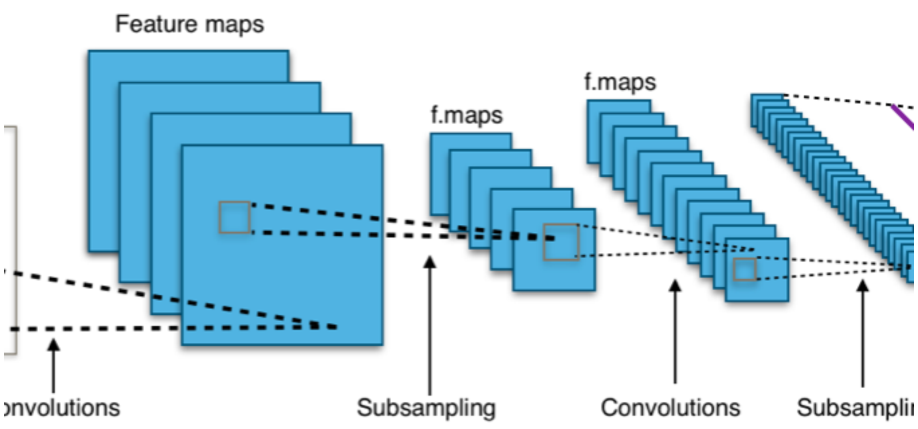
Convolutional Neural Networks Tutorial in TensorFlow

April 24, 2017 Andy Convolutional Neural Networks, Deep learning, TensorFlow 0



Full convolutional neural network

In the two previous tutorial posts, an introduction to neural networks and an introduction to TensorFlow, three layer neural networks were created and used to predict the MNIST dataset. They performed pretty well, with a successful prediction accuracy on the order of 97-98%. However, to take the next step in improving the accuracy of our networks, we need to delve into *deep learning*. A particularly useful type of deep learning neural network for image classification is the *convolutional neural network*. It should be noted that convolutional neural networks can also be used for applications other than images, such as time series prediction. However, this tutorial will concentrate on image classification only.

SEARCH …

CATEGORIES

Convolutional Neural Networks
Deep learning
Neural networks
Optimisation
TensorFlow

POPULAR TUTORIALS

Neural Networks Tutorial – A Pathway to Deep Learning
Python TensorFlow Tutorial – Build a Neural Network
Convolutional Neural Networks Tutorial in TensorFlow

This convolutional neural networks tutorial will introduce these networks by building them in TensorFlow. If you're not familiar with TensorFlow, I'd suggest checking out my previously mentioned tutorial, which is a gentle introduction. Otherwise, you're welcome to wing it.

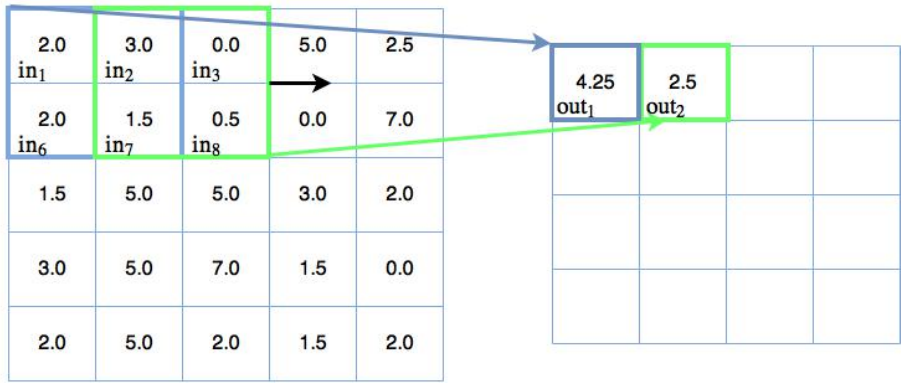
What's the problem?

As shown in the previous tutorials, multi-layer neural networks can perform pretty well in predicting things like digits in the MNIST dataset. This is especially true if we apply some improvements. So why do we need any other architecture? Well, first off – the MNIST dataset is quite simple. The images are small (only 28 x 28 pixels), are single layered (i.e. greyscale, rather than a coloured 3 layer RGB image) and include pretty simple shapes (digits only, no other objects). Once we start trying to classify things in more complicated colour images, such as buses, cars, trains etc. , we run into problems with our accuracy. What do we do?

Well, first, we can try to increase the number of layers in our neural network to make it *deeper*. That will increase the complexity of the network and allow us to model more complicated functions. However, it will come at a cost – the number of parameters (i.e. weights and biases) will rapidly increase. This makes the model more prone to overfitting and will prolong training times. In fact, learning such difficult problems can become intractable for normal neural networks. This leads us to a solution – convolutional neural networks.

