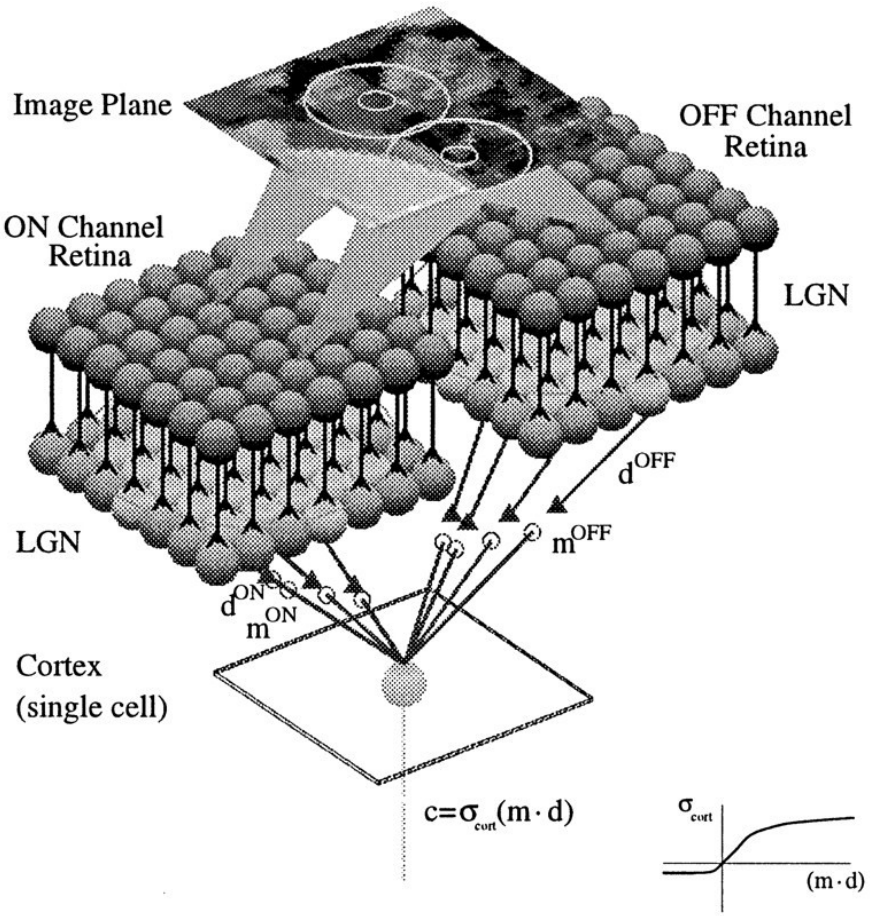
Stephen Barter Follow
Explorer of data and algorithmic alchemist.
Sep 16 · 10 min read

Convolutional Neural Net in Tensorflow

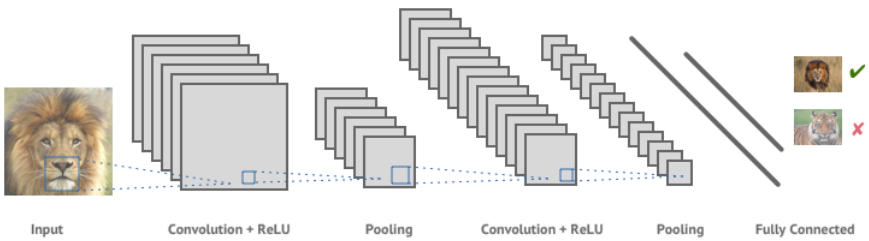
One of the most exciting areas of deep learning is computer vision. Through recent advances in convolutional neural nets we have been able to create self driving cars, facial detection systems and automated medical imagery analysis that out performs specialists just to name a few. In this article I will show you the fundamentals of convolutional neural nets and how you can create one yourself to classify hand written digits.



