CE4042 Assignment

Name: Gavin Neo Jun Hui

Matric number: U1921265L

Part A: Classification Problem

1a) Use the training dataset to train the model for 50 epochs. Note: Use 50 epochs for subsequent experiments.

```
#DNN with 1 hidden layer

def adam_optimizer(no_neurons):
    network = Sequential([
        Dense(no_neurons, activation='relu'),
        Dropout(0.3),
        Dense(10, activation='softmax')
    ])

    network.compile(optimizer='Adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
    return network
```

```
history = adam_optimizer(no_neurons=16).fit(X_train, y_train,
                                            batch_size=1,
                                            epochs=50,
                                            verbose = 2,
                                            use_multiprocessing=True,
                                            validation_data=(X_test, y_test))
```

1b) Plot accuracies on training and test data against training epochs and comment on the plots.

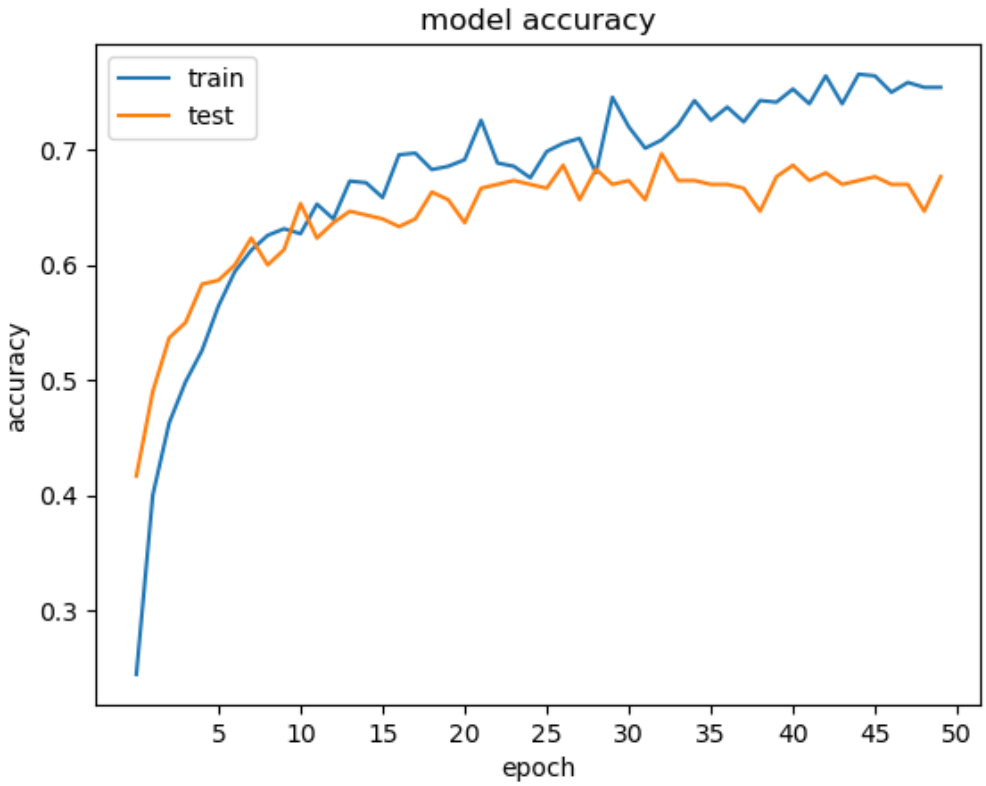


Figure 1: Accuracies against training epoch

The accuracies for both generally increase, and reaches stagnant of accuracy of 0.75 for train accuracy, and 0.67 for test accuracy

1c) Plot the losses on training and test data against training epochs. State the approximate number of epochs where the test error begins to converge.

