Softmax Regression without convolution layer:

```
# Import MINST data
import input_data
mnist = input_data.read_data_sets("data/", one_hot=True)
import tensorflow as tf
# Set parameters
learning_rate = 0.01
training_iteration = 30
batch_size = 100
display_step = 2
# TF graph input
x = tf.placeholder("float", [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder("float", [None, 10]) # 0-9 digits recognition => 10 classes
# Create a model
# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
# Construct a linear model
model = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax
# Minimize error using cross entropy
# Cross entropy
cost_function = -tf.reduce_sum(y*tf.log(model))
# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)
# Initializing the variables
init = tf.initialize_all_variables()
# Launch the graph
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    for iteration in range(training_iteration):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={x: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost_function, feed_dict={x: batch_xs, y: batch_ys})/total_batch
    # Test the model
    predictions = tf.equal(tf.argmax(model, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(predictions, "float"))
    print "Accuracy:", accuracy.eval({x: mnist.test.images, y: mnist.test.labels})
```

Accuracy : 92%(approx)
We used Gradient descent method to minimize the function with step_length(learning rate) of 0.01

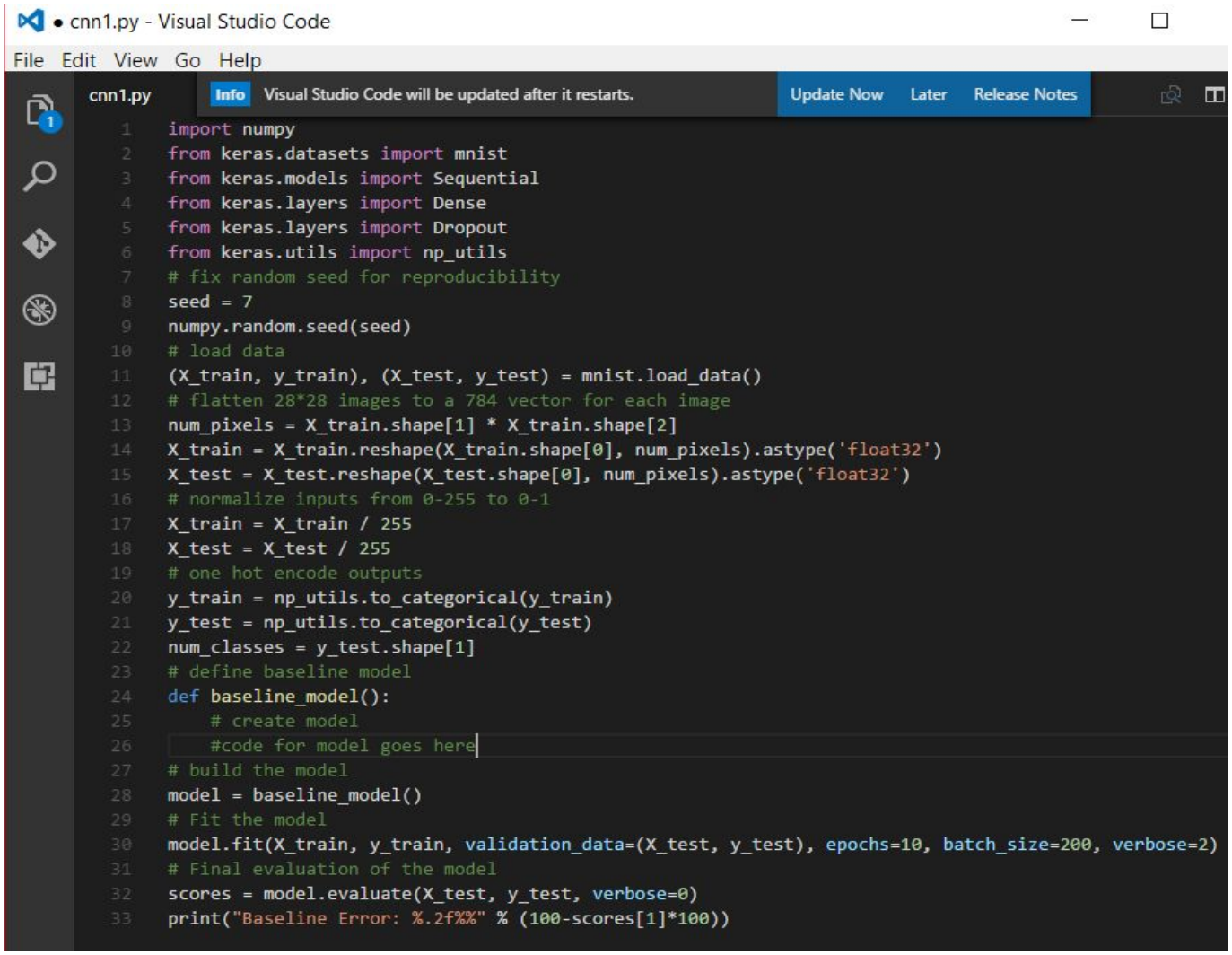
Softmax Regression with convolution layer

