01 - Последовательные нейронные сети на полносвязных слоях

Рис. 1: 11289dc1da918d44450a93680c493677.png

Нейроны полносвязного слоя взаимодействуют со всеми нейронами предыдущего слоя. Чаще всего полносвязные слои используются в финальной части нейронных сетей для классификации.

Создание модели нейронной сети

Это можно сделать всего в несколько строк с помощью библиотеки **Keras**. Подключите основу – класс создания последовательной модели **Sequential**.

Важно! Для ускорения обучения модели в Colab стоит переключиться на **GPU** (в верхнем меню Colab):

```
Среда выполнения --> Сменить среду выполнения --> Аппаратный ускоритель
```

```
from tensorflow.keras.models import Sequential
```

С помощью него создайте экземпляр вашей модели:

```
model = Sequential()
```

Это и есть ваша модель. Сейчас она пуста. Чтобы она что-то делала, нужно поместить в нее какой-нибудь механизм. Это не механизм в обычном смысле слова, потому что вы будете оперировать не предметами, а информацией. Механизм будет принимать на вход и выдавать на выходе какие-то данные.

Объекты

Так из чего же вы можете создать механизм? Для начала определитесь, сколько информации вы будете давать нейросети на вход. Один экземпляр такой информации называется **объектом**, который всегда состоит из чисел. Позже вы рассмотрите, как они устроены и какими бывают.

Допустим, вы решили, что каждый ваш входной объект состоит из **10** чисел. Настройте сеть на вход из **10** чисел:

```
from tensorflow.keras.layers import Dense
model.add(Dense(32, input_dim=10))
```

Внутри нейросеть состоит из слоев нейронов, и только что вы создали один из них. Этот первый слой называется ***Dense***-слоем (линейным или полносвязным слоем). Здесь же вы указали с помощью параметра **input_dim**, что ваша сеть принимает на вход последовательность из **10** чисел:

Полносвязный слой чаще других используется в нейросетях. Как механизм делится по слоям, так и некоторые слои тоже делятся на составляющие элементы. В разных слоях они имеют разные функции и названия. Например, в линейном слое этими элементами выступают **полносвязные нейроны**.

Их количество задается самым первым аргументом (в примере: **32**).

Нужно ли создавать выход сети? На самом деле нет. Результат, который выдает последний слой, и есть выход сети.

Значит, сеть готова к работе? Еще нет, потому что для работы нужно ее еще скомпилировать (собрать, подготовить к обучению) и обучить.

Для подготовки к обучению вам понадобятся еще две вещи – ***оптимизатор*** и ***функция потерь*** (или ***функция ошибки***). Они задаются с помощью метода модели `.compile()`:

```
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam'
)
```

При обучении нейронной сети обязательно указывают оптимизатор и функцию ошибки.

Оптимизаторы

Оптимизатор — это метод достижения лучших результатов, помощь в ускорении обучения.

Это алгоритм, используемый для незначительного изменения параметров, таких как веса и скорость обучения, чтобы модель работала правильно и быстро.

Импульсный оптимизатор Стохастический градиентный спуск - Stochasticy Gradient descent (SGD)

Берутся случайные объекты и подаются их в модель, затем и получаеется предсказание, считаем функцию потерь, обновляем веса, повторяем снова до тех пор пока функция ошибки не окажется в точке минимума. Есть три варианта:

1. Подаем **по одному** объекту набора данных. У каждой подачи своя функция ошибки, чуть отличная от других. Это может быть горбик на линии, немного другой изгиб графика и пр. Видно, что линия долго “блуждает”, прежде чем попасть в минимум.
2. BatchGD Подаем **весь набор** данных полностью (может требовать большие вычислительные мощности). При этом все ошибки усредняются, их графики становятся идентичными. Линия движения от текущей точки до минимума почти прямая, минимум отклонений от курса.
3. Mini-batchGD. Подаем **набор данных партиями, батчами** (их размер оптимизируется под имеющиеся вычислительные мощности). При этом все ошибки внутри батча усредняются, имеют идентичные графики, но между разными батчами есть небольшие отличия. Линия движения от текущей точки до минимума с небольшими отклонениями. Размер батча часто бывает степенью числа 2 (32,64,128), данные в батч выбираются случайным образом (при застревании в локальных минимумах некоторые шумные данные могут привести к выходу из этих минимумов). Такой подход объединяет в себе устойчивость однопакетной подачи и эффективность подачи всего пакета. **Это наиболее распространенная реализация** градиентного спуска, используемая в области глубокого обучения.

Основная проблема - попадание функции в локальные минимумы вместо нахождения глобального. Для ее решения используются модификации стохастического градиента, использующие скользящее среднее градиентов.

Оптимизатор импульса (Momentum) 1-я модификация SGD

Показывает меньшее количество колебаний, нежели SGD, помогает ускорить обучение. Учитывает прошлые градиенты, чтобы сгладить обновление. Основная идея в том, что мы ускоряемся (увеличиваем шаг), если движемся в одном направлении и замедляемся, если “путь” меняется.

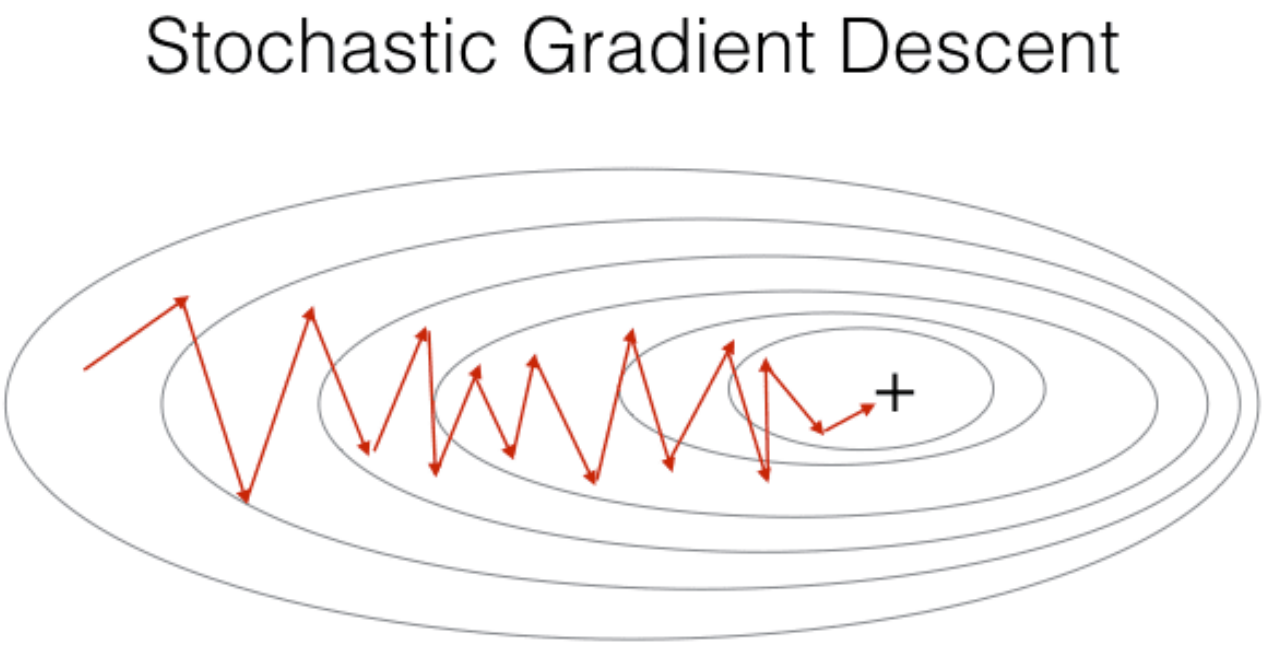


Рис. 2: e4b65b0e8aa49f1152c8a46c033f571c.png

При SGD с импульсом (или SGD with momentum) на каждой новой итерации оптимизации используется скользящее среднее градиента. Движение в направлении среднего прошлых градиентов добавляет в алгоритм оптимизации эффект импульса, что позволяет скорректировать направление очередного шага, относительно исторически доминирующего направления. Для этих целей достаточно использовать приближенное скользящее среднее и не хранить все предыдущие значения градиентов, для вычисления «честного среднего».

Оптимизатор импульса (Nesterov Momentum) 2-я модификация SGD

Nesterov accelerated gradient отличается от метода с импульсом, его особенностью является вычисление градиента при обновлении. Эта точка берётся впереди по направлению движения накопленного градиента. Отличие от обычного momentum на картинке ниже.

Красным вектором на первой части изображено направление градиента в текущей точке пространства параметров, такой градиент используется в стандартном SGD. На второй части красный вектор задает градиент сдвинутый на накопленное среднее. Зелеными векторами на обеих частях выделены импульсы, накопленные градиенты.

Пример использования оптимизатора **SGD**:

```
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
sgd = optimizers.SGD(learning_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='mean_squared_error', optimizer=sgd)
```

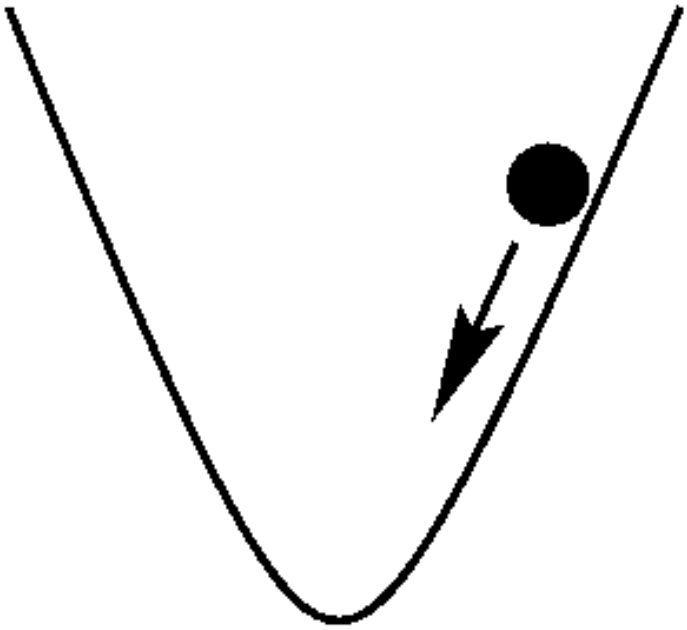
Документация:

