Table of Contents

Overview for DL Assignment 2 Part 1

Base Model (Word Embedding Model)

The initial issue was figuring out how data was parsed into the model as word embedding is used to convert each of the words into numbers to be used for the problem type. Necessary parameters were changed such that the output result of the model can predict 5 outputs as requested by the problem instead of only 2 outputs.

The next issue was finding out why this model had to use sparse categorical crossentropy was used instead of the usual categorical crossentropy, due to an issue with the loss function not being able to accept the latter. The main difference why this model is unable to use categorical crossentropy is due to the targets not being one hot encoded. Categorical crossentropy is only able to be used as a loss function if the targets are one-hot encoded. But since this model targets are using word embedding, sparse categorical crossentropy was used as the loss function instead.

My objective for this Base Model was to obtain a baseline model, where I can confidently use to progressively build my future models on top of it and improve the codes of the model as more fine-tuning techniques are used. This would ensure that my model is always progressing and is not using an already failed model unless there is a reason such as a decrease in validation accuracy due to certain hyperparameters being over-fine-tuned. The approach used was to attempt to code as simply as I can using the knowledge from my lectures and tutorials while still attaining a certain validation accuracy that is not too low to begin with.

The table below shows the key values recorded of the Base Model.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.4151	1st Epoch, Figure 0.1
Initial validation accuracy	0.4264	1st Epoch, Figure 0.1
Final training accuracy	0.5206	5th Epoch, Figure 0.2
Final validation accuracy	0.5144	5th Epoch, Figure 0.2
Initial training loss	1.4764	1st Epoch, Figure 0.1
Initial validation loss	1.4133	1st Epoch, Figure 0.1
Final training loss	1.1680	5th Epoch, Figure 0.2
Final validation loss	1.2042	5th Epoch, Figure 0.2
Testing Accuracy	**0.5031**	**-**

Base Model overfitted very early on at the 5th epoch onwards and both training and validation accuracies continued to increase steadily. This shows that the model is unpolished and requires fine-tuning to increase the accuracy.

Model 1 (Word Embedding Model)

Fine-tuning was used in Model 1 as the validation accuracy of Base Model is not satisfactory. The objective was to increase the overall accuracy to as well as testing accuracy to 51% and above after fine-tuning the model. This would justify my reasoning and methods to changing the model hyperparameters. The approach was to make calculated changes to the certain hyperparameters until the model obtained a favorable accuracy.

These values listed in the table below are the key data obtained from Model 1.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.3895	1st Epoch, Figure 1.1
Initial validation accuracy	0.4276	1st Epoch, Figure 1.1
Final training accuracy	0.5308	5th Epoch, Figure 1.2
Final validation accuracy	0.5150	5th Epoch, Figure 1.2
Initial training loss	1.4892	1st Epoch, Figure 1.1
Initial validation loss	1.1660	1st Epoch, Figure 1.1
Final training loss	1.1567	5th Epoch, Figure 1.2
Final validation loss	1.1660	5th Epoch, Figure 1.2
Testing Accuracy	**0.5051**	**-**

Overfitting on Model 1 occurs on the 2nd epoch although several methods have been deployed to obtain a higher accuracy while delaying overfitting, indicating that this model could be better fine-tuned to obtain a higher validation and testing accuracy.

Model 2 (Word Embedding Model)

The results for Model 1 were difficult to top as I had to keep thinking of hyperparameters to tune and justify for the actions. My objective for Model 2 is to delay overfitting, while increasing the validation and testing accuracy. The approach to this is to carefully tune the hyperparameters such as maximum length of words and other variables.

The table listed below contains the key data given by Model 2.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.4182	1st Epoch, Figure 2.1
Initial validation accuracy	0.4266	1st Epoch, Figure 2.1
Final training accuracy	0.5492	4th Epoch, Figure 2.2
Final validation accuracy	0.5339	4th Epoch, Figure 2.2
Initial training loss	1.4234	1st Epoch, Figure 2.1
Initial validation loss	1.3803	1st Epoch, Figure 2.1
Final training loss	1.1284	4th Epoch, Figure 2.2
Final validation loss	1.1461	4th Epoch, Figure 2.2
Testing Accuracy	**0.5367**	**-**

Model 2 overfitted on the 4th epoch in this model, compared to 5th epoch on both Base Model and Model 1. This model has the highest testing accuracy compared to 50.31% and 50.51% of Base Model and Model 1, respectively.

LSTM Model 1 (RNN Model – LSTM)

This is the base model of the RNN model using LSTM. No issues faced while loading the dataset and building the model. My objective here is to ensure that the validation accuracy

and testing accuracy of the model is comparable to the models using word embedding. I would expect to see an accuracy of 60% and above from this model. The approach to this model is like the Base Model. The only difference was some of the hyperparameters were changed and the build of the model is based on LSTM and not Word Embedding.

These are some of the key data from the model after testing.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.4934	1st Epoch, Figure 3.1
Initial validation accuracy	0.5467	1st Epoch, Figure 3.1
Final training accuracy	0.6242	2nd Epoch, Figure 3.2
Final validation accuracy	0.5949	2nd Epoch, Figure 3.2
Initial training loss	1.2143	1st Epoch, Figure 3.1
Initial validation loss	1.0741	1st Epoch, Figure 3.1
Final training loss	0.9376	2nd Epoch, Figure 3.2
Final validation loss	0.9850	2nd Epoch, Figure 3.2
Testing Accuracy	**0.5930**	**-**

This LSTM Model 1 overfits on the second epoch, which shows that it requires a lot of fine-tuning. This model has a testing accuracy almost hit over 60%, which is more than I was aiming for.

LSTM Model 2 (RNN Model – LSTM)

This is the fine-tuned RNN LSTM model. There were no issues with regularizing the model, other than the fact that it is rather hard to increase the validation and testing accuracy of this model. The objective of this model is to at least have a higher accuracy compared to LSTM Model 1. My approach to this was to place dropout layers and some recurrent dropout layers to this model.

These are the key values obtained from the model.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.4662	1st Epoch, Figure 4.1
Initial validation accuracy	0.5207	1st Epoch, Figure 4.1
Final training accuracy	0.6101	3rd Epoch, Figure 4.2
Final validation accuracy	0.5792	3rd Epoch, Figure 4.2
Initial training loss	1.2636	1st Epoch, Figure 4.1
Initial validation loss	1.1116	1st Epoch, Figure 4.1
Final training loss	0.9405	3rd Epoch, Figure 4.2
Final validation loss	0.9943	3rd Epoch, Figure 4.2
Testing Accuracy	**0.5767**	**-**

This LSTM model overfitted on the third epoch onwards, and a considerable number of fine-tuning was done to this model. This fine-tuned model has obtained a testing accuracy lower at 57.67%, than 59.30% of LSTM Model 1. It is unfortunate that this model was not able to achieve a greater testing accuracy than LSTM Model 1.

GRU Model 1 (RNN Model – GRU)

This is the first GRU Model that is built. There were no issues in the building process of the model as the dataset code can be used from the previous models. My objective for this model is to have an accuracy of 59% and above to remain competitive with the LSTM models. The approach to this model is like the LSTM models and Base Model. The differences are the hyperparameters for the model build is changed from LSTM to GRU.

These are the key data obtained from the GRU Model.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.5013	1st Epoch, Figure 5.1
Initial validation accuracy	0.5388	1st Epoch, Figure 5.1
Final training accuracy	0.6448	2nd Epoch, Figure 5.2
Final validation accuracy	0.6031	2nd Epoch, Figure 5.2
Initial training loss	1.1850	1st Epoch, Figure 5.1
Initial validation loss	1.0874	1st Epoch, Figure 5.1
Final training loss	0.8999	2nd Epoch, Figure 5.2
Final validation loss	0.9821	2nd Epoch, Figure 5.2
Testing Accuracy	**0.5963**	**-**

The base GRU model has already overfitted on the second epoch. This indicates that it can be quite thoroughly fine-tuned by changing multiple parameters. The testing accuracy is greater than 59%, which satisfies the objective I have set for this model.

GRU Model 2(RNN Model – GRU)

Although the testing accuracy for the base GRU model was quite decent, more things can be changed such that the validation and testing accuracy can be improved. My objective is to have the accuracy significantly improved from the base model. The approach for this model is like the base GRU model, but to fine tune and regularize the model accordingly.

The key obtained from this model are indicated in the table below.

Recorded data type	Value	Epoch (Page 20)
Initial training accuracy	0.4794	01st Epoch, Figure 6.1
Initial validation accuracy	0.5247	01st Epoch, Figure 6.1
Final training accuracy	0.6162	2nd Epoch, Figure 6.2
Final validation accuracy	0.5767	2nd Epoch, Figure 6.2
Initial training loss	1.2097	1st Epoch, Figure 6.1
Initial validation loss	1.0935	1st Epoch, Figure 6.1
Final training loss	0.9232	2nd Epoch, Figure 6.2
Final validation loss	0.9945	2nd Epoch, Figure 6.2
Testing Accuracy	**0.5737**	**-**

This GRU model overfitted on the third epoch despite rather extensive fine-tuning. However, after changing the hyperparameters, the testing accuracy is 57.37% compared to 59.63% of the base GRU model. This is a 2.26% decrease in the testing accuracy of the model between the fine-tuned and non-fine-tuned models.

Data Loading and Processing for DL Assignment 2 Part 1

1. Data Loading

```
# Load the emoji_dictionary
import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤️'}
A total of:  5 Emoji Icons
```

This section of code is used to load the emoji dictionary into the Jupyter notebook. This is done by loading the excel csv file “mapping.csv” into the notebook and allowing it to be saved as “df” after removing the commas. The emojis are then extracted and loaded into “emoji_dictionary” as an array. It is then printed and shown in the output below, with each emoji corresponding to each position it is placed at in the array.

```
# Load the dataset
dat = pd.read_csv('dataset.csv', delimiter=',')
texts = dat.loc[:,'TEXT'].values
labels = dat.loc[:,'Label'].values
```

This section of code allowed the notebook to read the excel csv file “dataset.csv” saves it as “dat”. The next line proceeds to save the texts of the dataset as “texts”. It also saves the labels of each corresponding text as “labels”.

```
# Check the maximum length of texts
max_len = -1
for example in texts:
    if len(example.split()) > max_len:
        max_len = len(example.split())

print('the maximum length of the text inputs is ', max_len)
```

```
the maximum length of the text inputs is  34
```

The section of code shown here displays the longest line of character available in the text of the dataset. As shown by the output, the longest text within the entire dataset is 34 characters long.

2. Data Processing

```
# Convert the texts and labels into numeric tensors
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np
from tensorflow.keras.layers import Embedding

embedding_layer = Embedding(20000, 25)
maxlen = 30
training_samples = 40000
validation_samples = 6500
max_words = 20000

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('Shape of data tensor:', data.shape)
print('Shape of label tensor:', labels.shape)
```

```
Found 54072 unique tokens.
Shape of data tensor: (42546, 30)
Shape of label tensor: (42546,)
```

The data processing code is next to be run in the notebook. It stores important values such as the “embedding_layer” and “maxlen”, which will be critical to the build and how the model accuracy will eventually turn out. This base model specifically has 25 dimensionalities, 30 characters set as the max length of words, and a maximum number of 20,000 words from the top will be used for the training and validation of this model.

```
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

A tokenizer is also in place to tokenize the texts, by fitting the words into the tokenizer to find out how many words in total. There is also a section in the 4th paragraph of the image above, finding out how many special characters are available within the maximum number of words that will be used to train the model.

