



Введение в искусственные нейронные сети

Keras

# На этом уроке

1. Изучим способы создания нейросетей
2. Познакомимся с Keras и основами синтаксиса
3. Попрактикуемся создать нейросеть на Keras

# Оглавление

[На этом уроке](#_cutwx3tzcepj)

[Оглавление](#_nu4r6zsec6gd)

[Обратное распространение ошибки](#_patdqhygha4k)

[Процесс обучения нейросетей](#_taa8i09s788y)

[Способы создания нейросетей](#_s6j5c21ctxeo)

[Что такое Keras](#_9qcdxit9b0g0)

[Основы синтаксиса](#_h7stk8q193ma)

[Установка и работа с данными](#_gvkb6sc091m5)

[Создание модели](#_4mxtb6ixblns)

[Компиляция модели](#_a6lckai9t3f)

[Передача данных для обучения нейросети](#_uc3tzgl4odf3)

[Оценка обученности нейронной сети](#_qwh8z3hwh6e5)

[Запуск нейронной сети для выполнения работы](#_9zc65pwd3pm1)

[Многоклассовая классификация](#_kcqardb3tpem)

[Простая нейросеть на Keras](#_3ncxm2wjjbiq)

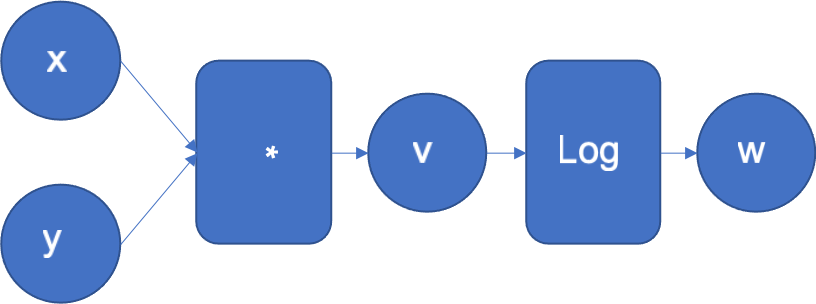
[Дополнительные материалы](#_z337ya)

[Используемые источники](#_i7c33zmgdbdr)

# Обратное распространение ошибки

На прошлом занятии было упомянуто обратное распространение ошибки. Для понимания этого метода рассмотрим такое понятие, как граф вычислений. Граф вычислений — это граф, узлами которого являются функции (обычно достаточно простые, взятые из заранее фиксированного набора), а ребра связывают функции со своими аргументами.

Рассмотрим пример: f(x, y) = log(x\*y). Построим для него граф вычислений:



Результат функции f(x, y) равен w. При подсчете результата функции граф проходится слева направо, в *прямом* направлении.

При обучении нейронной сети нас интересует градиент, вектор, направленный в сторону наискорейшего роста функции (а с минусом в сторону наискорейшего уменьшения). Градиент функции равен вектору ее частных производных.