- SGD в Керас https://keras.io/api/optimizers/sgd/

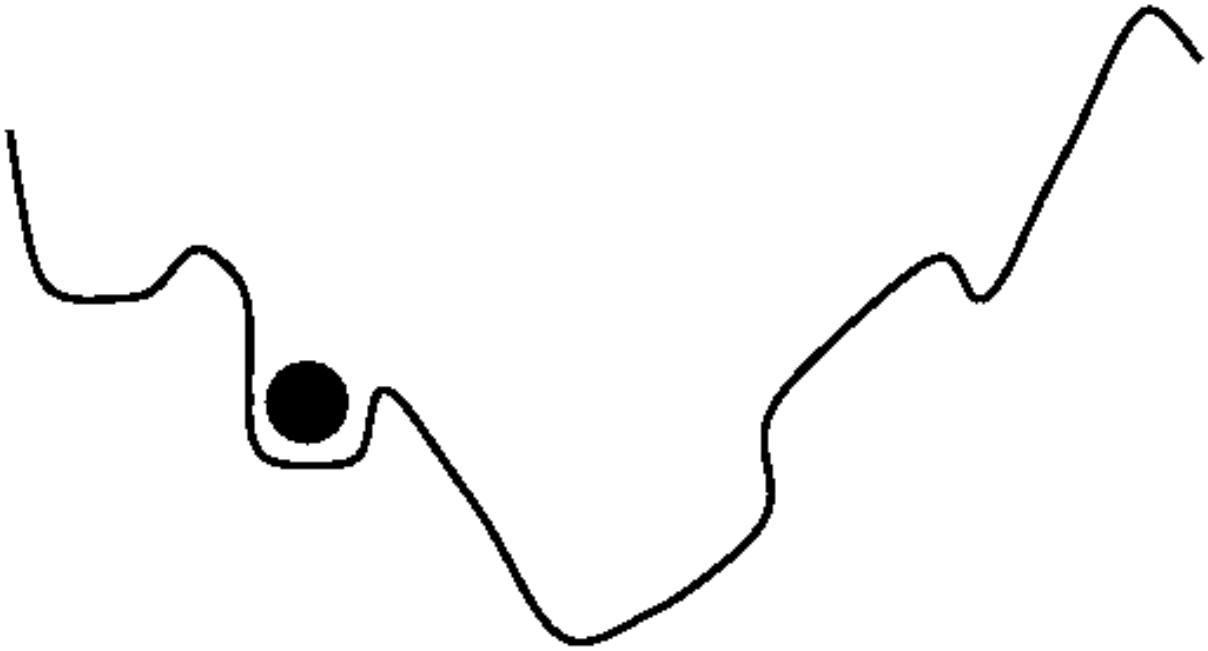
Документация по оптимизаторам на русском:

- https://ru-keras.com/optimizer/

Адаптивные оптимизаторы Цель адаптивных алгоритмов - отдельный learning rate для каждого из параметров. Чем чаще и сильнее меняется параметр, тем меньше его следующие изменения.



Гладкая выпуклая функция



Функция с множеством локальных минимумов (источник)

Рис. 3: 78d35d4cfb5c961f4d4b5162555798f6.png

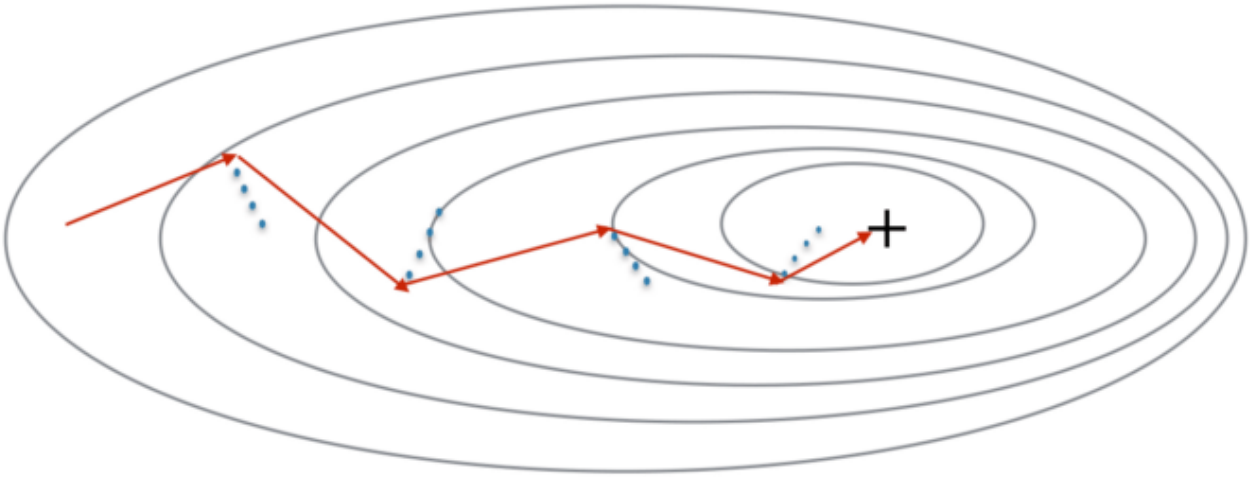


Рис. 4: f7cbaf9e46ab81a406ff875601d36919.png

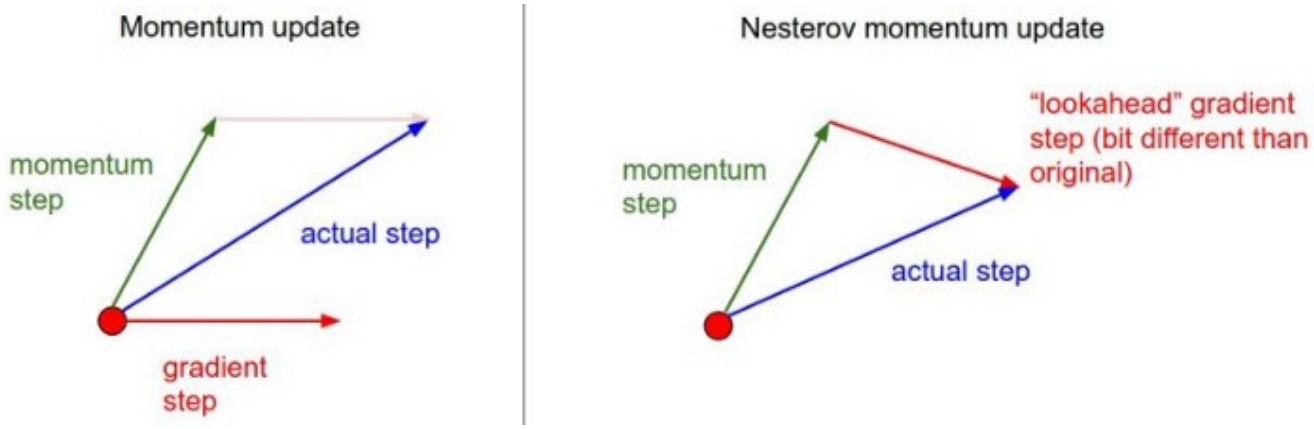


Рис. 5: eded36cfb01d572e9dc4b6bd29f0984b.png

Среднеквадратичное распространение корня Adagrad Например, какой-то из нейронов в каждой итерации немного изменяет свои значения(0,0.1,0,0.1,0.2) и есть другой, который колеблется от 0 до 10, к примеру. Эти скачки нужно сгладить. Для этого мы по каждому нейрону храним его историю.

Это дает нам возможность рассчитать размер следующий шаг как корень из произведения квадратов его двух предыдущих шагов.

Таким образом, чем больше были предыдущие шаги изменений для нейрона, тем меньше буду следующие, это помогает “успокоить” особо активные нейроны.

Но существует вероятность застрять в локальном минимуме, потому что скорость постоянно падает.

```
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
adagrad = optimizers.Adagrad(learning_rate=0.01)
model.compile(loss='mean_squared_error', optimizer=adagrad)
```

Документация:

- Adagrad в Keras https://keras.io/api/optimizers/adagrad/

Документация по оптимизаторам на русском:

- https://ru-keras.com/optimizer/

Экспоненциально затухающее среднее значение (RMSprop, root mean square) Существенным свойством RMSprop является то, что вы не ограничены только суммой прошлых градиентов, но вы более ограничены градиентами последних временных шагов. RMSprop вносит свой вклад в экспоненциально затухающее среднее значение прошлых «квадратичных градиентов». В RMSProp мы пытаемся уменьшить вертикальное движение, используя среднее значение.

Появляется новая зависимость размера шага от предыдущих. Уходит обязательность того, если 10 шагов назад был большой шаг, то сейчас должен быть меньший, и это позволяет выбраться из ямы. То есть он немного буксует в локальном минимуме, а потом снова начинает увеличиваться при необходимости, потому что зависимость от предыдущих шагов экспоненциальная.

Чаще всего используется в генеративных алгоритмах.

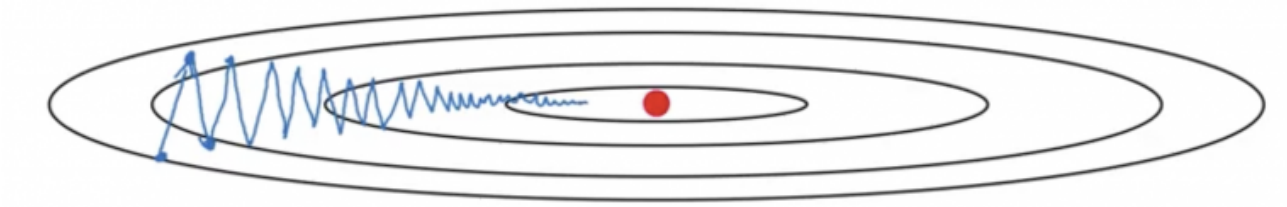


Рис. 6: 32f103a9184db35bff92c8af6458d061.png

Пример использования оптимизатора **RMSprop**:

```
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
rmsprop = optimizers.RMSprop(learning_rate=0.001, rho=0.9)
model.compile(loss='mean_squared_error', optimizer=rmsprop)
```

Документация:

- RMSprop в Keras https://keras.io/api/optimizers/rmsprop/

Документация по оптимизаторам на русском:

- https://ru-keras.com/optimizer/

Оптимизатор Adam

Adam — один из самых эффективных алгоритмов оптимизации в обучении нейронных сетей. Он сочетает в себе идеи RMSProp и оптимизатора импульса (momentum).

