# **Введение**

# Нам известно, что нейронные сети хороши в распознавании изображений. Причём хорошая точность достигается и обычными сетями прямого распространения, однако, когда речь заходит про обработку изображений с большим числом пикселей, то число параметров для нейронной сети многократно увеличивается. Причём настолько, что время, затрачиваемое на их обучение, становится невообразимо большим.

# Так, если требуется работать с цветными изображениями размером 64х64, то для каждого нейрона первого слоя полносвязной сети потребуется 64·64·3 = 12288 параметров, а если сеть должна распознавать изображения 1000х1000, то входных параметров будет уже 3 млн! А помимо входного слоя есть и другие слои, на которых, зачастую, число нейронов превышает количество нейронов на входном слое, из-за чего 3 млн запросто превращаются в триллионы! Такое количество параметров просто невозможно рассчитать быстро ввиду недостаточно больших вычислительных мощностей компьютеров.

# Главной особенностью свёрточных сетей является то, что они работают именно с изображениями, а потому можно выделить особенности, свойственные именно им. Многослойные персептроны работают с векторами, а потому для них нет никакой разницы, находятся ли какие-то точки рядом или на противоположных концах, так как все точки равнозначны и считаются совершенно одинаковым образом. Изображения же обладают локальной связностью. Например, если речь идёт об изображениях человеческих лиц, то вполне логично ожидать, что точки основных частей лица будут рядом, а не разрозненно располагаться на изображении. Поэтому требовалось найти более эффективные алгоритмы для работы с изображениями и ими оказались свёрточные сети.

# **Описание модели**

Архитектура LeNet - это одна из самых первых сверточных нейронных сетей, которая подтолкнула область глубокого обучения. Эта новаторская работа Янна ЛеКуна была названа [LeNet](http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf) после многих предыдущих успешных итераций с 1988 года! Архитектура LeNet5 была фундаментальной, в частности понимание того, что функции изображения распределены по всему изображению, а свертки с изучаемыми параметрами являются эффективным способом извлечения аналогичных функций в нескольких местах с несколькими параметрами. В то время не было графических процессоров, которые могли бы помочь в обучении, и даже процессоры работали медленно. Следовательно, возможность сохранять параметры и вычисления была ключевым преимуществом. Это отличается от использования каждого пикселя в качестве отдельного входа большой многослойной нейронной сети. LeNet5 пояснил, что их не следует использовать на первом уровне, поскольку изображения имеют высокую пространственную корреляцию, и использование отдельного пикселя изображения в качестве отдельных входных функций не будет использовать эти корреляции.

Возможности LeNet5 можно суммировать как:

* Сверточная нейронная сеть использует последовательность из 3 слоев: свертка, объединение, нелинейность -> Это может быть ключевой особенностью глубокого обучения для изображений, начиная с этой статьи!
* использовать свертку для извлечения пространственных объектов
* подвыборка с использованием пространственного среднего карт
* нелинейность в виде tanh или сигмоидов
* многослойная нейронная сеть (MLP) в качестве окончательного классификатора
* скудная матрица соединений между слоями, чтобы избежать больших вычислительных затрат

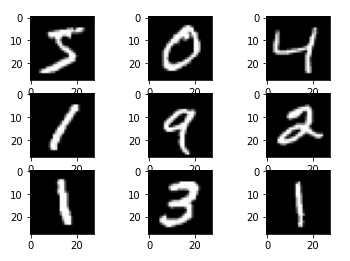
В целом эта сеть послужила источником многих современных архитектур и вдохновила многих людей в этой области.

Знаменитая сеть LeNet-5 может успешно классифицировать цифры, и применяется для распознавания цифр в чеках. Однако при более сложных задачах широта размаха и глубина сети растут, и становятся ограниченными вычислительными ресурсами и сдерживают производительность.

## **Датасет**

Набор MNIST — это большая коллекция рукописных цифр. Это очень популярный набор данных в области обработки изображений. Он часто используется для тестирования алгоритмов машинного обучения. MNIST — это сокращение от модифицированной базы данных Национального института стандартов и технологий. MNIST содержит коллекцию из 70 000 изображений 28 x 28  рукописных цифр от  0 до 9. Набор данных уже разделен на наборы для обучения и тестирования.