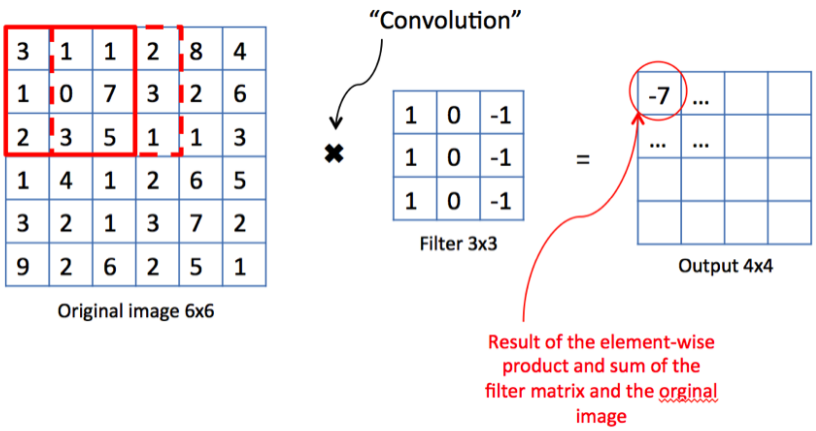
Unlike many fields of deep learning which are hyped to the public to seem like they are replications of biological functions in the human brain, convolutional neural nets come very close. Back in 1959, David Hubel and Torsten Wiesel conducted expirements on cats and monkeys

which gave important revelations of how the visual cortex functions. What they found was that many neurons have a small local receptive which only react to small finite areas of the total visual field. They showed that certain neurons react to low level patterns such as horizontal lines, vertical lines and others rounded. They also recognized that other neurons have larger receptive fields and are stimulated by more complex patterns which are combinations of information gathered by the lower level neurons. These findings laid the foundation for what we now call convolutional neural nets. Let's break down the building blocks one by one.

1. Convolutional Layer.

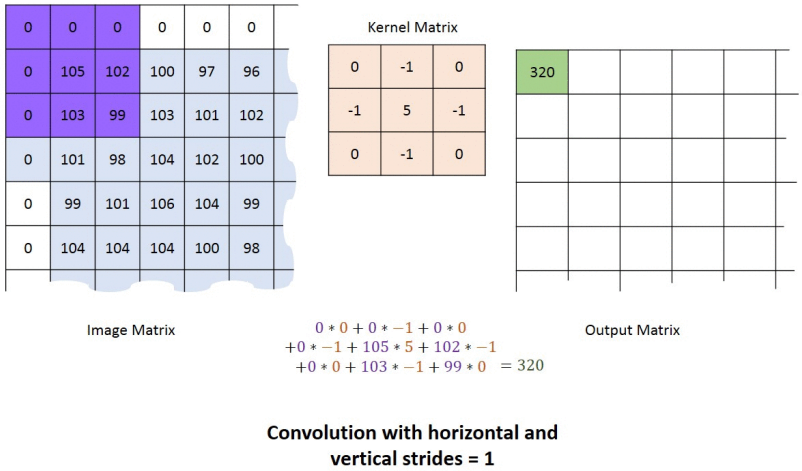


Each convolutional layer is composed of several feature maps. Each feature map has a unique filter composition which detect features such as horizontal lines or vertical lines. You can picture each filter like a window that slides over the dimensions of an image and detecting properties as it goes. The amount the of pixels the filter slides across the image is called the stride. A stride of 1 means the filter moves over one pixel at a time where as a a stride of 2 would skip 2 pixels forward.



In the example above, we have a vertical line detector. The original image is a 6x6, being scanned by with a 3x3 filter with a stride of 1 which results in a 4x4 dimension output. The filter is only interested in

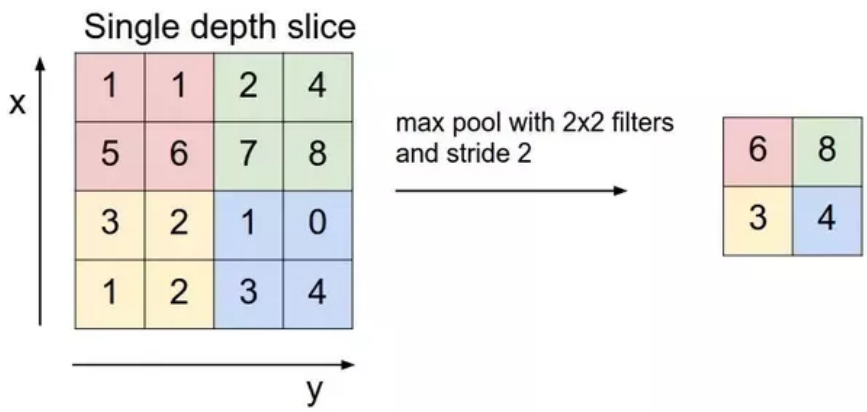
the sections in the left and right columns of its field of view. By summation and multiplying the inputs of the image by the configuration of the 3x3 filter we have 3+1+2 -1- 7-5= -7. The filter then moves to the right one stride which would then calculate 1+0+3 -2–3–1 = -2. -2 Would then go in the spot to the right of the -7. This process would continue until the 4x4 grid is complete. Afterwards, the next feature map will calculate its own values using a unique filter/kernel matrix of it's own.