Чем больше движемся в одну сторону, тем больше шаг и адаптация каждого параметра.

Вместо того чтобы адаптировать скорость обучения параметров на основе среднего первого момента (среднего значения), как в RMSProp, Adam также использует среднее значение вторых моментов градиентов. В частности, алгоритм вычисляет экспоненциальное скользящее среднее градиента и квадратичный градиент

Пример использования оптимизатора **Adam**:

```
from tensorflow.keras import optimizers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
adam = optimizers.Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999,
 ↪  amsgrad=False)
model.compile(loss='mean_squared_error', optimizer=adam)
```

Документация: - Adam в Keras https://keras.io/api/optimizers/adam/

Документация по оптимизаторам на русском:

- https://ru-keras.com/optimizer/

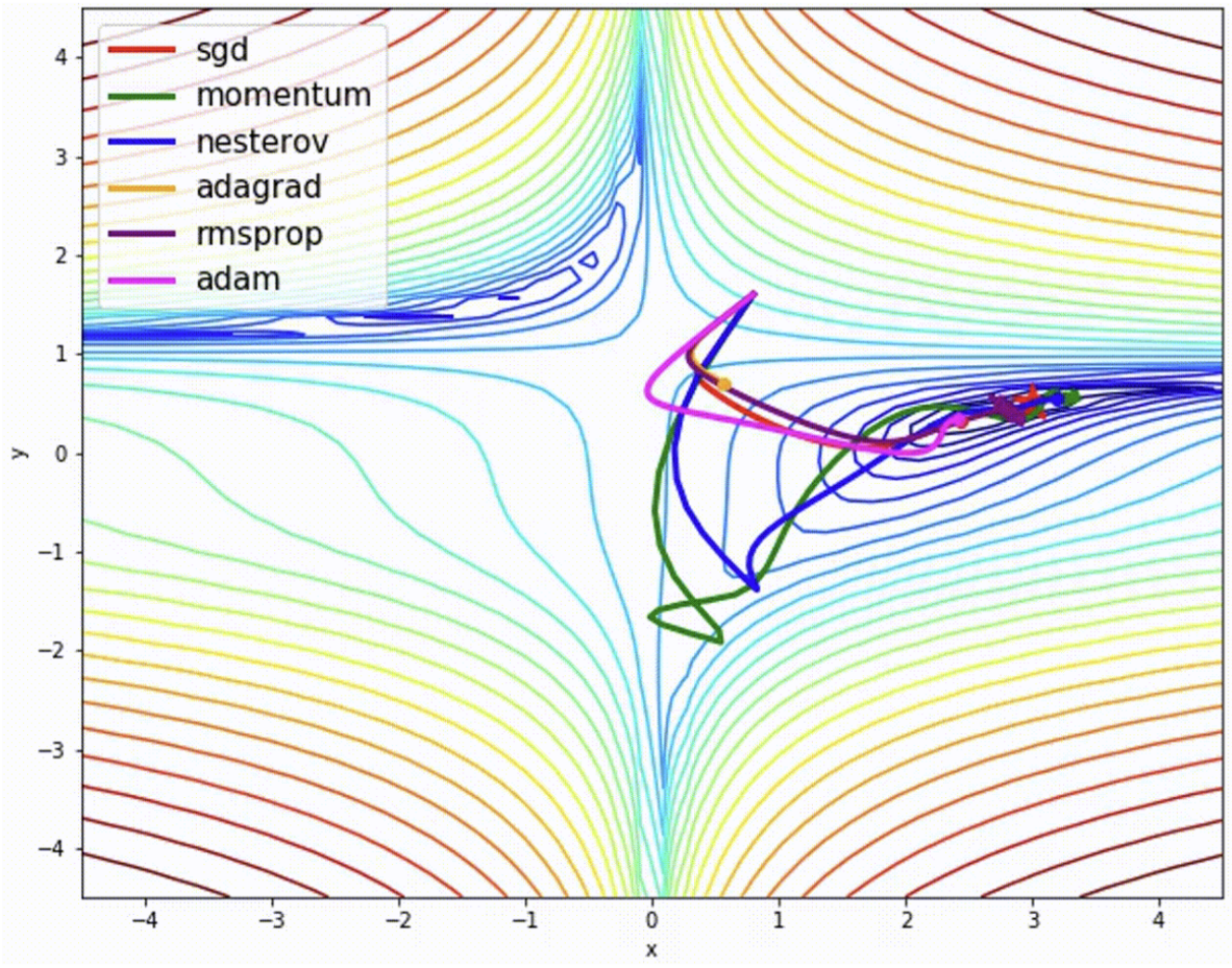


Рис. 7: 04f646b041c03895d3adc870f7d37ec9.png

Loss-функция / функция потерь нейронной сети (функция ошибки)

Это математическая дифференцируемая функция, характеризующая разницу между «истинным» значением целевой переменной и предсказанным нейронной сетью значением.

Актуально для такой нейронной сети, которая обучается с использованием обучающего набора (датасета), который содержит примеры с истинными значениями: тегами, классами, показателями.

Рассмотрим такие функции, как:

- средняя абсолютная MAE
- среднеквадратичная MSE
- категориальная кроссэнтропия
- бинарная кроссэнтропия

Средняя абсолютная ошибка (mean absolute error MAE) Средняя абсолютная ошибка представляет из себя модуль разности двух значений. Если сеть обучается на наборе данных (а не на одном примере), для подсчета ошибки берется среднее значение по всем примерам.

Чтобы рассчитать MAE, надо взять разницу между предсказанными значениями и истинными, усреднить по всему набору данных.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i^{(t)} - y_i^{(p)}|$$

где n = кол-во наблюдений, $y_i^{(t)}$ - истинное значение, $y_i^{(p)}$ - прогнозируемое значение

Пример подсчета MAE Рассмотрим простой пример подсчета средней абсолютной ошибки. Для этого вручную создадим два набора данных, которые будут соответствовать правильным ответам (y_true) и результатам работы нейронной сети (y_pred):

```
import numpy as np
y_true = np.array([15, 19, 11, -22, 23, -8, 18, 8, 9, -7, 17])
y_pred = np.array([12, 11, 7, -11, 12, -17, 10, 7, 1, -2, 11])
```

Посчитаем ошибку по указанной выше формуле:

```
abs_ = abs(y_true - y_pred)              # Получение вектора, содержащего модуль
 ↪  разности двух исходных наборов данных
mae = abs_.sum()/len(y_true)             # Получение среднего значения ошибки
print(mae)
```

Та же самая функция ошибки, но на другом примере. Потенциальная нейронная сеть дала более качественные результаты:

```
import numpy as np
y_true = np.array([15, 19, 11, -22, 23, -8, 18, 8, 9, -7, 17])
y_pred = np.array([14, 20, 11, -20, 21, -7, 17, 8, 7, -6, 15])

abs_ = abs(y_true - y_pred)
mae = abs_.sum()/len(y_true)
print(mae)
```

Как видим, чем точнее финальный результат (y_pred ближе к y_true), тем меньше значение ошибки.

```
_# Проверим расчеты встроенными методами tensorflow:

import tensorflow as tf
tf.keras.losses.mean_absolute_error(y_true, y_pred)

_# Пример использования средней абсолютной ошибки в Keras'e:

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
model.compile(loss='mean_absolute_error', optimizer='adam') # альтернативный
 ↪  вариант loss='mae'
```

Среднеквадратичная ошибка (mean squared error MSE) Средняя квадратичная ошибка (MSE): средняя квадратичная ошибка является наиболее распространенной функцией потерь. Функция потерь MSE широко **используется** в линейной регрессии в качестве показателя эффективности (в паре с функцией активации linear).

Чтобы рассчитать MSE, надо взять разницу между предсказанными значениями и истинными, возвести ее в квадрат и усреднить по всему набору данных.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}(y_i^{(t)} - y_i^{(p)})^2$$

где n = кол-во наблюдений, $y_i^{(t)}$ - истинное значение, $y_i^{(p)}$ - прогнозируемое значение

Пример подсчета Рассмотрим простой пример подсчета среднеквадратичной ошибки. Для этого вручную создадим два набора данных, которые будут соответствовать правильным ответам (y_true) и результатам работы нейронной сети (y_pred):

```
import numpy as np
y_true = np.array([15, 19, 11, -22, 23, -8, 18, 8, 9, -7, 17])
y_pred = np.array([12, 11, 7, -11, 12, -17, 10, 7, 1, -2, 11])
```

Посчитаем ошибку по указанной выше формуле:

```
sqr_ = abs(y_true - y_pred)**2  # Получение вектора, содержащего модуль разности
 ↪  двух исходных наборов данных
mse = sqr_.sum()/len(y_true)    # Получение значения ошибки
print(mse)
```

Та же самая функция ошибки, но на другом примере. Потенциальная нейронная сеть дала более качественные результаты:

```
import numpy as np
y_true = np.array([15, 19, 11, -22, 23, -8, 18, 8, 9, -7, 17])
y_pred = np.array([14, 20, 11, -20, 21, -7, 17, 8, 7, -6, 15])

sqr_ = abs(y_true - y_pred)**2
mse = sqr_.sum()/len(y_true)
print(mse)
```

```
_# Проверим расчеты встроенными методами tensorflow:

import tensorflow as tf
tf.keras.losses.mean_squared_error(y_true, y_pred)
```

```
_# Пример использования среднеквадратичной ошибки в Keras'e:

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
model.compile(loss='mean_squared_error', optimizer='adam') # альтернативный
 ↪  вариант loss='mse'
```

Кросс-энтропия (categorical crossentropy CCE) Кросс-энтропия (или логарифмическая функция потерь – log loss). Кросс-энтропия измеряет расхождение между двумя вероятностными распределениями. Если кросс-энтропия велика, это означает, что разница между двумя распределениями велика, а если кросс-энтропия мала, то распределения похожи друг на друга.

Кросс-энтропия определяется как:

$$logloss = -\sum_{i=1}^{output\_size} y_i \cdot \log \hat{y}_i$$

где: - $output\_size$ - количество наблюдений - y_i - истинное значение - $\hat{y}_i$ - прогнозируемое значение

Используется в мультиклассовой классификации, в паре с функцией активации softmax.