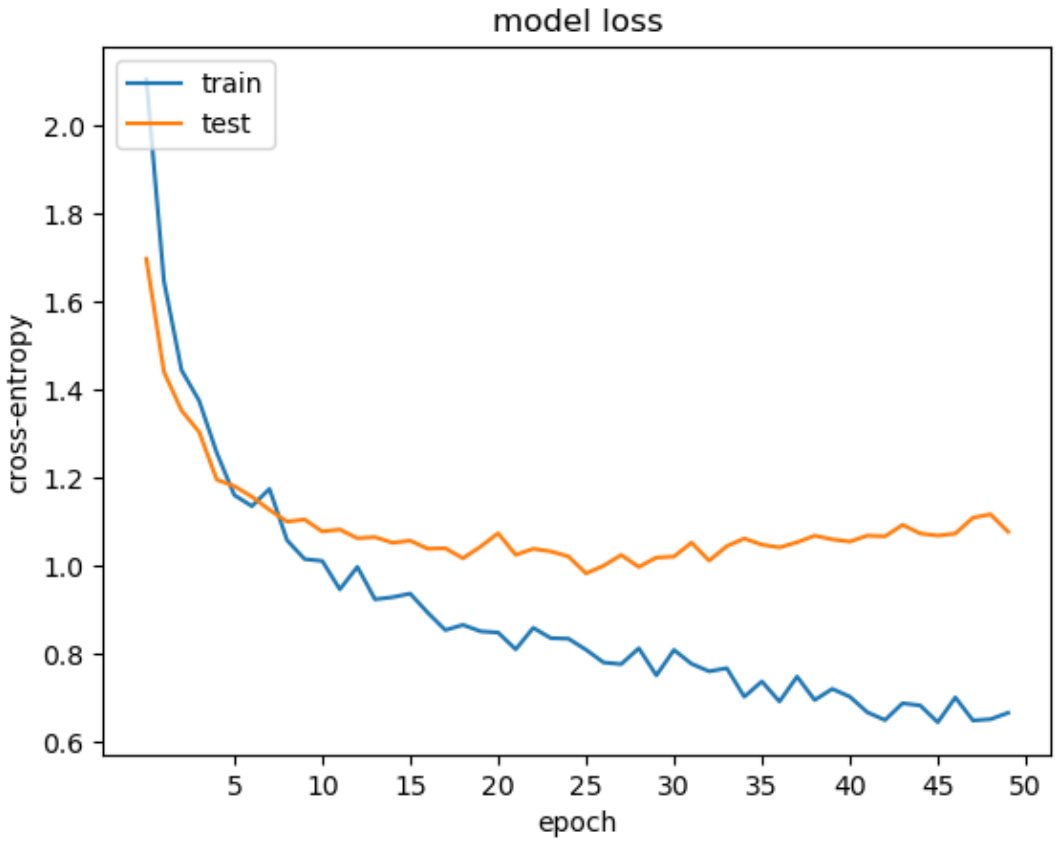


Figure 2: Accuracies against training epoch

Number of epochs where the test error begins to converge = 5

2a) Plot mean cross-validation accuracies over the training epochs for different batch sizes. Limit search space to batch sizes {1,4,8,16,32, 64}.

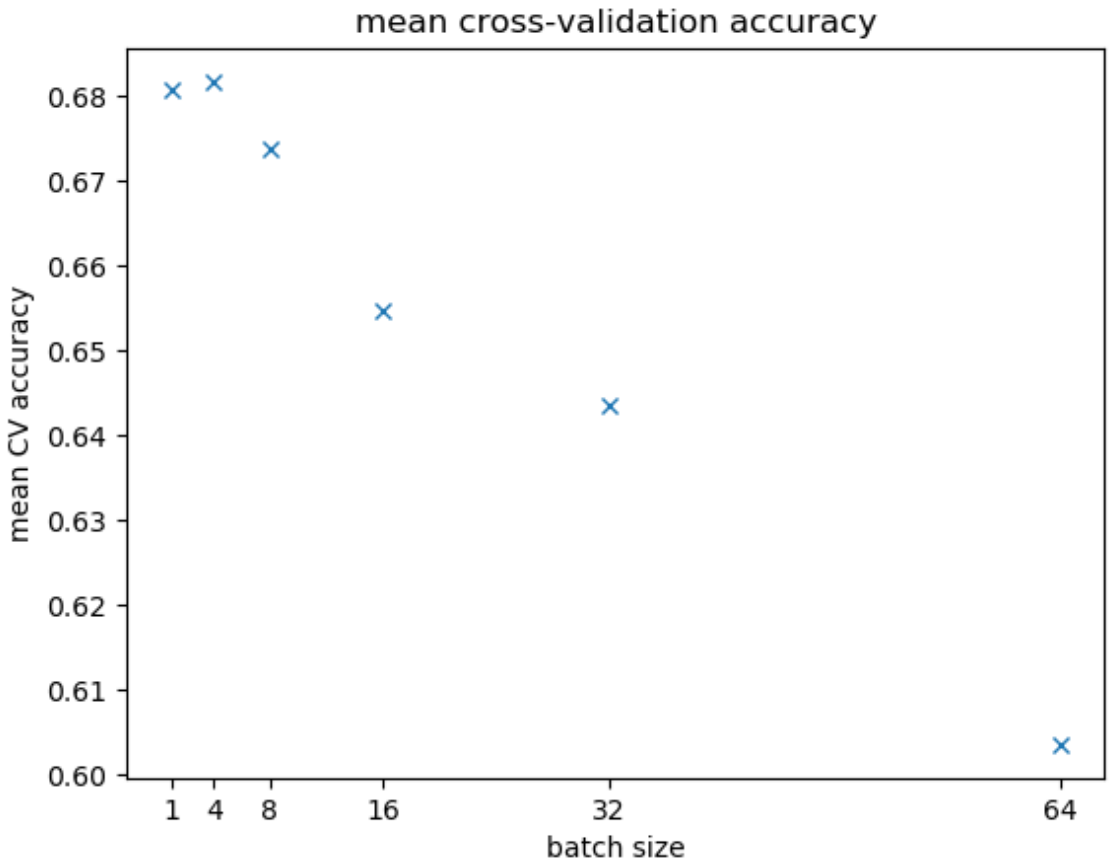


Figure 3: Mean CV accuracies against batch size

2b) Create a table of median time taken to train the network for one epoch against different batch sizes.

	1	4	8	16	35	64
median times	1.477793	1.058871	0.764039	0.595711	0.485089	0.407989

Figure 4: median times taken for each batch size

2c) Select the optimal batch size and state reasons for your selection.

Selecting 4 as the optimal batch size as it has the highest accuracy among all batch size, and with a small increase in the median time taken for training as compared to batch sizes 8 and above.

2d) What is the difference between mini-batch gradient descent and stochastic gradient descent and what does this mean for model training?

Stochastic gradient descent(SGD) is where the batch size is 1 and the update on the weights and bias is done on each epoch. While mini-batch GD is choosing batch size to be more than 1 and lesser than the number of data set, which the gradients are evaluated on the batch size in each epoch in random order.

Applying in model training, having higher batch size will generally decrease the accuracy of the training, but the time taken will be faster. Thus the consideration for the optimal batch size for training is to find the balance between the accuracy and the time taken.

2e) Plot the train and test accuracies against epochs for the optimal batch size.