```
data = pad_sequences(sequences, maxlen=maxlen)
```

The next paragraph shows the maximum number of words being padded such that there will be a consistent number of data and labels. This is done to ensure that the number of sequences and labels match and are not different.

In the following table (Table 2.1) which shows the difference in hyperparameters between different models. The most hyperparameter I tuned the most was the maximum number of words used (max_words), maximum length of each sequence (maxlen) and the number of words that will be used for training the model (training_samples).

The reason why I tuned the maximum number of words was simply because I believe that the more words the model is exposed to, the more it will increase the variety of words and the different emotions and emojis attached to each of the sentences. This has a chance of increasing the accuracy of the model as shown in the models later.

The maximum length of each sequence was also tuned as the model will be able to interpret the sentences differently if I used a maximum length of 15 and 30 characters. One example to display my point is this, “This movie is really funny with the choices of actors starred in this great movie. I really loved the love between characters as it was heartfelt and would love to watch this lovely show again.”. This sentence is not part of the dataset but is used to showcase this example of how the number of words used to train can determine the emoji that is predicted by the model. In this case, the first sentence would be predicted by the model as a laughing emoji. But when the model has an increased maximum length and looks at the second sentence, the emoji predicted would be most likely by the one at index zero, which is the heart from the eyes. Without the increased length, each sequence would only predict simple sentences and are unable to predict more complex ones. Hence, more words used to train one sequence would increase the accuracy of the texts and would increase the accuracy of the model.

Lastly, increasing the training sample increases number of texts to be trained by the model, evidently increasing the validation, testing, and prediction accuracy of the model.

Table 2.1

Model Name	Data Processing Hyperparameters

Base Model	`embedding_layer = Embedding(20000, 25)`
	`maxlen = 30`
	`training_samples = 40000`
	`validation_samples = 6500`
	`max_words = 20000`

Model 1	`embedding_layer = Embedding(40000, 100)`
	`maxlen = 15`
	`training_samples = 80000`
	`validation_samples = 12500`
	`max_words = 40000`

Model 2	`embedding_layer = Embedding(420000, 150)`
	`maxlen = 30`
	`training_samples = 500000`
	`validation_samples = 40000`
	`max_words = 4200000`

LSTM Model 1	`embedding_layer = Embedding(420000, 100)`
	`maxlen = 30`
	`training_samples = 500000`
	`validation_samples = 40000`
	`max_words = 420000`

LSTM Model 2	`embedding_layer = Embedding(420000, 100)`
	`maxlen = 30`
	`training_samples = 500000`
	`validation_samples = 40000`
	`max_words = 420000`

GRU Model 1	`embedding_layer = Embedding(420000, 100)`
	`maxlen = 30`
	`training_samples = 500000`
	`validation_samples = 40000`
	`max_words = 420000`

GRU Model 2	`embedding_layer = Embedding(420000, 100)`
	`maxlen = 30`
	`training_samples = 500000`
	`validation_samples = 40000`
	`max_words = 420000`

3. Data Sampling

```
# Split the X & y into train and test sets
from tensorflow.keras import preprocessing

X = data
y = labels

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 32)
# Refer the report Appendix
# Please enter the random_state assigned to your group

X_train = preprocessing.sequence.pad_sequences(X_train, maxlen=maxlen, truncating ='pre')
X_test = preprocessing.sequence.pad_sequences(X_test, maxlen=maxlen, truncating ='pre')

print(X_train[0])
print(len(X_train[0]))
```

```
[    0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     3     3 16587    34     2     7   363    12     4
    63  4383   166 16588   622 16589]
30
```

The data is split into X and Y, which is the data and labels, respectively. X and Y will then be split into training and testing data. The test size is 20% of the total training data. A random state will also be included as instructed by the assignment.

Afterwards, the training and testing data will be preprocessed using padding and truncation. Padding is added when the number of words in a sequence is less than the words required as defined by “maxlen”. Truncation is added only when the number of words exceed the “maxlen”, thus words will be dropped starting from the front until it reaches the specified number of words defined by “maxlen”.

Develop the Sentimental Analysis Models using Training Data for DL Assignment 2 Part 1

1. Base Model (Word Embedding)

a. Building the Base Model

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense

base_model = Sequential()
base_model.add(Embedding(training_samples, 25, input_length=maxlen))
base_model.add(Flatten())
base_model.add(Dense(5, activation='softmax'))

base_model.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
base_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 25)            1000000
_________________________________________________________________
flatten (Flatten)            (None, 750)               0
_________________________________________________________________
dense (Dense)                (None, 5)                 3755
=================================================================
Total params: 1,003,755
Trainable params: 1,003,755
Non-trainable params: 0
```

The base model build only has one embedding input layer, which has 25 dimensionalities and uses the maximum length as the input length. This input length is the maximum words from the top that is used to train and validate the model. A flatten layer is used to flatten the shape to be readily used for the output layer, which strictly requires 5 outputs.

Over a million trainable parameters were trained. This is considerably more than those in the practical, as this is testing thousands and thousands of words compared to the practical. There are no non-trainable parameters as this is not using a pretrained model.

b. Training the Base Model

```
# Train the Model
history = base_model.fit(X_train, y_train,
                         epochs=5,
                         batch_size=1028,
                         validation_split=0.2)
```

The ".fit" function allows the notebook to start training the models according to the specifications listed in the bracket as shown. It is training for the text and labels, only runs for 20 epochs and each batch size is 1028.

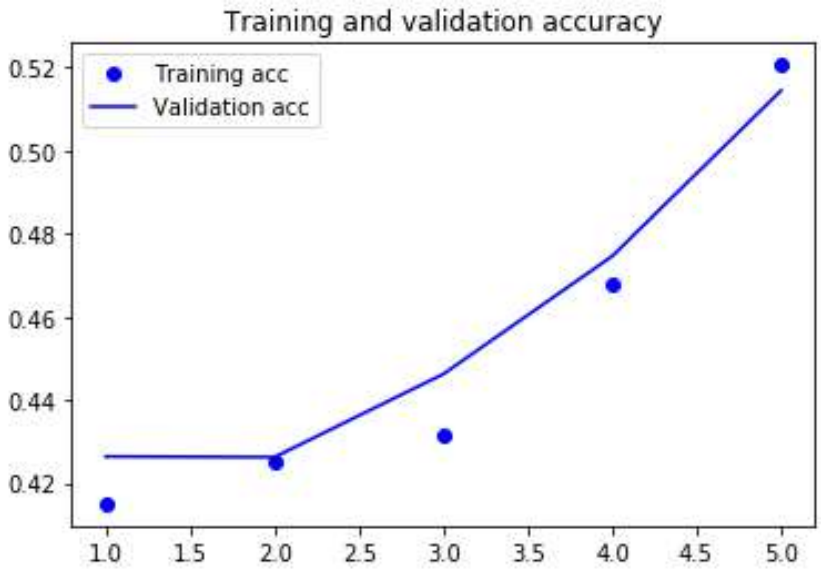
Batch size refers to the training samples to work through before the internal parameters of the models are updated. This means that 1028 training samples are passed before the internal parameters of the base model is updated.

A validation split of 20% is used to allow the model to test itself against data that is not trained by the model yet. This validation outcome, in the form of validation accuracy and validation loss, will be a rough estimate of the prediction accuracy of the model to the testing data.

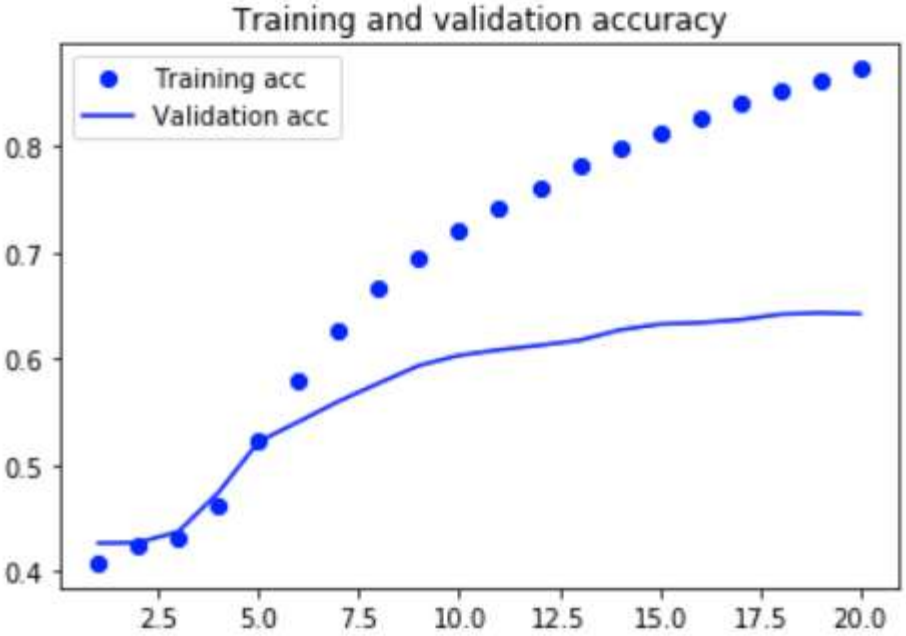
c. Plotting the graph of Base Model

Graph of Training & Validation Accuracy for Base Model

Final Graph

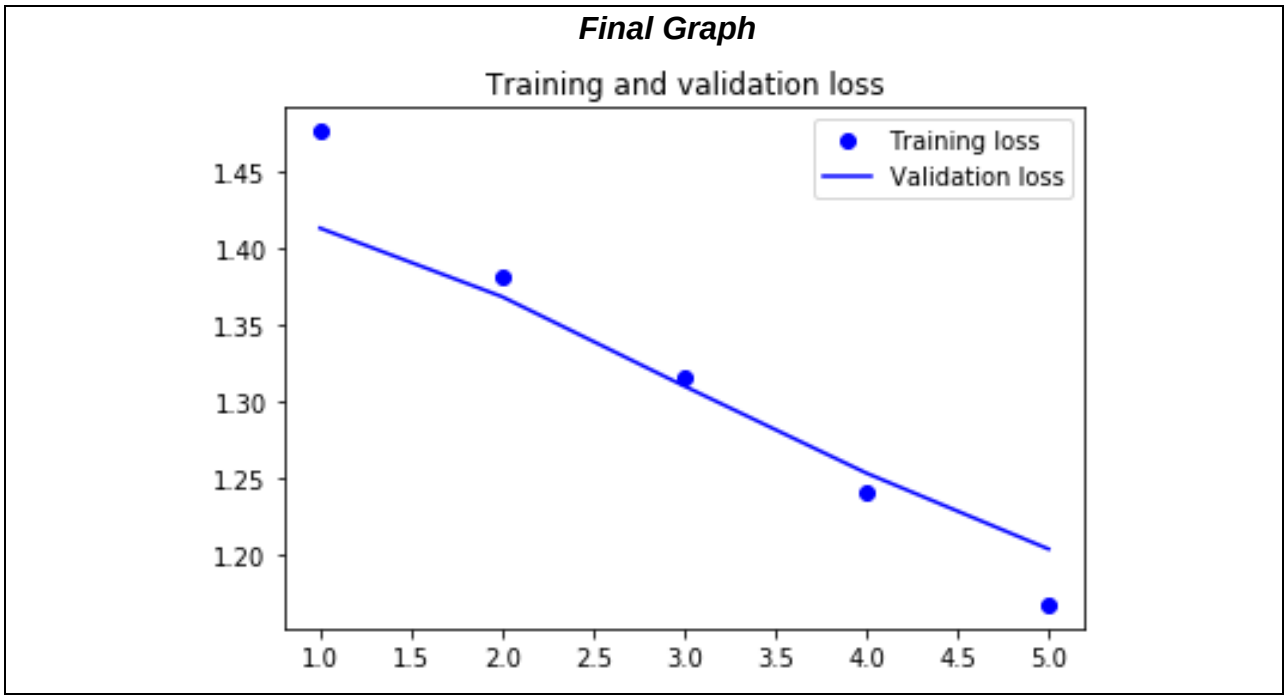


Extended Graph (Only for references, not the final graph)

