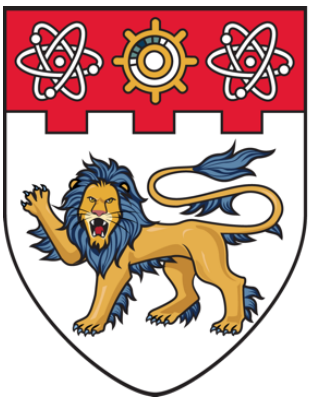


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042 NEURAL NETWORKS
PROJECT 1 REPORT

SHAO YIYANG (U1420722G)
ZHAO QINGMEI (U1422368A)

School of Computer Science and Engineering

Academic Year 17/18, Semester 1

Contents

1 Part A: Classification Problem

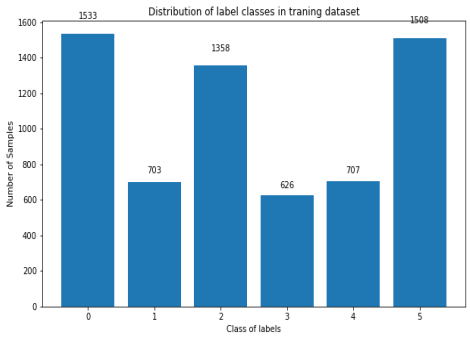
1.1 Problem Introduction

The problem of classification of Landsat satellite dataset aims to predict the class labels of the centre pixels, given the multi-spectral values. In this part, several tools were used: Theano for buiding and training the network and matplotlib for plotting the graphs. All performance data is collected by running the code on an Intel Quad Core i7-4900MQ CPU.

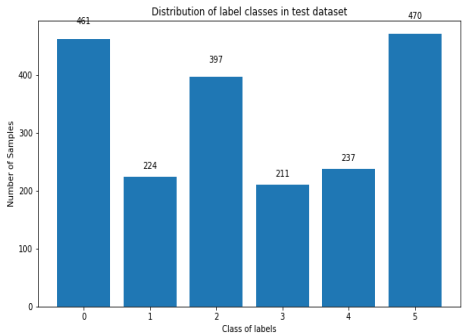
The dataset from Statlog (Landsat Satellite) Dataset [2] consists of the 36 attributes from the multi-spectral values of pixels in 3x3 neighbourhoods, and the classification associated with the central pixel in each neighbourhood. All the attributes are numerical, in the range 0 to 255. There are 6 existing classes which including red soil, cotton crop, grey soil, damp grey soil, soil with vegetation stubble, very damp grey soil.

To evaluate the performance of our model, the dataset is divided into training dataset and testing dataset. There are 4435 samples in training dataset and 2000 samples in testing dataset.

The distribution of label classes in the datasets are shown as Figure 1a, Figure 1b and Figure 2.



(a) The distribution of label classes in training dataset



(b) The distribution of label classes in testing dataset

Figure 1: The distribution of Label Classes in training and testing dataset

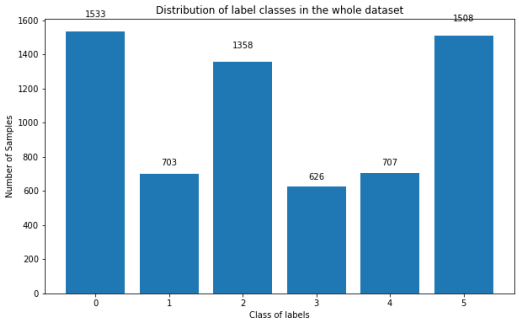


Figure 2: The distribution of label classes in the whole dataset

1.2 Problem A1 - Pre-processing

Before the process of training model, we need to do Pre-processing to get data ready to be feeded into the multi-layer feed-forward networks for solving classification problem.

1.2.1 Scaling & Normalization

We used Min-Max Normalization to normalize all the attributes in the datasets, as shown in the code listing section 1. The minimum and maximum values for each attributes are the minimum and maximum values of attributes in the training dataset. Hence, the testing dataset is normalized by the parameters as the training dataset. After normalization, all the values in the attributes are basically evenly ranged from 0 to 1. For example, the distribution of the first attribute after normalization is shown as Figure 3.

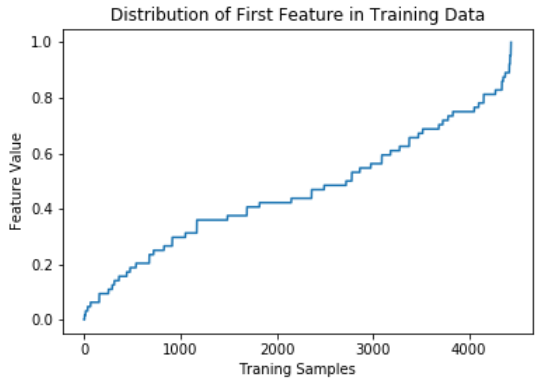


Figure 3: Distribution of First Attribute After Normalization in Training Data

```
# scale data
X_min = None
X_max = None
def scale(X):
    """
    Min-Max Normalization
    """
    return (X - X_min)/(X_max-np.min(X, axis=0))

# Use min max value of traning data to do scaling
X_min = np.min(trainX, axis=0)
X_max = np.max(trainX, axis=0)

trainX = scale(trainX)
testX = scale(testX)
```

Listing 1: Min-Max Normalization

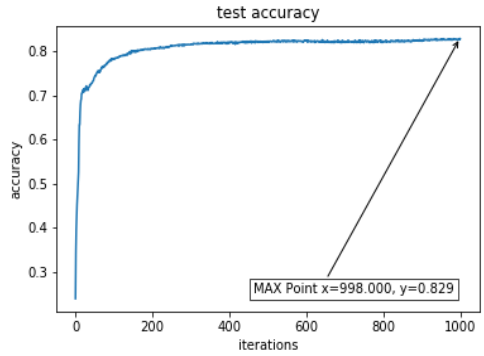
1.2.2 Implement the Neural Network

A 3-layer feedforward neural network which is consisting of a hidden-layer of 10 neurons having logistic activation function and an output softmax layer, is implemented by Theano Library as the code listing section 2 shown below.

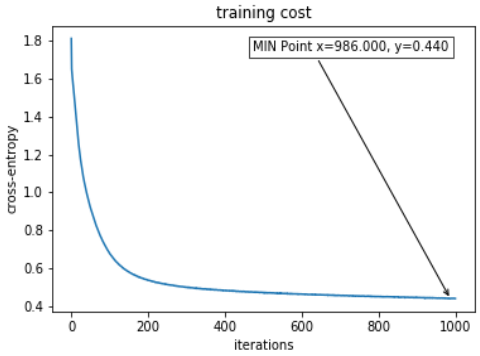
```
def train_network(trainX, trainY, testX, testY, decay, learning_rate, epochs, batch_size, num_neurons):
    """
    Struct the model with 1 hidden layer
    and train the model
    """
    # theano expressions
    X = T.matrix() #features
    Y = T.matrix() #output
    w1, b1 = init_weights(36, num_neurons), init_bias(num_neurons)
    #weights and biases from input to hidden layer
    w2, b2 = init_weights(num_neurons, 6, logistic=False), init_bias(6)
    #weights and biases from hidden to output layer

#     activation
    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)

    # train and test
    n = len(trainX)
    test_accuracy = []
    train_cost = []
    batch_time_used = []

    for i in range(epochs):
        if i % 100 == 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)):
            start_time = time.time()
            cost += train(trainX[start:end], trainY[start:end])
            batch_time_used.append(time.time()*1000-start_time*1000)

        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))
    return train_cost, test_accuracy, np.mean(batch_time_used), np.sum(batch_time_used)
```

Listing 2: 3-layer feedforward neural network

By setting decay = 1e-6, learning_rate = 0.01, epochs = 1000, num_neurons = 10, batch_size = 32, we could get a initial result of the model. As shown in the Figure 4a and 4b, the model achieves its highest accuracy 82.9% at 998th epoch and the minimum training cost 0.440 at the 986th epoch.



