

Муниципальное бюджетное общеобразовательное учреждение
“Средняя общеобразовательная школа №10”

Прикладная проектная работа
Тема: “Простейшая нейросеть на языке Python”

Выполнил: Смирнов Н. К.
ученик 10 “А” класса
Руководитель: Кабирова Л. И.
учитель информатики

Работа допущена к защите “ ____ ” _____ 2025 г.
Подпись руководителя проекта _____ (_____)

Введение.	3
1. Что такое нейросеть?	3
1.1 Определение нейронной сети.....	3
1.2 Устройство нейросети.....	3
1.3 Обучение нейросети.	4
2. Подготовка к созданию нейросети.....	5
2.1 Выбор библиотеки и исходных данных.	5
2.2 Установка зависимостей.....	5
2.3 Файл конфигурации.	5
2.4 Загрузка данных.....	5
2.5 Вывод картинок на экран.	7
3. Линейный слой.	7
3.1 Принцип работы.....	7
3.2 Реализация на языке Python.....	8
4. Функция активации.	9
4.1 Принцип работы.....	9
4.2 Реализация на языке Python.....	10
5. Функция потерь.	11
5.1 Принцип работы.....	11
5.2 Реализация на языке Python.....	11
6. Обратное распространение ошибки и оптимизация.	12
6.1 Стохастический градиентный спуск.	12
6.2 Обратное распространение ошибки.	13
7. Реализация алгоритма обучения нейросети на языке Python.	14
7.1 Функция потерь.....	14
7.2 “Softmax слой”.	15
7.3 Линейный слой.	16
8. Обучение нейросети и проверка результатов.....	17
8.1 Упрощение работы со слоями.	17
8.2 Обучение нейросети.	18
8.3 Проверка результатов.	20

Вывод.	22
Источники.	22

Введение.

Актуальность данной темы заключается в том, что на сегодняшний день происходит активное развитие в сфере искусственного интеллекта, нейросети становятся всё более массово используемыми.

Цель - создать рабочую нейросеть для распознавания рукописных цифр.

Продукт проекта - нейросеть по распознаванию рукописных цифр на языке Python.

Задачи проекта:

1. Описать общие принципы устройства и работы нейросети.
2. Рассмотреть математические формулы, лежащие в основе нейросети.
3. Создать и обучить нейросеть на языке Python.

1. Что такое нейросеть?

1.1 Определение нейронной сети.

Нейронная сеть (нейросеть) — математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации биологических нейронных сетей.

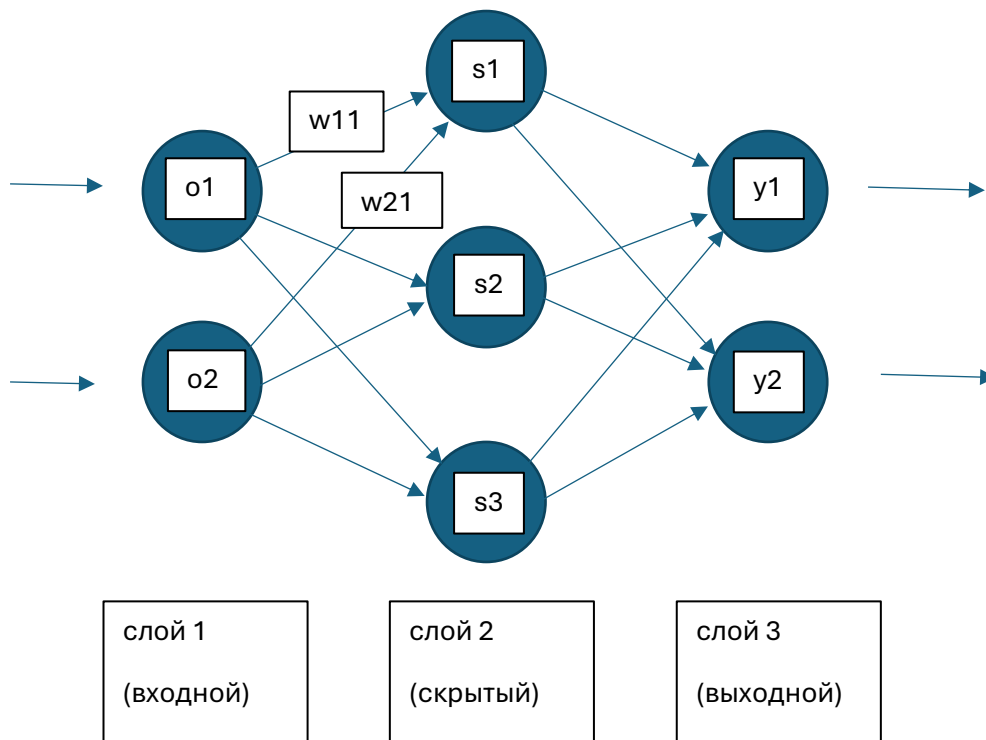
1.2 Устройство нейросети.

Нейросеть представляет собой систему соединённых и взаимодействующих между собой искусственных нейронов (узлов), объединённых в различные слои.

Данные в нейросети поступают на узлы входного слоя, вычисления происходят в скрытых слоях, окончательный результат обработки данных поступает из узлов выходного слоя.

Каждый узел скрытого слоя принимает сигналы от узлов предыдущего слоя и передает на следующий. Каждая связь между узлами имеет собственный вес.

Например:



$o1, o2, s1, s2, s3, y1, y2$ - нейроны (узлы)

$w11, w21$ и другие подобные связи между узлами - веса.

1.3 Обучение нейросети.

В основе работы нейронных сетей лежит процесс обучения.

В процессе обучения нейросети можно выделить следующие этапы:

1. Сначала производится вычисление ошибки (отклонения выдаваемого нейросетью результата от желаемого) на выходном слое на основе обработанных нейросетью данных.
2. Затем, происходит обратное распространение ошибок, т.е. вычисляется влияние каждого узла скрытого слоя на общую ошибку.
3. Наконец, на основе ошибок каждого узла происходит оптимизация весов.

Обучение повторяется циклично до тех пор, пока ошибка не будет минимизирована до определённого уровня.