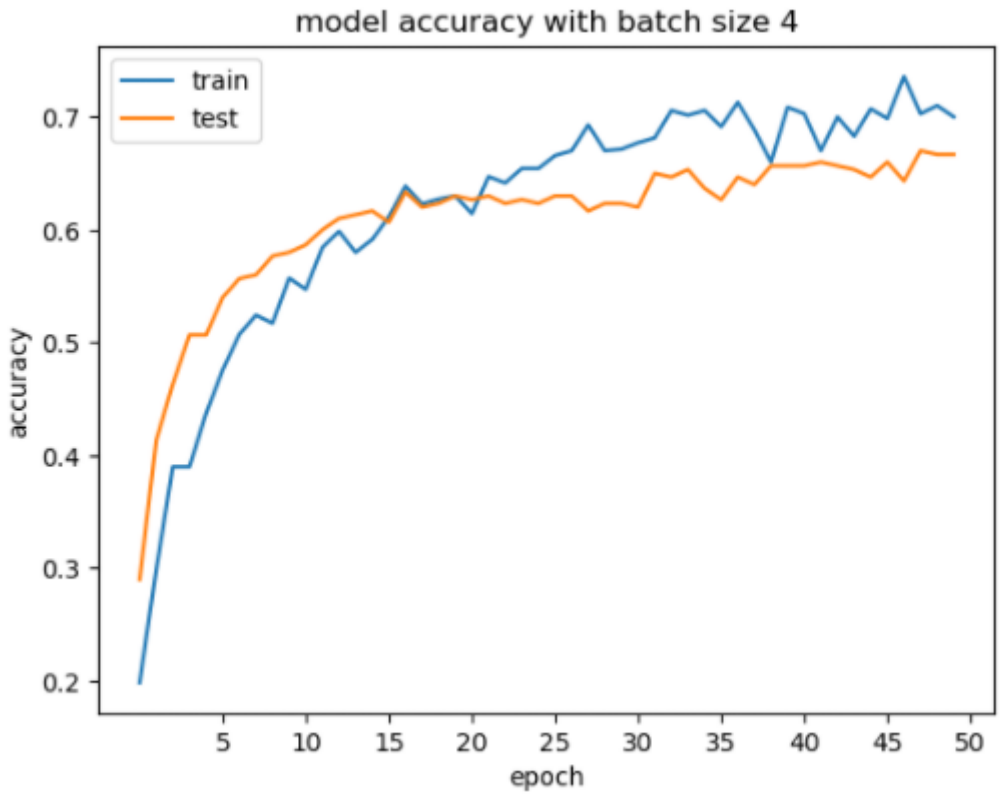


Figure 5: Accuracies against epoch with batch size 4

3a) Plot the cross-validation accuracies against training epochs for different numbers of hidden-layer neurons. Limit the search space of the number of neurons to {8, 16, 32, 64}. Continue using 3-fold cross validation on training dataset.

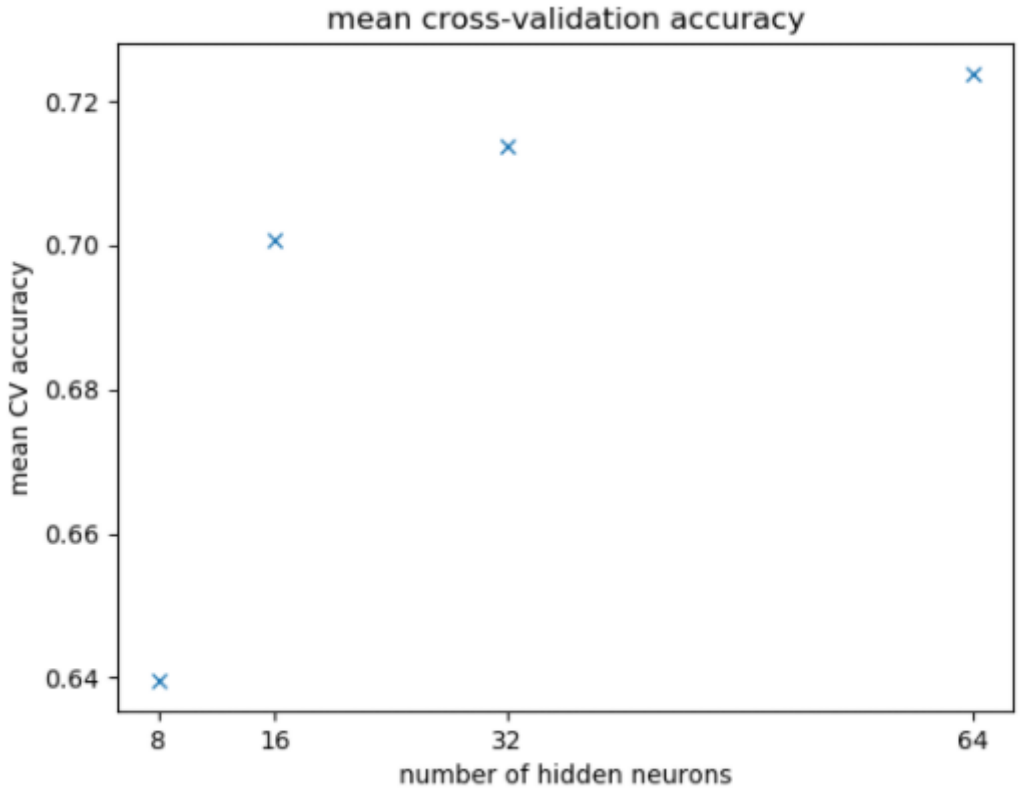


Figure 6: CV accuracies for different number of neurons

3b) Select the optimal number of neurons for the hidden layer. State the rationale for your selection.