Пример подсчета Рассмотрим простой пример подсчета средней абсолютной ошибки. Для этого вручную создадим два набора данных, которые будут соответствовать правильным ответам (y_true) и результатам работы нейронной сети (y_pred):

```
import numpy as np

y_true = np.array([[0, 0, 1, 0], [1, 0, 0, 0]])
y_pred = np.array([[0.02, 0.07, 0.9, 0.01], [0.91, 0.03, 0.04, 0.02]])

y_true.shape

# (2, 4)
```

Посчитаем значение ошибки по указанной выше формуле:

```
cce_ = - sum(y_true * np.log(y_pred))
print(cce_)

# [ 0.09431068 -0.          0.10536052 -0.        ]

_# Проверим расчеты встроенными методами tensorflow:

import tensorflow as tf
tf.losses.categorical_crossentropy(y_true, y_pred)

# <tf.Tensor: shape=(2,), dtype=float64, numpy=array([0.10536052, 0.09431068])>
```

Мы получили результирующий вектор из двух значений, поскольку входной набор состоит из двух элементов. Для получения результирующего значения ошибки нам необходимо получить среднее значение:

```
cce_ = sum(cce_) / len(cce_)
print(cce_)

# 0.04991779878226689

_# Пример использования категориальной кроссэнтропии в Keras'е:

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Бинарная классификация (binary crossentropy BCE) Это функция ошибок, которую можно использовать для количественной оценки разницы между двумя распределениями вероятностей. Она говорит о том, что если у нас есть события и вероятности, насколько вероятно, что события произойдут на основе вероятностей? Если это очень вероятно, у нас малая кросс-энтропия, а если маловероятно, у нас высокая кросс-энтропия.

Формула имеет вид:

$$Loss = -\frac{1}{output\_size} \sum_{i=1}^{output\_size} (y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i))$$

где: - output size - количество наблюдений - y_i - истинное значение - $\hat{y}_i$ - прогнозируемое значение

При двоичной классификации каждая предсказанная вероятность сравнивается с фактическим значением класса (0 или 1), и вычисляется оценка, которая штрафует вероятность на основе расстояния от ожидаемого значения.

Используется в задачах бинарной классификации, в паре с функцией активации sigmoid

Пример подсчета Рассмотрим простой пример подсчета средней абсолютной ошибки. Для этого вручную создадим два набора данных, которые будут соответствовать правильным ответам (y_true) и результатам работы нейронной сети (y_pred):

```
import numpy as np
y_true = np.array([0, 0, 1, 0])
y_pred = np.array([0.02, 0.07, 0.9, 0.01])
```

Посчитаем ошибку по указанной выше формуле:

```
bce_ = -((y_true * np.log(y_pred) + (1-y_true) * np.log(1 -
 ↪  y_pred))).sum()/len(y_true)
print(bce_)

# 0.05204606291592068

_# Проверим расчеты встроенными методами tensorflow:

import tensorflow as tf
tf.losses.binary_crossentropy(y_true, y_pred)

# <tf.Tensor: shape=(), dtype=float64, numpy=0.05204595749369875>

_# Пример использования бинарной кроссэнтропии в Keras'e:

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

model = Sequential()
model.add(Dense(64, input_shape=(10,), activation='softmax'))
model.compile(loss='binary_crossentropy', optimizer='adam')
```

Вернёмся к нейросетям

Посмотрим, как теперь выглядит сеть, вызывая метод .summary():

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 32)                352

=================================================================
Total params: 352
Trainable params: 352
Non-trainable params: 0
_________________________________________________________________
```

Внимательно посмотрите на получившееся количество параметров.

При входе сети из **10** чисел и Dense-слое с **32** нейронами количество весовых коэффициентов будет $10 * 32 = 320$. Но, как видите, оно получается на **32** больше. Это происходит из-за наличия в слое ***нейрона смещения***.

Для чего он нужен? Бывают ситуации, в которых нейросеть просто не сможет найти верное решение из-за того, что нужная точка будет находиться вне пределов досягаемости. Именно для этого и нужны такие нейроны, чтобы иметь возможность сместить область определения.

Схематически нейроны смещения обычно не обозначаются, их вес учитывается по умолчанию при работе нейрона.

Нейрон смещения **Нейрон смещения** или **bias-нейрон** — это третий вид нейронов, используемый в большинстве нейросетей. Это гиперпараметр, его наличие/отсутствие определяется вами.

Особенность этого типа нейронов заключается в том, что его вход и выход всегда равняются 1 и они никогда не имеют входных синапсов. Нейроны смещения могут, либо присутствовать в нейронной сети по одному на слое, либо полностью отсутствовать, 50/50 быть не может.

Соединения у нейронов смещения такие же, как у обычных нейронов — со всеми нейронами следующего уровня, за исключением того, что синапсов между двумя bias нейронами быть не

может. Следовательно, их можно размещать на входном слое и всех скрытых слоях, но никак не на выходном слое, так как им попросту не с чем будет формировать связь.

Применение: - дает возможность изменять выходной результат, путем сдвига графика функции активации вправо или влево - помогают в том случае, когда все входные нейроны получают на вход 0 и независимо от того какие у них веса, они все передадут на следующий слой 0, но не в случае присутствия нейрона смещения

Рассмотрим как он работает на примере полносвязной сети.

Возьмем для примера простейшую нейросеть для бинарной классификации (1 класс - кошка / 2 класс - собака).

Каждый экземпляр (изображение кошки и собаки) имеет два параметра (некие x1 и x2, рост и вес в цифрах, например), при помощи нейросети мы должны отнести его к одному из двух классов.

Подготавливаем данные, подаем в сеть, обрабатываем функцией активации f(x) и в итоге получаем прогноз, к какому классу сеть отнесла экземпляр.

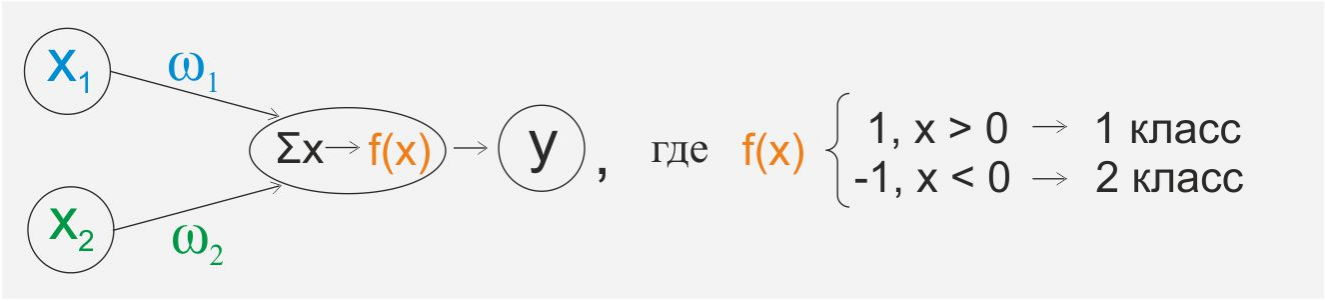
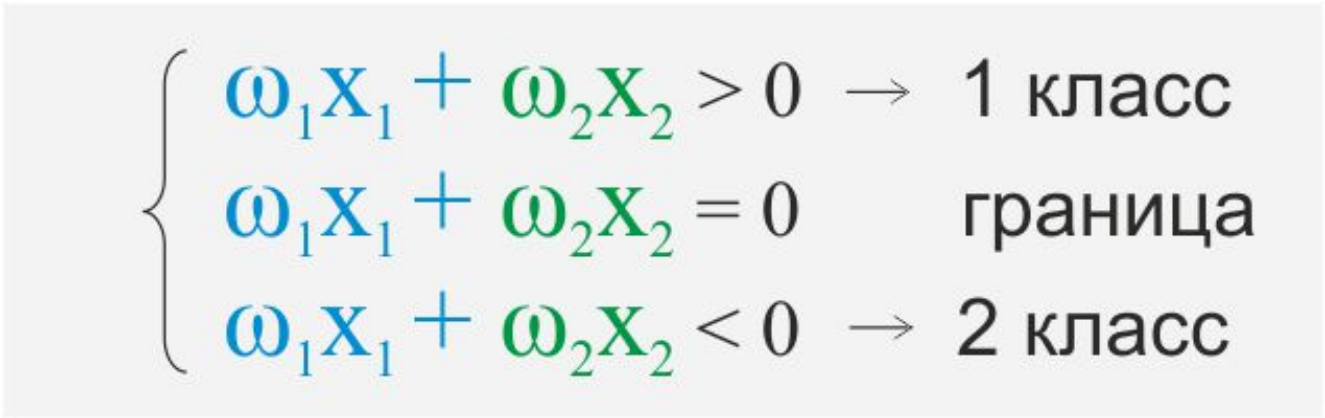


Рис. 8: bc088a3664156de55c7bafdee4938e71.png

Из формулы активационной функции f(x) следует, что граница разделения классов это x = 0. Если x больше нуля - то 1 класс, если меньше - то второй. В виде формул та же самая мысль выглядит вот так:



Рассмотрим внимательнее уравнение, которое описывает границу. Небольшое преобразование показывает нам, что это не что иное, как уравнение прямой. Она проходит через начало координат под некоторым наклоном к оси абсцисс (определяется угловым коэффициентом k = - w1/w2):

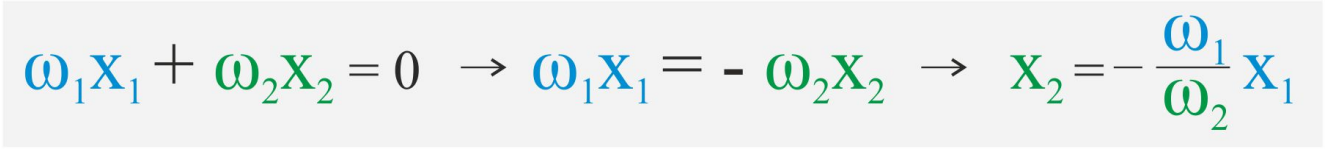


Рис. 9: e1ff86a98aff6473dbe1d5758a2da8e2.png

Относительно прямой можно определить множество точек класса 1 и класса 2 на плоскости. Они будут выше и ниже соотв., а сама прямая называется разделяющей (в многомерном - разделяющая гиперплоскость).

