УНИВЕРСИТЕТ
искусственного
интеллекта

Введение в нейронные сети

занятие 1

«Введение в нейронные сети»

Занятие № 1

Создание простой нейронной сети

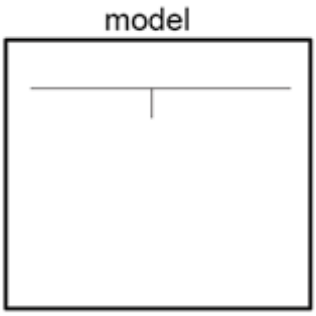
Библиотека Keras предоставляет два способа создания модели нейронной сети: модель прямого распространения (Sequential) и функциональное программирование (Functional API). На этом занятии мы рассмотрим первый вариант: модель Sequential.

Sequential – это модель, в которой слои располагаются последовательно один за другим, подобно цепочке блоков. Для создания такой модели первым шагом ВСЕГДА является создание пустого каркаса с помощью вызова конструктора модели:

```
model = Sequential()
```

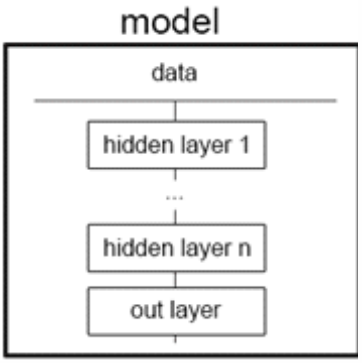
Здесь model – имя создаваемой модели (может быть произвольным именем), Sequential() – конструктор модели. С помощью данной конструкции мы создаем пустую модель, пока еще не содержащую ни одного слоя.

Создание простой нейронной сети



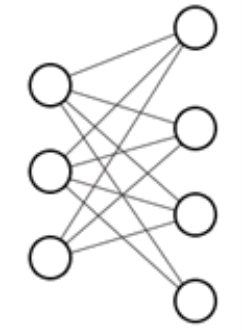
Для добавления слоев в модель используется метод .add():

```
model.add(Dense(32, input_dim=128))
```



Данный метод добавляет первый слой в модель (если модель еще пустая) или присоединяет слой к последнему созданному. Используя последовательность вызовов метода .add(), мы получаем возможность формировать необходимую нам архитектуру нейронной сети, состоящую из последовательных слоев.

Keras предоставляет широкий набор уже готовых слоев для решения различных задач (Dense, Conv2D, Flatten, Dropout, LSTM и др.), мы будем изучать их по мере прохождения курса (помимо этого имеется возможность создавать свои собственные слои, о чем будет рассказано на последующих занятиях).



Знакомство со слоями Keras начнем со слоя Dense – полносвязный слой. Это слой, все нейроны которого связаны с нейронами предыдущего слоя (или со всеми входными параметрами, если слой Dense является первым слоем модели):

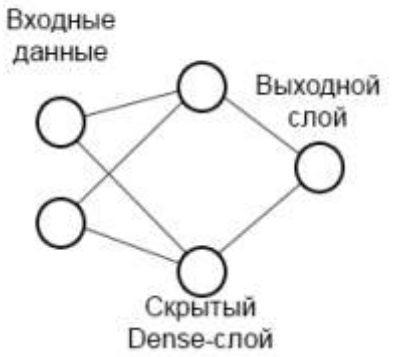
На примере выше указан способ добавления полносвязного слоя в модель с помощью метода .add(). Для создания слоя необходимо указать:

- количество нейронов в создаваемом слое (в примере – значение 32),
- количество входных параметров input_dim (данный параметр указывается только для первого слоя модели).

Создание простой нейронной сети

Существует еще ряд параметров, которые можно указать при создании полносвязного слоя (при их отсутствии будут установлены значения по умолчанию). Более подробную информацию можно посмотреть на сайте документации Keras (https://keras.io/layers/core/) и на сайте русскоязычной документации (https://ru-keras.com/?page_id=132). По мере прохождения курса мы рассмотрим и остальные параметры.

Знакомство со слоями Keras начнем со слоя Dense – полносвязный слой. Это слой, все нейроны которого связаны с нейронами предыдущего слоя (или со всеми входными параметрами, если слой Dense является первым слоем модели):



Пройдемся по всем шагам создания подобной модели.