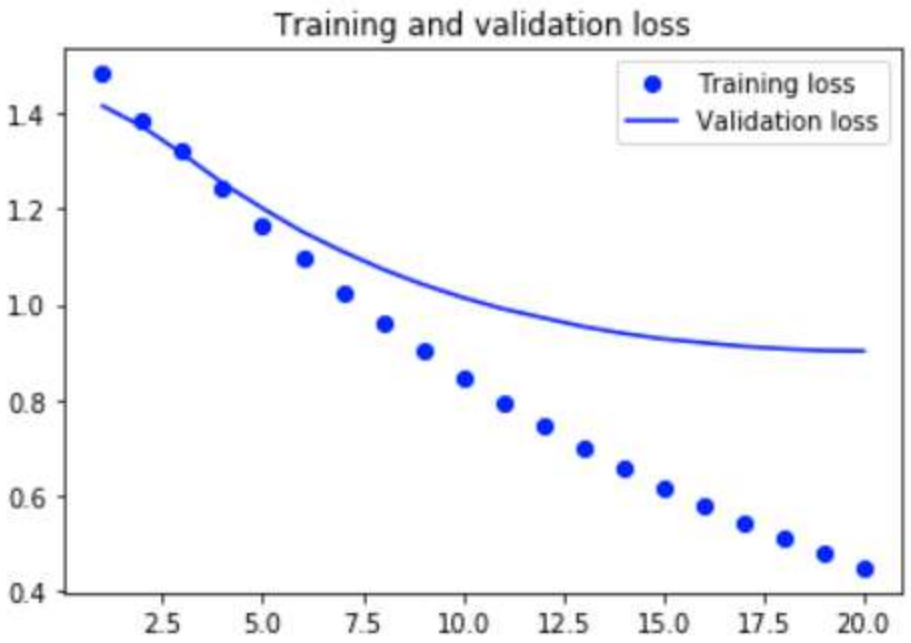


This graph of Base Model shows the training and validation accuracy. It shows that the model is overfitted from the 5th epoch onwards. The initial training and validation accuracy for the model is 41.51% and 42.64% respectively (Figure 0.1). There is a sharp increase in both accuracies on the third epoch towards the 5th epoch. The validation accuracy was initially higher but was soon lesser than training accuracy. Both graphs look stable and does not seem to have any issues since there are no sudden spikes and drop in accuracies as seen in the extended graph. The highest validation obtained was 51.44% recorded on the 5th epoch (Figure 0.2), while highest training accuracy was 52.06% (Figure 0.2) on the final epoch. This model shows that there is still much room available to train and regularize the model using methods such as dropout.

Graph of Training & Validation Loss for Base Model



Extended Graph (Only for references, not the final graph)



This is the training and validation loss graph. As seen in the graph, the initial training loss was higher at 1.4764 compared to 1.4133 of validation loss (Figure 0.1). At the third epoch, the validation loss was not decreasing at the same rate as training loss, which was going in a rather straight line. The final training loss reached a loss score of 1.1680 on the 5th epoch (Figure 0.2). The validation loss rate of loss slowed down around the 3rd epoch and the loss rate continued to decrease as it continues to train, reaching a final validation loss of 1.2042 on the 5th epoch (Figure 0.2).

From the extended accuracy and loss graphs, I can predict that the model is too used with the training data and can easily predict the sentiment, which is a sign of overfitting. Thus, perhaps a dropout layer could reduce the model from being able to predict the sentiment that easily.

```
Epoch 1/5
27228/27228 [==============================] - 1s 32us/sample - loss: 1.4764 - acc: 0.4151 - val_loss: 1.4133 - val_acc: 0.4264
```

Figure 0.1

```
Epoch 5/5
27228/27228 [==============================] - 0s 10us/sample - loss: 1.1680 - acc: 0.5206 - val_loss: 1.2042 - val_acc: 0.5144
```

Figure 0.2

2. Model 1 (Word Embedding)

a. Building Model 1

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras import layers

model1 = Sequential()
model1.add(Embedding(training_samples, 100, input_length=maxlen))
model1.add(layers.Dropout(0.6))
model1.add(Flatten())
model1.add(layers.Dropout(0.5))
model1.add(Dense(5, activation='softmax'))

model1.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
model1.summary()
```

```
WARNING:tensorflow:Large dropout rate: 0.6 (>0.5). In TensorFlow 2.x, dropout() uses dropout r
s intended.
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 15, 100)           8000000
_________________________________________________________________
dropout (Dropout)            (None, 15, 100)           0
_________________________________________________________________
flatten (Flatten)            (None, 1500)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 1500)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 7505
=================================================================
Total params: 8,007,505
Trainable params: 8,007,505
Non-trainable params: 0
```

The difference between the build of Base Model and Model 1 is the increased number of dimensionalities from 25 to 100. and the dropout layers which are included in this model.

b. Training Model 1

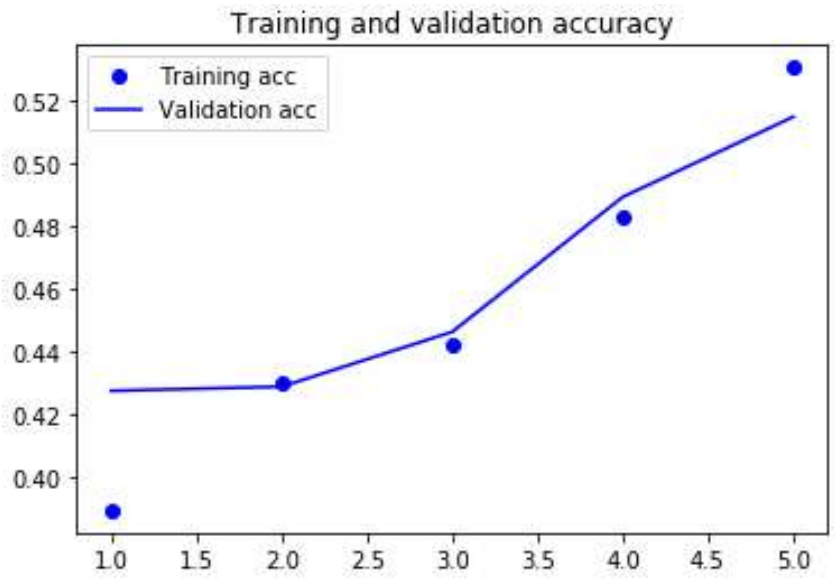
```
# Train the Model
history = model1.fit(X_train, y_train,
                     epochs=5,
                     batch_size=1028,
                     validation_split=0.2)
```

There are no differences between the training code between Base Model and Model 1.

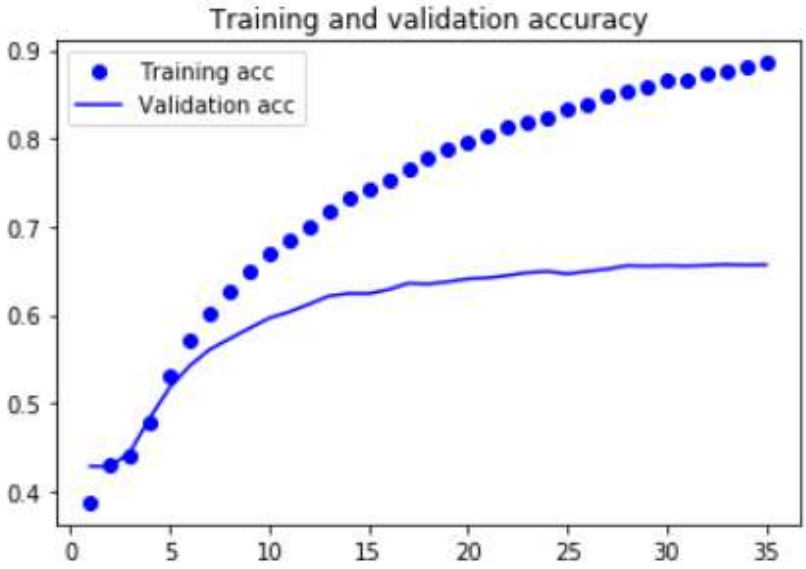
c. Plotting the graph of Model 1

Graph of Training & Validation Accuracy for Model 1

Final Graph



Extended Graph (Only for references, not the final graph)

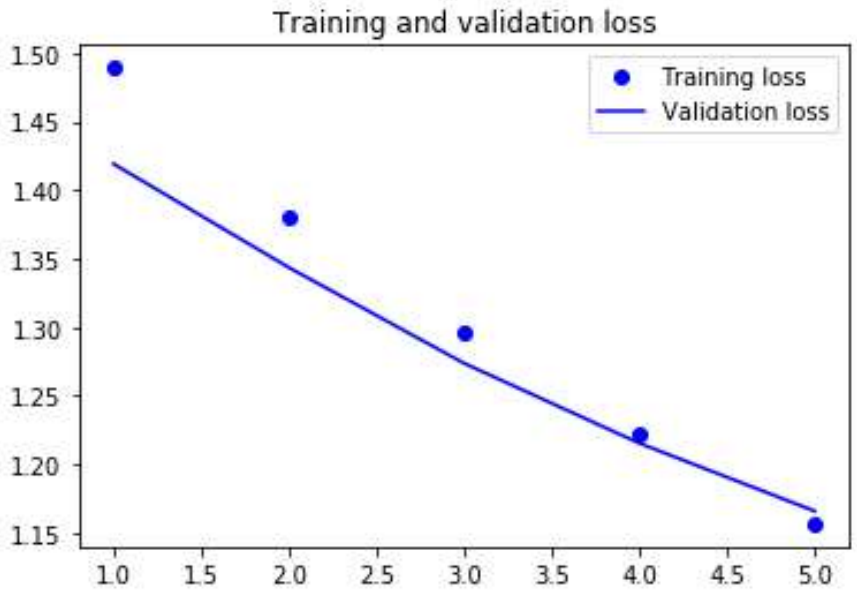


Model 1 has a similar looking graph with Base Model, due to the model build being so similar. The initial training and validation accuracy are 38.95% and 42.76% respectively (Figure 1.1). It then shows the accuracy suddenly increasing steeply on the third epoch onwards. This model overfitted on the 3rd epoch, like Base Model.

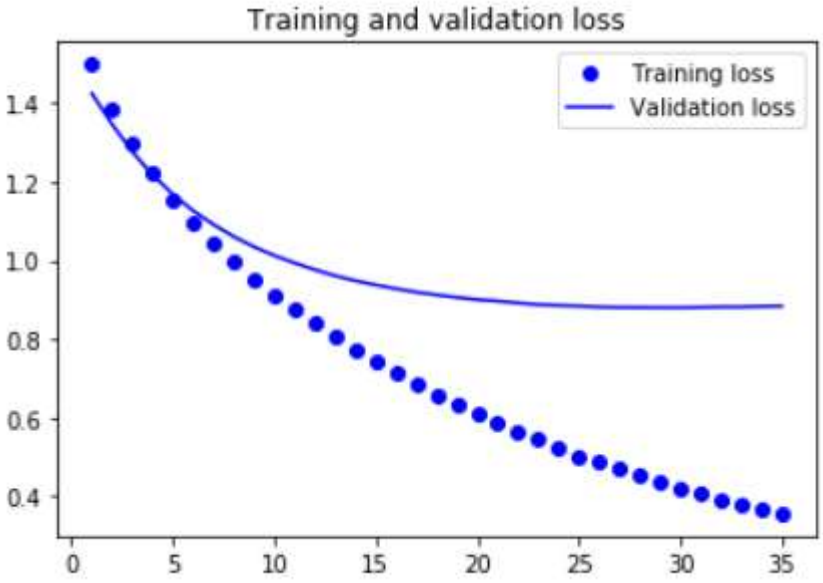
Both accuracies show signs of stability and there are no spikes and drops in accuracies. The validation model also flattens after around 10 epochs as seen on the extended graph. I think that this something that happens inevitably as no matter how many drop out layers was used, the training accuracy continued to skyrocket way past above the validation accuracy. The highest training accuracy achieved was 53.08% on the final epoch (Figure 1.2), while highest validation accuracy achieved was 51.50% on the 5th epoch (Figure 1.2). At this point, I do not see overfitting as a concern, since I am only concerned about the validation accuracy and loss of the model, which might give me a rough idea of what to expect if I use the model for prediction.

Graph of Training & Validation Loss for Model 1

Final Graph



Extended Graph (Only for references, not the final graph)



This loss graph starts off with validation loss lower than training loss, at 1.4243 and 1.4989 (Figure 1.1), respectively. However, after the first few epochs, the training loss continued to decrease at a rather fixed rate while validation loss quickly flattened after the 15th epoch as seen on the extended graph.

This resulted in the lowest validation loss of only 1.1660 (Figure 1.2) while training loss reached 1.1567 (Figure 1.2) at its lowest. Although the validation accuracy may have increased from Base Model, it was only a 0.0093 difference improvement in validation accuracy from Base Model. This shows that the model might not have had the proper number of epochs trained yet as the values are still increasing. However, increasing the number of epochs will increase the training accuracy drastically. This, for the next model. The number of epochs will remain the same as to not increase the training accuracy further.

Figure 1.1

Figure 1.2

3. Model 2 (Word Embedding)

a. Building Model 2

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras import layers

model2 = Sequential()
model2.add(Embedding(training_samples, 150, input_length=maxlen))
model2.add(layers.Dropout(0.5))
model2.add(Flatten())
model2.add(layers.Dropout(0.5))
model2.add(Dense(5, activation='softmax'))

model2.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
model2.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 150)           75000000
_________________________________________________________________
dropout (Dropout)            (None, 30, 150)           0
_________________________________________________________________
flatten (Flatten)            (None, 4500)              0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4500)              0
_________________________________________________________________
dense (Dense)                (None, 5)                 22505
=================================================================
Total params: 75,022,505
Trainable params: 75,022,505
Non-trainable params: 0
```

