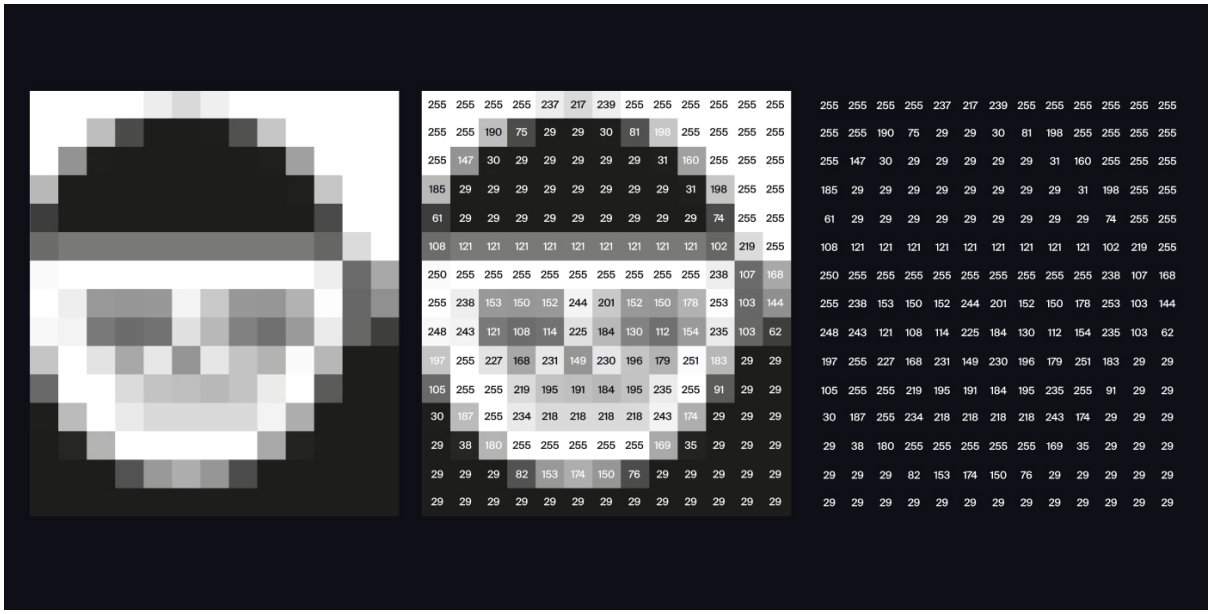
Конспект по теме «Полносвязные сети»

Задачи с нейронными сетями

Предположим, что *объект* — это фотография, а каждый пиксель — *признак* этого объекта.

Пример: на первой картинке чёрно-белое изображение низкого разрешения. На второй — это же изображение с указанием оттенков серого цвета от 0 (чёрный) до 255 (белый). Именно в таком диапазоне лежит значение яркости пикселя, то есть возможное значение признака. А на третьей — сами эти цифры.



Чтобы получить признаки, представим значения пикселей в виде вектора:

```
[255, 255, 255, 255, 237, 217, 239, 255, 255, 255, 255, 255, 255, 255, 255, 190, 75, 29, 29, 30, 81, 198, 255, 255, 255, 255, 2
```

Если у всех изображений датасета размер равен 1920×1080 пикселей, то каждое изображение описывается 2 073 600 признаками: 1920 умножаем на 1080. Когда признаков много, классические алгоритмы (например, градиентный бустинг) с обучением не справляются.

Разберём, что объединяет изображения и тексты:

1. Информация в них избыточна.
2. Соседние признаки связаны друг с другом.

Библиотека Keras

Познакомимся с открытой нейросетевой библиотекой **Keras**. По сути это интерфейс для работы с другой, более сложной библиотекой — **TensorFlow**. Есть ещё одна популярная нейросетевая библиотека **PyTorch**, с которой вы уже знакомы. Эта библиотека могла показаться лёгкой в применении, поскольку вы работали с готовой моделью. Однако для начинающих специалистов и *TensorFlow*, и *PyTorch* сложны.

Узнаем, как написать линейную регрессию в *Keras*. Линейная регрессия — это тоже нейронная сеть, но лишь с одним нейроном:

```
# подключаем Keras
from tensorflow import keras

# создаём модель
model = keras.models.Sequential()
# указываем, как устроена нейронная сеть
model.add(keras.layers.Dense(units=1, input_dim=features.shape[1]))
# указываем, как обучается нейронная сеть
model.compile(loss='mean_squared_error', optimizer='sgd')

# обучаем модель
model.fit(features, target)
```

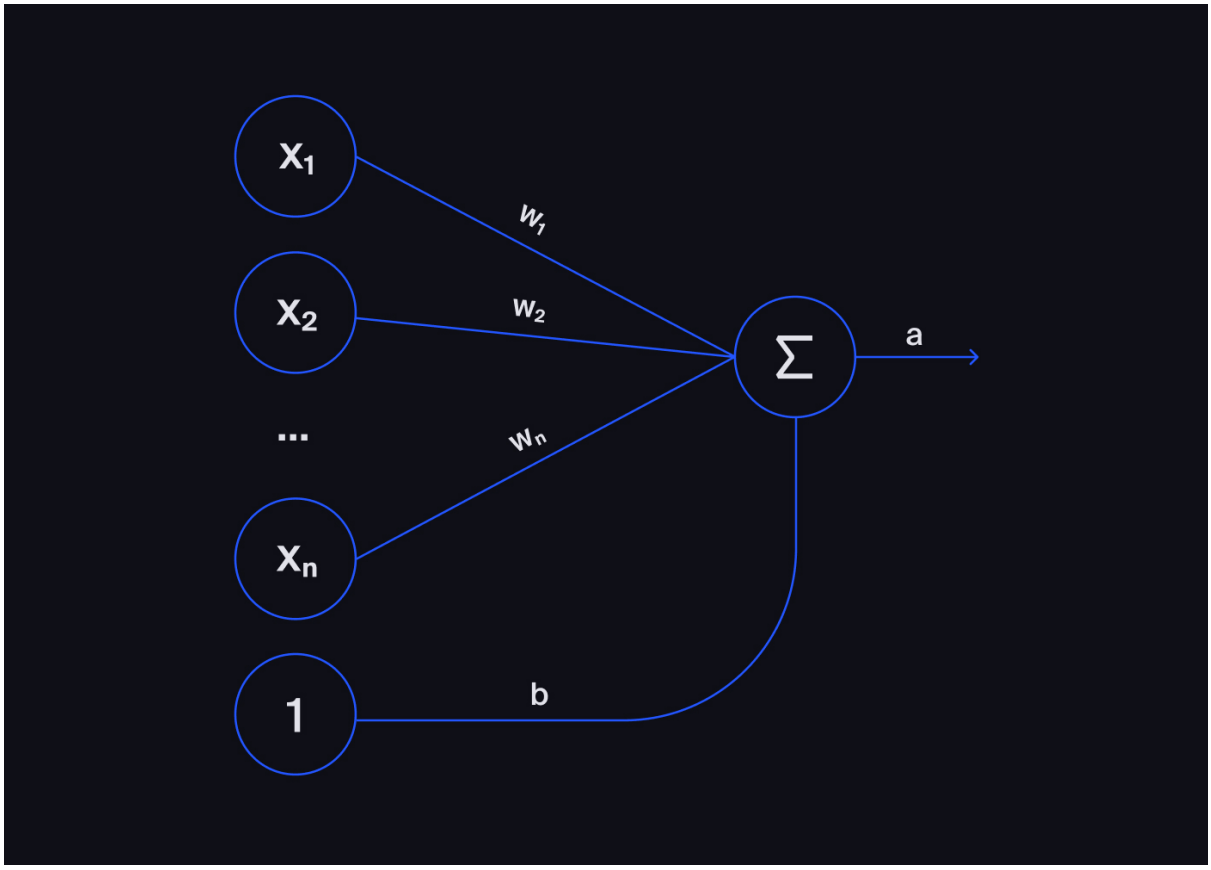
Разберём каждую строчку кода. Первая строка подключает *Keras* из библиотеки *tensorflow*. В тренажёре мы используем *TensorFlow* версии 2.1.0.

```
from tensorflow import keras
```

Следующая строка инициализирует модель, то есть нейронную сеть, которую мы построим. Модели зададим класс **Sequential**. Этот класс применяется для моделей, в которых слои идут последовательно. **Слой** (англ. *layer*) — набор нейронов с общим входом и выходом.

```
model = keras.models.Sequential()
```

Наша сеть будет состоять лишь из одного нейрона, или значения на одном выходе. В ней *n* входов, каждый умножается на свой вес. Например, x_1 умножается на w_1. Есть ещё один вход, который всегда равен единице. Его вес обозначается **b** (англ. *bias*, «смещение»). Именно из подбора весов *w* и *b* состоит процесс обучения нейронной сети. После того как все произведения значений входов на веса просуммированы, на выход подаётся ответ нейронной сети (*a*).

