NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042 – Neural Networks Project 1 Report

Prepared by:

Jordan Lam Yu Cheng JLAM014@e.ntu.edu.sg

Nathanael Raj nathanae001@e.ntu.edu.sg

Note: Reports for Part A and Part B have been organised separately.

Neural Networks Project 1 Part A

Introduction

In this project, we aim to classify the Landsat satellite dataset. The dataset consists of multi-spectral values of pixels in 3x3 neighbourhoods in a satellite image and the classification associated with the central pixel in each neighbourhood. Each data point has 36 features, and a corresponding true class. We performed Machine Learning using three and four layer neural networks and achieved an overall test accuracy of 87% after optimising several parameters.

Methods

We implemented the neural network in tensorflow with one hidden layer initially and tuned several parameters using a test set to improve accuracy. Subsequently, we added an additional hidden layer to compare the performance between a one and two hidden layer neural network. Details on how the neural network was implemented are contained in the next section.

Experiment

```
import math
import tensorflow as tf
import numpy as np
import pylab as plt


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

NUM_FEATURES = 36
NUM_CLASSES = 6
NUM_HIDDEN = 10
learning_rate = 0.01
epochs = 1000
batch_size = 32
num_neurons = 10
seed = 10
beta = 10 **-6
np.random.seed(seed)
```

The parameters for the 3 layer neural network is first initialised, as well as a scale function used to scale the input according to the maximum and minimum values

```
#read train data
train_input = np.loadtxt('sat_train.txt',delimiter=' ')
trainX, train_Y = train_input[:,:36], train_input[:,-1].astype(int)
trainXmin = np.min(trainX, axis=0)
trainXmax = np.max(trainX, axis=0)
trainX = scale(trainX, trainXmin, trainXmax)
train_Y[train_Y == 7] = 6

trainY = np.zeros((train_Y.shape[0], NUM_CLASSES))
trainY[np.arange(train_Y.shape[0]), train_Y-1] = 1 #one hot matrix

#read test data
test_input = np.loadtxt('sat_test.txt',delimiter=' ')
testX, test_Y = test_input[:,:36], test_input[:,-1].astype(int)
testX = scale(testX, trainXmin, trainXmax)
test_Y[test_Y == 7] = 6

testY = np.zeros((test_Y.shape[0], NUM_CLASSES))
testY[np.arange(test_Y.shape[0]), test_Y-1] = 1 #one hot matrix
```

Train and test data is read from the data. The labels of the class are converted into a one hot matrix so that it can be fed into the cross entropy loss function

```
x = tf.placeholder(tf.float32, [None, NUM_FEATURES])
#x is ? by 32
y_ = tf.placeholder(tf.float32, [None, NUM_CLASSES])

# Build the graph for the deep net
#w1 and b1 is the weight/bias to the hidden layer
w1 = tf.Variable(tf.truncated_normal([NUM_FEATURES, NUM_HIDDEN], stddev=1.0/math.sqrt(float(NUM_FEATURES))), name='weightToHidden')
b1  = tf.Variable(tf.zeros([NUM_HIDDEN]), name='biasestohidden')

#w2 and b2 are the weight/bias from hidden to output layer
w2 = tf.Variable(tf.truncated_normal([NUM_HIDDEN, NUM_CLASSES], stddev=1.0/math.sqrt(float(NUM_FEATURES))), name='weightToOutput')
b2  = tf.Variable(tf.zeros([NUM_CLASSES]), name='biasestooutput')

hidden_logits = tf.matmul(x, w1) + b1
hidden_activated = tf.sigmoid(hidden_logits)

output_logits = tf.matmul(hidden_activated, w2) + b2

cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(labels=y_, logits=output_logits)

loss = tf.reduce_mean(cross_entropy) + tf.square(tf.norm(w1, ord = 'fro', axis = (-2,-1)))*beta + tf.square(tf.norm(w2, ord = 'fro', axis =(-2,-1)))*beta

# Create the gradient descent optimizer with the given learning rate.
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train_op = optimizer.minimize(loss)

correct_prediction = tf.cast(tf.equal(tf.argmax(output_logits, 1), tf.argmax(y_, 1)), tf.float32)
accuracy = tf.reduce_mean(correct_prediction)
```

The neural network consist of 3 layers. The input goes into a 10 neuron hidden layer, where a sigmoid activation function is applied at each neuron. The output of this hidden layer is then sent to the final output layer, which consist of 6 neurons. A softmax function is applied at this layer and the

output of each neuron will correspond to the probability that the data point belongs to the class its neuron is representing. The neuron network is trained using a cross entropy loss function applied on the output. L2 regularisation penalty term is added to the loss function with beta parameter of 10^{-6}.

```
idx = np.arange(trainX.shape[0])
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    train_acc = []
    test_accs = []
    repetition_in_one_epoch = int(NUM_INPUT / batch_size)
    for i in range(epochs):
        np.random.shuffle(idx)
        trainX, trainY = trainX[idx], trainY[idx]
        start = -1 * batch_size
        end = 0
        for k in range(repetition_in_one_epoch):
            start += batch_size
            end += batch_size
            if end > NUM_INPUT:
                end = NUM_INPUT
            train_op.run(feed_dict={x: trainX[start:end], y_: trainY[start:end]})

        train_acc.append(accuracy.eval(feed_dict={x: trainX, y_: trainY}))
        test_accs.append(accuracy.eval(feed_dict={x: testX, y_: testY}))
        if i % 100 == 0:
            print('iter %d: accuracy %g'%(i, train_acc[i]))
            print('Test acc for iter %d: accuracy %g'%(i, test_accs[i]))
    print("Final test accuracy :", test_accs[-1])


# plot learning curves0.83
plt.figure("Train accuracy Vs Epochs")
plt.plot(range(epochs), train_acc)
plt.xlabel(str(epochs) + ' Epochs')
plt.ylabel('Train accuracy')

plt.figure("Test accuracy Vs Epochs")
plt.plot(range(epochs), test_accs)
plt.xlabel(str(epochs) + ' Epochs')
plt.ylabel('Test accuracy')
plt.show()
```