(a) Test Accuracy Against the Number of Epochs



(b) Training Cost Against the Number of Epochs

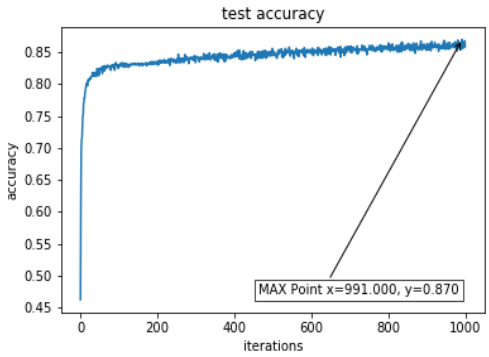
Figure 4: Result of Initial Model

1.3 Problem A2 - Selecting the Optimal Batch Size

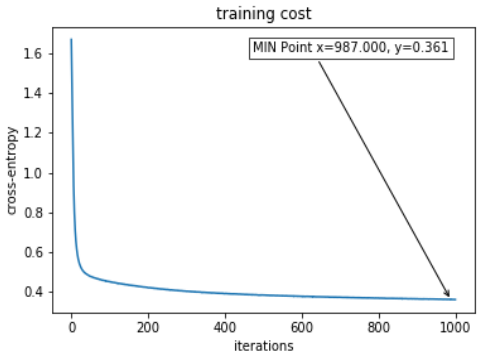
To find the optimal batch size, we used the setting of decay = 1e-6, learning_rate = 0.01, epochs = 1000, num_neurons = 10 and batch_size = [4, 8, 16, 32, 64]. We will choose one value for batch size from [4, 8, 16, 32, 64] which has the best performance.

1.3.1 Training Error and Test Accuracy

The following section shows the graph of the training error and test accuracy against number of epochs for the 3-layer network for each batch size (4,8,16,32,64), as shown in Figure 5, 6, 7, 8, and 9.

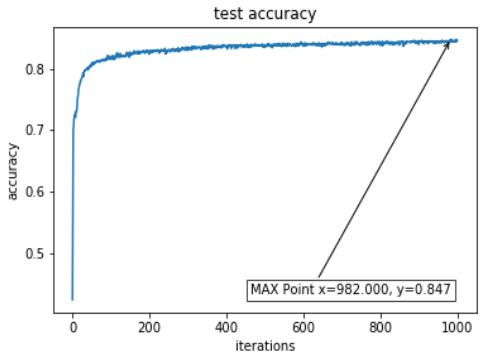


(a) Test Accuracy Against the Number of Epochs

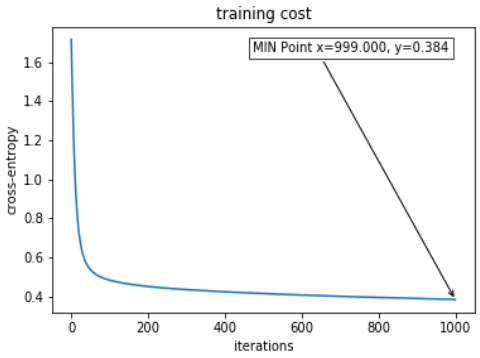


(b) Training Cost Against the Number of Epochs

Figure 5: Result of Model with Batch_Size = 4

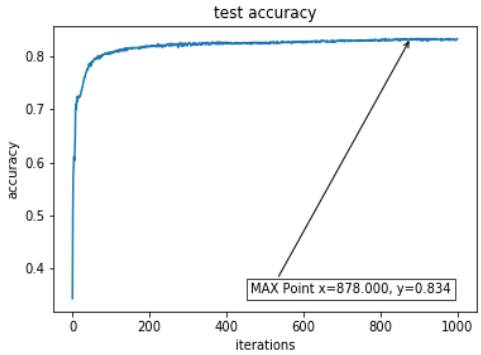


(a) Test Accuracy Against the Number of Epochs

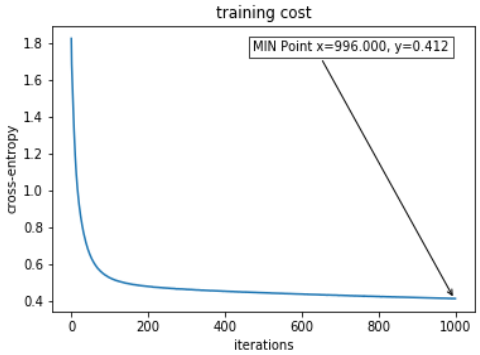


(b) Training Cost Against the Number of Epochs

Figure 6: Result of Model with Batch_Size = 8

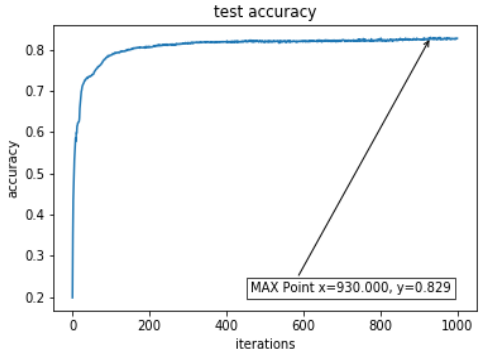


(a) Test Accuracy Against the Number of Epochs

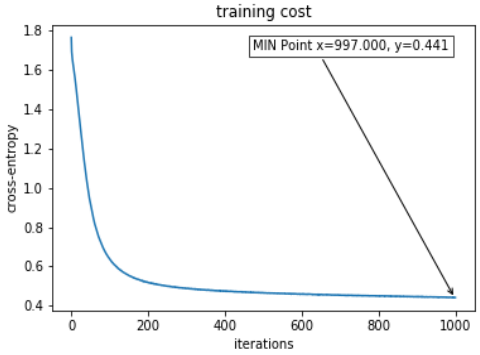


(b) Training Cost Against the Number of Epochs

Figure 7: Result of Model with Batch_Size = 16

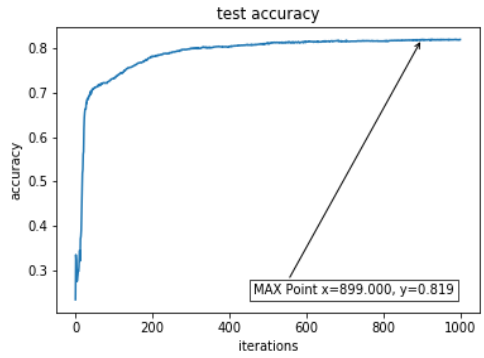


(a) Test Accuracy Against the Number of Epochs

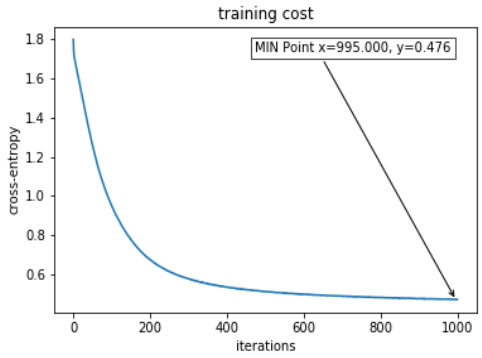


(b) Training Cost Against the Number of Epochs

Figure 8: Result of Model with Batch_Size = 32



(a) Test Accuracy Against the Number of Epochs



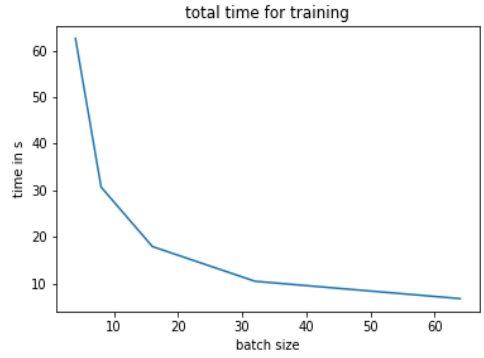
(b) Training Cost Against the Number of Epochs

Figure 9: Result of Model with Batch_Size = 64

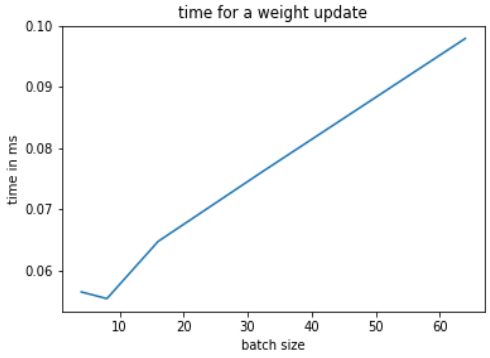
1.3.2 Time Taken

The following section shows the graph of the time taken to update parameters of the network for each sample for different batch sizes (Figure 10b) (The time taken for training one batch of data samples), and the graph of total time taken to train the model for different batch size (Figure 10a).