What is a convolutional neural network?

The most commonly associated idea with convolutional neural networks is the idea of a *"moving filter"* which passes through the image. This moving filter, or convolution, applies to a certain neighbourhood of nodes (which may be the input nodes i.e. pixels) as shown below, where the filter applied is 0.5 x the node value:

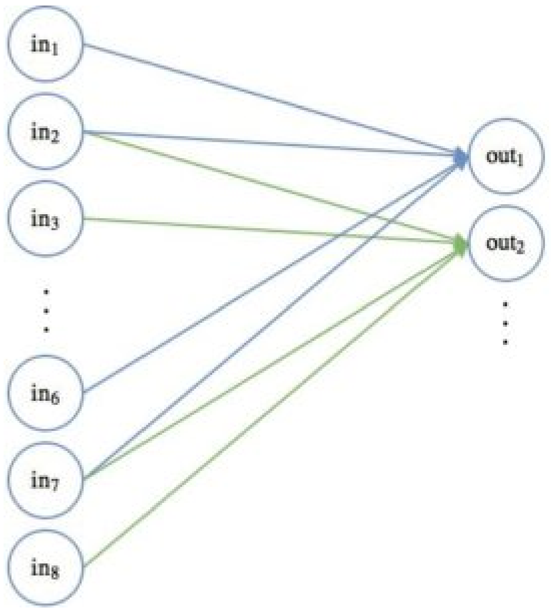


Moving 2×2 filter (all weights = 0.5)

As can be observed, only two outputs of the moving/convolutional filter have been shown – here we are mapping a 2×2 input square into a single output node. The weight of the mapping of each input square, as previously mentioned, is 0.5 across all four inputs. In other words, the following calculations were performed:

$$\begin{aligned} out_1 &= 0.5in_1 + 0.5in_2 + 0.5in_6 + 0.5in_7 \\ &= 0.5 \times 2.0 + 0.5 \times 3.0 + 0.5 \times 2.0 + 0.5 \times 1.5 \\ &= 4.25 \\ out_2 &= 0.5in_2 + 0.5in_3 + 0.5in_7 + 0.5in_8 \\ &= 0.5 \times 3.0 + 0.5 \times 0.0 + 0.5 \times 1.5 + 0.5 \times 0.5 \\ &= 2.5 \end{aligned}$$

In a convolution operation, this 2×2 moving filter would shuffle across each possible *x* and *y* co-ordinate combination to populate the output nodes. This operation can also be illustrated using our standard neural network node diagrams:



Moving 2×2 filter – node diagram

The first position of the moving filter connections is shown with the blue lines, the second (x + 1) is shown with the green lines. The weights of these connections, in this example, are all equal to 0.5.

A couple of things can be observed about this convolutional operation, in comparison to our previous understanding of standard neural networks:

- *Sparse* connections – notice that not every input node is connected to the output nodes. This is contrary to fully connected neural networks, where every node in one layer is connected to every node in the following layer.
- Constant filter parameters / weights – each filter has constant parameters. In other words, as the filter moves around the image the same weights are being applied. Each filter therefore performs a certain transformation across the whole image.   This is in contrast to fully connected neural networks, which have a different weight value for every connection
    - Note, I am not saying that each weight is constant *witihin* the filter, as in the example above (i.e. with weights [0.5, 0.5, 0.5, 0.5]). The weights *within* the filter could be any combination of values depending on how the filters are trained.

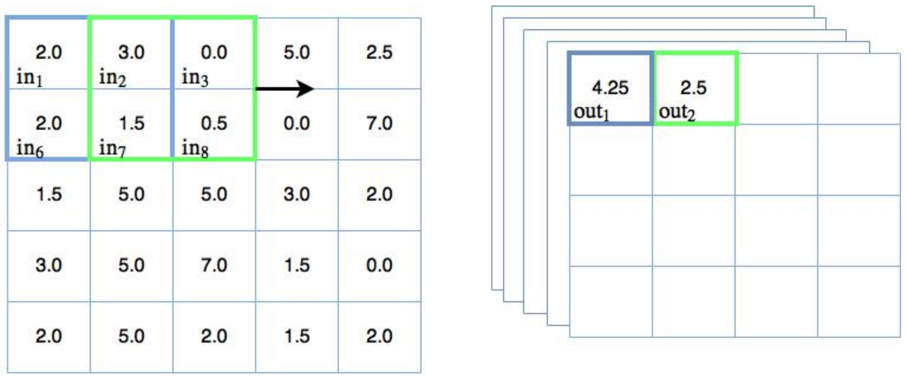
These two features of convolutional neural networks can significantly reduce the number of parameters required in the network, compared to fully connected neural networks.