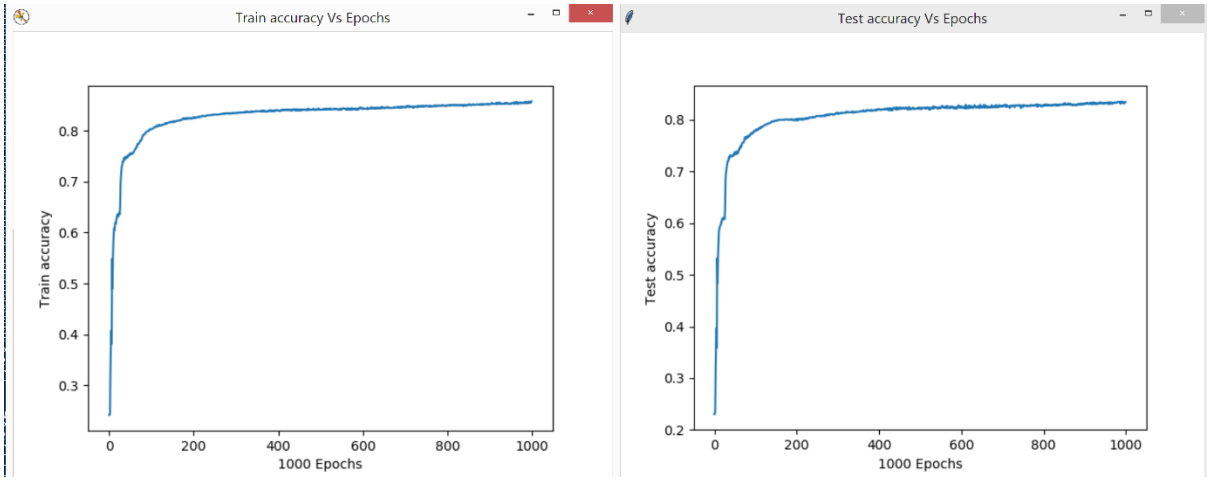
Finally the model is trained using the training data. One epoch consist of all the iterations required for all data points in the training set to be visited. The train data is shuffled before each epoch begins to avoid overfitting of the model to a particular sequence of training points. This three layer network was trained for 1000 epochs. The four layer network had the same setup as this three layer one.

Conclusion and Results

In conclusion, we found that the best performance for this problem was achieved with the following parameters: one hidden layer, batch size of 4, 15 neurons in the hidden layer and decay parameter of 10^{-6}. This yielded a test accuracy of 88.8%. The results of the questions are contained below

Question 1

The plots of the training and test accuracy are as below:

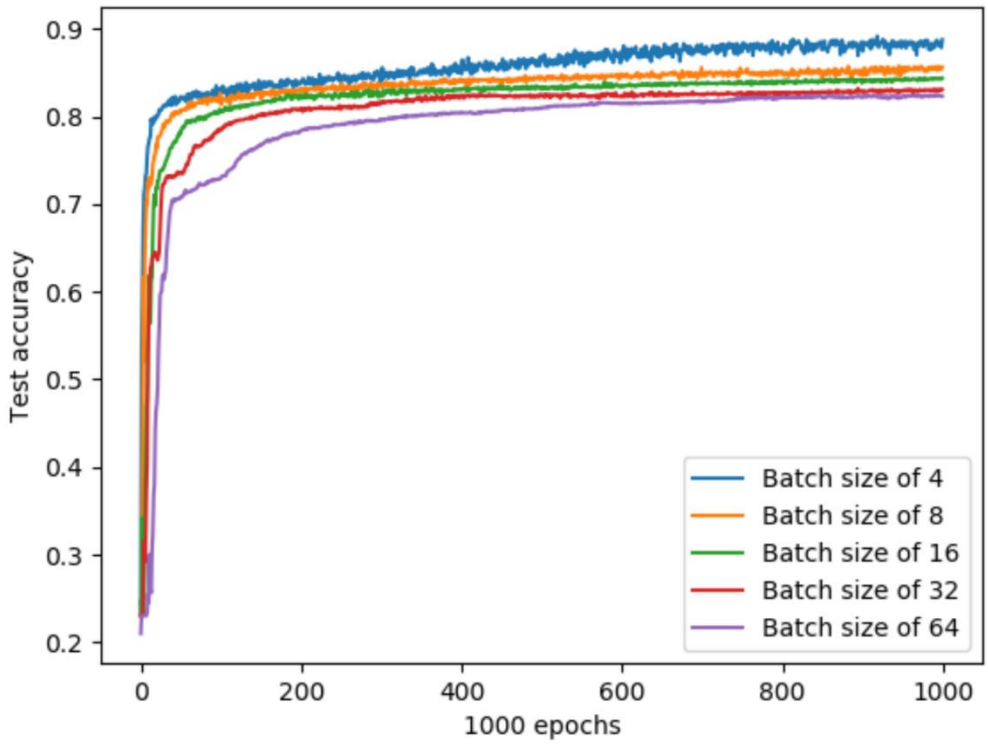


The three layer network attained a test accuracy of 83.4%.