As shown in the Figure 10b, the time taken to update parameters of the network is directly proportional to the batch size. As larger the batch size, it needs to compute the loss from more data samples and hence, the longer time is required.



(a) Total time to train the model for different batch size



(b) Time to update parameters of the network for each sample for different batch sizes

Figure 10: Time Comparison of Models with Different Batch Size

1.3.3 Rational for Selecting the Optimal Batch Size

According to Figure 10a, the longest training time is about 1 minute which is acceptable. Hence, we have decided that we will select the optimal batch size by only looking at accuracy, ignoring the time taken to train the model.

batch size = the number of training examples in one forward/backward pass. As batch size = 4 increased to batch size = 64, the highest accuracy is decreased from 0.870 (batch size =

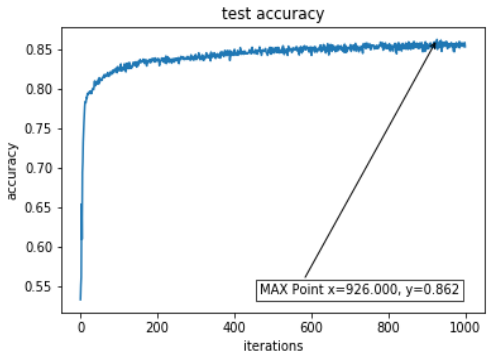
4), 0.847, 0.834, 0.829, and finally 0.819 (batch size = 6). This observation is same as the observation of Keskar Nitish Shirish *It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize. The lack of generalization ability is due to the fact that large-batch methods tend to converge to sharp minimizers of the training function* [1]. Although when the batch size is small it takes longer time to train the model with same number of epochs, the total time for our model is still quite short and in an acceptable period of time. Hence, in order to have better generelization ability, we have decided to choose 4 as the optimal batch size, which can produce the highest testing accuracy 0.870.

1.4 Problem A3 - Selecting the Optimal Number of Hidden Neurons

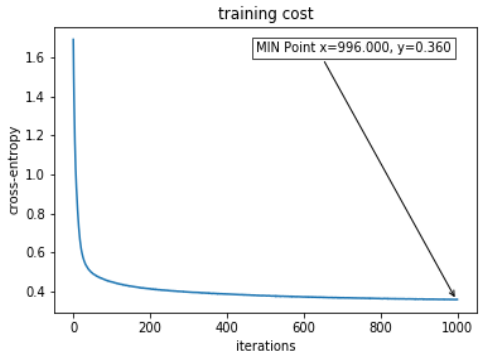
As stated in the section 1.3.3, we have found that batch_size = 4 is the optimal batch size for mini-batch gradient descent. In the following sections, we will start to use optimal batch_size = 4 and the setting decay = 1e-6 learning_rate = 0.01 epochs = 1000 num_neurons = [5, 10, 15, 20, 25] to select the optimal number of hidden neurons in the hidden layer.

1.4.1 Training Error and Test Accuracy

The following section shows the graph of the training error and test accuracy against number of epochs for the 3-layer network for each number of hidden neurons in the hidden layer [5, 10, 15, 20, 25] as shown in Figure 11, 12, 13, 14, and 15.

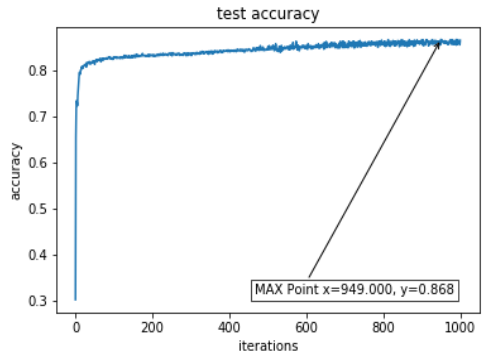


(a) Test Accuracy Against the Number of Epochs

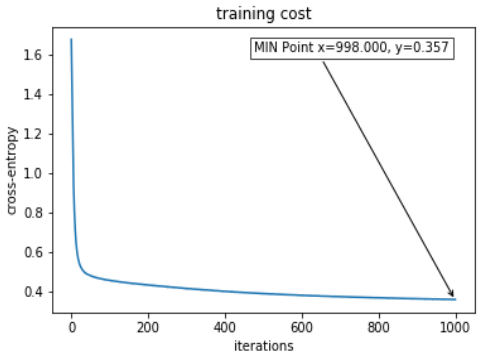


(b) Training Cost Against the Number of Epochs

Figure 11: Result of Model with Number of Neurons = 5 in the Hidden Layer

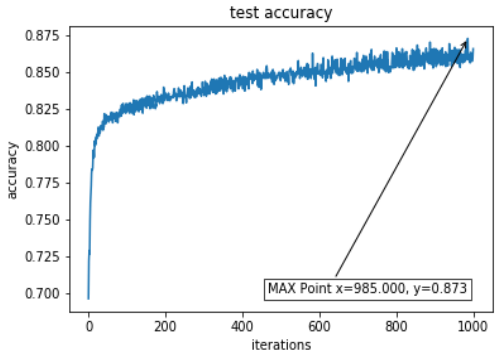


(a) Test Accuracy Against the Number of Epochs

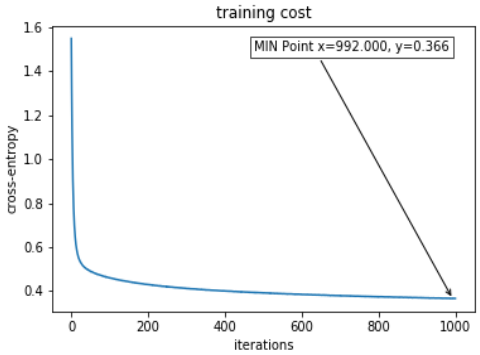


(b) Training Cost Against the Number of Epochs

Figure 12: Result of Model with Number of Neurons = 10 in the Hidden Layer

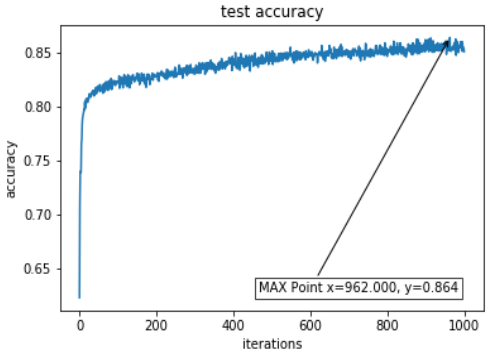


(a) Test Accuracy Against the Number of Epochs

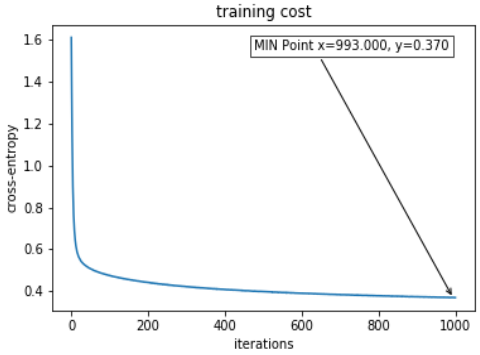


(b) Training Cost Against the Number of Epochs

Figure 13: Result of Model with Number of Neurons = 15 in the Hidden Layer

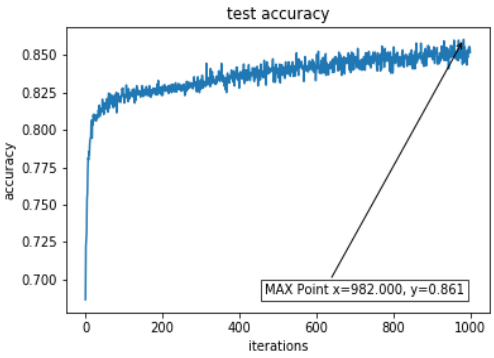


(a) Test Accuracy Against the Number of Epochs

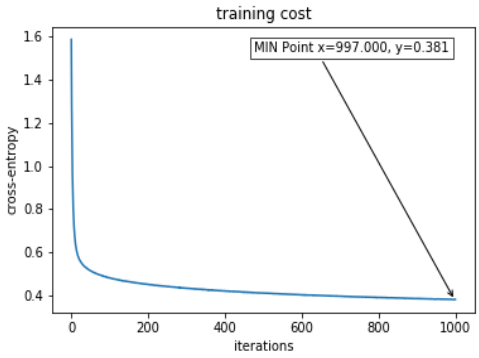


(b) Training Cost Against the Number of Epochs

Figure 14: Result of Model with Number of Neurons = 20 in the Hidden Layer



(a) Test Accuracy Against the Number of Epochs



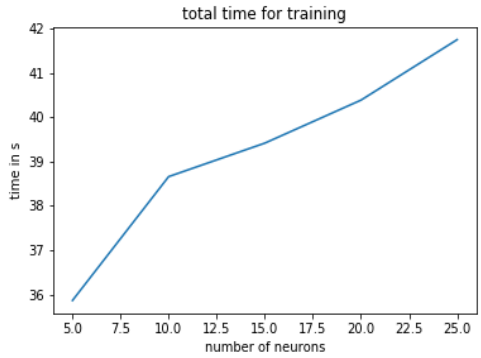
(b) Training Cost Against the Number of Epochs