Using Convolutional Neural Networks:-

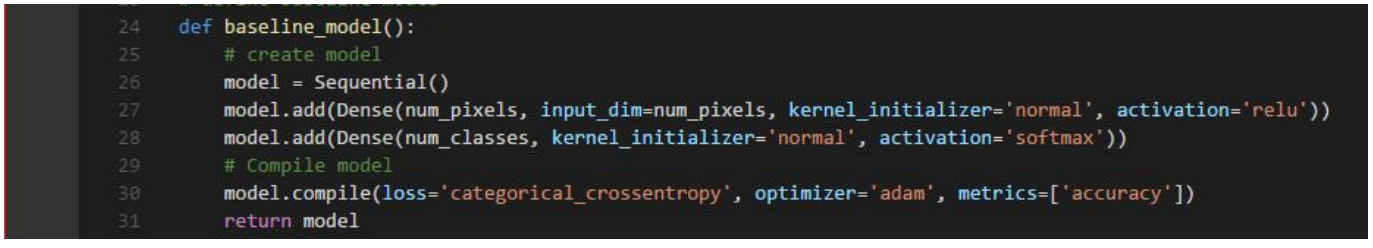
We have used Tensorflow library keras and numpy library for this assignment.

Initially we implemented the neural network without having any convolution layer.Then we increased the no of convolution layers to increase the accuracy.

In our code cnn created in function name base_model. We have provided snapshots for code template and different cnn models.



Baseline_model for without convolutional layer(cnn1)



Baseline_model for one convolutional layer(cnn2)

```python
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(32, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Baseline_model for larger convolutional network(cnn)

```python
def larger_model():
    # create model
    model = Sequential()
    model.add(Conv2D(30, (5, 5), input_shape=(1, 28, 28), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(15, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Our larger model contains 2 convolutional layers, 2 dense layers, 2 pooling layers, 1 dropout layer and flatten layer.

For without layer model we got max error % 1.88 which is provided in the below screenshot

```
 - 4s - loss: 0.2810 - acc: 0.9207 - val_loss: 0.1414 - val_acc: 0.9575
Epoch 2/10
 - 2s - loss: 0.1115 - acc: 0.9679 - val_loss: 0.0911 - val_acc: 0.9714
Epoch 3/10
 - 2s - loss: 0.0713 - acc: 0.9798 - val_loss: 0.0779 - val_acc: 0.9774
Epoch 4/10
 - 2s - loss: 0.0502 - acc: 0.9859 - val_loss: 0.0741 - val_acc: 0.9767
Epoch 5/10
 - 2s - loss: 0.0372 - acc: 0.9893 - val_loss: 0.0674 - val_acc: 0.9789
Epoch 6/10
 - 2s - loss: 0.0268 - acc: 0.9928 - val_loss: 0.0623 - val_acc: 0.9808
Epoch 7/10
 - 2s - loss: 0.0206 - acc: 0.9948 - val_loss: 0.0606 - val_acc: 0.9814
Epoch 8/10
 - 2s - loss: 0.0139 - acc: 0.9972 - val_loss: 0.0620 - val_acc: 0.9804
Epoch 9/10
 - 2s - loss: 0.0106 - acc: 0.9978 - val_loss: 0.0569 - val_acc: 0.9828
Epoch 10/10
 - 2s - loss: 0.0081 - acc: 0.9985 - val_loss: 0.0596 - val_acc: 0.9812
Baseline Error: 1.88%
```