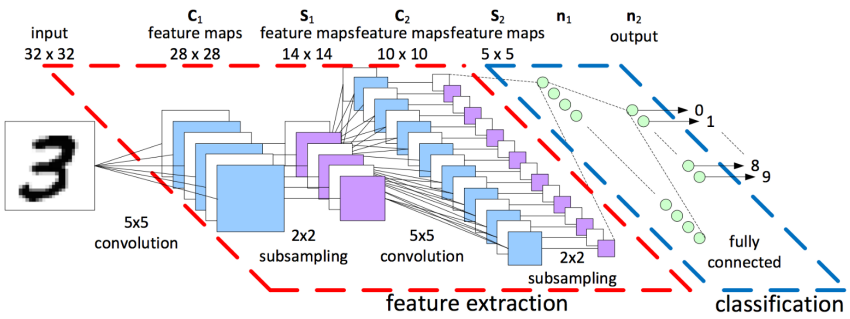
Convolution with horizontal and vertical strides = 1

2. Pooling Layers

The goal of the pooling layer is to further reduce dimensionality by aggregating the values collected by the convolutional layer or what's called sub-sampling. This will reduce computational load in addition to providing some regularization to your model to avoid over fitting. They follow the same sliding window idea as the conv layer but rather than calculate all values, they pick the max or average of it's inputs. This is called max pooling and average pooling respectively.



These 2 components are the key building blocks of a convolution layer. You then would typically repeat this recipe further reducing the dimensions of your feature maps, though increasing their depth. Each feature map will specialize in recognizing it's own unique shapes. At the end of the convolutions you will place a fully connected layer/layers with an activation function such as Relu or Selu which is used to reshape the dimensions into a vector suitable to feed into your classifier. For example if your final conv layer outputs a 3x3x128 matrix but you are only predicting 10 different classes, you will want to reshape that into a 1x1152 vector and gradually reduce its size before feeding to your classifier. The fully connected layers will also learn their own functions as in a typical deep neural network.



Now let's see a implementation in Tensorflow on the MNIST handwritten digit dataset. First we will load our libraries. Using fetch_mldata from sklearn we load the mnist dataset and assign the images and labels to the X and y variables. Then we will create our train/test sets. Lastly, we will plot a few examples to get an idea of the task ahead.

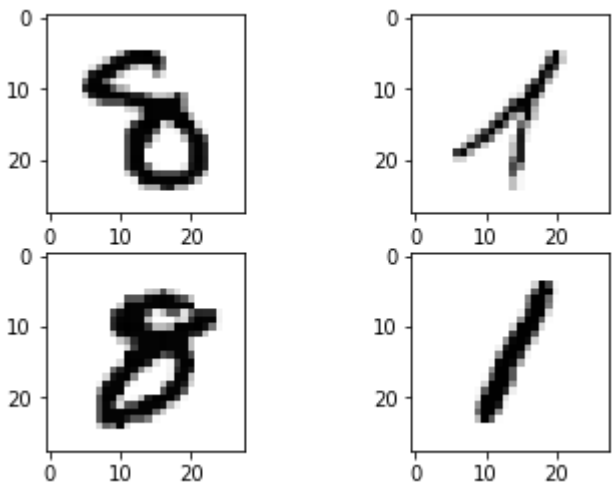
In [1]:
```
from sklearn.datasets import fetch_mldata
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [61]:
```
mnist = fetch_mldata('MNIST original')
```

In [62]:
```
X = mnist.data
y = mnist.target
```

In [5]:
```
m = len(X)
perm = np.random.permutation(m)
x_train, x_test = X[perm][10000:], X[perm][:10000]
y_train, y_test = y[perm][10000:], y[perm][:10000]