Figure 15: Result of Model with Number of Neurons = 25 in the Hidden Layer

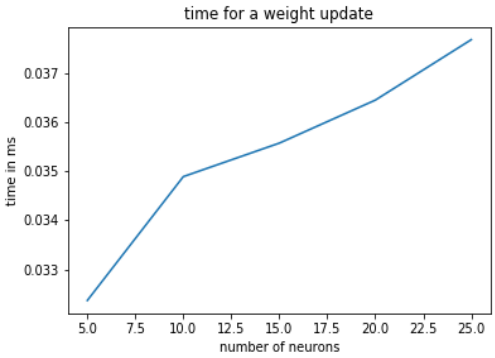
1.4.2 Time Taken

The following section shows the graph of the time taken to update parameters of the network for each sample for different number of hidden neurons (Figure 16b) (The time taken for training one batch of data samples), and the graph of total time taken to train the model for different number of hidden neurons (Figure 16a).

As the number of hidden neurons increased from 5 to 25, the time taken for a weight update and total time for training is also increasing. It is because that as larger number of hidden neurons means there are more parameters to be computed and updated. Hence, longer time is needed to compute more parameters.



(a) Total time to train the model for different number of hidden neurons



(b) Time to update parameters of the network for each sample for different number of hidden neurons

Figure 16: Time Comparison of Models with Different Number of Hidden Neurons

1.4.3 Rational for Selecting the Optimal Number of Hidden Neurons

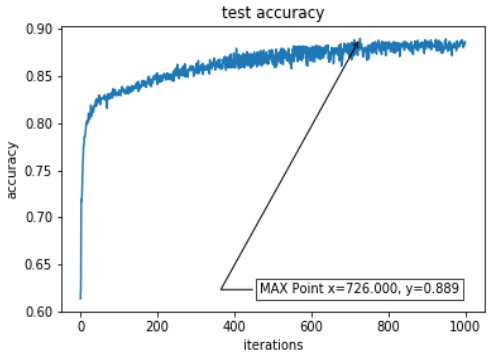
According to Figure 16, the longest training time is around 1 minute which is acceptable. Hence, we have decided that we will select the optimal number of hidden neurons by only looking at accuracy,ignoring the time taken to train the model. We have decided to choose neurons = 15 as the optimal number of hidden neurons, as it could produce the highest accuracy 0.873.

1.5 Problem A4 - Selecting Optimal Decay Parameter

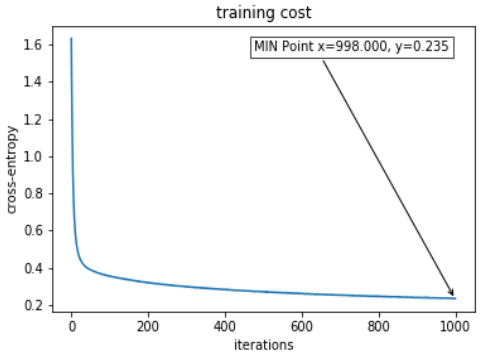
As stated in the section 1.4.3, we have found that neurons = 15 is the optimal number of neurons. In the following sections, we will start to use optimal neurons = 15 and the setting decays = [0, 1e-3, 1e-6, 1e-9, 1e-12] learning_rate = 0.01 epochs = 1000 to select the optimal value of decay parameter.

1.5.1 Training Error and Test Accuracy

The following section shows the graph of the training error and test accuracy against number of epochs for the 3-layer network for each value of decay parameter [0, 1e-3, 1e-6, 1e-9, 1e-12], as shown in Figure 17, 18, 19, 20, and 21.

