

NNDL ASSIGNMENT 1



14/10/2021

*Neural Networks and Deep
Learning*

Vedula Kartikeya

U1923891G

SCSE, Year 3

Table of Contents

1. Part A.....	3
2. Part B.....	11

Brief Description of Parts



Part A of this assignment focuses on building neural networks to classify the GTZAN dataset.

The aim is to predict the genre of the corresponding audio files in the test dataset after training the neural network on the training dataset. The genres are blues, classical, country, disco, hip-hop, jazz, metal, pop, reggae, and rock.

We read the data from the pre-processed csv file: features_30_sec.csv. Each data sample is a row of 60 columns, which consist of filename (where you can find the original audio file if you like), length of audio, label, and the 57 features which I will be using in the experiments.

We use the start_1a.ipynb file for this part.



Part B of this assignment focuses on the resale prices of Housing Development Board (HDB) flats. The aim is to analyze the data more deeply to see if there are other factors behind this increase. This brings us to 2 main goals of the experiments under this part:

1. Perform retrospective2 prediction of HDB housing prices
2. Identify the most important features that contributed to the prediction by using Recursive Feature Extraction algorithm

We use the start_1b.ipynb file for this part.

PART-A

Question 1

a)

```
num_epochs=50
batch_size=1

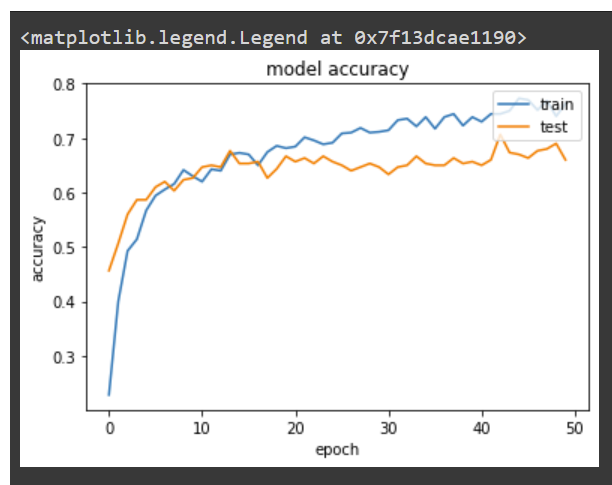
model.compile(optimizer='adam',
              loss=tf.keras.losses.CategoricalCrossentropy(),
              metrics=
                [#tf.keras.losses.CategoricalCrossentropy(name='ce'),
                 'accuracy'])

history = model.fit(X_train, y_train,
                   epochs=num_epochs,
                   batch_size = batch_size,
                   validation_data=(X_test,y_test))

Epoch 50/50
700/700 [=====] - 2s 3ms/step - loss: 0.6503 - accuracy: 0.7629 - val_loss: 1.0631 - val_accuracy: 0.6600
```

I used the training dataset to train the model for 50 epochs with batch size = 1. I used Categorical cross-entropy as a loss function which is used quite widely in multi-class classification tasks. Attained an accuracy of 0.7629 on the train data and an accuracy of 0.6600 on the test data.

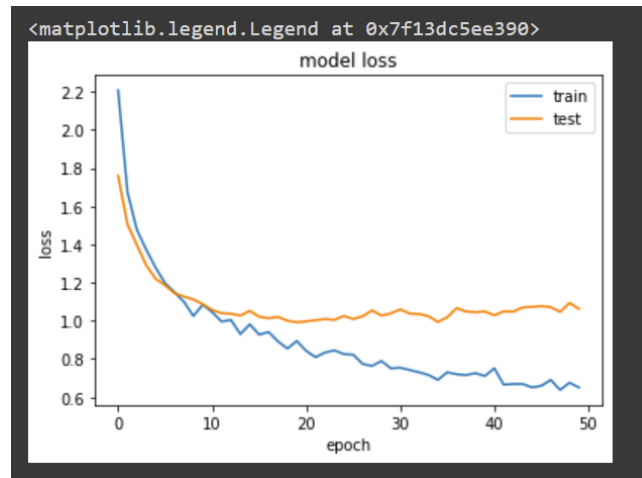
b)



Plotted the accuracies of the training and test data against the training epochs. I observed that the train and test data accuracies converge till the 17th epoch and then diverge afterwards. The snippet of that epoch is as shown below:

```
Epoch 17/50
700/700 [=====] - 2s 3ms/step - loss: 0.9412 - accuracy: 0.6500 - val_loss: 1.0133 - val_accuracy: 0.6567
Epoch 18/50
700/700 [=====] - 2s 3ms/step - loss: 0.8922 - accuracy: 0.6743 - val_loss: 1.0201 - val_accuracy: 0.6267
Epoch 19/50
700/700 [=====] - 2s 3ms/step - loss: 0.8544 - accuracy: 0.6857 - val_loss: 1.0001 - val_accuracy: 0.6433
Epoch 20/50
```

c)



Plotted the losses of the training and test data against the training epochs. I observed that the train and test data losses converge at the 7th epoch. The snippet of that epoch is as follows:

```
Epoch 6/50
700/700 [=====] - 2s 3ms/step - loss: 1.1967 - accuracy: 0.5943 - val_loss: 1.1860 - val_accuracy: 0.6100
Epoch 7/50
700/700 [=====] - 2s 3ms/step - loss: 1.1493 - accuracy: 0.6057 - val_loss: 1.1443 - val_accuracy: 0.6200
Epoch 8/50
700/700 [=====] - 2s 3ms/step - loss: 1.0998 - accuracy: 0.6157 - val_loss: 1.1272 - val_accuracy: 0.6033
```