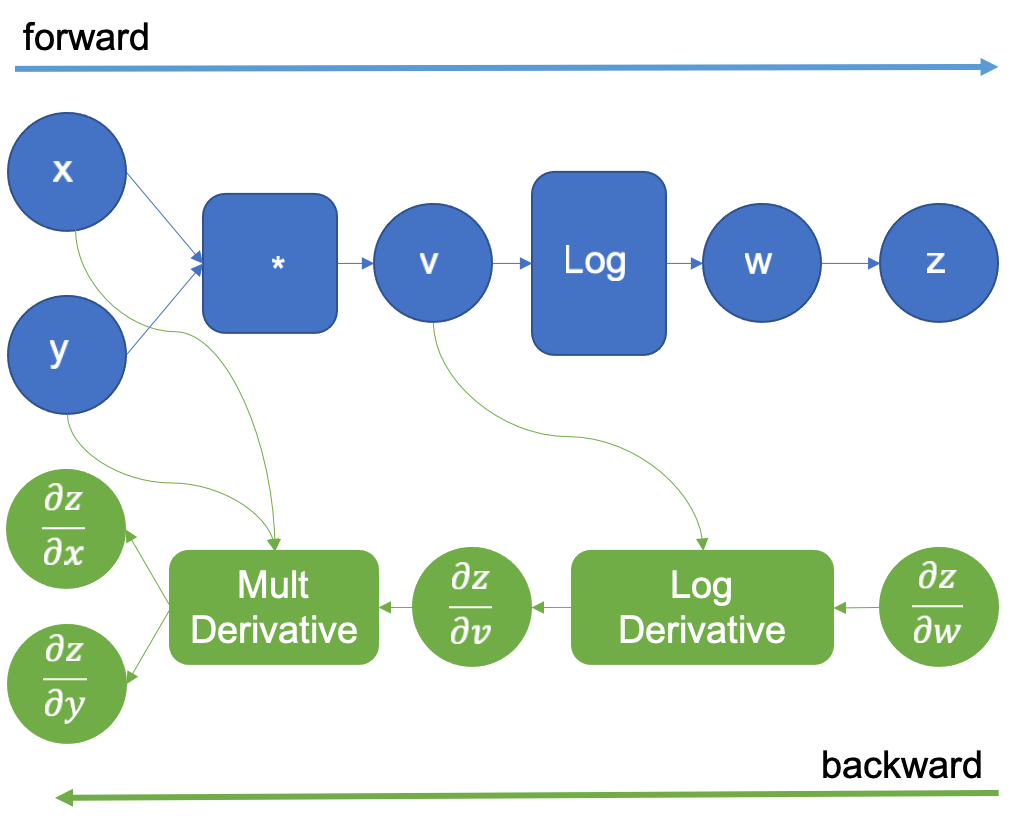
Чтобы найти частные производные используем формулу частной производной сложной функции:

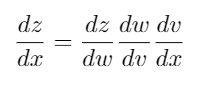
.

Вернемся к функции из примера и рассмотрим ее, как сложную функцию:

; .

Тогда дифференцирование этой функции можно рассматривать, как на изображении ниже:



Чтобы подсчитать частную производную всей функции по x нужно пройти такую цепочку:.

Можно раскрывать эту формулу, проходя граф слева направо, начиная с dv/dx. Это будет называться *прямым* прохождением.

Но также можно раскрывать ее, начиная с последней операции так, как изображено на зеленом графе выше. Это и есть *обратное* прохождение.

Когда нейронная сеть получает данные, вычисления происходят, начиная с первых слоев. Но когда мы считаем градиенты функции ошибки по параметрам (весам) нейронной сети, применяется метод *обратного распространения ошибки* от последних слоев к первым.

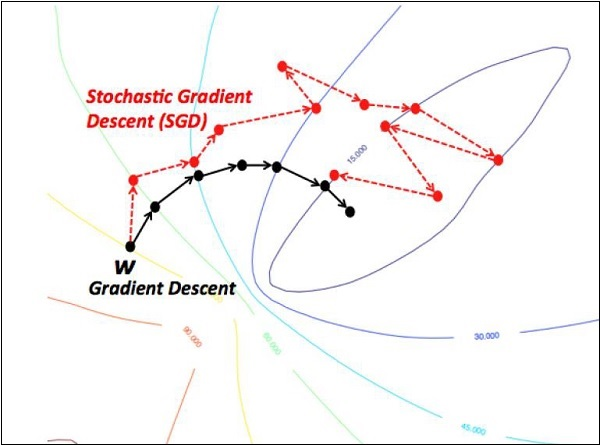
Подробнее в википедии: [Метод обратного распространения ошибки](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%BE%D0%B1%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B3%D0%BE_%D1%80%D0%B0%D1%81%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F_%D0%BE%D1%88%D0%B8%D0%B1%D0%BA%D0%B8)

# Процесс обучения нейросетей

Теперь мы полностью понимаем, как нейронная сеть обучается, как математическая модель. Но для практики необходимо уточнить еще такие понятия, как стохастический градиентный спуск, батчи и шаг обучения (learning rate).

