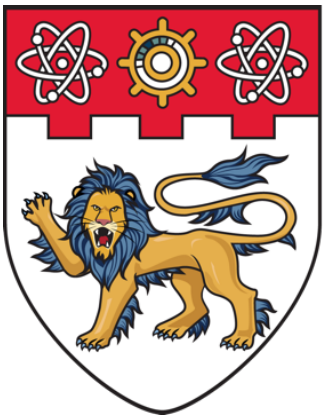


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ042 NEURAL NETWORKS
ASSIGNMENT 1

ZHANG WANLU U1420638G
ZHU YIMIN

SCHOOL OF COMPUTER ENGINEERING

1 Part 1

1.1 Introduction

The first part of this project is to use neural network to deal with classification problem. The aim is to predict class labels in the test dataset after training the neural networks on the training data.The dataset contains multispectral values of pixels in a 3x3 neighbourhoods in a satellite images and class labels of the centre pixels in each neighbourhood. Since different parameters will affect the overall performance, it's necessary to try out several of them and select the one given the best result. In this section, we use multilayer feedforward networks to solve the problem.

1.2 Notes

For Part 1, the main tools we use are Theano, numpy and matplotlib. All the training and performance measurements are done on a notebook with 4.0 GHz Intel Core i7-6700k.

1.3 Data Preparation

The training data is read from file *sat_train.txt* while the test data is read from file *sat_test.txt*. Then we perform scaling on train and test data to make them distributed within a certain range. The min-max value method is used here. The code for data preparation is here.

```
# scale data
def scale(X, X_min, X_max):
  return (X - X_min)/(X_max-X_min)

#read train data
train_input = np.loadtxt('sat_train.txt',delimiter=' ')
trainX, train_Y = train_input[:,:36], train_input[:,-1].astype(int)
trainX_min, trainX_max = np.min(trainX, axis=0), np.max(trainX, axis=0)
trainX = scale(trainX, trainX_min, trainX_max)

train_Y[train_Y == 7] = 6
trainY = np.zeros((train_Y.shape[0], 6))
trainY[np.arange(train_Y.shape[0]), train_Y-1] = 1

#read test data
test_input = np.loadtxt('sat_test.txt',delimiter=' ')
testX, test_Y = test_input[:,:36], test_input[:,-1].astype(int)

testX_min, testX_max = np.min(testX, axis=0), np.max(testX, axis=0)
testX = scale(testX, testX_min, testX_max)

test_Y[test_Y == 7] = 6
testY = np.zeros((test_Y.shape[0], 6))
testY[np.arange(test_Y.shape[0]), test_Y-1] = 1
```

1.4 Question 1

Question 1 is to design and implement a 3-layer feedforward neural network with a hidden layer of 10 neurons. The learning curves and testing results are in the result section.

1.4.1 Method

The model contains 3 layers, input layer, hidden layer and output layer. The activation function for the hidden layer is logistic activation. The output layer uses softmax function. The weight and bias are random initialized. We set the default batch size as 32. We use Stochastic gradient descent (shortened to SGD) with mini batch to train the network. The cost function is defined as

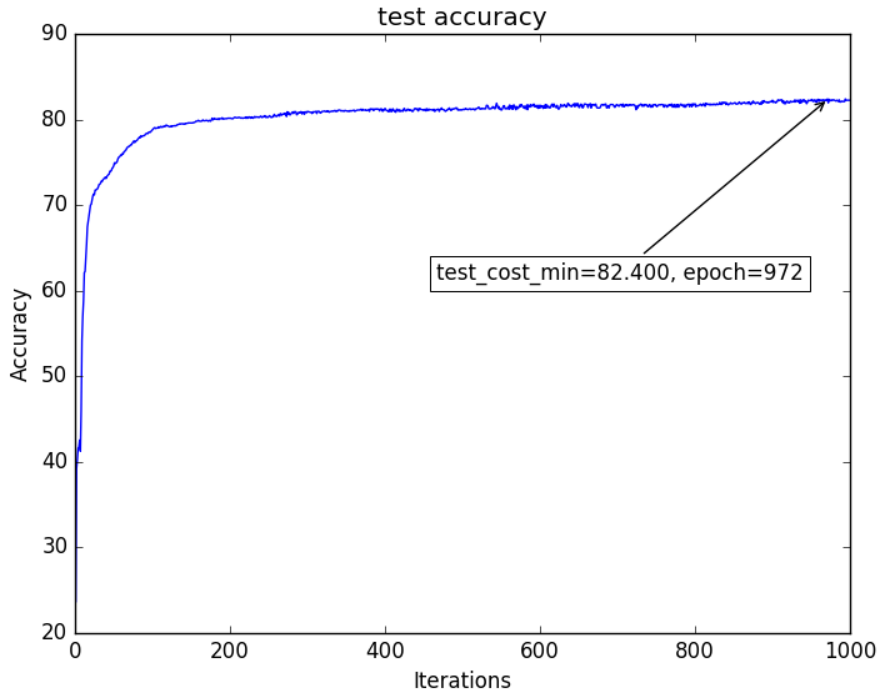
the mean of the crossentropy of predicted and actual value then plus regularization term. Once the predefined Theano function *train* is called, the cost will be computed then the weights and biases will be updated by the SGD function. The learning rate $\alpha = 0.01$. The decay parameter $\beta = 10^{-6}$.

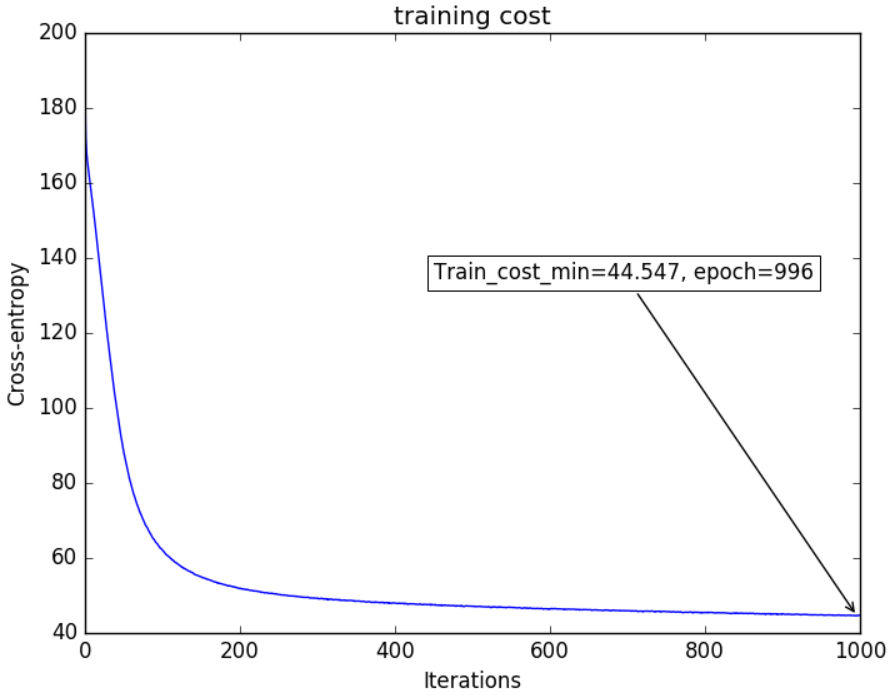
```
# theano expressions
X = T.matrix() #features
Y = T.matrix() #output

  #weights and biases from input to hidden layer
w1, b1 = init_weights(36, hnno), init_bias(hnno)
#weights and biases from hidden to output layer
w2, b2 = init_weights(hnno, 6, logistic=False), init_bias(6)

h1 = T.nnet.sigmoid(T.dot(X, w1) + b1)
py = T.nnet.softmax(T.dot(h1, w2) + b2)

y_x = T.argmax(py, axis=1)

cost = T.mean(T.nnet.categorical_crossentropy(py, Y)) + decay*(T.sum(T.sqr(w1)+T.
  sum(T.sqr(w2))))
params = [w1, b1, w2, b2]
updates = sgd(cost, params, learning_rate)

# compile
train = theano.function(inputs=[X, Y],
                        outputs=cost,
                        updates=updates,
                        allow_input_downcast=True)
predict = theano.function(inputs=[X],
                          outputs=y_x,
                          allow_input_downcast=True)
```

At each epoch, the data will be shuffled. An average cost for this epoch is computed and stored in the list train_cost.

```
# train and test
n = len(trainX)
test_accuracy = []
train_cost = []
for i in range(epochs):
  if i % 1000 == 0:
    print(i)

  trainX, trainY = shuffle_data(trainX, trainY)
  cost = 0.0
  for start, end in zip(range(0, n, batch_size), range(batch_size, n, batch_size)):
    cost += train(trainX[start:end], trainY[start:end])
  train_cost = np.append(train_cost, cost/(n // batch_size))

  test_accuracy = np.append(test_accuracy, np.mean(np.argmax(testY, axis=1) ==
    predict(testX)))

print('%.1f accuracy at %d iterations'%(np.max(test_accuracy)*100, np.argmax(
    test_accuracy)+1))
```

1.4.2 Result


test accuracy
test_cost_min=82.400, epoch=972
Accuracy
Iterations
90
80
70
60
50
40
30
20
0
200
400
600
800
1000


training cost
Train_cost_min=44.547, epoch=996
Cross-entropy
Iterations
200
180
160
140
120
100
80
60
40
0
200
400
600
800
1000

1.5 Question 2

Question 2 is to find the optimal batch size for mini-batch gradient descent.