Model 2 build is also no different from Model 1, with the only difference being the dimensionality. I wanted to test out if changing only the dimensionality would increase the overall accuracy. This model has been tested multiple times to find out that if the dimensionality is set too big (e.g. 250), the model will have too much dimensionality and the model will run slowly with an accuracy as high as a model using 100 as the dimensionality. Base Model has 1,003,755 trainable parameters, Model 1 has 8,007,505 while Model 2 has a total of 75,002,505 inputs.

By increasing the dimensionality, I am indirectly increasing the input of the model. There are no non-trainable parameters for Base Model, Model 1, and Model 2 because it is not from a pretrained model.

b. Training Model 2

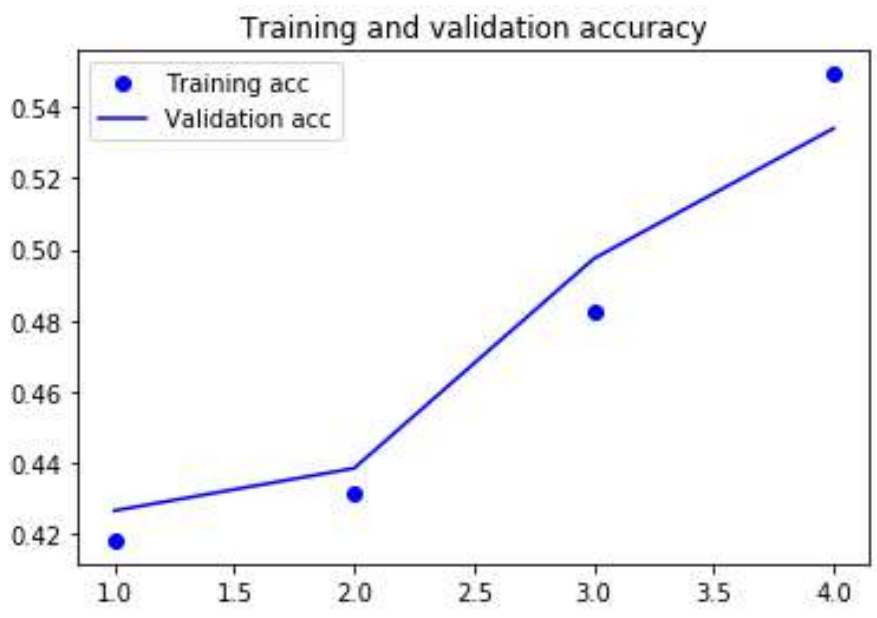
```
# Train the Model
history = model2.fit(X_train, y_train,
                     epochs=4,
                     batch_size=1028,
                     validation_split=0.2)
```

As written above, the number of epochs remained the same. Thus, no difference between the training code of Model 1 and Model 2.

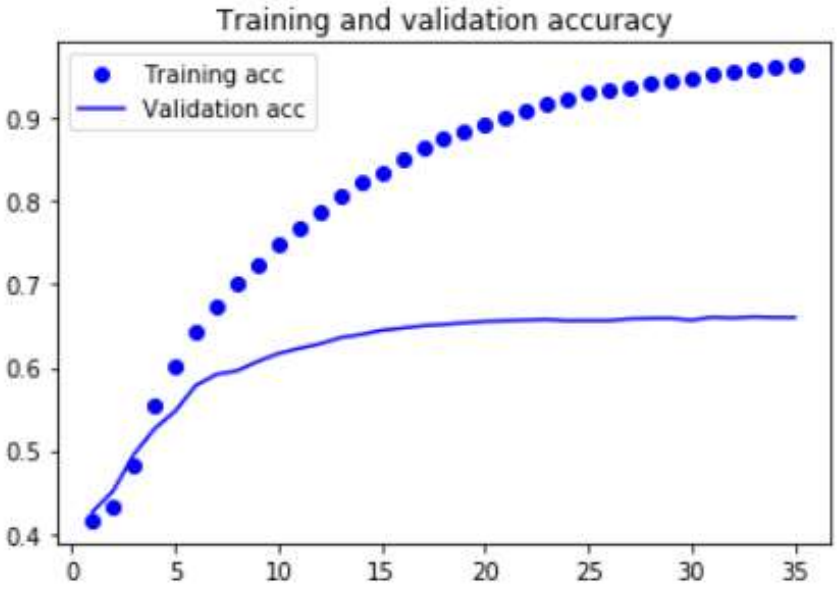
c. Plotting the graph of Model 2

Graph of Training & Validation Accuracy for Model 2

Final Graph



Extended Graph (Only for references, not the final graph)

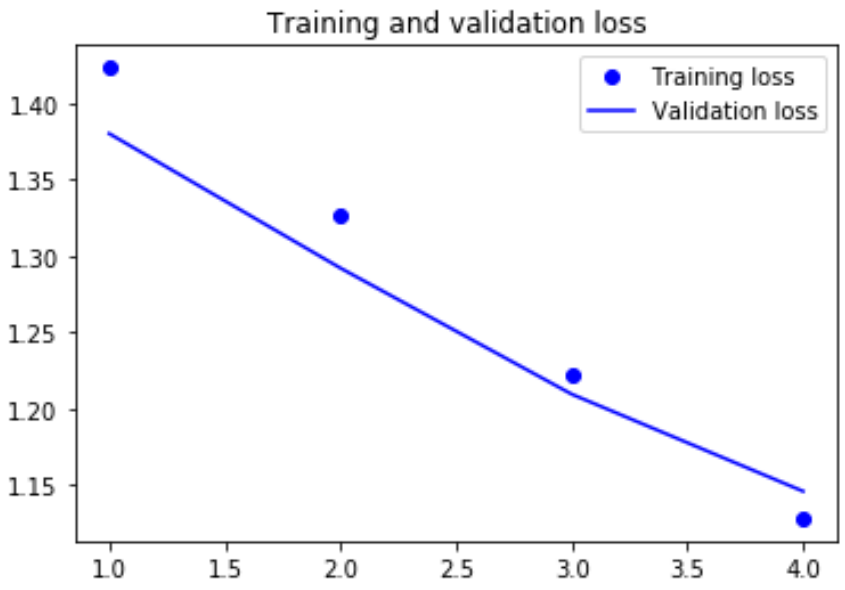


The model looks similar to Base Model and Model 1, however, there is a slight spike in validation accuracy towards the beginning of model training as shown in the extended graph. This is shown by the small bump on the extended graph. The model overfitted on the 4th epoch as shown in the final graph above. The initial training and validation accuracy are 41.62% and 42.64% (Figure 2.1), respectively. In this extended graph, there is no sudden increase in both graphs as seen on the extended graph on the third epoch onwards unlike Base Model and Model 1.

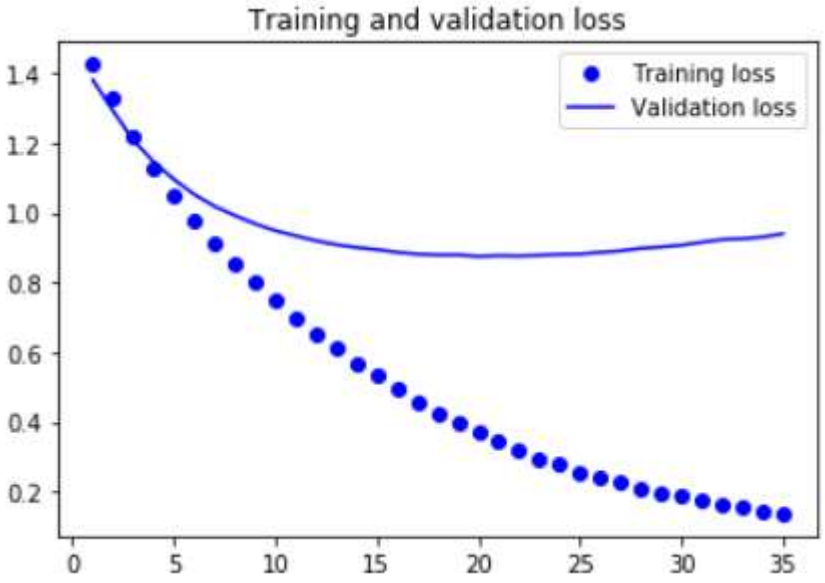
There is only one slight increase in the accuracy as seen in the extended graph. There is no indication of instability of the model. The validation accuracy seems to be flatter each time, as Model 2 highest recorded validation accuracy is 53.39% (Figure 2.2), compared to 51.50% accuracy of Model 1 (Figure 1.2). The training and validation accuracy constantly increases, eventually reaching a final accuracy of 54.92% and 53.39% respectively (Figure 2.2). Despite fine-tuning and changing certain hyperparameters, there is not much improvement in the overall accuracy of the Word Embedding models.

Graph of Training & Validation Loss for Model 2

Final Graph



Extended Graph (Only for references, not the final graph)



The extended graph here looks different as compared to the ones seen in Base Model and Model 1. This is all due to the dimensionality of the model. The more the dimensionality of the model, the training loss graph would be less straight.

This is because the more dimensionality, the more input data it is receiving. It causes the training loss graph to not decrease at a consistent rate, but at a decelerated rate.