1. Создаем пустой каркас для нашей сети:

```
model = Sequential()
```

2. Добавляем Dense-слой из двух нейронов:

```
model.add(Dense(2, input_dim=2, use_bias=False))
```

Мы добавили в нашу модель Dense-слой. Указали, что слой состоит из двух нейронов, и указали, что модель принимает на вход 2 параметра (input_dim=2). Также здесь присутствует дополнительный параметр: use_bias=False. Этот параметр указывает на использование нейрона смещения (более подробно о нем можно посмотреть в материалах к следующему занятию).

3. Добавляем еще один Dense-слой:

```
model.add(Dense(1, use_bias=False))
```

Мы добавили еще один Dense-слой в нашу модель. Указали, что он состоит из одного нейрона. В данном случае уже нет необходимости указывать параметр input_dim (Keras автоматически пересчитает размерность данных). И также отключаем использование нейрона смещения.

Создание простой нейронной сети

Таким образом, в трех строчках кода мы создали требуемую модель. С помощью метода .summary() можно отобразить структуру нашей модели:

```
model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 2)                 4
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 2
=================================================================
Total params: 6
Trainable params: 6
Non-trainable params: 0
```

В первой колонке отображаются имена слоев созданной модели (dense и dense_1) и типы этих слоев (Dense – полносвязный слой). Во второй колонке отображается размерность данных на выходе слоя. В третьей – количество параметров (весов). Внизу – общее количество параметров (6), количество изменяемых параметров (6, для нашей модели эти значения совпадают) и количество неизменяемых параметров (0).

Весовые коэффициенты модели

Весовые коэффициенты модели или просто веса модели – основные числовые характеристики нейронной сети, которые изменяются в процессе обучения модели для достижения оптимального результата. При добавлении очередного слоя в модель методом .add() веса этого слоя инициализируются случайными сгенерированными значениями. Простыми словами, веса – это коэффициенты, на которые умножаются или входные данные, или выходные значения слоя.

Для созданной выше модели есть 6 весовых коэффициентов:

Весовые коэффициенты модели

В итоге работа данной модели сводится к следующему. У нас есть два входных параметра: x1 и x2, два скрытых нейрона: N1 и N2, один выходной нейрон Y и шесть весовых коэффициентов: w1,w2, w3, w4, w5, w6.

Y = N1*w5 + N2* w6

N1 = x1*w1 + x2*w2

N2 = x1*w3 + x2*w4

Keras предоставляет возможность как получить, так и установить веса модели. Получение весов осуществляется с помощью метода .get_weights():

```
weights = model.get_weights()
```

Установка весов методом .set_weights():

```
model.set_weights(new_weight)
```

Ручной подсчет результатов работы модели

Установим веса вручную и посчитаем выход сети. Вначале зададим 6 весовых коэффициентов (значения выбраны случайно).

```
w1 = 0.42
w2 = 0.15
w3 = -0.56
w4 = 0.83
w5 = 0.93
w6 = 0.02
```

Создадим список, в котором будем хранить наши веса:

```
new_weight = [np.array([[w1,w3],[w2,w4]]),
np.array([[w5],[w6]])]
```

Наша модель имеет два слоя (один скрытый и один выходной). Поэтому список весов должен состоять из двух элементов. В первом – веса скрытого слоя (4 веса: w1, w3 – веса первого нейрона, w2, w4 – веса второго нейрона), во втором – веса выходного слоя (2 веса: w5, w6). Установим нашей модели сформированные веса:

```
model.set_weights(new_weight)
```

Ручной подсчет результатов работы модели

Теперь зададим два значения, которые будут являться входными данными:

```
x1 = 7.2
x2 = -5.8
```

Посчитаем значения нейронов N1 и N2:

```
N1 = x1*w1+x2*w2=7.2*0.42+(-5.8)*0.15 = 3,024-0,87 = 2,154
N2 = x1*w3+x2*w4=7.2*(-0.56)+(-5.8)*0.83 = -4,032-4,814 = -8,846
```

Посчитаем значение выходного нейрона Y:

```
Y =N1*w5+N2*w6=2,154*0.93+(-8.846)*0.02=2,00322-0,17692 = 1,8263
```

Мы получили значение (1,8263), которое возвращает наша модель для входных данных: [7.2, – 5.8].