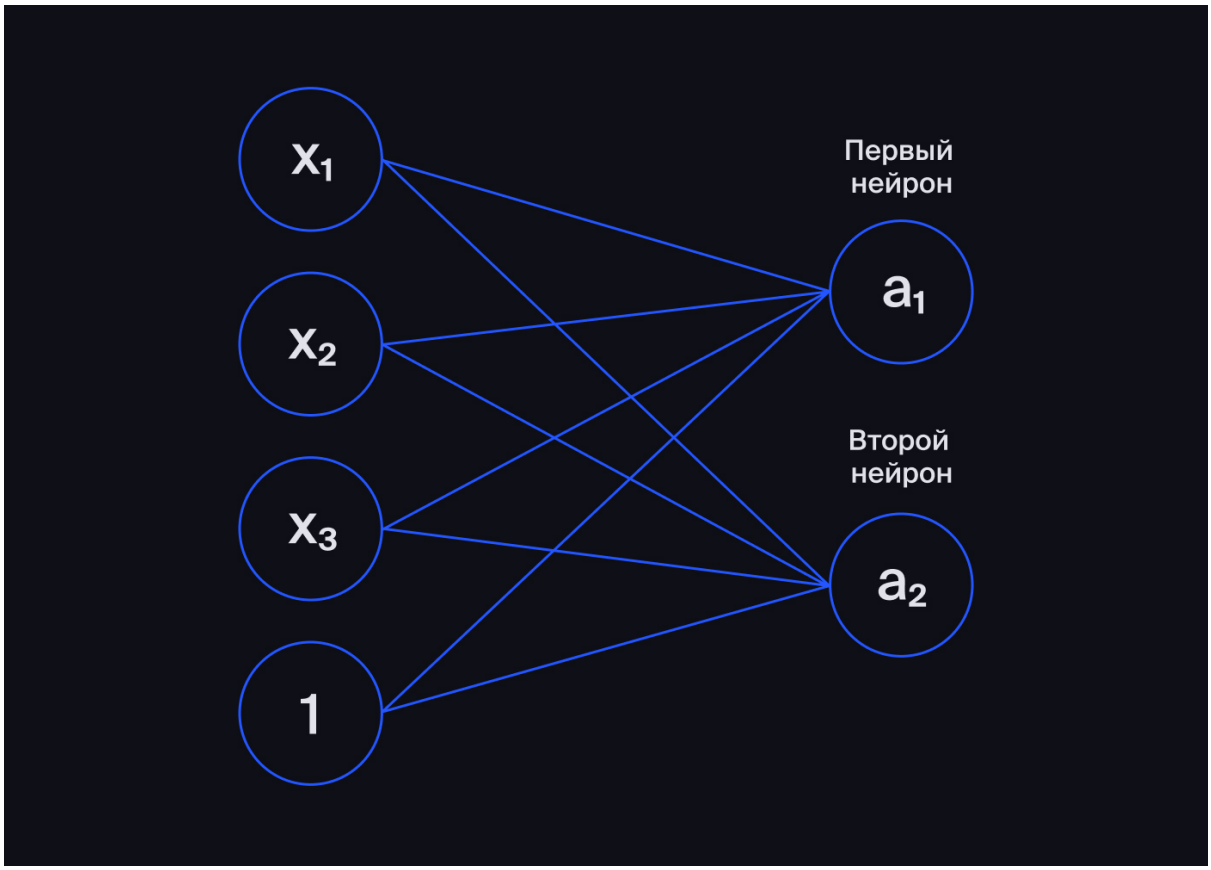


Команда **keras.layers.Dense()** создаёт один слой нейронов. В этом слое каждый вход соединён с каждым нейроном, или выходом. Параметр **units** задаёт количество нейронов в слое, а **input_dim** (от англ. *dimension*, «размерность») — количество входов в слое. Причём в этом параметре смещение не учитывается.

Чтобы создать слой для сети, напишем:

```
# количество входов возьмём из обучающей выборки
keras.layers.Dense(units=1, input_dim=features.shape[1])
```

Слои, в которых все входы соединены со всеми нейронами, называются **полносвязными слоями** (англ. *fully connected layers*). Полносвязный слой, задаваемый командой `keras.layers.Dense(units=2, input_dim=3)`, выглядит так:



Чтобы к модели добавить созданный слой, вызовем метод `model.add()`:

```
model.add(keras.layers.Dense(units=1, input_dim=features.shape[1]))
```

Разберём эту строчку:

```
model.compile(loss='mean_squared_error', optimizer='sgd')
```

Она готовит модель к обучению. После этой команды конструкцию сети уже нельзя будет изменить. В параметре *loss* укажем функцию потерь для задачи регрессии — MSE. В параметре `optimizer='sgd'` зададим метод градиентного спуска. Повторим: нейронные сети обучаются SGD.

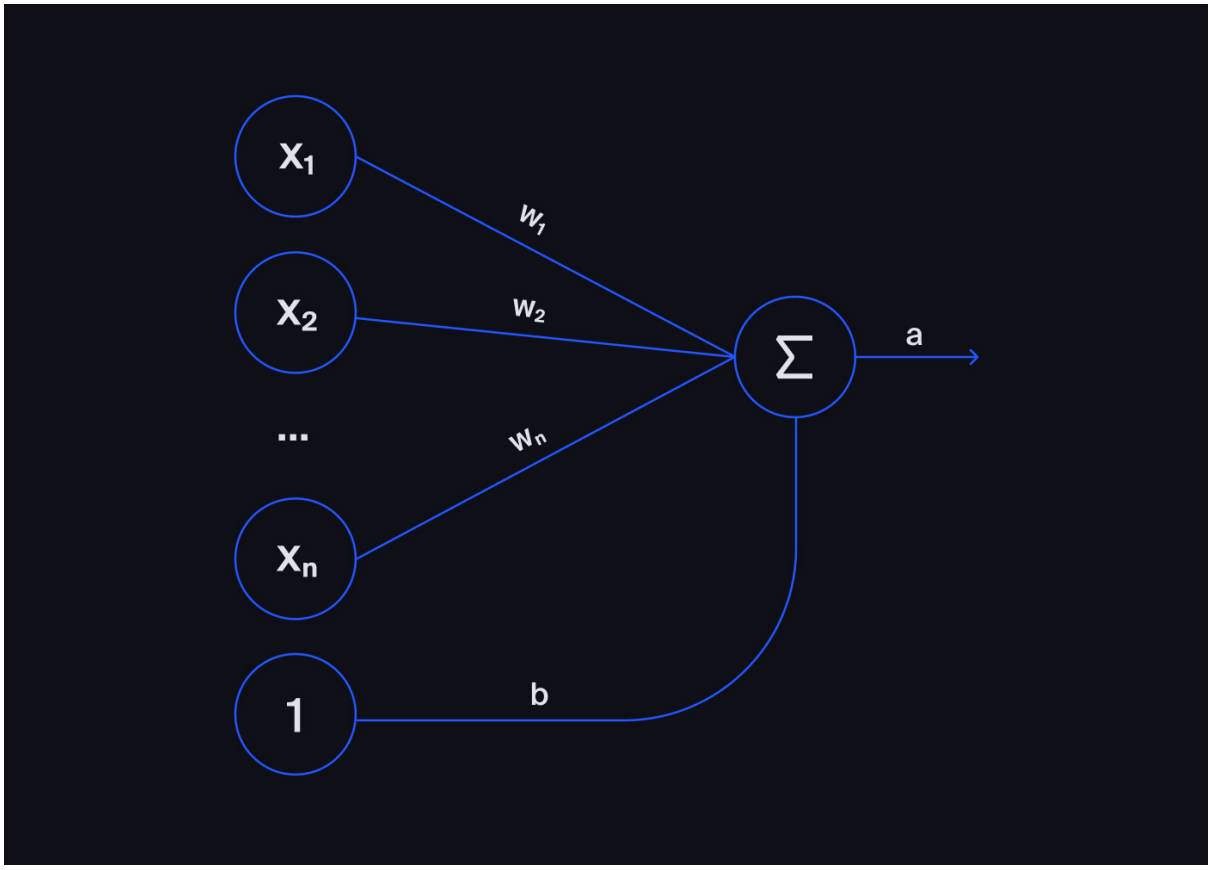
Теперь можно запустить обучение модели:

```
model.fit(features, target)
```