Одна итерация (повторение) такого цикла называется эпохой.

2. Подготовка к созданию нейросети.

2.1 Выбор библиотеки и исходных данных.

Для работы с математическими вычислениями была выбрана сторонняя библиотека NumPy, т.к. её критически важные модули написаны на языках C и C++, которые работают в несколько раз быстрее Python.

Для отображения картинок на экране была выбрана сторонняя библиотека Matplotlib.

Данные для обучения нейросети были получены из базы данных рукописных цифр MNIST (60000 образцов).

2.2 Установка зависимостей.

Открываем папку проекта в терминале и устанавливаем необходимые библиотеки с помощью системы управления пакетами pip:

```
pip install numpy  
pip install matplotlib
```

В папке проекта создаём подкаталог `data`. В этот подкаталог скачиваем файл с названием `mnist.npz` (ссылка на файл: <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>) В нём и находится массив с данными для обучения.

2.3 Файл конфигурации.

В папке проекта создаём файл `config.py`.

```
config.py:  
N_CLASSES = 10  
CLASS_NAMES = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
```

В этом файле содержатся общие данные, используемые в различных модулях проекта, которые при необходимости можно поменять.

Переменная `N_CLASSES` содержит информацию о том, сколько всего вариаций (классов) того, что может быть изображено на картинке. В данном случае `N_CLASSES = 10`, т.к. на картинке могут быть изображены лишь цифры (символы от 0 до 9).

В переменной `CLASS_NAMES` - массив с “именами” этих вариаций. То есть просто перечисление цифр.

В дальнейшем в файл будут добавляться новые переменные.

2.4 Загрузка данных.

В каталоге data создаём файл `mnist.py`.

`mnist.py`:

```
import numpy as np
from pathlib import Path
from config import N_CLASSES

def get_mnist(shuffle: bool = False) -> tuple[np.ndarray, np.ndarray]:
    with np.load(Path(__file__).parent.absolute() / "mnist.npz") as f:
        images, classes = f["x_train"], f["y_train"]
        images = images.astype("float64") / 255
        images = np.reshape(images, (images.shape[0], images.shape[1] *
images.shape[2]))
        classes = np.eye(N_CLASSES)[classes]

    if shuffle:
        p = np.random.permutation(len(images))
        images = images[p]
        classes = classes[p]
    return images, classes
```

Функция `get_mnist` загружает данные из файла `mnist.npz`, преобразует их для корректной работы нейросети и возвращает в кортеже из массивов NumPy.

В переменной `images` массив изображений с рукописными цифрами размерностью 60000x24x24.

Каждая ячейка массива принимает значение от 0 до 255, поэтому необходимо поделить его на 255, чтобы оно находилось в диапазоне от 0 до 1:

```
images = images.astype("float64") / 255
```

Также необходимо, чтобы каждая картинка представляла собой 784-мерный вектор (вместо 24x24 матрицы):

```
images = np.reshape(images, (images.shape[0], images.shape[1]*images.shape[2]))
```

В переменной же `classes` находятся данные о том, какой класс (в нашем случае это цифра) соответствует каждой картинке массива `images`. Размерность `classes` – 60000.

Но необходимо, чтобы каждая цифра была представлена в виде 10-мерного (т.к. цифры - символы от 0 до 9, `N_CLASSES` = 10) “one-hot” вектора.

Например, цифра 5 должна быть представлена в виде массива

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

Для этого изменим значение переменной `classes` следующим образом:

```
classes = np.eye(N_CLASSES)[classes]
```

Итоговая размерность массива `classes` – 60000x10.

Если при вызове функции аргумент `shuffle = True`, то данные перемешиваются, что помогает при обучении нейросети.

2.5 Вывод картинок на экран.

В папке проекта создадим каталог `utils` и файл `images.py` в этом каталоге.

`images.py`:

```
import matplotlib.pyplot as plt
import numpy as np
from config import CLASS_NAMES

def imshow(images: np.ndarray, classes: np.ndarray | list, amount: int = 6) ->
None:
    for i in range(amount):
        plt.subplot(2, 3, i + 1)
        img = images[i].reshape([28, 28])
        plt.imshow(img, cmap="gray")
        plt.title(CLASS_NAMES[int(classes[i])])
    plt.show()
```

Функция `imshow` принимает следующие данные:

`images` - массив с картинками

`classes` - массив с классами, соответствующими каждой картинке из `images`

`amount` - количество картинок, которое требуется вывести на экран (по умолчанию 6)

Затем, сделав некоторые преобразования с массивами, выводит картинки и соответствующие им классы на экран.

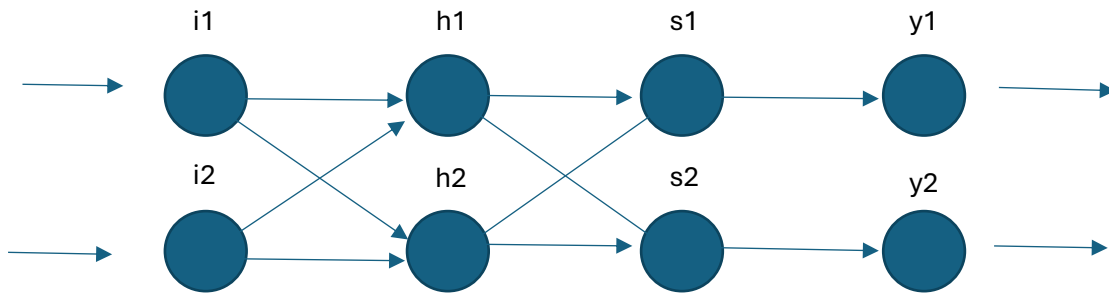
Из преобразований стоит отметить преобразование каждой картинки из 784-мерного вектора обратно в 28x28 матрицу:

```
img = images[i].reshape([28, 28])
```

3. Линейный слой.