Однако не всегда точки на плоскости распределены так, что прямая через начало координат может разграничить два класса.

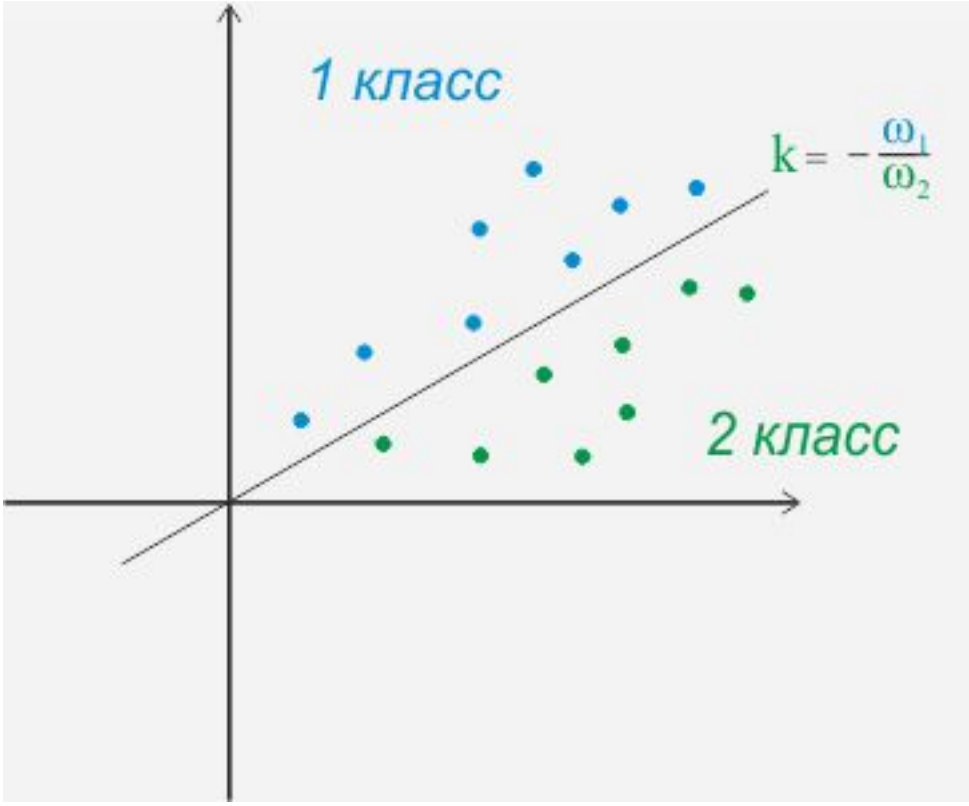


Рис. 10: 2122f57a258b279a6d160dba9594b0ce.png

Для демонстрации поднимем выше все точки на графике. Видим, что прямая перестала быть однозначной границей, и даже изменение ее угла не помогает разрешить ситуацию, минимизировать loss.

Чтобы прямая снова стала разделяющей, ее необходимо поднять по оси ординат вверх на некоторую величину. Это и есть bias - b, порог, смещение.

Это третий вход x3, который всегда имеет значение +1 и некий вес w3.

Перепишем нейросеть с учетом новых данных:

Как создавать нейрон смещения в Keras?

Параметру **use_bias** в описании слоя присваивается значение True. На текущий момент это значение по умолчанию, поэтому если вы не упомянете нейрон смещения - сеть автоматически его добавит. Чтобы отключить use_bias = False.

Создадим полносвязную сеть из одного входного (на 784 нейрона), одного скрытого (на 300 нейронов) и одного выходного (на 10 нейронов) слоев.

Посчитаем, сколько весов описывают нейросеть: - 784 * 300 = 235 200 весов - 10 * 300 = 3000 весов - 1 * 10 = 10 весов

```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import plot_model

model = Sequential()
model.add(Dense(300, activation = "relu", use_bias = False, input_shape = (784,),
 ↪  name ='Dense_300'))
model.add(Dense(10, activation = "softmax", name ='Dense_10'))

model.summary()
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Dense_300 (Dense)            (None, 300)               235200
```

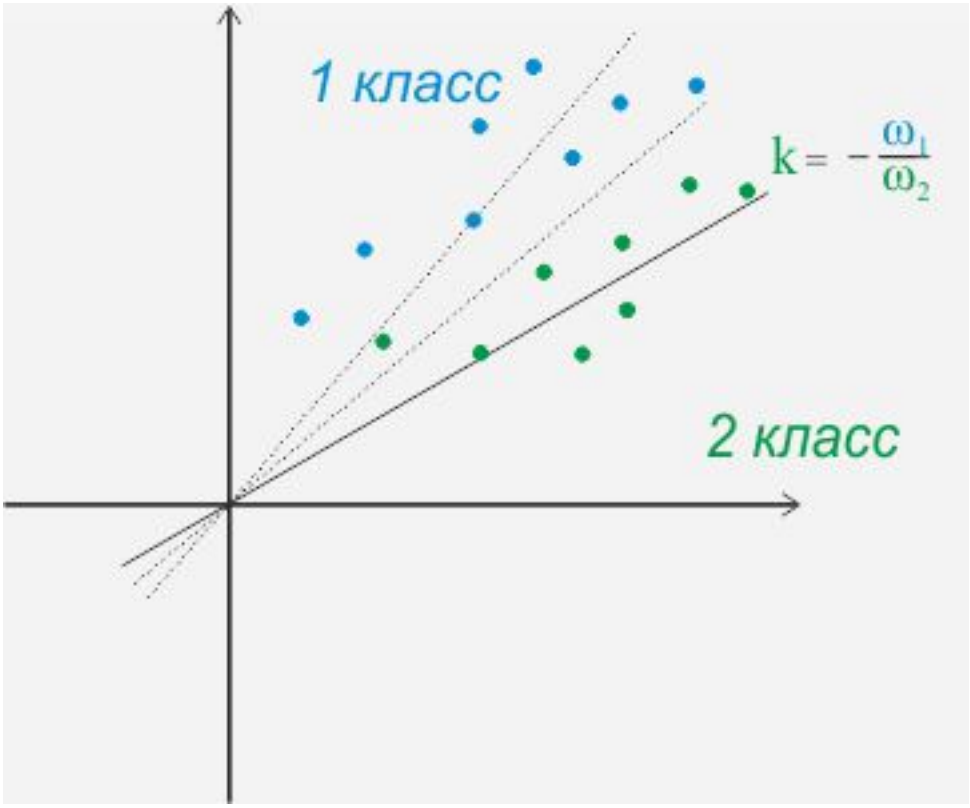


Рис. 11: e775548048dd56ddac4cdabf92ea448d.png

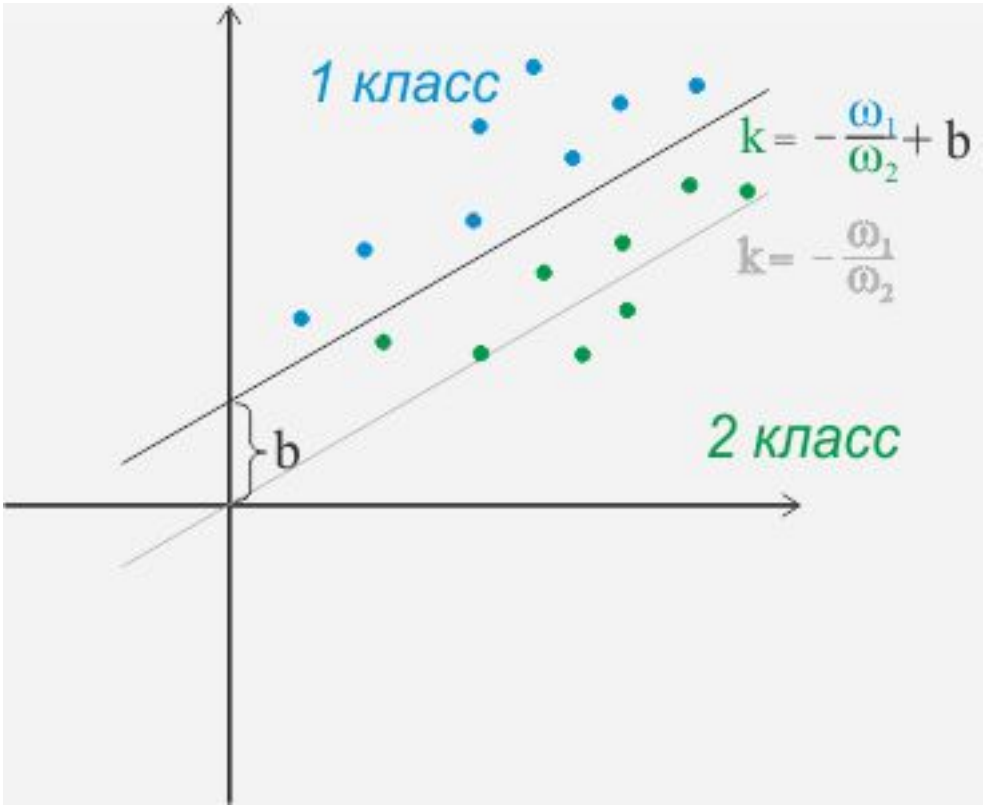


Рис. 12: 8824269c0bf9cfaf4acdb7806a59c5c8.png

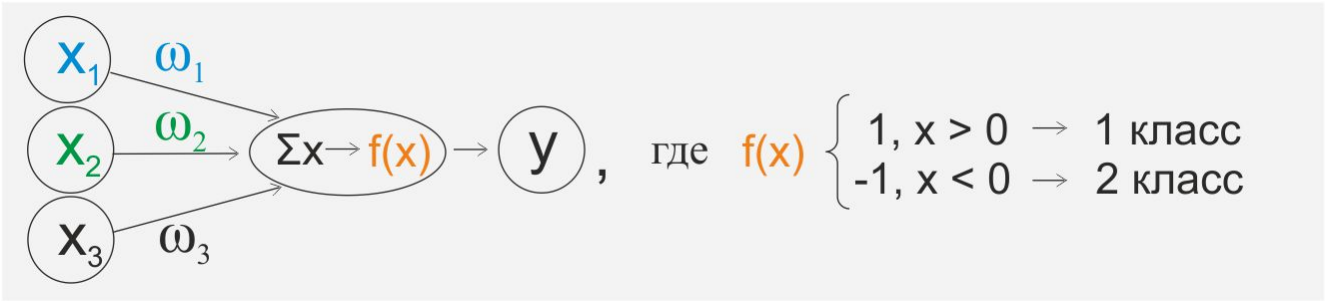


Рис. 13: 09627dc11dffef284f67129596a57dd7.png

```
_________________________________________________________________
Dense_10 (Dense)             (None, 10)                3010
=================================================================
Total params: 238,210
Trainable params: 238,210
Non-trainable params: 0
_________________________________________________________________
```