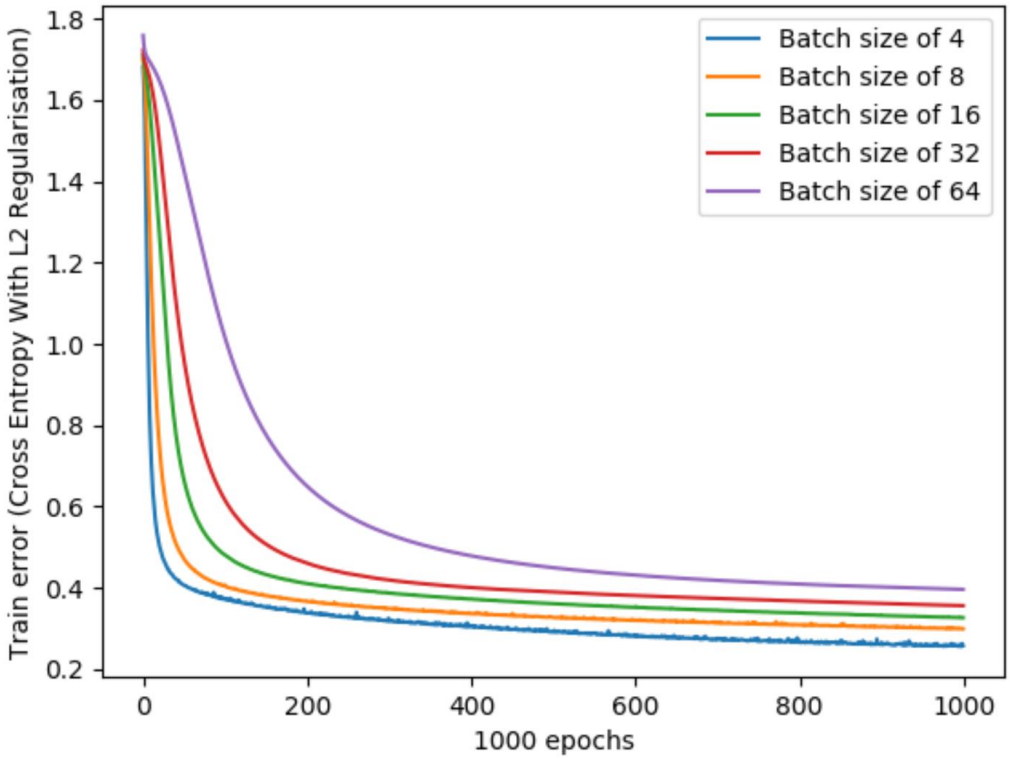
Question 2

2a

Test Accuracy Against Epochs for each Batch Size

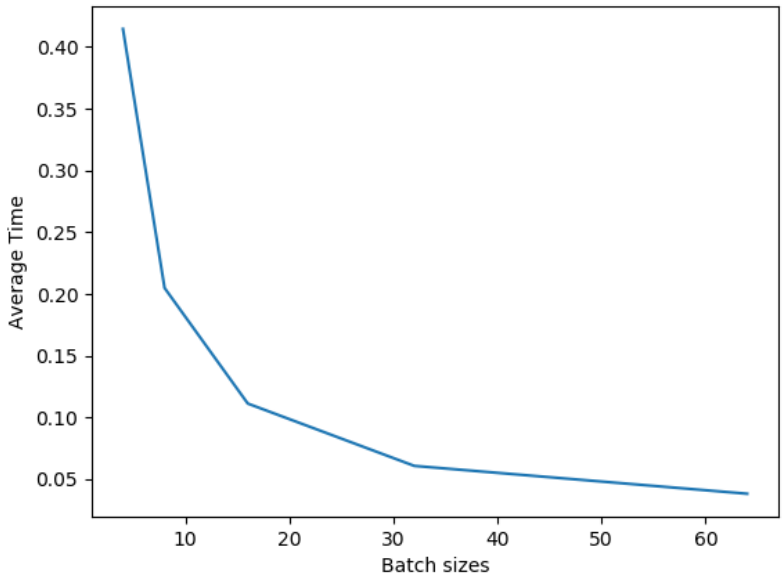


Train Error Against Epochs for each Batch Size



2b

Average Time Taken (seconds) for one Epoch Against Batch Size



Average time for one epoch was derived by taking the time taken to train the network for 1000 epochs and dividing it by 1000 epochs. Values are in seconds

2c

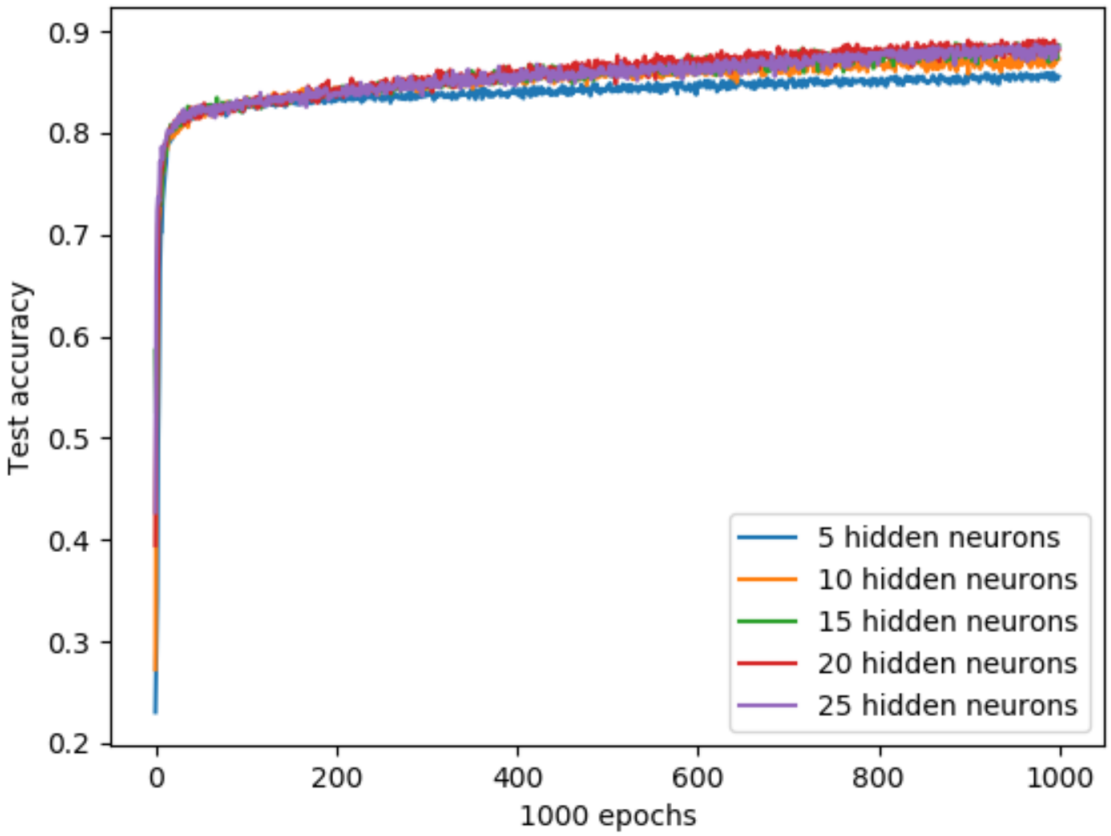
Chosen batch size: 4

While there is a trade off between accuracy and time as batch size increases, the time taken to train for an epoch with batch size 4 is still tolerable at 0.43 seconds and it achieved the highest accuracy at 88.75%. At 1000 epochs, training with batch size 4 takes around 7 minutes to train which is a relatively small price to pay for the improved accuracy in the model.