x_train = x_train/255
x_test = x_test/255
```

In [81]:
```
f, ax = plt.subplots(2,2)
ax[0, 0].imshow(X[50000].reshape(28,28), cmap = 'Greys')
ax[0, 1].imshow(X[61199].reshape(28,28), cmap = 'Greys')
ax[1, 0].imshow(X[50420].reshape(28,28), cmap = 'Greys')
ax[1, 1].imshow(X[61933].reshape(28,28), cmap = 'Greys')
plt.show()

```



As you can see there are some challenging examples. Both on the right side are 1's.

Next we will do some data augmentation which is a sure way to improve your models performance. By creating slight variations of the training images you in effect create regularization for your model. We will use Scipy's ndimage module to shift our images by 1 pixel right, left, up and down. Not only does this provide a wider variety of examples it will increase the size of our training set considerably which is usually always a good thing.

```
In [6]:
from scipy.ndimage.interpolation import shift
```

```
In [7]:
def shift_image(image, dx, dy):
    image = image.reshape((28,28))
    shifted_image = shift(image, [dx, dy], cval = 0, mode = 'constant')
    return shifted_image.reshape([-1])
```

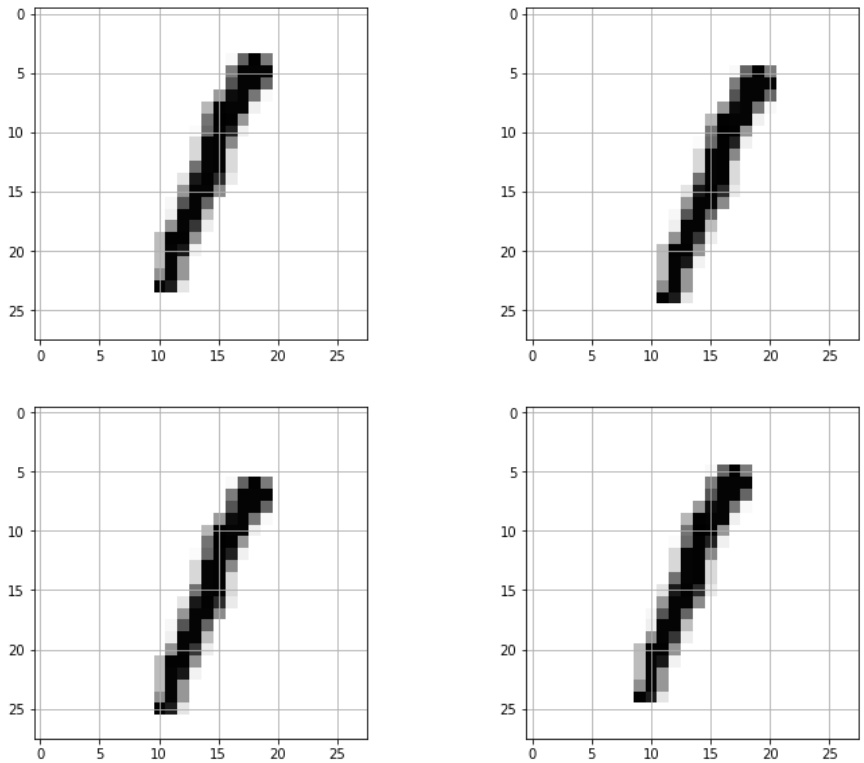
```
In [8]:
x_train_shift = []
y_train_augmented = []

for dx, dy in ((1,0), (-1,0), (0,1), (0, -1)):
    for image, label in zip(x_train, y_train):
        x_train_shift.append(shift_image(image, dx, dy))
        y_train_augmented.append(label)
```

```
In [9]:
x_train_shift = np.array(x_train_shift)
y_train_augmented = np.array(y_train_augmented)
```

```
In [10]:
x_train_shift.shape, y_train_augmented.shape
```

```
Out[10]: ((240000, 784), (240000,))
```

```
In [12]:
f, ax = plt.subplots(2,2, figsize = (12, 10))
ax[0,0].imshow(x_train_shift[60001].reshape(28,28), cmap = 'Greys')
ax[1,0].imshow(x_train_shift[1].reshape(28,28), cmap = 'Greys')
ax[0,1].imshow(x_train_shift[120001].reshape(28,28), cmap = 'Greys')
ax[1,1].imshow(x_train_shift[180001].reshape(28,28), cmap = 'Greys')
ax[0,0].grid()
ax[1,0].grid()
ax[0,1].grid()
ax[1,1].grid()
```



The last form of data augmentation I'll show you will be to create horizontal flips of the images using the cv2 library. We will also need to create new labels for these flipped images which is as easy as duplicating the original labels.