1.5.1 Method

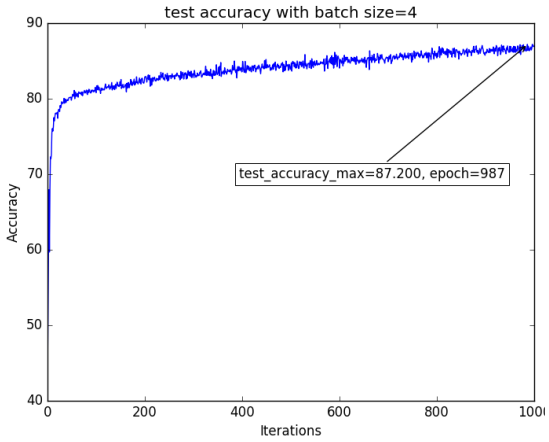
We tried different batch size from the list batch_sizeA {4,8,16,32,64}. A for loop is used to iterate through the list and execute the code. In each iteration, a new model is generated and initialized for training. The list *times* is used to record the time taken to update parameters of the nerwork for each batch size.

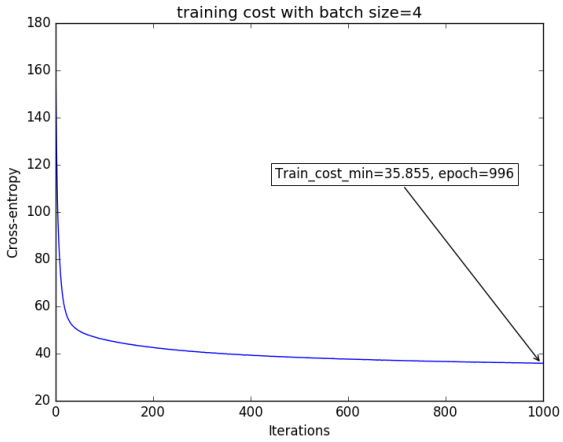
```
def tnt(batch_size, hnno, decay, trainX, trainY):
  initialize(hnno, decay)
  global test_accuracy, train_cost, epochs, times, Accuracy_plot
  t1 = time.time()
  n = len(trainX)
  for i in range(epochs):
    trainX, trainY = shuffle_data(trainX, trainY)
    cost = 0.0
    for start, end in zip(range(0, n, batch_size), range(batch_size, n, batch_size))
    :
      cost += train(trainX[start:end], trainY[start:end])
    train_cost = np.append(train_cost, cost/(n // batch_size))
    test_accuracy = np.append(test_accuracy, np.mean(np.argmax(testY, axis=1) ==
    predict(testX)))
  times.append(time.time()-t1)
  print('Time: %.2fs'%times[-1])
  accuracy = (np.max(test_accuracy)*100, np.argmax(test_accuracy)+1)
  print('%.2f accuracy at %d iterations'%accuracy)

times = []
batch_sizeA = [4,8,16,32,64]

for batch_size in batch_sizeA:
  hnno = 10
  decay = 1e-6
  test_accuracy = []
  train_cost = []
  print('Batch size = %d running'%batch_size)
  tnt(batch_size, hnno, decay, trainX, trainY)
  plots(batch_size)
timeplot(batch_sizeA)
```

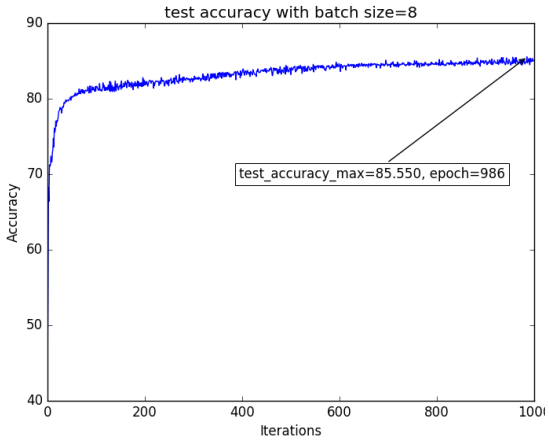
The function *tnt* is also used in the program for following questions.

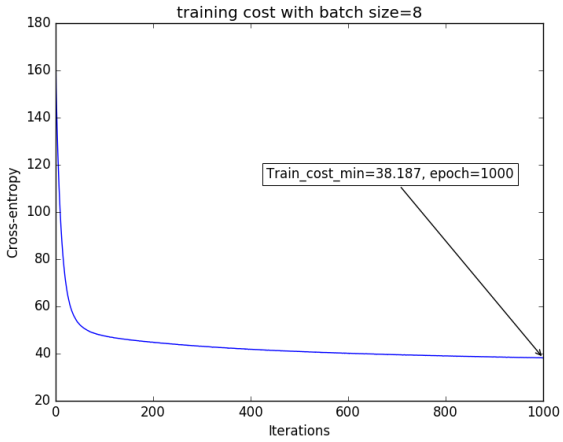
1.5.2 Result

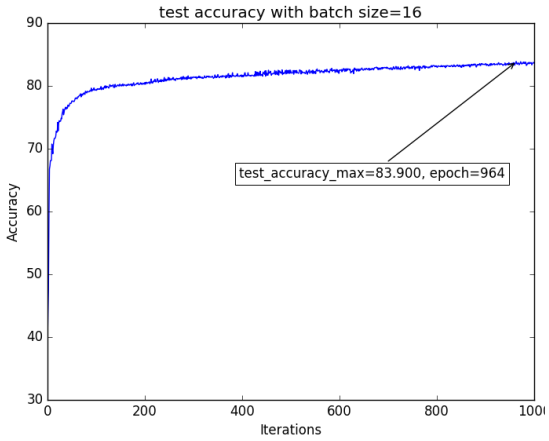
The figures for training errors and test accuracies against the number of epochs with different batch size.

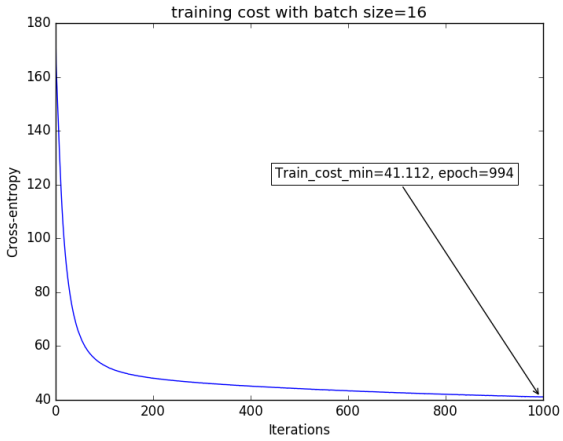


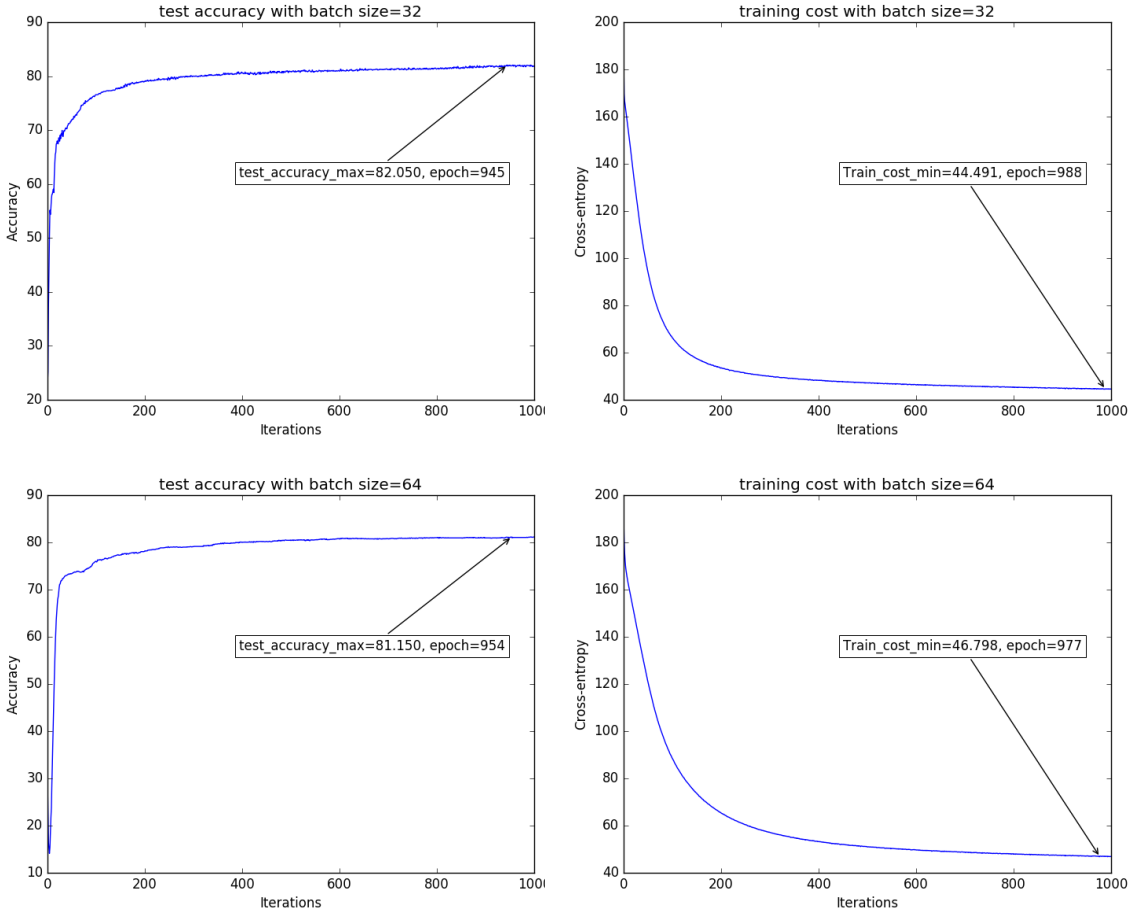




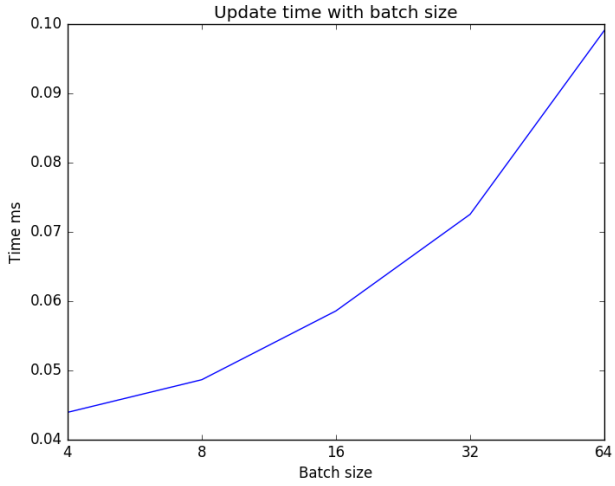








As shown above, we can see that the performance is best when **batch size = 4**. Batch size determines how many examples are used to update the weight. The lower it is, the noisier the training signal is going to be, the higher it is, the longer it will take to compute the gradient for each step. Furthermore, too big batch size may result in local optimum. The following figure shows the time taken to update parameters.