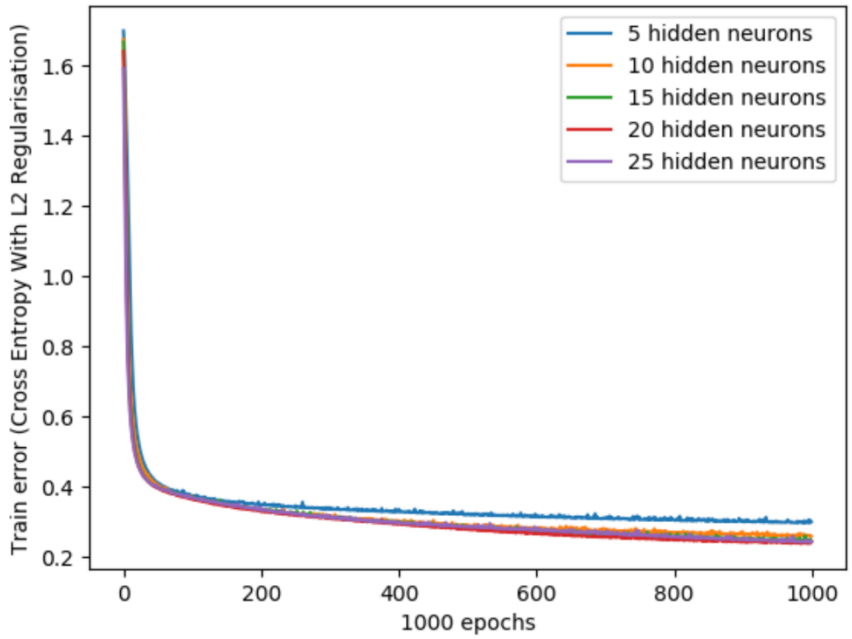
Question 3

3a

Test Accuracy against Epochs plotted for each number of Hidden Layer Neuron

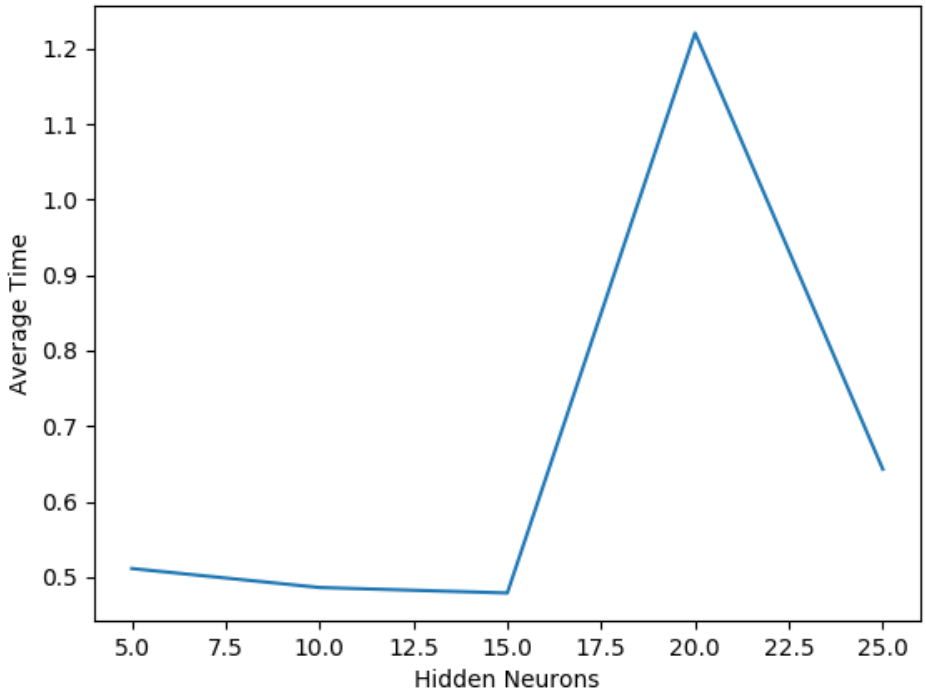


Training error against Epochs plotted for each number of Hidden Layer Neuron



3b

Average Time taken for one Epoch against Number of Hidden Layer Neurons



Average time here is computed the same way as in question 2.

3c

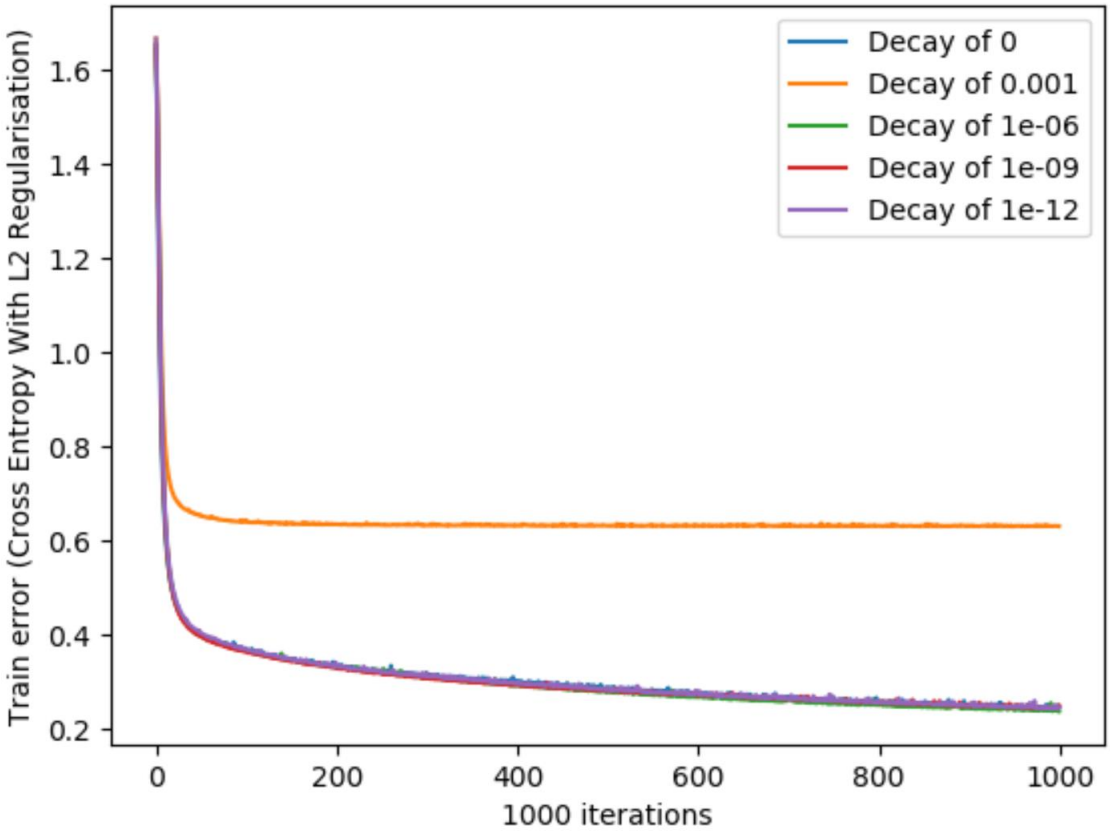
Chosen number of hidden neurons: 15

The test accuracy of the network was highest at 88.6% for 15 neurons. Thus, 15 hidden neurons were chosen since it also took the least average time.