Question 2

a)

```
batch_sizes=[1,4,8,16,32, 64]
accuracy=[]
mean_accuracy={}
num_folds=3
#replace X_train and Y_train with X and Y
for size in batch_sizes:
    history=compute_model(X_train,y_train,num_folds,size)
    accuracy.append(history.history['val_accuracy'][num_epochs-1])
    mean_accuracy[size]=np.mean(accuracy)
print(mean_accuracy)
```

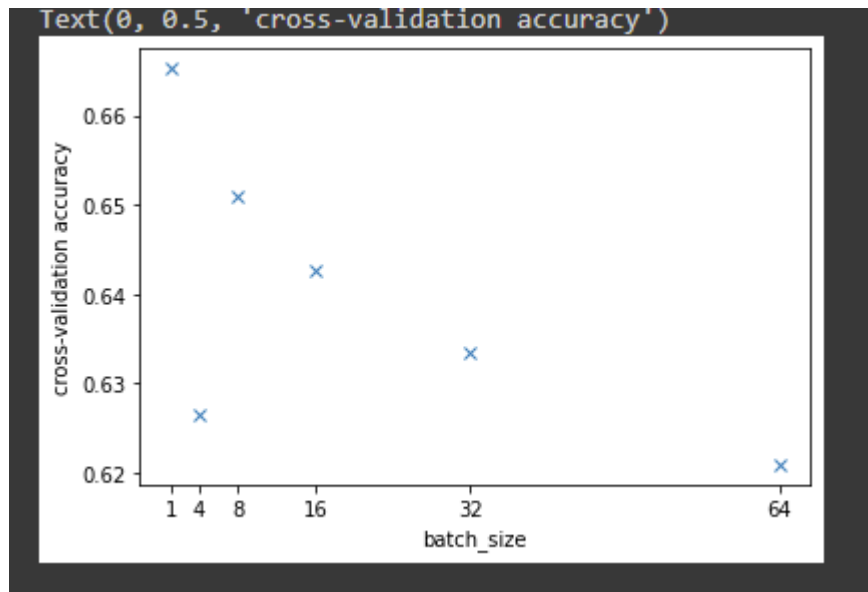
```
Epoch 46/50
8/8 [=====] - 0s 9ms/step - loss: 1.3276 - ce: 1.3301 - accuracy: 0.5310 - val_loss: 1.3025 - val_ce: 1.2964 - val_accuracy: 0.5322
Epoch 47/50
8/8 [=====] - 0s 9ms/step - loss: 1.3389 - ce: 1.3421 - accuracy: 0.5353 - val_loss: 1.2944 - val_ce: 1.2884 - val_accuracy: 0.5279
Epoch 48/50
8/8 [=====] - 0s 8ms/step - loss: 1.2716 - ce: 1.2634 - accuracy: 0.5460 - val_loss: 1.2856 - val_ce: 1.2796 - val_accuracy: 0.5451
Epoch 49/50
8/8 [=====] - 0s 11ms/step - loss: 1.3105 - ce: 1.3285 - accuracy: 0.5118 - val_loss: 1.2772 - val_ce: 1.2711 - val_accuracy: 0.5536
Epoch 50/50
8/8 [=====] - 0s 10ms/step - loss: 1.2875 - ce: 1.3101 - accuracy: 0.5332 - val_loss: 1.2694 - val_ce: 1.2631 - val_accuracy: 0.5579
{1: 0.6652360558509827, 4: 0.626094446182251, 8: 0.6509299079577128, 16: 0.6427038758993149, 32: 0.6334764003753662, 64: 0.6208869814872742}
```

Implemented a function *compute_model()* which takes in the parameters() and returns the history object:

We need to use 3-fold cross validation on training partition to perform parameter selection hence the num_folds is 3.

Accuracy is a list that stores the validation accuracies throughout the 3-fold cross validation via the history object.

Mean_accuracy is a dictionary that stores mean cross validation accuracies for each batch size.



Plotted the mean cross-validation accuracies over the training epochs for different batch sizes(1,2,4,8,16,32)

- b) Implemented a timing callback function to capture the time taken by each batch to execute an epoch

```
class TimingCallback(Callback):
    def __init__(self, logs={}):
        self.times=[]
    def on_train_batch_begin(self, batch, logs={}):
        self.starttime = time.time()
    def on_train_batch_end(self, batch, logs={}):
        self.times.append(time.time()-self.starttime)
```

Implemented a function median_time_taken that calculates the median time taken by each batch to execute an epoch using the callback "cb". Returns median_time dictionary that contains information of median time taken for every batch

This is the median_accuracy dictionary which stores the median time taken returned by the function median_time_taken:

```
{1: 0.002267599105834961, 4: 0.0027136802673339844, 8: 0.002705812454223633, 16: 0.0027011632919311523, 32: 0.002711772918701172, 64: 0.0027788877487182617}
```