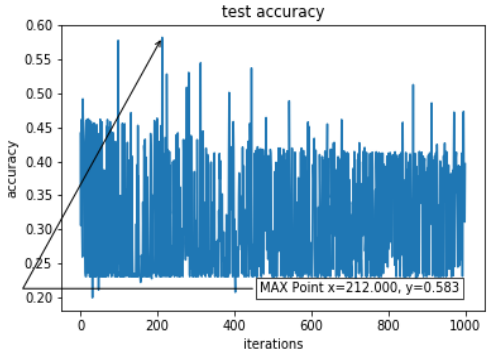


(a) Test Accuracy Against the Number of Epochs

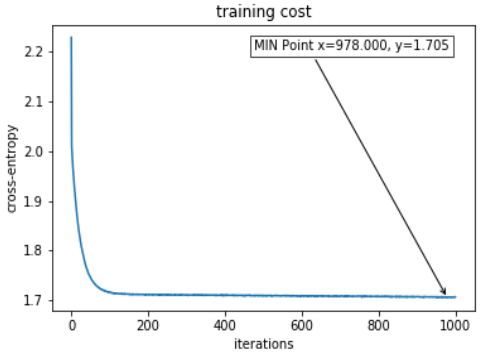


(b) Training Cost Against the Number of Epochs

Figure 17: Result of Model with Decay = 0

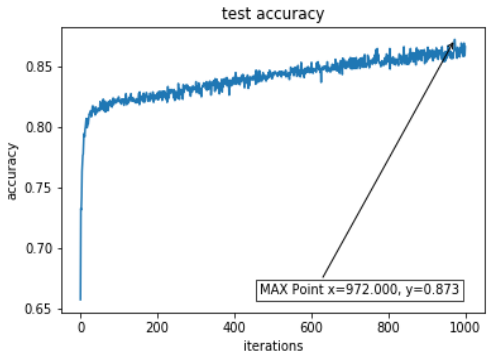


(a) Test Accuracy Against the Number of Epochs

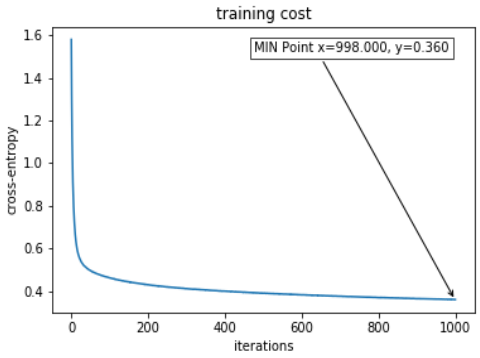


(b) Training Cost Against the Number of Epochs

Figure 18: Result of Model with Decay = 0.001

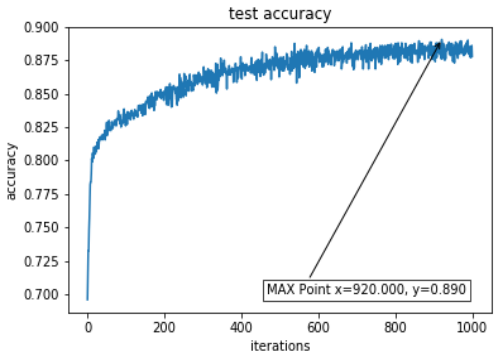


(a) Test Accuracy Against the Number of Epochs

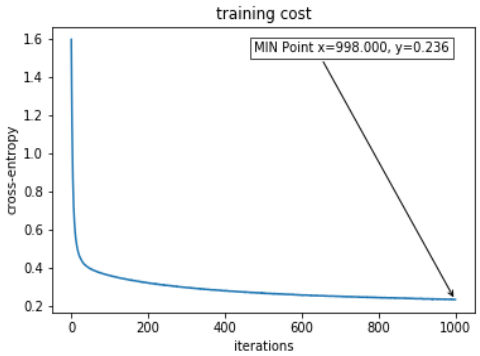


(b) Training Cost Against the Number of Epochs

Figure 19: Result of Model with Decay = 1e-06

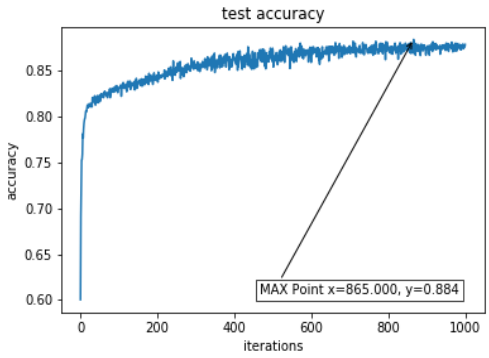


(a) Test Accuracy Against the Number of Epochs

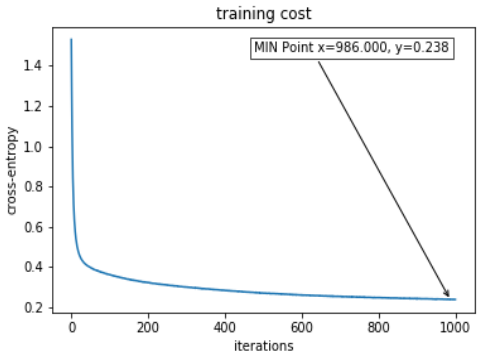


(b) Training Cost Against the Number of Epochs

Figure 20: Result of Model with Decay = 1e-09



(a) Test Accuracy Against the Number of Epochs



(b) Training Cost Against the Number of Epochs

Figure 21: Result of Model with Decay = 1e-12

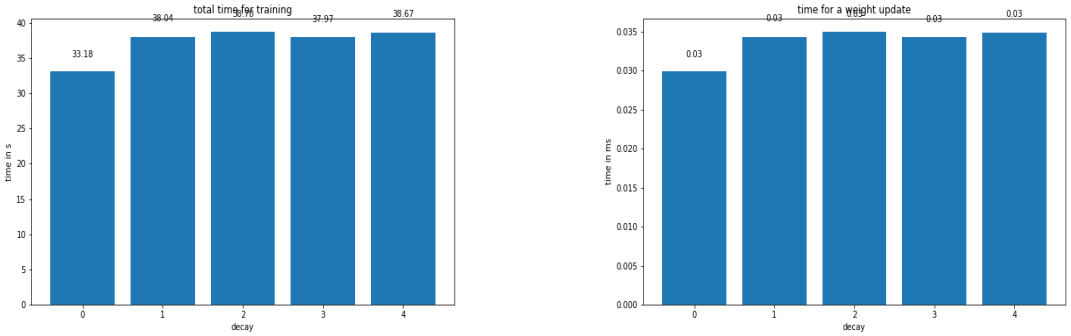
1.5.2 Time Taken

The following section shows the graph of the time taken to update parameters of the network for each sample for different values of decay parameter (Figure 22b) (The time taken for training

one batch of data samples), and the graph of total time taken to train the model for different values of decay parameter (Figure 22a).

As shown in the 22, when decay = 0 (bar 0), it takes obvious shorter time to train than other values of decay parameter. It is because that when decay = 0 there is no need to compute the decay, the computation cost is 0 for computing the decay. Hence, it takes significant shorter time than other values of decay parameter.

The time taken to train the model are about the same for decay = [1e-3, 1e-6, 1e-9, 1e-12] (bar 1,2,3,4), as the value of decay does not affect the computational cost.



(a) Total time to train the model for different different values of decay parameter

(b) Time to update parameters of the network for each sample for different values of decay parameter

Figure 22: Time Comparison of Models with Different different values of decay parameter, 0: 0 decay, 1: 1e-3 decay, 2: 1e-6 decay, 3: 1e-9 decay, 4: 1e-12 decay

1.5.3 Rational for Selecting Optimal Decay Parameter

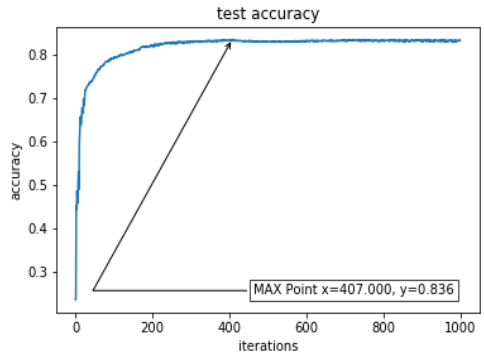
According to Figure 22, the longest training time is around 1 minute which is acceptable. Hence, we have decided that we will select the optimal decay parameter by only looking at accuracy,ignoring the time taken to train the model. We have decided to choose decay = 1e-9 as the optimal value of decay parameter, as it could produce the highest accuracy 0.890.

1.6 4-Layer Network

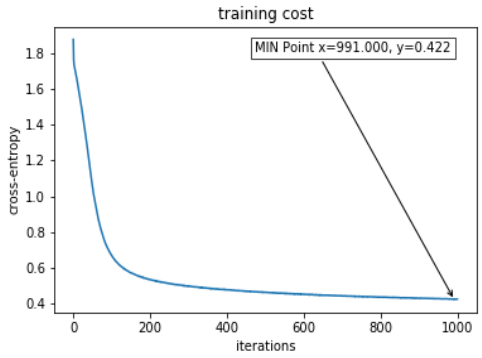
A 4-layer feed-forward neural network which is consisting of 2 hidden-layers. Each hidden-layer consists of 10 neurons with logistic activation functions, batch size of 32 and decay parameter 10-6. The 4 layer network is implemented by Theano Library as the code listing section 3 shown below. We could get a initial result of 4-layer model. As shown in the Figure 23, the highest accuracy 0.836 is achieved at 407th epoch.

```
def train_4layers(trainX, trainY, testX, testY, decay, learning_rate, epochs, batch_size, num_neurons):
    """
    Struct the model with 2 hidden layers
    and train the model
    """
    # theano expressions
    X = T.matrix() #features
    Y = T.matrix() #output
    w1, b1 = init_weights(36, num_neurons), init_bias(num_neurons)
    # weights and biases from input to hidden layer
    w2, b2 = init_weights(num_neurons, num_neurons), init_bias(num_neurons)
    w3, b3 = init_weights(num_neurons, 6, logistic=False), init_bias(6)
    # weights and biases from hidden to output layer
    # activation
    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)))) + decay*(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)
    # train and test
    n = len(trainX)
    test_accuracy = []
    train_cost = []
    batch_time_used = []
    for i in range(epochs):
        if i % 100 == 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)):
            start_time = time.time()
            cost += train(trainX[start:end], trainY[start:end])
            batch_time_used.append(time.time()*1000-start_time*1000)

        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))
    return train_cost, test_accuracy, np.mean(batch_time_used), np.sum(batch_time_used)
```

Listing 3: 3-layer feedforward neural network



(a) Test Accuracy Against the Number of Epochs



(b) Training Cost Against the Number of Epochs

Figure 23: Result of 4 Layer Model

1.7 Conclusions

After selecting the optimal batch size 4, number of hidden neuron 15, decay parameter 1e-9, the highest accuracy we could produce from 3 layer network is 0.890. We could still further improve the accuracy of 3 layer model by selecting the optimal learning rate, selecting the ideal activation function and also tuning other parameters.

The accuracy of the 4 layer model is 0.836 which is much lower than the highest accuracy produced from 3 layer network. However, all the parameters of the 4 layer model are not tuned yet and we believed that after parameter tuning, the 4 layer can greatly improve from the current result 0.836.