Теперь создадим аналогичную нейросеть, но с нейроном смещения. Это значит, что для соединения нейрона со скрытым слоем потребуетс еще 1 * 300 связей, и в сумме количество параметров должно увеличиться на 300. Проверим:

```
from keras.models import Sequential
from keras.layers import Dense
from tensorflow.keras.utils import plot_model

model = Sequential()
model.add(Dense(300, activation = "relu", use_bias = True, input_shape = (784,),
 ↪  name ='Dense_300'))
model.add(Dense(10, activation = "softmax", name ='Dense_10'))

model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Dense_300 (Dense)            (None, 300)               235500
_________________________________________________________________
Dense_10 (Dense)             (None, 10)                3010
=================================================================
Total params: 238,510
Trainable params: 238,510
Non-trainable params: 0
_________________________________________________________________
```

Резюме:

- нейрон смещения, в ряде случаев, позволяет при решении задачи обойтись меньшим количеством нейронов/связей.
- нейроны смещения могут либо присутствовать в каждом слое или отсутствовать (кроме выходного слоя).
- в случае с нелинейными функциями активации, например, сигмоидой, веса позволяют изменять наклон сигмоиды, а смещением по оси y управляет нейрон смещения.

Продолжение про Sequential модель

Вы можете заметить, что у сети есть название **“sequential”**. Оно автоматически присваивается при создании.

У слоев также есть названия. Они указаны в левой колонке.

Колонка **“Output Shape”** показывает форму данных на выходе нейронного слоя.

В данном случае у вас получается: * на вход нейронной сети подается последовательность из **10** элементов (вы указали это с помощью параметра **input_dim**) * нейронная сеть состоит из одного слоя (**Dense**), который состоит из **32** нейронов (количество нейронов вы указали при создании слоя) * на выходе нейронной сети будет последовательность из **32** элементов (выход нейронной сети равен выходу последнего слоя)

В выведенной информации вы можете увидеть строку **“Total params: 352”**. В ней указано общее количество параметров модели.

Параметры

Параметры модели – это все веса, внутренние настройки сети, которые определяют, как будет преобразован объект, подаваемый в сеть, прежде чем оказаться на выходе. Они автоматически изменяются при обучении.

Вы можете добавлять неограниченное количество слоев к сети:

```
model = Sequential()
model.add(Dense(32, input_dim=10))
model.add(Dense(5))
model.add(Dense(1))

model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_1 (Dense)              (None, 32)                352

 dense_2 (Dense)              (None, 5)                 165

 dense_3 (Dense)              (None, 1)                 6

=================================================================
Total params: 523
Trainable params: 523
Non-trainable params: 0
_________________________________________________________________<!--  -->
```

Выше добавлено еще два линейных слоя. Теперь выход изменился: сеть на выходе выдает одно число. Как видите, также изменилось и общее число параметров. Чем больше слоев и нейронов в слоях, тем больше параметров сети.

Обучение последовательной сети с полносвязными слоями. Распознавание рукописных цифр MNIST

Далее вы узнаете, как обучить модель нейронной сети.

Теперь вы научитесь делать удивительные вещи благодаря нейронным сетям. Если быть точнее, вы буквально в несколько строчек кода создадите собственную нейронную сеть и научите ее с высокой точностью распознавать рукописные цифры, изображенные на картинках. То есть на вход нейросети будет приходить картинка с изображением цифры, а на выходе вы получите значение цифры (число).

Подготовка данных Для начала добавьте в проект все необходимые модули: * **mnist** – модуль для загрузки набора данных рукописных цифр, который вы используете при обучении нейронной сети; * **Sequential** – модуль для создания последовательной модели нейронной сети; * **Dense** – линейный (полносвязный) слой. Из таких слоев будет создана ваша нейросеть; * **utils** – модуль с полезными инструментами для подготовки данных; * **plt** – модуль рисования графиков.

```
from tensorflow.keras.datasets import mnist      # Библиотека с базой рукописных
 ↪ цифр
from tensorflow.keras.models import Sequential  # Подключение класса создания
 ↪ модели Sequential
from tensorflow.keras.layers import Dense       # Подключение класса Dense -
 ↪ полносвязный слой
```

```
from tensorflow.keras import utils                  # Утилиты для подготовки данных
import numpy as np                                  # Работа с массивами
import matplotlib.pyplot as plt                     # Отрисовка изображений

# Отрисовка изображений в ячейках ноутбука
%matplotlib inline
```

Чтобы чему-то научить вашу нейронную сеть, понадобится набор данных для задачи, которую вы собираетесь решать. У вас это набор картинок, на которых изображены рукописные цифры от **0** до **9**. Следующей строкой кода вы скачаете эти данные: * **x_train_org**, **y_train_org** – изображения для обучения нейронной сети; * **x_test_org**, **y_test_org** – изображения для тестирования нейронной сети.

```
# Загрузка из облака данных Mnist
(x_train_org, y_train_org), (x_test_org, y_test_org) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.np
11490434/11490434 [==============================] - 0s 0us/step
```

Сейчас те самые картинки, с помощью которых вы будете обучать нейронную сеть, представляют из себя матрицы чисел, где каждое число – это значение яркости пиксела (от **0** до **255**). Таких изображений в вашем тренировочном наборе данных **60000**, и их размер **28** на **28** пикселов. Поэтому форма массива данных выглядит так:

```
# Вывод формы данных для обучения
x_train_org.shape
```

```
(60000, 28, 28)
```

Убедитесь, что ваши данные действительно представляют собой картинки с числами.

Для этого выберите из массива x_train_org какую-нибудь матрицу чисел и отобразите ее в серой шкале:

```
# Номер картинки
n = 143

# Отрисовка картинки
plt.imshow(x_train_org[n], cmap='gray')

# Вывод n-й картинки
plt.show()
```

Видите изображение цифры **2**?

Если же вы взглянете на массив **y_train_org**, который содержит в себе метки для картинок (то есть правильные значения распознаваемых цифр на картинках), то увидите, что этой картинке соответствует значение **2**:

```
# Вывод метки класса для n-го изображения
print(y_train_org[n])

# 2
```

Сейчас ваши данные имеют сложную структуру, где каждая картинка представляет собой двумерный массив данных.

Для обучения нейронной сети вам необходимо преобразовать изображение в более простой вид – в одномерную последовательность чисел (вектор).

Сделайте это с помощью метода `.reshape()`:

```
# Изменение формы входных картинок с 28x28 на 784
# первая ось остается без изменения, остальные складываются в вектор
x_train = x_train_org.reshape(x_train_org.shape[0], -1)
x_test = x_test_org.reshape(x_test_org.shape[0], -1)

# Проверка результата
print(f'Форма обучающих данных: {x_train_org.shape} -> {x_train.shape}')
print(f'Форма  тестовых данных: {x_test_org.shape} -> {x_test.shape}')
```

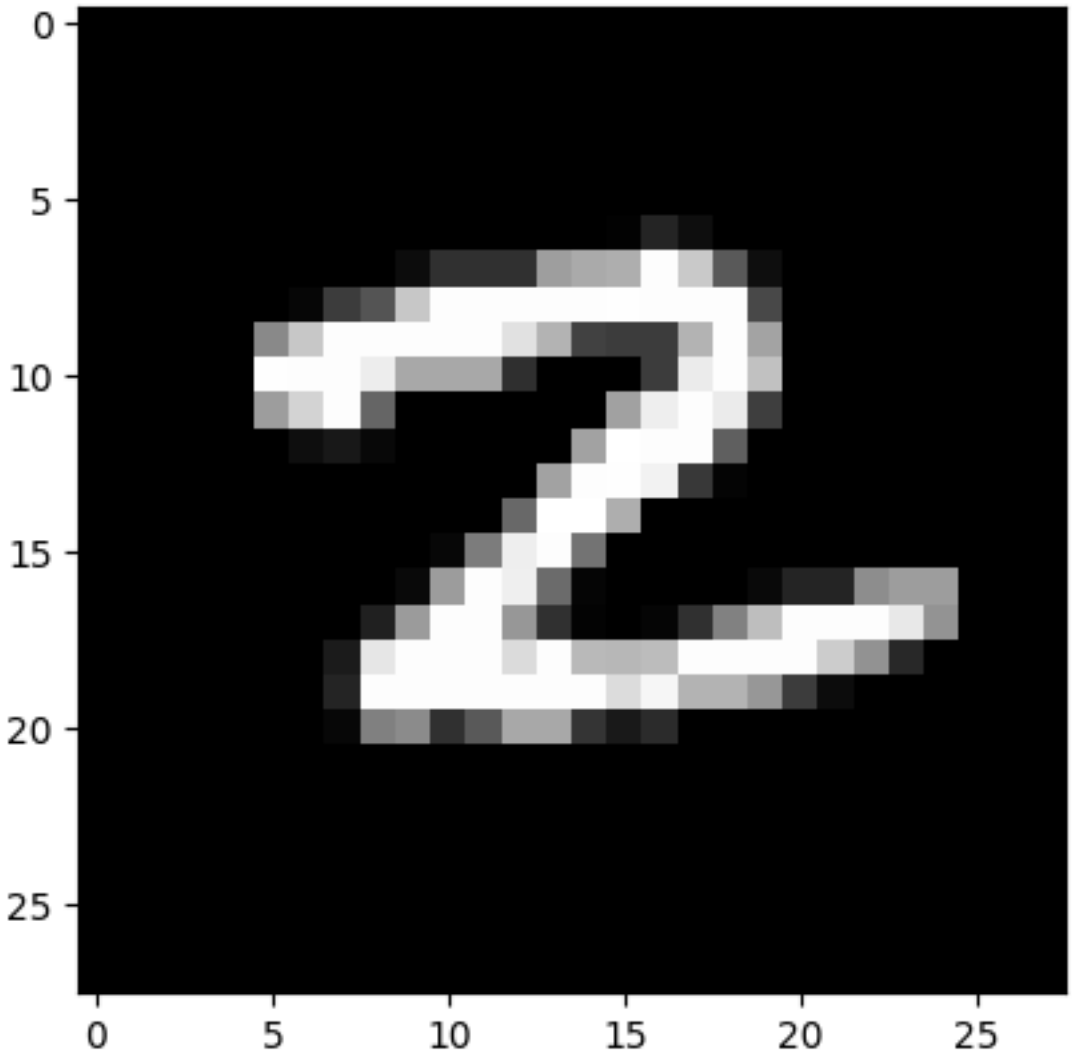


Рис. 14: d08b8f48e9fb525bd17bcd7757c9157b.png

```
Форма обучающих данных: (60000, 28, 28) -> (60000, 784)
Форма  тестовых данных: (10000, 28, 28) -> (10000, 784)
```

Теперь каждая картинка представлена в вашем наборе данных последовательностью из **784** чисел **(28x28)**.