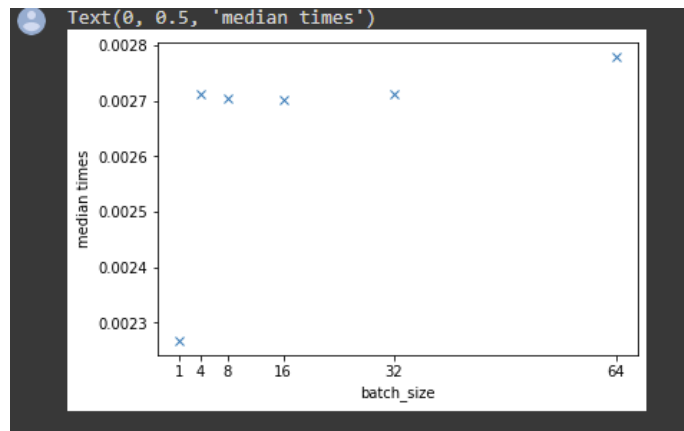
The median table is as follows:

```
#print median time taken in the format of table
print("{:<10} {:<10}".format('Batch Size', 'Median time taken'))

# print each data item.
for key, value in median_time.items():
    print("{:<10} {:<10}".format(key, value))

Batch Size Median time taken
1          0.002267599105834961
4          0.0027136802673339844
8          0.002705812454223633
16         0.0027011632919311523
32         0.002711772918701172
64         0.0027788877487182617
```

c)



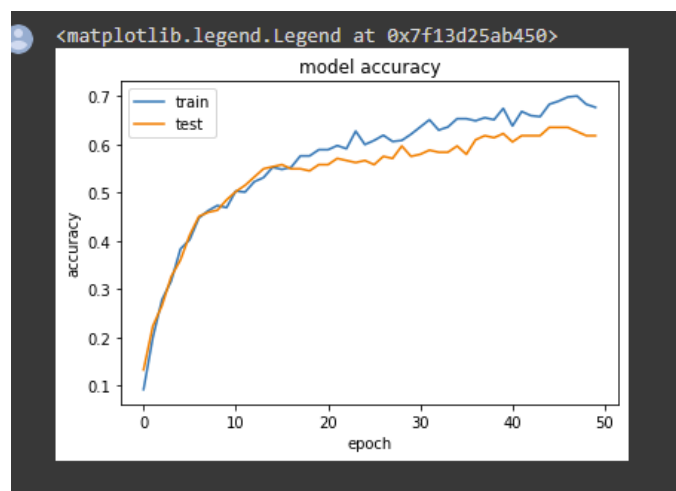
Optimal batch size is 8.

We can see from the plot of cross-validation accuracy against batch size that the validation accuracy for batch size 1 and 4 is very irregular whereas for batch sizes 8,16,32 the values follow a trend and cross validation accuracy of batch size 8 is higher than batch sizes 16,32,64. From the plot of median time taken against batch size we can infer that after batch size 8, median time taken is approximately constant. Hence, the optimal batch size is 8.

d) Deep Learning models seek more data. If we have more data, the chances of the model being good are higher. Batch size 1 is essentially stochastic gradient descent. In stochastic gradient descent, we train the neural network for every training input.

Batch size greater than 1 and less than the training dataset is mini batch gradient descent learning. In mini batch gradient descent, we train the neural network on a batch of training inputs which is less than actual size of dataset.

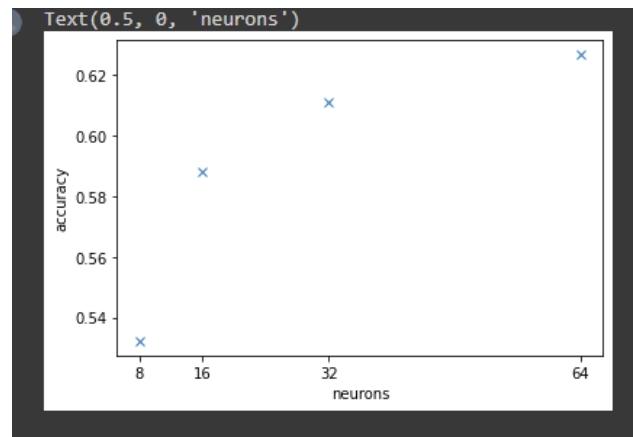
e)



Plotted the train and test accuracies against epochs for the optimal batch size i.e., 8

Question 3

a)

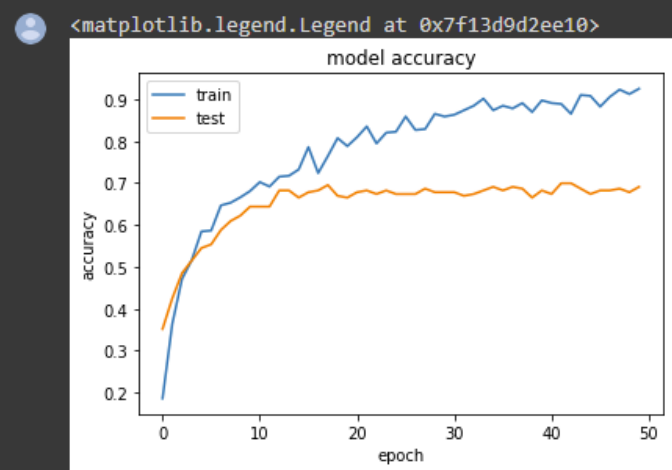


Plotted the cross validation accuracies against the hidden-layer neurons, number of neurons: 8,16,32,64

b) & c) Combined because they can be answered together without the information being redundant.