The validation loss graph decreases at a decreasing rate, but eventually increasing the loss after a certain number of epochs. This means that the model can predict with a good accuracy due to higher accuracy compared to Base Model and Model 1, but Model 2 is able to predict with lesser confidence due to the increased loss score. The initial training and validation loss are 1.4234 and 1.3803, respectively (Figure 2.1). And the lowest recorded training and validation loss scores are 1.1284 and 1.1461 (Figure 2.2). Model 2 final validation loss score is lower than Model 1 and Base Model, as Model 1 achieved a loss score of 1.1660 (Figure 1.2) and Base Model was 1.2042 (Figure 0.2), while Model 2 was 1.1461 (Figure 2.2).

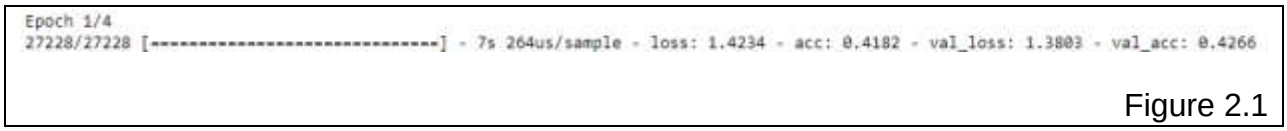


Figure 2.1

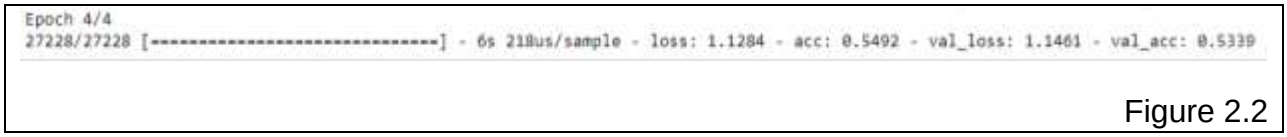


Figure 2.2

4. LSTM Model 1 (RNN Model – LSTM)

a. Building LSTM Model 1

```
# Build the Model
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.layers import Embedding, SimpleRNN
from tensorflow.keras import layers
from tensorflow.keras.layers import LSTM

LSTM_Model1 = Sequential()
LSTM_Model1.add(Embedding(max_words, 100, input_length=maxlen))
LSTM_Model1.add(LSTM(32, return_sequences=True))
LSTM_Model1.add(LSTM(16))
LSTM_Model1.add(Dense(5, activation='softmax'))

LSTM_Model1.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
LSTM_Model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 100)           42000000
_________________________________________________________________
lstm (LSTM)                  (None, 30, 32)            17024
_________________________________________________________________
lstm_1 (LSTM)                (None, 16)                3136
_________________________________________________________________
dense (Dense)                (None, 5)                 85
=================================================================
Total params: 42,020,245
Trainable params: 42,020,245
Non-trainable params: 0
```

This LSTM Model 1 build has one embedding layer and 2 LSTM layers. The dimensionality was set to 100 as shown in the table of codes a few pages before. The total trainable parameters, which is the same as trainable parameters, is 42,020,245. This is much more significant compared to Word Embedding Base Model.

b. Training LSTM Model 1

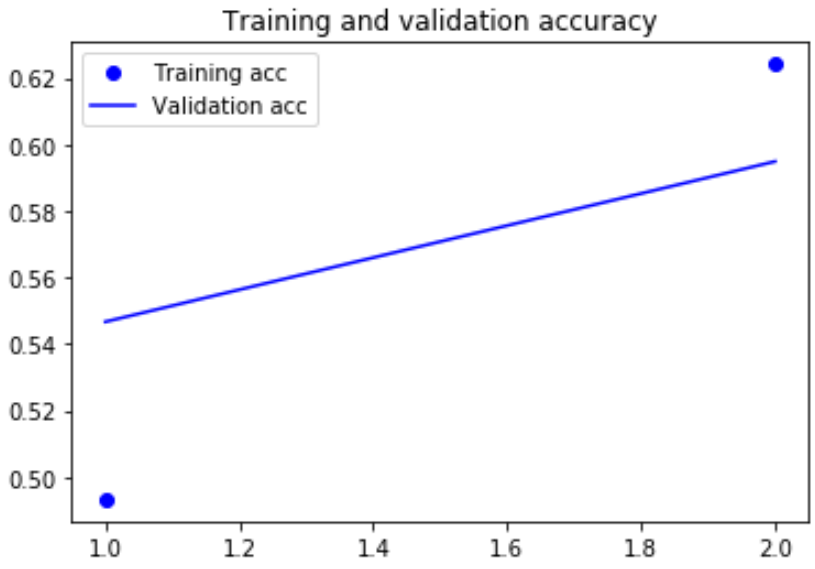
```
# Train the Model
history = LSTM_Model1.fit(X_train, y_train,
                          epochs=2,
                          batch_size=128,
                          validation_split=0.2)
```

The code above shows the training portion of the code. It has the same parameters as the word embedding models as I did not find a need to change the batch size or epochs since it will be overfitted within the first few epochs.

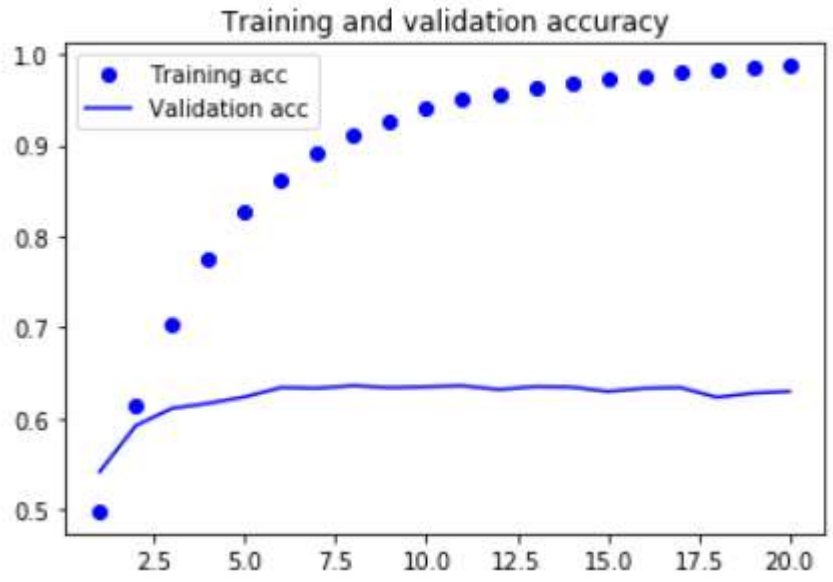
c. Plotting the graph of LSTM Model 1

Graph of Training & Validation Accuracy for LSTM Model 1

Final Graph



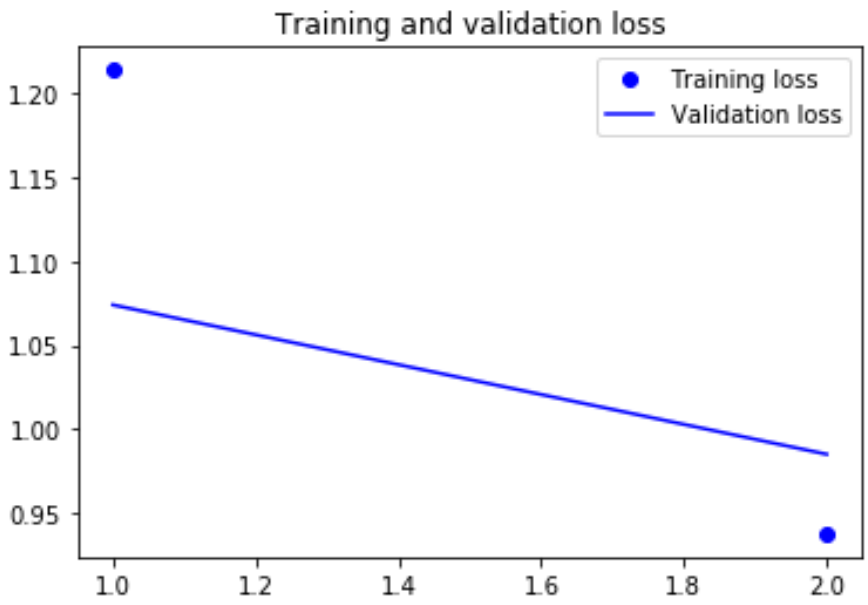
Extended Graph (Only for references, not the final graph)



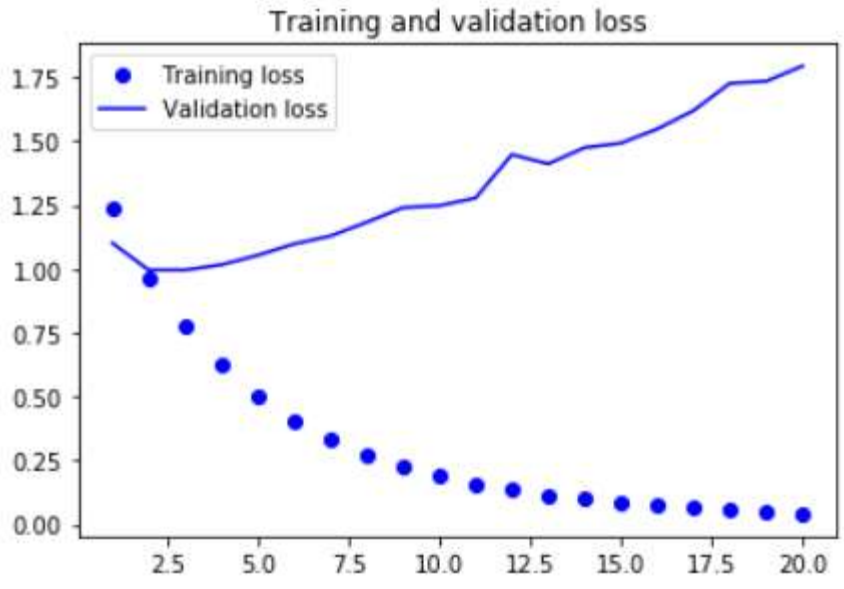
This is the training and validation accuracy of the LSTM Model 1. This model has an initial training and validation accuracy at 49.34% and 54.67% respectively (Figure 3.1). The graph did not have a sudden increase in accuracy on the 3rd epoch just like the word embedding ones as seen on the extended graph. It has a steep accuracy from the start. The model was already overfitted on the 2nd epoch. The training graph showed a stable increase until it almost reached a 100% accuracy. While validation graph was not as stable, the highest validation accuracy was 59.49% (Figure 3.2). The final training and validation accuracy are 62.42% and 59.49% (Figure 3.2). The validation accuracy was quite good already considering this is a model that does not have any fine-tuning in place, but has certain parameters changed as I would like to use the progress made in previous models to improve the future models made.

Graph of Training & Validation Loss for LSTM Model 1

Final Graph



Extended Graph (Only for references, not the final graph)



This shows the training and validation loss of LSTM Model 1. The training loss started at 1.2143 while validation loss started at 1.0741 (Figure 3.1). Training loss decreased stably in a decreasing rate as seen on the extended graph. The validation loss also decreased for one epoch then began to increase. There is a spike in validation loss around the 12th epoch as seen in the extended graph above. The final training and validation are 0.9376 and 0.9850, respectively (Figure 3.2).