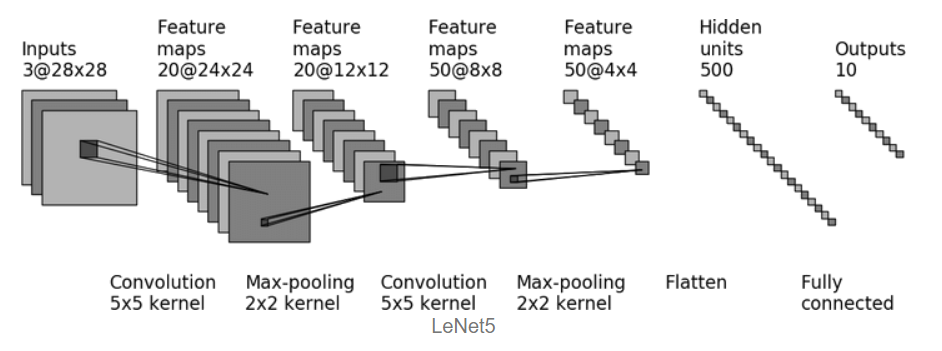
Пример данных:



## **Архитектура**

Свёрточные нейронные сети состоят из базовых блоков, благодаря чему их можно собирать как конструктор, добавляя слой за слоем и получая всё более мощные архитектуры. Основными блоками свёрточных нейронных сетей являются свёрточные слои, слои подвыборки (пулинга), слои активации и полносвязные слои.

LeNet5 – одна из первых свёрточных сетей, которая победила в ImageNet, состояла из 7 слоёв: слой свёртки, слой пулинга, ещё один слой свёртки ещё один слой пулинга и трёхслойная полносвязная нейронная сеть.



# Реализация модели

## Программные средства

Реализация модели в рамках данной работы производилась на облачной платформе Colab, на языке Python. Список используемых библиотек:

* Tensorflow версии 1.15.3 – для построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов
* NumPy – работа с векторами и массивами.

## **Код программы**

На вход этой сети подаётся изображение размером 32⨉32⨉1 пикселя. Изображение чёрно-белое, поэтому слой всего один. На схеме каждый convolutional и pooling слой считаются отдельными слоями.

Датасет MNIST скачивается при помощи tensorflow:

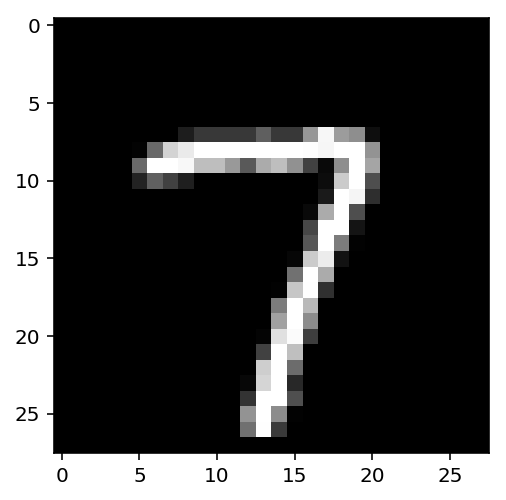
mnist = tf.contrib.learn.datasets.load\_dataset("mnist")train\_data = mnist.train.images  
train\_labels = np.asarray(mnist.train.labels, dtype=np.int32)eval\_data = mnist.test.images  
eval\_labels = np.asarray(mnist.test.labels, dtype=np.int32)

Для описания модели сети создана функция, которая соответствует интерфейсу Estimator API. Эта функция принимает в качестве параметров features, labels и model mode (одно из трёх значений: TRAIN, EVAL или PREDICT).

def cnn\_model\_fn(features, labels, mode):

Каждый пример представляет собой вектор, с shape (784,), то есть это картинка с разрешением 28⨉28 пикселей. Вот так выглядит пример с индексом 100:

plt.imshow(train\_data[100].reshape(28,28), cmap="gray")



Метод layers, использован для построения  convolutional  и pooling слоёв, ожидает на входе тензор c измерениями [batch\_size, image\_width, image\_height, channels]. У нас же матрица имеет размерность [batch\_size, image\_width \* image\_height].

Для того, чтобы переконвертировать размерность, используется метод reshape:

input\_layer = tf.reshape(features["x"], [-1,28,28,1])

Значение -1 указывает на то, что это измерение будет динамически рассчитано в зависимости от количества примеров features[“x”].