Optimal Neurons (Assumed)	Accuracy Graph
16	<p>A line graph titled 'model accuracy' showing training (blue) and test (orange) accuracy over 50 epochs for 16 neurons. The y-axis ranges from 0.2 to 0.7. Both accuracies increase and stabilize around 0.65 after 20 epochs.</p>
32	<p>A line graph titled 'model accuracy' showing training (blue) and test (orange) accuracy over 50 epochs for 32 neurons. The y-axis ranges from 0.2 to 0.8. Both accuracies increase and stabilize around 0.7 after 20 epochs.</p>

64

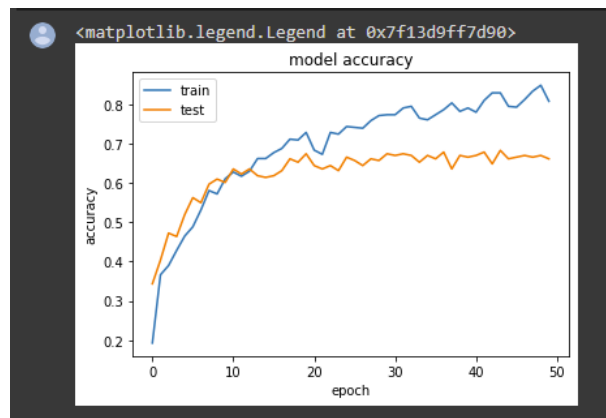


Optimal number of neurons is 16. We can see from the plots that more neurons in a layer means for smaller batch size, there are more neurons, and these neurons have less training inputs to learn before updating their internal parameters thus leading to overfitting. We can see from the above plots that the accuracies of train and test data are almost equal when the number of neurons is 16 whereas for 32 and 64 neurons the graph starts diverging after a particular number of epochs.

- d) The parameter that must be tuned is number of epochs, as it helps us to gauge better how the model is behaving.

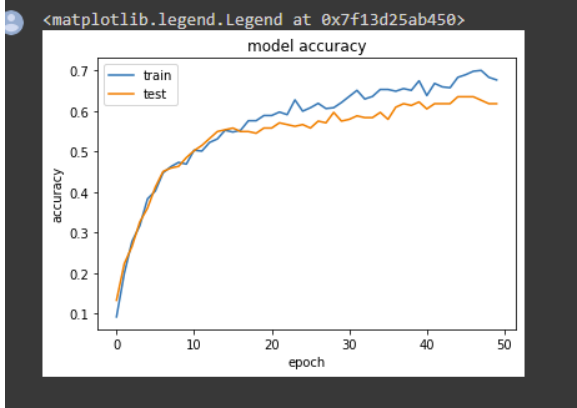
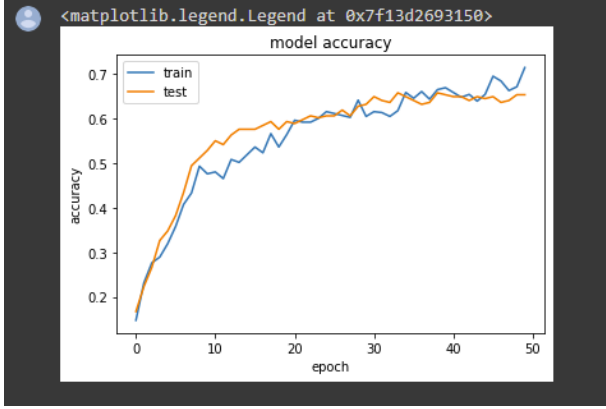
Question 4

a)



Plotted the train and test accuracies of the 3-layer network against the number of epochs

b)

Performance	Accuracy Plot
Question 2 performance	
Question 3 performance	

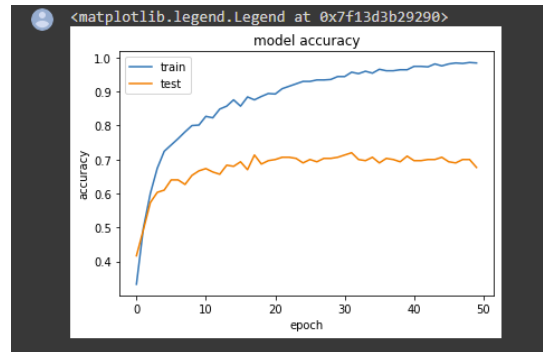
The performance has worsened once we increase the number of hidden layers from 2 to 3. This could be because the model now has more layers of neurons but less data and hence each neuron will start to memorize the patterns of data resulting in overfitting.

Question 5

- a) We add dropouts to prevent over-fitting of the model

```
model=Sequential([
    Dense(16, activation='relu'),
    #Dropout(0.3),
    Dense(10, activation='softmax')
])
```

As shown in the snippet above, I have commented out the dropouts to proceed with this experiment.



- b) As we can see, our model greatly overfits once we remove the dropouts. This proves that dropouts are important to ensure that our model learns well and gives us good results.
- c) We can use early stopping to avoid further running of the model once we notice that the accuracy does not improve anymore.

