Сверточный слой (convolutional layer)

Архитектура сверточных сетей предложена [Яном Лекуном](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%83%D0%BD,_%D0%AF%D0%BD) в 1988 году[[1]](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%91%D1%80%D1%82%D0%BE%D1%87%D0%BD%D0%B0%D1%8F_%D0%BD%D0%B5%D0%B9%D1%80%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C#cite_note-1). Предназначалась, главным образом, для анализа изображений.

Если взять для примера черно-белое изображение, то это некоторая матрица, например, размером 100х100х1 пикселей. Если бы изображение было цветным (RGB), то размерность была бы 100х100х3, т.е. фактически три картинки с цифрами отражающими градации каждого цвета.

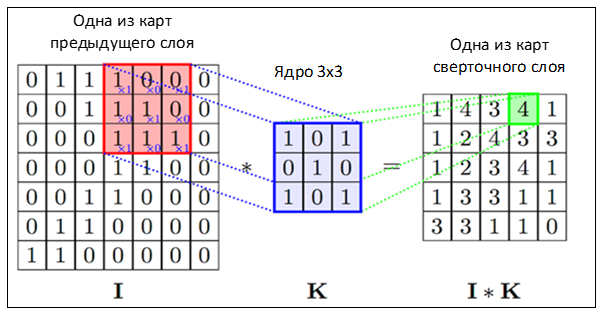
Дискретная свертка

Сверточный слой реализует операцию дискретной свертки, широко применяемый в цифровой обработке сигналов. При этой операции одна из функций (ядро свертки) сдвигается относительно другой и производится перемножение отсчетов и суммирование произведений.

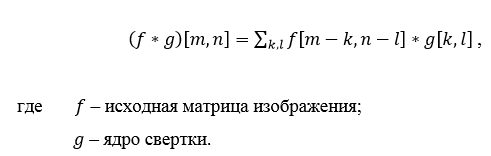
***Ядро свертки*** — это матрица определенной ширины kW и высоты kH. Чаще всего матрица квадратная, но в общем случае kW может быть не равно kH.

После операции свертки на выходе формируется ***выходная карта (матрица) признаков*** — ***output feature maps*** или просто ***карта***. Исходное изображение — это также матрица (карта), которая обрабатывается ядром свертки:

* Через ядро свертки, как через окно (фильтр), идет «просмотр» исходной матрицы.
* Элементы исходной матрицы перемножаются поэлементно на соответствующие значения в ядре свертки. Например, на рисунке ниже результаты перемножения указаны синим на первой карте: x1, x0.
* Полученные произведения суммируются. Например, после суммирования произведений на картинке в результате получилась цифра 4.
* Полученные значения — карта признаков, подаваемая на следующий слой.

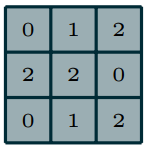


Математически операция дискретной свертки представляется в виде формулы:

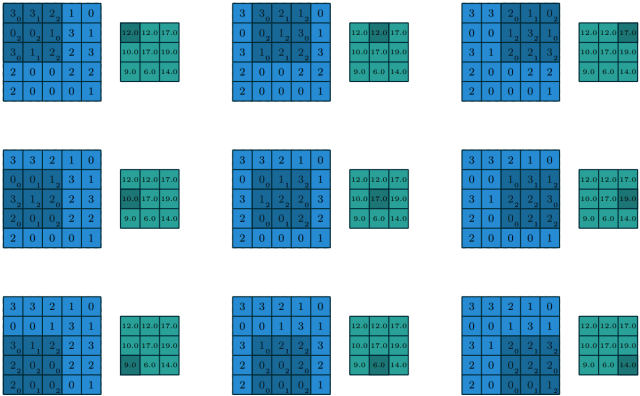


Чтобы на выходе получилась матрица, ядро свертки сдвигают на некоторый шаг (**stride**) по вертикали и горизонтали. Шаг может быть одинаковым для смещения по оси х и y, а может различаться.

Например, свертка матрицы с ядром свертки:



Дает следующие значения:



Пример вычисления дискретной свертки из[работы](https://arxiv.org/pdf/1603.07285.pdf).