Как вы уже знаете, нейронные сети обучаются благодаря градиентному спуску. Чтобы сделать один шаг для этого алгоритма нужно сначала прогнать модель на всех данных, затем рассчитать градиент для всех данных и вычесть его из весов. Но когда мы говорим о количестве данных для нейронных сетей речь идет о, как минимум, тысячах элементов. У обычных компьютеров вряд ли хватит оперативной памяти, чтобы хранить там все данные и еще выполнять какие-то операции с ними. Кроме того, такой алгоритм работает довольно медленно, ведь все данные используются только для того, чтобы сделать один шаг.

Как альтернатива такому подходу, существует **стохастический градиентный спуск**. От обычного [градиентного спуска](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%B4%D0%B8%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D1%83%D1%81%D0%BA) он отличается тем, что градиент оптимизируемой функции считается на каждом шаге не как сумма градиентов от каждого элемента выборки, а как градиент от одного, случайно выбранного элемента. По изображению видно, что такой алгоритм достигает минимума за большее количество итераций. Но фактически за счет того, что каждая итерация выполняется быстрее по времени, стохастический градиентный спуск может сходиться к минимуму на больших данных гораздо быстрее обычного градиентного спуска



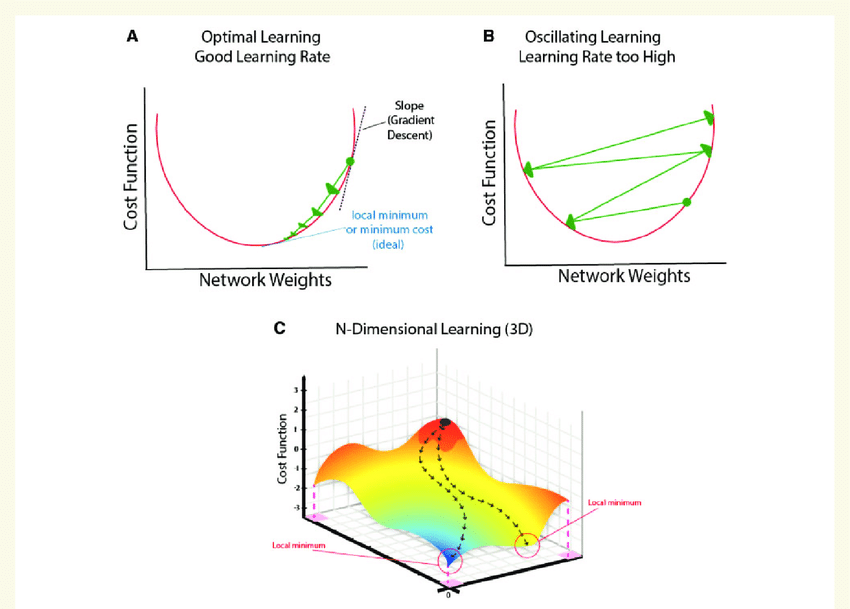
Компромиссом между вычислением истинного градиента и градиента по одному тренировочному примеру может быть вычисление градиента по более чем одному тренировочному примеру (называемому батчем) на каждом шаге. Это может быть существенно лучше, чем описанный стохастический градиентный спуск, поскольку код может использовать библиотеки [векторных форм](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%B5%D0%BE%D0%B1%D1%80%D0%B0%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2_%D0%B2%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%BD%D1%83%D1%8E_%D1%84%D0%BE%D1%80%D0%BC%D1%83&action=edit&redlink=1), а не вычислять каждый шаг отдельно. Это может также привести к более гладкой сходимости, так как градиент, вычисляемый на каждом шаге, усредняется по большему числу тренировочных примеров.

Обычно при обучении нейронных сетей используют за одну итерацию именно такой набор случайно выбранных элементов — **батч**. Градиентный спуск с использованием батчей называют также стохастическим градиентным спуском, хотя в некоторых источниках используют более точное mini-batch gradient descent (градиентный спуск по мини-батчам).

Введем еще один важный термин - **шаг обучения** (скорость обучения, **learning rate**). В формуле градиентного спуска w - это параметр модели, - градиент функции ошибки, а  - шаг обучения:

.