```
In [13]:
import cv2
```

```
In [14]:
def horizontal_flip(images):
    flipped_images = []
    for img in images:
        flipped_img = cv2.flip(img, flipCode = 1)
        flipped_images.append(flipped_img)
    return (flipped_images)
```

```
In [15]:
flipped_imgs = horizontal_flip(x_train.reshape(-1, 28,28))
```

```
In [16]:
flipped_imgs = np.array(flipped_imgs)
flipped_labels = np.array(y_train[:])
```

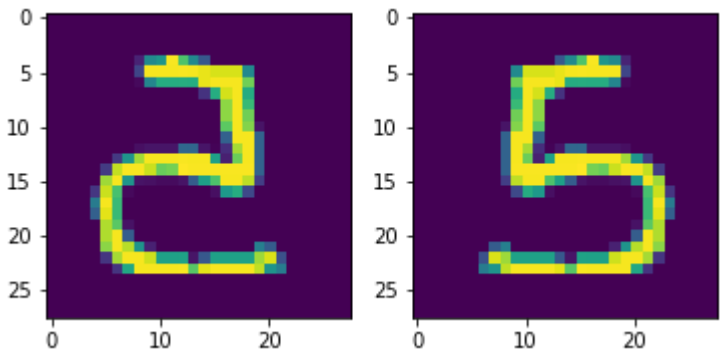
```
In [18]:
flipped_imgs = flipped_imgs.reshape(-1, 28,28)
```

```
In [19]:
flipped_imgs.shape, flipped_labels.shape
```

```
Out[19]: ((60000, 28, 28), (60000,))
```

```
In [20]:
f, ax = plt.subplots(1,2)
ax[0].imshow(flipped_imgs[100].reshape(28,28))
ax[1].imshow(x_train[100].reshape(28,28))
```

Setting "flipCode = 0" will produce vertical flips.



Next we will create a helper function for feeding random mini batches to our neural net input. Due to the nature of convolutional layers, they require massive amounts of memory during the forward and backward propogation steps. Consider a layer with 4x4 filters, outputting 128 feature maps with stride of 1 and SAME padding with a RGB image input of dimension 299x299. The number of parameters would equal (4x4x3+1) x 128 = 6272. Now consider each of those 128 feature maps is computing 299x299 neurons and each of these neurons is computing a weighted sum of 4x4x3 inputs. That's now 4x4x3x299x299x150 = 643,687,200 calculations. That's just for one training example. As you can imagine this quickly get's out of hand. The way to get around this is to feed small batches at a time to the network using a python generator which by nature keeps items out of memory until they are required.

In [21]:
```
def shuffle_batch(x,y, batch_size):
    rnd_idx = np.random.permutation(len(x))
    n_batches = len(x)//batch_size
    for batch_idx in np.array_split(rnd_idx, n_batches):
        x_batch, y_batch = x[batch_idx], y[batch_idx]
        yield x_batch, y_batch
```

We are ready to start creating our network architecture. First, we create our placeholders for our training data/labels. We will need to reshape them into a (-1, 28, 28, 1) matrix as the tensorflow conv2d layer expects a 4 dimensional input. We will set the first dimension to None to allow arbitrary batch sizes to be fed to the placeholder.

In [47]:
```
tf.reset_default_graph()
```

In [48]:
```
with tf.name_scope('placeholders'):