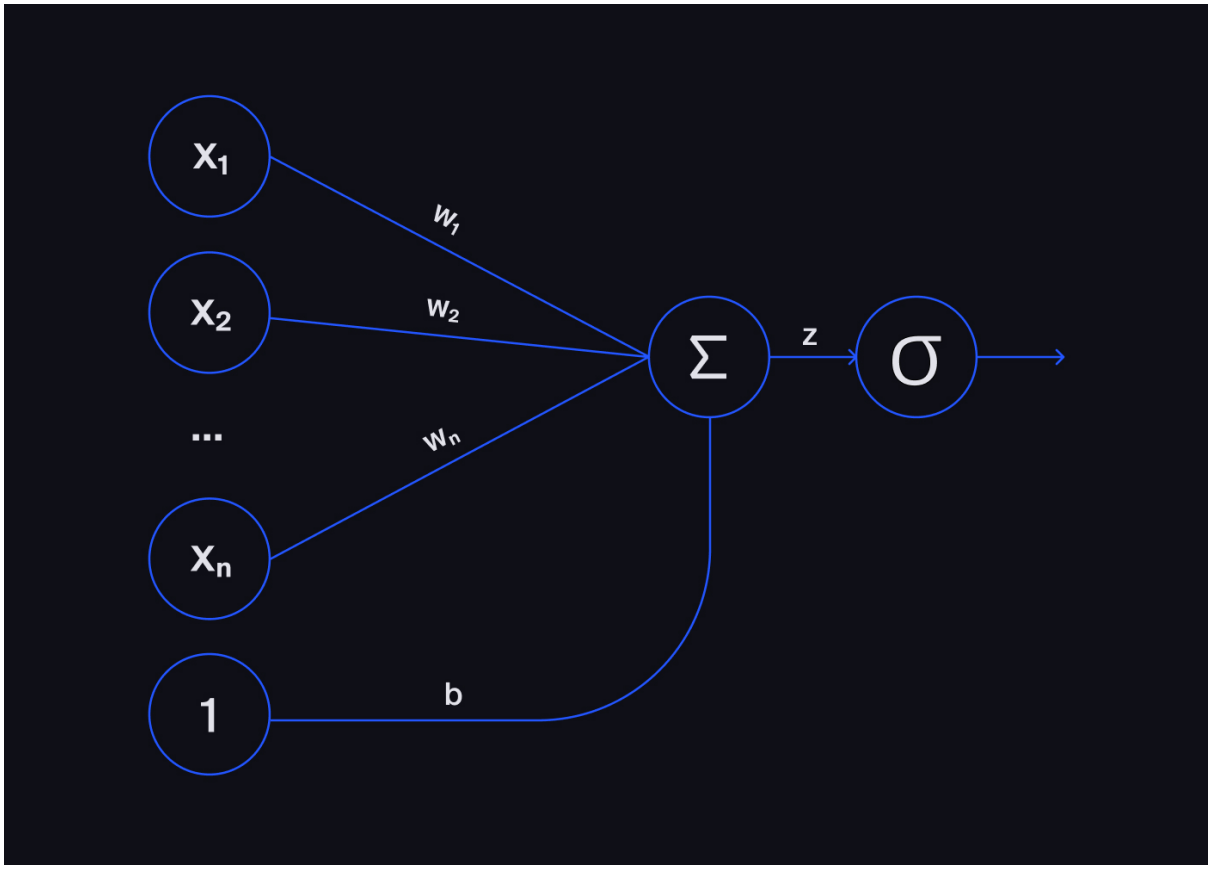
Логистическая регрессия

Линейная регрессия — это нейронная сеть с одним нейроном. Логистическая — тоже. Если классов у объектов всего два, то разница между линейной и логистической регрессиями почти незаметна. Нужно добавить всего один элемент.

Изобразим линейную регрессию:



Логистическая регрессия выглядит так:

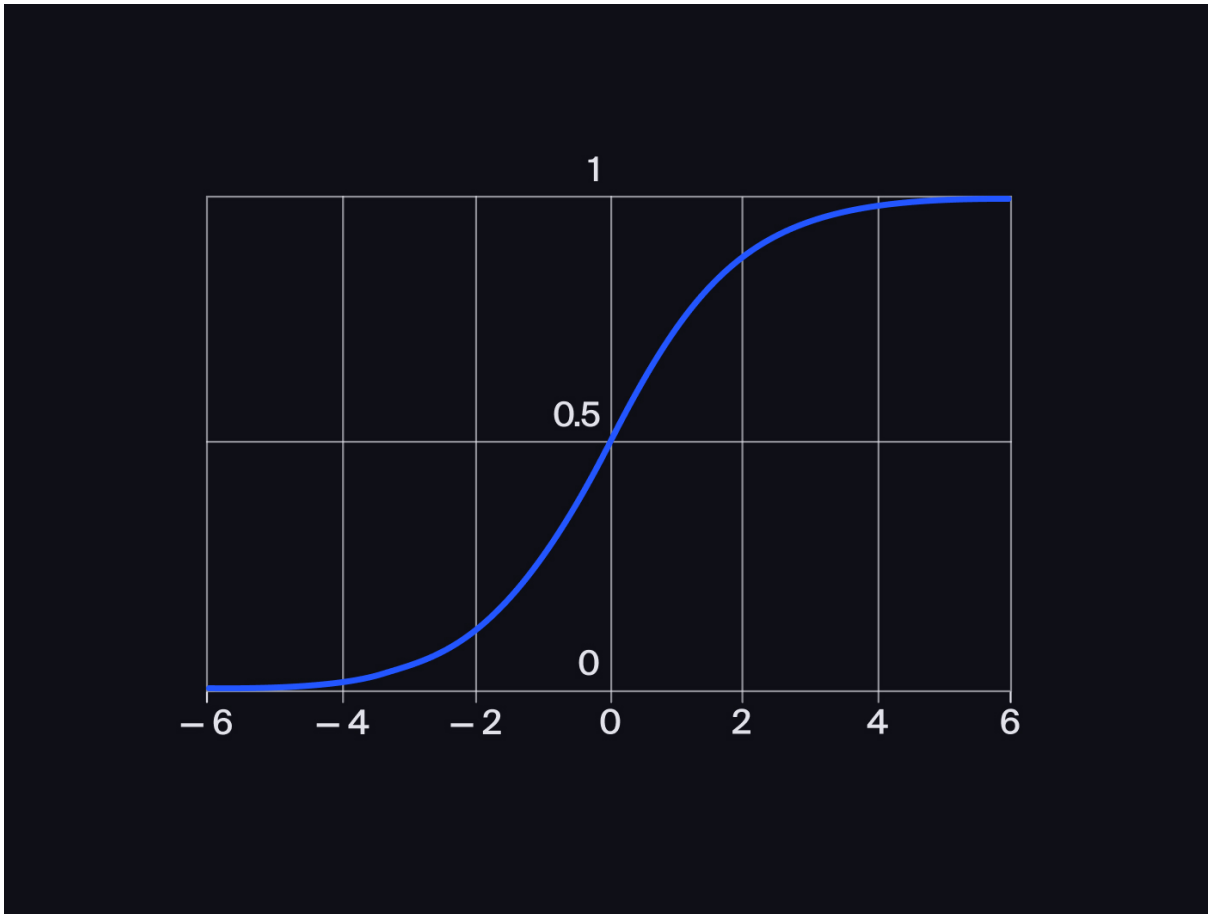


В схему добавилась **сигмоидная функция** (англ. *sigmoid function*), или знакомая вам функция активации нейрона. На вход она принимает любое действительное число, а возвращает число в диапазоне от 0 (активации нет) до 1 (активация есть).

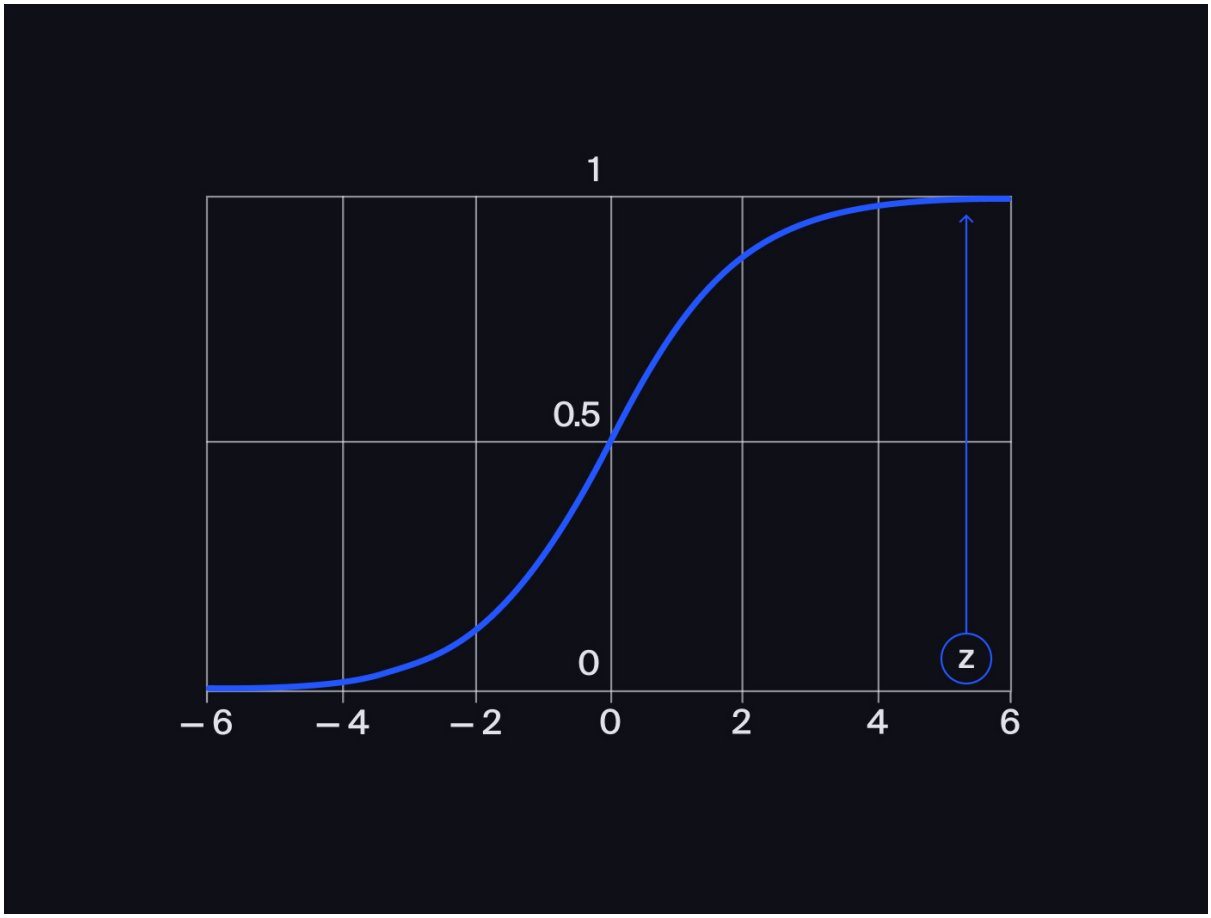
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

где *e* — число Эйлера; приблизительно равно 2.718281828.