The output of the convolutional mapping is then passed through some form of non-linear activation function, often the rectified linear unit activation function.

This step in convolutional neural networks is often called *feature mapping*. Before we move onto the next main feature of convolutional neural networks, *pooling*, it is worth saying a few things about this idea.

Feature mapping and multiple channels

Earlier I mentioned that the filter parameters i.e. the weights, are held constant as the filter moves through the input. This allows the filter to be trained to recognise certain features within the input data. In the case of images, it may learn to recognise shapes such as lines, edges and other distinctive shapes. This is why the convolution step is often called *feature mapping*. However, in order to classify well, at each convolutional stage we usually need multiple filters. So in reality, the moving filter diagram above looks like this:



Multiple convolutional filters

On the right you can now see stacked outputs, and that the separately trained filters each produce their own 2D output (for a 2D image). This is often referred to as having multiple *channels*. Each of these channels will end up being trained to detect certain key features in the image. Therefore, the output of the convolutional layer will actually be 3 dimensional (again, for a 2D image). If the input is itself multi-channelled, as in the case of a colour image with RGB layers, the output of the convolutional layer will be **4D**. Thankfully, as will be shown later, TensorFlow can handle all of this mapping quite easily.

Don't forget that the convolutional output for each node, over all the channels, are passed through an activation function.

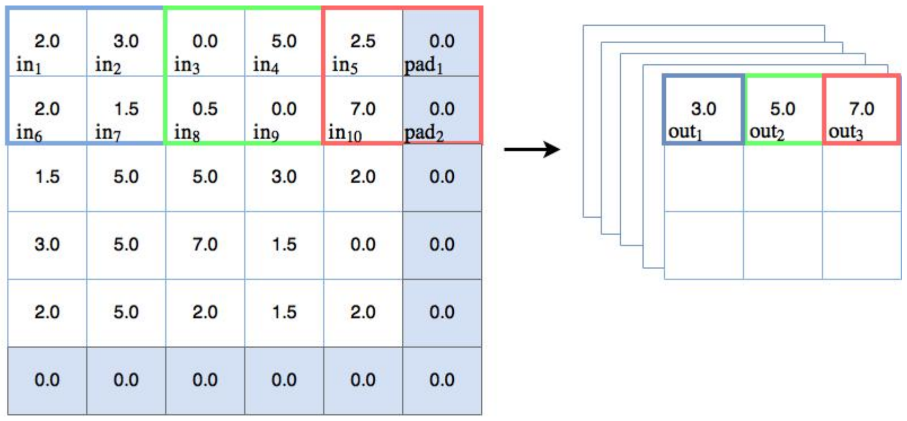
The next important part of convolutional neural networks is called *pooling.*

Pooling

The idea of pooling in convolutional neural networks is to do two things:

- Reduce the number of parameters in your network (pooling is also called “down-sampling” for this reason)
- To make feature detection more robust by making it more impervious to scale and orientation changes

So what is pooling? Again it is a “sliding window” type technique, but in this case, instead of applying weights the pooling applies some sort of statistical function over the values within the window. Most commonly, the function used is the max() function, so *max pooling* will take the maximum value within the window. The diagram below shows some max pooling in action:



Max pooling example (with padding)

We’ll go through a number of points relating to the diagram above:

Basic function

As can be observed in the diagram above, the different coloured boxes on the input nodes / squares represent a sliding 2×2 window. Max pooling is performed on the nodes within the sliding window i.e. the simple maximum is taken of the output of the nodes. In other words:

$$\begin{aligned} out_1 &= max(in_1, in_2, in_6, in_7) \\ out_2 &= max(in_3, in_4, in_8, in_9) \\ out_3 &= max(in_5, pad_1, in_{10}, pad_2) \end{aligned}$$

Strides and down-sampling

You may have noticed that in the convolutional / moving filter example above, the 2×2 filter moved only a single place in the *x* and *y* direction through the image / input. This led to an overlap of filter areas. This is called a *stride* of [1, 1] – that is, the filter moves 1 step in the *x* and *y* directions. With *max pooling*, the stride is usually set so that there is no overlap between the regions. In this case, we need a stride of 2 (or [2, 2]) to avoid overlap. This can be observed in the figure above when the *max pooling* box moves two steps in the *x* direction. Notice that having a stride of 2 actually reduces the dimensionality of the output. We have gone from a 5×5 input grid (ignoring the 0.0 padding for the moment) to a 3×3 output grid – this is called down-sampling, and can be used to reduce the number of parameters in the model.