При сдвиге ядра свертки произойдет уменьшение размера получаемой карты. Размерность выходной карты признаков будет определяться по следующей формуле.

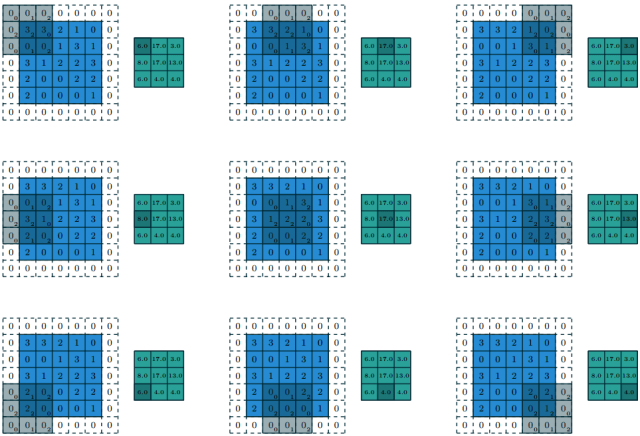
(Wout, Hout) = (Win — Wk + 1, Hin — Hin + 1), где

* (Wout, Hut) — вычисляемый размер сверточной карты;
* Win — ширина предыдущей карты;
* Hin — высота предыдущей карты;
* Wk — ширина ядра;
* Hk — высота ядра;

Например, если исходное изображение было 100х100 и по нему прошлись ядром свертки (5, 5), то на выходе будет матрица (100 — 5 + 1, 100 — 5 + 1) = (96, 96). Более общая формула следующая:

Padding = «same»

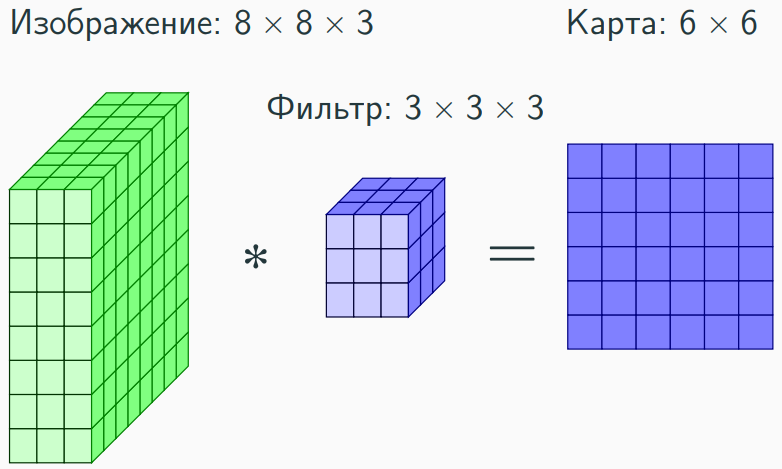
В некоторых случаях уменьшение выходной карты признаков неудобно. Например, нужно согласовывать размеры матриц. Чтобы размерность карты после свертки оставалась той-же, добавляют padding: матрицу обрамляют нулями (zero-padding).



Выполнение двумерной (N =2 ) дискретной свертки при s (strides) = 2, размере ядра свертки k(ernel) = 3х3, размере исходной матрицы i = 5х5, p(adding) = 1.

Ядро свертки скользит не с начала реальных значений входной матрицы, а с обрамления/набивки (padding-а), поэтому размер выходнйо карты признаков не уменьшается. На практике встречаются и другие варианты padding-а.

В случае если размерность входной матрицы, например, 3D, то ядро свертки будет представлять кубойд, который будет сдвигаться по ширине, высоте и глубине. При свертке исходной 3D матрицы c кубойдом на выходе получится двумерная карта признаков.