PART-B

Question 1: -

a)

```
[6] #Split data
train_dataframe = df[df['year'] <= 2020] # 0.8393471285568813
val_dataframe = df[df['year'] > 2020]
train_ds = dataframe_to_dataset(train_dataframe)
val_ds = dataframe_to_dataset(val_dataframe)
print(train_ds)

train_ds = train_ds.batch(128)
val_ds = val_ds.batch(128)

<ShuffleDataset shapes: (month: (), year: (), full_address: (), nearest_stn: (), dist_to_nearest_stn: (), dist_to_dhoby: (), degree_centrality:
```

I have divided the dataset ('HDB_price_prediction.csv') into train and test sets by using entries from year 2020 and before as training data (with the remaining data from year 2021 used as test data).

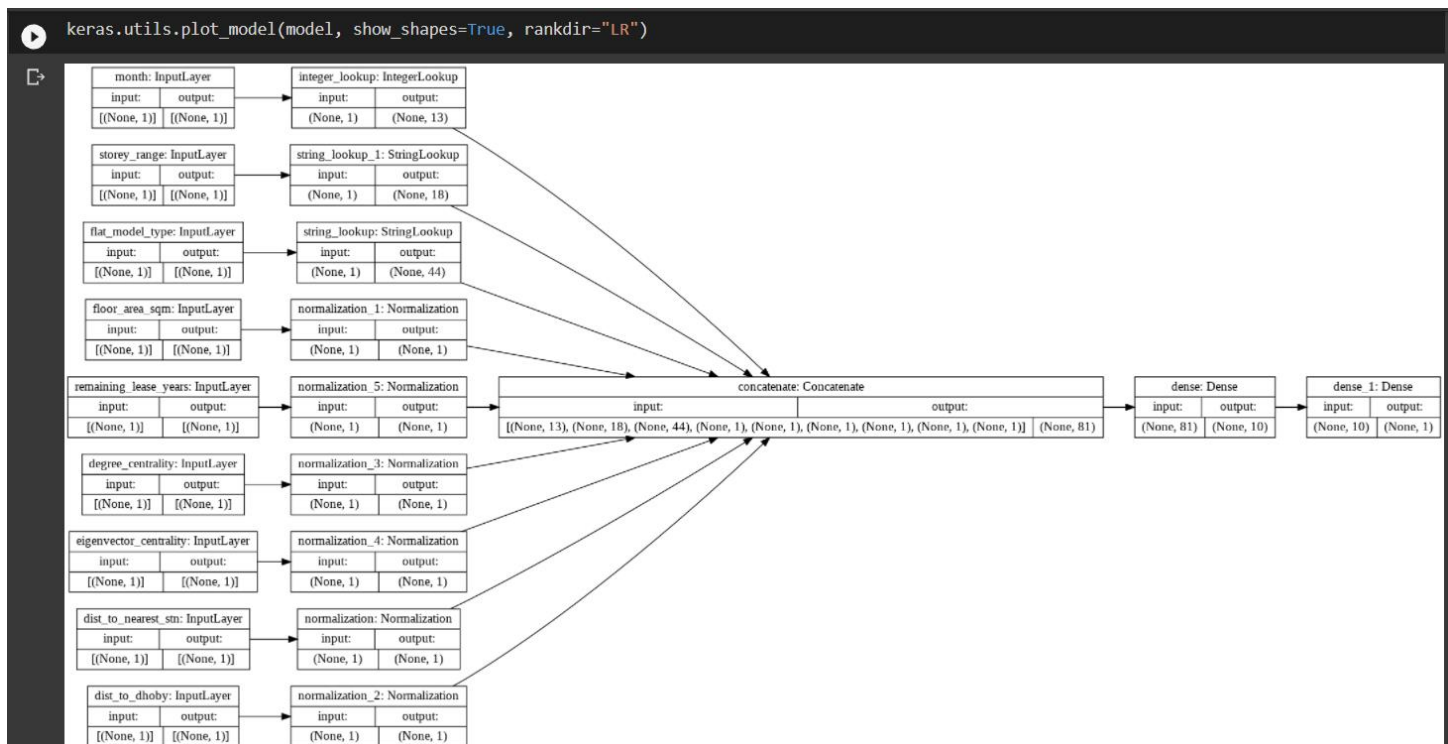
This is done instead of random train/test splits because there can be an issue of stratification.

If I do a random train/test split, I will be totally disregarding the distribution of data or proportion of classes in a dataset. In this scenario, I might end up with a train and a test set with totally different data distributions. This will create a model that will train on a vastly different data distribution than the test set and hence will perform inferiorly at validation.

b) Here will initialize all the required features as keras.Input features. This done so that later after proper encoding, we can concatenate them into a layer.

We Integer Encode the categorical variable "month", whereas, we String Encode the categorical variables "flat_model_type", "storey_range".