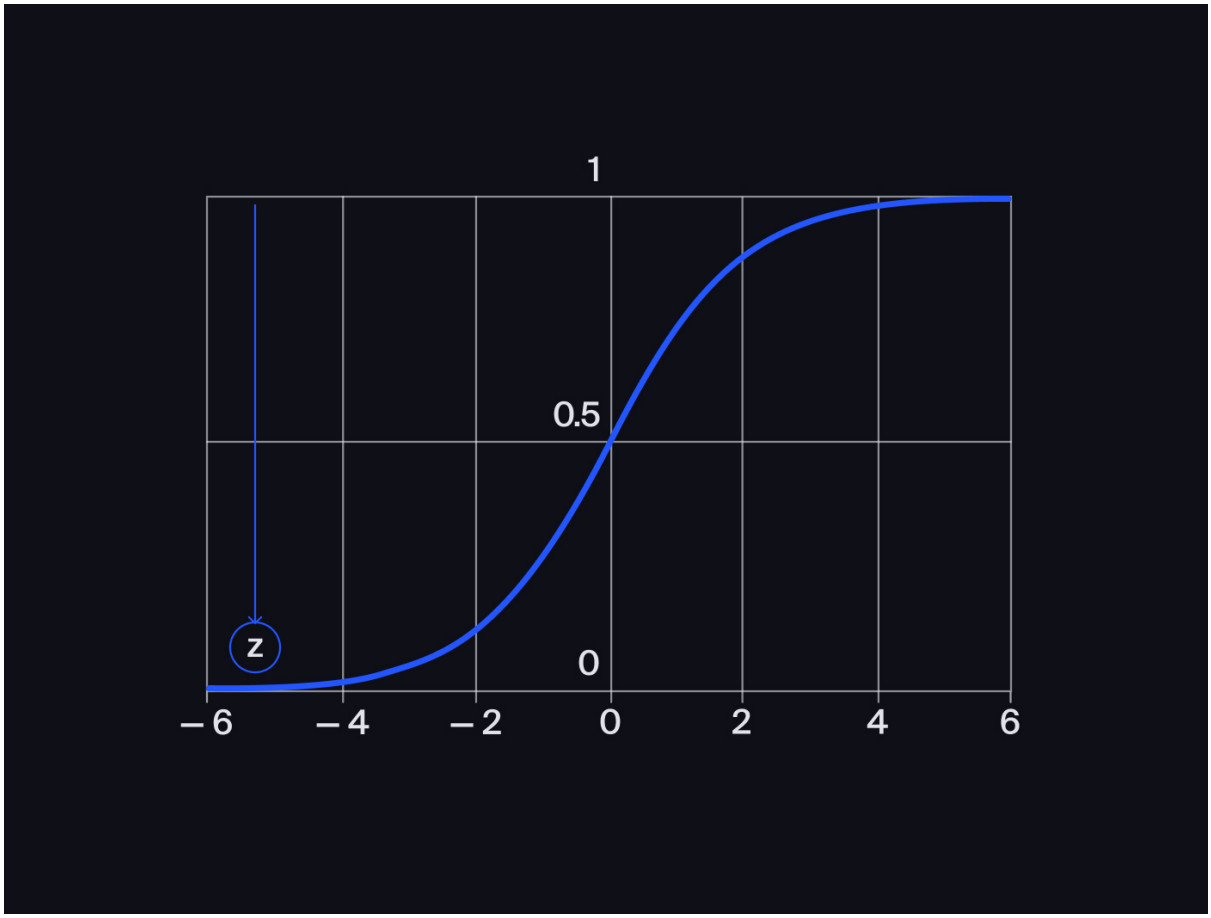
Это число в диапазоне от 0 до 1 можно трактовать как предсказание нейронной сети, к какому классу относится объект — отрицательному или положительному.



Если сумма произведений значений входов на веса (z) очень большая, то на выходе сигмоиды получим число, близкое к единице:



Если сумма, наоборот, — большое отрицательное число, то функция вернёт число, близкое к нулю:



Функция потерь меняется в зависимости от типа нейронной сети. Если в задаче регрессии применяли MSE, то для бинарной классификации подходит **Binary Cross-Entropy**, **BCE** (англ. «бинарная кросс-энтропия»). Метрику *accuracy* применять не можем: у неё нет производной, поэтому SGD работать не будет.

BCE вычисляется так:

$$BCE = -\log(p)$$

где *p* — вероятность правильного ответа. Основание логарифма не играет роли, потому что изменение основания — это умножение функции потерь на константу, и оно не меняет минимум.

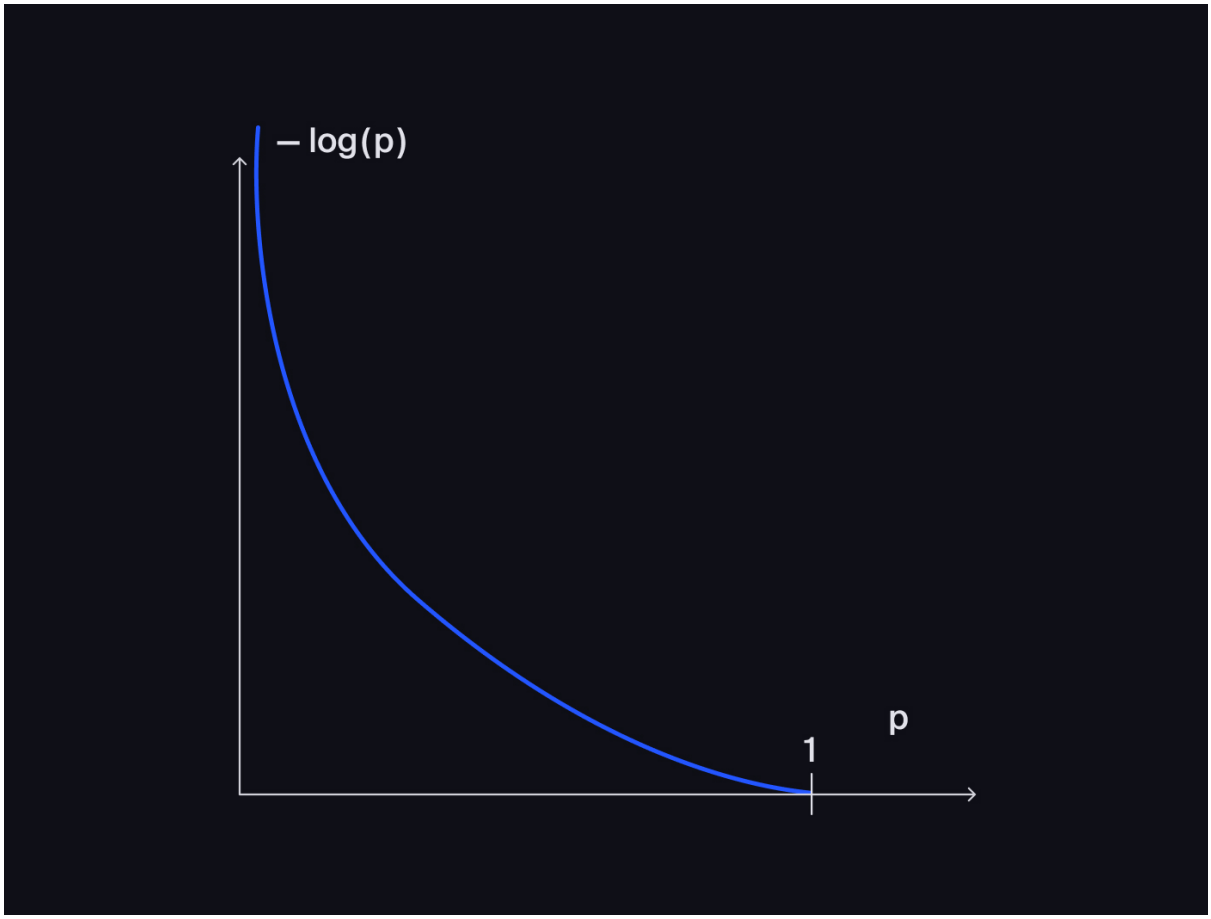
Если значение целевого признака — 1, то вероятность правильного ответа равна:

$$p = \sigma(z)$$

Если значение целевого признака — 0, то *p* равна:

$$p = (1 - \sigma(z))$$

Чтобы лучше разобраться с функцией *BCE*, рассмотрим её график:



Если вероятность правильного ответа *p* приблизительно равна единице, то *-log(p)* — положительное число, близкое к нулю. То есть ошибка маленькая. Когда вероятность правильного ответа *p≈0*, то *-log(p)* — большое положительное число. Ошибка тоже большая.

