Lab Course Machine Learning
Exercise Sheet 10
Krithika Murugesan 277537

Dataset : MNIST database

The dataset is a database of handwritten digits with 60,000 training examples and 10,000 test examples. It requires minimal or no pre-processing as data is normalized and centered. The x values are the pixel values of the images while the y values are the digit from 0 to 9, a multi class classification problem.

Exercise 1: Neural Network with Tensorflow

The task given is to create a neural network using Tensorflow for digit classification. The data can be imported directly from Tensorflow tutorial package in python, the data is already split into train and test, mentioning the validation split ratio at the time reading data will separate train data into required train and test.

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True,validation_size=18000)
```

Next to set up a simple neural network, the parameters and layers have to be defined prior to the execution of the net. The learning rate is set to 0.1, this can be modified by the algorithm during the learning process. The number of epochs is set to 500 and batches are used to loop over during training, a batch size of 50 is used here. The number of neurons at each layer also has to be initialized.

The input and output(labels) are converted to Tensorflow objects so that they can be fed into the network. Similarly the weights and biases are initialized, this depends on the number of layers, since we are implementing a single layer neural net here, we have only one hidden layer and the output layer.

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'out': tf.Variable(tf.random_normal([n_hidden_1, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

Now that all the initializations are done, the neural net can be built as follows

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_1, weights['out']) + biases['out']
    return out_layer
```

The tf.reduce_mean function along with soft_max_entropy is used as the loss function, Adam optimizer is used out of curiosity, it is like an extension to stochastic gradient descent, which is an effective optimizer. Using the defined functions the model is trained to reduced to the loss.

```
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)
```

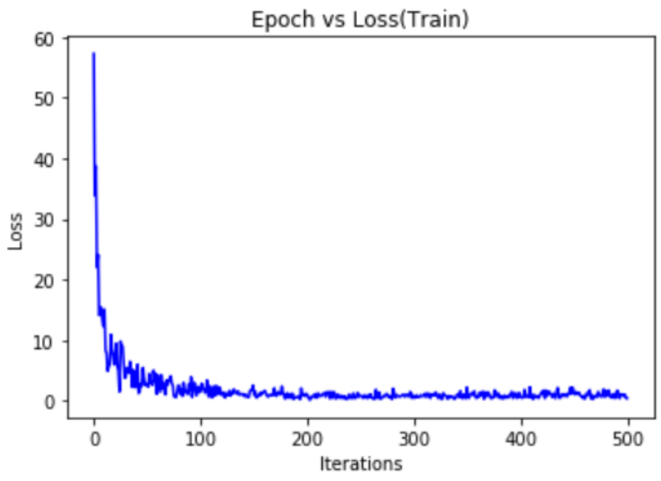
Then the initializer is run, the session is saved used **tf.train.Saver()**, at every epoch the loss and accuracy are stored.

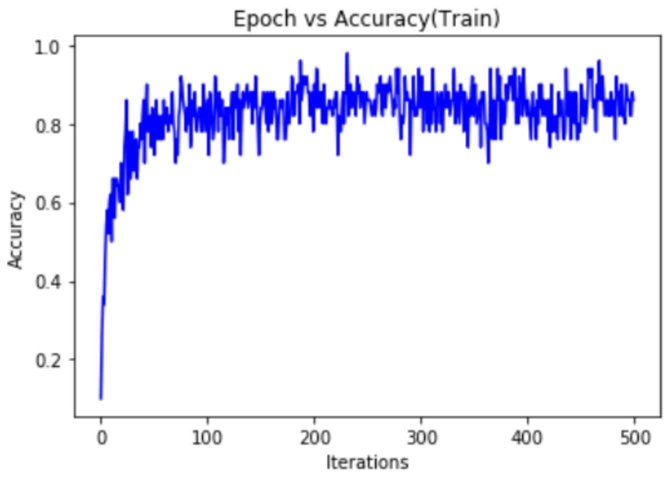
```
# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)
    #Saving session
    saver = tf.train.Saver()
    lossTrain,accTrain,lossValid,accValid,ii = [],[],[],[],[]
    for i in range(0, num_steps):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        valid_x,valid_y = mnist.validation.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        # Calculate batch loss and accuracy
        lossT, accT = sess.run([loss_op, accuracy], feed_dict={X: batch_x,Y: batch_y})
        lossV, accV = sess.run([loss_op, accuracy], feed_dict={X: valid_x,Y: valid_y})
        lossTrain.append(lossT)
        accTrain.append(accT)
        lossValid.append(lossV)
        accValid.append(accV)
        ii.append(i)
        saver.save(sess, r'./nnModel1',global_step = i+1)
```