1.6 Question 3

Question 3 is to find the optimal number of hidden neurons. According to last part, the batch size is set as 4.

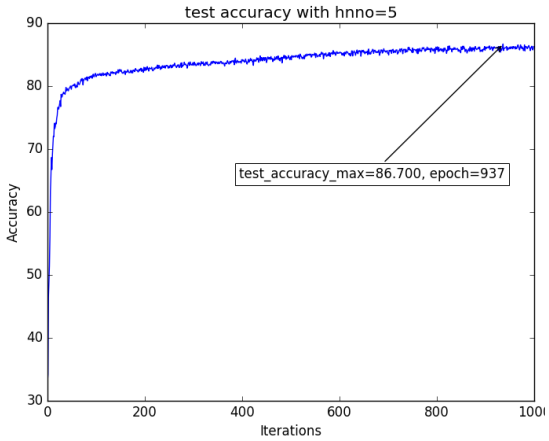
1.6.1 Method

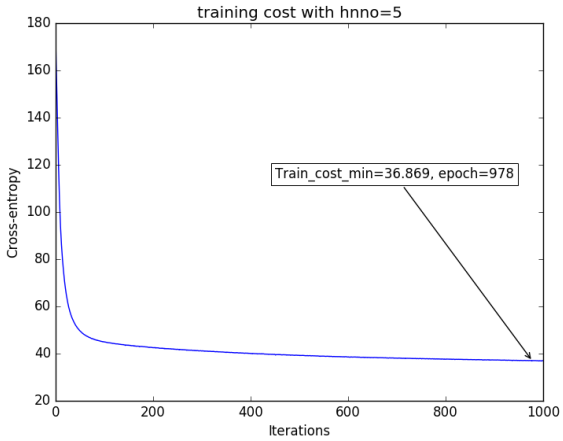
Method is similar as the batch size part. The number of hidden neurons are from the list hnnoA.

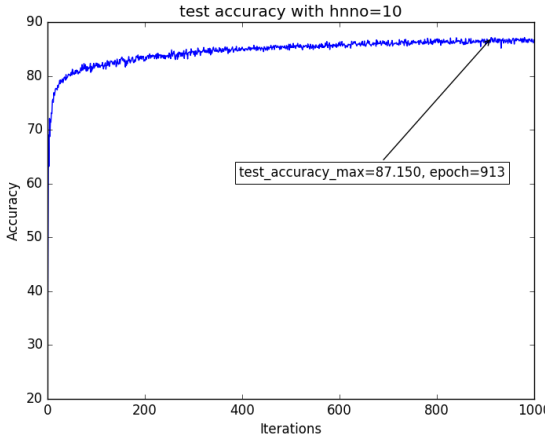
```
hnnoA = [5,10,15,20,25]
times = []

for hnno in hnnoA:
  batch_size = 4
  decay = 1e-6
  test_accuracy = []
  train_cost = []
  print('No. of hidden neurons = %d running'%hnno)
  tnt(batch_size, hnno, decay, trainX, trainY)
  plots(hnno)
timeplot(hnnoA)
```

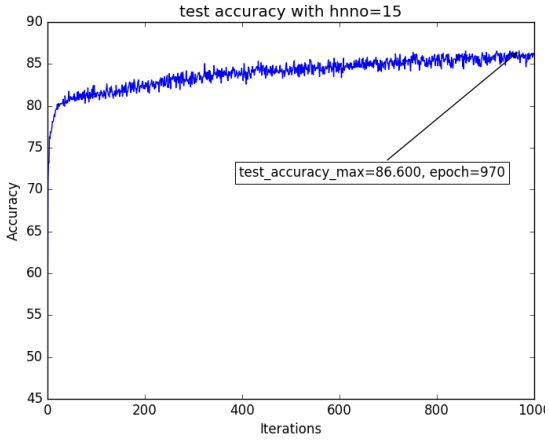
1.6.2 Result

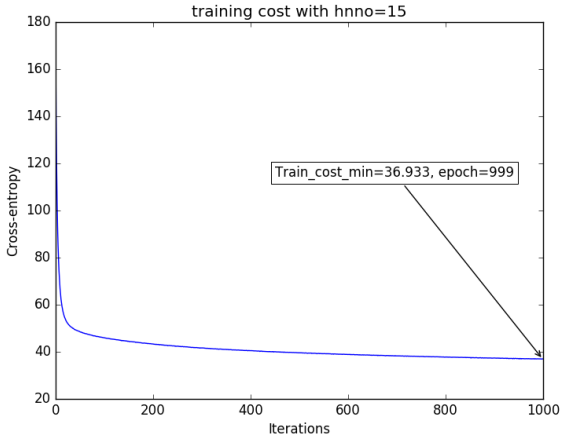
The following figures show the training errors and test accuracies against the number of epochs with different number of hidden-layer neurons.

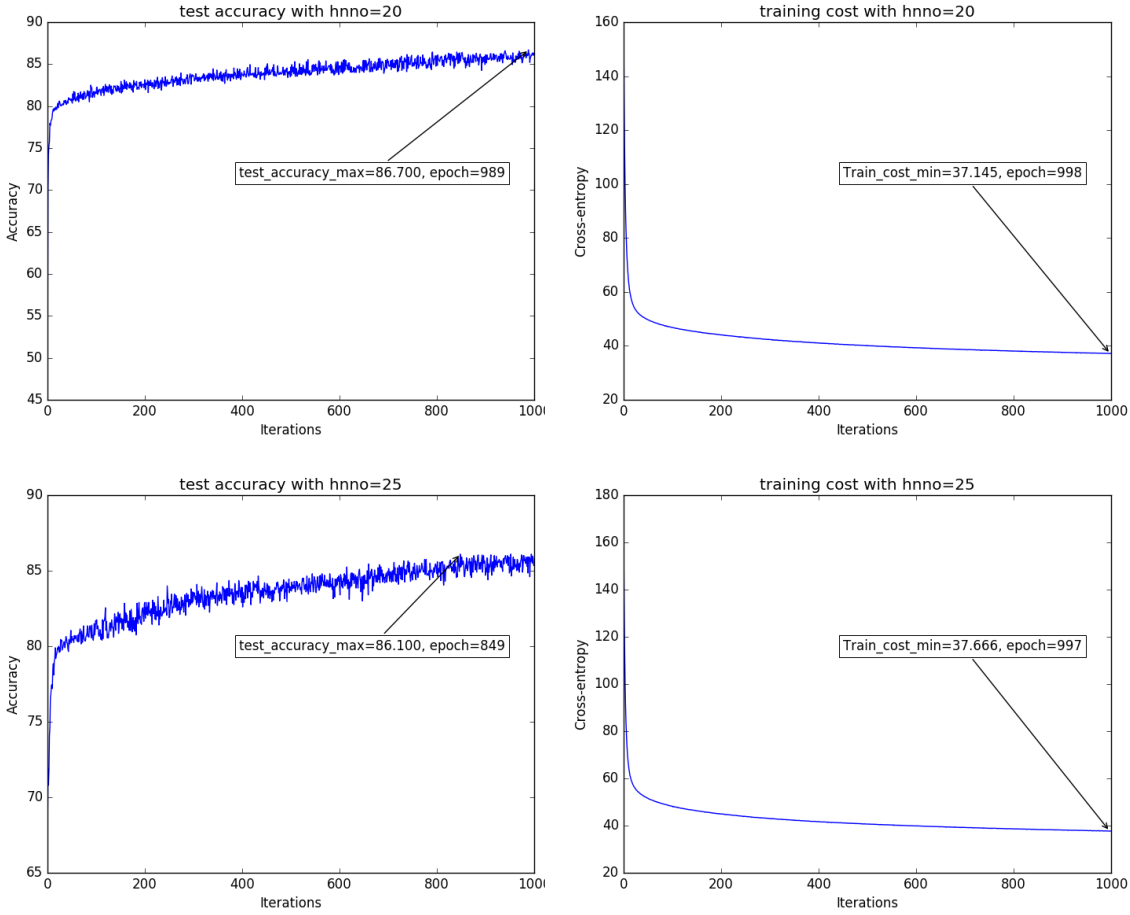




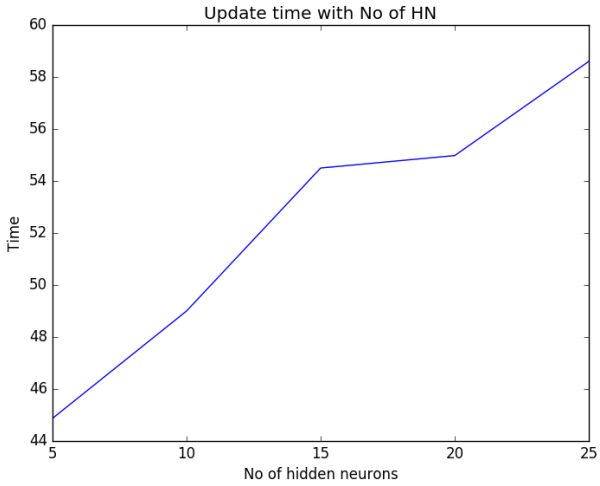








From the figures above, we can see that when n = 10 the performance is best. The number of neurons may lead to underfitting while too many neurons may result in overfitting. The following figure shows the time to update parameters.