Логистическая регрессия в Keras

Чтобы получить логистическую регрессию, код линейной нужно поменять лишь в двух местах:

1. К полносвязному слою применить функцию активации:

```
keras.layers.Dense(units=1, input_dim=features_train.shape[1],
                   activation='sigmoid')
```

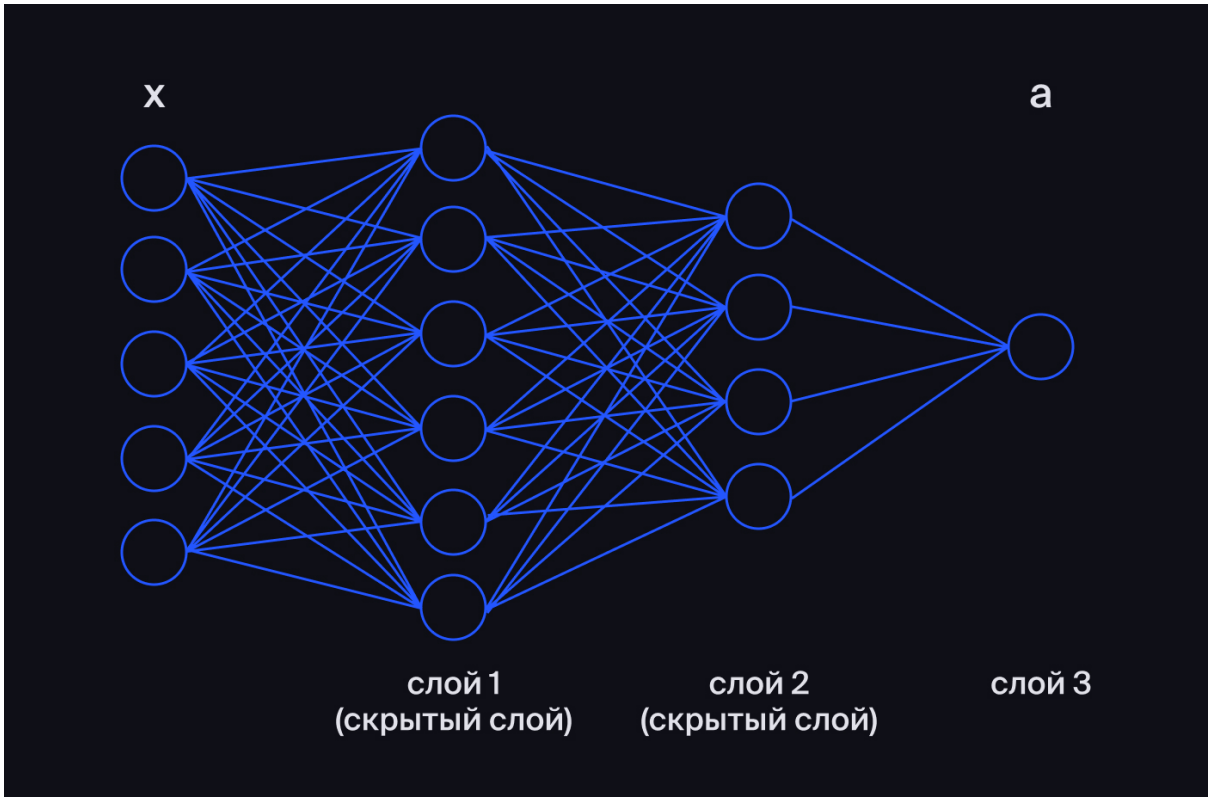
2. Поменять функцию потерь MSE на `binary_crossentropy`:

```
model.compile(loss='binary_crossentropy', optimizer='sgd')
```

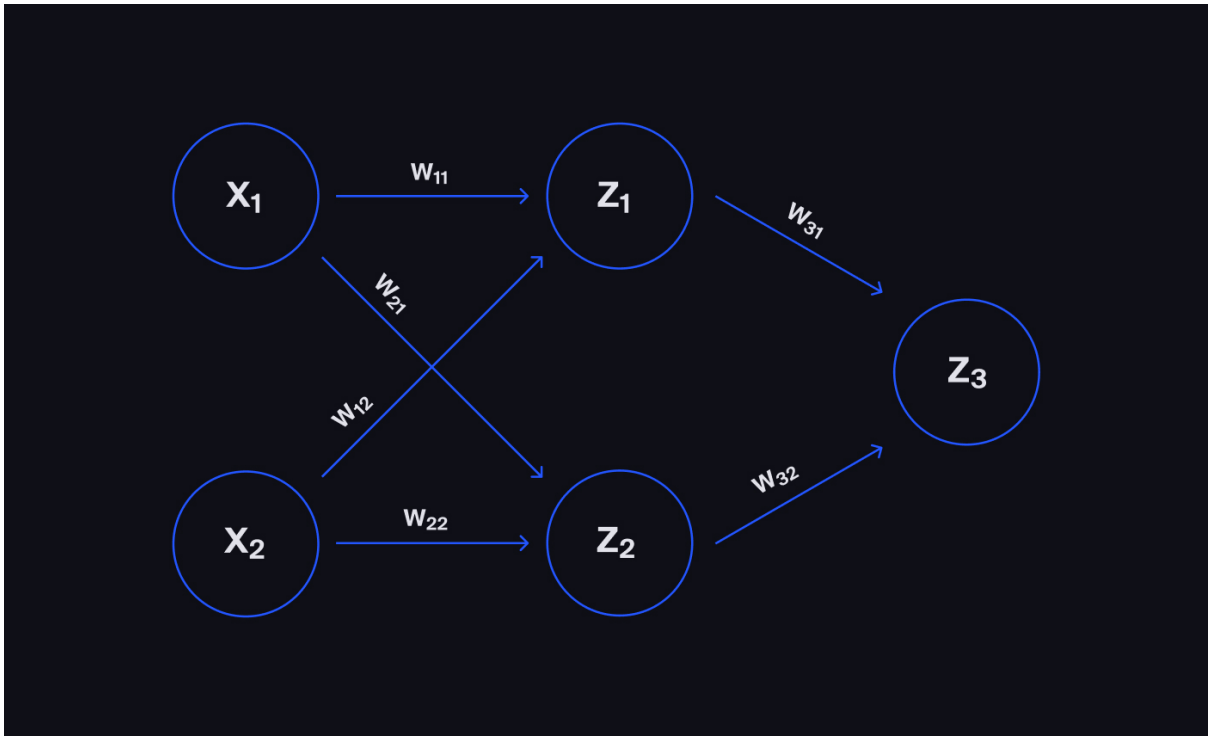
Полносвязные нейронные сети

Познакомимся с **полносвязными нейронными сетями** (англ. *fully connected neural networks*), где в каждом слое любой нейрон связан с каждым нейроном предыдущего слоя.

Вот пример полносвязной сети. **Скрытыми слоями** (англ. *hidden layers*) называются все слои, кроме входного и выходного.



Разберём устройство полносвязных сетей. В них после каждого нейрона, кроме последнего, есть функция активации. Интересно, почему? Рассмотрим на примере такой сети без веса *b*:



Чтобы получить z, просуммируем произведения значений входов на веса:

```
z1 = x1 * w11 + x2 * w12
z2 = x1 * w21 + x2 * w22
z3 = z1 * w31 + z2 * w32
```

Получим:

```
z3 = (x1 * w11 + x2 * w12) * w31 + (w21 * x1 + w22 * x2) * w32
```

Вынесем за скобки x_1 и x_2:

```
z3 = x1 * (w11 * w31 + w21 * w32) + x2 * (w12 * w31 + w22 * w32)
```

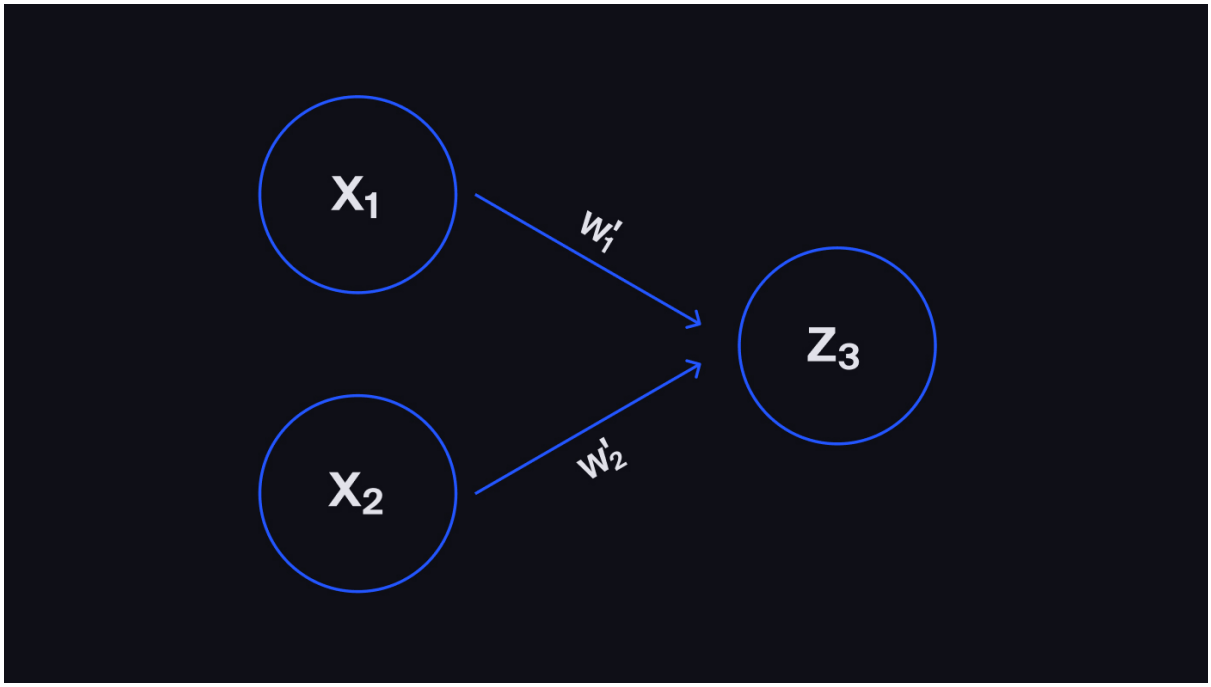
Для удобства введём новые обозначения w_1 и w_2:

```
w1' = w11 * w31 + w21 * w32
w2' = w12 * w31 + w22 * w32
```

Получим:

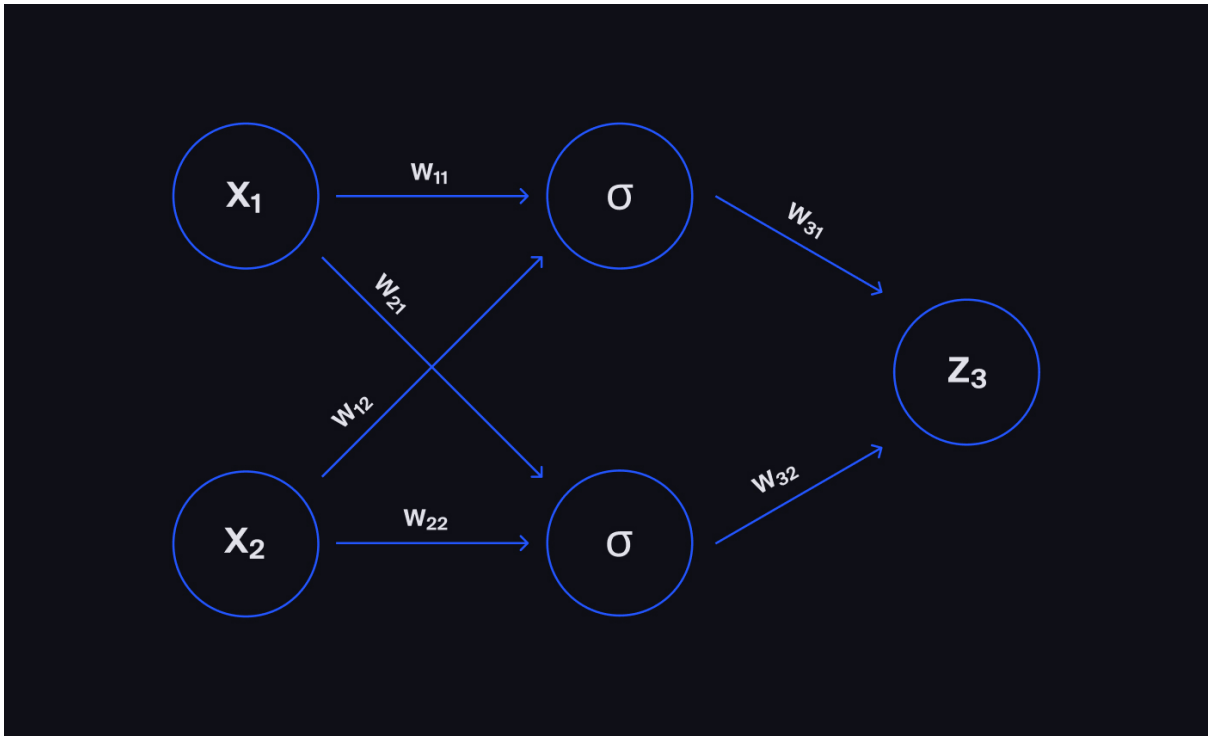
```
z3 = x1 * w1' + x2 * w2'
```

Проиллюстрируем эту формулу:



Ничего не напоминает? Многослойная сеть — это сеть с одним нейроном!

Чтобы это исправить, вернём сигмоиду и посмотрим, как поменяется сеть:



Запишем это формулой:

```
z1 = σ(x1 * w11 + x2 * w12)
z2 = σ(x1 * w21 + x2 * w22)
z3 = z1 * w31 + z2 * w32
```

Получим:

```
z3 =  σ(x1 * w11 + x2 * w12) * w31 + σ(w21 * x1 + w22 * x2) * w32
```

Из-за сигмоид вынести за скобки x_1 и x_2 нельзя, а значит, это уже не просто один нейрон. Сигмоида позволяет сделать сеть сложнее.

Как учатся нейронные сети

Как и линейная регрессия, многослойные сети обучаются градиентным спуском. Параметры — это веса у нейронов в каждом полносвязном слое. А обучение заключается в поиске таких параметров, при которых функция потерь минимальна.

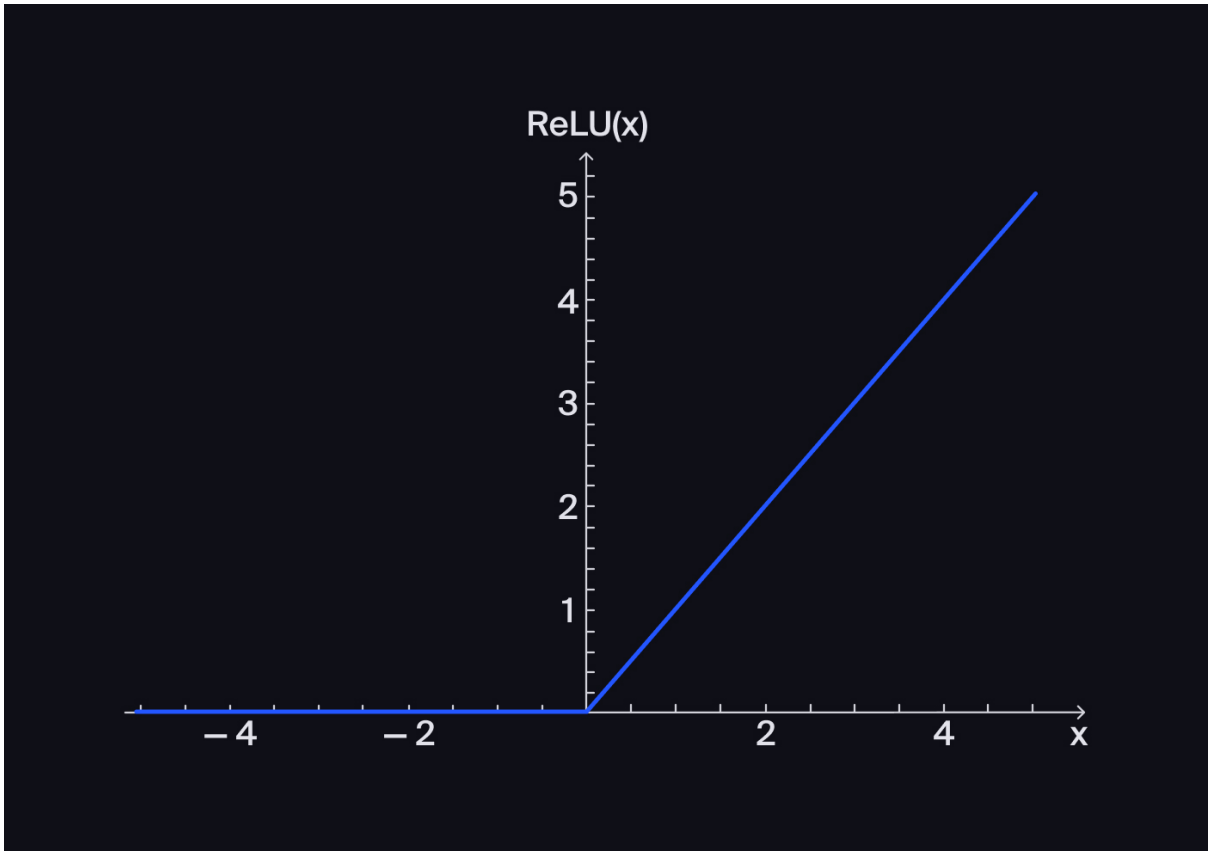
Интересно, как меняется нейронная сеть с добавлением нейронов и слоёв? Чтобы это узнать, решите несколько задач на сайте **TensorFlow Playground**. Площадка позволяет обучить небольшие нейронные сети на модельных данных с двумя признаками для задачи классификации.

В левой части интерфейса *TensorFlow Playground* можно выбрать набор данных. В центре изображено устройство сети. Справа выводится результат обучения. В верхней панели можно управлять обучением модели: менять значение эпохи, скорость обучения или функцию активации.

Посмотрите на такую сеть. В ней много слоёв и сигмоидная функция активации. Попробуйте обучить сеть. Она не обучится ни при какой величине шага.

Разберём почему. С увеличением количества слоёв обучение ухудшается. Чем больше слоёв в сети, тем меньше сигнала от входа доходит до выхода сети. Это называется **затуханием сигнала** (англ. *vanishing signal*). Причина затухания кроется в сигмоиде, преобразующей большие значения в маленькие по многу раз.

Чтобы избавиться от затухания сигнала, можно выбрать другую функцию активации. Например, **ReLU** (англ. *Rectified Linear Unit*, «выпрямленное линейное преобразование»). Изобразим её:



Вот формула ReLU:

$$ReLU(x) = \max(0, x)$$

ReLU приводит к нулю всё, что меньше 0, и пропускает положительные значения без изменений.

У этой сети поменяйте функцию активации с сигмоиды на *ReLU*. Убедитесь, что нейронная сеть теперь обучается корректно. Из-за смены функции активации разделяющая точки разных цветов поверхность выглядит как многоугольник. Вид фигуры зависит от исходной инициализации весов сети.