Selecting 64 as the optimal number of neurons, as the model have the highest accuracy with 64 neurons.

3c) Plot the train and test accuracies against training epochs with the optimal number of neurons.

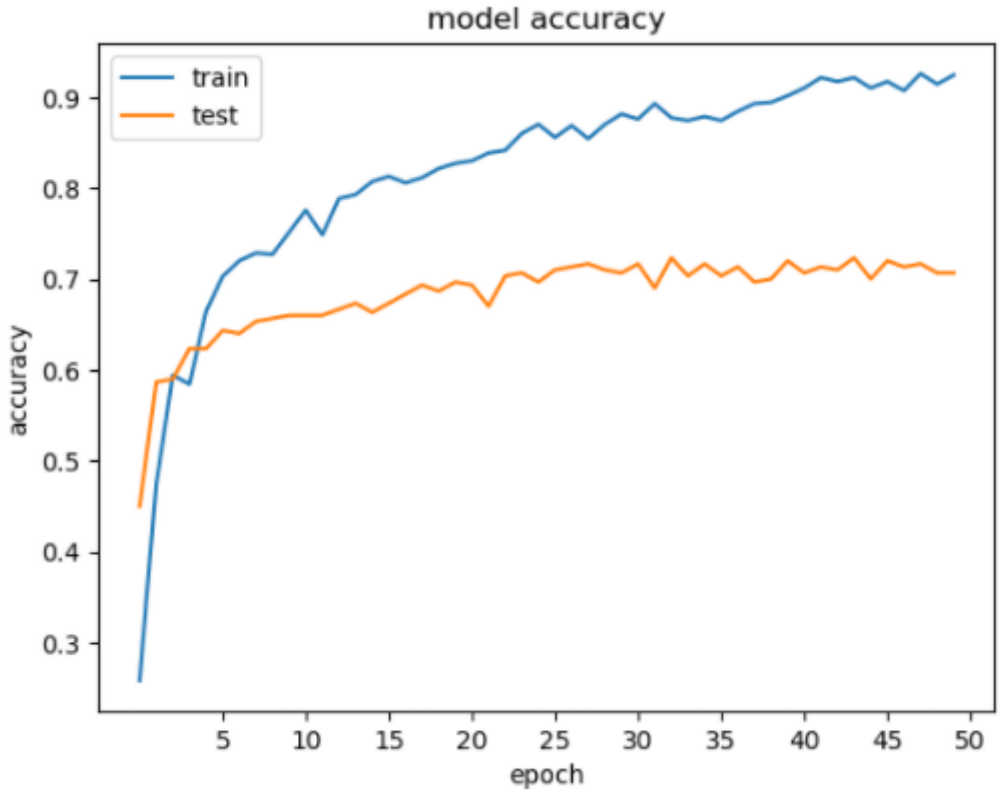


Figure 7: Accuracies against epoch with 64 hidden neurons

3d) What other parameters could possibly be tuned?

Other parameters such as the learning rate and the number of hidden layers can be tuned for the model.

4a) Plot the train and test accuracy of the 3-layer network against training epochs.

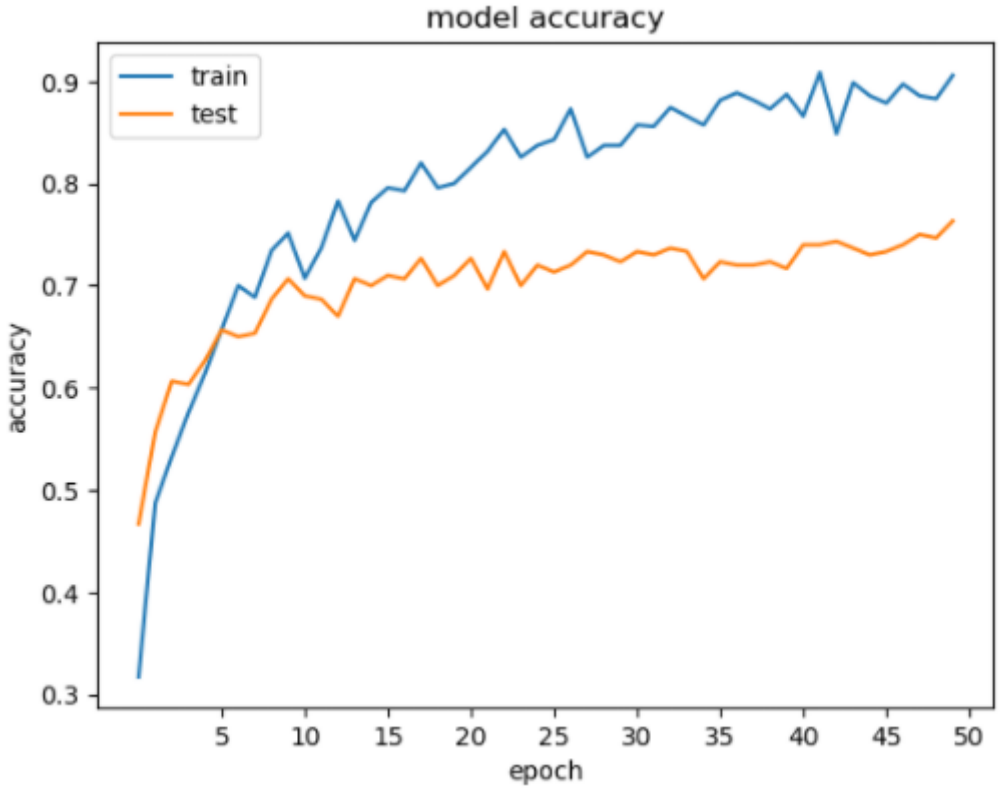


Figure 8: Accuracies against epoch for 3-layer network

4b) Compare and comment on the performances of the optimal 2-layer network from your hyperparameter tuning in Question 2 and 3 and the 3-layer network.