Шаг обучения отвечает за скорость изменения весов. Несмотря на то, что отрицательный градиент функции направлен в сторону наискорейшего спуска, это не значит, что следуя ему мы обязательно попадем в сторону глобального минимума. Во-первых, как на изображении под пунктом B вектор градиента может оказаться настолько большим, что попасть в глобальный минимум оказывается просто невозможным, либо, как на пункте A, вектор градиента может быть довольно маленьким и процесс обучения тогда будет очень долгим. Чтобы контролировать скорость обучения как раз и используют такой параметр как learning rate.



К сожалению, универсального шага обучения нет, и зачастую его приходится подбирать вручную методом проб и ошибок. Иногда слишком большой или маленький шаг обучения делают достижение глобального минимума невозможным, особенно, если модель попадает в локальный минимум. Чтобы немного автоматизировать этот процесс используются такие адаптивные методы оптимизации, основанные на стохастическом градиентном спуске, как Adagrad, Adadelta, RMSProp, Adam и другие. Иногда в процессе обучения исследователь может предположить, что на определенных итерациях шаг обучения нужно уменьшать определенным образом, для этого используется планировщик (scheduler).

# Способы создания нейросетей

Нейросети — это математические модели. Программированием на любом языке можно решать задачи, связанные с математикой. Однако встаёт вопрос, какой язык подойдёт для этого больше. Не считая учебных, нейросети, как правило, работают с большим количеством данных. Чтобы обучение нейросетей происходило с приемлемой скоростью, нужно использовать быстрый язык, например Си. Но такие языки обычно обладают низким уровнем абстракции, поэтому их сложнее изучать.

Для этих целей хорошо может подойти язык Python. С одной стороны, он имеет высокий уровень абстракции, с другой — операции с массивами данных могут сделать его библиотеки, написанные на Си. Этим способом мы пользовались первые 2 урока. Но если таким образом писать нейросети, это приведёт к повторяющемуся коду (поскольку их архитектуры остаются одинаковыми и зачастую меняются только с точки зрения параметров). Для того чтобы реализовывать архитектуры, может понадобиться их хорошее знание архитектур. Такая работа будет затруднительна для людей, не имеющих достаточной подготовки, а для профессионалов — наоборот, будет рутинной.

Существуют фреймворки для создания нейронных сетей (это, пожалуй, основной рабочий способ). Вот их неполный перечень:

* TensorFlow
* PyTorch
* Keras
* Microsoft Cognitive Toolkit (CNTK)
* Caffe
* Apache MXNet

Упрощение создания нейронных сетей на них не заканчивается. Также можно использовать инструменты, позволяющие создавать нейронные сети без навыков программирования, строя нейросети графически. Примеры: Neural Designer, Deep Learning Studio. Кроме того, существуют инструменты, самостоятельно создающие нейронные сети: AutoML инструменты. Самые популярные популярных из них:

* MLBox
* TPOT
* Autokeras

Перечисленные инструменты проранжированы в порядке возрастания уровня абстракции. Говоря о плюсах и минусах того или иного инструмента нужно в первую очередь понимать плюсы и минусы повышения уровня абстракции. Чем он выше, тем меньше его производительность и гибкость, и наоборот.

Наиболее востребованным в рабочих целях является уровень абстракции, который дают фреймворки — мы будем их изучать и использовать. Самый популярный фреймворк для создания нейросетей — TensorFlow. Самый популярный для обучения **—** Keras. Их мы изучим в этом и последующем уроке. Также стоит отметить, что фреймворки взаимосвязаны: Keras, как правило, работает поверх TensorFlow, а сам TensorFlow позволяет при необходимости пользоваться средствами Keras.

# Что такое Keras

Keras появился относительно недавно — в 2015 г. Но за это время он стал одним из самых популярных фреймоворков для создания нейросетей и фактически стандартом для использования его начинающими.