1.7 Question 4

Question 4 is to find the optimal decay parameter. According to last 2 parts, the batch size is set as 4 and number of neurons is 10.

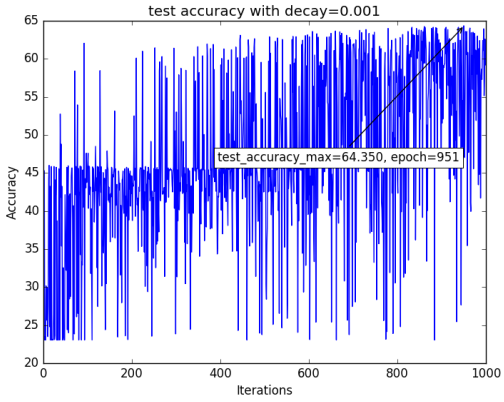
1.7.1 Method

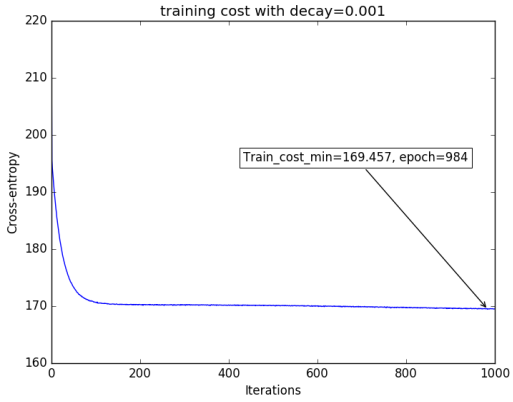
Method is similar as the last part. The decay parameters are from the list decayA.

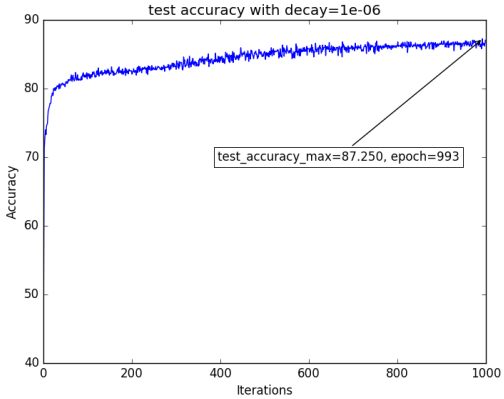
```
decayA = [1e-3,1e-6,1e-9,1e-12,0]
times = []
Accuracy_plot = []

for decay in decayA:
  batch_size = 4
  hnno = 10
  test_accuracy = []
  train_cost = []
  print('Decay parameter = '+str(decay)+' running')
  tnt(batch_size, hnno, decay, trainX, trainY)
  plots(decay)
decayplot(decayA)
```

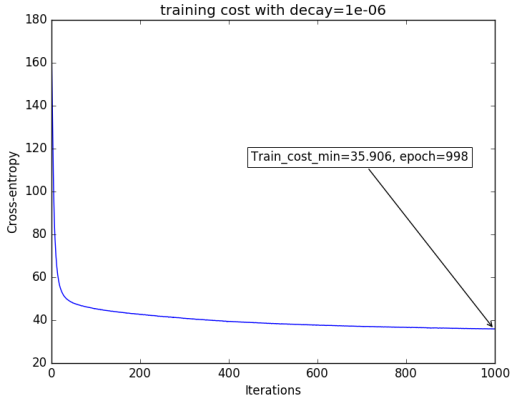
1.7.2 Result

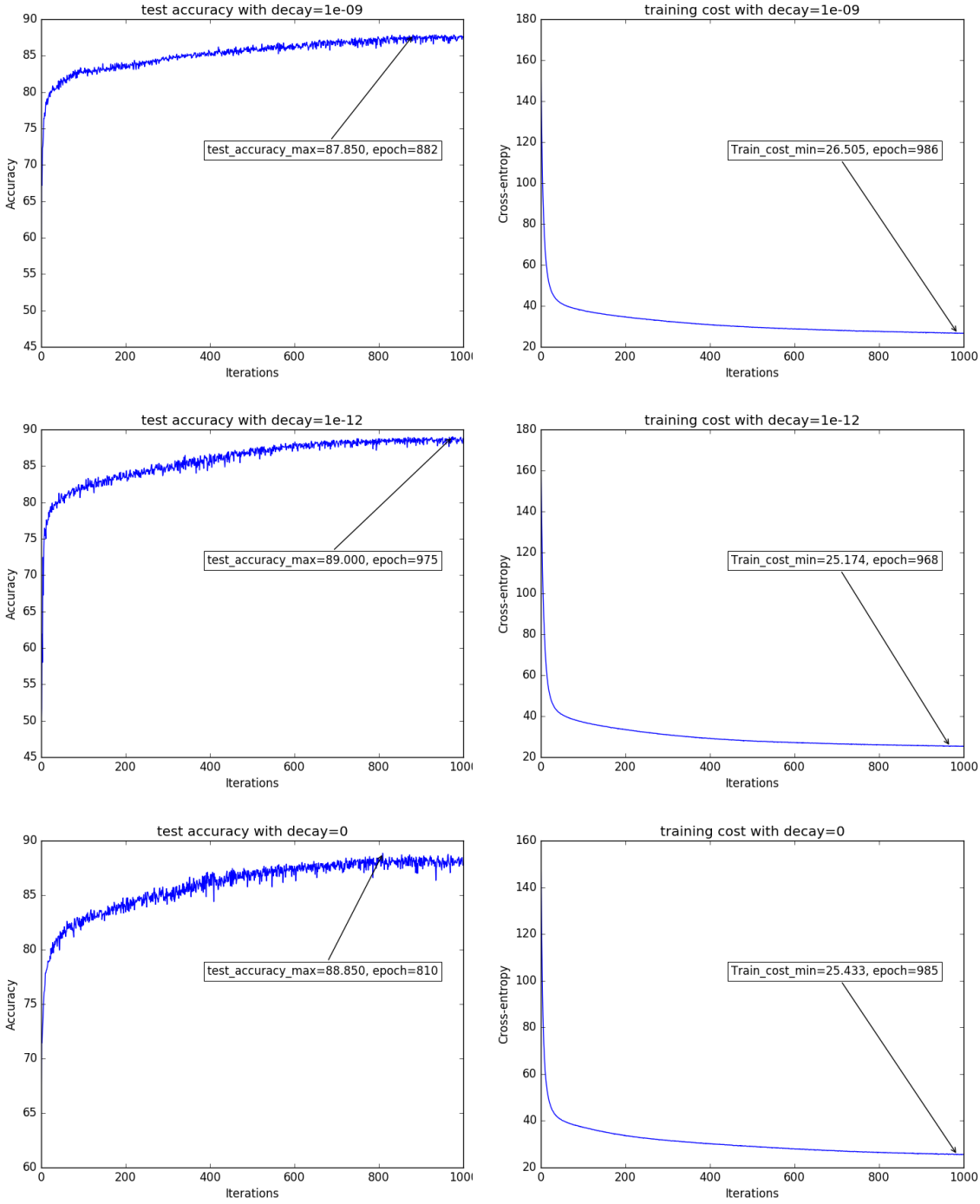
The following figures show the training errors and test accuracies against the number of epochs with different number of hidden-layer neurons.



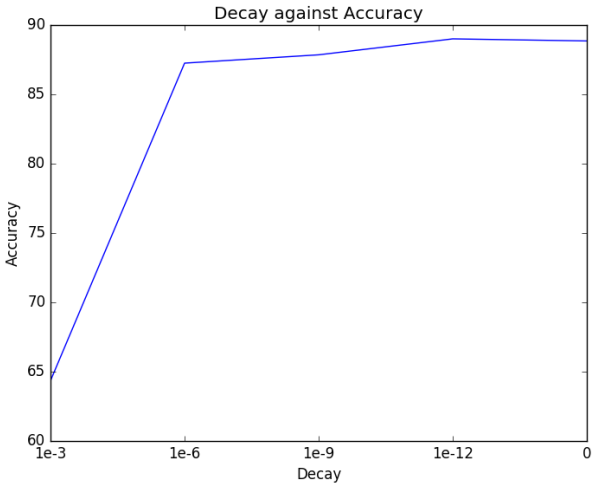








Weight decay is a regularization term that penalizes big weights. From the above figures, we can see that the result is extremely bad when decay = 0.001 is used. This might be due to underfitting since the regularization term is relatively large for the network. When choose decay = 10^{-12}, the the training performance is the best. The following curve shows the test accuracy for different decay values. As the figure shown, decay = 10^{-12} also provides the best test result.



1.8 Question 5

Question 5 is to design a 4-layer network with two hidden-layers and compare the performance of it with 3-layers network.

1.8.1 Method

The model contains 4 layers, input layer, 2 hidden layers and output layer. The activation function for two hidden layers is logistic activation. The output layer uses softmax function. The weights and bias are random initialized. We set the default batch size as 32. We use Stochastic gradient descent (shortened to SGD) with mini batch to train the network. The cost function is defined as the mean of the crossentropy of prediction and actual value then plus regularization term. Once the predefined Theano function train is called, the cost will be computed then the weight and bias will be updated by the SGD function. The learning rate $\alpha = 0.01$. The decay parameter $\beta = 10^{-6}$.