Keras позволяет получить выходное значение модели с помощью метода .predict(). Если выполнить этот метод для нашей модели и передать ему наши данные x1 и x2, то метод должен вернуть значение 1,8263.

```
x_train = np.expand_dims(np.array([x1, x2]), 0)
y = model.predict(x_train)
```

Мы сформировали массив x_train, состоящий из двух элементов x1 и x2, имеющий размерность (1,2). Размерность массива – важная деталь при формировании набора данных для передачи в модель (здесь 2 – количество входных параметров, 1 – количество примеров). Далее мы вызываем метод .predict() и передаем ему сформированный массив входных параметров, результат метода записываем в переменную y. Если вывести значение переменной y на экран, оно будет равно 1,8263.

Функция активации

Сейчас мы можем простыми словами ответить на вопрос: в чем заключается принцип работы нейрона? Он считает взвешенную сумму своих входных данных и передает значение на вход следующему слою нейронов.

Функция активации

Функция активации позволяет установить условия, которые будут определять: передавать выходное значение в следующий слой (активировать нейрон) или не передавать.

В созданной нами модели используется линейная функция активации (linear). Это самая простая функция активации, которая используется по умолчанию. Она никак не изменяет значение на выходе нейрона и просто передает его в следующий слой:

$$f(x) = x$$

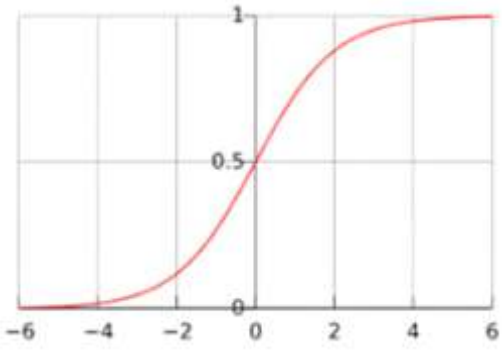
Значение на выходе нейрона может быть в диапазоне от $-\infty$ до $+\infty$. Это не самый оптимальный подход, а потому линейная функция активации используется крайне редко (большей частью в качестве активации выходного слоя модели).

Keras содержит целый набор возможных функций активации и помимо этого предоставляет возможность создавать свои собственные.

Не существует однозначных критериев, когда следует использовать ту или иную функцию активации. Подбор подходящей функции осуществляется в процессе обучения нейронной сети и сводится к перебору наиболее подходящих активационных функций.

Рассмотрим основные функции, которые наиболее часто применяются при проектировании архитектуры нейронных сетей.

Сигмоида (sigmoid)



Функция активации, преобразующая значение нейрона в значение из диапазона (0, 1). Формула сигмоиды:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

Простыми словами, значение нейрона подставляется в указанную формулу на место переменной x. Полученное значение передается дальше на вход нейронам следующего слоя.

Функция активации

Функция активации позволяет установить условия, которые будут

По графику сигмоиды можно сказать, что если значение нейрона меньше -4, то данный нейрон практически не активируется (значение сигмоиды в этих точках близко к 0), если значение больше 4, то нейрон однозначно определяет признак, за который отвечает (значение сигмоиды в этих точках близко к 1).

Функция активации указывается в качестве параметра activation при добавлении слоя методом .add().

```
model.add(Dense(2, activation='sigmoid'))
```

Таким образом указывается функция активации “сигмоида” для Dense-слоя. Это означает, что ко всем выходам данного слоя будем применена функция-сигмоида. По умолчанию параметр activation равен linear (линейная активация).

Создадим модель, аналогичную модели из первого раздела, но в качестве функции активации укажем 'sigmoid':

```
model_sigmoid = Sequential()
model_sigmoid.add(Dense(2, input_dim = 2,
activation='sigmoid', use_bias=False))
model_sigmoid.add(Dense(1, activation='sigmoid',
use_bias=False))
```

Как можно убедиться, код практически тот же самый, за исключением параметра activation. По сути, добавление этого параметра просто изменит способ подсчета значений нейронов.

Если в исходной модели значения Y, N1 и N2 были равны:

```
Y = N1*w5 + N2* w6
N1 = x1*w1 + x2*w2
N2 = x1*w3 + x2*w4
```

То добавление функции активации изменит эти формулы на:

```
Y = f(x)=f(N1*w5 + N2* w6)
N1 = f(x)=f(x1*w1 + x2*w2)
N2 = f(x)=f(x1*w3 + x2*w4)
```

где f – установленная функция активации (в данном случае сигмоида).