Question 4

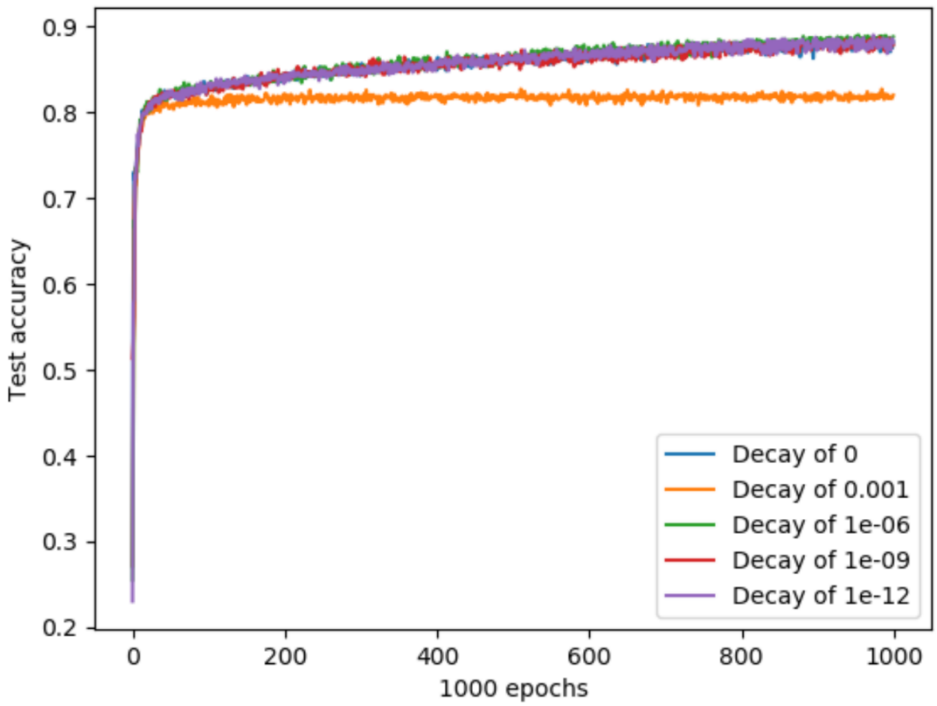
4a

Training error against Epochs plotted for each Beta Value



4b

Test Accuracy against Epochs plotted for each Beta Value

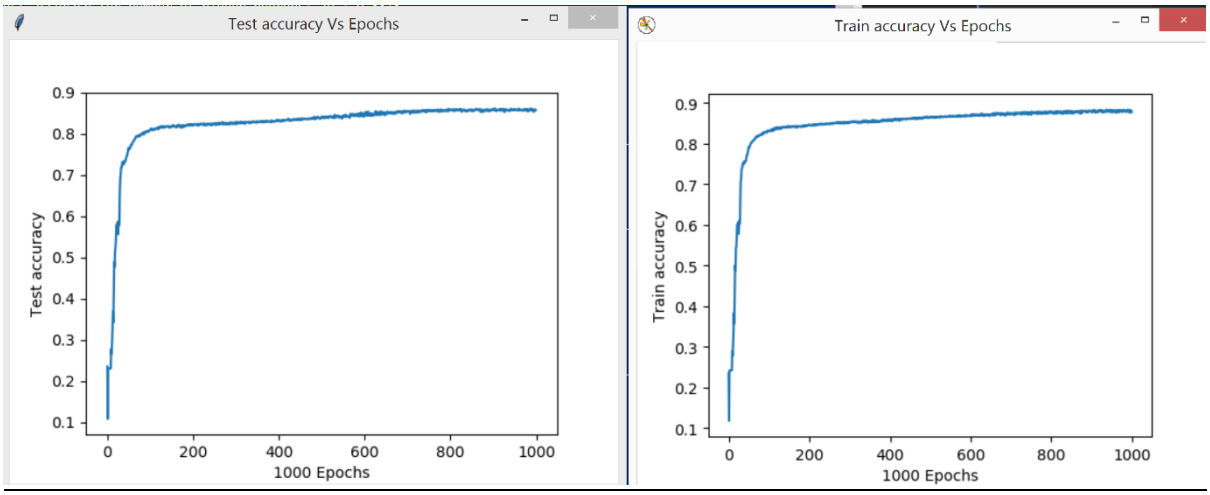


4c

Similar trends were observed for both the training error plot and test accuracy plot for varying decay parameters. As such, the only factor which differentiated each decay parameter was accuracy. Thus, decay parameter of beta = 10^{-6} was chosen since it yielded the highest test accuracy at 88.8%.

Question 5

5a



5b

The 4 layer neural network gave a slightly higher final test accuracy of 85.5% compared to the 83.4% test accuracy of the 3 layer neural network.

Both 3 and 4 layer neural networks also had similar shaped plots, training and testing plateau after around 300 epochs. In both networks the shapes of the train and test accuracy plots were also similarly shaped.

The similar performance of both networks indicate that one hidden layer was enough to introduced non-linearity into the model for it to learn the parameters of this problem. Hence, the additional non-linearity introduced by the second hidden layer could not be utilised and did not provide much better results.

Part B: Regression Problem

(i) introduction

Using features like median income, housing median age, total rooms, total bedrooms, population, households, latitude, and longitude we are trying to predict housing prices.

This part will focus on selecting the best hyperparameters, mainly, number of neurons and learning rate.

(ii) methods

The dataset is large with over 20k datapoint. Using a 30/70 split, the test set will have size 6192 and the training/validation set will have size 14448. To do this, we utilised a GTX960M GPU and the tensorflow-gpu library. This achieved roughly 10x increase in computation speed.

One thing to note is that due to the large values of housing prices ~500K, calculations using GPU resulted in overflow and NaNs and to solve this, we scaled the prices down by a factor of 1000.

Normalization

We normalized by zeroing the inputs by removing the mean and standard deviation calculated from the training set for both training and test set as shown below:

```
testX = (testX- np.mean(trainX, axis=0))/ np.std(trainX, axis=0)

trainX = (trainX- np.mean(trainX, axis=0))/ np.std(trainX, axis=0)
```

Shuffling

For each epoch, shuffling was done before the mini batch occurred to remove possible biases in the initial shuffle.

```
for epoch in range(epochs):

        idx = np.arange(trainX.shape[0])

        np.random.shuffle(idx)

        train_x, train_y = train_x[idx], train_y[idx]
```

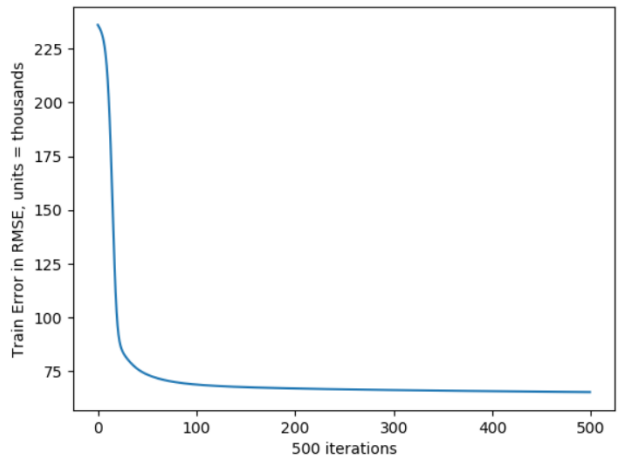
Plotting

The Root Mean Squared Error was plotted where required as this is more human readable and relates more directly to actual housing prices.

```
err = np.sqrt(error.eval(feed_dict={x: trainX, y_: trainY}))
```