    X = tf.placeholder(np.float32, shape = [None, 28, 28], name = 'X')
    X_reshaped = tf.reshape(X, [-1, 28, 28, 1 ], name = 'X_reshaped')
    y = tf.placeholder(np.int32, shape = None, name = 'y')

```

Now we will design our convolutional layer. I will be taking inspiration from the Le-NET5 (pioneered by Yann LeCun) network architecture which is known for it's success in classifying hand written digits. However, there are many thing's I did differently. I recommend you study the Le-NET5 as well as other proven models to get some intuition of what kind of convolutional networks work for different tasks. Here is a link to his white paper.
http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.

In [87]:
```
with tf.name_scope('conv'):

    conv1 = tf.layers.conv2d(X_reshaped, 12, [3,3], strides = 1, padding = 'SAME', name = 'conv1')
    pool1 = tf.layers.max_pooling2d(conv1, [3,3], strides = 2, name = 'pool1')

    conv2 = tf.layers.conv2d(pool1, 16, [3,3], strides = 1, padding = 'SAME', name = 'conv2')
    pool2 = tf.layers.max_pooling2d(conv2, [3,3], strides = 2, name = 'pool2')

    pool2_flatten = tf.reshape(pool2, shape = (-1, 6*6*16))

    fc1 = tf.layers.dense(pool2_flatten, 256, activation = tf.nn.selu, name = 'fc1')
    fc2 = tf.layers.dense(fc1, 100, activation = tf.nn.selu, name = 'fc2')

    logits = tf.layers.dense(fc1, 10, activation = tf.nn.selu, name = 'output')
```

The first layer consists of 12 feature maps, using a 3x3 filter with stride of 1. We chose SAME padding which will maintain the dimensions of the image by adding a pad of zeros around the input. We then apply a max pooling layer with another 3x3 filter and strides of 2 which will output a 13x13x12 matrix. So we started with a 28x28x1 image, and have produced filter maps of which are less than half its size but much deeper in depth. We then pass this matrix along to the 2nd conv layer which has depth of 16, 3x3 filters, stride = 1 and padding SAME followed by the same max pooling layer as before. This outputs a

6*6*16 dimension matrix. You can see we are reducing the dimension space of our feature maps, but going deeper and and deeper. This is where we are learning to put together the lower level shapes learning in the first layer and form more complex patterns in the 2nd layer. Next we prepare the outputs for the fully connected layer by reshaping it into a 1 dimensional row vector consisting of 6x6x16 = 576 values. We use two dense layers with Selu activation to reduce the number of inputs by around half at each layer til finally feeding them to our logits which will output 10 predictions.

In [ ]:
```
with tf.name_scope('loss'):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits = logits, labels = y)
    loss = tf.reduce_mean(xentropy)
```

In [ ]:
```
with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(0.001)
    training_op = optimizer.minimize(loss)
```

In [ ]:
```
with tf.name_scope('eval'):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, np.float32))
```

We create our loss function which in this case will be softmax cross entropy which will output multi class probabilities. You can think of cross entropy as a measure of distance between various data points. We choose the AdamOptimizer (adaptive moment estimation) which automatically adjusts it's learning rate as it moves down the gradient. Finally, we create a means of evaluating our results. Tensorflow's in_top_k function will compute our logits and pick the top score. We then use our accuracy variable to output a percentage between 0–1%.

Now we are all ready for the training phase. First I will train the model without any augmented features or fancy bells and whistles. Let's see how well our model performs.

In [93]:
```
batch_size = 128

out = []

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    out = []
    for epoch in range(20):
        for x_batch, y_batch in shuffle_batch(x_train, y_train, batch_size):
            sess.run(training_op, feed_dict = {X: x_batch, y: y_batch})
        if epoch % 1 == 0:
            batch_acc = accuracy.eval(feed_dict = {X: x_batch, y: y_batch})
            val_acc = accuracy.eval(feed_dict = {X: x_test, y: y_test})
            print(epoch, "Batch Accuracy = ", batch_acc, "Validation Accuracy =", val_acc)
            outputs = sess.run(logits, feed_dict = {X: x_test})
            out.append(outputs)
```