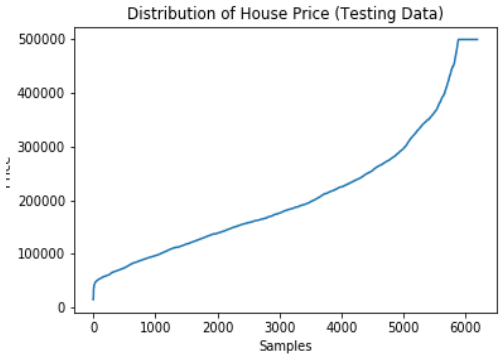
2 Part B: Approximation Problem

2.1 Problem Introduction

This part of the project is an approximation problem of housing prices in the California Housing dataset and the use of multi-layer feed-forward networks for solving the prediction problem. In this part, several tools were used: Theano for buiding and training the network, scikit-learn for doing k-fold validation and matplotlib for plotting the graphs. All performance data is collected by running the code on an Intel Quad Core i7-4900MQ CPU.

The dataset contains 20640 samples and it has been divided into 14448 training data samples and 6192 testing data samples. For each sample, there are 8 attributes and the corresponding house price (Y value). As shown in the Figure 24, after dividing, the distribution of house price in training data and testing data is approximately same, which is an ideal situation. It means that the training dataset is a good representation of testing dataset.

(a) Distribution of House Price (Training Data)



(b) Distribution of House Price (Testing Data)

Figure 24: Distribution of House Price

2.2 Problem B2 - Pre-processing

Before conducting the training of the model and tunning of all parameters. We did pre-processing to both training data and testing data and implement the network using Theano.

2.2.1 Data Partitioning

The data is divided into test and train set at a ratio of 0.3: 0.7.

```
#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:]
```

Listing 4: Seperate Training and Test data

2.2.2 Scaling & Normalization

We used Min-Max Normalization to scale all the attributes in the datasets. The minimum and maximum values for each attributes are the minimum and maximum values of attributes in the training dataset. Hence, the testing dataset is normalized by the parameters as the training dataset. After normalization, all the values in the attributes are basically evenly ranged from 0 to 1. For example, the distribution of the first attribute in training data after normalization is shown as Figure 25.

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

# scale and normalize data
trainX_max, trainX_min =  np.max(trainX, axis=0), np.min(trainX, axis=0)
testX_max, testX_min =  np.max(testX, axis=0), np.min(testX, axis=0)

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

Listing 5: Scaling of Training and Test data

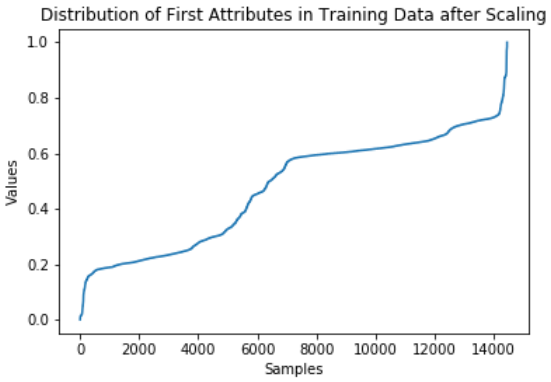


Figure 25: Distribution of First Attributes in Training Data after Scaling

2.2.3 Implementation of the Neural Network

For three layers neural network, a function for training is implemented and a 5-fold validation is also conducted. Training and test data, and all other training parameters can be passed in to the function.

```
def train_network_validation(trainX, trainY, testX, testY,
    learning_rate, epochs, batch_size, no_hidden1, plot_filename):
    floatX = theano.config.floatX
    no_features = trainX.shape[1]
    x = T.matrix('x') # data sample
    d = T.matrix('d') # desired output
    no_samples = T.scalar('no_samples')
    # 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)
    # learning rate
    alpha = theano.shared(learning_rate, 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
            )
    test = theano.function(
        inputs = [x, d],
        outputs = [y, cost, accuracy],
        allow_input_downcast=True
        )
```

Listing 6: Define the network

```
    # Training
    kf = KFold(n_splits=5)
    val_itr = 0
    t = time.time()
    train_costs = []
    test_costs = []
    val_costs = []
    for train_index, val_index in kf.split(trainX):
        # Some Initialization
        # ...
        val_itr += 1
        print("k fold validation: " + str(val_itr))
        print("TRAIN: "+str(train_index) + " VALID: "+str(val_index))
        val_set_X = trainX[val_index]
        val_set_Y = trainY[val_index]
        train_set_X = trainX[train_index]
        train_set_Y = trainY[train_index]
```

Listing 7: 5-fold Validation

```
        for iter in range(epochs):
            trainX, trainY = shuffle_data(trainX, trainY)
            train_cost[iter] = train(trainX, np.transpose(trainY))
            val_pred, val_cost[iter], val_accuracy[iter] = test(val_set_X, np.transpose(val_set_Y))
            pred, test_cost[iter], test_accuracy[iter] = test(testX, np.transpose(testY))
            if test_cost[iter] < min_error:
                best_iter = iter
                min_error = test_cost[iter]
                best_w_o = w_o.get_value()
                best_w_h1 = w_h1.get_value()
                best_b_o = b_o.get_value()
                best_b_h1 = b_h1.get_value()
        #set weights and biases to values at which performance was best
        w_o.set_value(best_w_o)
        b_o.set_value(best_b_o)
        w_h1.set_value(best_w_h1)
        b_h1.set_value(best_b_h1)
        test_costs.append(test_cost)
        train_costs.append(train_cost)
        val_costs.append(val_cost)
        best_pred, best_cost, best_accuracy = test(testX, np.transpose(testY))
        print('Minimum error: %.1f, Best accuracy %.1f, Number of Iterations: %d'
                %(best_cost, best_accuracy, best_iter))
    plot_train_val_error(plot_filename, np.mean(train_costs, axis=0), np.mean(val_costs, axis=0), epochs)
    plot_test_error(plot_filename, np.mean(test_costs, axis=0), epochs)
```

Listing 8: Training the network with gradient decent

2.3 Problem B2 - Question 1

This question requires an initial training on the basic 3-layer neural network. In this question, I used the mini-batch gradient descent with batch size 32 and set the learning rate = 0.0001 and number of neurons in hidden layer to 30. Then I trained the network up to 1000 epochs.

2.3.1 Implementation

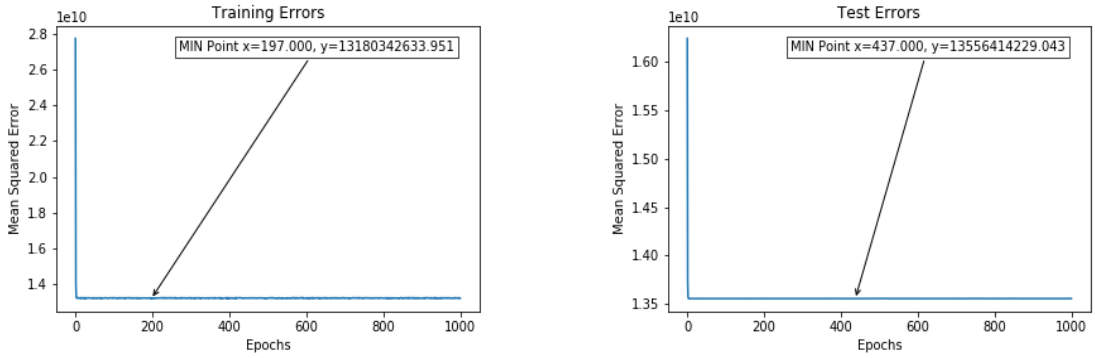
The overall implementation is similar with the code listed in Section 2.2.3. The detailed implementation of mini-batch is listed below.

```
cost = 0.0
n = len(trainX)
for start, end in zip(range(0, n, batch_size), range(batch_size, n, batch_size)):
    cost += train(trainX[start:end], np.transpose(trainY[start:end]))
#   train_cost[iter] = train(trainX, np.transpose(trainY))
train_cost[iter] = cost/(n // batch_size)
```

Listing 9: Implementation of mini-batch gradient decent training

2.3.2 Result

Figure 26 shows the training error against number of epochs and the final test errors of prediction by the network. From the graphs we can see that the network achieves a relatively low error within the first few epochs. Then the error actually goes up and down for the remaining epochs. We assume that it is probably because that the network is already converged and we have ingested too much data for training.