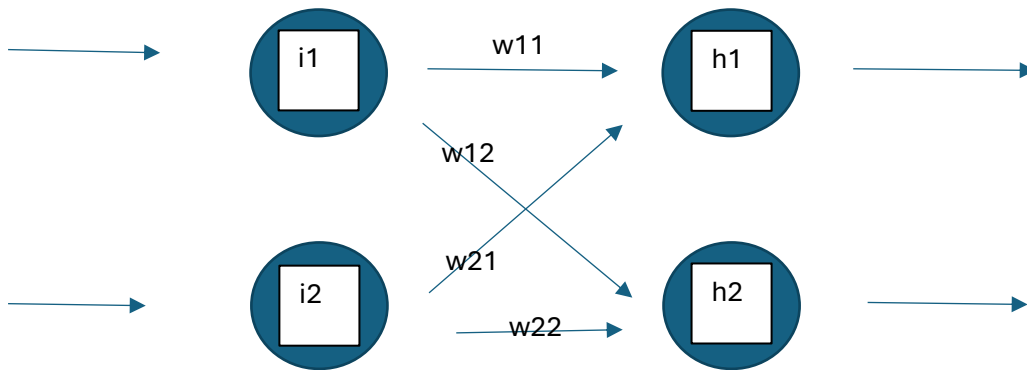
3.1 Принцип работы.

Итак, для упрощения дальнейших рассуждений представим следующую нейросеть, которая, впрочем, имеет такой же принцип работы как и та, которую мы собираемся создать :



В этой нейросети узлы $h1$ и $h2$ будут представлять линейный слой.

Этот слой обрабатывает данные, поступающие из предыдущего слоя, выполняя их матричное умножение со своими весами:



$$h_1 = i_1 w_{11} + i_2 w_{21} \quad h_2 = i_1 w_{12} + i_2 w_{22}$$

Тогда общий вид: $h_j = \sum_{i=1}^n o_i w_{ij}$, где:

h_j - значение, которое вычисляет узел линейного слоя.

n - количество узлов в предыдущем слое.

o_i - значение, которое было вычислено узлом предыдущего слоя.

w_{ij} - вес связи между узлом предыдущего слоя и текущим узлом линейного слоя.

Подобные вычисления как раз удобно представить в виде умножения вектора на матрицу. Для вышеупомянутой нейросети это будет выглядеть так:

$$(i_1, i_2) \times \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} = (i_1 w_{11} + i_2 w_{21}, i_1 w_{12} + i_2 w_{22}) = (h_1, h_2)$$

3.2 Реализация на языке Python.

В папке проекта создадим каталог `network`, а в каталоге `network` - файл `layers.py`.

`layers.py`:


```
import numpy as np
from abc import ABC, abstractmethod

class AbstractLayer(ABC):
    @abstractmethod
    def __call__(self, X: np.ndarray) -> np.ndarray: ...
```

Слои будет удобно представить в виде классов. Для того, чтобы со слоями можно было взаимодействовать единым образом, был создан абстрактный класс (“шаблон” для остальных слоёв) `AbstractLayer`, который постепенно будет дополняться другими методами, которые необходимо реализовать в дочерних классах.

`layers.py`:

```
class LinearLayer(AbstractLayer):
    def __init__(self, inputs: int, outputs: int) -> None:
        self.inputs = inputs
        self.outputs = outputs
        self.weights = np.random.uniform(0, 1 / np.sqrt(self.inputs),
        [self.inputs, self.outputs])

    def __call__(self, X: np.ndarray) -> np.ndarray:
        signal = X @ self.weights
        return signal
```

В классе `LinearLayer` дополнительно был реализован метод `__init__`, т.к. необходимо указать количество узлов предыдущего слоя и кол-во узлов в самом слое.

Значение весов `weights` выбирается случайным образом из интервала $(0; \frac{1}{\sqrt{n}})$, где n - количество узлов предыдущего слоя.

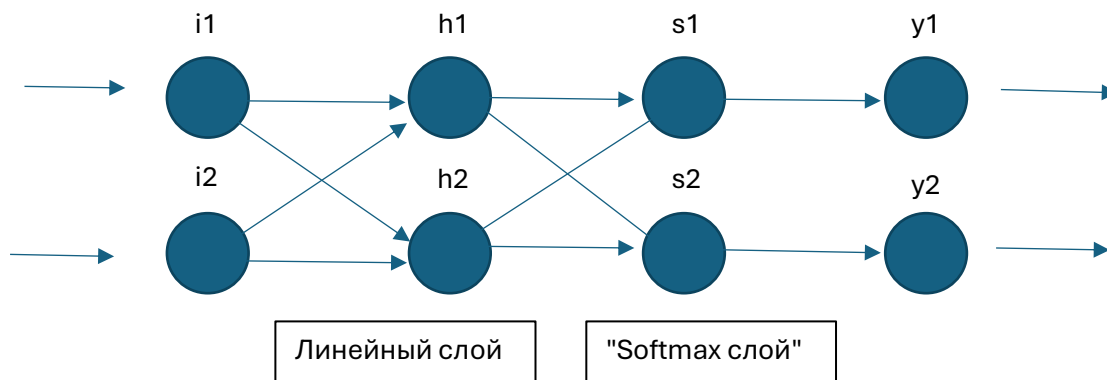
В методе же `__call__` и производится умножение вектора входных данных `X` на матрицу весов линейного слоя.

4. Функция активации.

4.1 Принцип работы.

Для преобразования обработанных входных данных в выходные используется функция активации. Для задач классификации подходит

многопеременная логистическая функция (Softmax).



Для удобства в ранее упомянутой нейросети функция активации была представлена в виде отдельного слоя ("Softmax слой") с узлами $s1$ и $s2$. В таком слое количество узлов всегда будет соответствовать количеству узлов предыдущего слоя. Выходной слой (узлы $y1$ и $y2$) будет принимать и давать на выход те же значения, что и "Softmax слой".

Сама функция Softmax выглядит так:

$$s_j = \frac{e^{o_i}}{\sum_{k=1}^n e^{o_k}}, \text{ где:}$$

s_j - значение, которое вычисляет узел Softmax слоя.

o_i - значение узла предыдущего слоя, соответствующего узлу Softmax слоя, т.е $i = j$.

o_k - значение узла предыдущего слоя.

Смысл Softmax функции в том, что она генерирует распределение вероятностей принадлежности картинки тому или иному классу.