For cnn2 model we got 1.14% error

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 5s - loss: 0.2256 - acc: 0.9357 - val_loss: 0.0779 - val_acc: 0.9756
Epoch 2/10
 - 4s - loss: 0.0711 - acc: 0.9788 - val_loss: 0.0447 - val_acc: 0.9849
Epoch 3/10
 - 4s - loss: 0.0509 - acc: 0.9846 - val_loss: 0.0438 - val_acc: 0.9858
Epoch 4/10
 - 4s - loss: 0.0390 - acc: 0.9879 - val_loss: 0.0401 - val_acc: 0.9871
Epoch 5/10
 - 4s - loss: 0.0324 - acc: 0.9899 - val_loss: 0.0346 - val_acc: 0.9883
Epoch 6/10
 - 4s - loss: 0.0267 - acc: 0.9918 - val_loss: 0.0336 - val_acc: 0.9889
Epoch 7/10
 - 4s - loss: 0.0221 - acc: 0.9931 - val_loss: 0.0350 - val_acc: 0.9894
Epoch 8/10
 - 4s - loss: 0.0191 - acc: 0.9942 - val_loss: 0.0341 - val_acc: 0.9883
Epoch 9/10
 - 4s - loss: 0.0156 - acc: 0.9951 - val_loss: 0.0314 - val_acc: 0.9893
Epoch 10/10
 - 4s - loss: 0.0143 - acc: 0.9956 - val_loss: 0.0337 - val_acc: 0.9886
CNN Error: 1.14%
```

For larger cnn model we got very low .86% error

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] - 5s 86us/step - loss: 0.3913 - acc: 0.8806 - val_loss: 0.0892 - val_acc: 0.9713
Epoch 2/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0999 - acc: 0.9697 - val_loss: 0.0559 - val_acc: 0.9820
Epoch 3/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0736 - acc: 0.9776 - val_loss: 0.0416 - val_acc: 0.9857
Epoch 4/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0607 - acc: 0.9816 - val_loss: 0.0368 - val_acc: 0.9878
Epoch 5/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0518 - acc: 0.9840 - val_loss: 0.0363 - val_acc: 0.9881
Epoch 6/10
60000/60000 [==============================] - 15s 255us/step - loss: 0.0441 - acc: 0.9857 - val_loss: 0.0297 - val_acc: 0.9904
Epoch 7/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0396 - acc: 0.9879 - val_loss: 0.0333 - val_acc: 0.9891
Epoch 8/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0361 - acc: 0.9886 - val_loss: 0.0277 - val_acc: 0.9896
Epoch 9/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0327 - acc: 0.9895 - val_loss: 0.0247 - val_acc: 0.9926
Epoch 10/10
60000/60000 [==============================] - 4s 64us/step - loss: 0.0309 - acc: 0.9902 - val_loss: 0.0285 - val_acc: 0.9914
Large CNN Error: 0.86%
```

Effect of change in number of convolutional layers:-

By our examples it is obvious that increasing convolutional layers will increase accuracy of the model. As each convolution layer can give much specific details for the input. Increasing cnn layers will also contribute for overfitting. So we have to select required number of cnn layers based on requirement.