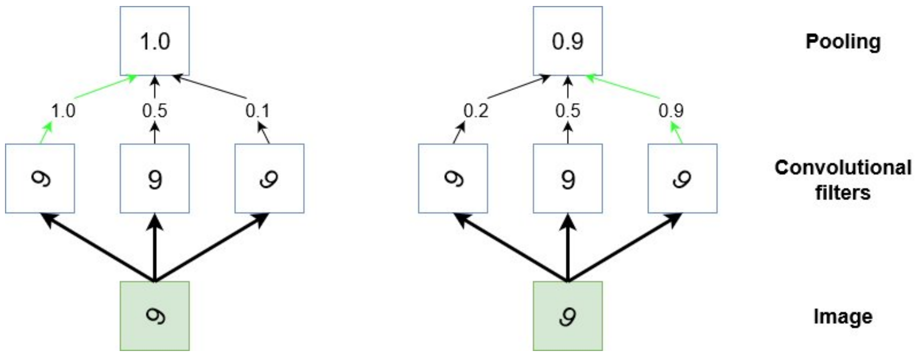
Padding

In the image above, you will notice the grey shaded boxes around the outside, all with 0.0 in the middle. These are padding nodes – dummy nodes that are introduced so that 2×2 max pooling filter can make 3 steps in the *x* and *y* directions with a stride of 2, despite there being only 5 nodes to traverse in either the *x* or *y* directions. Because the values are 0.0, with a rectified linear unit activation of the previous layer (which can't output a negative number), these nodes will never actually be selected in the max pooling process. TensorFlow has padding options which need to be considered, and these will be discussed later in the tutorial.

This covers how pooling works, but why is it included in convolutional neural networks?

Why is pooling used in convolutional neural networks?

In addition to the function of down-sampling, pooling is used in convolutional neural networks to make the detection of certain features in the input invariant to scale and orientation changes. Another way of thinking about what they do is that they *generalise* over lower level, more complex information. Consider the case where we have a number of convolutional filters that, during training, have learnt to detect the digit "9" in various orientations within the input images. In order for the convolutional neural network to learn to classify the appearance of "9" in the image correctly, it needs to activate in some way no matter what the orientation of the digit is (except when it looks like a “6” that is). That is what pooling can assist with, consider the diagram below:



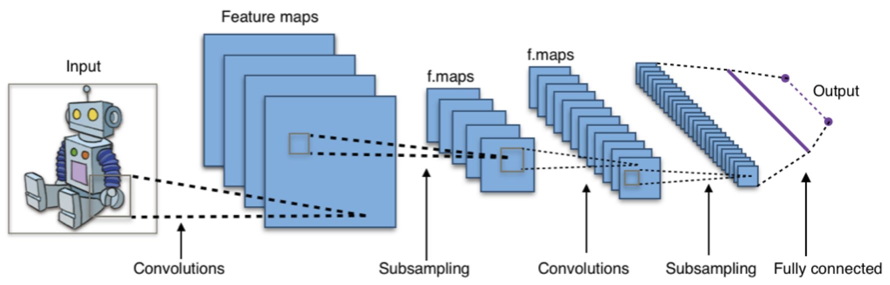
Stylised representation of pooling

The diagram above is a kind of stylised representation of the pooling operation. Consider a small region of an input image that has the digit “9” in it (green box). During training we have a few convolutional filters that have learnt to activate when they “see” a “9” shape in the image, but they activate most strongly depending on what orientation that “9” is. We want the convolutional neural network to recognise a “9” regardless of what orientation it is in. So the pooling “looks” over the output of these three filters and will give a high output so long as *any one* of these filters has a high activation.

Pooling acts as a *generaliser* of the lower level information and so enables us to move from high resolution *data* to lower resolution *information*. In other words, pooling coupled with convolutional filters attempt to detect *objects* within an image.

The final picture

The image below from Wikipedia shows the final image of a fully developed convolutional neural network:



Full convolutional neural network – By Aphex34 (Own work) [CC BY-SA 4.0], via Wikimedia Commons

Let's step through this image from left to right. First we have the input image of a robot. Then multiple convolutional filters (these would include rectified linear unit activations), followed by pooling / sub-sampling. Then we have another layer of convolution and pooling. Notice the number of channels (the stacked blue squares) and the reduction in the *x, y* sizes of each channel as the sub-sampling / down-sampling occurs in the pooling layers. Finally, we reach a fully connected layer before the output. This layer hasn't been mentioned yet, and deserves some discussion.