Для вышеуказанной нейросети Softmax слой будет давать на выход следующий вектор (он же будет результатом работы всей этой нейросети):

$$\left(\frac{e^{h1}}{e^{h1} + e^{h2}}, \frac{e^{h2}}{e^{h1} + e^{h2}} \right) = (s1, s2) = (y1, y2)$$

4.2 Реализация на языке Python.

layers.py

```

class SoftmaxLayer(AbstractLayer):
    def __call__(self, X: np.ndarray) -> np.ndarray:
        z = X - np.max(X)
        signal = np.exp(z) / np.sum(np.exp(z))
        return signal
    
```

В классе SoftmaxLayer нет метода `__init__`, потому что слой не имеет весов, а количество узлов зависит от предыдущего слоя.

Также стоит отметить, что на практике во избежание слишком высоких значений экспоненты из всех значений вектора входных данных вычитается максимальное:

```
z = X - np.max(X)
```

5. Функция потерь.

5.1 Принцип работы.

Для вычисления отклонения (ошибки) выдаваемого нейросетью результата от желаемого к данным из выходного слоя применяется функция потерь.

Для задач классификации подходит такая функция потерь как перекрёстная энтропия (Cross-Entropy). Она оценивает разницу между двумя распределениями вероятностей. В нашем случае первое распределение - вектор, полученный из Softmax функции (“предсказанные” значения); второе распределение - one-hot вектор с классом картинки (“действительные” значения).

Формула для перекрёстной энтропии:

$$L = - \sum_{i=1}^n y_i \ln p_i, \text{ где:}$$

L - значение (ошибка), которое вычисляет функция потерь.

n - кол-во классов (измерений) в векторах (должно совпадать).

i - индекс сравниваемых значений.

y_i - “действительное” значение.

p_i - “предсказанное” значение.

5.2 Реализация на языке Python.

В каталоге network создадим файл loss.py.

loss.py:

```
import numpy as np
from abc import ABC, abstractmethod
from config import EPSILON

class AbstractLoss(ABC):
    @abstractmethod
    def __call__(self, pred: np.ndarray, ref: np.ndarray) -> np.float32: ...

class CrossEntropyLoss(AbstractLoss):
    def __call__(self, pred: np.ndarray, ref: np.ndarray) -> np.float32:
        pred_e = pred + ((np.abs(pred) < EPSILON) * EPSILON)
        loss = -1 * np.sum(ref * np.log(pred_e))
        return loss
```

А в файле `config.py` в папке проекта добавим следующую переменную:

```
EPSILON = 1e-7
```

Как и в случае со слоями используется абстрактный класс

`AbstractLoss`. Метод `__call__` принимает вектор “предсказанных” значений `pred` и вектор “действительных” `ref`.

Для того, чтобы в логарифм случайно не попал 0, значения, модуль которых меньше `EPSILON`, становятся равны 10^{-7} :

```
pred_e = pred + ((np.abs(pred) < EPSILON) * EPSILON)
```

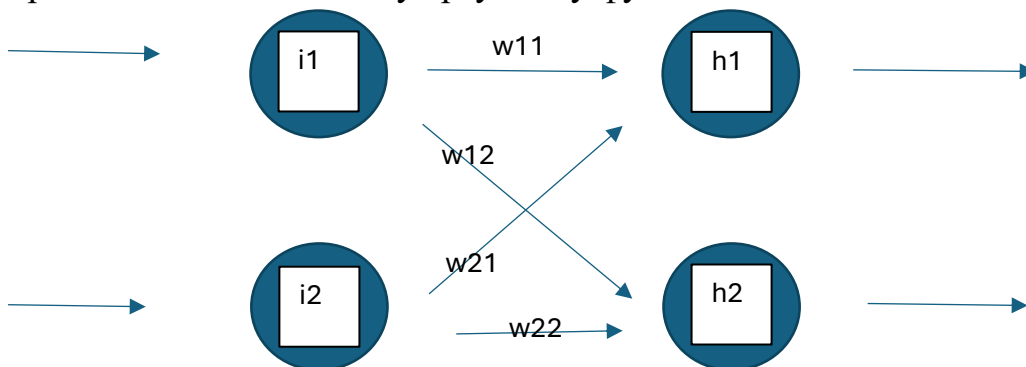
6. Обратное распространение ошибки и оптимизация.

6.1 Стохастический градиентный спуск.

Итак, наша нейросеть может обрабатывать данные и вычислять на их основе общую ошибку. Теперь же для полноценной работы нейросети необходимо реализовать алгоритм её обучения.

Ошибка нейросети зависит от множества переменных, а нам необходимо эту ошибку минимизировать. Для этой задачи подойдут методы, связанные с нахождением минимума функции, в частности метод стохастического градиентного спуска.

Метод стохастического градиентного спуска заключается в нахождении локального минимума функции при помощи движения в обратном направлении её градиенту - вектору, указывающему направление наискорейшего роста функции, компонентами которого являются частные производные по каждому аргументу функции.



Например, найдём градиент для узла $h1$. Известно, что $h_1(w_{11}, w_{21}) = i_1 w_{11} + i_2 w_{21}$.

Найдём частные производные по каждому аргументу узла:

$$\frac{dh_1}{dw_{11}} = i_1, \quad \frac{dh_1}{dw_{21}} = i_2.$$

Тогда градиентом узла $h1$ будет следующий вектор: (i_1, i_2) .

Но надо учесть, что узел $h1$ оказывает влияние на общую ошибку.

Пусть будет функция потерь $E = E(h1, h2)$. Воспользовавшись цепным правилом дифференцирования сложной функции, получим следующий

вектор: $\left(\frac{dh_1}{dw_{11}} \frac{dE}{dh_1}, \frac{dh_1}{dw_{21}} \frac{dE}{dh_1}\right) = \left(i_1 \frac{dE}{dh_1}, i_2 \frac{dE}{dh_1}\right) = (\Delta w_{11}, \Delta w_{21})$.