Как вы уже знаете, чтобы нейронной сети было легче учиться, необходимо привести входные данные к некоему стандартному виду. В нашем случае числа, которые описывают картинку (те самые **784** числа), должны находиться в диапазоне от **0** до **1**, хотя сейчас они в диапазоне от **0** до **255**, ведь они описывают интенсивность каждого пиксела. Поделив все эти значения на **255**, вы нормализуете входные данные:

```
_# Нормализация входных картинок
# Преобразование x_train в тип float32 (числа с плавающей точкой) и нормализация
x_train = x_train.astype('float32') / 255.

# Преобразование x_test в тип float32 (числа с плавающей точкой) и нормализация
x_test = x_test.astype('float32') / 255.
```

Также нужно провести некоторые преобразования и с метками классов, то есть с теми числами, которые отвечают на вопрос «Что же изображено на картинке?».

Для этого нужно привести все метки к виду ***one hot encoding***.

Это значит, что каждое число будет представлять собой последовательность (вектор) значений **0** или **1**. Последовательность будет длиной **10**, потому что всего существует **10** цифр, которые вы будете распознавать (от **0** до **9**). В векторе one hot encoding везде стоят нули, кроме позиции самой метки. * Если ответ равен **5**, то one hot encoding представление будет таким:

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

- А, например, для класса **3**:

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

Чтобы не указывать везде явно число классов, хорошим тоном будет назначить константу. Тогда, если вам захочется использовать удачную архитектуру модели на других данных с другим количеством классов, не придется менять это значение в каждом месте, где оно требуется:

```
_# Задание константы количества распознаваемых классов

CLASS_COUNT = 10
```

Преобразуйте выходные данные в векторы one hot encoding с помощью функции `to_categorical()` модуля **utils**:

```
_# Преобразование ответов в формат one_hot_encoding
y_train = utils.to_categorical(y_train_org, CLASS_COUNT)
y_test = utils.to_categorical(y_test_org, CLASS_COUNT)
```

Теперь выходные данные для вашей нейронной сети выглядят следующим образом:

```
_# Вывод формы y_train
# 60 тысяч примеров, каждый длины 10 по числу классов
print(y_train.shape)
```

```
(60000, 10)
```

```
_# Вывод примера одного выходного вектора
print(y_train[0])
```

```
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

Для сравнения - в оригинальных выходных данных все метки классов - просто одиночные числа:

```
_# Вывод формы массива меток
print(y_train_org.shape)
```

```
(60000,)
```

```
_# Вывод метки, соответствующей 36-му элементу
print(y_train_org[36])
```

```
6
```

Создание нейронной сети Теперь самое интересное. Буквально за несколько строчек кода вы создадите свою собственную нейронную сеть, а потом научите ее распознавать цифры!

Для начала создайте объект нейронной сети с помощью класса **Sequential**:

```
model = Sequential()
```

Сейчас это пустая нейронная сеть, не содержащая в себе никаких слоев и нейронов. Добавьте в нее несколько слоев нейронов, идущих друг за другом, последовательно, по образцу:

```
model.add(Dense(400, activation='relu'))
```

В данном случае **400** – это количество нейронов в слое, а **'relu'** – функция активации, которая будет применяться после умножения значений входов нейрона на его веса.

```
_# Создание последовательной модели
model = Sequential()

# Добавление полносвязного слоя на 800 нейронов с relu-активацией
model.add(Dense(800, input_dim=784, activation='relu'))

# Добавление полносвязного слоя на 400 нейронов с relu-активацией
model.add(Dense(400, activation='relu'))

# Добавление полносвязного слоя с количеством нейронов по числу классов с
 ↪  softmax-активацией
model.add(Dense(CLASS_COUNT, activation='softmax'))
```

В целом, вы уже создали нейронную сеть. Теперь нужно подготовить ее к обучению (скомпилировать) и запустить само обучение.

Следующей строкой кода вы скомпилируете модель:

```
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

В методе `.compile()` вы назначаете функцию ошибки (**'categorical_crossentropy'**), оптимизатор нейронной сети (**'adam'**) и метрики, которые будут подсчитываться в процессе обучения нейросети (**['accuracy']**).

Метод `.summary()` выведет на экран структуру вашей нейронной сети в виде таблицы:

```
_# Компиляция модели
model.compile(loss='categorical_crossentropy', optimizer='adam',
 ↪  metrics=['accuracy'])

# Вывод структуры модели
print(model.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense_4 (Dense)              (None, 800)               628000

 dense_5 (Dense)              (None, 400)               320400

 dense_6 (Dense)              (None, 10)                4010

=================================================================
Total params: 952,410
Trainable params: 952,410
Non-trainable params: 0
_________________________________________________________________
None
```

Функция plot_model() модуля utils нарисует наглядную схему (граф) нейронной сети, она удобна для понимания и более сложных моделей.

Эта функция принимает следующие аргументы:

model - модель, схему которой вы хотите построить (обязательный параметр); to_file - имя файла или путь к файлу, в который сохраняется схема (обязательный параметр); show_shapes - Показывать или нет формы входных/выходных данных каждого слоя (необязательный параметр, по умолчанию False); show_layer_names - показывать или нет название каждого слоя (необязательный параметр, по умолчанию True).

```
utils.plot_model(model, to_file='model_plot.png', show_shapes=True,
↪   show_layer_names=False)
```

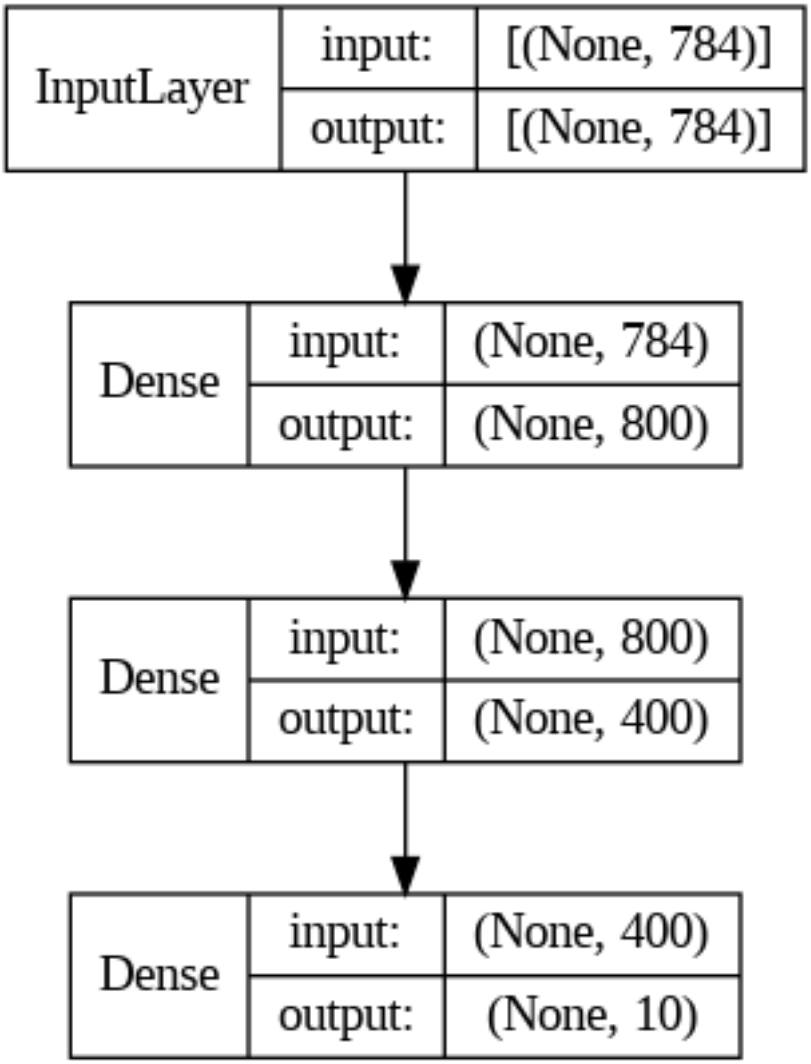


Рис. 15: 2a44888ca434027f7079634948391195.png

Здесь вы впервые сталкиваетесь с сохранением данных.

Сейчас сохранение графа модели произойдёт в хранилище виртуальной машины ноутбука. Найти сохраненный файл можно, нажав на иконку **“Файлы”** в левой части рабочего пространства Google Colab.

В данном случае при завершении сеанса все файлы будут удалены вместе с виртуальной машиной.

Если необходимо воспользоваться файлами в дальнейшем, можно сохранить их на свой Google-диск - это постоянное хранилище. Для этого необходимо подключить его:

```
from google.colab import drive
drive.mount('/content/drive/')
```

Необходимо перейти по ссылке и разрешить доступ.

Теперь данные могут быть загружены на диск, но для этого необходимо указывать полный путь для сохранения

Обучение нейронной сети Одной строчкой кода вы запускаете обучение нейронной сети и сможете наблюдать за процессом.