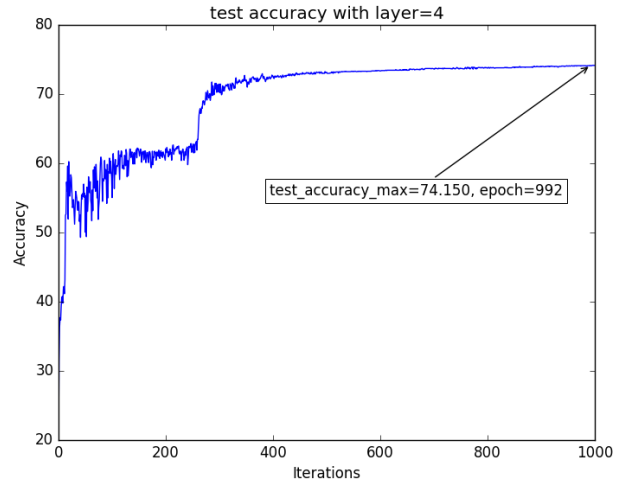
```
# theano expressions
X = T.matrix() #features
Y = T.matrix() #output

#weights and biases from input to 1st-hidden layer
w1, b1 = init_weights(36, 10), init_bias(10)
#weights and biases from 1st-hidden to 2nd-hidden layer
w2, b2 = init_weights(10, 10), init_bias(10)
#weights and biases from 2nd-hidden to output layer
w3, b3 = init_weights(10, 6, logistic=False), init_bias(6)

h1 = T.nnet.sigmoid(T.dot(X, w1) + b1)
h2 = T.nnet.sigmoid(T.dot(h1, w2) + b2)
py = T.nnet.softmax(T.dot(h2, w3) + b3)

y_x = T.argmax(py, axis=1)

cost = T.mean(T.nnet.categorical_crossentropy(py, Y)) + decay*(T.sum(T.sqr(w1)+T.
  sum(T.sqr(w2)+T.sum(T.sqr(w3)))))
params = [w1, b1, w2, b2, w3, b3]
updates = sgd(cost, params, learning_rate)

# compile
train = theano.function(inputs=[X, Y],
                        outputs=cost,
                        updates=updates,
```

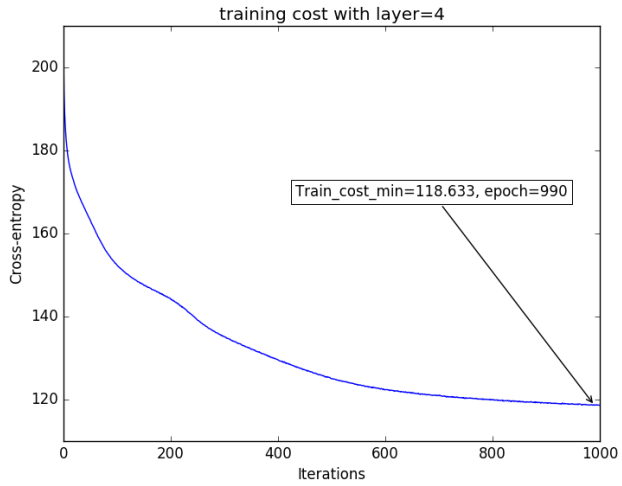
```
                                allow_input_downcast=True)
predict = theano.function(inputs=[X],
                          outputs=y_x,
                          allow_input_downcast=True)
```

At each epoch, the data will be shuffled. An average cost for this epoch is computed and stored in the list train_cost. This part of code is same as 1.4.1.

1.8.2 Result

The following figures show the accuracy and train cost of 4-layers network.





1.8.3 Discussion

Obviously, 4-layers network cannot perform as well as 3-layers network. The 3-layers network gets about 5 percentage point higher than 4-layers. This might because we didn't optimize the parameters used for 4-layers. Overfitting since more neurons are involved can also be a reason. According to the figures, it seems that 4-layers network fell into some local optimum.

2 Part 2

2.1 Introduction

The second part of the project is to use Neuron Networks to deal with regression problem. The goal is to predict the house price in California by training the network using the provided California Housing dataset. Since different parameters will affect the overall performance, it's necessary to try out several of them and select the one given the best result. In this section, we use multilayer feedforward networks to solve the problem.

2.2 Notes

For Part 2, the main tools we use are Theano, numpy, scikit-learn and matplotlib. All the training and performance measurements are done on a Macbook Pro with 2.6 GHz Intel Core i5.

2.3 Data Preparation

The data is read from file *cal_housing.data* and separated into two parts with training and testing ratio of 0.7:0.3. Then perform scaling on training and testing data to let the transformed data be distributed within a certain range. Note that we use the min-max value computed from train data for the scaling on test data. This is because we pretend that the test data is unseen data which is only used for evaluating the performance of the network.

```
def scale(X, X_min, X_max):
    return (X - X_min)/(X_max - X_min)

def read_data():
    #read and divide data into test and train sets
    cal_housing = np.loadtxt('cal_housing.data', delimiter=',')
    X_data, Y_data = cal_housing[:,:8], cal_housing[:,-1]
    Y_data = (np.asmatrix(Y_data)).transpose()

    X_data, Y_data = shuffle_data(X_data, Y_data)
    #separate train and test data
    m = 3*X_data.shape[0] // 10
    testX, testY = X_data[:m],Y_data[:m]
    trainX, trainY = X_data[m:], Y_data[m:]

    # scale data
    trainX_max, trainX_min =  np.max(trainX, axis=0), np.min(trainX, axis=0)

    trainX = scale(trainX, trainX_min, trainX_max)
    testX = scale(testX, trainX_min, trainX_max)

    return trainX, trainY, testX, testY
```

2.4 Question 1

The first question is to develop a basic 3-layer feedforward neuron network with the hidden layer having 30 neurons. The plots for training error and prediction error are shown in the result section.

2.4.1 Method

We set the parameters for the model as follow,

batch_size = 32; learning_rates = 1e-4; epochs = 1000; no_hidden_neurons = 30.

The model contains 3 layers, input layer, hidden layer and output layer, and the activation functions for the hidden layer and output layer are sigmoid and linear respectively with random initialized weights and bias. We use mini-batch gradient descent to train the network with the loss function

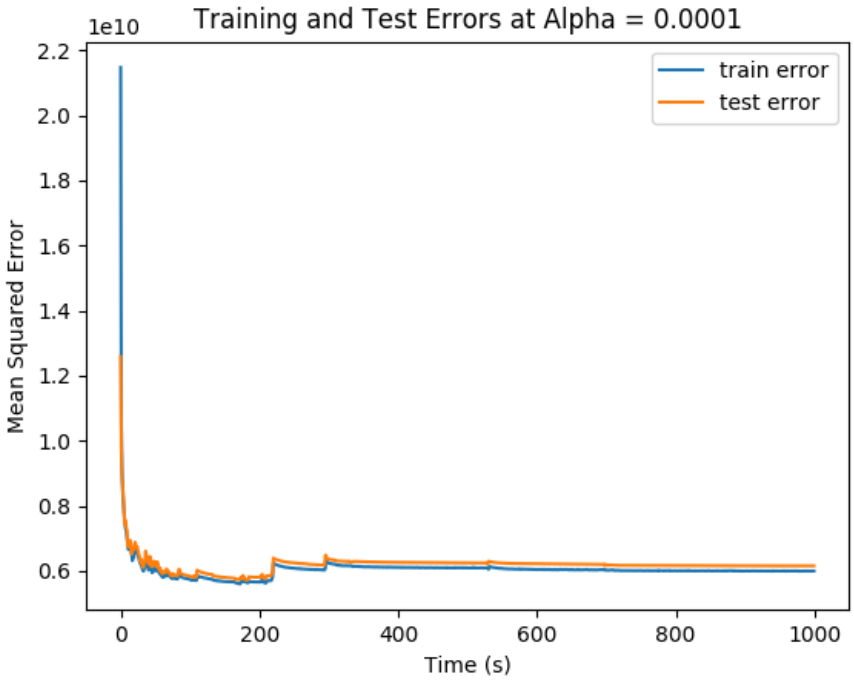
defined as the absolute mean square error between the generated predictions(d), and the actual value(y). Once the predefined theano function *train* is being called, the corresponding cost would be computed, meanwhile, the weights and bias are updated by the gradient values.

```
# initialize weights and biases for hidden layer(s) and output layer
w_o = theano.shared(np.random.randn(no_hidden1)*.01, floatX )
b_o = theano.shared(np.random.randn()*.01, floatX)
w_h1 = theano.shared(np.random.randn(no_features, no_hidden1)*.01, floatX )
b_h1 = theano.shared(np.random.randn(no_hidden1)*0.01, floatX)

#Define mathematical expression:
h1_out = T.nnet.sigmoid(T.dot(x, w_h1) + b_h1)
y = T.dot(h1_out, w_o) + b_o

cost = T.abs_(T.mean(T.sqr(d - y)))
accuracy = T.mean(d - y)

#define gradients
dw_o, db_o, dw_h, db_h = T.grad(cost, [w_o, b_o, w_h1, b_h1])

train = theano.function(
        inputs = [x, d],
        outputs = cost,
        updates = [[w_o, w_o - alpha*dw_o],
                   [b_o, b_o - alpha*db_o],
                   [w_h1, w_h1 - alpha*dw_h],
                   [b_h1, b_h1 - alpha*db_h]],
        allow_input_downcast=True
        )
```

At each epoch, shuffle the data and train the model using mini batches. An average cost for this epoch is computed and stored in the train_cost list. The training procedure is shown as follow,

```
for iter in range(epochs):
    if iter % 100 == 0:
        print(iter)

    trainX, trainY = shuffle_data(trainX, trainY)
    n = len(trainX)
    cost = 0
    for start in range(0, n, batch_size):
        end = start + batch_size
        cost += train(trainX[start:end], np.transpose(trainY[start:end]))
    train_cost[iter] = cost/(n // batch_size)
    pred, test_cost[iter], test_accuracy[iter] = test(testX, np.transpose(testY))
```

2.4.2 Result



As we can see, the error curve is a bit noisy at beginning. This probably because when use mini-batch, the frequent updates can result in a noisy gradient signal, which may cause the model parameters and in turn the model error to jump around (having a higher variance over training epochs). One positive point for the noisy update process is that it can allow the model to avoid local optimum. The model eventually converges at a later point.