Осталось лишь обновить веса (оптимизировать) для приближения к минимуму E :

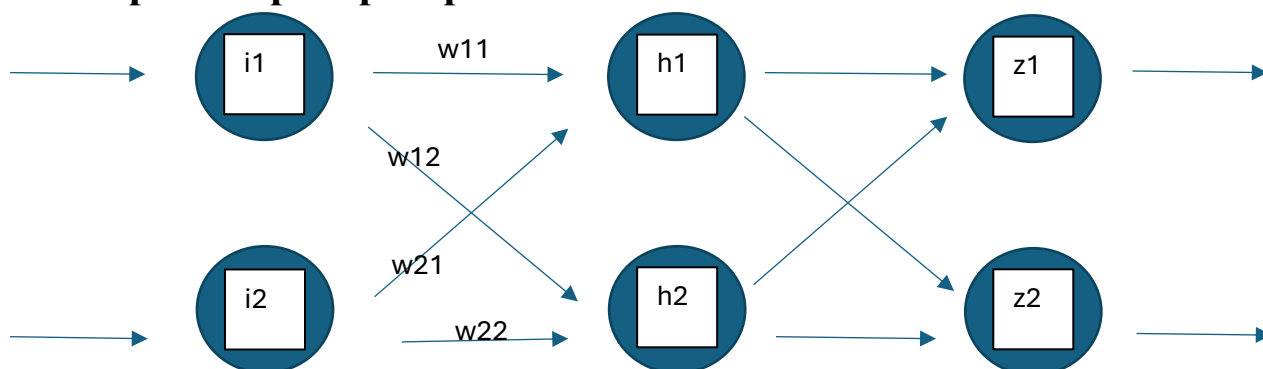
$w'_{11} = w_{11} - \eta \cdot \Delta w_{11}$, $w'_{21} = w_{21} - \eta \cdot \Delta w_{21}$, где:

w' - новое значение веса.

η - темп обучения ("Размер шага". Нужен, чтобы при движении против градиента случайно не "перепрыгнуть" минимум).

Для обучения нейросети необходимо в каждой эпохе обновлять веса каждого линейного слоя.

6.2 Обратное распространение ошибки.



Но как, например, оптимизировать вес $w11$ в случае, если нейросеть состоит из нескольких скрытых слоёв?

Добавим ещё один слой с узлами $z1, z2$ к предыдущей нейросети.

Попробуем оптимизировать вес $w11$:

Пусть функция потерь $E = E(z1, z2)$, тогда: $\frac{dE}{dw_{11}} = \frac{dE}{dh_1} \frac{dh_1}{dw_{11}}$. Но как найти $\frac{dE}{dh_1}$?

Для этого подойдёт метод обратного распространения ошибки. В общем виде он выглядит так:

$\frac{dE}{do_i} = \sum_{j=1}^n \frac{do_j}{do_i} \frac{dE}{do_j}$, где:

n - количество узлов в следующем слое.

o_j - узел следующего слоя.

o_i - узел текущего слоя.

Тогда: $\frac{dE}{dh_1} = \frac{dz_1}{dh_1} \frac{dE}{dz_1} + \frac{dz_2}{dh_1} \frac{dE}{dz_2}$

Следовательно, $\frac{dE}{dw_{11}} = \frac{dh_1}{dw_{11}} \left(\frac{dz_1}{dh_1} \frac{dE}{dz_1} + \frac{dz_2}{dh_1} \frac{dE}{dz_2} \right)$.

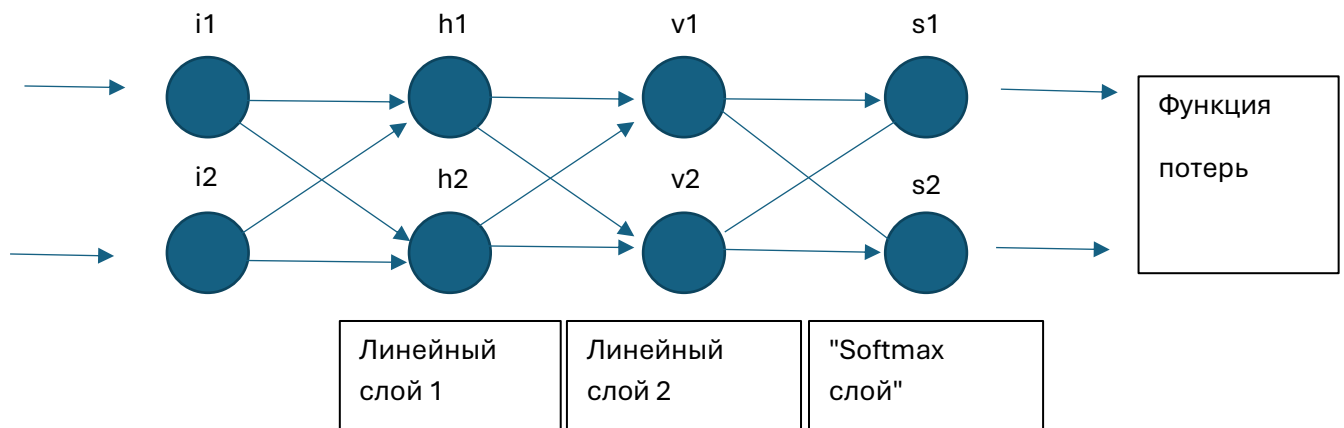
Эти вычисления также удобно представить в виде умножения вектора на матрицу. Для узлов $h1, h2$:

$$\begin{pmatrix} \frac{dE}{dz_1} & \frac{dE}{dz_2} \end{pmatrix} \times \begin{pmatrix} \frac{dz_1}{dh_1} & \frac{dz_1}{dh_2} \\ \frac{dz_2}{dh_1} & \frac{dz_2}{dh_2} \end{pmatrix} = \begin{pmatrix} \frac{dE}{dh_1} & \frac{dE}{dh_2} \end{pmatrix}$$

7. Реализация алгоритма обучения нейросети на языке Python.

7.1 Функция потерь.

Представим нашу будущую нейросеть в упрощённом виде:



Последний слой в этой нейросети - “Softmax” слой (значения в выходном слое будут такими же, поэтому он был убран) с узлами $s1, s2$.