We get the same architecture as expected:

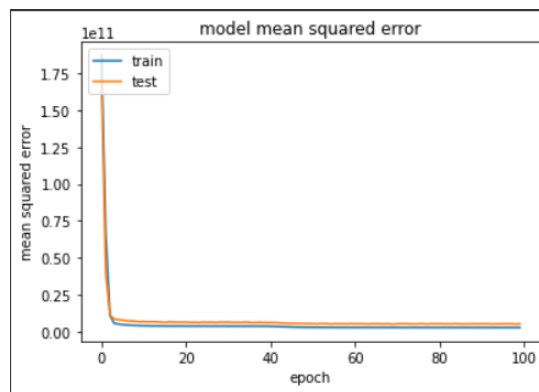


c)

```
[8] # feedforward network with one hidden layer
x = layers.Dense(10, activation="relu")(all_features)
# x = layers.Dropout(0.5)(x)
output = layers.Dense(1, activation="linear")(x)

[9] model = keras.Model(all_inputs, output)
opt=keras.optimizers.Adam(lr=0.05)
model.compile(optimizer=opt, loss='mse', metrics=[tf.keras.metrics.MeanSquaredError()])
history = model.fit(train_ds, epochs=100, validation_data=val_ds)#100 epochs needed
```

Training for 100 epochs is preferred not just because it is instructed, but it helps our model learn better. Point to note: Our inputs to the feedforward network are the “all_features” which includes the encoded inputs. This is done so that the neural networks can make proper sense of the features and make accurate predictions. After training, we plot the model mean square error vs epochs as shown below:

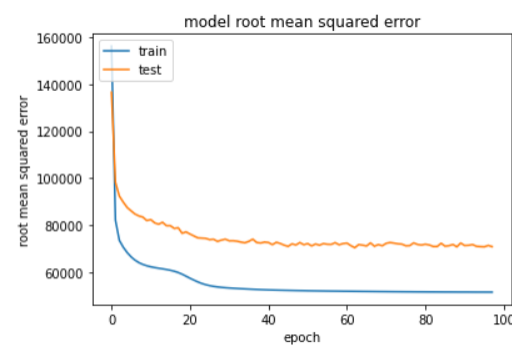


This plot shows us that the train and test mean squared error of the above model configuration are close to zero.

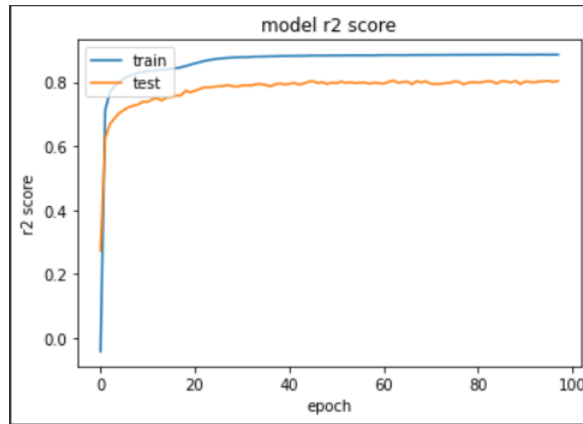
Indicating that after a certain number of epochs, difference between the original and predicted values is greatly minimized implying that our model makes good predictions and there is no deviation in performance.

d) The train and test root mean square errors (RMSE) against epochs are as shown below:

After certain epochs, the root mean squared error of the train and test eventually becomes constant in a similar manner. We can infer from this that the prediction data is more concentrated around the line of best fit.



The corresponding plot for r^2 scores for this model:



e) The calculation for test error at a particular epoch, r^2 score at that epoch is as follows:

1. Stored the entire validation loss in a single variable (lowest_test_error)
2. Found the minimum of this list by using the inbuilt function “min()”
3. Calculated the index of this minimum error via list comprehension
4. Epoch of the minimum error is index+1(because index starts from 0 and epoch starts from 1)
5. Calculated r^2 value at that epoch is calculated by accessing the “val_r2_score” attribute of *history* and index of the best epoch we found before.

These were the results obtained: -

65th epoch is the one with lowest test error.

R^2 value at that epoch is 0.8056017756462097

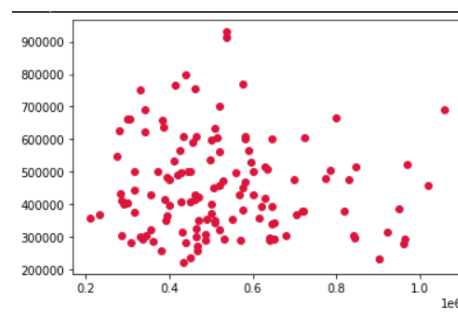
Loss value at that epoch is 4950943744.0

f) Used *Checkpoints* to capture weights of the model when we stop the model once the validation root mean square error no longer improves from a certain value.

```
Epoch 00086: val_root_mean_squared_error did not improve from 69553.62500
Epoch 87/100
683/683 [=====] - 3s 4ms/step - loss: 2614929152.0000 - root_mean_squared_error: 51136.3789 - val_loss: 5135481856.0000
```

We then load these weights into the model such that we use the model from the best epoch to plot the predicted values and target values for a batch of 128 test samples.

We use the python take() function to get a random batch of 128 test samples. The scatter plot for it is as follows:



There seems to be low correlation between random batch of 128 test samples and the predictions. This may also show that there is weak linear relationship between random batch of 128 test samples and the predictions.