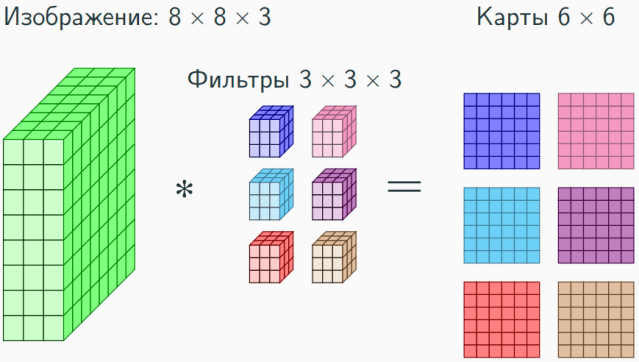


Размерность карты признаков на выходе 6 х 6 x 1, поскольку, S = 1, P = 0, Wk = 3, Hk = 3, Win = 3, Hin = 8:

* Сдвиг ядра 3 х 3 вдоль оси z (8 х 3) = 8 — 3 + 1 = 6.
* Сдвиг ядра 3 х 3 вдоль оси y (8 х 3) = 8 — 3 + 1 = 6.
* Сдвиг ядра 3 х 3 вдоль оси x (3 х 3) = 3 — 3 + 1 = 1.

У сверточного слоя есть параметр filters (F) — это количество уникальных фильтров (ядер свертки), которые применяются к входной карте для выполнения операции свертки.

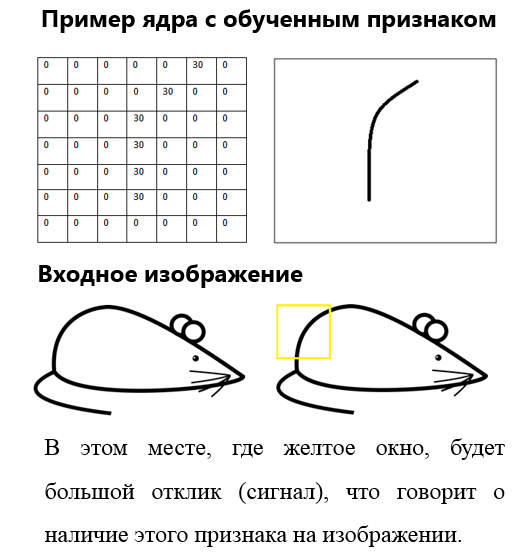
На выходе образуется несколько карт признаков по количеству фильтров (F). Каждый фильтр (ядро свертки) настраивается на выявление какой-то определенной фичи (признака) в исходной матрице.



Количество фильтров — гиперпараметр. Например, при 6 ядрах 3 х 3 и входной матрице 8 х 3 на выходе получится 6 двумерных карт признаков по числу фильтров, которые объединяются в тензор 6 х 6 х 6.

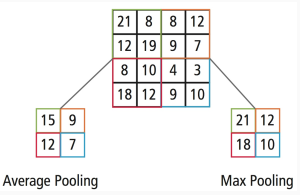
Веса ядра свертки фиксированны для фильтра. Оптимизатор подбирает параметры каждого фильтра так, чтобы минимизировать выбранный loss.

Например, после тренировки сети получили ядро (фильтр) 7х7, которое выявляет характерный изгиб на изображении мыши. Если на изображении такой изгиб присутствует, то при выполнении оперции дискретной свертки в определенном месте карты получится большое значение, если же признак не обнаружен, то значение будет небольшим.

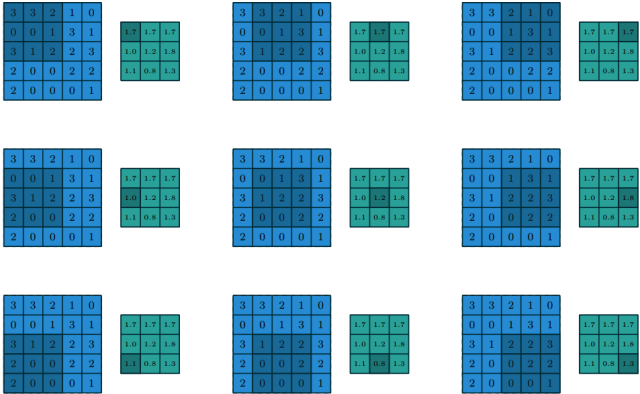


Pooling — это ещё один строительный элемент в построении CNN. Операция pooling уменьшает размер карты признаков, используя некоторую функцию для объединения ближайших элементов матрицы. В качестве функции может использоваться среднее арифметическое или максимум.