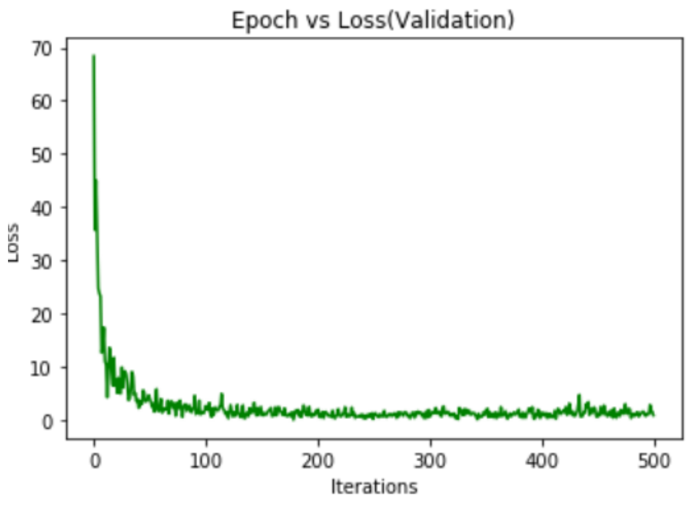
Later the saved model is used to predict the labels in test set,

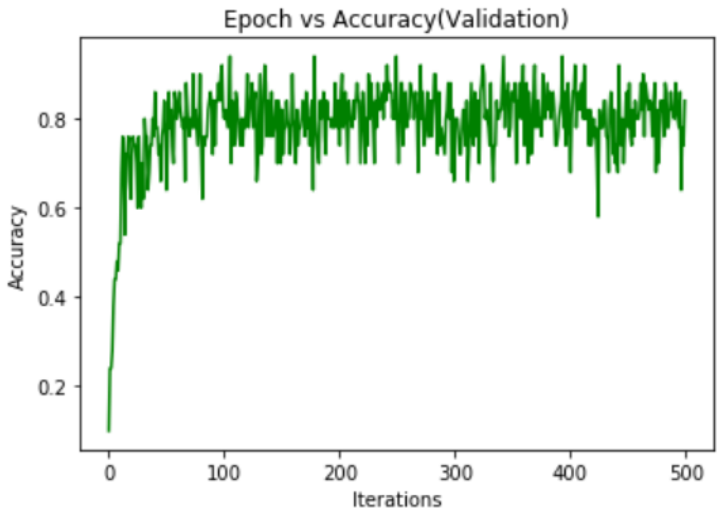
```
# Calculate accuracy for MNIST test images
sess = tf.Session()
new_saver = tf.train.import_meta_graph('nnModel1-500.meta')
new_saver.restore(sess, tf.train.latest_checkpoint('./'))
print("Testing Accuracy:", sess.run(accuracy, feed_dict={X: mnist.test.images,Y: mnist.test.labels}))
```

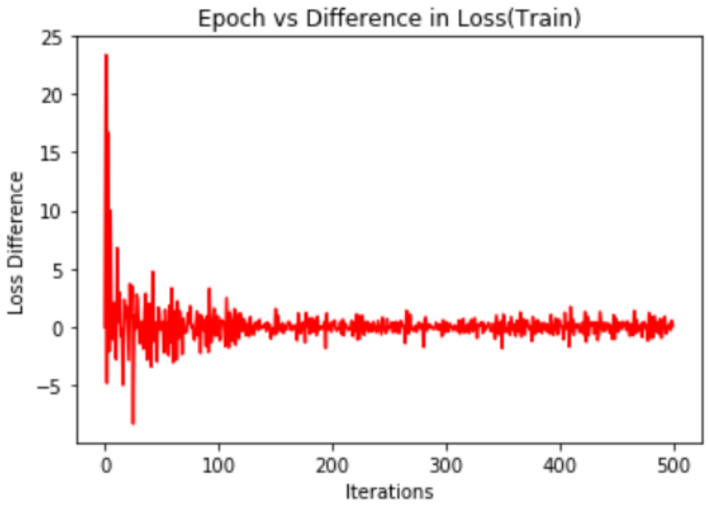
The test data accuracy is 82.52% which is pretty good considering the simple neural net we used. The loss and accuracy variations across each epoch are given below;





