(True)

    ax.set_xlabel('Epoch')

    ax.set_ylabel('Accuracy on validation')

```

```

ax.set_title('Accuracy on the validation data')

plt.show()

def plot_overlay(validation_accuracy, training_accuracy, num_epochs, ylim=0.9, xmin=0):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(np.arange(xmin, num_epochs), training_accuracy, color='orange', label='Accuracy on the train data')
    ax.plot(np.arange(xmin, num_epochs), validation_accuracy, color='blue', label='Accuracy on the validation data')
    ax.grid(True)
    ax.set_xlim([xmin, num_epochs])
    ax.set_ylim([ylim, 1])
    ax.set_xlabel('Epoch')
    ax.set_ylabel('Accuracy')
    ax.set_title('Compare accuracy on test and validation')
    plt.legend(loc="lower right")
    plt.show()

def make_plots(data, num_epochs, training_cost_xmin=0,
               train_accuracy_xmin=0,
               validation_cost_xmin=0,
               validation_accuracy_xmin=0,
               overlay_ylim=0.9):
    val_cost, val_acc, train_cost, train_acc = data[0], data[1], data[2], data[3]
    plot_training_cost(train_cost, num_epochs, training_cost_xmin)
    plot_training_accuracy(train_acc, num_epochs, train_accuracy_xmin)
    plot_val_cost(val_cost, num_epochs, validation_cost_xmin)
    plot_validation_accuracy(val_acc, num_epochs, validation_accuracy_xmin)
    plot_overlay(val_acc, train_acc, num_epochs, ylim=overlay_ylim)

```