```
0 Batch Accuracy =  0.96875 Validation Accuracy = 0.976
1 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9842
2 Batch Accuracy =  0.984375 Validation Accuracy = 0.9842
3 Batch Accuracy =  1.0 Validation Accuracy = 0.9865
4 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9825
5 Batch Accuracy =  1.0 Validation Accuracy = 0.9861
6 Batch Accuracy =  0.984375 Validation Accuracy = 0.9852
7 Batch Accuracy =  1.0 Validation Accuracy = 0.9883
8 Batch Accuracy =  1.0 Validation Accuracy = 0.9867
9 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9896
10 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9873
11 Batch Accuracy =  1.0 Validation Accuracy = 0.9889
12 Batch Accuracy =  1.0 Validation Accuracy = 0.9903
13 Batch Accuracy =  1.0 Validation Accuracy = 0.9897
14 Batch Accuracy =  1.0 Validation Accuracy = 0.9891
15 Batch Accuracy =  1.0 Validation Accuracy = 0.989
16 Batch Accuracy =  1.0 Validation Accuracy = 0.9908
17 Batch Accuracy =  1.0 Validation Accuracy = 0.9884
18 Batch Accuracy =  1.0 Validation Accuracy = 0.9872
19 Batch Accuracy =  1.0 Validation Accuracy = 0.9907
```

At epoch 19 we reach our highest percentage correct at 0.9907. This is already better than the results of any machine learning algorithm so convolutional has taken the lead. Let's now try and use our shifted features/ flipped features as well as add two new elements to our network. Dropout and Batch Normalization.

In [98]:
```
tf.reset_default_graph()
```

In [99]:
```
with tf.name_scope('placeholders'):

    X = tf.placeholder(np.float32, shape = [None, 28, 28], name = 'X')
    X_reshaped = tf.reshape(X, [-1, 28, 28, 1 ], name = 'X_reshaped')
    y = tf.placeholder(np.int32, shape = None, name = 'y')

    bn1_train = tf.placeholder_with_default(True, shape = (None), name = 'bn1_holder')
    bn2_train = tf.placeholder_with_default(True, shape = (None), name = 'bn2_holder')

    drop1 = tf.placeholder_with_default(True, shape = (None), name = 'Drop1')
    drop2 = tf.placeholder_with_default(True, shape = (None), name = 'Drop2')
```

We modify our existing placeholders with placeholder_with_default nodes which will hold the values produced by the batch normalization and drop out layers. During training we set these values to True and during testing we will turn them off by setting to False.

In [100]:

```
with tf.name_scope('conv'):

    conv1 = tf.layers.conv2d(X_reshaped, 12, [3,3], strides = 1, padding = 'SAME', name = 'conv1')
    pool1 = tf.layers.max_pooling2d(conv1, [3,3], strides = 2, name = 'pool1')
    bn1 = tf.layers.batch_normalization(pool1, training = bn1_train,  momentum = 0.9)
    dropout1 = tf.layers.dropout(bn1, 0.5, training = drop1)

    conv2 = tf.layers.conv2d(bn1, 16, [3,3], strides = 1, padding = 'SAME', name = 'conv2')
    pool2 = tf.layers.max_pooling2d(conv2, [3,3], strides = 2, name = 'pool2')
    bn2 = tf.layers.batch_normalization(pool2, training = bn2_train, momentum = 0.9)
    dropout2 = tf.layers.dropout(bn2, 0.5, training = drop2)

    bn2_flatten = tf.reshape(dropout2, shape = (-1, 6*6*16))

    fc1 = tf.layers.dense(bn2_flatten, 256, activation = tf.nn.selu, name = 'fc1')
    fc2 = tf.layers.dense(fc1, 100, activation = tf.nn.selu, name = 'fc2')

    logits = tf.layers.dense(fc1, 10, activation = tf.nn.selu, name = 'output')


```

Batch normalization simply centers and normalizes the data of each batch. We assign a momentum of 0.9. Drop out regularization assigns a probability (in our case 1 -0.5) to randomly turn nodes off completely during training. This results in the rest of the nodes having to pick up the slack thus improving their effectiveness. Imagine a company that decided to randomly choose 50 employees each week to stay home. The rest of the staff would have to handle the extra work effectively improving their skills in other areas. Not sure that would work in real life but in deep learning it's been proven effective.

In [102]:

```
with tf.name_scope('loss'):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits = logits, labels = y)
    loss = tf.reduce_mean(xentropy)
```

In [103]:

```
with tf.name_scope('train'):
    optimizer = tf.train.AdamOptimizer(0.001)
    training_op = optimizer.minimize(loss)
```

In [104]:

```
with tf.name_scope('eval'):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, np.float32))
```

In [105]:

```
batch_size = 128

out = []

extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    out = []
    for epoch in range(20):
        for x_batch, y_batch in shuffle_batch(final_x, final_y, batch_size):
            sess.run([training_op, extra_update_ops], feed_dict = {X: x_batch, y: y_batch})
        if epoch % 1 == 0:
            batch_acc = accuracy.eval(feed_dict = {X: x_batch, y: y_batch})
            val_acc = accuracy.eval(feed_dict = {bn1_train: False, bn2_train: False,
                                                 drop1: False, drop2: False, X: x_test, y: y_test})
            print(epoch, "Batch Accuracy = ", batch_acc, "Validation Accuracy =", val_acc)
            outputs = sess.run(logits, feed_dict = {bn1_train: False, bn2_train: False,
                                                    drop1: False, drop2: False, X: x_test})
            out.append(outputs)

```

We create our loss, train and eval steps as before then apply a few modifications to our execution phase. The computations performed by batch normalization are saved as update operations during each iteration. In order to access these we assign a variable extra_update_ops = tf.get_collection(tf.GraphKeys.UPDATE_OPS). During our training operation we feed this into sess.run as a item of a list along with training_op. Finally when performing our validation/test predictions we assign our placeholders False values via the feed_dict. We do not want there to be any randomization during our prediction phase. In order to get outputs we run the logits

operation using our test set. Let's see how well this model performs now that we've added regularization/normalization and are using augmented features. When using dropout the model will take longer to train so we'll increase our number of iterations to 30.

```
0 Batch Accuracy =  0.953125 Validation Accuracy = 0.988
1 Batch Accuracy =  0.953125 Validation Accuracy = 0.9911
2 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9918
3 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9915
4 Batch Accuracy =  1.0 Validation Accuracy = 0.9924
5 Batch Accuracy =  0.984375 Validation Accuracy = 0.9928
6 Batch Accuracy =  1.0 Validation Accuracy = 0.9902
7 Batch Accuracy =  0.96875 Validation Accuracy = 0.9929
8 Batch Accuracy =  1.0 Validation Accuracy = 0.9946
9 Batch Accuracy =  0.984375 Validation Accuracy = 0.993
10 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9935
11 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9937
12 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9941
13 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9933
14 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9941
15 Batch Accuracy =  0.984375 Validation Accuracy = 0.9932
16 Batch Accuracy =  0.984375 Validation Accuracy = 0.9932
17 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9933
18 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9929
19 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9932
20 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9932
21 Batch Accuracy =  0.984375 Validation Accuracy = 0.9942
22 Batch Accuracy =  1.0 Validation Accuracy = 0.9942
23 Batch Accuracy =  0.984375 Validation Accuracy = 0.9937
24 Batch Accuracy =  1.0 Validation Accuracy = 0.9941
25 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9934
26 Batch Accuracy =  1.0 Validation Accuracy = 0.9944
27 Batch Accuracy =  0.9765625 Validation Accuracy = 0.9947
28 Batch Accuracy =  0.9921875 Validation Accuracy = 0.9943
29 Batch Accuracy =  0.9921875 Validation Accuracy = 0.995
```

On epoch 29 we achieved 99.5% accuracy on our test set of 10,000 digits. As you can see the model acheieved > 99% accuracy on only the 2nd epoch compared to the 16th with our model before. Though 0.05% may not sound like much, this is a substantial improvement when dealing with huge amounts of data. Finally i'll show you how to extract predictions using np.argmax on our logits output.

```
In [111]:  1  y_hat = np.argmax(outputs, axis = 1)

In [112]:  1  from sklearn.metrics import accuracy_score

In [113]:  1  accuracy_score(y_hat, y_test)

Out[113]: 0.995

In [115]:  1  y_hat[:20]

Out[115]: array([9, 8, 0, 8, 6, 7, 1, 6, 4, 2, 2, 4, 4, 1, 5, 9, 5, 7, 8, 1],
                dtype=int64)

In [117]:  1  y_test[:20]

Out[117]: array([9, 8, 0, 8, 6, 7, 1, 6, 4, 2, 2, 4, 4, 1, 5, 9, 5, 7, 8, 1])
```

If there is anything you feel I missed or need to explain further please comment below. I'll try my best to explain. Follow the link below if you'd like to see some of the most famous convolutional networks. It's a great idea to study these and see how these models were crafted to get some intuition on what may work for your application.