Функция активации

В ноутбуке к занятию разбирается путь по ручному подсчету выхода созданной модели, использующей сигмоиду в качестве функцию активации. Как и для исходной модели значение, подсчитанное вручную, и значение, полученное с помощью метода .predict(),

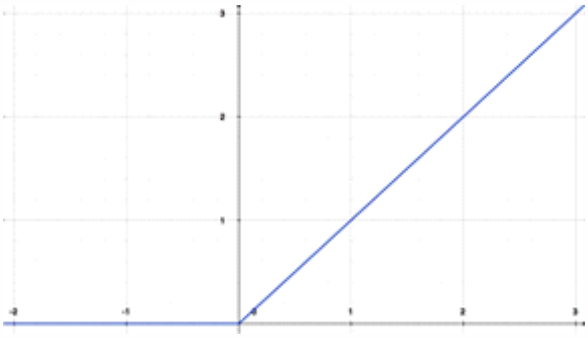
Линейный выпрямитель (relu)

Функция активации, преобразующая выходное значение нейрона в значение из диапазона [0, + ∞). Формула линейного выпрямителя:

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geqslant 0 \end{cases}$$

Простыми словами, все отрицательные значения функция преобразует в 0, а для всех положительных используется линейная активация, то есть значение не изменяется.

График функции:



Для задания этой функции параметр activation должен получить значение 'relu'. линейная активация, то есть значение не изменяется.

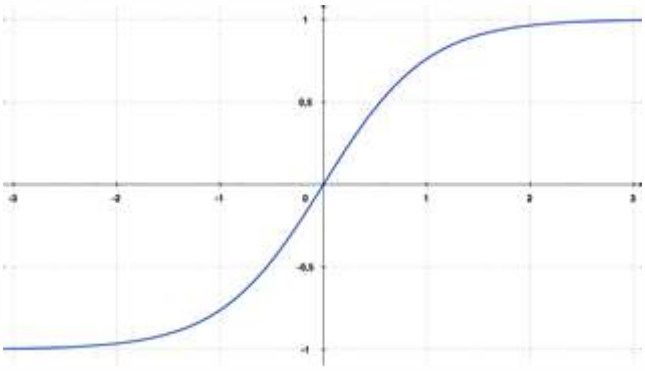
Гиперболический тангенс (tanh)

Функция активации, преобразующая выходное значение нейрона в значение из диапазона [−1, +1]. Формула гиперболического тангенса:

$$f(x) = \operatorname{th}(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

Функция активации

График функции:



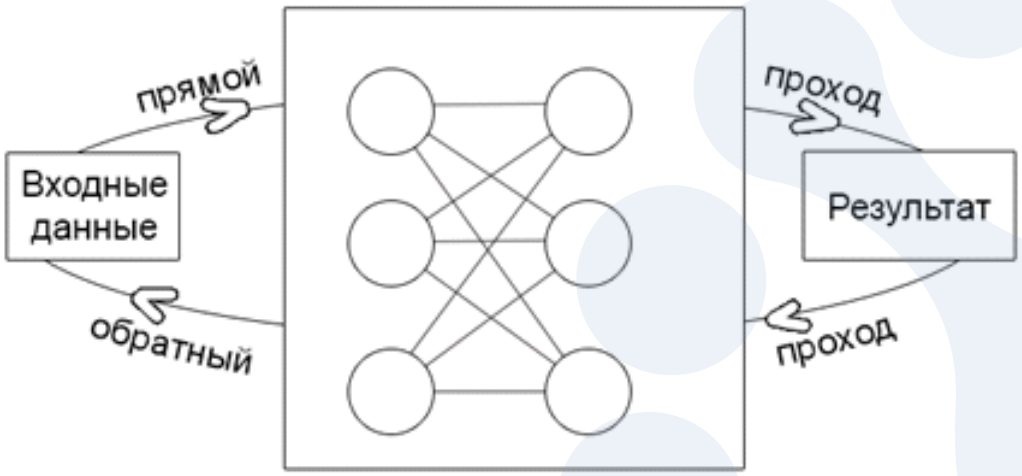
Для задания этой функции параметр activation должен получить значение 'tanh'.

Помимо рассмотренных Keras имеет следующий набор активационных функций: elu, softmax, selu, softplus, softsign, hard_sigmoid, exponential, LeakyReLU, PReLU, ELU, ThresholdedReLU, Softmax, ReLU.

Функция ошибки

Процесс обучения или настройки нейронной сети сводится к подбору весовых коэффициентов таким образом, чтобы результат, который получается на выходе модели, был максимально близок к образцу.