Полносвязные нейронные сети в Keras

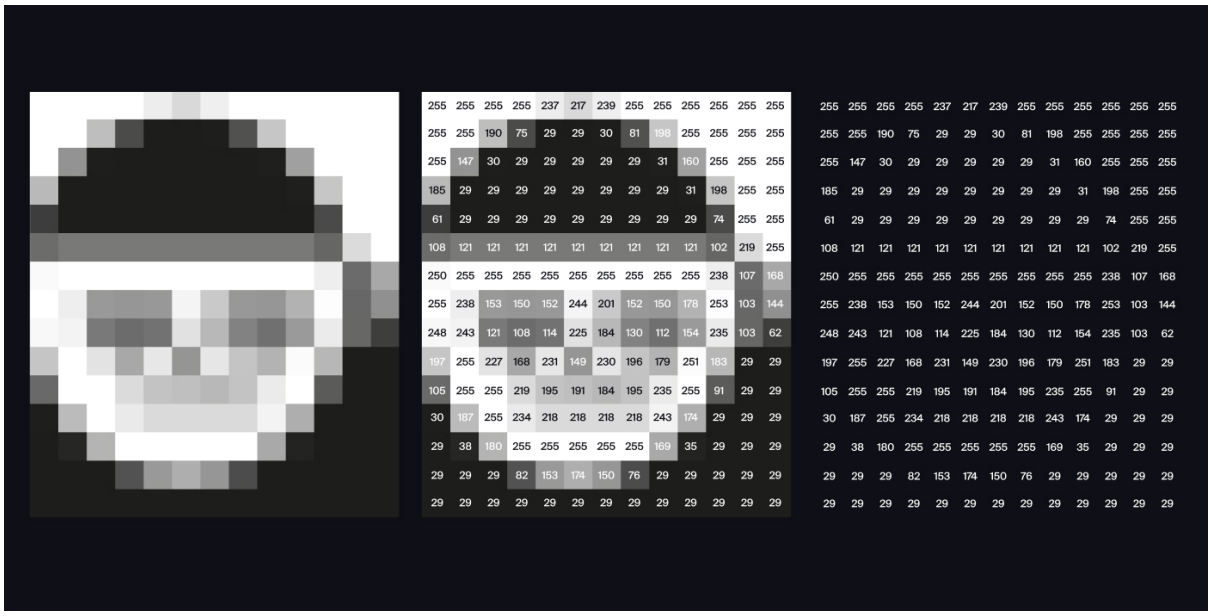
Полносвязные слои в *Keras* создаются вызовом метода *Dense()*. Чтобы построить многослойную полносвязную сеть, нужно несколько раз добавить полносвязный слой. Чем больше слоёв, тем сложнее модель. Пусть в скрытом слое будет 10 нейронов, а выходной слой состоит из одного нейрона:

```
model = keras.models.Sequential()
model.add(keras.layers.Dense(units=10, input_dim=features_train.shape[1],
                             activation='sigmoid'))
model.add(keras.layers.Dense(units=1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(features_train, target_train, epochs=10, verbose=2,
          validation_data=(features_valid, target_valid))
```

Работа с изображениями в Python

Вы уже знаете, что изображения — это набор чисел. Если изображение чёрно-белое, то в каждом пикселе хранится число от 0 (чёрный) до 255 (белый).



Откроем это изображение средствами библиотеки **PIL** (англ. *Python Imaging Library*). Затем будем работать с ним, как с *NumPy* массивом:

```
import numpy as np
from PIL import Image

image = Image.open('image.png')
image_array = np.array(image)
print(image_array)
```

```
[[255 255 255 255 237 217 239 255 255 255 255 255 255]
 [255 255 190  75  29  29  30  81 198 255 255 255 255]
 [255 147  30  29  29  29  29  29  31 160 255 255 255]
 [185  29  29  29  29  29  29  29  29  31 198 255 255]
 [ 61  29  29  29  29  29  29  29  29  29  74 255 255]
 [108 121 121 121 121 121 121 121 121 121 102 219 255]
 [250 255 255 255 255 255 255 255 255 255 238 107 168]
 [255 238 153 150 152 244 201 152 150 178 253 103 144]
 [248 243 121 108 114 225 184 130 112 154 235 103  62]
 [197 255 227 168 231 149 230 196 179 251 183  29  29]
 [105 255 255 219 195 191 184 195 235 255  91  29  29]
 [ 30 187 255 234 218 218 218 218 243 174  29  29  29]
 [ 29  38 180 255 255 255 255 255 169  35  29  29  29]
 [ 29  29  29  82 153 174 150  76  29  29  29  29  29]
 [ 29  29  29  29  29  29  29  29  29  29  29  29  29]]
```

Получили двумерный массив.

Построим изображение вызовом функции **plt.imshow()**:

```
plt.imshow(image_array)
```

Построить изображение в чёрно-белой цветовой гамме можно, указав параметр `cmap='gray'`. Чтобы добавить шкалу цвета, нужно вызвать метод `plt.colorbar()`:

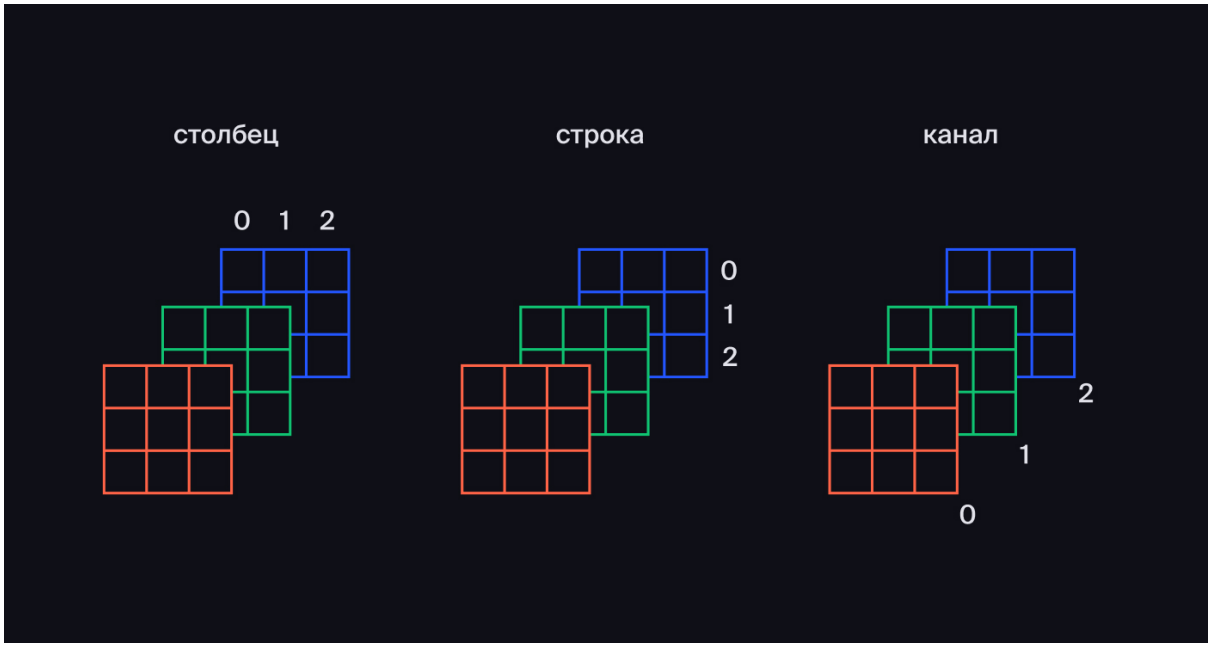
```
plt.imshow(image_array, cmap='gray')
plt.colorbar()
```

Для лучшего обучения нейронных сетей обычно на вход им передают изображения в диапазоне от 0 до 1. Чтобы привести масштаб [0, 255] к [0, 1], поделите все значения двумерного массива на 255:

```
image_array = image_array / 255.
```

Цветные изображения

Цветные, или **RGB-изображения**, состоят из трёх каналов: **красного** (англ. *red*), **зелёного** (англ. *green*) и **синего** (англ. *blue*). По сути такие изображения — это трёхмерные массивы, в ячейках которых могут быть целые числа от 0 до 255.



Трёхмерные массивы в *NumPy* устроены так же, как и двумерные.

Сравним, как они создаются:

```
np.array([[0, 255],
          [255, 0]])
```

```
np.array([[[0, 255, 0], [128, 0, 255]],
          [[12, 89, 0], [5,  89, 245]]])
```