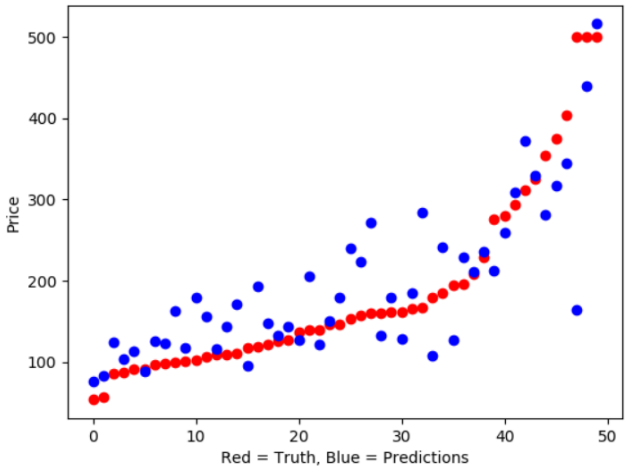
(iii) experiments and results

1a)



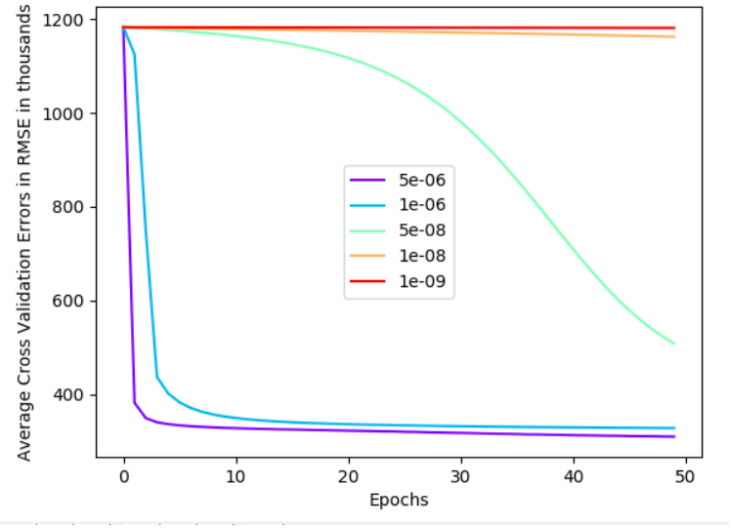
Model has converged at about 300 iterations. RMSE is about 50,000 which is quite robust when predicting houses that ranges from 100k to 500k showing that the model has learned most of the underlying trend.

1b)



The original prices were sorted in increasing order to better show the trend. Prices are in thousands. The figure above shows a strong correlation between model prediction on the test set and the actual prices showing that the model has managed to learn somewhat the underlying truth. Overall, it seems that the model predicts more accurately in terms of absolute value the price of cheaper houses as seen by the roughly diverging cone pattern of the blue dots.

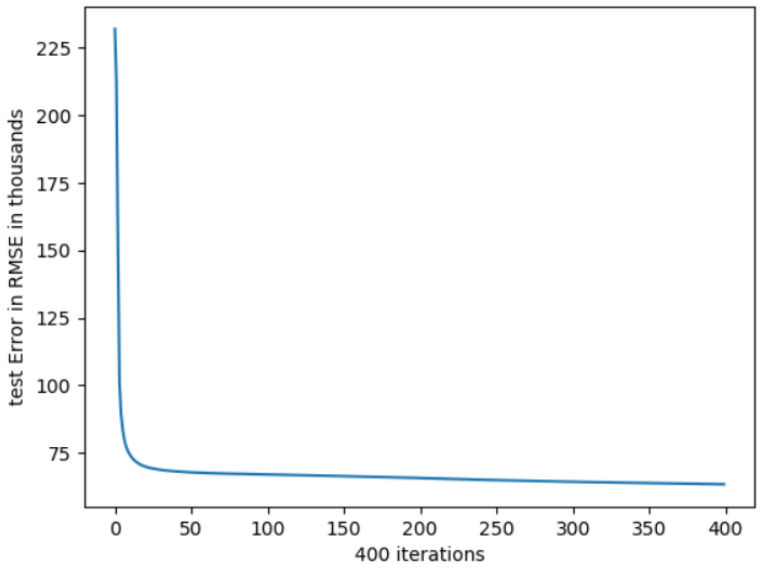
2a)



Note: Units for Epochs is 10 epochs so total of 500 epochs done.

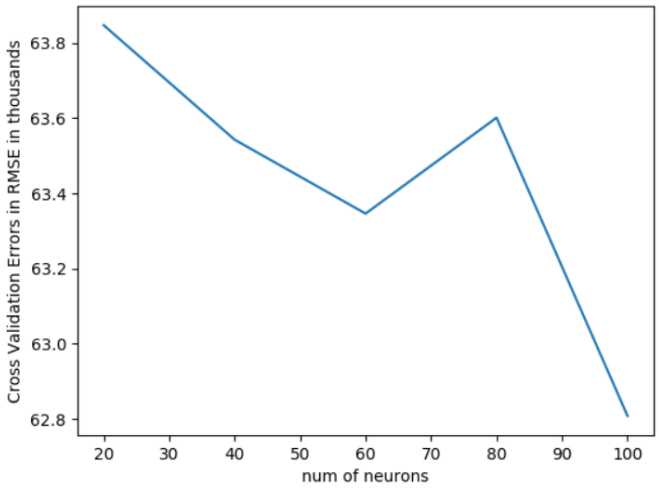
Optimal learning_rate = 5e-6 as it converges, converges quickly and has the lowest RMSE at 500 iterations. A slightly slower learning rate may have had better results at maybe 1000 iterations but due to time considerations that would be impractical and 5e-6 is best for now.