Question 2: -

- a) We use Word Embeddings to provide a richer representation of words and their relative meanings. We now modify our existing encoded features to add an embedding layer after the one hot embeddings for categorical variables. Here is a snippet of one of the modifications of encoding:

In this snippet we modify the encoding of the month categorical variable:

```
from math import *
# Categorical features encoded as integers using embedding to make more meaningful sense of words
month_unique = df['month'].unique()
month_vocab_size = len(month_unique)
month_output_dim = month_vocab_size//2 #output_dim = floor(num_categories/2)
month_embedded = layers.Embedding(month_vocab_size,month_output_dim)(month_encoded)
month_flattened = layers.Flatten()(month_embedded)
```

We concatenate all these newly encoded variables into a new variable called *all_features_new*

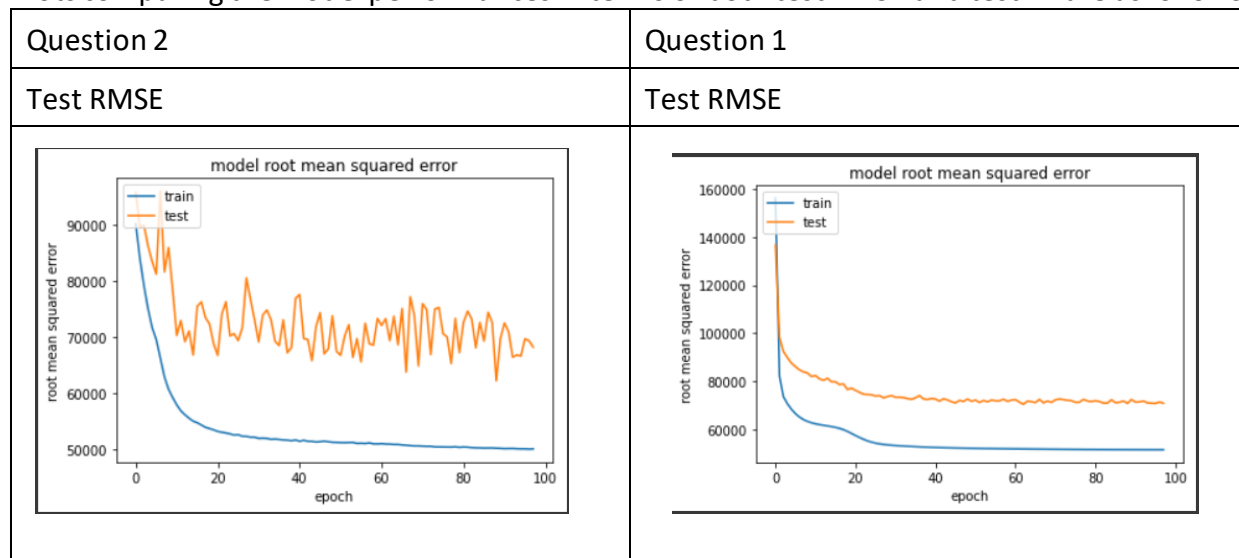
- b) The output of the Embedding layer is a 2D vector with one embedding for each word in the input sequence of words.

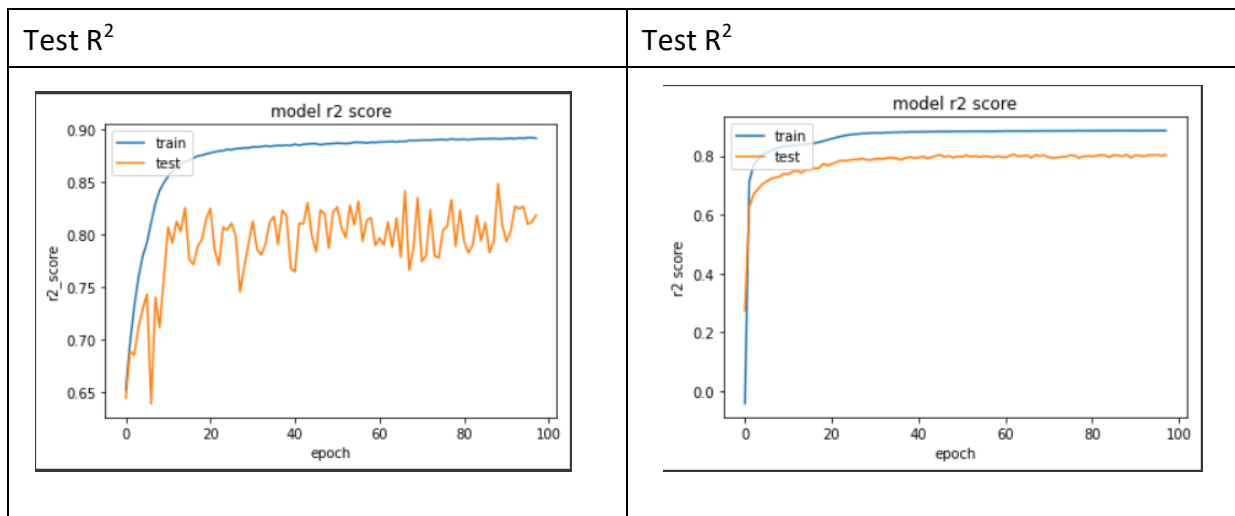
After looking into the Keras layers API, I realized that I have to add a flatten layer to such that all features can be concatenated.

This is because, to connect a Dense layer directly to an Embedding layer, we must first flatten the 2D output matrix to a 1D vector using the Flatten layer.

(We know that neurons in the feed forward neural network, take only 1D inputs)

- c) Plots comparing the model performances in terms of both test RMSE and test R^2 are as follows:





Performance of model after we add embedding layer(question2) is slightly worse than when it did not have a embedding layer(question1).

Question 3: -

- a) We use early stopping to stop halt the training of neural networks at the right time. In technical terms, Early stopping is a method that allows us to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a validation dataset.

We use the model from previous question and re-fit the model by introducing the early stopping callback attribute as an extra parameter.