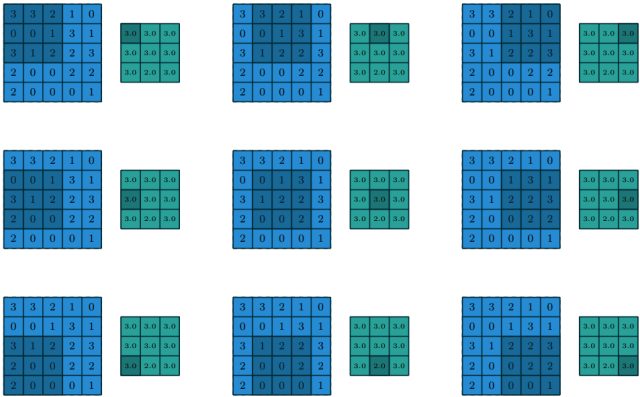
При выполнении операции pooling, как и в случае со сверткой, окно определенного размера скользит вдоль оси и над содержимым попавшем в окно выполняется pooling function.



Если рассмотреть эту операцию более подробно, то в чем-то она напоминает операцию дискретной свертки:



Выполнение операции 3 х 3 average pooling над матрицей 5 х 5 с шагом 1 х 1.



Выполнение операции 3 х 3 max pooling над матрицей 5 х 5 с шагом 1 х 1.

При pooling размерность выходной матрицы вычисляется по формуле:

* Wout = (Win — Wk)/S + 1
* Hout = (Hin — Hk)/S + 1

**ПРАКТИКА**

Как это работает:

* Сверточные слои извлекают различные признаки из изображений.
* Пулинг слои уменьшают размерность данных, сохраняя важные информационные признаки.
* Полносвязные слои выполняют классификацию на основе извлеченных признаков.
* Функции активации, такие как ReLU и softmax, вводят нелинейность и помогают модели учить сложные зависимости в данных.

Эта структура позволяет нейронной сети эффективно обучаться на изображениях и классифицировать их в соответствии с заданными классами благодаря комбинации сверточных и полносвязных слоев.

На прошлом занятии мы научились разрабатывать глубокие нейронные сети, которые способны классифицировать изображения элементов одежды из набора данных Fashion MNIST.

Результаты, которых мы добились в процессе работы над нейронной сетью, были впечатляющими — 88% точность классификации. И это в несколько строк кода (не учитывая код построения графиков и изображений)!

Но как нам сделать эту модель ещё лучше и точнее? Один из способов этого добиться — использовать свёрточные нейронные сети, сокращённо СНС. СНС показывают большую точность при решении задач классификации изображений, нежели стандартные полносвязные нейронные сети с которыми мы сталкивались на предыдущих занятиях. Именно по этой причине СНС стали такими популярными и именно благодаря им стал возможен технологический прорыв в области машинного зрения.  
  
На этом занятии мы узнаем, насколько легко разработать СНС-классификатор с нуля используя TensorFlow и Keras. Мы будем использовать тот же набор данных Fashion MNIST, который использовали на предыдущем занятии. В конце этого занятия мы сравним точность классификации элементов одежды предыдущей нейронной сети со свёрточной нейронной сетью из этого занятия.  
  
Перед тем как погружаться в разработку стоит немного углубиться в принцип работы свёрточных нейронных сетей.  
  
Две основные концепции в свёрточных нейронных сетях:

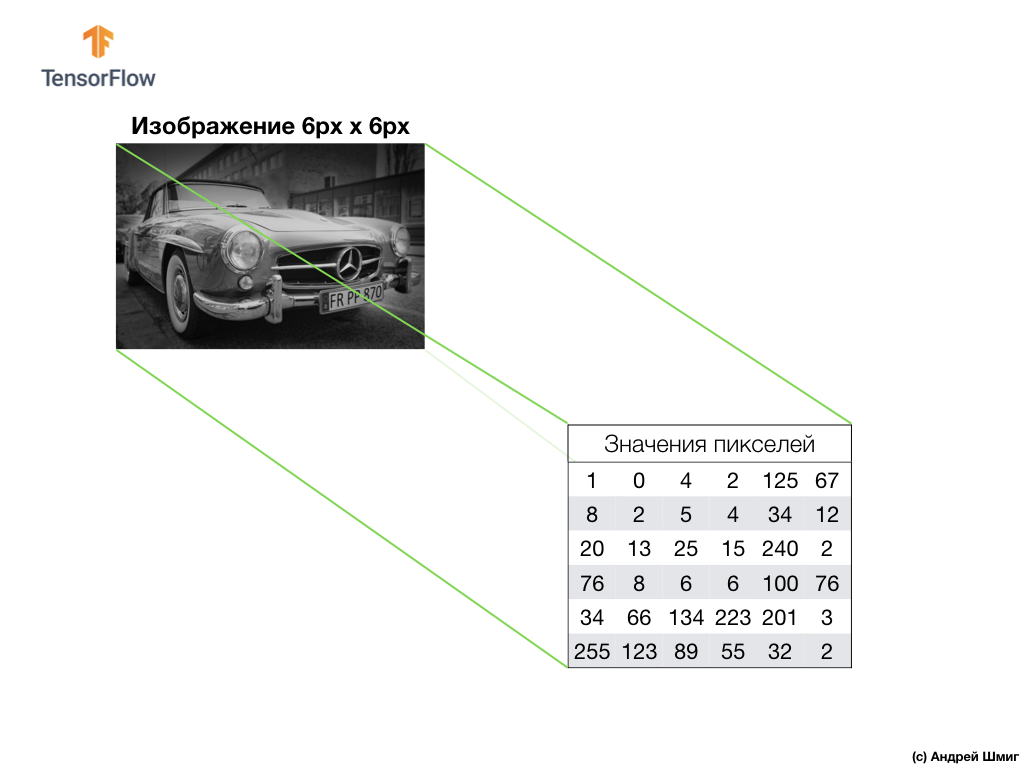
* свёртка
* операция подвыборки (pooling, max pooling)

Давайте рассмотрим их подробнее.

Свёртка

В этой части нашего занятия мы изучим технику, которая называется свёрткой. Давайте посмотрим, как же она работает.  
  
Возьмём изображение в оттенках серого и, для примера, представим, что его размеры 6 px в высоту и 6 px в ширину.

Наш компьютер интерпретирует изображение как двумерный массив пикселей. Так как наше изображение в оттенках серого, то значение каждого пикселя будет находиться в интервале от 0 до 255. 0 — черный, 255 — белый.  
  
На изображении ниже мы видим представление изображения 6px x 6px и соответствующие значения пикселей:



Как вы уже знаете, перед работой с изображениями значения пикселей необходимо нормализовать — привести значения к интервалу от 0 до 1. Однако в данном примере, для удобства пояснений, сохраним значения пикселей изображения и не будет их нормализовать.  
  
Суть свёртки заключается в создании другого набора значений, который называется ядром или фильтром. Пример можно увидеть на изображении ниже — матрица размером 3 х 3:

## Convolutional layers на Tensorflow

После подробного разбора теории создадим нейронную сеть на сверочных слоях. [Код к примеру.](https://colab.research.google.com/drive/1-aaePl6aUMz29HqwHDEgqImeTLBQFUCn) Отмечу, что, как правило, сверточные (conviolutional) слои используются в паре с полносвязными (dense), поскольку на выходе архитектуры нередко требуется выполнить классификацию. Поэтому сверточные слои соединяются с dense на выходе которых уже выдается, например, вероятность отнесения объекта к тому или иному классу.

Отмечу, что TensorFlow — штука достаточно капризная, поэтому может потребоваться откат к предыдущей версии Tensroflow на котором пример работает нормально.

!pip3 uninstall tensorflow-gpu==1.15.0

!pip3 install tensorflow-gpu==1.13.1

В Keras сверточный слой добавляется следующим образом:

model.add(Conv2D(32, (5,5)), где

F = 32 - количество фильтров,

Wk = 5, Hk = 5.

При (Win, Hin) = (100, 100) -> (Wout, Hout) = (96, 96)

Чтобы сохранить размерность выходной карты признаков:

model.add(Conv2D(32, (5,5), padding = 'same')

Можно указать шаг (stride):

model.add(Conv2D(32, (5,5), strides = 2, padding = 'same')

тогда

При (Win, Hin) = (100, 100) -> (Wout, Hout) = (50, 50)

* В Tensorflow перемещение ядра свертки по слоям возможно не только по оси x и y, как в Keras, но и z. Можно применять операцию свертки к определенным каналам по оси Z, например, брать каждый 3 слой.
* В Tensorflow можно указать применять сверточный слой к определенным batch-ам. Это может быть полезно при обработке видео.