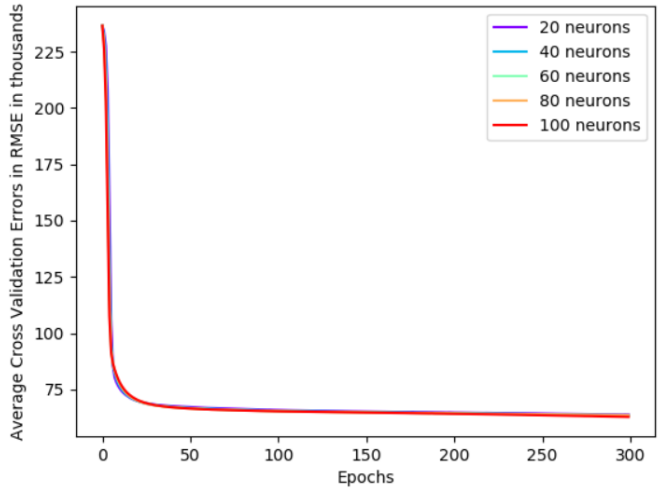
2b)



Test error plateaus and does not rise showing that overfitting did not occur. This shows that the model complexity is not too high for predicting housing prices. The large amounts of data (over 20,000 data point) also aided in preventing over-fitting.

3a)



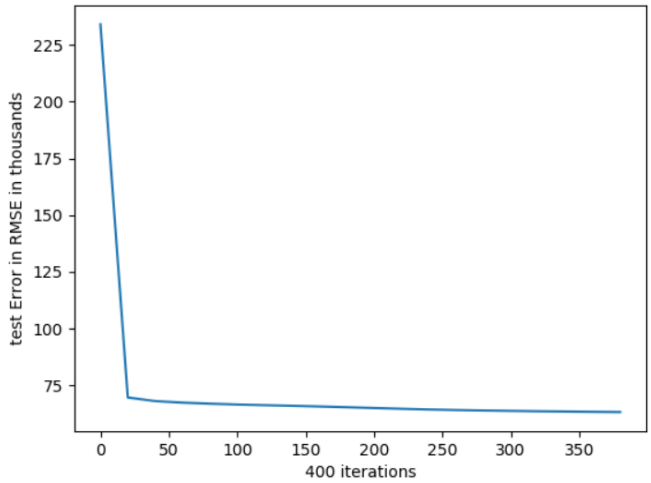


All models start to converge after about 50 epochs.

Time taken =[627, 643, 638, 640, 639]

Not much difference in time taken probably due to GPU being used. In this case we will use 100 neurons as it had the lowest validation error.

3b)

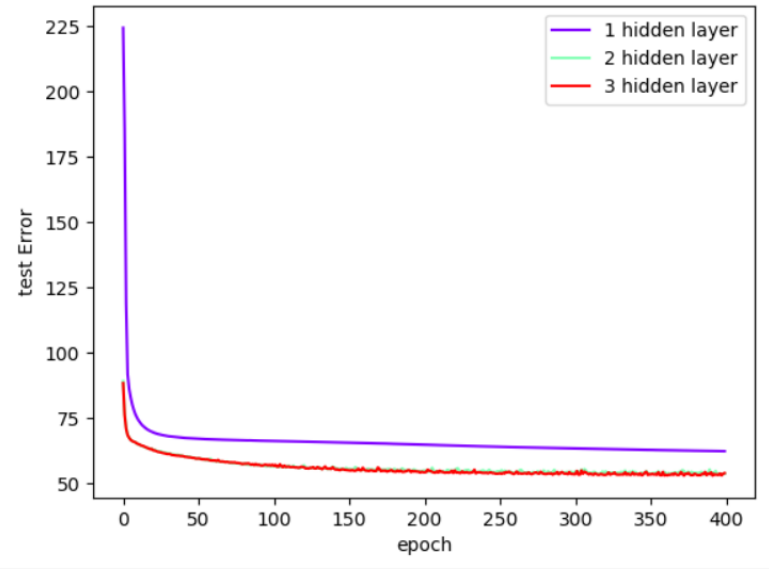


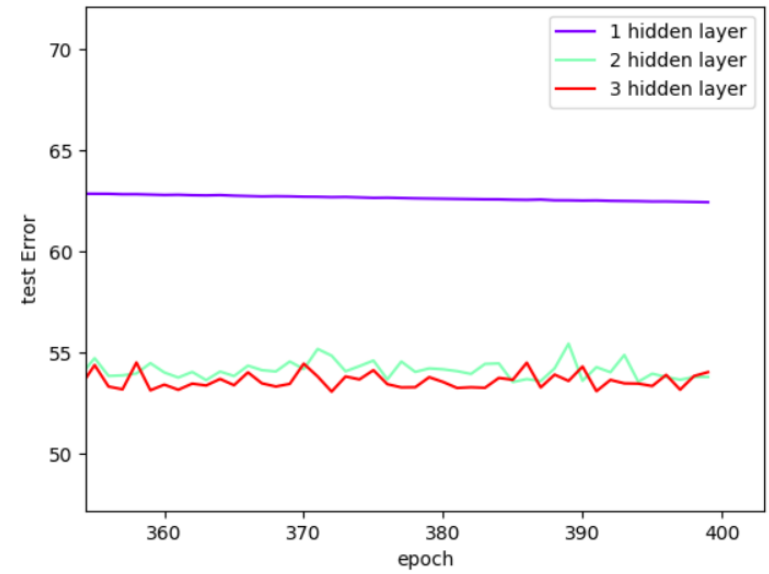
Test error plateaus and does not rise showing that overfitting did not occur. This shows that the model complexity is not too high for predicting housing prices even though 100 neuron per layer was used. The large amounts of data (over 20,000 data point) also aided in preventing over-fitting.

3c) The computation time for the different number of neurons did not change much due to GPU being used. The utilisation was usually below 10% showing that the computation power was not the limiting factor and we can use the most complex model without having a much longer training time.

Thus, the 100 neuron model was chosen as it had the lowest validation error. Part 3b also showed that the model was not too complex such that over-fitting occurs.

4)