Весь процесс обучения нейронной сети можно разделить на два прохода: прямой и обратный. При прямом проходе рассчитывается результат модели (predict) и определяется значение ошибки между выходом модели и образцом, при обратном проходе веса модели пересчитываются таким образом, чтобы значение этой ошибки уменьшилось. Это процесс продолжается до тех пор, пока не будет достигнуто приемлемое значение ошибки.



Функция ошибки

Возьмем нашу исходную архитектуру модели и установим всем слоям функцию активации relu.

```
model_relu = Sequential()
model_relu.add(Dense(2, input_dim = 2, activation='relu', use_bias=False))
model_relu.add(Dense(1, activation='relu', use_bias=False))
```

С помощью метода .set_weights() установим веса нашей модели:

```
model_relu.set_weights(new_weight)
```

Теперь выберем произвольное значение, которое будет являться образцом (или значением, к которому должна стремиться наша модель).

```
y_real = 0.34
```

Посчитаем значение, которое возвращает наша модель:

```
y_relu = model_relu.predict(x_train)
```

Здесь x_train – сформированный ранее набор из двух значений. Полученное значение y_relu равно 2.0032198. В итоге мы получаем два значения: текущий результат модели (y_relu =2.0032198) и эталонный (y_real=0.34).

Для оценки этих двух значений используются различные функции ошибок. Keras содержит набор уже готовых функций, а также позволяет писать свои собственные функции ошибок.

Рассмотрим несколько основных функций ошибок, применяемых при разработке нейронных сетей.

Средняя абсолютная ошибка (mean_absolute_error (mae))

Формула ошибки: $$\mathbf{MAE} = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} = \frac{\sum_{i=1}^{n} |e_i|}{n}.$$

Средняя абсолютная ошибка представляет из себя модуль разности двух значений. Для нашего случае эта ошибка будет равна: |0.34 – 2.0032198| = 1,6632198

В том случае, если сеть обучается на наборе данных (а не на одном примере, как в нашем случае), для подсчета ошибки берется среднее значение по всем примерам.

Функция ошибки

Среднеквадратичная ошибка (mean_squared_error (mse))

Формула ошибки: $$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2.$$

Среднеквадратичная ошибка – это квадрат разности двух значений. Для нашего примера ошибка будет равна: (0.34 – 2.0032198) ^2 = 2,7663

В том случае, если сеть обучается на наборе данных (а не на одном примере, как в нашем случае), для подсчета ошибки берется среднее значение по всем примерам.

Бинарная кроссэнтропия (binary_crossentropy)

Формула ошибки:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Очевидно, что, чем меньше значение ошибки, тем лучше. Значение ошибки, близкое к 0 означает, что модель предсказывает значение, близкое к образцу.

Помимо рассмотренных, Keras содержит следующий набор функций ошибок: mean_absolute_percentage_error, hinge, mean_squared_logarithmic_error, squared_hinge, kullback_leibler_divergence, poisson, categorical_hinge, huber_loss, logcosh, sparse_categorical_crossentropy, cosine_proximity, is_categorical_crossentropy.

После подсчета ошибки начинается обратный проход модели, при котором оптимайзер будет перестраивать веса модели таким образом, чтобы значение ошибки уменьшилось.

Оптимизатор

Мы не будем подробно рассматривать принцип работы обратного распространения ошибки, опишем только основной подход.

Оптимизатор

Для изменения каждого отдельного весового коэффициента берется частная производная по этому коэффициенту от функции ошибки. В самом простом случае полученное значение, умноженное на параметр lerning_rate (шаг обучения), вычитается из текущего значения веса.

```
weight = weight - d * lr
```

Здесь weight – один из весов модели, d – частная производная по этому весу от функции ошибки, lr – шаг обучения.

Таким образом обновляются все веса модели и процесс обучения возвращается в фазу прямого прохода.

Существенный плюс Keras'a заключается в том, что все это скрыто внутри и не требует нашего вмешательства. Единственный параметр, которым мы можем управлять – это шаг обучения. Это довольно важный момент процесса обучения модели, поскольку слишком большой значение шага может привести к ситуация, когда ошибка будет бесконечно то убывать, то возрастать. В то же время слишком малое значение может увести ошибку в локальный минимум, и сеть не добьется максимума своих возможностей.

Механизм, который отвечает за процесс обновления весов в Keras называется оптимизатором. Как и в случае с ошибками и активационными функциями Keras предоставляет набор готовых оптимизаторов, а также возможность создавать свои собственные.