В чем причина его популярности? Keras позволяет создавать на высоком уровне абстракции — можно оперировать слоями, количеством нейронов в них, выбором функции активации и т.д. В то же время keras содержит инструментарий для всего того, что может понадобиться для работы, например ряд встроенных датасетов и возможность обрабатывать изображения.

В техническом плане Keras — это оболочка над инструментами меньшей степени абстракции, он может работать поверх TensorFlow, Microsoft Cognitive Toolkit, R, Theano, PlaidML.

Keras также пользуется на соревнованиях Kaggle. Однако стоит отметить, что в реальных проектах чаще используется TensorFlow, который мы будем изучать в следующих уроках. Keras, как и любой высокоабстрактный инструмент, имеет изъяны в качестве меньшей гибкости и производительности, чем тот же tensorflow.

Стоит также отметить, что Google официально поддерживает Keras (его автор, François Chollet, является сотрудником Google). TensorFlow сам, в свою очередь, позволяет использовать возможности Keras, т.е. в нём заложена возможность переходить на более высокой уровень абстракции.

В этом уроке мы рассмотрим пример обучения нейронной сети с помощью Keras. Но прежде изучим основы синтаксиса Keras и стандартные задачи, которые нужно выполнить при обучении нейронной сети.

# Основы синтаксиса

## Установка и работа с данными

Для начала необходимо установить keras. Надо полагать, вы хорошо знакомы с командой pip.

| sudo python3 pip install keras |
| --- |

Давайте попробуем получить датасет mnist и проанализировать его содержимое. Это ещё не будет синтаксис Keras, но мы рассмотрим часто встречающуюся задачу. Не обращайте внимание на предупреждения от TensorFlow: их часто бывает много и при необходимости их можно подавить.

| import mnist  # The first time you run this might be a bit slow, since the  # mnist package has to download and cache the data.  train\_images = mnist.train\_images()  train\_labels = mnist.train\_labels()  print(train\_images.shape) # (60000, 28, 28)  print(train\_labels.shape) # (60000,) |
| --- |

Что мы смогли узнать в данном случае? Что тренировочный датасет mnist состоит из 60 000 изображений 28 на 28 пикселей. Такие небольшие датасеты с маленькими изображениями встретятся и в других учебных датасетах.

Что нужно делать теперь? Если скачанный датасет не имеет разделения на тренировочный и тестовый, то поделить их. В нашем случае тренировочный датасет состоит из 60 000 изображений, тестовый — из 10 000, и они поделены по умолчанию.

Теперь нужно конвертировать значения пикселей из вида от 1 до 255 в набор значений от -0.5 до 0.5.

| import numpy as np  import mnist  train\_images = mnist.train\_images()  train\_labels = mnist.train\_labels()  test\_images = mnist.test\_images()  test\_labels = mnist.test\_labels()  *# Normalize the images.*  train\_images = (train\_images / 255) - 0.5  test\_images = (test\_images / 255) - 0.5  *# Flatten the images.*  train\_images = train\_images.reshape((-1, 784))  test\_images = test\_images.reshape((-1, 784))  print(train\_images.shape) *# (60000, 784)*  print(test\_images.shape) *# (10000, 784)* |
| --- |

## Создание модели

После первичной подготовки данных обычно следует создание модели нейронной сети, которая будет на них учиться.

Ниже типичный код учебной нейросети:

| # define the keras model  model = Sequential()  model.add(Dense(12, input\_dim=8, activation='relu'))  model.add(Dense(8, activation='relu'))  model.add(Dense(1, activation='sigmoid')) |
| --- |

Разберёмся с командами, которые встретились в этом коде.

* Sequential: позволяет создать нейросети, где слои имеют форму стека. Сигнал в них передаётся от одного слоя к другому. В противовес этой разновидности есть нейросети, где сигнал может передаваться не сразу, а попадать в цикл. Такие нейросети мы разберём в следующих уроках
* Dense: позволяет каждому нейрону быть связанным с другим. В противовес этому может быть необходимость не делать так много связей. Неполносвязные архитектуры также будут разобраны в курсе, они являются основой компьютерного зрения. Цифры 12, 8, 1 обозначают количество нейронов в каждом конкретном слое
* Activation: позволяет определить формулу, по которой будет активироваться нейрон

## Компиляция модели

На этапе компиляции создаётся модель с заданными ранее параметрами. Типичный учебный пример:

| # создание keras модели  model.compile(loss='binary\_crossentropy', optimizer='adam', metrics=['accuracy']) |
| --- |

На этой стадии необходимо сделать дополнительные настройки нейронной сети. Разберём команды из кода выше.