Функция потерь: $E = L = -\sum_{i=1}^n y_i \ln p_i$

Тогда: $\frac{dE}{ds_i} = -\frac{y_i}{s_i}$, т.к. для этой нейросети $s_i = p_i$.

В `network/loss.py`:

Добавим новый метод в `AbstractLoss`:

```
@abstractmethod
def calculate_grad(self, *args, **kwargs) -> None: ...
```

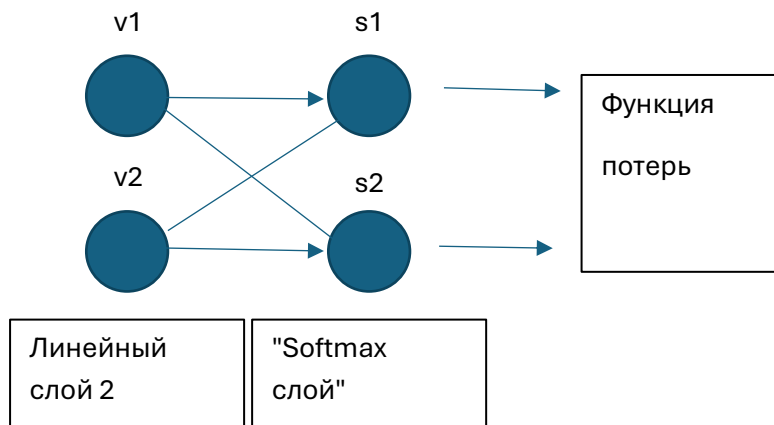
Реализуем его в `CrossEntropyLoss`:

```
def calculate_grad(self, pred: np.ndarray, ref: np.ndarray) -> None:
    self.grad = -1 * ref / pred
```

Градиент будет сразу вычисляться в методе `__call__`:

```
self.calculate_grad(pred_e, ref)
```

7.2 “Softmax слой”.



Известно, что $s_j = \frac{e^{o_j}}{\sum_{k=1}^n e^{o_k}}$, тогда для части нейросети на картинке: $\frac{ds_1}{dv_1} =$

$$\frac{e^{v_1}(e^{v_1}+e^{v_2})-e^{2v_1}}{(e^{v_1}+e^{v_2})^2} = \frac{e^{v_1}}{e^{v_1}+e^{v_2}} - \left(\frac{e^{v_1}}{e^{v_1}+e^{v_2}}\right)^2 = s_1(1-s_1)$$

$$\frac{ds_1}{dv_2} = \frac{0 - e^{v_1}e^{v_2}}{(e^{v_1}+e^{v_2})^2} = -s_1s_2 \quad (v_2 \text{ тоже вносит свой вклад})$$

Учитывая градиент по s_1 и s_2 и пользуясь методом обратного распространения ошибки, представим градиент по v_1 и v_2 в виде умножения вектора на матрицу:

$$\begin{pmatrix} \frac{dE}{ds_1} & \frac{dE}{ds_2} \end{pmatrix} \times \begin{pmatrix} \frac{ds_1}{dv_1} & \frac{ds_1}{dv_2} \\ \frac{ds_2}{dv_1} & \frac{ds_2}{dv_2} \end{pmatrix} = \begin{pmatrix} \frac{dE}{ds_1} & \frac{dE}{ds_2} \end{pmatrix} \times \begin{pmatrix} s_1(1-s_1) & -s_1s_2 \\ -s_2s_1 & s_2(1-s_2) \end{pmatrix} = \begin{pmatrix} \frac{dE}{dv_1} & \frac{dE}{dv_2} \end{pmatrix}$$

Стоит отметить, что матрицу можно представить в виде:

$$\begin{pmatrix} s_1(1-s_1) & -s_1s_2 \\ -s_2s_1 & s_2(1-s_2) \end{pmatrix} = \begin{pmatrix} s_1 & 0 \\ 0 & s_2 \end{pmatrix} - \begin{pmatrix} s_1^2 & s_1s_2 \\ s_2s_1 & s_2^2 \end{pmatrix}$$

Также стоит отметить, что если транспонировать такую матрицу, то это ни на что не повлияет.

В совокупности эти два свойства существенно облегчат программную реализацию.

В `network/layers.py`:

Добавим новые методы в `AbstractLayer`:

```
@abstractmethod
def calculate_derivatives(self, *args, **kwargs) -> None: ...
@abstractmethod
def backpropagate(self, next_grad: np.ndarray) -> np.ndarray: ...
```

Реализуем их в `SoftmaxLayer`:

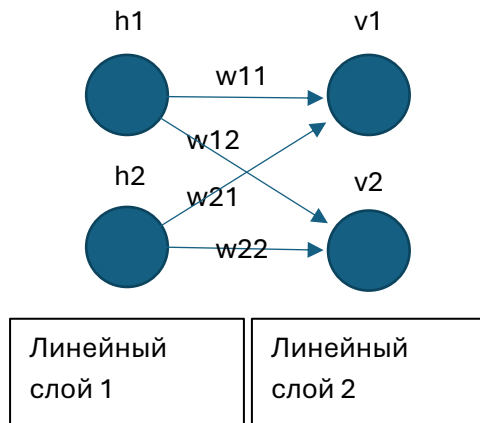
```
def calculate_derivatives(self) -> None:
    self.derivatives_to_prev = np.diagflat(self.signal) - self.signal[:,
np.newaxis] * self.signal[np.newaxis, :]
def backpropagate(self, next_grad: np.ndarray) -> np.ndarray:
    grad = next_grad @ self.derivatives_to_prev
```

```
return grad
```

В методе `calculate_derivatives` как раз и были применены свойства матрицы частных производных для “Softmax слоя”. Матрица производных также будет вычисляться в методе `__call__`:

```
self.calculate_derivatives()
```

7.3 Линейный слой.