Градиентный спуск

Продемонстрируем на простом примере процесс градиентного спуска (обновления весов модели). Создадим модель с двумя входными параметрами и всего одним выходным слоем с одним нейроном.

```
model_d = Sequential()
model_d.add(Dense(1, input_dim=2, activation='linear', use_bias=False))
model_d.compile(optimizer=Adam(0.001), loss='mse')
```

Наша модель имеет всего два весовых коэффициента, что позволит нам построить график ошибки на трехмерной плоскости (поверхность от двух переменных).

Градиентный спуск

Зададим входные данные модели:

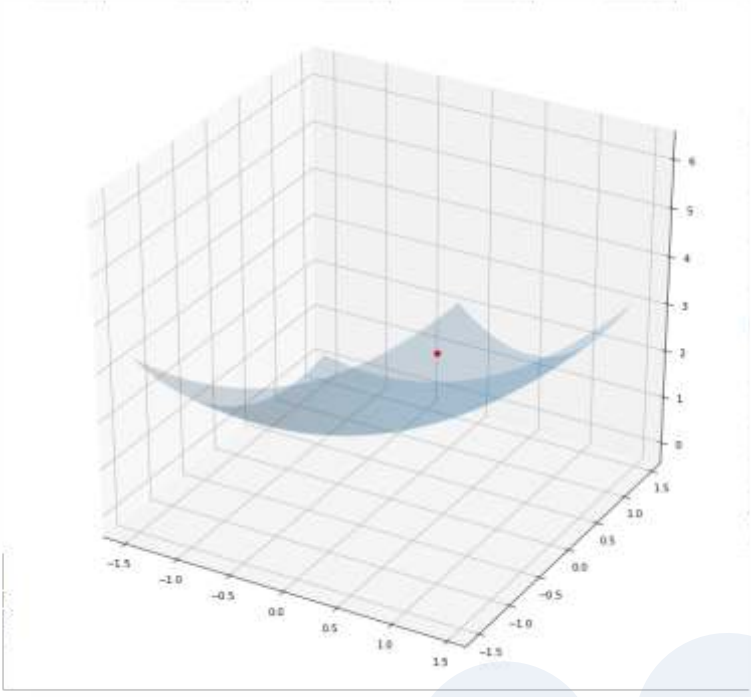
```
x_train = np.array([[1,0],[0,1]])
```

Мы будем использовать два набора данных: [1,0] и [0,1], следовательно, у нас должно быть два значения-образца:

```
y_train = np.array([1,-1])
```

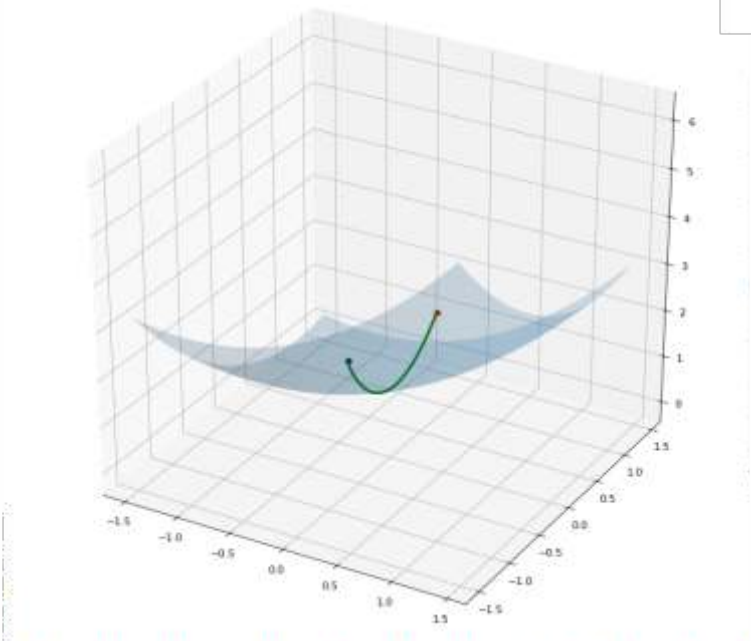
Заданные значения означают, что если в модель пришли значения 1 и 0, то на выходе должна быть единица. Если же пришли 0 и 1, то – минус единица.

Построим график ошибки для нашей модели (диапазон весов от -1.5 до +1.5)



Красная точка – это текущее значение ошибки при весах [1, +1].

Запустим процесс обучения модели, сохраним значение ошибки на каждом шаге обучения и отобразим на графике после завершения обучения.