## **Convolutional layer 1 (C1)**

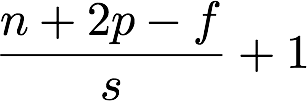
Для создания этого слоя использован метод conv2d модуля layers:

conv1 = tf.layers.conv2d(  
 inputs=input\_layer,  
 filters=6,  
 kernel\_size=[5,5],  
 padding='same',  
 activation=tf.nn.tanh)

Параметр inputs ожидает на вход тензор c размерностью [batch\_size, image\_width, image\_height, channels], соответственно ему передается input\_layers, подготовленный выше.

Параметр filters отвечает за количество фильтров, а kernel\_size — за их размер, который в этом случае равен [5,5]. При этом, если фильтр квадратный, то в качестве значения параметра можно просто указать скаляр 5.

Параметр [padding](https://medium.com/@congyuzhou/padding-32266fa95816) может быть same или valid. В данном случае используем same, потому как на исходной диаграмме показано, что на вход первого слоя подаётся картинка с разрешением 32⨉32⨉1, а на выходе получается тензор 28⨉28⨉6. Воспользовавшись формулой:



Согласно подсчетам, padding должен быть равен нулю, то есть значение параметра должно быть valid, размер фильтра 5⨉5, а stride 1. Однако, на входе картинка с разрешением 28⨉28⨉1. Для того, чтобы получить желаемый тензор 28⨉28⨉6, padding должен быть равен 2, то есть параметр должен быть same. В качестве значения параметра activation здесь указан tf.nn.tanh.

## **Pooling layer 1 (S2)**

Pooling layer создается при помощи метода max\_pooling2d()

pool1 = tf.layers.max\_pooling2d(  
 inputs=conv1,   
 pool\_size=[2,2],   
 strides=2)

Также, как и у conv2d(), параметр inputs на вход ожидает получить тензор c shape [batch\_size, image\_width, image\_height, channels]. Параметр pool\_size — это размер фильтра, в этом случае [2,2]. Параметр stride здесь равен 2. Сочетание pool\_size=[2,2] и stride=2 уменьшают длину и ширину матрицы вдвое, то есть на выходе получается тензор 14⨉14⨉6.

# **Convolutional layer 2 (C3) и Pooling layer 2 (S4)**

Для следующего свёрточного слоя используется 16 фильтров 5⨉5 и valid padding:

conv2 = tf.layers.conv2d(  
 inputs=pool1,  
 filters=16,  
 kernel\_size=[5,5],  
 padding='valid',  
 activation=tf.nn.tanh)

На выходе получается тензор 10⨉10⨉16. Pooling слой с параметрами, аналогичными предыдущему pooling слою, уменьшает длину и ширину матрицы вдвое:

pool2 = tf.layers.max\_pooling2d(  
 inputs=conv2,   
 pool\_size=[2,2],   
 strides=2)

На выходе получается тензор 5⨉5⨉16.

## **Fully connected layers**

Следующие два слоя — это обычные слои. Для того, чтобы их создать, нужно изменить размерность при помощи операции reshape:

pool2\_flat = tf.reshape(pool2, [-1, 16 \* 5 \* 5])

Получившийся двухмерный тензор с shape равным **[batch\_size, 400]**, передаётся на вход слою dense со 120 нейронами и активационной функцией tanh:

dense1 = tf.layers.dense(  
 inputs=pool2\_flat,   
 units=120,  
 activation=tf.nn.tanh)

и, затем, следующему слою, с 84 нейронами и той же активационной функцией tanh:

dense2 = tf.layers.dense(  
 inputs=dense1,   
 units=84,   
 activation=tf.nn.tanh)

## **Logits layer**

Так как у датасета 10 классов( цифры от 0 до 9), соответственно, последний слой сети будет иметь именно столько нейронов:

logits = tf.layers.dense(inputs=dense2, units=10)

## **Предсказание**

Последний слой возвращает данные о вероятности принадлежности класса в виде тензора, имеющего размер [batch\_size, 10]. Из этого тензора для каждого примера можно получить два значения: предсказанный класс и вероятность принадлежности к классу.

Для получения предсказанного класса, используется функция tf.argmax:

tf.argmax(input=logits, axis=1)

Её параметр inputs принимает на вход тензор, а параметр axis отвечает за то, в отношении какого измерения будет производиться вычисление.