The 3-layer network generally have higher accuracy than the optimal 2-layer network. This shows that the adjusting the number of layers affects the accuracy more compared to hyperparameter tuning of batch size and hidden neurons from questions 2 and 3.

5a) Why do we add dropouts? Investigate the purpose of dropouts by removing dropouts from your original 2-layer network (before changing the batch size and number of neurons). Plot accuracies on training and test data with neural network without dropout. Plot as well the losses on training and test data with neural network without dropout.

Removing dropout with 64 neurons and 1 batch size

```
#DNN without dropout

def adam_no_dropout(no_neurons):
    network = Sequential([
        Dense(no_neurons, activation='relu'),
        Dense(10, activation='softmax')
    ])

    network.compile(optimizer='Adam',
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])
    return network
```

```
no_dropout = adam_no_dropout(no_neurons=16).fit(X_train, y_train,
                                                batch_size=1,
                                                epochs=50,
                                                verbose = 2,
                                                use_multiprocessing=True,
                                                validation_data=(X_test, y_test))
```

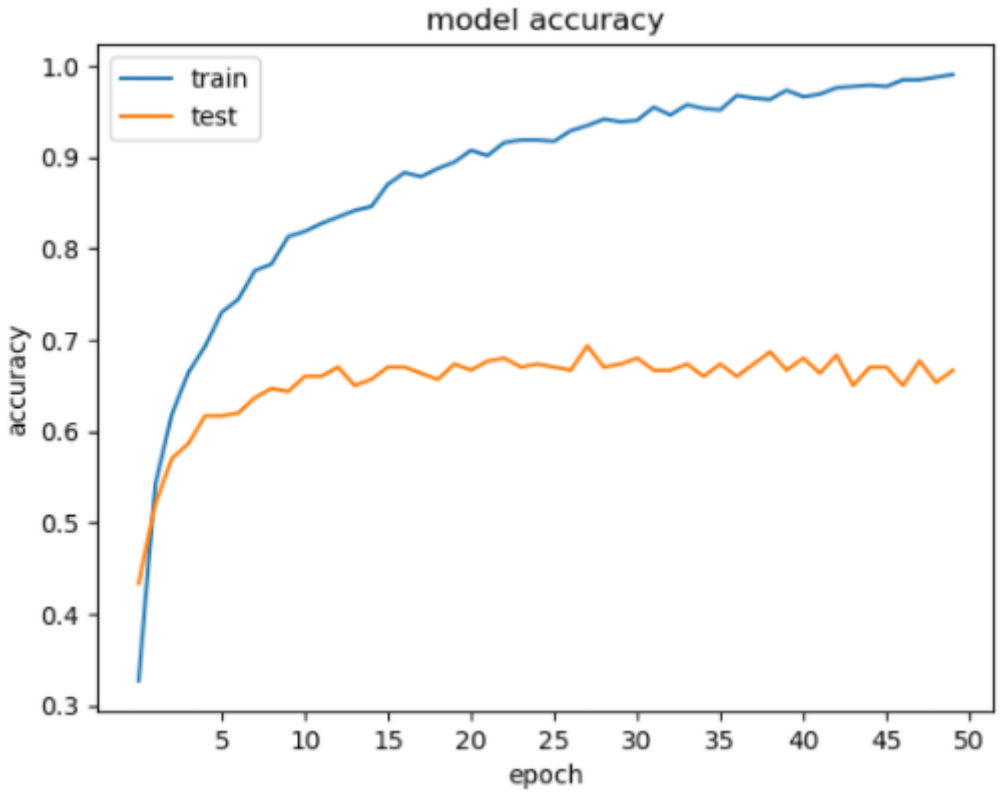


Figure 9: Accuracies against epoch without dropout

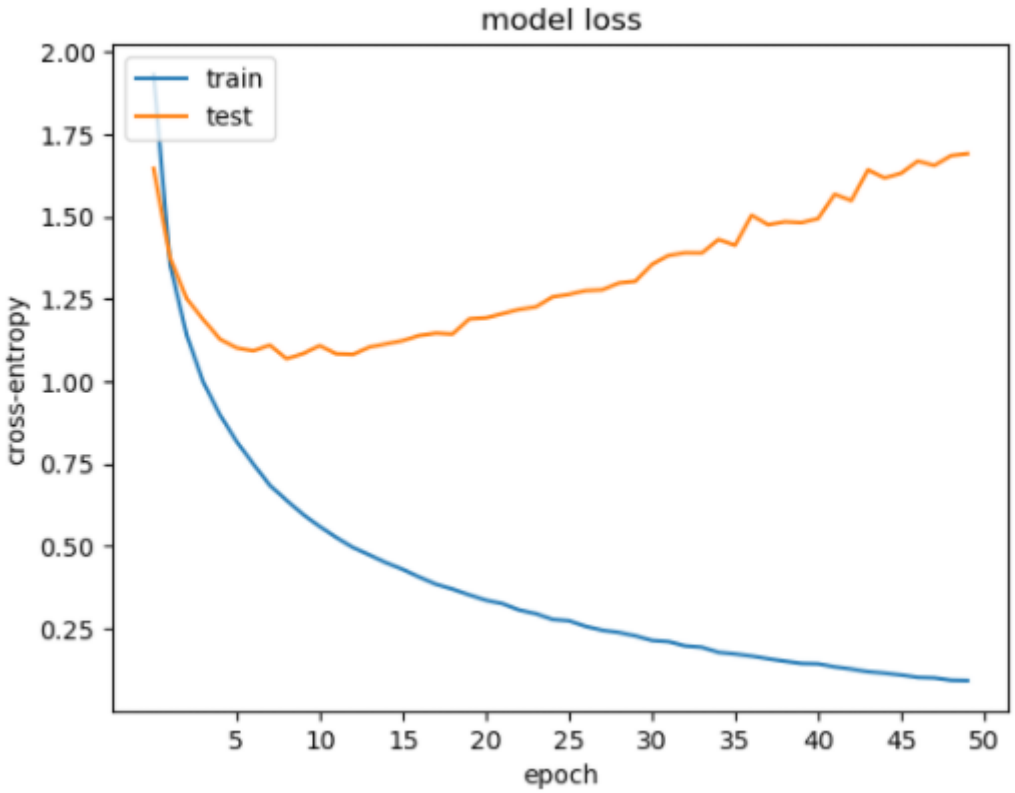


Figure 10: Losses against epoch without dropout

5b) Explain the effect of removing dropouts.