At times, the first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better, i.e. the model may stop training at a local minimum of number of epochs instead of a global minimum of number of epochs.

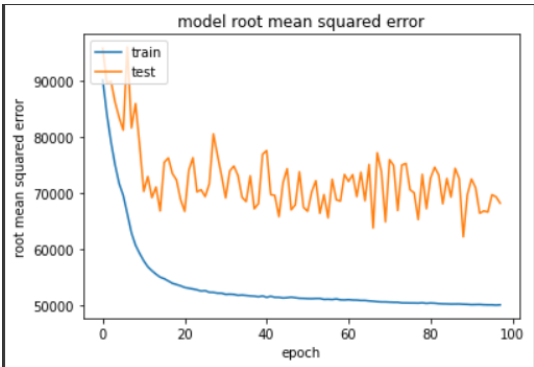
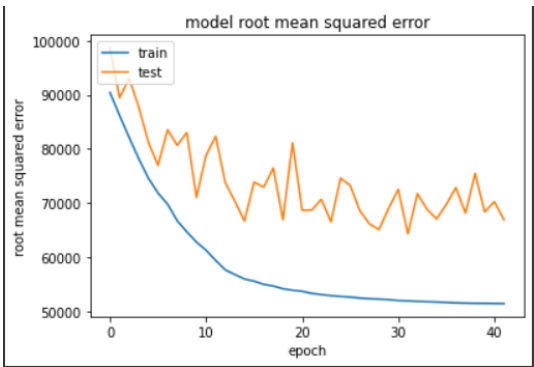
We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the “*patience*” argument. We choose a patience of 10 in the question for our experiment.

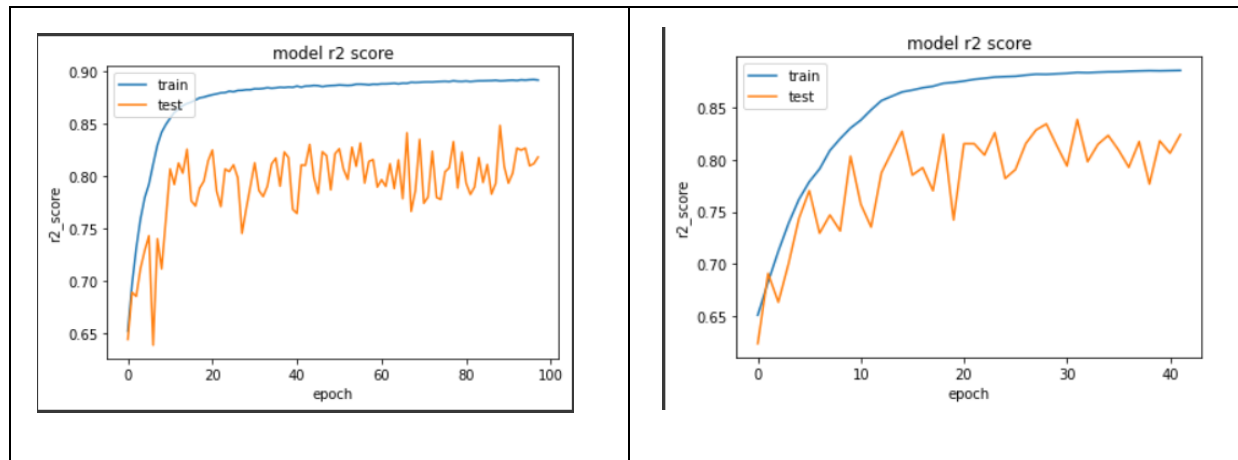
- b) Points to note in the implementation of Recursive Feature Elimination on the encoded input features is as follows:
1. Function: RecursiveFeatureExtraction(feature_list,dummy_list,best_performance_model)
 2. Variables:
feature_list = all_features_new(Refer to question 2a) and the updated versions of all_features_new once we recursively call this function. Our final feature_list will be the features that are important for prediction
dummy_list = String representation of feature_list

feature_combinations = one of the combinations of *feature_list*
feature_dummy = one of the combinations of *dummy_list*
best_performance_model = A list that stores the final list of features that are useful for prediction of hdb prices and history object of the model that ran on the final list of features
rfe = Dictionary whose *key* is the feature that was removed in one of the combinations of input features from the *dummy_list* and *value* is the model corresponding to that combination
comp_val_loss = Dictionary whose *key* is the feature that was removed in one of the combinations of input features from the *dummy_list* and *value* is the least validation loss corresponding to the model when *key* was removed from the *dummy_list*
best_history_model = *key* value of *comp_val_loss* that had the least validation loss
good_features = Those *feature_combinations* which gave a validation loss less than the validation loss that of question 2.
best_features_history = =model which gave a validation loss less than the validation loss that of question 2
best_features = Stores the features of the dummy list which gave a validation loss less than the validation loss that of question 2

3. Return:
Returns best_performance_model list

c)

Question 2	Question 3
RMSE	RMSE
	
R ²	R ²



- d) I have printed out the tensor shapes of the features that are important for prediction.
 I have printed out the history object of your best model as well.
 Our final model can miss any one of the features such as eigen_vector, month, degree centrality etc.
 Importance of these features varies greatly on how our model trains and learns from the data.
 The final model obtained after Recursive Feature Elimination has improved its performance significantly from how it performed in question 2 with all 9 features.

Thank you