Для этого вызовите метод модели `.fit()` и передайте ему данные для обучения - **x_train, y_train**:

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

batch_size – размер батча, который указывает нейросети на то, сколько картинок она будет обрабатывать за один раз.

epochs – количество циклов обучения, то есть сколько раз нейронная сеть повторит просмотр и обучение на всех ваших данных.

```
model.fit(x_train,        # обучающая выборка, входные данные
          y_train,        # обучающая выборка, выходные данные
          batch_size=128, # кол-во примеров, которое обрабатывает нейронка перед
 ↪  одним изменением весов
          epochs=15,      # количество эпох, когда нейронка обучается на всех
 ↪  примерах выборки
          verbose=1)      # 0 - не визуализировать ход обучения, 1 -
 ↪  визуализировать
```

```
Epoch 1/15
469/469 [==============================] - 7s 4ms/step - loss: 0.2073 - accuracy: 0.9383
Epoch 2/15
469/469 [==============================] - 2s 4ms/step - loss: 0.0755 - accuracy: 0.9762
Epoch 3/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0477 - accuracy: 0.9853
Epoch 4/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0336 - accuracy: 0.9893
Epoch 5/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0270 - accuracy: 0.9912
Epoch 6/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0206 - accuracy: 0.9931
Epoch 7/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0185 - accuracy: 0.9936
Epoch 8/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0158 - accuracy: 0.9948
Epoch 9/15
469/469 [==============================] - 2s 5ms/step - loss: 0.0125 - accuracy: 0.9959
Epoch 10/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0157 - accuracy: 0.9947
Epoch 11/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0095 - accuracy: 0.9969
Epoch 12/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0122 - accuracy: 0.9958
Epoch 13/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0094 - accuracy: 0.9973
Epoch 14/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0080 - accuracy: 0.9973
Epoch 15/15
469/469 [==============================] - 2s 3ms/step - loss: 0.0127 - accuracy: 0.9958
<keras.callbacks.History at 0x7f40d00bf970>
```

Только что вы наблюдали процесс обучения нейронной сети на 15 эпохах. После каждого цикла обучения вы можете видеть среднее значение ошибки. Обратите внимание, что практически каждую эпоху значение метрики точности (accuracy) увеличивается. Это означает, что ваша нейронная сеть с каждым разом делает все более точное распознавание!

Теперь сохраните веса вашей модели, чтобы потом можно было снова их использовать:

```
model.save_weights('model.h5')
model.load_weights('model.h5')
```

Распознавание рукописных цифр Вы научили нейронную сеть распознавать цифры на картинках с “огромной” точностью – более **99**%.

Теперь вы можете использовать модель по прямому назначению. Выведите на экран какой-нибудь образец из тестового набора данных:

```
_# Номер тестовой цифры, которую будем распознавать
n_rec = np.random.randint(x_test_org.shape[0])

# Отображение картинки из тестового набора под номером n_rec
plt.imshow(x_test_org[n_rec], cmap='gray')
plt.show()
```

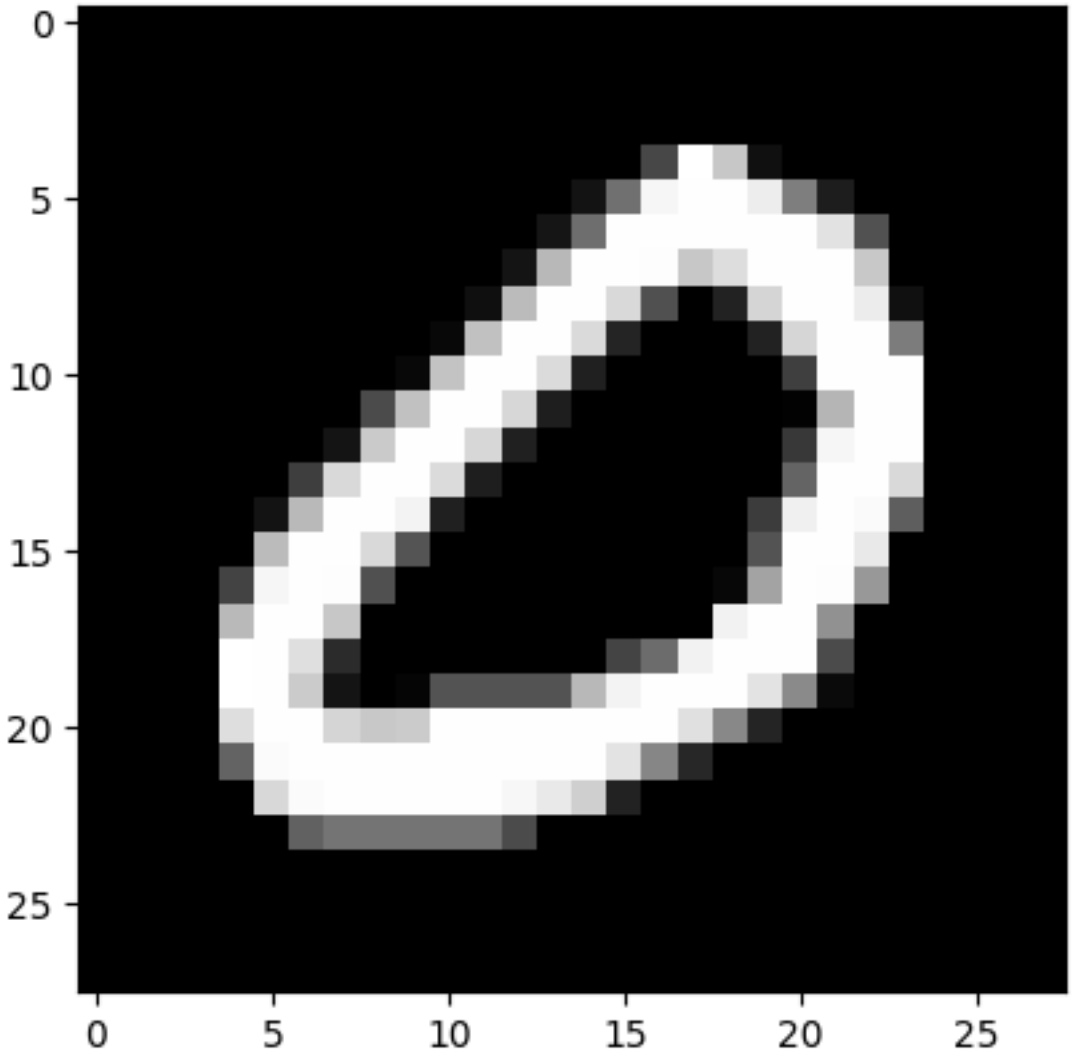


Рис. 16: 2b7508f9760221c1a2b481ad38367ecd.png

Теперь сохраняем в переменную **x** эту картинку в виде набора из **784** чисел.

Это нужно для того, чтобы нейросеть сделала ***предсказание***, ведь она обучалась работать именно с такими последовательностями чисел:

```
_# Выбор нужной картинки из тестовой выборки
x = x_test[n_rec]

# Проверка формы данных
print(x.shape)
```

```
(784,)
```

```
_# Добавление одной оси в начале, чтобы нейронка могла распознать пример
# Массив из одного примера, так как нейронка принимает именно массивы примеров
  ↪  (батчи) для распознавания
x = np.expand_dims(x, axis=0)

# Проверка формы данных
print(x.shape)
```

```
(1, 784)
```

Чтобы ваша сеть сделала предсказание, нужно вызвать метод `.predict()` и передать в него данные для распознавания:

```
_# Распознавание примера
prediction = model.predict(x)
```

```
1/1 [==============================] - 0s 77ms/step
```

Ответом вашей нейронной сети будет такой массив чисел:

```
_# Вывод результата - вектор из 10 чисел
print(prediction)
```

```
[[1.0000000e+00 1.2950315e-27 5.8127598e-15 4.3103217e-25 2.2470348e-24
  6.9105766e-28 2.4424716e-20 4.8785067e-22 2.0655704e-23 7.1440537e-18]]
```

Эти числа характеризуют вероятности принадлежности к конкретному классу. Самое первое число в этой последовательности отвечает на вопрос, какова вероятность, что на картинке изображена цифра **0**. Второе число говорит то же самое про цифру **1**. Сумма всех вероятностей равна единице, то есть предполагатся полный набор событий (только цифры от **0** до **9** и ничего другого):

```
sum(prediction[0])
```

```
1.0000000000000058
```

Таким образом, индекс самой большой вероятности в этом списке чисел и будет ответом вашей нейронной сети:

```
_# Получение и вывод индекса самого большого элемента (это значение цифры, которую
 ↪  распознала сеть)
pred = np.argmax(prediction)
print(f'Распознана цифра: {pred}')
```

```
Распознана цифра: 0
```

Как видите, ваша сеть находит правильный ответ.

```
_# Вывод правильного ответа для сравнения
print(y_test_org[n_rec])
```

Выводы В НС нейроны отвечают за выделение определенного признака, при этом некоторое количество нейронов организовано в слои. Несколько слоев нейронов могут называться полноценной НС.

Сама по себе сеть без обучения не может хорошо решать задачи. Для этого ей требуются две выборки – обучающая и проверочная. Используя их, НС узнает, как правильно выполнить задачу. В этом ей помогает функция ошибки, которая указывает, в правильную ли сторону НС меняет свои веса. После этого вы познакомились с полносвязным слоем, нейроны которого связаны со всеми входными нейронами.

Далее мы попробовали создать свою первую модель нейронной сети. Подгрузили в ноутбук основу **Sequential**, которая отвечает за построение модели, и создали начальную модель: `model = Sequential()`. Но она выглядела как пустая коробка, в которую необходимо что-то положить. И первым стал слой **Dense** (полносвязный слой). Так появилась модель, но ее еще нужно было обучить, и для этого вы указали оптимизатор и функцию потерь при помощи метода `.compile()`. Завершили все это вызовом метода `.summary()`, посмотрев, как выглядит структура НС.

Затем мы перешли к практике и решили первую задачу по распознаванию рукописных цифр. Для этого импортировали все необходимые инструменты и загрузили данные **MNIST**. Определили форму массива данных и представление данных в виде картинки. Преобразовали данные для модели НС, превратив картинку **28x28** пикселов в последовательность из **784** чисел. Не оставили без внимания и метки классов. Чтобы сеть лучше классифицировала, перевели метки в формат **one hot encoding**. По изученной схеме создали НС и приступили к ее обучению. В завершение проверили, как обученная НС распознает отдельные изображения рукописных цифр из набора, на котором сеть не обучалась.