With the increase in validation loss seen in the extended graph, this shows that the model is not confident in predicting the sentiment but is very well at obtaining high accuracy when predicting. This model requires more fine-tuning such as dropout to hopefully reduce the overfitting, which might make the model more stable.

```
Epoch 1/2
27228/27228 [==============================] - 30s 1ms/sample - loss: 1.2143 - acc: 0.4934 - val_loss: 1.0741 - val_acc: 0.5467
```

Figure 3.1

```
Epoch 2/2
27228/27228 [==============================] - 26s 942us/sample - loss: 0.9376 - acc: 0.6242 - val_loss: 0.9850 - val_acc: 0.5949
```

Figure 3.2

5. LSTM Model 2 (RNN Model – LSTM)

a. Building LSTM Model 2

```
# Build the Model
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.layers import Embedding, SimpleRNN
from tensorflow.keras import layers
from tensorflow.keras.layers import LSTM

LSTM_Model2 = Sequential()
LSTM_Model2.add(Embedding(max_words, 100, input_length=maxlen))
LSTM_Model2.add(LSTM(128, dropout=0.5, recurrent_dropout=0.5, return_sequences=True))
LSTM_Model2.add(LSTM(64, dropout=0.5, recurrent_dropout=0.5, return_sequences=True))
LSTM_Model2.add(LSTM(32, dropout=0.4, recurrent_dropout=0.4))
LSTM_Model2.add(Dense(5, activation='softmax'))

LSTM_Model2.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
LSTM_Model2.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 100)           42000000
_________________________________________________________________
lstm (LSTM)                  (None, 30, 128)           117248
_________________________________________________________________
lstm_1 (LSTM)                (None, 30, 64)            49408
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                12416
_________________________________________________________________
dense (Dense)                (None, 5)                 165
=================================================================
Total params: 42,179,237
Trainable params: 42,179,237
Non-trainable params: 0
```

LSTM Model 2 saw an increase in LSTM layers, an inclusion of dropout layer and recurrent dropout layer. The dimensionality remains the same. The total parameter has increased from 42,020,245 in LSTM Model 1 to 42,179, 237 in LSTM Model 2.

b. Training LSTM Model 2

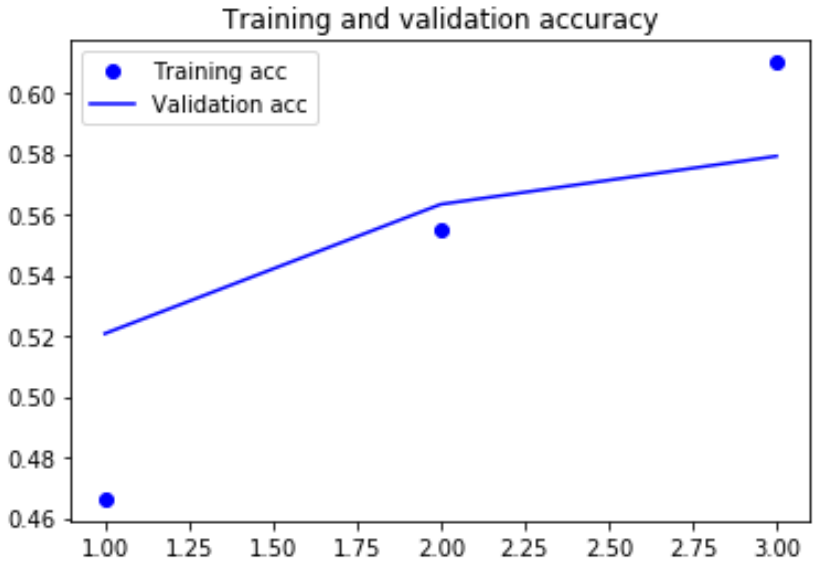
```
# Train the Model
history = LSTM_Model2.fit(X_train, y_train,
                          epochs=3,
                          batch_size=128,
                          validation_split=0.2)
```

This training code remains the same as LSTM Model 1, as there was no need to change or reduce the number of epochs as it will overfit very quickly anyways.

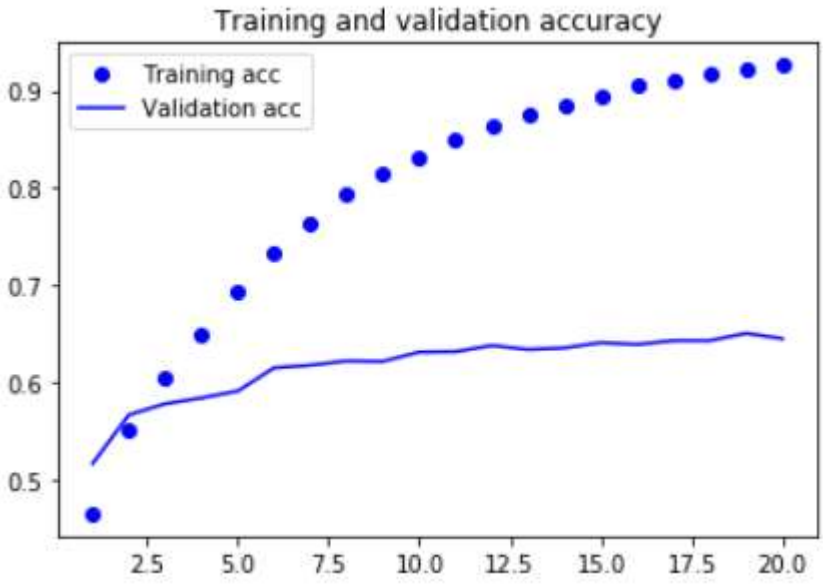
c. Plotting the graph of LSTM Model 2

Graph of Training & Validation Accuracy for LSTM Model 2

Final Graph



Extended Graph (Only for references, not the final graph)

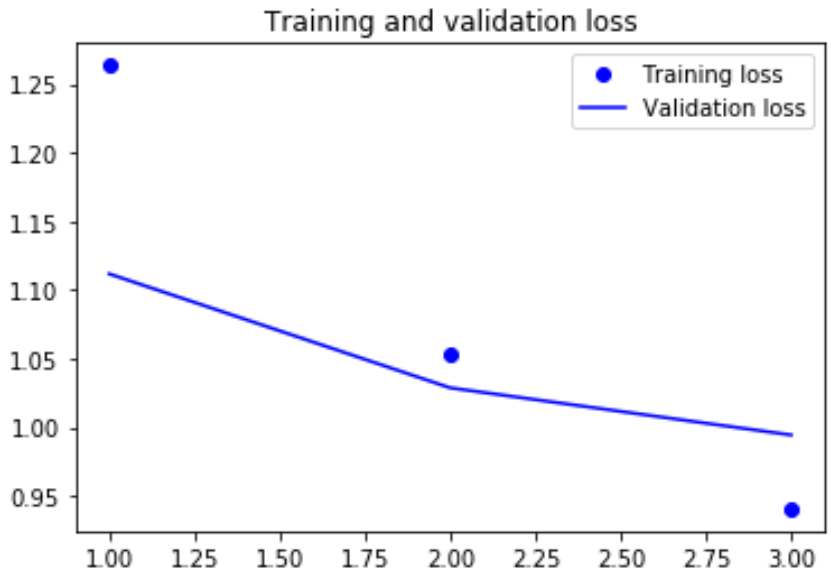


This graph shows that the model is quite stable overall, but the validation accuracy graph does show a slight instability due to the increase in accuracy not being that smooth as shown in the extended graph. The model overfitted on the 3rd epoch. The initial validation accuracy is 52.07% compared to 46.62% of training accuracy (Figure 4.1). The overall training graph seems to be increasing in accuracy at a decreasing rate as the graph increases very little in accuracy at the top seen from the extended graph.

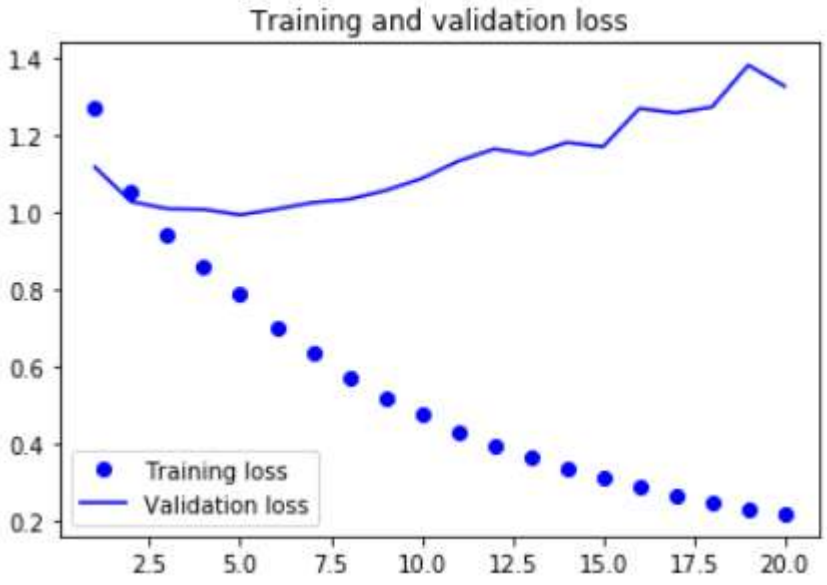
The validation graph saw a small spike in accuracy as seen in the extended graph above but maintained the accuracy and improved on it further. This led to the high validation accuracy of the model. The final training and validation accuracy of this LSTM Model 2 is 61.01% and 57.92% respectively (Figure 4.2). This model has reduced the rate at which it increases in training accuracy if the graph for LSTM Model 1 and LSTM Model 2 were compared, the difference between the rate of increase of accuracy would be higher on LSTM Model 1 than LSTM Model 2, hence showing that the dropout layers and recurrent dropout layers works.

Graph of Training & Validation Loss for LSTM Model 2

Final Graph



Extended Graph (Only for references, not the final graph)



The loss graph of LSTM Model 2 seems similar to LSTM Model 1, just that the validation loss was not as stable as LSTM Model 1 validation loss as seen in the extended graph. The initial training and validation loss were 1.2636 and 1.1116, respectively (Figure 4.1). The rate at which the training loss decreases slows down over time.

While validation loss decreased slightly for the first few epochs, then increase significantly and had a few small dips and spikes to the loss score as seen on the extended graph. This increase in validation loss score means the model will be less confident in predicting the sentiment, but due to the high accuracy, it should predict it at with high accuracy.

Comparing LSTM Model 1 and LSTM Model 2, LSTM Model 2 seems to be a worse fit for the best model between the two models. This is because LSTM Model 2 has a lower final validation accuracy of 57.92% (Figure 4.2) compared to 59.49% of LSTM Model 1 (Figure 3.2). The reason why the LSTM Model 2 has a lower validation accuracy might be due to the fine-tuning in place on LSTM Model 2.