Вероятность получается при помощи функции tf.nn.softmax:

tf.nn.softmax(logits, name="softmax\_tensor")

Оба предсказания обернуты в dictionary и, если функция была вызвана с параметром PREDICT, возвращается объект EstimatorSpec:

predictions = {  
 "classes": tf.argmax(input=logits, axis=1),  
 "probabilities": tf.nn.softmax(logits, name="softmax\_tensor")  
}if mode == tf.estimator.ModeKeys.PREDICT:  
 return tf.estimator.EstimatorSpec(  
 mode=mode,  
 predictions=predictions)

## **Вычисление потерь**

Для многоклассовой классификации обычно используется кросс энтропия, которая реализована в функции tf.losses.sparce\_softmax\_crossentropy:

loss = tf.losses.sparse\_softmax\_cross\_entropy(  
 labels=labels,logits=logits)

## **Настройка обучения**

В качестве оптимизационного алгоритма использован СГД с параметром α=0.001:

if mode == tf.estimator.ModeKeys.TRAIN:  
 optimizer = tf.train.GradientDescentOptimizer(  
 learning\_rate=0.001) train\_op = optimizer.minimize(  
 loss=loss,  
 global\_step=tf.train.get\_global\_step())return tf.estimator.EstimatorSpec(  
 mode=mode, loss=loss, train\_op=train\_op)

## **Метрики**

Для добавления accuracy метрики к модели создан dictionary:

eval\_metric\_ops = {  
 "accuracy": tf.metrics.accuracy(  
 labels=labels,  
 predictions=predictions["classes"])  
}return tf.estimator.EstimatorSpec(  
 mode=mode, loss=loss,eval\_metric\_ops=eval\_metric\_ops)

## **Создание объекта Estimator**

Теперь, когда модель готова, можно приступить к созданию объекта класса Estimator:

mnist\_classifier = tf.estimator.Estimator(  
 model\_fn=cnn\_model\_fn,  
 model\_dir='/tmp/lenet5')

Параметр model\_fn принимает на вход функцию, которая описывает модель и используется для обучения, тестирования и предсказания. Параметр model\_dir задаёт папку, в которой будут храниться промежуточные данные модели.

## **Logging hook**

Сеть может обучаться довольно долго идля обеспечения возможности видеть, что происходит на текущий момент использована LoggingTensorHook для логирования значений softmax слоя через каждые 50 итераций:

tensor\_to\_log = {"probabilities": "softmax\_tensor"}logging\_hook = tf.train.LoggingTensorHook(  
 tensors=tensor\_to\_log,  
 every\_n\_iter=50)

## **Обучение модели**

Для обучения модели, создана функция tf.estimator.inputs.numpy\_input\_fn:

train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
 x = {"x": train\_data},  
 y = train\_labels,  
 batch\_size = 100,  
 num\_epochs = None,  
 shuffle = True)

Параметры x и y — это обучающие данные и их классы. Параметр batch\_size отвечает за размер пакета данных для СГД. Параметр num\_epochs=None означает, что модель будет обучаться до тех пор, пока не будет достигнуто заданное количество шагов. Параметр shuffle=True указывает на то, что данные будут перемешаны. Затем нужно запустить процесс обучения, вызвав метод train у созданного ранее объекта Estimator, передав созданную функцию в качестве параметра input\_fn:

mnist\_classifier.train(  
 input\_fn = train\_input\_fn,  
 steps = 50000,  
 hooks = [logging\_hook])

Параметр steps отвечает, за количество шагов обучения.Созданный ранее logging\_hook передаём в качестве параметра hooks.

## **Оценка модели**

После того, как обучение завершилось, следует протестировать модель. Здесь, как и на этапе обучения, нужно создать тестировочную функцию и передать её в качестве параметра input\_fn, при вызове метода evaluate объекта Estimator:

eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(  
 x = {"x": eval\_data},  
 y = eval\_labels,  
 num\_epochs = 1,  
 shuffle = False)eval\_results = mnist\_classifier.evaluate(input\_fn=eval\_input\_fn)print(eval\_results)

Параметр num\_epochs = 1, указывает на то, что данные будут обработаны однократно. Параметр shuffle = False запрещает перемешивание данных. Модель, показывает 97 % accuracy на тестовых данных после 50000 шагов обучения:

{'accuracy': 0.9742, 'loss': 0.09026045, 'global\_step': 50000}

## **Заключение**

Таким образом, в рамках данной работы была исследована архитектура модели LaNet, проанализированы её сильные и слабые стороны. С использованием программных средств данная модель была с нуля реализована и протестирована на различных примерах изображений. Модель была проанализирована при помощи оценки качества прогнозирования разных уровнях. По результатам тестирования сделан вывод, что модель даёт прогнозы удовлетворительного уровня, несмотря на свою простую структуру. Тем не менее данная модель в настоящее время является устаревшей и нерелевантной.