Предположим, что на входе x — изображение 28 х 28 х1 , которое уже было преобразовано ранее reshape-ом в вектор 784 х 1. Поэтому нужно будет сделать обратное преобразование, либо использовать исходное изображение.

# Параметры сети

learningRate = 0.001

epochs = 10

batchSize = 200

dropout = 0.3

# Объявляем placeholders для тренировочного набора

# Размерность x = 28 x 28 пикселей = 784

x = tf.placeholder(tf.float32, [**None**, 784]) # None пока неизвестное количество batch-ей

# Трансформируем х в 4х мерный тензор [количество объектов в сете, ширина, высота, количество каналов]

xShaped = tf.reshape(x, [-1, 28, 28, 1]) # batchSize = -1, т.к. мы точно знаем другие параметры данного сета, так что

# tensorFlow автоматически распределит равномерно по batch-ам

# Объявляем выход placeholder, т.е. 10 цифр

y = tf.placeholder(tf.float32, [**None**, 10])

Зададим функцию для построения сверточного слоя:

* inputData — входные данные,
* numInputChannels — количество входных каналов. Например, для RGB изображения оно равно 3-м.
* numFilters — это количество фильтров (F),
* filterShape — размерность ядра свертки (Wk, Hk),
* poolShape — размерность ядра MaxPooling.

Размерность матрицы для весов сверточного слоя определяются — convFiltShape. Поскольку для слоя задано нектрое количество фильтров F x Wk x Hk — количество весов для одного слоя изображения. В случаеесли numInputChannels = 3, то полученую размерность нужно будет утроить, поскольку каждое ядро свертки (фильтр) будет применятся к каждому каналу в изображении.

Нейрон смещения соединится только с каждым фильтром, поэтому его размерность будет равна количеству фильтров.

Для сверточного слоя strides = [1, 1, 1, 1] — шаги по батчам и затем по трем каналам: x, y, z.

Для maxpooling strides = [1, 2, 2, 1] — шаги по батчам. Затем через два шага по x, y, т.е. размерность по x и y уменьшится в два раза. По оси z выбираются все каналы.

**def** newConvLayer(inputData, numInputChannels, numFilters, filterShape, poolShape, name):

# Параметры фильтра сверточного слоя

convFiltShape = [filterShape[0], filterShape[1], numInputChannels,

numFilters] #высота, ширина, количество входящих каналов, количество фильтров

# Инициализируем веса и отклонение для сверточного слоя

weights = tf.Variable(tf.truncated\_normal(convFiltShape, stddev=0.03), # нормальное распределение+усечение с точки насыщения sigmoid-функции,т.е. когда нейронка перестает учиться

name=name+'\_W')

bias = tf.Variable(tf.truncated\_normal([numFilters]), name=name+'\_b')

# Параметры сверточного слоя

outLayer = tf.nn.conv2d(inputData, weights, [1, 1, 1, 1], padding='SAME') # padding = SAME,

# т.е. формат [количество объектов в сете, ширина, высота, количество каналов] сохраняется

# Добавляем смещение

outLayer += bias

# Применяем функцию активации ReLU

outLayer = tf.nn.relu(outLayer)

# Применяем max pooling

ksize = [1, poolShape[0], poolShape[1], 1]

strides = [1, 2, 2, 1]

outLayer = tf.nn.max\_pool(outLayer, ksize=ksize, strides=strides,

padding='SAME')

**return** outLayer

С помощью этой функции создаем сверточные слои. У первого слоя:

* Поскольку у MNIST количество слоев в изображении 1, то первый параметр будет 1.
* 32 — количество фильтров (ядер свертки).
* [5, 5] — размерность ядра свертки.
* [2, 2] — размерность ядра maxpooling.