Comparing LSTM Model 2 with Model 2, Model 2 has a lower final recorded validation accuracy at 53.39% (Figure 2.2) compared to 57.92% of LSTM Model 2 (Figure 4.2). Model 2 validation loss also does not increase drastically after decreasing. Hence, predicting the sentiment with greater confidence but less accurate compared to LSTM Model 2.

```
Epoch 1/3
27228/27228 [==============================] - 53s 2ms/sample - loss: 1.2636 - acc: 0.4662 - val_loss: 1.1116 - val_acc: 0.5207
```

Figure 4.1

```
Epoch 3/3
27228/27228 [==============================] - 46s 2ms/sample - loss: 0.9405 - acc: 0.6101 - val_loss: 0.9043 - val_acc: 0.5792
```

Figure 4.2

6. GRU Model 1 (RNN Model – GRU)

a. Building GRU Model 1

```
# Build the Model
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.layers import Embedding, SimpleRNN
from tensorflow.keras import layers
from tensorflow.keras.layers import GRU

GRU_Model1 = Sequential()
GRU_Model1.add(Embedding(max_words, 100, input_length=maxlen))
GRU_Model1.add(GRU(32, return_sequences=True))
GRU_Model1.add(GRU(16))
GRU_Model1.add(Dense(5, activation='softmax'))

GRU_Model1.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
GRU_Model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 100)           42000000
_________________________________________________________________
gru (GRU)                    (None, 30, 32)            12864
_________________________________________________________________
gru_1 (GRU)                  (None, 16)                2400
_________________________________________________________________
dense (Dense)                (None, 5)                 85
=================================================================
Total params: 42,015,349
Trainable params: 42,015,349
Non-trainable params: 0
```

This is GRU Model 1. The code for the built is identical to LSTM Model 1 except the LSTM from LSTM Model 1 has been changed to GRU since this is a GRU Model. This GRU should perform like LSTM Model 1 as no other parameters need to be changed.

b. Training GRU Model 1

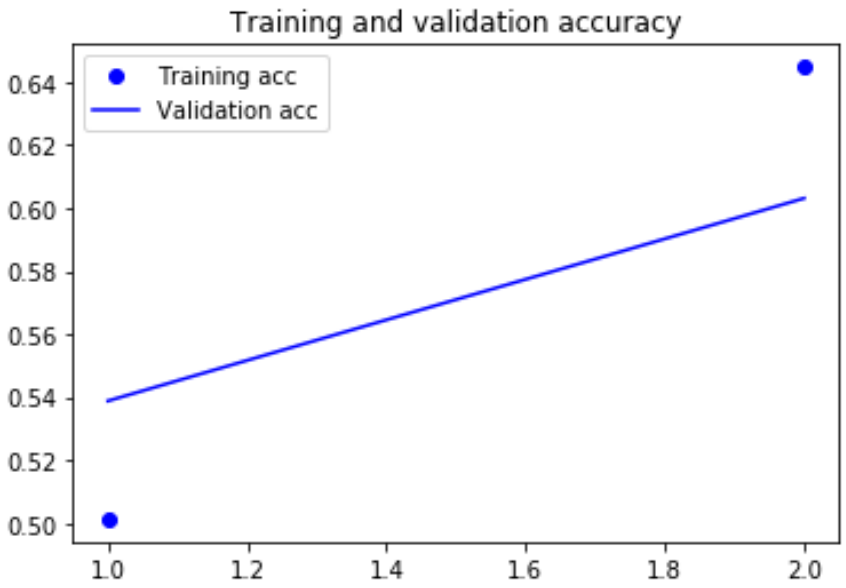
```
# Train the Model
history = GRU_Model1.fit(X_train, y_train,
                         epochs=2,
                         batch_size=128,
                         validation_split=0.2)
```

There is no change in the training code for GRU Model 1 as I would like to compare this model with LSTM Model 1. To ensure fair comparison, these hyperparameters were kept.

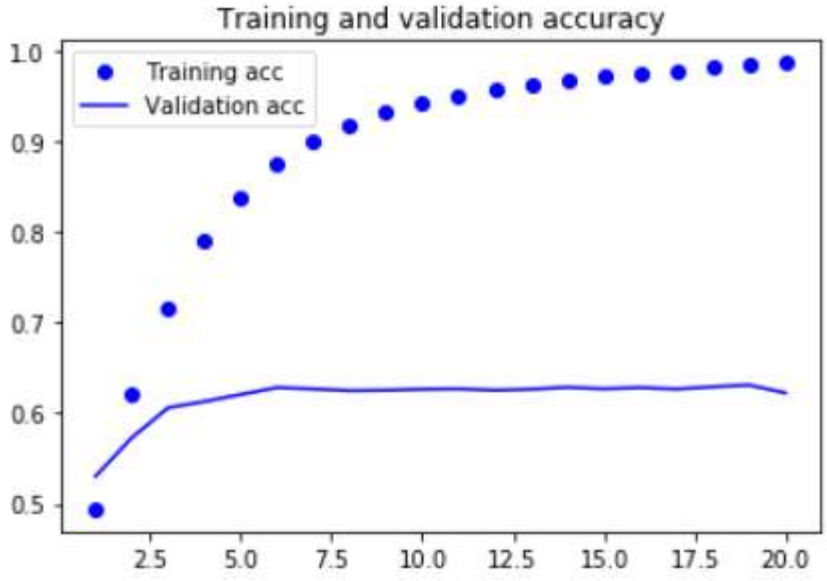
c. Plotting the graph of GRU Model 1

Graph of Training & Validation Accuracy for GRU Model 1

Final Graph



Extended Graph (Only for references, not the final graph)

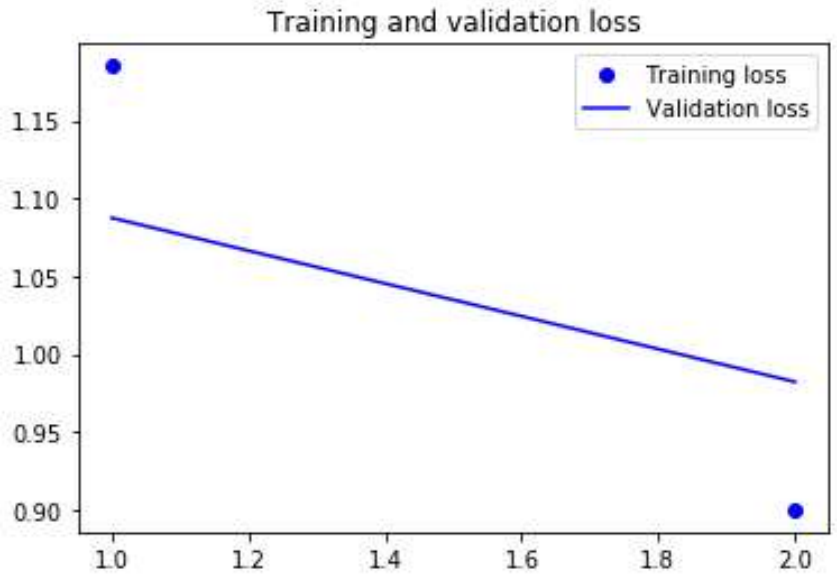


GRU Model 1 graph is really similar to LSTM Model 1. The rate of increase for the training accuracy is similar to the rate found in LSTM Model 1 training accuracy as seen from the extended graph of GRU Model 1. The initial training and validation accuracy were 50.13% and 53.88% respectively (Figure 5.1). This model overfitted on the 2nd epoch onwards. Showing that there is much fine-tuning to do for this model to obtain the best accuracy.

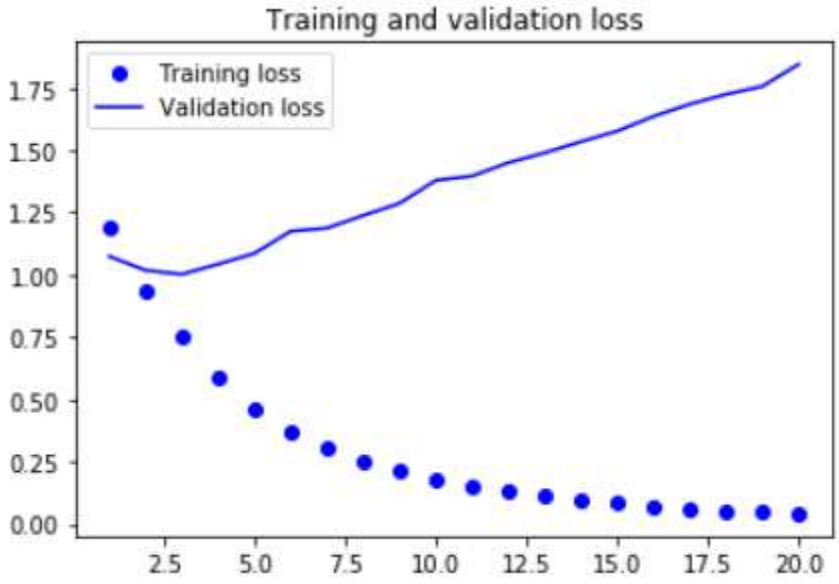
The validation accuracy does not show signs of instability, but the accuracy seems stagnant after the 6th epoch as seen on the extended graph. This might mean that this model has reached its limit for validation accuracy. The final training and validation accuracy are 64.48% and 60.31% (Figure 5.2). when comparing this to LSTM Model 1, LSTM Model 1 has a lower final validation accuracy recorded of 59.49% (Figure 3.2). Thus, some hyperparameters should be changed for the validation accuracy of this model to increase further.

Graph of Training & Validation Loss for GRU Model 1

Final Graph



Extended Graph (Only for references, not the final graph)



This loss graph is similar to LSTM Model 1. The training loss for GRU Model 1 decreases at a faster rate here compared to LSTM Model 1. The extended validation graph seems rather stable, other than the fact that there are two small spikes in validation loss. The initial training and validation loss were 1.1850 and 1.0874 respectively (Figure 5.1).

Comparing GRU Model 1 validation loss with LSTM Model 1 accuracy, the initial validation loss was lower on LSTM Model 1 at 1.0741 (Figure 3.1) while 1.0874 at GRU Model 1 (Figure 5.1). The final validation loss obtained was 0.9821 (Figure 5.2) for GRU Model 1 and 0.9850 on LSTM Model 1 (Figure 3.2) This shows that GRU Model 1 might be able to predict the sentiment more confidently than compared to LSTM Model 1 just from the values seen on the validation loss.

Thus, this model still requires quite a lot of fine-tuning like dropout for it to obtain better validation accuracies.

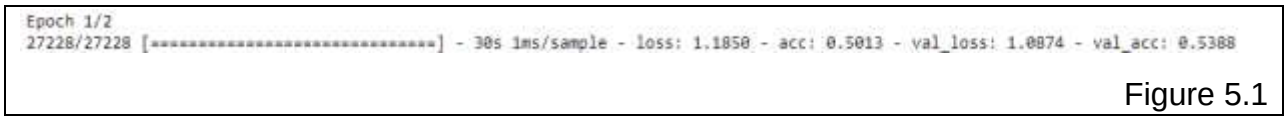


Figure 5.1

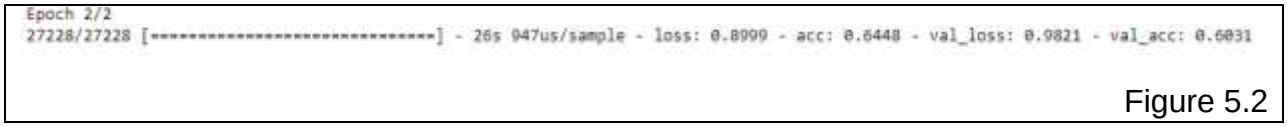


Figure 5.2

7. GRU Model 2 (RNN Model – GRU)

a. Building GRU Model 2

```
# Build the Model
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.layers import Embedding, SimpleRNN
from tensorflow.keras import layers
from tensorflow.keras.layers import GRU

GRU_Model2 = Sequential()
GRU_Model2.add(Embedding(max_words, 100, input_length=maxlen))
GRU_Model2.add(GRU(128, dropout=0.5, recurrent_dropout=0.5, return_sequences=True))
GRU_Model2.add(GRU(64, dropout=0.5, recurrent_dropout=0.5, return_sequences=True))
GRU_Model2.add(GRU(32, dropout=0.4, recurrent_dropout=0.4))
GRU_Model2.add(Dense(5, activation='softmax'))

GRU_Model2.compile(optimizer='rmsprop', loss='sparse_categorical_crossentropy', metrics=['acc'])
GRU_Model2.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 30, 100)           42000000
_________________________________________________________________
gru (GRU)                    (None, 30, 128)           88320
_________________________________________________________________
gru_1 (GRU)                  (None, 30, 64)            37248
_________________________________________________________________
gru_2 (GRU)                  (None, 32)                9408
_________________________________________________________________
dense (Dense)                (None, 5)                 165
=================================================================
Total params: 42,135,141
Trainable params: 42,135,141
Non-trainable params: 0
```