(a) Training Error Against the Number of Epochs (b) Test Error Against the Number of Epochs

Figure 26: Initial Result of 3 Layer Model for Question B

2.4 Problem B2 - Question 2

This question aims to choose the best learning rate for the network by selecting in the range 10^{-3}, $0.5 * 10^{-3}$, 10^{-4}, $0.5 * 10^{-4}$, 10^{-5}. In this question, 5-fold validation is performed and normal gradient decent is used.(No batch needed)

2.4.1 Implementation

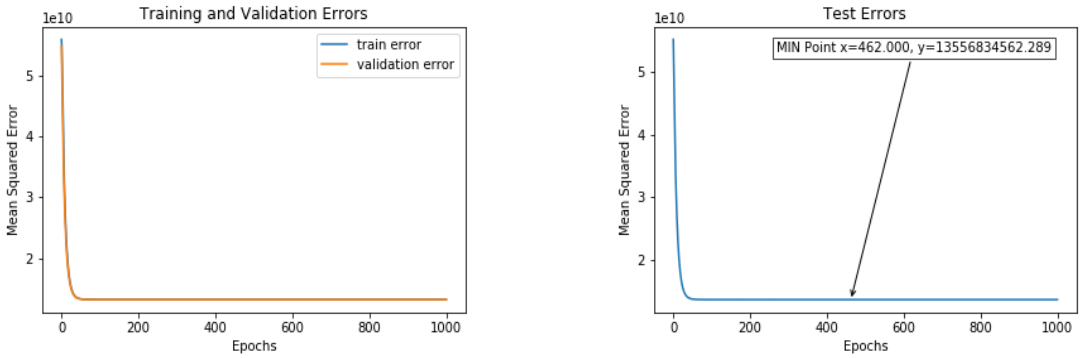
The code shows how we ran multiple training with different learning rates.

```
# Question 2

epochs = 1000
batch_size = 32 # NOT USED
no_hidden1 = 30 #num of neurons in hidden layer 1
learning_rates = [1e-3, 0.5e-3, 1e-4, 0.5e-4, 1e-5]

for learning_rate in learning_rates:
    print("Running Learning Rate: "+ str(learning_rate))
    train_network_validation(trainX, trainY, testX, testY,
        learning_rate, epochs, batch_size, no_hidden1,
        './theano_graph/2/learning_rate_'+str(learning_rate))
```

Listing 10: Implementation of Running different learning rates

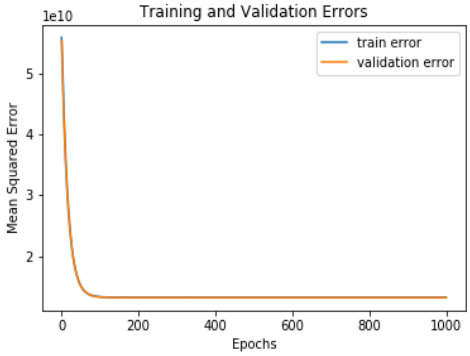
2.4.2 Result

Figure 27a, 28, 29 shows training errors and validation errors against number of epochs. And Figure 27b shows the test errors against number of epochs for the optimum learning rate, which is 0.001 in our case. Generally, we discover that with a larger learning rate, the network will converge faster.

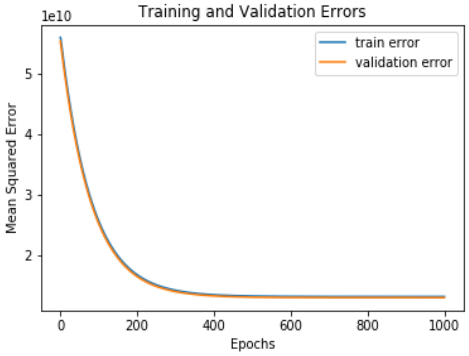


(a) Training Error Against the Number of Epochs (b) Test Error Against the Number of Epochs

Figure 27: Result of 3 Layer Model with 0.001 Learning Rate for Question B

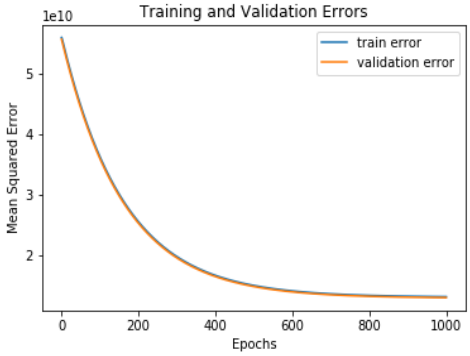


(a) Training Error Against the Number of Epochs with 0.0005 Learning Rate

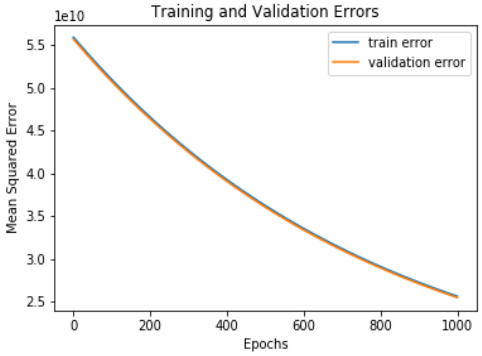


(b) Training Error Against the Number of Epochs with 0.0001 Learning Rate

Figure 28: Result of 3 Layer Model with 0.0005 and 0.0001 Learning Rate for Question B



(a) Training Error Against the Number of Epochs with 5e-05 Learning Rate



(b) Training Error Against the Number of Epochs with 1e-05 Learning Rate

Figure 29: Result of 3 Layer Model with 5e-05 and 1e-05 Learning Rate for Question B

2.4.3 Rationale

We choose 0.001 as our best learning rate. The reason is that the network converges within 1000 epochs and it is the fastest for network to converge. The figures above shows clearly the speed of convergence of networks with different learning rates. For learning rate $= 0.5 * 10^{-4}$, 10^{-5}, the networks did not converge before epoch 1000. And among learning rate $= 10^{-3}$, $0.5 * 10^{-3}$, 10^{-4}, the speed of convergence is obviously different. The network with learning rate 0.001 took much less time to converge and find a local minimal.

2.5 Problem B2 - Question 3

This question aims to choose the best neuron numbers in hidden layer for the network by selecting in the range 20, 30, 40, 50, 60. In this question, 5-fold validation is performed and normal gradient decent is used.(No batch needed)

2.5.1 Implementation

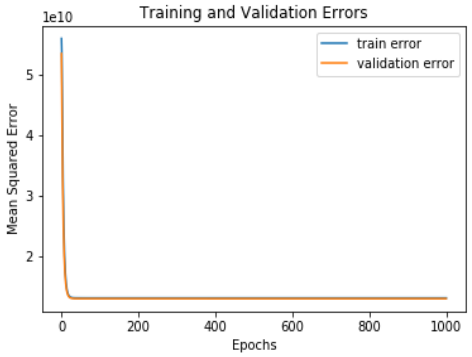
The code shows how we ran multiple training with numbers of neurons in the hidden layer.

```
# Question 3

epochs = 1000
batch_size = 32
nos_hidden1 = [20, 30, 40, 50 ,60] #num of neurons in hidden layer 1
learning_rate = 1e-3

for no_hidden1 in nos_hidden1:
    print("Running Num of Hidden 1: "+ str(no_hidden1))
    train_network_validation(trainX, trainY, testX, testY, learning_rate,
    epochs, batch_size, no_hidden1, './theano_graph/3/no_hidden1_'+str(no_hidden1))
```

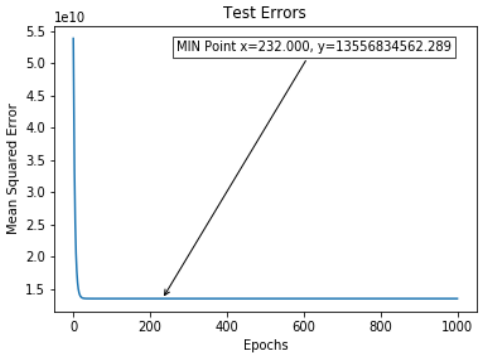
Listing 11: Implementation of Running different numbers of neurons

2.5.2 Result

Figure 30a, 31, 32 shows training errors and validation errors against number of epochs. And Figure 30b shows the test errors against number of epochs for the optimal number of neurons in hidden layer, which is 60 in our case. Generally, we discover that more number of neurons in the hidden layer, the network will converge faster.