Without dropouts, overfitting occurs which thus the model is unable to predict correct outputs to novel inputs, which thus causing the accuracy and loss to be worst as compared to adding dropouts.

5c) What is another approach that you could take to address overfitting in the model?

Using early stopping when overfitting starts to occur.

Conclusion

Of all the parameters tuned, adding a hidden layer is the most impactful in model performance and compared to others. A possible reason is due to having more calculations involved for better precisions and accuracies.

A limitation of using feed forward network to predict based on features obtained can be that features of analog waves will be lost when processing factors , which results in inaccuracy.

Part B: Regression Problem

1a) Divide 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). Why is this done instead of random train/test splits?

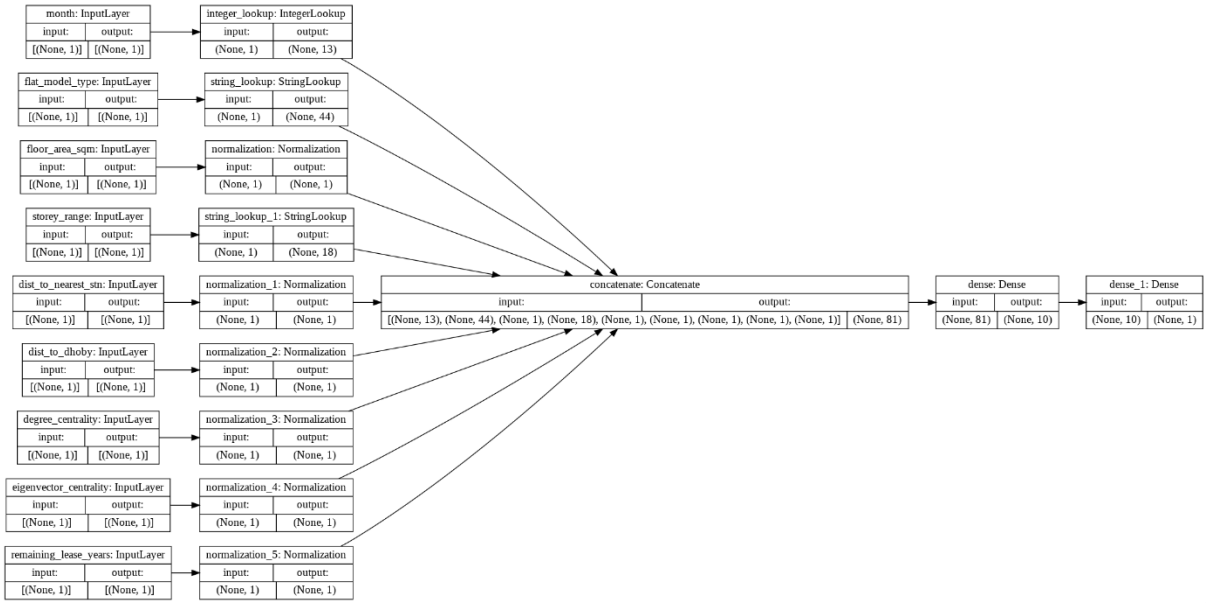
Code to split the dataset:

```
#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)

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

This is such that the year would not be used as a factor of the resale price and to look in other possible factors instead.

1b) Design a 2-layer feedforward neural network

Architecture of the model:



1c) On the training data, train the model for 100 epochs using mini-batch gradient descent with batch size = 128, Use 'adam' optimiser with a learning rate of α = 0.05 and mean square error as cost function. (Tip: Use smaller epochs while you're still debugging. On Google Colaboratory, 100 epochs take around 10 minutes even without GPU.)

Code for training model:

```
model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.05), loss="mse", metrics=r_square)

model.fit(train_ds,
          batch_size=128,
          epochs=100,
          verbose = 2,
          use_multiprocessing=True,
          validation_data=val_ds,
          callbacks=[model_checkpoint_callback])
```

1d) Plot the train and test root mean square errors (RMSE) against epochs (Tip: skip the first few epochs, else the plot gets dominated by them).

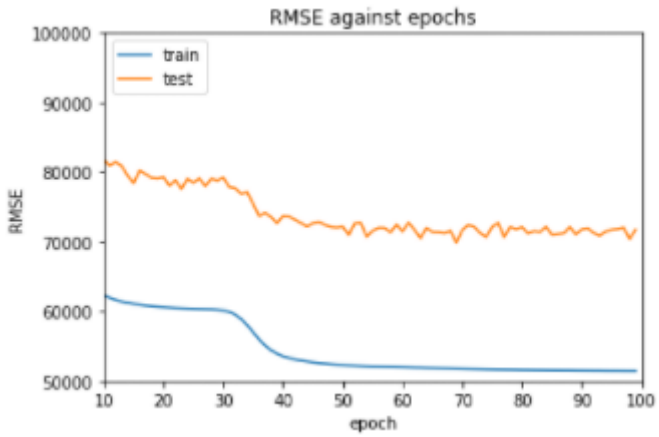


Figure 11: RMSE against epochs

1e) State the epoch with the lowest test error. State the test R2 value at that epoch. (Hint: Check the output returned by model.fit(). Use a custom metric for computing R2 .)

Lowest test error = 69847.4974211675

Epoch number = 70

R2 value = 0.8131668567657471

1f) Using the model from that best epoch, plot the predicted values and target values for a batch of 128 test samples. (Hint: Use a callback to restore the best model weights. Find out how to retrieve a batch from tf.BatchDataset. A scatter plot will suffice.)

Callback to store best model weights:

```
#Callback to restore best model weights for qns 1f.
checkpoint_filepath = './best_weights_file'
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_filepath,
    save_weights_only=False,
    monitor='val_loss',
    mode='auto',
    save_best_only=True)
```

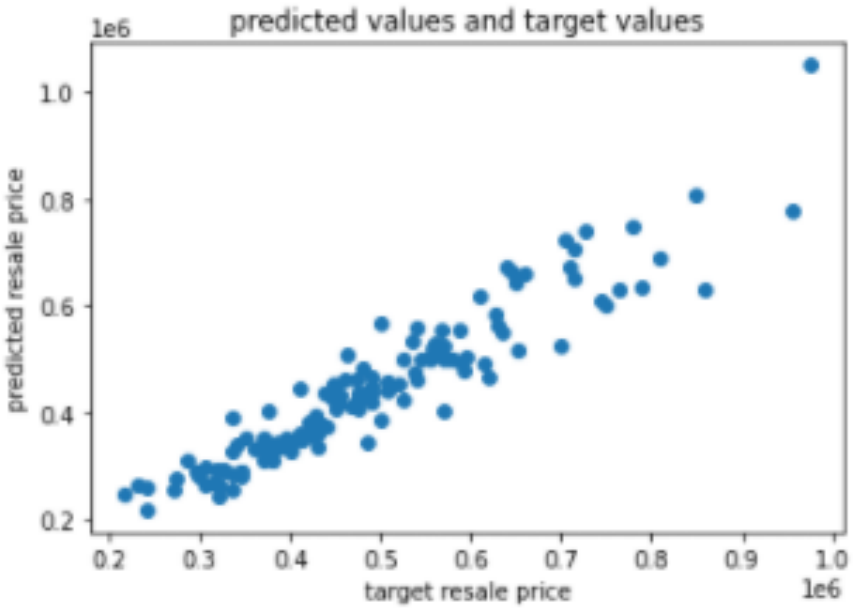


Figure 12: Predicted values against target values

2a) Add an Embedding layer with output_dim = floor(num_categories/2) after the one-hot embeddings for categorical variables. (Hint: Use the tf.keras.layers.Embedding() later. Read the documentation carefully to ensure that you define the correct function parameters4 .)

Code to implement lookup function:

```
#embedded categorical features instead of one-hot encoding
def embed_categorical_feature(feature, name, dataset, is_string):
    lookup_class = StringLookup if is_string else IntegerLookup
    # Create a lookup layer which will turn strings into integer indices
    lookup = lookup_class(output_mode="int")

    # Prepare a Dataset that only yields our feature
    feature_ds = dataset.map(lambda x, y: x[name])
    feature_ds = feature_ds.map(lambda x: tf.expand_dims(x, -1))

    # Learn the set of possible string values and assign them a fixed integer index
    lookup.adapt(feature_ds)

    # Turn the string input into integer indices
    encoded_feature = lookup(feature)
    return encoded_feature
```

```
[11] #embed categorical inputs
month_embed = embed_categorical_feature(month, "month", train_ds, False)

flat_model_type_embed = embed_categorical_feature(flat_model_type, "flat_model_type", train_ds, True)

storey_range_embed = embed_categorical_feature(storey_range, "storey_range", train_ds, True)