Известно, что $h_j = \sum_{i=1}^n o_i w_{ij}$, тогда для части нейросети на картинке:

$$\frac{dv_1}{dh_1} = w_{11} \quad \frac{dv_1}{dh_2} = w_{21}$$

Тогда градиент получается в ходе следующих вычислений:

$$\left(\frac{dE}{dv_1}, \frac{dE}{dv_2} \right) \times \begin{pmatrix} \frac{dv_1}{dh_1} & \frac{dv_1}{dh_2} \\ \frac{dv_2}{dh_1} & \frac{dv_2}{dh_2} \end{pmatrix} = \left(\frac{dE}{dv_1}, \frac{dE}{dv_2} \right) \times \begin{pmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \end{pmatrix} = \left(\frac{dE}{dh_1}, \frac{dE}{dh_2} \right)$$

Стоит отметить, что матрица частных производных - транспонированная матрица весов для “Линейного слоя 2”.

Так как градиент для узлов линейного слоя и оптимизация весов уже были рассмотрены в пункте 6.2, сразу перейдём к реализации.

В `network/layers.py`:

Реализуем методы `calculate_derivatives` и `backpropagate` в `LinearLayer`:

```
def calculate_derivatives(self, X: np.ndarray) -> None:
    self.derivatives_to_prev = self.weights.T
    self.derivatives_to_weights = np.repeat(X[np.newaxis, :], self.outputs, 0)
def backpropagate(self, next_grad: np.ndarray) -> np.ndarray:
    delta_weights = (next_grad[:, np.newaxis] * self.derivatives_to_weights).T
    grad_by_prev = next_grad @ self.derivatives_to_prev
    return grad_by_prev
```

Стоит обратить внимание на следующую строку:

```
delta_weights = (next_grad[:, np.newaxis] * self.derivatives_to_weights).T
```


Вместо того, чтобы по отдельности умножать частную производную по узлу на его градиент по весам, в NumPy можно сделать нечто подобное (с математической точки зрения уравнение ниже не имеет смысла!!!):

$$\begin{pmatrix} \frac{dE}{dv_1} \\ \frac{dE}{dv_2} \end{pmatrix} * \begin{pmatrix} \frac{dv_1}{dw_{11}} & \frac{dv_1}{dw_{21}} \\ \frac{dv_2}{dw_{12}} & \frac{dv_2}{dw_{22}} \end{pmatrix} = \begin{pmatrix} \frac{dE}{dw_{11}} & \frac{dE}{dw_{21}} \\ \frac{dE}{dw_{12}} & \frac{dE}{dw_{22}} \end{pmatrix} = \begin{pmatrix} \Delta w_{11} & \Delta w_{21} \\ \Delta w_{12} & \Delta w_{22} \end{pmatrix}$$

То есть несколько градиентов были “объединены” в матрицу. Это существенно облегчает программную реализацию.

Реализуем также метод `optimize_weights`:

```
def optimize_weights(self, delta_weights: np.ndarray) -> None:
    self.weights -= LEARNING_RATE * delta_weights
```

В этом методе веса так же оптимизируются с помощью транспонированной “матрицы градиентов”:

$$\begin{pmatrix} w'_{11} & w'_{12} \\ w'_{21} & w'_{22} \end{pmatrix} = \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{pmatrix} - \eta \begin{pmatrix} \Delta w_{11} & \Delta w_{21} \\ \Delta w_{12} & \Delta w_{22} \end{pmatrix}^T$$

Веса будут оптимизироваться сразу при обратном распространении ошибки в методе `backpropagate`:

```
self.optimize_weights(delta_weights)
```

Переменная `LEARNING_RATE` была добавлена в файл `config.py`:

```
LEARNING_RATE = 0.000125
```

Значение этой переменной можно изменять в зависимости от результатов обучения нейросети.

8. Обучение нейросети и проверка результатов.

8.1 Упрощение работы со слоями.

Для того, чтобы не работать с каждым слоем по отдельности, создадим специальный класс `Network`.

`network/network.py`:

```
import numpy as np
import json
from pathlib import Path
from .layers import AbstractLayer

class Network:
    def __init__(self, layers: list[AbstractLayer]) -> None:
        self.layers = layers

    def __call__(self, X: np.ndarray) -> np.ndarray:
        x = X
        for layer in self.layers:
            x = layer(x)
```

```

        return x

    def backpropagate(self, grad: np.ndarray) -> None:
        for layer in reversed(self.layers):
            grad = layer.backpropagate(grad)

    def save_weights(self, path: str | Path) -> None:
        weights_list = [layer.weights.tolist() for layer in self.layers if
hasattr(layer, "weights")]
        with open(path, "w") as f:
            json.dump(weights_list, f)

    def load_weights(self, path: str | Path) -> None:
        with open(path, "r") as f:
            weights_list = json.load(f)
        for i in range(len(weights_list)):
            layer = self.layers[i]
            if hasattr(layer, "weights"):
                layer.weights = np.array(weights_list[i])

```

Класс `Network` осуществляет работу со списком, в котором содержатся по порядку слои нейросети (`self.layers`).

В методе `__call__` входные данные последовательно обрабатываются в каждом слое.

В методе `backpropagate` происходит, начиная с конца, обратное распространение ошибки на основе градиента функции потерь по узлам последнего слоя (`grad`) и оптимизация весов.

В методе `save_weights` веса линейных слоёв записываются в файл формата `.json` по пути `path`.

В методе `load_weights` считываются сохранённые веса из `.json` файла и записываются в веса линейных слоёв.

8.2 Обучение нейросети.

Создадим файл `model.py`, где будут храниться переменные с объектами `Network` и `CrossEntropyLoss`.

`model.py`:

```

from network.network import Network
from network.layers import *
from network.loss import CrossEntropyLoss

model = Network(
    [
        LinearLayer(784, 100),
        LinearLayer(100, 10),

```

```

        SoftmaxLayer()
    ]
)
loss_fn = CrossEntropyLoss()

```

В списке при создании объекта `Network` указаны слои нейросети:

1. Линейный слой со 100 узлами, принимающий на вход 784-мерный вектор.
2. Линейный слой с 10 узлами, принимающий на вход 100-мерный вектор.
3. “Softmax слой”, он же и выходной слой.

Также добавим новые переменные в файл конфигурации.

`config.py`:

```

EPOCHS = 15
WEIGHTS_PATH = Path(__file__).parent.absolute() / "weights.json"

```

`EPOCHS` - количество эпох обучения.

`WEIGHTS_PATH` - путь для файла с весами `weights.json`.