# Создаем сверточные слои, используя нашу функцию

layer1 = newConvLayer(xShaped, 1, 32, [5, 5], [2, 2], name='layer1')

На выходе из-за maxpooling размерность карты признаков с 28х28 уменьшилась до 14х14. У второго слоя:

* Поскольку на выходе первого сверточного слоя количество слоев равно количеству фильтров, то количество каналов будет 32.
* 64 — количество фильтров во втором слое.
* [5, 5] — размерность ядра свертки.
* [2, 2] — размерность ядра maxpooling.

layer2 = newConvLayer(layer1, 32, 64, [5, 5], [2, 2], name='layer2')

На выходе из-за maxpooling размерность карты признаков с 14 х 14 уменьшилась до 7 х 7 и количество фильтров 64, значит матрица будет размерностью 7 х 7 х 64. Для подачи на dense слой нужно сделать flatten. Аналога слоя Flatten в Tensorflow нет, поэтому используем обычный reshape:

flattened = tf.reshape(layer2, [-1, 7 \* 7 \* 64])

Добавляем два полносвязных слоя. Подробно про dense слои в Tensorflow я разбирал в [предыдущей статье](http://www.bizkit.ru/2020/02/22/16928/).

Первый слой на 1000 нейронов с активационной функцией Relu и слоем dropout.

# В первом слое задаем матрицу весов, bias(отклонение) и применяем функцию активации ReLU

Wd1 = tf.Variable(tf.truncated\_normal([7 \* 7 \* 64, 1000], stddev=0.03), name='Wd1') # матрица со значениями по нормальному распределению и стандартным отклонением 0.3

Bd1 = tf.Variable(tf.truncated\_normal([1000], stddev=0.01), name='Bd1') # вектор отклонений, 1000 - количество нейронов в скрытом слое

denseLayer1 = tf.matmul(flattened, Wd1) + Bd1

denseLayer1 = tf.nn.relu(denseLayer1)

denseLayer1 = tf.nn.dropout(denseLayer1, dropout)

Второй слой на 10 нейронов для классификации с активационной функцией softmax.

# Второй слой аналогичный, но 10 - количество нейронов не скрытого слоя, а выходного

Wd2 = tf.Variable(tf.truncated\_normal([1000, 10], stddev=0.03), name='Wd2')

Bd2 = tf.Variable(tf.truncated\_normal([10], stddev=0.01), name='Bd2')

denseLayer2 = tf.matmul(denseLayer1, Wd2) + Bd2

y\_ = tf.nn.softmax(denseLayer2)

Функция loss используется такая-же, как в [предыдущей статье](http://www.bizkit.ru/2020/02/22/16928/).

crossEntropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(logits=denseLayer2, labels=y))

В данном случае в качестве оптимизатора импользуется TensorFlow аналог Adam.

# Задаем оптимайзер

optimiser = tf.train.AdamOptimizer(learning\_rate=learningRate).minimize(crossEntropy)

Точность считается также, как в [предыдущей статье](http://www.bizkit.ru/2020/02/22/16928/):

# Определение accuracy

correctPrediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))

accuracy = tf.reduce\_mean(tf.cast(correctPrediction, tf.float32))

И запускаем тренировку сети:

# Запускаем сессию

sess = tf.Session()

sess.run(initOp)

totalBatch = int(len(mnist.train.labels) / batchSize)

**for** epoch **in** range(epochs):

avgLoss = 0 # сюда добавляем среднее значение

**for** i **in** range(totalBatch): # для каждого batch

batchX, batchY = mnist.train.next\_batch(batch\_size=batchSize)

\_, c = sess.run([optimiser, crossEntropy], # т.к. нам не нужно для каждой итерации запоминать вычисления непосредственно самого оптимизатора, то \_

# для функции потерь - присваеваем переменную c

feed\_dict={x: batchX, y: batchY})

avgLoss += c / totalBatch # вычисляем среднюю ошибку

testAcc = sess.run(accuracy,

feed\_dict={x: mnist.test.images, y: mnist.test.labels}) # подсчет точности

print("Epoch:", (epoch + 1), "Loss =", "{:.3f}".format(avgLoss), "test accuracy:"," {:.3f}".format(testAcc))

print("Точность:",sess.run(accuracy, feed\_dict={x: mnist.test.images, y: mnist.test.labels})\*100,"%")

sess.close()