2.5 Question 2

By choosing different learning rates to do gradient update, the performance varies. The second question is to find the optimum learning rate for the network.

2.5.1 Method

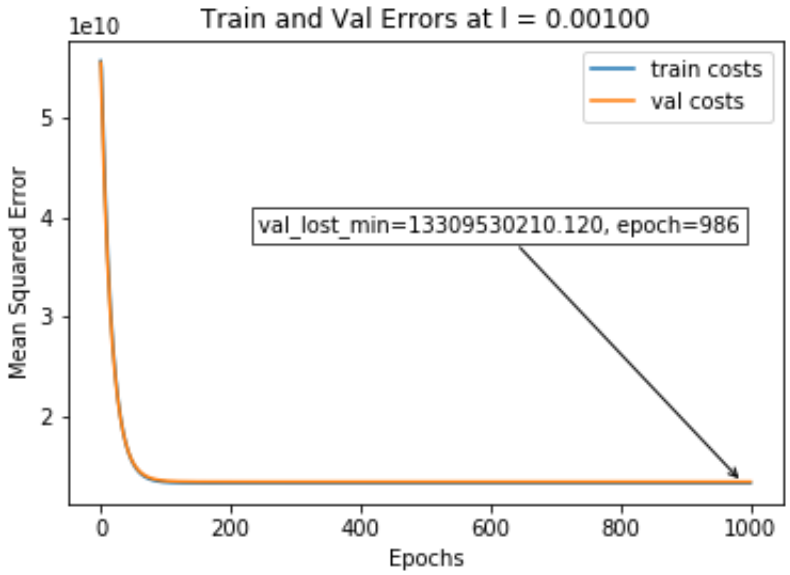
Five-fold cross validation is implemented using Kfold method from scikit-learn. The basic idea behind is that the training data will be divided into five equally sized parts. Each time one set of the data will be picked for validation purpose, the rest is used for training. Repeat the procedure for 5 times, with each of the 5 parts is used exactly once as the validation data. In the end, an average result will be generated from the previous five as a single estimate. Note that in each iteration, a new model is generated for training. We tried different learning rate from the list $\{10^3$, $0.5*10^3$, 10^4, $0.5*10^4$, $10^5\}$. Use a for loop to iterate through the list and execute the code.

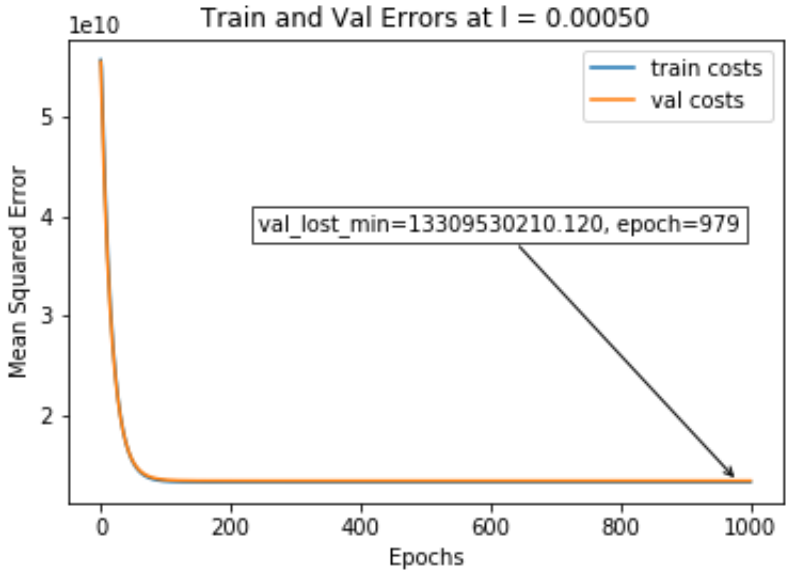
```
for l in learning_rates:
    alpha.set_value(l)
    for train_index, val_index in kf.split(trainX):
        val_set_X = trainX[val_index]
        val_set_Y = trainY[val_index]
        train_set_X = trainX[train_index]
        train_set_Y = trainY[train_index]
```

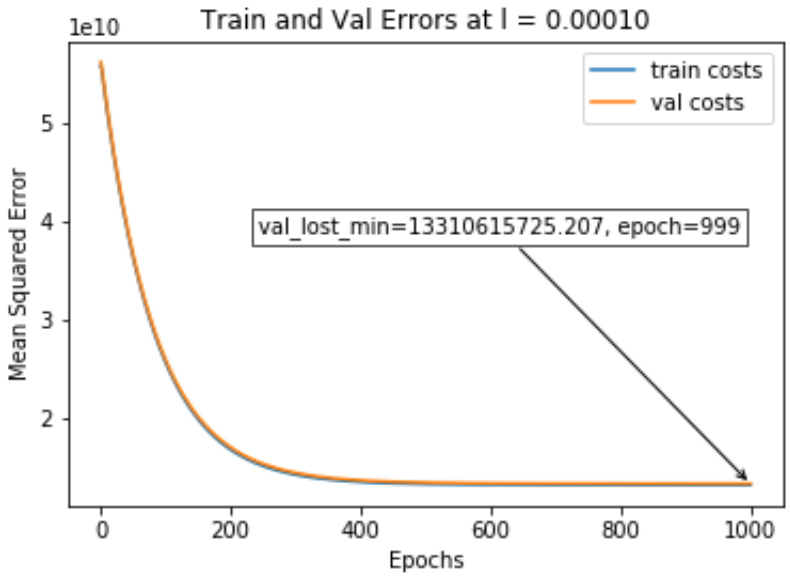
```
        new_model()
        for iter in range(epochs):
    # ... followed by the same code
```

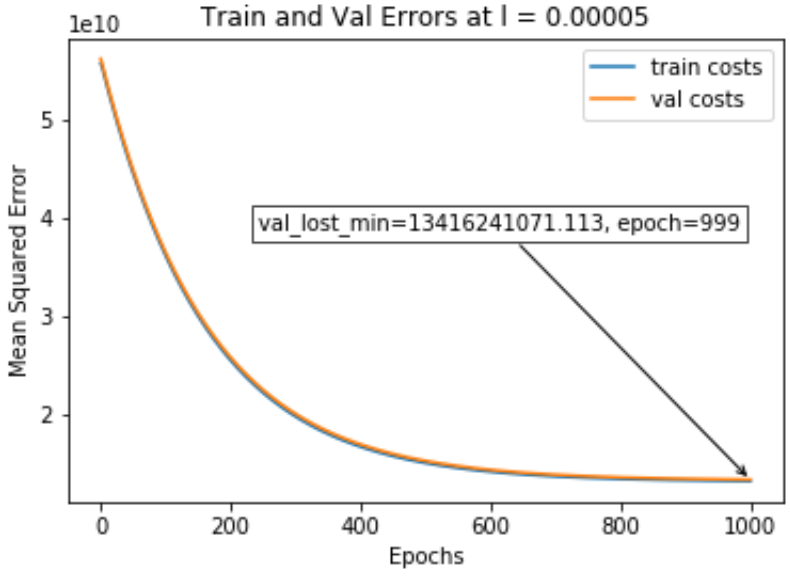
2.5.2 Result

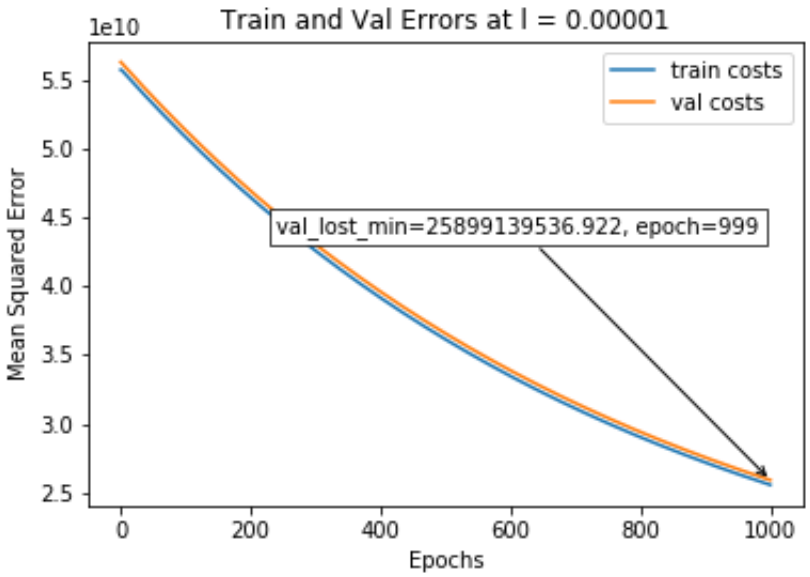
The following figures show the training and validation errors for models under different learning rates.