# **Список литературы.**

1. <https://medium.com/@congyuzhou/lenet-5-своими-руками-b60ae3727cd3>
2. <https://programforyou.ru/poleznoe/convolutional-network-from-scratch-part-zero-introduction>
3. http://www.100byte.ru/python/MNIST\_NN/mnist\_nn.html

## Приложение

import numpy as np

import tensorflow as tf

tf.logging.set\_verbosity(tf.logging.INFO)

mnist = tf.contrib.learn.datasets.load\_dataset("mnist")

train\_data = mnist.train.images

train\_labels = np.asarray(mnist.train.labels, dtype=np.int32)

eval\_data = mnist.test.images

eval\_labels = np.asarray(mnist.test.labels, dtype=np.int32)

def cnn\_model\_fn(features, labels, mode):

    """Model function for CNN"""

    # Input layer

    input\_layer = tf.reshape(features["x"], [-1,28,28,1])

    # Convolutional Layer #1

    # Input 28x28x1

    # Output 28x28x6

    conv1 = tf.layers.conv2d(

        inputs=input\_layer,

        filters=6,

        kernel\_size=[5,5],

        padding='same',

        activation=tf.nn.tanh)

    # Pooling Layer #1

    # Output 14x14x6

    pool1 = tf.layers.max\_pooling2d(inputs=conv1, pool\_size=[2,2], strides=2)

    # Convolutional Layer #2

    # Output 10x10x16

    conv2 = tf.layers.conv2d(

        inputs=pool1,

        filters=16,

        kernel\_size=[5,5],

        padding='valid',

        activation=tf.nn.tanh)

    # Pooling Layer #2

    # Output 5x5x16

    pool2 = tf.layers.max\_pooling2d(inputs=conv2, pool\_size=[2,2], strides=2)

    # Dense layers

    pool2\_flat = tf.reshape(pool2, [-1, 16 \* 5 \* 5])

    dense1 = tf.layers.dense(

        inputs=pool2\_flat,

        units=120,

        activation=tf.nn.tanh)

    dense2 = tf.layers.dense(inputs=dense1, units=84, activation=tf.nn.tanh)

    # Logits layer

    logits = tf.layers.dense(inputs=dense2, units=10)

    predictions = {

        # Generate predictions (for both TRAIN and EVAL mode)

        "classes": tf.argmax(input=logits, axis=1),

        # Add 'softmax-tensor to the graph. It is used for PREDICT and by

        # 'logging\_hook'

        "probabilities": tf.nn.softmax(logits, name="softmax\_tensor")

    }

    if mode == tf.estimator.ModeKeys.PREDICT:

        return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

    # Calculate loss (for both TRAIN and EVAL modes)

    loss = tf.losses.sparse\_softmax\_cross\_entropy(labels=labels, logits=logits)

    # Configure the Training Op (for TRAIN mode)

    if mode == tf.estimator.ModeKeys.TRAIN:

        optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.001)

        train\_op = optimizer.minimize(

            loss=loss,

            global\_step=tf.train.get\_global\_step())

        return tf.estimator.EstimatorSpec(

            mode=mode,

            loss=loss,

            train\_op=train\_op)

    # Add evaluation metrics (for EVAL mode)

    eval\_metric\_ops = {

        "accuracy": tf.metrics.accuracy(

            labels=labels,

            predictions=predictions["classes"])}

    return tf.estimator.EstimatorSpec(

        mode=mode,

        loss=loss,

        eval\_metric\_ops=eval\_metric\_ops)

mnist\_classifier = tf.estimator.Estimator(

  model\_fn=cnn\_model\_fn,

  model\_dir='/tmp/lenet5')

    # Setup a logging hook for predictions

tensor\_to\_log = {"probabilities": "softmax\_tensor"}

logging\_hook = tf.train.LoggingTensorHook(

  tensors=tensor\_to\_log,

  every\_n\_iter=50)

    # Train the model

train\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(

  x = {"x": train\_data},

  y = train\_labels,

  batch\_size = 100,

  num\_epochs = None,

  shuffle = True)

mnist\_classifier.train(

  input\_fn = train\_input\_fn,

  steps = 50000,

  hooks = [logging\_hook])

    # Evaluate the model and print results

eval\_input\_fn = tf.estimator.inputs.numpy\_input\_fn(

  x = {"x": eval\_data},

  y = eval\_labels,

  num\_epochs = 1,

  shuffle = False)