print(flat_model_type_embed)
```

Code to add embedding layer:

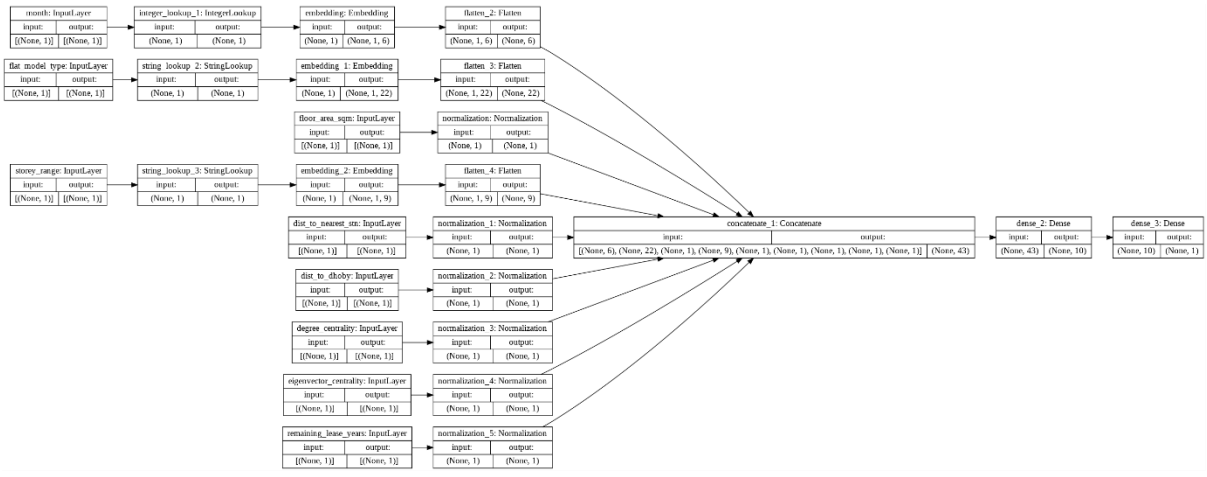
```
# Categorical features embedded
month_embedded = layers.Embedding(1, math.floor(13/2))(month_embed)
flat_model_type_embedded = layers.Embedding(1, math.floor(44/2))(flat_model_type_embed)
storey_range_embedded = layers.Embedding(1,math.floor(18/2))(storey_range_embed)
```

2b) The Embedding layer produces a 2D output (3D, including batch), which cannot be concatenated with the other features. Look through the Keras layers API to determine which layer to add in, such that all the features can be concatenated. Train the model using the same configuration as Q1.

Using flatten function for concatenation:

```
# flatten 3d layes
month_embedded_flatten = layers.Flatten()(month_embedded)
flat_model_type_flatten = layers.Flatten()(flat_model_type_embedded)
storey_range_flatten = layers.Flatten()(storey_range_embedded)

all_features2 = layers.concatenate(
    [
        month_embedded_flatten,
        flat_model_type_flatten,
        floor_area_sqm_encoded,
        storey_range_flatten,
        dist_to_nearest_stn_encoded,
        dist_to_dhoby_encoded,
        degree_centrality_encoded,
        eigenvector_centrality_encoded,
        remaining_lease_years_encoded,

    ]
)
```

Architecture of model with embedding layer:



2c) Compare the current model performances in terms of both test RMSE and test R 2 with the model from Q1 (at their own best epochs) and suggest a possible reason for the difference in performance.

Results of current model:
lowest test error = 66350.97177886697
epoch number = 57
R2 value = 0.8314036130905151

The model from question 2 has a lower test error and a higher R2 value, which is better. Embedding helps to encode more meaningful relationships among the categorise, which allows better accuracy in predicting as compared to just cramming all information of a category into a number.

3a) Continue with the model architecture you have after Q2. Via a callback, introduce early stopping (based on val_loss, with patience of 10 epochs) to the model.

Code to introduce early stopping:

```
#Defining early stopping
def early_stop_callback(name):
  return [tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=10)]
```

3b) Start by removing one input feature whose removal leads to the minimum drop (or maximum improvement) in performance5 . Repeat the procedure recursively on the reduced input set until the optimal number of input features is reached6 . Remember to remove features one at a time. Record the RMSE of each experiment neatly in a table

Removing first feature:

	month	flat_model_type	floor_area_sqm	storey_range	dist_to_nearest_stn	dist_to_dhoby	degree_centrality	eigenvector_centrality	remaining_lease_years
RMSE	72136.017301	80252.621938	73829.208312	73255.007092	75722.572381	88468.666182	69148.660725	66281.637155	73539.011253

From the table, removing the eigenvector_centrality results in lower RMSE value, thus is chosen for elimination.

Removing second feature:

	month	flat_model_type	floor_area_sqm	storey_range	dist_to_nearest_stn	dist_to_dhoby	degree_centrality	remaining_lease_years
RMSE	63122.325939	79419.213393	65637.292814	69513.885117	75135.090284	92852.525825	64749.505265	71949.262123

From the table, removing the month will result in lower RMSE value, thus chosen for elimination.

Removing third feature:

	flat_model_type	floor_area_sqm	storey_range	dist_to_nearest_stn	dist_to_dhoby	degree_centrality	remaining_lease_years
RMSE	79528.212051	65999.648969	72613.136883	75303.513437	94466.61243	63544.384237	69634.937438

From the table, removing any of the feature will have RMSE higher than the previous experiment, and thus no feature will be eliminated.

Therefore, only eigenvector_centrality and the month are removed during RFE.

3c) Compare the performances of the model with all 9 input features (from Q2) and the best model arrived at by RFE, in terms of both RMSE and R2 .

Results of the best model from RFE
lowest test error = 57597.74440028012
epoch number = 39
R_square value = 0.8729548454284668

The model arrived at by RFE has a much lower test error and higher R2 value as compared to the model from question 2. Overall the model has a better result.

3d) By examining the changes in model performance whenever a feature is removed, evaluate the usefulness of each feature for the task of HDB resale price prediction.

From the table which testes the removal of each of 9 features, the highest RMSE value means the feature is the most useful for prediction and vice versa. Comparing the RMSE value, the most useful feature is dist_to_dhoby, followed by flat_model_type, dist_to_nearest_stn, floor_area_sqm, remaining_lease_years, storey_range, degree_centrality, month and lastly is least useful is eigenvector_centrality.

Conclusion

RFE can be used to figure out whether each feature is important for prediction. We can also classify into categories in each feature that results in a higher resale price.