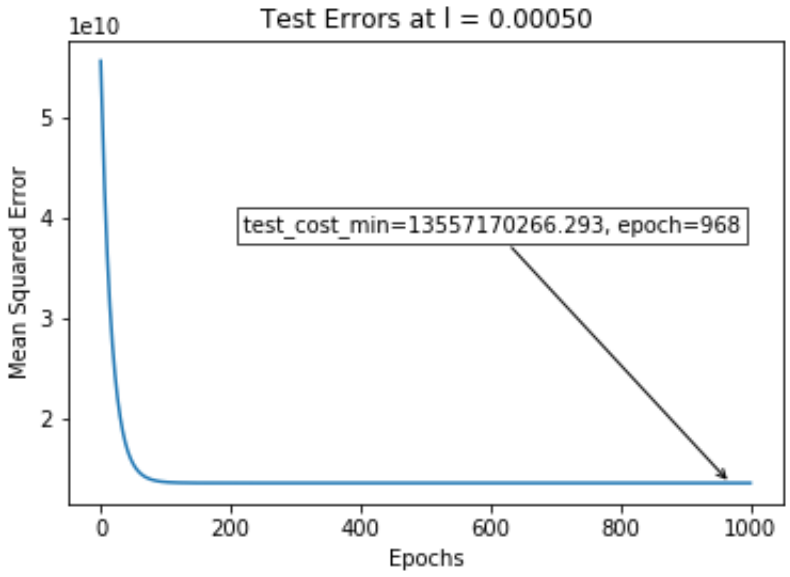



Train and Val Errors at l = 0.00010
1e10
train costs
val costs
val_lost_min=13310615725.207, epoch=999
Mean Squared Error
Epochs
0
200
400
600
800
1000
2
3
4
5


Train and Val Errors at l = 0.00005
1e10
train costs
val costs
val_lost_min=13416241071.113, epoch=999
Mean Squared Error
Epochs
0
200
400
600
800
1000
2
3
4
5



As shown above, we can see that **l = 5e-4** gives the best performance since it takes much less time to converge. Having very large learning rate may not decrease the gradient on every iteration and thus it may not converge well, or it might take some time to converge. While although small learning rate will lead to convergence in the end, it's always much slower. For example, when l=1e-5, it's obviously not converged at epoch 1000. The test errors curve with optimum learning rate is shown as follow,



2.6 Question 3

The number of neurons also determines the final performance, thus it's important to select a proper amount of neurons for the hidden layer.

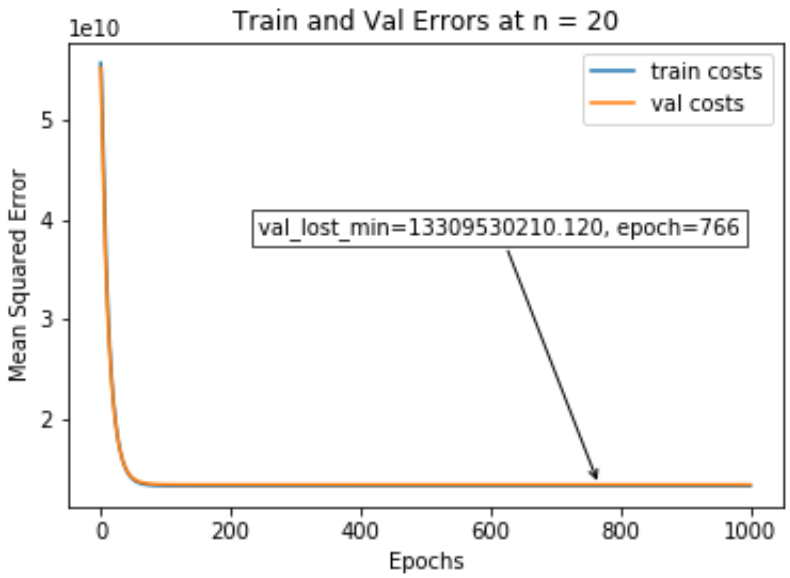
2.6.1 Method

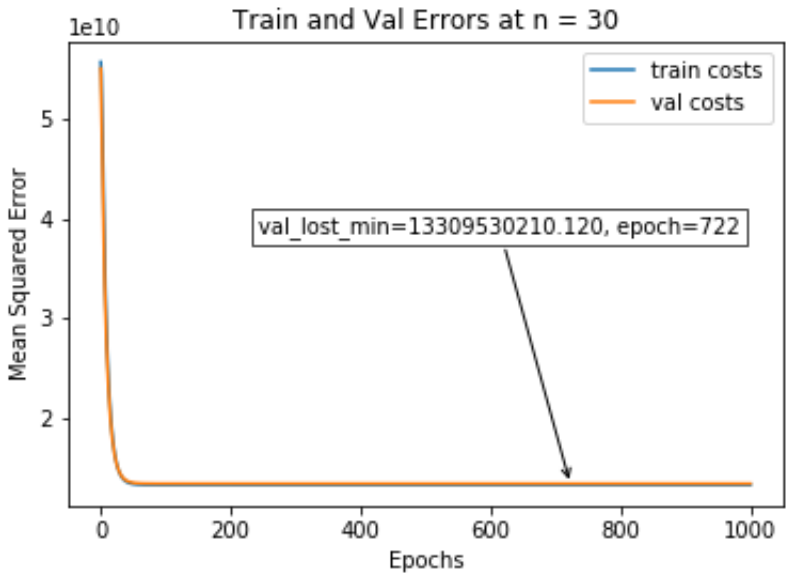
We set learning rate to be 5e-4, which is the optimal one derived from Question 2, and use a for loop to evaluate the model with different number of hidden neurons. Same as Q2, we perform the 5 fold cross validation on the train data for choosing the best model.

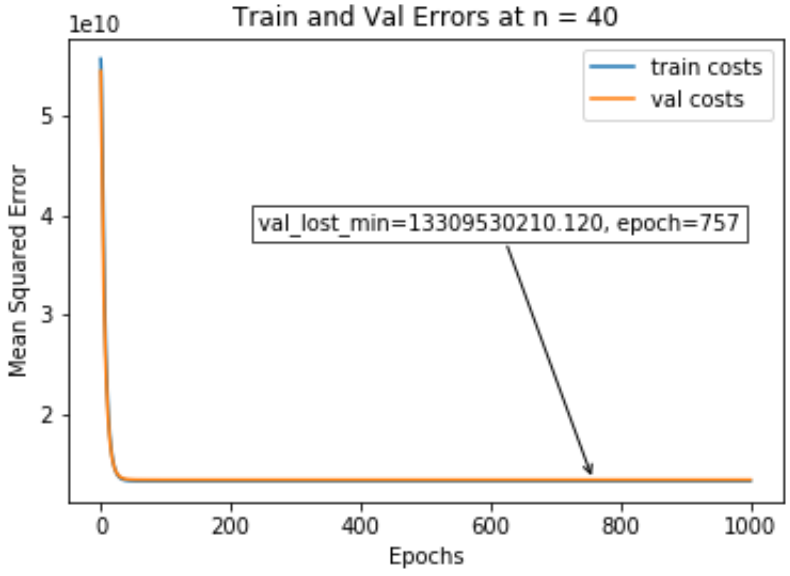
```
for no_hidden1 from no_hidden1_list:
    for train_index, val_index in kf.split(trainX):
        val_set_X = trainX[val_index]
        val_set_Y = trainY[val_index]
        train_set_X = trainX[train_index]
        train_set_Y = trainY[train_index]
        new_model()
        for iter in range(epochs):
   # ... followed by the same code
```

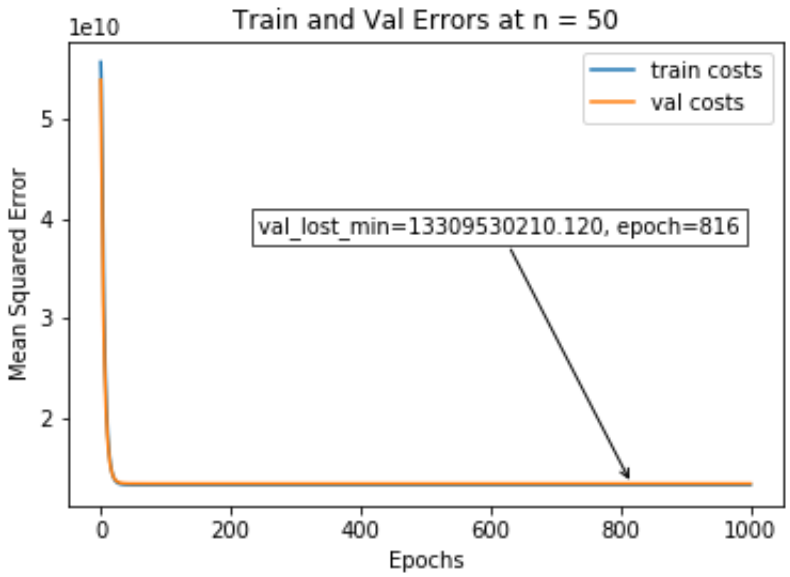
2.6.2 Result

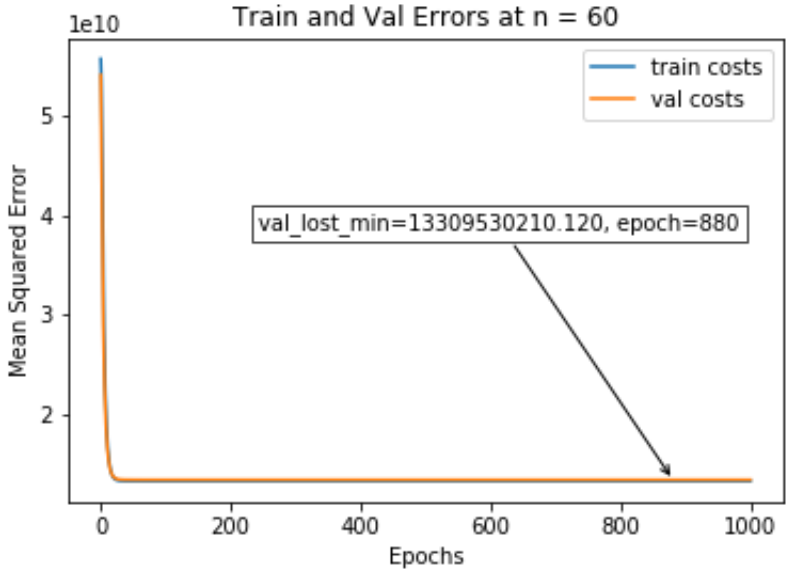
The following figures show the training and validation error for the models under different hidden-layer neurons.