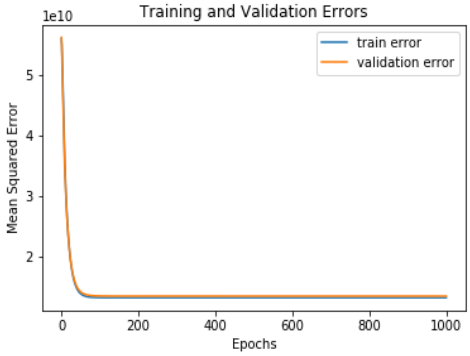


(a) Training Error Against the Number of Epochs

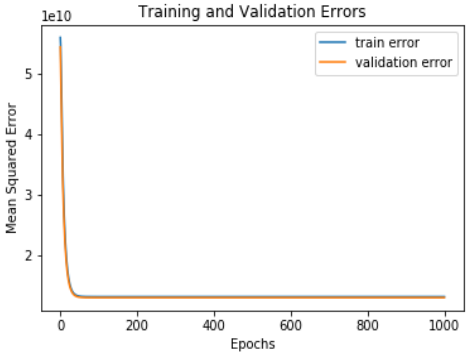


(b) Test Error Against the Number of Epochs

Figure 30: Result of 3 Layer Model with 60 hidden neurons for Question B

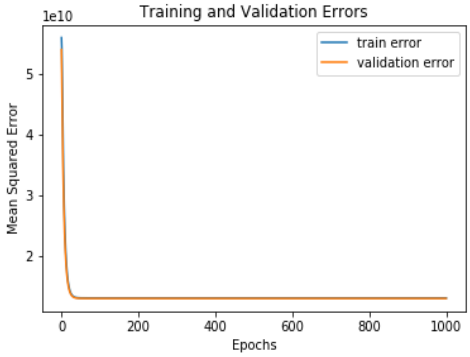


(a) Training Error Against the Number of Epochs with 20 Hidden Neurons

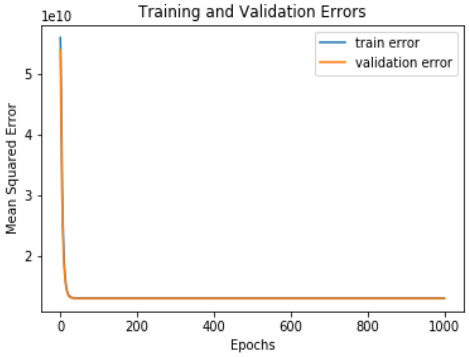


(b) Training Error Against the Number of Epochs with 30 Hidden Neurons

Figure 31: Result of 3 Layer Model with 20 and 30 Hidden Neurons for Question B



(a) Training Error Against the Number of Epochs with 40 Hidden Neurons



(b) Training Error Against the Number of Epochs with 50 Hidden Neurons

Figure 32: Result of 3 Layer Model with 40 and 50 Hidden Neurons for Question B

2.5.3 Rationale

We choose 60 as our best number of neuron in hidden layer. Actually, the graph looks similar among different numbers of neurons chosen. However, there is still a difference in convergence speed. The network with 60 neurons in the hidden layer obviously took less time to converge than others.

2.6 Problem B2 - Question 4

This question requires to implement a 4-layer and 5-layer network and compare them with the original 3-layer network in order to see how the depth of the network will affect the whole neural network.

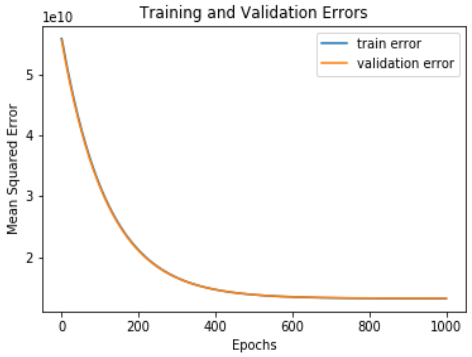
2.6.1 Implementation

The parameters are set according to requirements. We have learning rate = 10^{-4}, number of neurons in first hidden layer = 60, number of neurons in second and third hidden layer = 20.

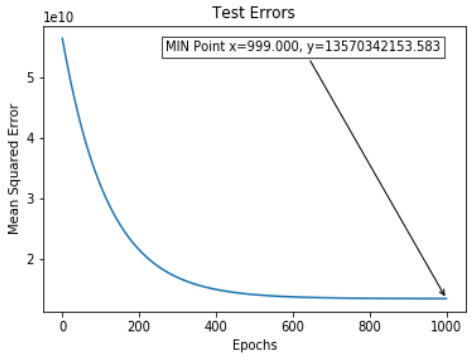
An implementation of 5-layer network is listed below for example (Listing 12).

2.6.2 Result

Figure 33 shows the training error and test error of the 4-layer network. Figure 34 shows the training error and test error of the 5-layer network.

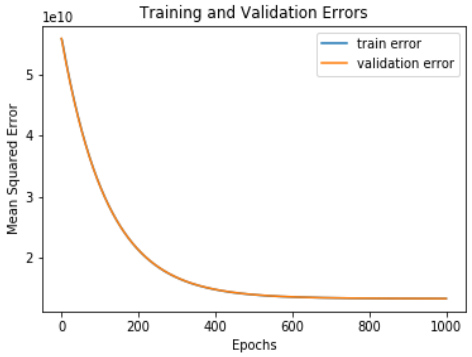


(a) Training Error Against the Number of Epochs

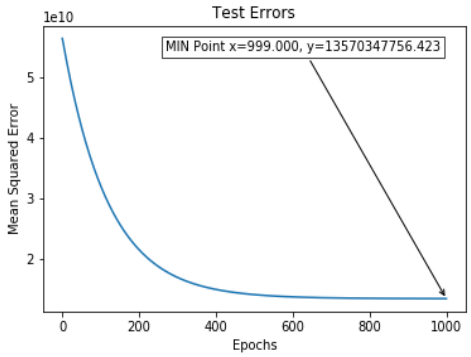


(b) Test Error Against the Number of Epochs

Figure 33: Result of 4 Layer Model for Question B



(a) Training Error Against the Number of Epochs



(b) Test Error Against the Number of Epochs

Figure 34: Result of 5 Layer Model for Question B

2.6.3 Comparison

Compare Figure 33 and 34 with Figure 26, The 4-layer and 5-layer networks did not converge completely so that the final test error is not as good as the 3-layer network. For a deeper neural network, we may need to train it for more epochs and ingest more data for it in order to achieve a better performance. From this, we learn that we not only need to choose the right parameters for the network, but also need to select the network size and depth wisely so that best performance can be gained.

```
floatX = theano.config.floatX

no_features = trainX.shape[1]
x = T.matrix('x') # data sample
d = T.matrix('d') # desired output
no_samples = T.scalar('no_samples')

# initialize weights and biases for hidden layer(s) and output layer
w_o = theano.shared(np.random.randn(no_hidden3)*.01, floatX )
b_o = theano.shared(np.random.randn()*.01, floatX)
w_h3 = theano.shared(np.random.randn(no_hidden2, no_hidden3)*.01, floatX )
b_h3 = theano.shared(np.random.randn(no_hidden3)*0.01, floatX)
w_h2 = theano.shared(np.random.randn(no_hidden1, no_hidden2)*.01, floatX )
b_h2 = theano.shared(np.random.randn(no_hidden2)*0.01, 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)

# learning rate
alpha = theano.shared(learning_rate, floatX)

#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)
h3_out = T.nnet.sigmoid(T.dot(h2_out, w_h3) + b_h3)
y = T.dot(h3_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_h1, db_h1, dw_h2, db_h2, dw_h3, db_h3 =
            T.grad(cost, [w_o, b_o, w_h1, b_h1, w_h2, b_h2, w_h3, b_h3])

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],
               [w_h3, w_h3 - alpha*dw_h3],
               [b_h3, b_h3 - alpha*db_h3]],
        allow_input_downcast=True
        )

test = theano.function(
    inputs = [x, d],
    outputs = [y, cost, accuracy],
    allow_input_downcast=True
        )
```


Listing 12: Define 5-layer network

References

[1] Nitish Shirish Keskar et al. "On large-batch training for deep learning: Generalization gap and sharp minima". In: *arXiv preprint arXiv:1609.04836* (2016).

[2] UCI Machine Learning Repository. *Statlog (Landsat Satellite) Data Set*. URL: `https://archive.ics.uci.edu/ml/datasets/Statlog%20(Landsat%20Satellite)`.