The fully connected layer

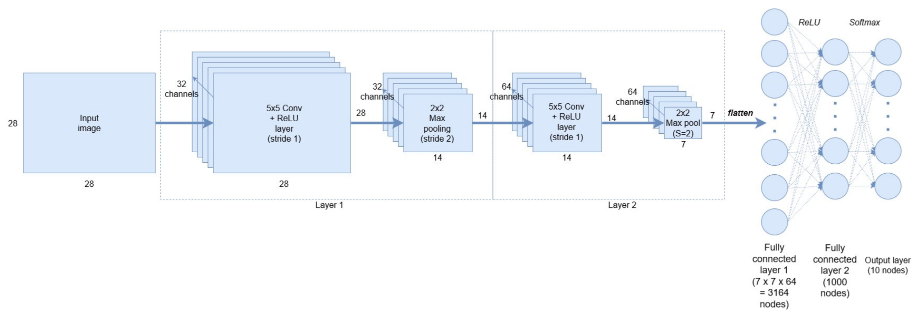
At the output of the convolutional-pooling layers we have moved from high resolution, low level *data* about the pixels to representations of *objects* within the image. The purpose of these final, fully connected layers is to make classifications regarding these objects – in other words, we bolt a **standard neural network classifier** onto the end of a trained object detector. As you can observe, the output of the final pooling layer is many channels of *x* x *y* matrices. To connect the output of the pooling layer to the fully connected layer, we need to *flatten* this output into a single (N x 1) tensor.

Let's say we have 100 channels of 2 x 2 pooling matrices. This means we need to flatten all of this data into a vector with one column and 2 x 2 x 100 = 400 rows. I'll show how we can do this in TensorFlow below.

Now we have covered the basics of how convolutional neural networks are structured and why they are created this way. It is now time to show how we implement such a network in TensorFlow.

A TensorFlow based convolutional neural network

TensorFlow makes it easy to create convolutional neural networks once you understand some of the nuances of the framework's handling of them. In this tutorial, we are going to create a convolutional neural network with the structure detailed in the image below. The network we are going to build will perform MNIST digit classification, as we have performed in previous tutorials (here and here). As usual, the full code for this tutorial can be found here.


28
Input image
28
32 channels
5x5 Conv + ReLU layer (stride 1)
28
28
32 channels
2x2 Max pooling (stride 2)
14
14
64 channels
5x5 Conv + ReLU layer (stride 1)
14
14
64 channels
2x2 Max pool (S=2)
7
7
flatten
Layer 1
Layer 2
ReLU
Softmax
Fully connected layer 1 (7 x 7 x 64 = 3164 nodes)
Fully connected layer 2 (1000 nodes)
Output layer (10 nodes)

Example convolutional neural network

As can be observed, we start with the MNIST 28×28 greyscale images of digits. We then create 32, 5×5 convolutional filters / channels plus ReLU (rectified linear unit) node activations. After this, we still have a height and width of 28 nodes. We then perform down-sampling by applying a 2×2 max pooling operation with a stride of 2. Layer 2 consists of the same structure, but now with 64 filters / channels and another stride-2 max pooling down-sample. We then flatten the output to get a fully connected layer with 3164 nodes, followed by another hidden layer of 1000 nodes. These layers will use ReLU node activations. Finally, we use a softmax classification layer to output the 10 digit probabilities.

Let’s step through the code.

Input data and placeholders

The code below sets up the input data and placeholders for the classifier.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import
input_data
mnist = input_data.read_data_sets("MNIST_data/",
one_hot=True)

# Python optimisation variables
learning_rate = 0.0001
epochs = 10
batch_size = 50

# declare the training data placeholders
# input x - for 28 x 28 pixels = 784 - this is the
flattened image data that is drawn from
# mnist.train.nextbatch()
x = tf.placeholder(tf.float32, [None, 784])
# dynamically reshape the input
x_shaped = tf.reshape(x, [-1, 28, 28, 1])
# now declare the output data placeholder - 10 digits
y = tf.placeholder(tf.float32, [None, 10])
```

TensorFlow has a handy loader for the MNIST data which is sorted out in the first couple of lines. After that we have some variable declarations which determine the optimisation behaviour (learning rate, batch size etc.). Next, we declare a placeholder (see this tutorial for explanations of placeholders) for the image input data, *x*. The image input data will be extracted using the mnist.train.nextbatch() function, which supplies a flattened 28×28=784 node, single channel greyscale representation of the image. However, before we can use this data in the TensorFlow convolution and pooling functions, such as conv2d() and max_pool() we need to reshape the data as these functions take 4D data only.

The format of the data to be supplied is [i, j, k, l] where *i* is the number of training samples, *j* is the height of the image, *k* is the weight and *l* is the channel number. Because we have a greyscale image, *l* will always be equal to 1 (if we had an RGB image, it would be equal to 3). The MNIST images are 28 x 28, so both *j* and *k* are equal to 28. When we reshape the input data *x* into *x_shaped*, theoretically we don't know the size of the first dimension of *x*, so we don't know what *i* is. However, tf.reshape() allows us to put -1 in place of *i* and it will dynamically reshape based on the number of training samples as the training is performed. So we use [-1, 28, 28, 1] for the second argument in tf.reshape().

Finally, we need a placeholder for our output training data, which is a [?, 10] sized tensor – where the 10 stands for the 10 possible digits to be classified. We will use the mnist.train.next_batch() to extract the digits labels as a one-hot vector – in other words, a digit of "3" will be represented as [0, 0, 0, 1, 0, 0, 0, 0, 0, 0].

Defining the convolution layers

Because we have to create a couple of convolutional layers, it's best to create a function to reduce repetition:

```
def create_new_conv_layer(input_data,
num_input_channels, num_filters, filter_shape,
pool_shape, name):
    # setup the filter input shape for tf.nn.conv_2d
    conv_filt_shape = [filter_shape[0], filter_shape[1],
num_input_channels,
                      num_filters]

    # initialise weights and bias for the filter
    weights =
tf.Variable(tf.truncated_normal(conv_filt_shape,
stddev=0.03),
                                      name=name+'_W')
    bias =
tf.Variable(tf.truncated_normal([num_filters]),
name=name+'_b')

    # setup the convolutional layer operation
    out_layer = tf.nn.conv2d(input_data, weights, [1, 1,
1, 1], padding='SAME')

    # add the bias
    out_layer += bias

    # apply a ReLU non-linear activation
    out_layer = tf.nn.relu(out_layer)

    # now perform max pooling
    ksize = [1, 2, 2, 1]
    strides = [1, 2, 2, 1]
    out_layer = tf.nn.max_pool(out_layer, ksize=ksize,
strides=strides,
                               padding='SAME')

    return out_layer
```

I'll step through each line/block of this function below:

```
conv_filt_shape = [filter_shape[0], filter_shape[1],
num_input_channels,
                      num_filters]
```

This line sets up a variable to hold the shape of the weights that determine the behaviour of the 5×5 convolutional filter. The format that the conv2d() function receives for the filter is: [filter_height, filter_width, in_channels, out_channels]. The height and width of the filter are provided in the filter_shape variables (in this case [5, 5]). The number of input channels, for the first convolutional layer is simply 1, which corresponds to the single channel greyscale MNIST image. However, for the second convolutional layer it takes the output of the first convolutional layer, which has a 32 channel output. Therefore, for the second convolutional layer, the input channels is 32. As defined in the block diagram above, the number of output channels of the first layer is 32, and for the second layer it is 64.

```
# initialise weights and bias for the filter
weights =
tf.Variable(tf.truncated_normal(conv_filt_shape,
stddev=0.03),
                                  name=name+'_W')
bias = tf.Variable(tf.truncated_normal([num_filters]),
name=name+'_b')
```

In these lines we create the weights and bias for the convolutional filter and randomly initialise the tensors. If you need to brush up on these concepts, check out this tutorial.

```
# setup the convolutional layer operation
out_layer = tf.nn.conv2d(input_data, weights, [1, 1, 1,
1], padding='SAME')
```

This line is where we setup the convolutional filter operation. The variable *input_data* is self-explanatory, as are the weights. The size of the weights tensor show TensorFlow what size the convolutional filter should be. The next argument [1, 1, 1, 1] is the *strides* parameter that is required in conv2d(). In this case, we want the filter to move in steps of 1 in both the *x* and *y* directions (or height and width directions). This information is conveyed in the strides[1] and strides[2] values – both equal to 1 in this case. The first and last values of *strides* are always equal to 1, if they were not, we would be moving the filter between training samples or between channels, which we don't want to do. The final parameter is the padding. Padding determines the output size of each channel and when it is set to "SAME" it produces dimensions of:

out_height = ceil(float(in_height) / float(strides[1]))
out_width = ceil(float(in_width) / float(strides[2]))

For the first convolutional layer, *in_height = in_width = 28*,
and *strides[1] = strides[2] = 1*. Therefore the padding of the input with 0.0 nodes will be arranged so that the *out_height = out_width = 28* – there will be no change in size of the output. This padding is to avoid the fact that, when traversing a *(x,y)* sized image or input with a convolutional filter of size *(n,m)*, with a stride of 1 the output would be *(x-n+1,y-m+1)*. So in this case, without padding, the output size would be (24,24). We want to keep the sizes of the outputs easy to track, so we chose the "SAME" option as the padding so we keep the same size.