Size of convolution kernel filter:-

We have used cnn2 model for this exercise as it contains only 1 cnn layer and the results can be viewed easily. In our above cnn2 model we have used filter size of 5x5. We have tested cnn2 on different filter sizes

1.2x2 (error % 1.42%)

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 6s - loss: 0.3128 - acc: 0.9112 - val_loss: 0.1384 - val_acc: 0.9596
Epoch 2/10
 - 5s - loss: 0.1134 - acc: 0.9680 - val_loss: 0.0796 - val_acc: 0.9757
Epoch 3/10
 - 5s - loss: 0.0803 - acc: 0.9760 - val_loss: 0.0675 - val_acc: 0.9787
Epoch 4/10
 - 5s - loss: 0.0634 - acc: 0.9813 - val_loss: 0.0592 - val_acc: 0.9798
Epoch 5/10
 - 5s - loss: 0.0514 - acc: 0.9847 - val_loss: 0.0583 - val_acc: 0.9819
Epoch 6/10
 - 5s - loss: 0.0424 - acc: 0.9867 - val_loss: 0.0516 - val_acc: 0.9829
Epoch 7/10
 - 5s - loss: 0.0361 - acc: 0.9893 - val_loss: 0.0462 - val_acc: 0.9845
Epoch 8/10
 - 5s - loss: 0.0316 - acc: 0.9903 - val_loss: 0.0532 - val_acc: 0.9816
Epoch 9/10
 - 5s - loss: 0.0263 - acc: 0.9917 - val_loss: 0.0444 - val_acc: 0.9851
Epoch 10/10
 - 5s - loss: 0.0243 - acc: 0.9922 - val_loss: 0.0461 - val_acc: 0.9858
CNN Error: 1.42%
```

2.10x10 (error % 1.05)

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 4s - loss: 0.2629 - acc: 0.9258 - val_loss: 0.0811 - val_acc: 0.9751
Epoch 2/10
 - 3s - loss: 0.0764 - acc: 0.9772 - val_loss: 0.0488 - val_acc: 0.9845
Epoch 3/10
 - 3s - loss: 0.0532 - acc: 0.9837 - val_loss: 0.0460 - val_acc: 0.9840
Epoch 4/10
 - 3s - loss: 0.0418 - acc: 0.9875 - val_loss: 0.0367 - val_acc: 0.9881
Epoch 5/10
 - 3s - loss: 0.0351 - acc: 0.9890 - val_loss: 0.0320 - val_acc: 0.9892
Epoch 6/10
 - 3s - loss: 0.0292 - acc: 0.9903 - val_loss: 0.0312 - val_acc: 0.9896
Epoch 7/10
 - 3s - loss: 0.0258 - acc: 0.9919 - val_loss: 0.0290 - val_acc: 0.9908
Epoch 8/10
 - 3s - loss: 0.0229 - acc: 0.9929 - val_loss: 0.0273 - val_acc: 0.9909
Epoch 9/10
 - 3s - loss: 0.0189 - acc: 0.9939 - val_loss: 0.0270 - val_acc: 0.9914
Epoch 10/10
 - 3s - loss: 0.0164 - acc: 0.9946 - val_loss: 0.0316 - val_acc: 0.9895
CNN Error: 1.05%
```

3.1x1 (error % 2.80)