* loss: позволяет задать формулы, по которой будет определяться степень ошибки нейронной сети
* optimizer: позволяет задать алгоритм, который будет осуществлять изменения весов по всей нейронной сети (backpropagation)
* Metrics: позволяет определить критерии, по которым будет оцениваться степень обученности нейросети

## Передача данных для обучения нейросети

После того как нейросеть создана, можно передавать ей данные для обучения. Ниже типичный пример кода для этого:

| # передача обучающего датасета keras модели  model.fit(X, y, epochs=150, batch\_size=10, verbose=0) |
| --- |

Разберём команды из этого примера. X, y содержат все обучающие данные, epochs определяет, сколько раз весь набор данных должен пройти через нейросеть. bath\_size определяет количество обучающих примеров, передающихся нейросети на каждой итерации обучения. verbose позволяет определять информацию, которую вы видите во время обучения нейронной сети.

## Оценка обученности нейронной сети

Следующей стадией может быть проверка обученности нейронной сети. Команда Keras для этих целей:

| results = model.evaluate(x\_test, y\_test, batch\_size=128) |
| --- |

В данном случае мы просто указываем, какую модель на каких данных мы хотим проверить.

## Запуск нейронной сети для выполнения работы

На этой стадии попробуем запустить нейронную сеть на данных, которые мы бы хотели оценить с её помощью. Например, осуществить распознавание объекта на фотографии. Код для этих целей:

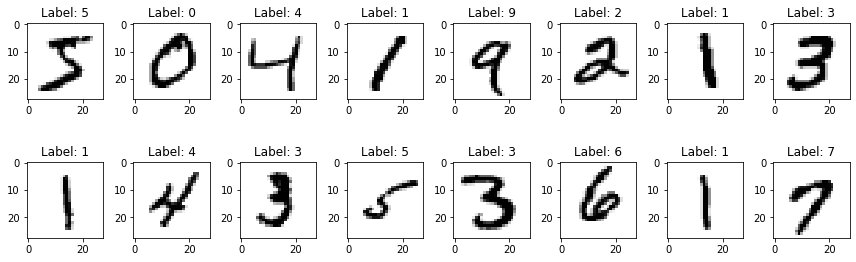
| predictions = model.predict(x\_test[:3]) |
| --- |

В качестве аргумента здесь указывается массив данных, содержащих, например, фотографию в виде массива чисел.

Мы рассмотрели основные стадии процесса обучения нейросети и команды Keras, в нём задействованные. Конечно, здесь приведён далеко не полный перечень возможностей Keras: она умеет сохранять созданную нейросеть, запускать уже имеющиеся, в ней есть различные средства для создания нейросетей разных архитектур и много другое. С чем-то из арсенала мы разберёмся по ходу курса, а с остальными функциями вы можете познакомиться на сайте Keras в разделе документация.

# Многоклассовая классификация

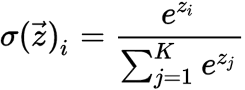
В следующем примере мы будем классифицировать рукописные цифры на 10 классов. Нейронная сеть будет угадывать для каждого изображения, какая цифра (от 0 до 9) на ней изображена.



На самом деле модель будет возвращать не класс, она будет возвращать 10 цифр, каждая из которых будет обозначать вероятность того, что входные данные принадлежат конкретному классу, т.е. что на входном изображении нарисована конкретная цифра.

Чтобы получить вероятность, на выходном слое должна находиться функция активации, которая принимает значения от 0 до 1. Обычно используют функцию сигмоиды (логистической функции).