Как видим, значение ошибки постепенно уменьшалось и “скатилось” в минимум графика. В финальной точке значение ошибки близко к 0, а веса соответствующие этому значению равны -1 и +1.

Градиентный спуск

Проверим эти веса для наших данных. Если на вход модели приходи 1 и 0 (модель должна выдать 1):

1 * 1 + (–1) * 0 = 1 – Верно!

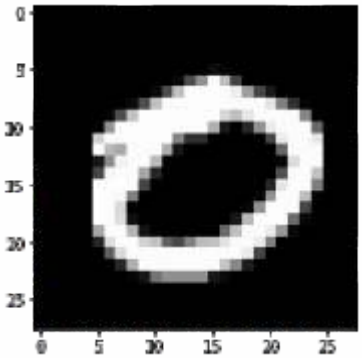
Если на вход приходит 0 и 1 (модель должна выдать –1):

1 * 0 + (–1) * 1 = –1 – Верно!

Распознавание рукописных цифр набора MNIST

Подготовка данных

Keras предоставляет готовые наборы данных для обучения создаваемых нейронных сетей. Одним из таких наборов является набор рукописных цифр MNIST. Он включает в себя 70 000 черно-белых картинок размером 28x28 пикселей с примерами рукописных цифр:



Наша задача – написать нейронную сеть, которая определяет, какая цифра изображена на картинке.

Для создания модели прежде всего подключаем необходимые модули библиотеки tensorflow.keras:

```
from tensorflow.keras.datasets import mnist #Библиотека с базой Mnist
from tensorflow.keras.models import Sequential # Подлючаем класс создания модели Sequential
from tensorflow.keras.layers import Dense # Подключаем класс Dense - полносвязный слой
```

Распознавание рукописных цифр набора MNIST

```
from tensorflow.keras.optimizers import Adam # Подключаем оптимизатор Adam
from tensorflow.keras import utils #Утилиты для to_categorical
from tensorflow.keras.preprocessing import image #Для отрисовки изображения
```

Keras предоставляет готовые наборы данных для обучения

Для загрузки набора данных MNIST используется следующая конструкция:

```
(x_train_org, y_train_org), (x_test_org, y_test_org) = mnist.load_data()
```

Метод .load_data() загрузит данные из набора и распределит их по 4ем переменным:

x_train_org – numpy-массив размерностью (60000, 28, 28). Здесь будут храниться картинки цифр, на которых будем обучать нашу модель.

y_train_org – numpy-массив размерностью (60000, 1). Здесь будут храниться метки ответов (от 0 до 9: какая цифра изображена на соответствующей картинке в массиве x_train_org)

x_test_org – numpy-массив размерностью (10000, 28, 28). Здесь будут храниться картинки цифр, на которых будем проверять качество работы нашей модели.

y_test_org – numpy-массив размерностью (10000, 1). Здесь будут храниться метки ответов (от 0 до 9: какая цифра изображена на соответствующей картинке в массиве x_test_org).

После получения данных необходимо сделать некоторые преобразования. Во-первых, изменить размерность входных данных. Сейчас это картинка размером 28x28 пикселей. Вытянем ее в один вектор размером 784 элемента (28 * 28 = 784):

```
x_train = x_train_org.reshape(60000, 784)
x_test = x_test_org.reshape(10000, 784)
```

Во-вторых, сделаем нормализацию данных. Нейронные сети показывают значительно лучшие результаты с данными, которые каким-либо образом нормализованы. Мы воспользуемся простым способом и разделим все значения на 255, тем самым приведя все значения к диапазону от 0 до 1 (255 – максимальное значение пикселя изображения).

Распознавание рукописных цифр набора MNIST

```
x_train = x_train.astype('float32')
x_train = x_train / 255
x_test = x_test.astype('float32')
x_test = x_test / 255
```

И, наконец, преобразуем наши ответы. Сейчас это просто цифры от 0 до 9. Преобразуем каждое значение в формат one-hot-encoding. Это вектор, состоящий из 0 и одной 1 (индекс 1 в векторе равен исходной метке). Например, цифра 8 будет представлена в виде: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0], цифра 0 – [1, 0, 0, 0, 0, 0, 0, 0, 0, 0] и т.п. Подобное преобразование выполняется с помощью метода to_categorical():

```
y_train = utils.to_categorical(y_train_org, 10)
y_test = utils.to_categorical(y_test_org, 10)
```

Метод .to_categorical() принимает два параметра: преобразуемый массив данных и количество элементов в векторе one-hot-encoding.