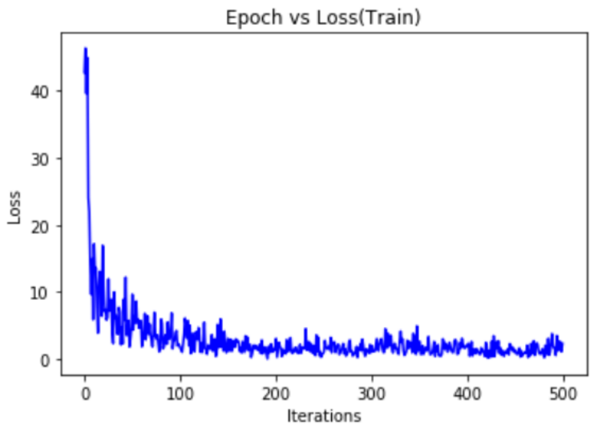


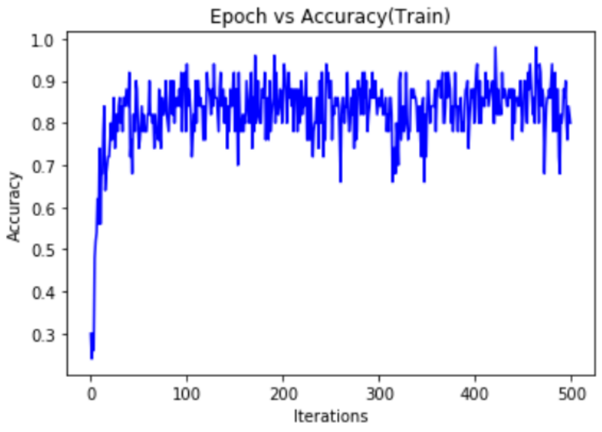


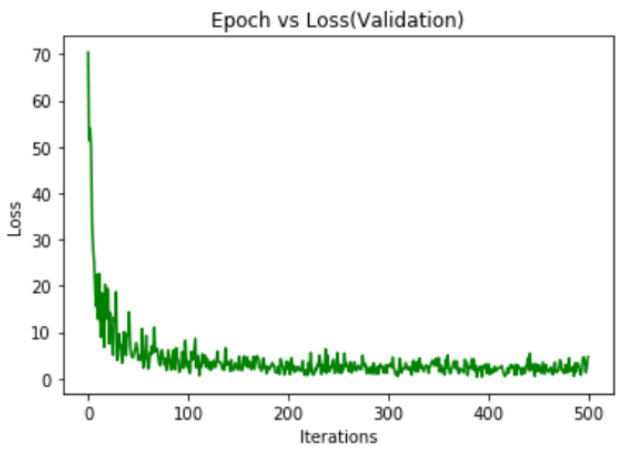


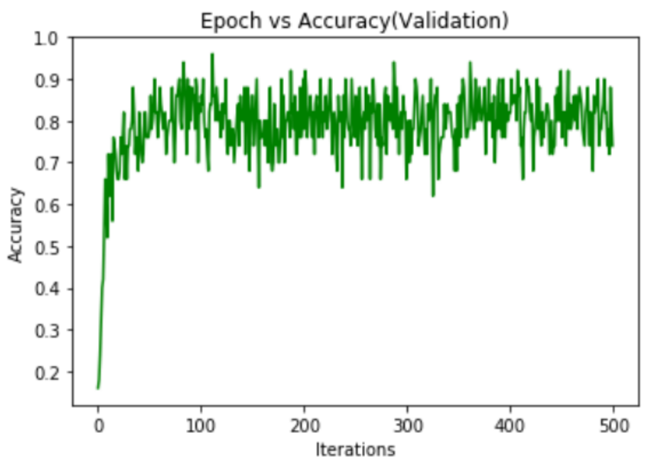
Grid search is performed for the combinations of hidden layers = {1,2} and nodes per layer as {15,25}. To accommodate the repetitive use of code, all the above methods are incorporated in a function model with hidden layers and number of nodes as parameters. The graphs are remaining combinations of parameters are:

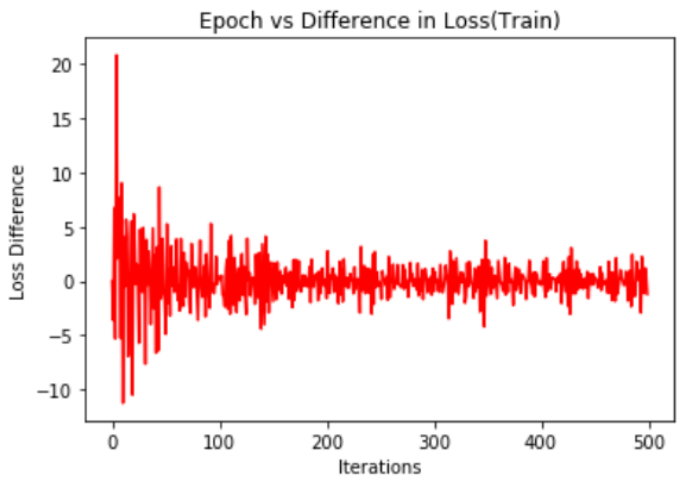
a) 1 layer, 25 nodes : Test accuracy - 81.67%



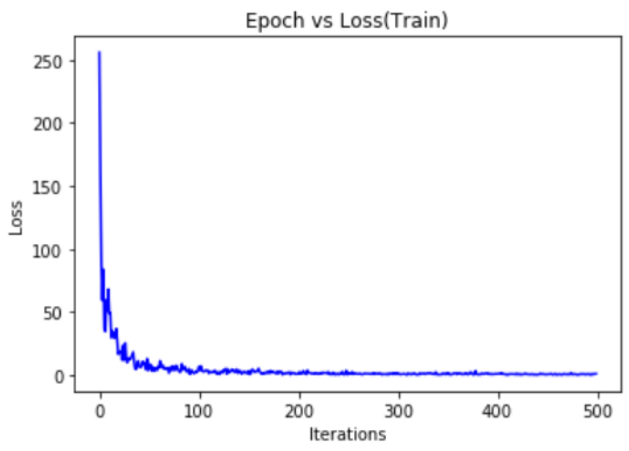


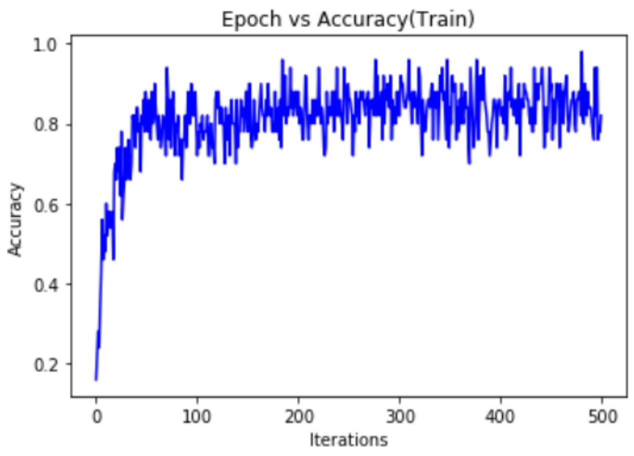