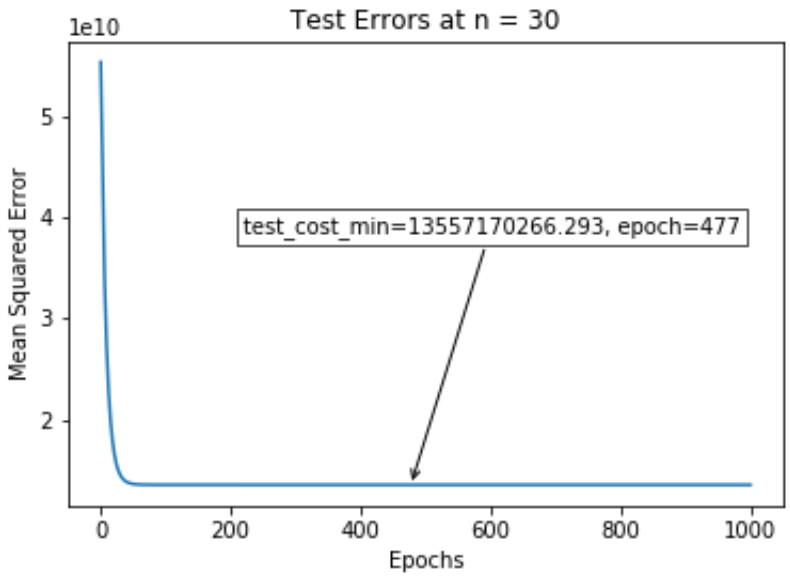

Train and Val Errors at n = 30
1e10
train costs
val costs
val_lost_min=13309530210.120, epoch=722
Mean Squared Error
Epochs
0
200
400
600
800
1000
2
3
4
5


Train and Val Errors at n = 40
1e10
train costs
val costs
val_lost_min=13309530210.120, epoch=757
Mean Squared Error
Epochs
0
200
400
600
800
1000
2
3
4
5





As we can see, the validation costs are the same for all the models. But each converges at a different time. The overall trend is that along with the number of neurons in a hidden layer getting larger, the time taken for convergence is longer. But it seems that n=30 converges faster than n=20. This is probably because that a bit more neurons would help to learn more relevant features from data, and thus could be earlier to reach optimum. Based on the result, we pick the one with n=30. The figure for test errors with no_hidden = 30 is shown as follow,



2.7 Question 4

The main purpose for this part is to see whether a deeper network will bring a better result or not. Implement a four-layer neural network and five-layer neural network using the optimal parameters.

2.7.1 Method

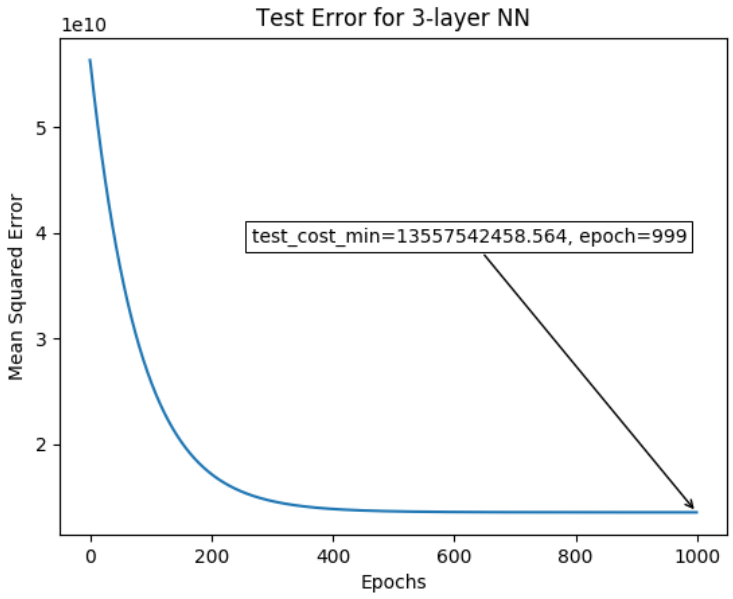
On top of the previous implementation, we further add in another hidden layer with 20 neurons. The modifications are done based on the network structure as well as the connections between each layer. Similar modification is done for the five-layer neural network. For learning rate as well as number of neurons in the first hidden layer, we use l = 0.0001, no_hidden1 = 30 which are the optimal ones chosen before. Note that in the section, we removed the k-fold cross validation because we are purely looking at the impact of depth of the network, rather than selecting the best model.

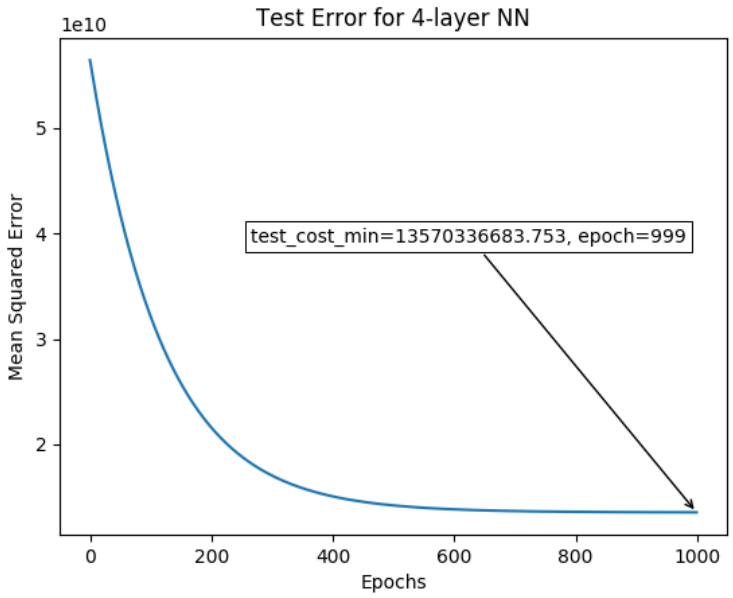
```
# define gradients
dw_o, db_o, dw_h1, db_h1, dw_h2, db_h2 = T.grad(cost, [w_o, b_o, w_h1, b_h1, w_h2,
    b_h2])

# define mathematical expression:
h1_out = T.nnet.sigmoid(T.dot(x, w_h1) + b_h1)
h2_out = T.nnet.sigmoid(T.dot(h1_out, w_h2) + b_h2)
y = T.dot(h2_out, w_o) + b_o

# modify the train function
train = theano.function(
        inputs = [x, d],
        outputs = cost,
        updates = [[w_o, w_o - alpha*dw_o],
                   [b_o, b_o - alpha*db_o],
                   [w_h1, w_h1 - alpha*dw_h1],
                   [b_h1, b_h1 - alpha*db_h1],
                   [w_h2, w_h2 - alpha*dw_h2],
                   [b_h2, b_h2 - alpha*db_h2]],
        allow_input_downcast=True)

# generate new weight and bias
```

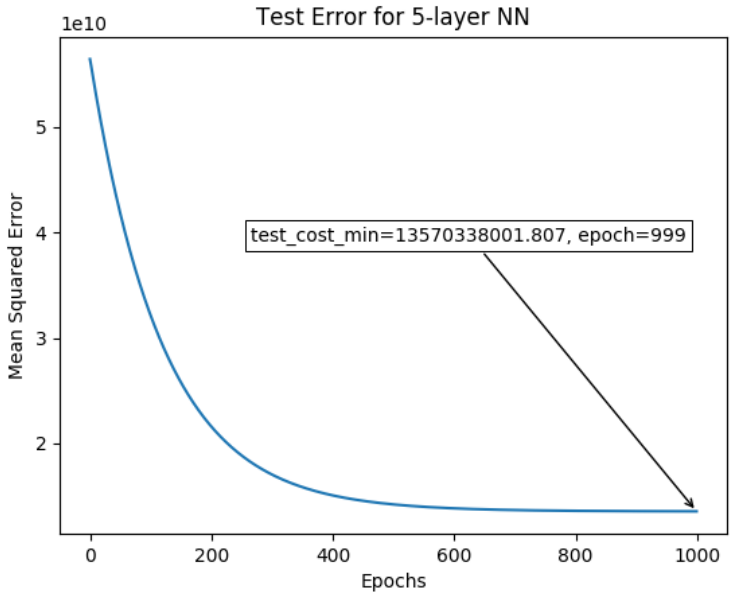
```
22 def new_model(no_hidden1, no_hidden2, no_features):
23     w_o.set_value(np.random.randn(no_hidden2)*.01)
24     b_o.set_value(np.random.randn()*.01)
25     w_h1.set_value(np.random.randn(no_features, no_hidden1)*.01)
26     b_h1.set_value(np.random.randn(no_hidden1)*.01)
27     w_h2.set_value(np.random.randn(no_hidden1, no_hidden2)*.01)
28     b_h2.set_value(np.random.randn(no_hidden2)*.01)
```

2.7.2 Result

The test errors curves for the networks with different depth are shown as follow,







2.7.3 Discussion

Compare with the three-layer network, it seems that the overall performance drop a bit when the network getting deeper since it takes longer time to converge. The result may be better if we choose another set of parameters for deeper networks. But if with a certain predefined parameters, deeper network may not always outperform others, especially with increasingly computational effort.