Создание модели

Создадим нейронную сеть, которая будет состоять из двух скрытых слоев (800 и 400 нейронов) и выходного слоя (10 нейронов).

```
model = Sequential()
model.add(Dense(800, input_dim=784, activation="relu"))
model.add(Dense(400, activation="relu")
model.add(Dense(10, activation="softmax"))
```

В качестве функции активации выходного слоя выбрана функция softmax. Эта функция преобразует все 10 значений выходного слоя таким образом, что сумма значений будет равна 1. Таким образом, мы получим 10 значений, соответствующих вероятностям, с которыми модель определила ту или иную цифру.

Остается скомпилировать нашу модель. В качестве оптимизатора выберем Adam. Ошибка categorical_crossentropy (это наиболее оптимальный вариант ошибки для задач классификации данных). Укажем также метрику для оценки точности работы нашей модели – accuracy.

```
model.compile(loss="categorical_crossentropy",
optimizer="adam", metrics=["accuracy"]) # Компилируем модель
```

Распознавание рукописных цифр набора MNIST

Создание модели

Создадим нейронную сеть, которая будет состоять из двух скрытых слоев (800 и 400 нейронов) и выходного слоя (10 нейронов).

```
model = Sequential()
model.add(Dense(800, input_dim=784, activation="relu"))
model.add(Dense(400, activation="relu")
model.add(Dense(10, activation="softmax"))
```

В качестве функции активации выходного слоя выбрана функция softmax. Эта функция преобразует все 10 значений выходного слоя таким образом, что сумма значений будет равна 1. Таким образом, мы получим 10 значений, соответствующих вероятностям, с которыми модель определила ту или иную цифру.

Остается скомпилировать нашу модель. В качестве оптимизатора выберем Adam. Ошибка categorical_crossentropy (это наиболее оптимальный вариант ошибки для задач классификации данных). Укажем также метрику для оценки точности работы нашей модели – accuracy.

```
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"]) # Компилируем модель
```

Обучение модели

Для обучения модели воспользуемся методом .fit().

```
model.fit(x_train, y_train, batch_size=128, epochs=15, verbose=1)
```

Данный метод принимает следующие параметры:

x_train – набор данных, на которых обучается наша модель;

y_train – набор правильных ответов (меток);

batch_size – размер партии данных для одного прохода по слоям нейронной сети (в данной задаче обучающий набор данных (x_train) состоит из 60000 элементов. За один проход нейронная сеть будет обрабатывать 128 примеров из этого набора данных);

epochs – количество итераций обучения (этот параметр указывает, сколько раз мы будем обучать нейронную сеть на всем обучающем наборе из 60000 элементов);

verbose – флаг отображения информации о процессе обучения (1 – отображать информацию, 0 – не отображать).

Распознавание рукописных цифр набора MNIST

Распознавание рукописных цифр

Давайте проверим работу нашей обученной нейронной сети. Для проверки нам потребуется набор данных x_test.

Возьмем произвольную картинку из набора x_test и добавим размерность:

```
x = x_test[1024]
x = np.expand_dims(x, axis=0)
```

Мы добавили размерность для того, чтобы данную картинку можно было передать на вход нейронной сети. Метод .expand_dims() добавит одну размерность в начало (axis=0). В результате размерность нашей картинки станет: (1, 784).

Вызываем метод .predict() нашей модели и передаем выбранную картинку:

```
prediction = model.predict(x) #Распознаём наш пример
```

В итоге в переменную prediction будет записан вектор из 10 значений, соответствующих вероятности определения каждой из возможных цифр.

```
print(prediction) #Выводим результат, это 10 цифр
[[1.40226162e-20 4.33944144e-17 4.43580097e-22 3.62223012e-12
  9.00776552e-20 1.00000000e+00 4.02172600e-17 1.09502567e-16
  6.57342670e-12 2.54268467e-10]]
```

Это означает, что модель определила: на картинке изображена цифра 0 с вероятностью: 1.40226162e-20, цифра 1 – 4.33944144e-17 и т.д. Можно увидеть, что сеть практически со 100% вероятностью определила на изображении цифру 5.

Мы можем воспользоваться методом np.argmax для того, чтобы получить индекс максимального элемента массива и соответственно получить ответ модели:

```
print(np.argmax(prediction))
```

5

Проверим правильный ответ:

```
print(y_test_org[1024])  #выводим правильный ответ для сравнения
```

5

Как видим, сеть сработала абсолютно точно.