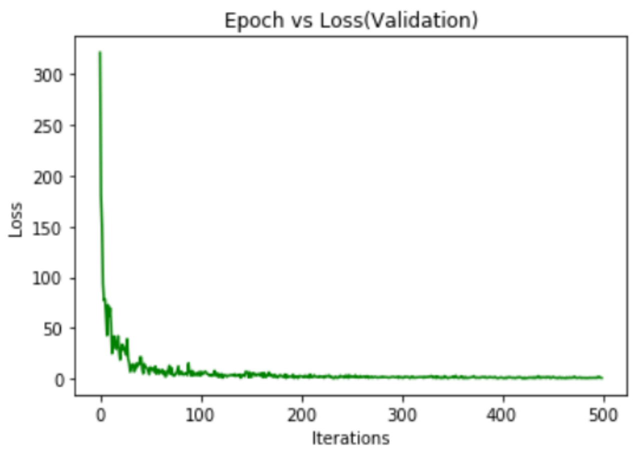


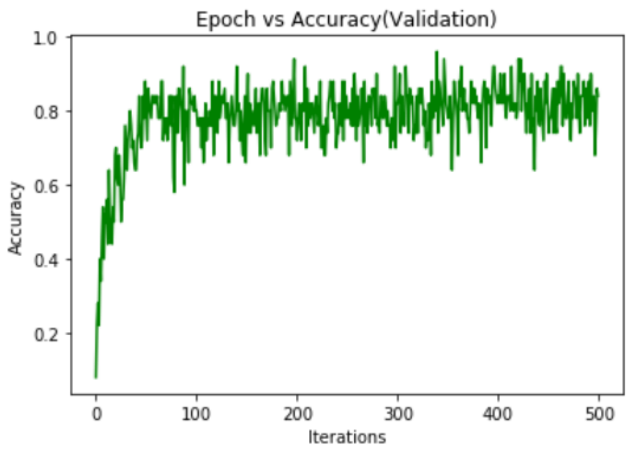


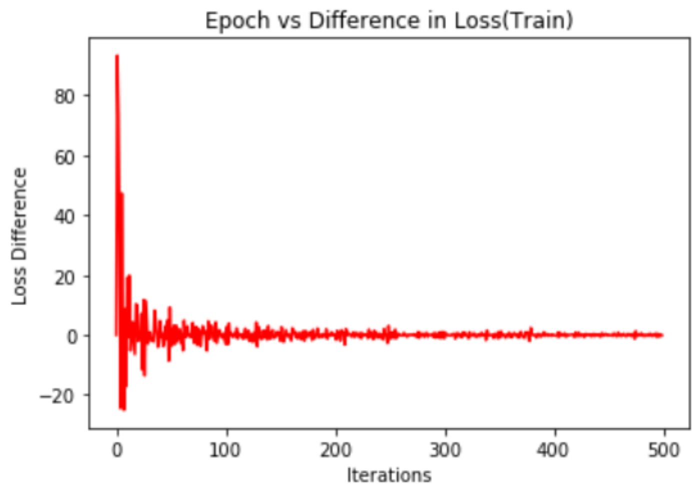
b) 2 layers, 15 nodes : Testing accuracy - 83.6%