```
# add the bias
out_layer += bias
# apply a ReLU non-linear activation
out_layer = tf.nn.relu(out_layer)
```

In the two lines above, we simply add a bias to the output of the convolutional filter, then apply a ReLU non-linear activation function.

```
# now perform max pooling
ksize = [1, pool_shape[0], pool_shape[1], 1]
strides = [1, 2, 2, 1]
out_layer = tf.nn.max_pool(out_layer, ksize=ksize,
strides=strides,
                               padding='SAME')
```

```
return out_layer
```

The max_pool() function takes a tensor as its first input over which to perform the pooling. The next two arguments *ksize* and *strides* define the operation of the pooling. Ignoring the first and last values of these vectors (which will always be set to 1), the middle values of *ksize* (*pool_shape[0]* and *pool_shape[1]*) define the shape of the max pooling window in the *x* and *y* directions. In this convolutional neural networks example, we are using a 2×2 max pooling window size. The same applies with the *strides* vector – because we want to down-sample, in this example we are choosing strides of size 2 in both the *x* and *y* directions (*strides[1]* and *strides[2]*). This will halve the input size of the (*x,y*) dimensions.

Finally, we have another example of a padding argument. The same rules apply for the 'SAME' option as for the convolutional function conv2d(). Namely:

out_height = ceil(float(in_height) / float(strides[1]))
out_width = ceil(float(in_width) / float(strides[2]))

Punching in values of 2 for *strides[1]* and *strides[2]* for the first convolutional layer we get an output size of (14, 14). This is a halving of the input size (28, 28), which is what we are looking for. Again, TensorFlow will organise the padding so that this output shape is what is achieved, which makes things nice and clean for us.

Finally we return the out_layer object, which is actually a sub-graph of its own, containing all the operations and weight variables within it. We create the two convolutional layers in the main program by calling the following commands:

```
# create some convolutional layers
layer1 = create_new_conv_layer(x_shaped, 1, 32, [5, 5],
[2, 2], name='layer1')
layer2 = create_new_conv_layer(layer1, 32, 64, [5, 5],
[2, 2], name='layer2')
```

As you can see, the input to *layer1* is the shaped input *x_shaped* and the input to *layer2* is the output of the first layer. Now we can move on to creating the fully connected layers.

The fully connected layers

As previously discussed, first we have to flatten out the output from the final convolutional layer. It is now a 7×7 grid of nodes with 64 channels, which equates to 3136 nodes per training sample. We can use tf.reshape() to do what we need:

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

Again, we have a dynamically calculated first dimension (the -1 above), corresponding to the number of input samples in the training batch. Next we setup the first fully connected layer:

```
# setup some weights and bias values for this layer,
then activate with ReLU
wd1 = tf.Variable(tf.truncated_normal([7 * 7 * 64,
1000], stddev=0.03), name='wd1')
bd1 = tf.Variable(tf.truncated_normal([1000],
stddev=0.01), name='bd1')
dense_layer1 = tf.matmul(flattened, wd1) + bd1
dense_layer1 = tf.nn.relu(dense_layer1)
```

If the above operations are unfamiliar to you, please check out my previous TensorFlow tutorial. Basically we are initialising the weights of the fully connected layer, multiplying them with the flattened convolutional output, then adding a bias. Finally, a ReLU activation is applied. The next layer is defined by:

```
# another layer with softmax activations
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')
dense_layer2 = tf.matmul(dense_layer1, wd2) + bd2
y_ = tf.nn.softmax(dense_layer2)
```

This layer connects to the output, and therefore we use a soft-max activation to produce the predicted output values *y_*. We have now defined the basic structure of our convolutional neural network. Let's now define the cost function.

The cross-entropy cost function

We could develop our own cross-entropy cost expression, as we did in the previous TensorFlow tutorial, based on the value *y_*. However, then we have to be careful about handling NaN values. Thankfully, TensorFlow provides a handy function which applies soft-max followed by cross-entropy loss:

```
cross_entropy =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(l
ogits=dense_layer2, labels=y))
```

The function *softmax_cross_entropy_with_logits()* takes two arguments – the first (*logits*) is the output of the matrix multiplication of the final layer (plus bias) and the second is the training target vector. The function first takes the soft-max of the matrix multiplication, then compares it to the training target using cross-entropy. The result is the cross-entropy calculation per training sample, so we need to reduce this tensor into a scalar (a single value). To do this we use *tf.reduce_mean()* which takes a mean of the tensor.

The training of the convolutional neural network

The following code is the remainder of what is required to train the network. It is a replication of what is explained in my previous TensorFlow tutorial, so please refer to that tutorial if anything is unclear. We'll be using mini-batches to train our network. The essential structure is:

- Create an optimiser
- Create correct prediction and accuracy evaluation operations
- Initialise the operations
- Determine the number of batch runs within an training epoch
- For each epoch:
  - For each batch:
    - Extract the batch data
    - Run the optimiser and cross-entropy operations
    - Add to the average cost
  - Calculate the current test accuracy
  - Print out some results
- Calculate the final test accuracy and print

The code to execute this is:

```
# add an optimiser
optimiser =
tf.train.AdamOptimizer(learning_rate=learning_rate).mini
mize(cross_entropy)

# define an accuracy assessment operation
correct_prediction = tf.equal(tf.argmax(y, 1),
tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,
tf.float32))

# setup the initialisation operator
init_op = tf.global_variables_initializer()

with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    total_batch = int(len(mnist.train.labels) /
batch_size)
    for epoch in range(epochs):
        avg_cost = 0
        for i in range(total_batch):
            batch_x, batch_y =
mnist.train.next_batch(batch_size=batch_size)
            _, c = sess.run([optimiser, cross_entropy],
                            feed_dict={x: batch_x, y:
batch_y})
            avg_cost += c / total_batch
        test_acc = sess.run(accuracy,
                       feed_dict={x: mnist.test.images,
y: mnist.test.labels})
        print("Epoch:", (epoch + 1), "cost =", "
{:.3f}".format(avg_cost), "
                 test accuracy:
{:.3f}".format(test_acc))

    print("\nTraining complete!")
    print(sess.run(accuracy, feed_dict={x:
mnist.test.images, y: mnist.test.labels}))
```

The final code can be found on this site's GitHub repository. Note the final code on that repository contains some TensorBoard visualisation operations, which have not been covered in this tutorial and will have a dedicated future article to explain.

Caution: This is a relatively large network and on a standard home computer is likely to take at least 10-20 minutes to run.

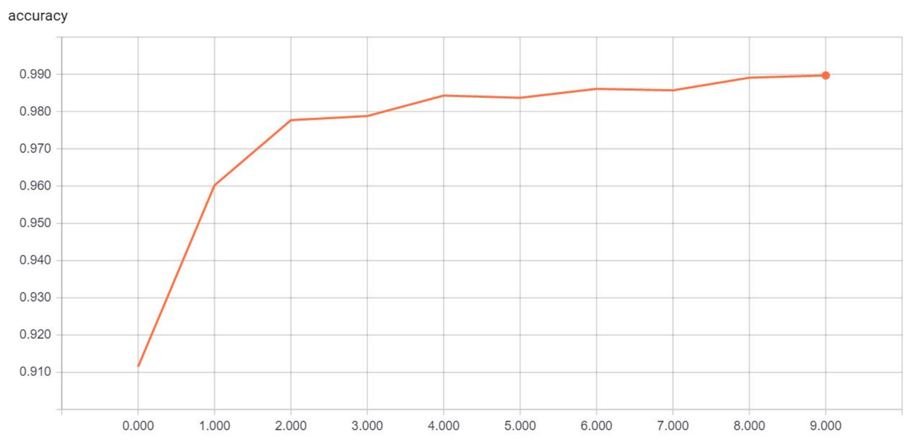
The results

Running the above code will give the following output:

```
Epoch: 1 cost = 0.739  test accuracy: 0.911
Epoch: 2 cost = 0.169  test accuracy: 0.960
Epoch: 3 cost = 0.100  test accuracy: 0.978
Epoch: 4 cost = 0.074  test accuracy: 0.979
Epoch: 5 cost = 0.057  test accuracy: 0.984
Epoch: 6 cost = 0.047  test accuracy: 0.984
Epoch: 7 cost = 0.040  test accuracy: 0.986
Epoch: 8 cost = 0.034  test accuracy: 0.986
Epoch: 9 cost = 0.029  test accuracy: 0.989
Epoch: 10 cost = 0.025  test accuracy: 0.990

Training complete!
0.9897
```

We can also plot the test accuracy versus the number of epoch's using TensorBoard (TensorFlow's visualisation suite):



Convolutional neural network MNIST accuracy

As can be observed, after 10 epochs we have reached an impressive prediction accuracy of 99%. This result has been achieved without extensive optimisation of the convolutional neural network's parameters, and also without any form of regularisation. This is compared to the best accuracy we could achieve in our standard neural network ~98% – as can be observed in the previous tutorial.

The accuracy difference will be even more prominent when comparing standard neural networks with convolutional neural networks on more complicated data-sets, like the CIFAR data. However, that is a topic for another day. Have fun using TensorFlow and convolutional neural networks!

« PREVIOUS
Python TensorFlow Tutorial – Build a Neural Network