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 6s - loss: 0.4605 - acc: 0.8711 - val_loss: 0.2673 - val_acc: 0.9200
Epoch 2/10
 - 5s - loss: 0.2537 - acc: 0.9233 - val_loss: 0.1993 - val_acc: 0.9402
Epoch 3/10
 - 5s - loss: 0.1899 - acc: 0.9419 - val_loss: 0.1561 - val_acc: 0.9534
Epoch 4/10
 - 5s - loss: 0.1524 - acc: 0.9529 - val_loss: 0.1331 - val_acc: 0.9603
Epoch 5/10
 - 5s - loss: 0.1281 - acc: 0.9597 - val_loss: 0.1159 - val_acc: 0.9634
Epoch 6/10
 - 5s - loss: 0.1109 - acc: 0.9658 - val_loss: 0.1027 - val_acc: 0.9676
Epoch 7/10
 - 5s - loss: 0.0971 - acc: 0.9698 - val_loss: 0.1003 - val_acc: 0.9687
Epoch 8/10
 - 5s - loss: 0.0861 - acc: 0.9727 - val_loss: 0.0964 - val_acc: 0.9691
Epoch 9/10
 - 5s - loss: 0.0788 - acc: 0.9753 - val_loss: 0.0952 - val_acc: 0.9707
Epoch 10/10
 - 5s - loss: 0.0720 - acc: 0.9768 - val_loss: 0.0886 - val_acc: 0.9720
CNN Error: 2.80%
```

By the above observations decreasing the filter size to 1x1 will heavily decreased the accuracy. But decreasing to 2x2 increased the model accuracy and increasing filter size to 10x10 also increased model accuracy. But this observations doesn't mean increase in cnn filter size will increase model accuracy.

Size of cnn kernel size depends on the requirement because if we take larger kernel size we will lose many details of the image and if we take filter size of 1x1 we will get pixel level details which may lead to confusion in the model. That's why in our above case of filter size 1x1 error % increased.

Effect of Activation functions:-

For this we have used cnn2 model. In cnn2 model we have used relu(rectified linear unit). We have tested on different activation functions

1.sigmoid(error % 1.75)

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 3s - loss: 0.6943 - acc: 0.7869 - val_loss: 0.2689 - val_acc: 0.9196
Epoch 2/10
 - 2s - loss: 0.2426 - acc: 0.9254 - val_loss: 0.1568 - val_acc: 0.9518
Epoch 3/10
 - 2s - loss: 0.1652 - acc: 0.9494 - val_loss: 0.1158 - val_acc: 0.9649
Epoch 4/10
 - 2s - loss: 0.1281 - acc: 0.9610 - val_loss: 0.0960 - val_acc: 0.9695
Epoch 5/10
 - 2s - loss: 0.1073 - acc: 0.9673 - val_loss: 0.0809 - val_acc: 0.9736
Epoch 6/10
 - 2s - loss: 0.0907 - acc: 0.9718 - val_loss: 0.0701 - val_acc: 0.9772
Epoch 7/10
 - 2s - loss: 0.0800 - acc: 0.9755 - val_loss: 0.0627 - val_acc: 0.9786
Epoch 8/10
 - 2s - loss: 0.0714 - acc: 0.9771 - val_loss: 0.0572 - val_acc: 0.9817
Epoch 9/10
 - 2s - loss: 0.0663 - acc: 0.9796 - val_loss: 0.0509 - val_acc: 0.9830
Epoch 10/10
 - 2s - loss: 0.0598 - acc: 0.9813 - val_loss: 0.0540 - val_acc: 0.9825
CNN Error: 1.75%
```

2.tanh(error % 1.55)

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
 - 3s - loss: 0.3401 - acc: 0.9018 - val_loss: 0.1401 - val_acc: 0.9567
Epoch 2/10
 - 2s - loss: 0.1336 - acc: 0.9591 - val_loss: 0.0976 - val_acc: 0.9675
Epoch 3/10
 - 2s - loss: 0.0956 - acc: 0.9703 - val_loss: 0.0731 - val_acc: 0.9771
Epoch 4/10
 - 2s - loss: 0.0773 - acc: 0.9754 - val_loss: 0.0710 - val_acc: 0.9771
Epoch 5/10
 - 2s - loss: 0.0677 - acc: 0.9790 - val_loss: 0.0671 - val_acc: 0.9790
Epoch 6/10
 - 2s - loss: 0.0575 - acc: 0.9811 - val_loss: 0.0600 - val_acc: 0.9820
Epoch 7/10
 - 2s - loss: 0.0500 - acc: 0.9844 - val_loss: 0.0549 - val_acc: 0.9833
Epoch 8/10
 - 2s - loss: 0.0463 - acc: 0.9852 - val_loss: 0.0520 - val_acc: 0.9835
Epoch 9/10
 - 2s - loss: 0.0398 - acc: 0.9870 - val_loss: 0.0498 - val_acc: 0.9842
Epoch 10/10
 - 2s - loss: 0.0368 - acc: 0.9887 - val_loss: 0.0503 - val_acc: 0.9845
CNN Error: 1.55%
```

In both cases of activation(sigmoid and tanh) error % is more compared to relu activation. In the activation case of sigmoid, gradient becomes low at large values of x and may be vanished. This means the network is unable to learn further which causes increase in error %. Which is also the same case of tanh. In case of relu gradient is constant and in our case relu yielded better result.