Далее, создадим цикл для обучения нейросети.

`train.py`:

```

import numpy as np
from data.mnist import get_mnist
from model import *
from config import EPOCHS, WEIGHTS_PATH

images, classes = get_mnist(shuffle=True)

if WEIGHTS_PATH.is_file():
    model.load_weights(WEIGHTS_PATH)

for e in range(EPOCHS):
    running_corrects = 0
    running_loss = 0.0
    for img, ref in zip(images, classes):
        pred = model(img)
        loss_val = loss_fn(pred, ref)
        model.backpropagate(loss_fn.grad)

        pred_class = np.argmax(pred)
        ref_class = np.argmax(ref)

        running_corrects += (pred_class == ref_class)
        running_loss += loss_val
    epoch_acc = running_corrects / len(images) * 100
    epoch_loss = running_loss / len(images)
    print(e + 1, epoch_acc, epoch_loss, sep="\t")

```

```
model.save_weights(WEIGHTS_PATH)
```

Данные записываются в переменные `images` и `classes` с помощью функции `get_mnist()`, данные также перемешиваются, т.к `shuffle=True`.

Далее, если файл с сохранёнными весами нейросети уже существует, то веса из него загружаются:

```
if WEIGHTS_PATH.is_file():  
    model.load_weights(WEIGHTS_PATH)
```

Нейросеть в цикле обучается отдельно по каждой из 60000 картинок:

```
for img, ref in zip(images, classes)
```

На основе каждой картинки высчитывается текущая потеря (`running_loss`) и количество правильных классификаций (`running_corrects`), а на их основе в конце эпохи вычисляется общая точность (`epoch_acc`) и общая потеря (`epoch_loss`), которые затем выводятся в терминале.

Так как класс картинки (“действительное” значение, `ref_class`) и, соответственно, выходные данные нейросети (“предсказанное” значение, `pred_class`) представлены в виде 10-мерного вектора с распределением вероятностей, то берётся их максимальное значение:

```
pred_class = np.argmax(pred)  
ref_class = np.argmax(ref)
```

Осталось лишь обучать нейросеть до тех пор, пока точность не будет максимизирована, а потеря - минимизирована.

8.3 Проверка результатов.

В ходе обучения нейросети удалось достичь максимальной точности в 93,4%. Для проверки результатов создадим файл `test.py`:

`test.py`:

```
import numpy as np  
from model import model  
from config import WEIGHTS_PATH, NUM_IMAGES  
from utils.images import imshow  
from data.mnist import get_mnist  
  
model.load_weights(WEIGHTS_PATH)  
  
images, classes = get_mnist(shuffle=True)  
sel_images, sel_classes = images[:NUM_IMAGES], classes[:NUM_IMAGES]  
  
preds = np.zeros(NUM_IMAGES)
```

```

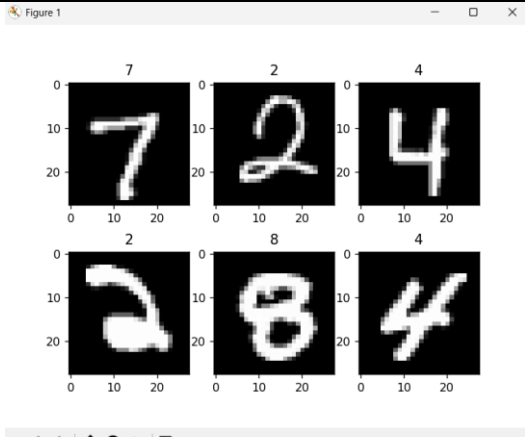
for i in range(NUM_IMAGES):
    pred = model(sel_images[i])
    preds[i] = np.argmax(pred)

print(preds == np.argmax(sel_classes, -1))
imshow(sel_images, preds)

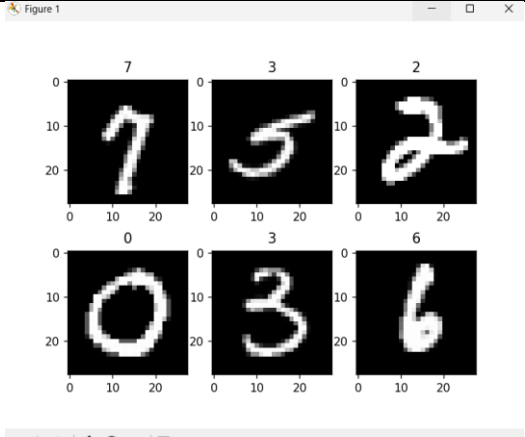
```

На экран будут выводиться случайно выбранные картинки (`sel_images`) и их “предсказанные” классы (`preds`), а в терминал - массив с данными о соответствии “предсказанных” и “действительных” классов.

Например:

Экран	Терминал
	<pre>[True True True True True True]</pre>

Но из-за того, что точность не равна 100%, могут возникать ошибки:

Экран	Терминал
	<pre>[True False True True True True]</pre>

Нейросеть в данном случае идентифицировала цифру “5” как “3”.

Вывод.

Основная цель проекта была выполнена - получилось создать рабочую нейросеть по распознаванию рукописных цифр. Код продукта проекта был выложен в общий доступ по следующей ссылке:

<https://github.com/5kadi/network-school-project>

Касаясь задач проекта:

1. Основные принципы устройства и работы нейросети были описаны в общих чертах ещё в первой главе (1. Что такое нейросеть?), а затем информация о них постепенно дополнялась в следующих главах.
2. Прежде чем программно реализовать ту или иную составляющую нейросети, уделялось особое внимание необходимым математическим формулам. Но для понимания этих формул зачастую необходимы более углубленные познания в математике.
3. Нейросеть удалось программно реализовать и обучить. Но, к сожалению, не удалось достигнуть 100% точности предсказаний - нейросеть иногда будет неправильно идентифицировать цифры.

Источники.

- <https://www.youtube.com/watch?v=EMXfZB8FVUA&list=PLqnsIRFeH2UrcDBWF5mfPGpqQDSta6VK4>
- https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R167000Dx_ZCJB-3pi
- <https://habr.com/ru/articles/714988/>