Для многоклассовой классификации используют функцию Softmax, которая является обобщением сигмоиды для нескольких классов:

.

Функция преобразует вектор z размерности K в вектор 𝝈 той же размерности, где каждая координата полученного вектора представлена [числом](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D1%89%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) в интервале [0, 1] и сумма координат равна 1. Последнее свойство очень удобно, если рассматривать результаты функции, как вероятность.

В качестве функции потерь для задач классификации обычно используют перекрестную энтропию (кросс-энтропию):



Эта функция основана на понятии энтропии и на расстоянии Кульбака-Лейблера двух распределений. Неформально говоря, она подсчитывает, насколько распределение ответов нейронной сети q, похоже или непохоже на «истинное» распределение, задаваемое входными данными p.

Подробнее здесь: <https://towardsdatascience.com/deep-learning-concepts-part-1-ea0b14b234c8>

# Простая нейросеть на Keras

Попробуем сделать нейросеть на Keras, используя полученные выше знания. Обучим нейросеть различать рукописные цифры.

| *# The full neural network code!*  *###############################*  import numpy as np  import mnist  from keras.models import Sequential  from keras.layers import Dense  from keras.utils import to\_categorical  train\_images = mnist.train\_images()  train\_labels = mnist.train\_labels()  test\_images = mnist.test\_images()  test\_labels = mnist.test\_labels()  *# Normalize the images.*  train\_images = (train\_images / 255) - 0.5  test\_images = (test\_images / 255) - 0.5  *# Flatten the images.*  train\_images = train\_images.reshape((-1, 784))  test\_images = test\_images.reshape((-1, 784))  *# Build the model.*  model = Sequential([  Dense(64, activation='relu', input\_shape=(784,)),  Dense(64, activation='relu'),  Dense(10, activation='softmax'),  ])  *# Compile the model.*  model.compile(  optimizer='adam',  loss='categorical\_crossentropy',  metrics=['accuracy'],  )  *# Train the model.*  model.fit(  train\_images,  to\_categorical(train\_labels),  epochs=5,  batch\_size=32,  )  *# Evaluate the model.*  model.evaluate(  test\_images,  to\_categorical(test\_labels)  )  *# Save the model to disk.*  model.save\_weights('model.h5')  *# Load the model from disk later using:*  *# model.load\_weights('model.h5')*  *# Predict on the first 5 test images.*  predictions = model.predict(test\_images[:5])  *# Print our model's predictions.*  print(np.argmax(predictions, axis=1)) *# [7, 2, 1, 0, 4]*  *# Check our predictions against the ground truths.*  print(test\_labels[:5]) *# [7, 2, 1, 0, 4]* |
| --- |

# Дополнительные материалы

1. [Градиентный спуск: всё, что нужно знать](https://neurohive.io/ru/osnovy-data-science/gradient-descent/)
2. [Эпоха, батч, итерация — в чем различия?](https://neurohive.io/ru/osnovy-data-science/jepoha-razmer-batcha-iteracija/)
3. Софтмакс и кросс-энтропия: <https://habr.com/ru/post/484756/>
4. Keras: <https://habr.com/ru/post/482126/>
5. Оптимизаторы: <https://habr.com/ru/post/318970/>

# Используемые источники

1. Глубокое обучение — Николенко С. И., Кадурин 2018
2. [Википедия - стохастический градиентный спуск](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%BE%D1%85%D0%B0%D1%81%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B3%D1%80%D0%B0%D0%B4%D0%B8%D0%B5%D0%BD%D1%82%D0%BD%D1%8B%D0%B9_%D1%81%D0%BF%D1%83%D1%81%D0%BA)
3. <https://en.wikipedia.org/wiki/Learning_rate>
4. <https://pytorch.org/blog/overview-of-pytorch-autograd-engine/>
5. <https://keras.io/>
6. <https://ru.wikipedia.org/wiki/Softmax>
7. <https://towardsdatascience.com/deep-learning-concepts-part-1-ea0b14b234c8>
8. Шакла Н. — Машинное обучение и TensorFlow 2019