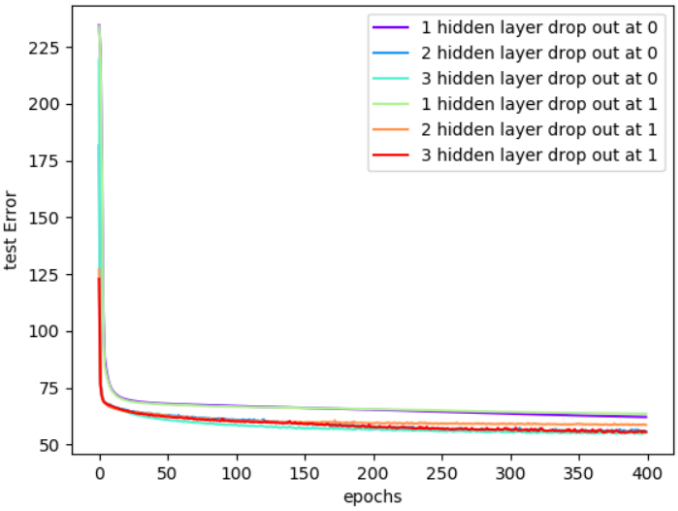


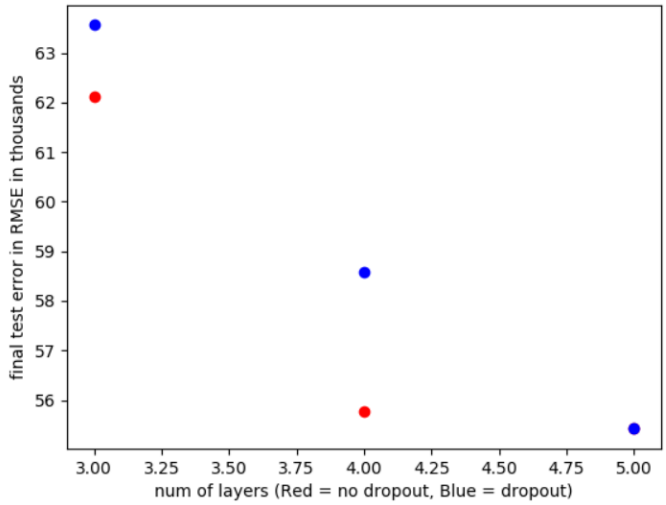


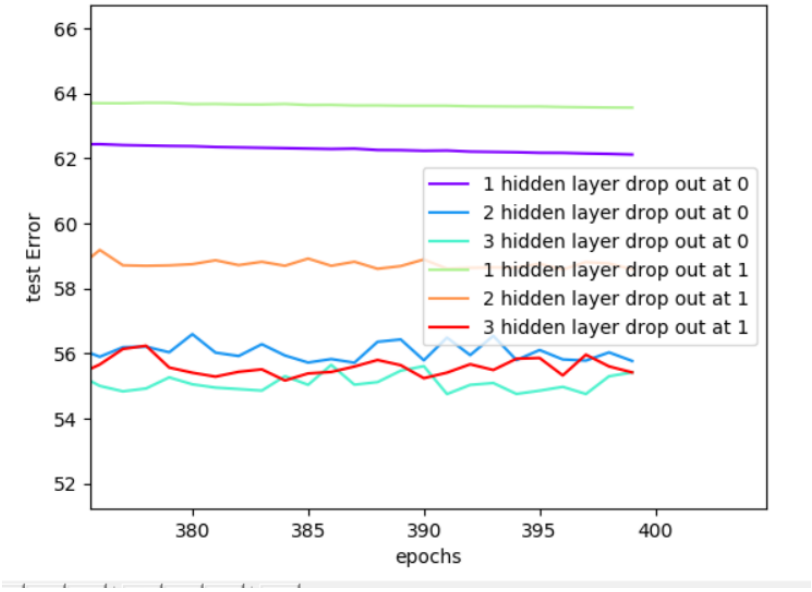
(x axis should say number of layers)

Having 4 and 5 layers produced better results than 3 layers as the model complexity increases and can better predict housing prices. Observing the variations between 2 and 3 hidden layer in the zoomed in section shows that 3 hidden layer should produce better results.

4b)


1 hidden layer drop out at 0
2 hidden layer drop out at 0
3 hidden layer drop out at 0
1 hidden layer drop out at 1
2 hidden layer drop out at 1
3 hidden layer drop out at 1
test Error
epochs
225
200
175
150
125
100
75
50
0
50
100
150
200
250
300
350
400


final test error in RMSE in thousands
num of layers (Red = no dropout, Blue = dropout)
63
62
61
60
59
58
57
56
3.00
3.25
3.50
3.75
4.00
4.25
4.50
4.75
5.00



Overall, it seems that there is little effect of dropout on the results.

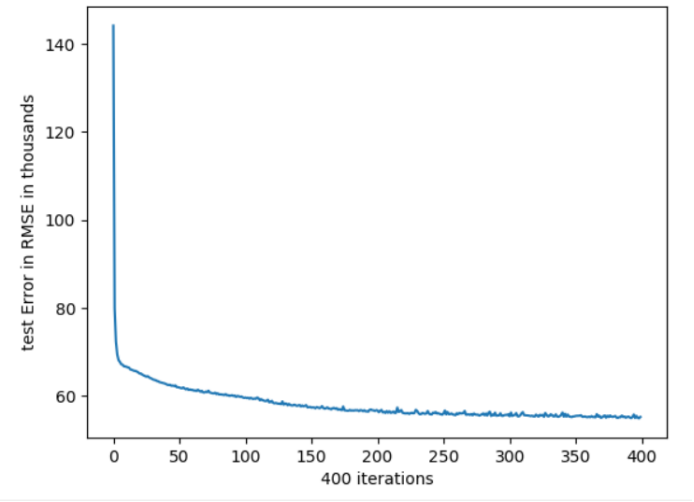
The main aim of dropouts is a form of regularization which prevents overfitting. However, as test errors did not show any increasing trend, this phenomenon did not occur. One reason could be that the data, more than 20,000 data points was a lot and more than sufficient to prevent over fitting from occurring as the underlying trend in such a large number of data points overpowers noise in the data.

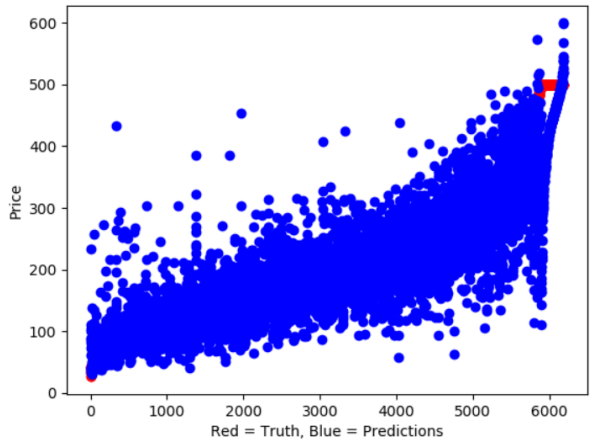
Thus, dropout did not do much to improve accuracy of the model.

The best model is 3 hidden layers with no dropout.

iv) Conclusion

With the final 5-layer model with no dropout, the final test error after 400 epochs is 3140.9082.





A quick plot shows a strong correlation between predicted price and actual price. The model also seems to predict lower prices better than higher prices given by the range of prediction.

Overall, the model can predict within +- 100k of most predictions. However, this accuracy is probably not within a good accuracy range for most uses.

One way to make this better is to make a more complex model. Currently, over fitting does not occur which seems to suggest that our model is at least not much more complex than the true underlying model.

Also, we could investigate incorporating more features as our current feature set may not truly capture the price of houses. We could add features such as style of the house, pictures of the house, prices of nearby houses, region that the house is in, distance from city, distance from MRT etc.