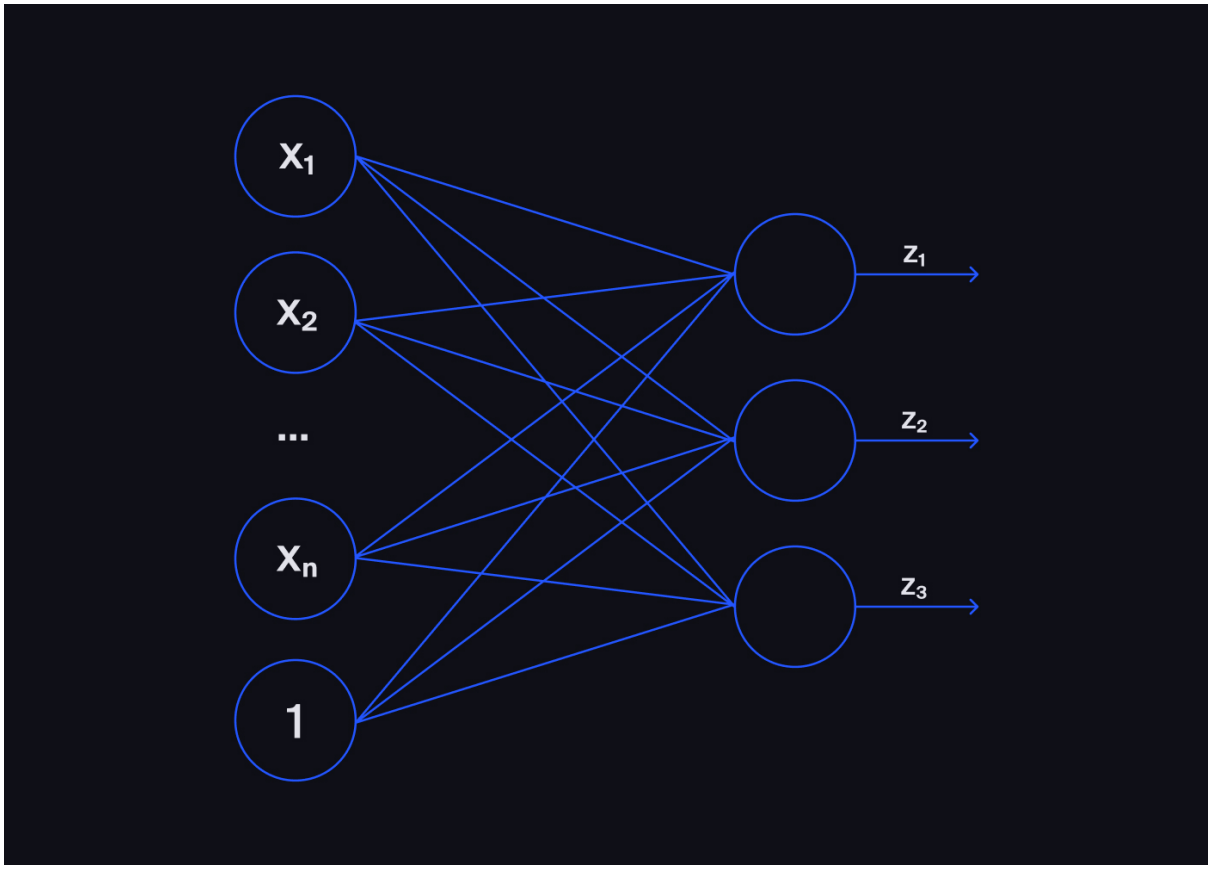
В трёхмерном массиве, полученном из изображения, всё так же: первая координата — это номер строки, вторая — номер столбца. Ещё добавляется третья координата — номер канала.

То есть трёхмерный массив — это всё тот же двумерный массив, аналогичный чёрно-белому изображению. Только в каждом пикселе этого массива хранятся три числа: яркость красного, зелёного и синего каналов.

Многоклассовая классификация

Разберём, что такое **многоклассовая классификация** (англ. *multi-class classification*) и как она устроена. В такой классификации объекты принадлежат не одному из двух классов, а одному из нескольких.

Допустим, у нас всего три класса. Изобразим логистическую регрессию в виде нейронной сети:



Получили полносвязную сеть, в выходном слое которой не один, а три нейрона. Каждый нейрон отвечает за свой класс. Если на выходе *z1* будет очень большое положительное значение, то нейронная сеть посчитает, что у объекта класс «1».

Как вычислить функцию потерь? Повторим бинарную кросс-энтропию:

$$BCE = -\log(p)$$

Если значение целевого признака — 1, то вероятность правильного ответа равна:

$$p = \sigma(z)$$

Если значение целевого признака — 0, то *p* равна:

$$p = (1 - \sigma(z))$$

В нашем примере классов три, но вычисление функции потерь не поменяется. Только называться она будет **CE** (англ. *cross-entropy*, «кросс-энтропия»):

$$CE = -\log(p)$$

где p — вероятность правильного класса, которую вернула нейросеть.

Откуда взять вероятность? Раньше её получали сигмоиды. Что если после каждого нейрона в выходном слое поставить сигмоиду?

```
p_1 (вероятность первого класса) = σ(z_1)
p_2 (вероятность второго класса) = σ(z_2)
p_3 (вероятность третьего класса) = σ(z_3)
```

Все вероятности варьируются от 0 до 1, но сумма этих вероятностей необязательно будет равна единице. Если предположить, что объект относится только к одному классу, надеемся получить такое равенство:

```
p_1 + p_2 + p3 = 1
```

Функция активации, которая нам подойдёт, называется **SoftMax** (англ. «мягкий максимум»). Она принимает несколько выходов сети и возвращает вероятности, сумма которых равна единице.

$$\mathrm{SoftMax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Вероятности будут вычисляться так:

```
p1 = SoftMax(z1) = e^z1 / (e^z1 + e^z2 + e^z3)
p2 = SoftMax(z2) = e^z2 / (e^z1 + e^z2 + e^z3)
p3 = SoftMax(z3) = e^z3 / (e^z1 + e^z2 + e^z3)
```

Теперь p_1 варьируется от 0 до 1.

Причём если z_1 значительно больше z_2 и z_3, то в формуле `e^z1 / (e^z1 + e^z2 + e^z3)` числитель примерно равен знаменателю, то есть $p_1 \approx 1$.

Если z_1 значительно меньше z_2 или z_3, то в формуле `e^z1 / (e^z1 + e^z2 + e^z3)` числитель сильно меньше знаменателя, то есть $p_1 \approx 0$.

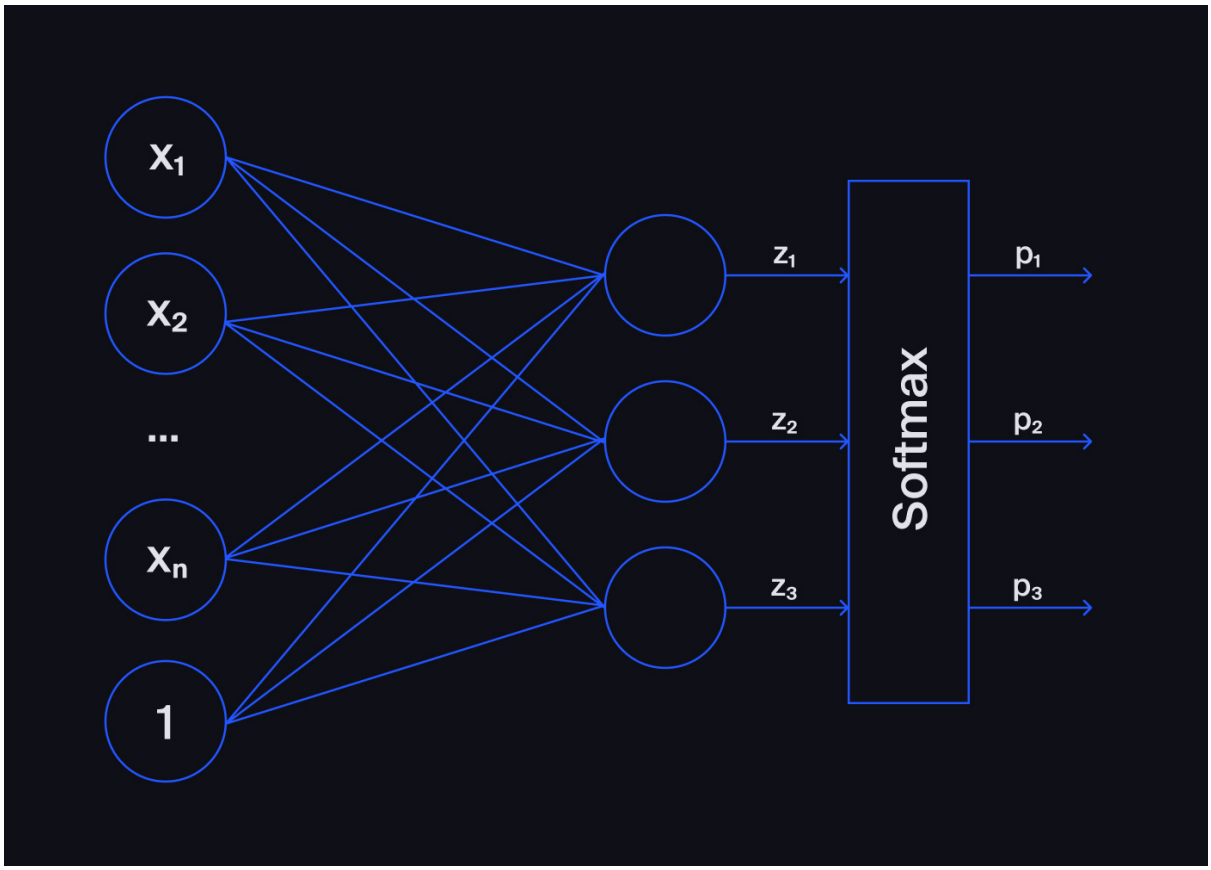
Теперь сумма вероятностей равна единице:

```
p1 + p2 + p3 = SoftMax(z1) + SoftMax(z2) + SoftMax(z3) =

= e^z1 / (e^z1 + e^z2 + e^z3) + e^z2 / (e^z1 + e^z2 + e^z3) + e^z3 / (e^z1 + e^z2 + e^z3) =

= (e^z1 + e^z2 + e^z3) / (e^z1 + e^z2 + e^z3) = 1
```

Схема нашей нейросети с функцией активации *SoftMax* выглядит так:



Почему изобразили *SoftMax* как блок, зависящий от всех выходов из сети? Чтобы получить каждую вероятность, нам действительно нужны все выходы.

Если классов будет больше трёх, нейронов в выходном слое будет столько же, сколько классов, а все их выходы передадим в *SoftMax*.

Вероятности из *SoftMax* на этапе обучения перейдут в кросс-энтропию, которая и посчитает ошибку. Функция потерь будет минимизирована методом градиентного спуска. Ему достаточно, чтобы у функции была производная по всем параметрам: весам и смещению нейронной сети.

Если в бинарной классификации инициализация последнего слоя выглядела так:

```
Dense(units=1, activation='sigmoid'))
```

В многоклассовой классификации она будет такой:

```
Dense(units=3, activation='softmax'))
```