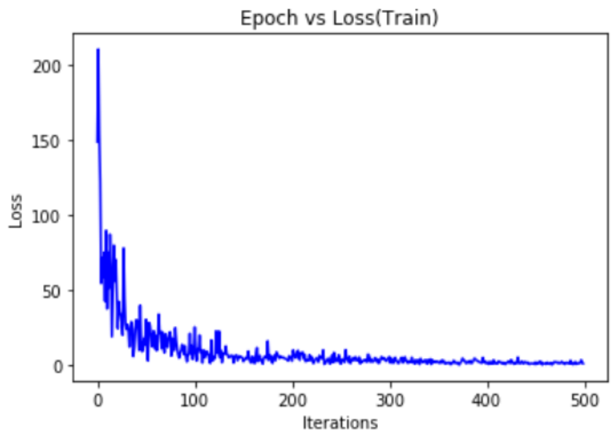


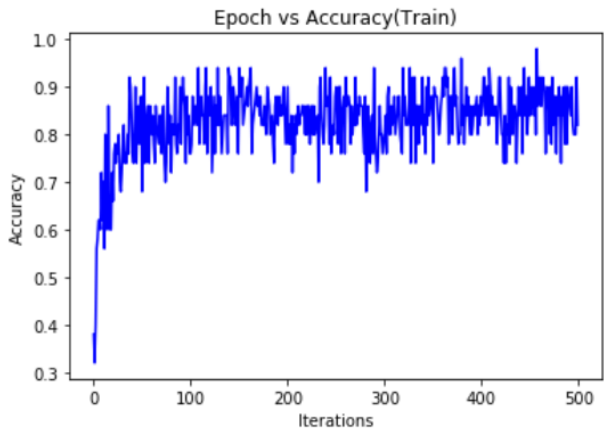


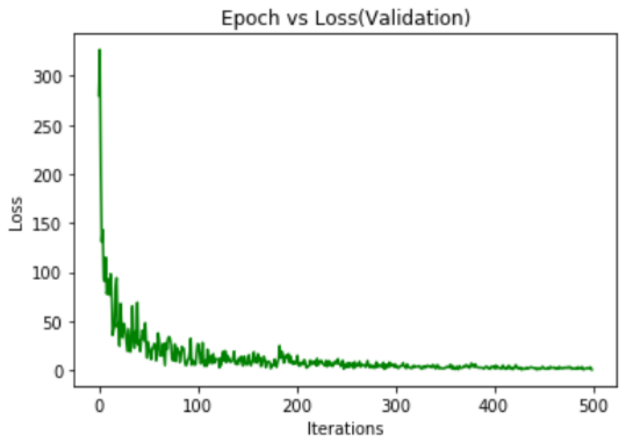


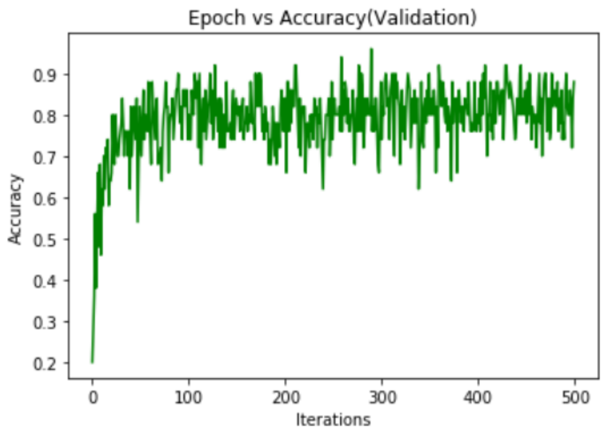


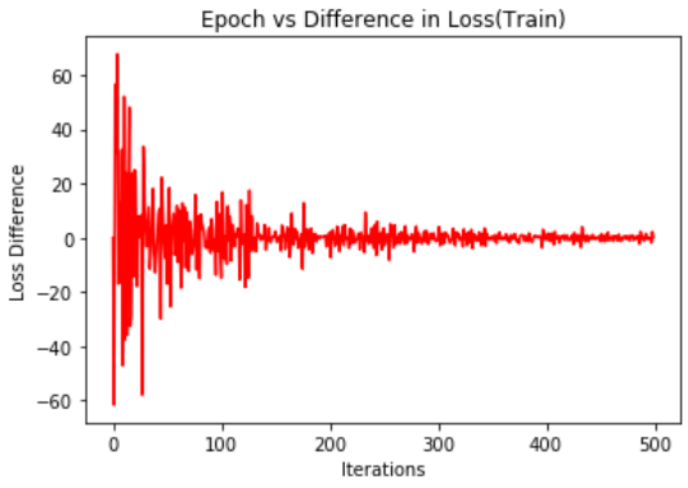
c) 2 layers, 25 nodes : Test accuracy - 85.31%











The remaining combination of 1 layer and 15 nodes is given in the first set of graphs, it can be seen that the 2 layer 25 node neural net performs best on the test data. While this model is a little slower in convergence compared to the rest, which can be seen in the huge initial variations in loss difference between the epochs. When the number of neurons is less, it cannot learn all the underlying data patterns, but after a optimal point it makes no difference. In our implementation we have considered only fairly smaller number of neurons therefore the performance increases with number nodes. Adding too many hidden layers will decrease the effectiveness of back propagation and also may tend to over fitting of data.

Note : Normal.ipynb is simple implementation, while other ipython file has the same implementation with codes embedded in functions for ease of use in grid search.