GRU Model 2 has more layers, with dropout and recurrent dropout which is used to help regularize the model and reduce the steepness of the training accuracy curve. The dimensionality remains the same as GRU Model 1.

b. Training GRU Model 2

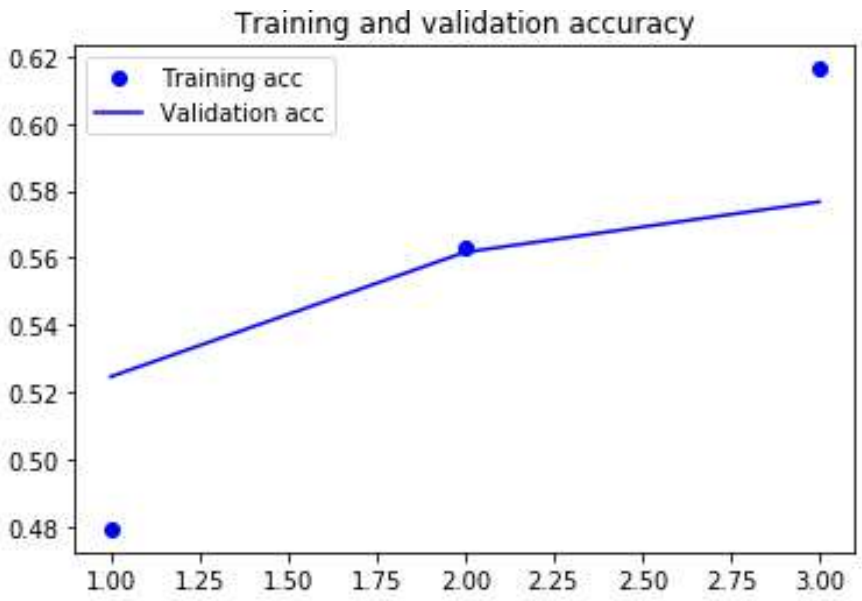
```
# Train the Model
history = GRU_Model2.fit(X_train, y_train,
                         epochs=3,
                         batch_size=128,
                         validation_split=0.2)
```

This is the same GRU Model 1 as I saw no reason to change the number of epochs as it overfitted very quickly. Batch size was not needed to be changed as the larger the batch size, the more likely it is to cause lower accuracy due to lesser internal weight updates as there are lesser batches due to the increased batch size.

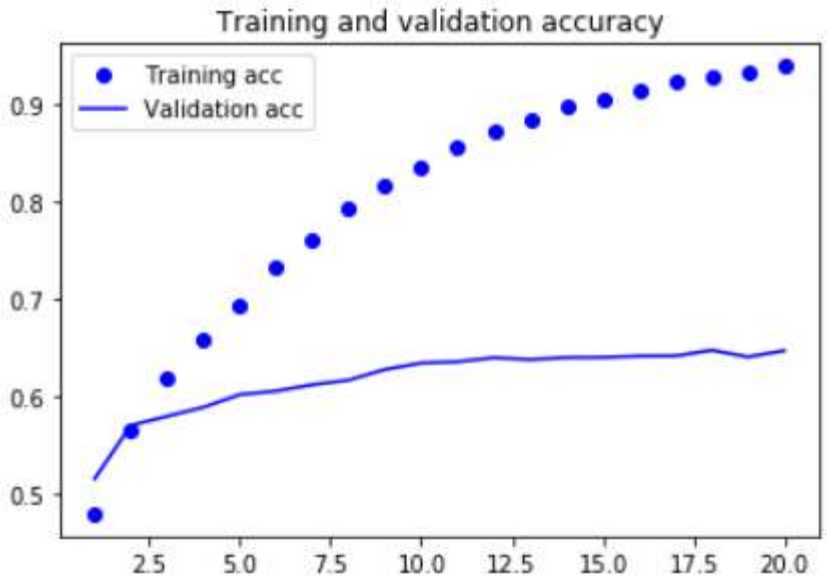
c. Plotting the graph of GRU Model 2

Graph of Training & Validation Accuracy for GRU Model 2

Final Graph



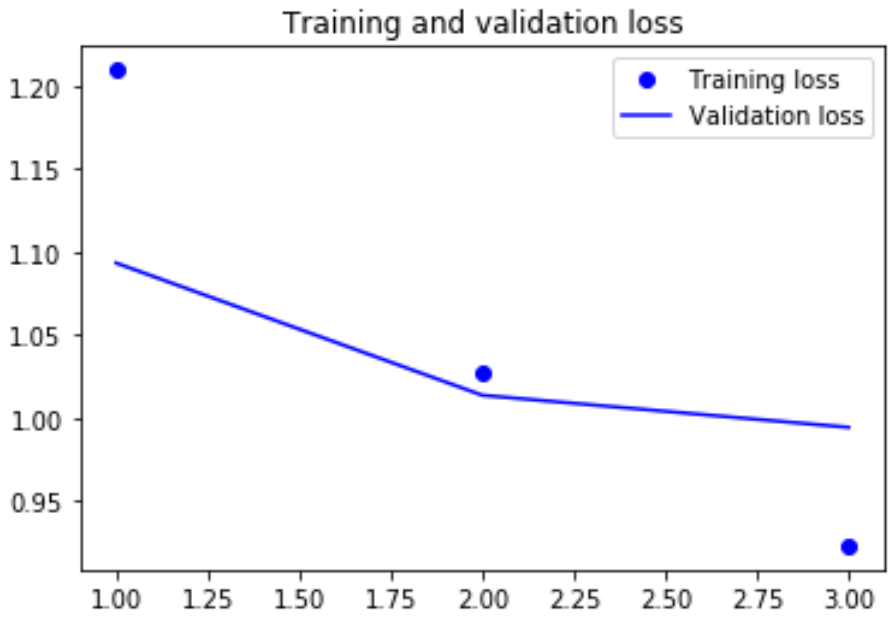
Extended Graph (Only for references, not the final graph)



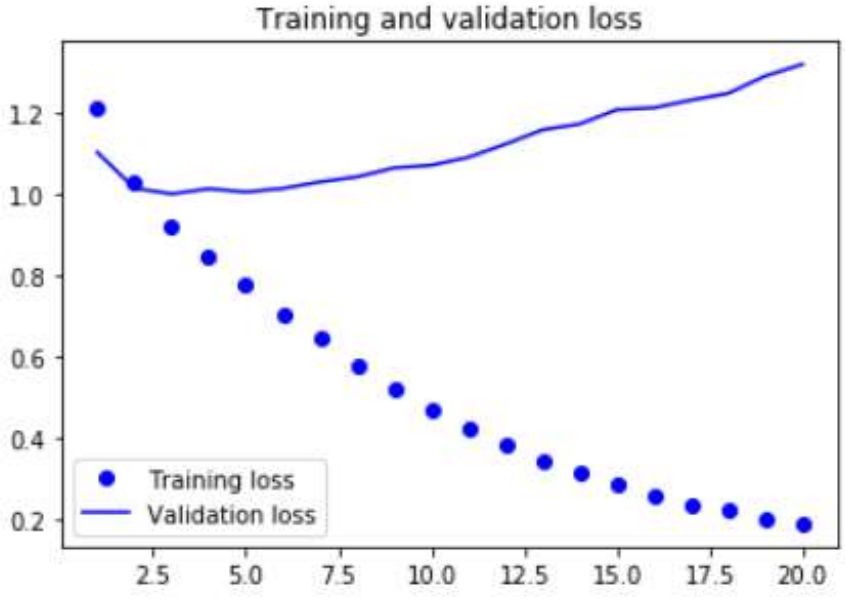
This extended graph shows that the training accuracy is steadily increasing at a decreasing rate. Due to the dropout layers and recurrent dropout layers involved in the build of the model, the curve was much less steep compared to the sharp increase then flatness of the training accuracy graph of GRU Model 1. The initial training and validation accuracy were 47.94% and 52.47% respectively (Figure 6.1). The validation accuracy starts off at a higher accuracy only to be overfitted on the 2nd epoch. The extended graph shows that the validation line graph is still increasing every slowing as shown in the extended graph. The final training and validation accuracy are 61.62% and 57.67% respectively (Figure 6.2).

Graph of Training & Validation Loss for GRU Model 2

Final Graph



Extended Graph (Only for references, not the final graph



The training loss graph seems less steep compared to LSTM Model 2 as seen from the extended graph. The validation graph is very stable compared to LSTM Model 2. There are no spikes or dips in validation loss as seen in the extended graph. The initial training and validation loss are 1.2097 and 1.0935 respectively (Figure 6.1). The validation loss increased for the first few epochs and proceeded to increase in a stable manner. It had a gentle increase and there were no spikes or dips as the validation loss increased. Hence, showing that this model is stable and has been fine-tuned.

Comparing with GRU Model 1, this training accuracy graph is flatter and less steep. The validation accuracy graph is also not flat, but constantly increasing but just a little. GRU Model 1 has a recorded final validation accuracy of 60.31% (Figure 5.2) while GRU Model 2 has it at 57.67% (Figure 6.2). This shows that there was no significant improvement in terms of accuracy of the model at predicting the sentiment.

Comparing this with LSTM Model 2, I think that GRU Model 2 has a less steep curve compared to LSTM Model 2. The difference is hardly noticeable. But as for validation accuracy, LSTM Model 1 had the final validation accuracy of 57.92% (Figure 3.2) compared to 57.67% (Figure 6.2) of GRU Model 2. This indicates that LSTM Model 2 might have a better prediction accuracy compared to GRU Model 2.

Comparing with Model 2, GRU Model 2 has a lower final validation accuracy of 57.67% (Figure 6.2) compared to 53.39% validation accuracy of Model 2 (Figure 2.2). This indicates that GRU Model 2 has higher validation accuracy and is most likely to better predict the accuracy of the sentiment.

```
Epoch 1/3
27228/27228 [==============================] - 60s 2ms/sample - loss: 1.2097 - acc: 0.4794 - val_loss: 1.0935 - val_acc: 0.5247
```

Figure 6.1

```
Epoch 3/3
27228/27228 [==============================] - 53s 2ms/sample - loss: 0.9232 - acc: 0.6162 - val_loss: 0.9945 - val_acc: 0.5767
```

Figure 6.2

Model Evaluation using Testing Data for DL Assignment 2 Part 1

1. Base Model (Word Embedding)

```
#Testing Base Model for the accuracy on Test Data
results = base_model.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 0s 52us/sample - loss: 1.2097 - acc: 0.5032
[1.2097038780900482, 0.50317276]
```

This is the code used to test Base Model 1. The same code will be used for every model as this test is ran immediately after the training and testing of data. This code runs the test data and labels to output a result, which is the percentage of the accuracy of the model at predicting the sentiment. The testing accuracy of Base Model is 50.31%, which is comparable to the final validation accuracy of 51.50% (Figure 0.2).

2. Model 1 (Word Embedding)

```
#Testing Base Model for the accuracy on Test Data
results = model1.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 0s 54us/sample - loss: 1.1749 - acc: 0.5052
[1.174852136915635, 0.5051704]
```

Model 1 has a testing accuracy slightly higher than Base Model, although the dimensionality of Model 1 is 100 and Base Model is 25. Other hyperparameters that might affect the testing accuracy is the maximum length, which has been decreased from 30 in Base Model to 15 in Model 1. Training sample changed from 40,000 in Base Model to 80,000 in Model 1, validation samples and the maximum number of words used.

These are all factors that should be taken into consideration, but not as much as the model build itself, which is generally the same as Model 1. The testing accuracy of Model 1 is the same as its final validation accuracy of 50.51% (Figure 1.2).

3. Model 2 (Word Embedding)

```
# Testing Base Model for the accuracy on Test Data
results = model2.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 0s 55us/sample - loss: 1.1513 - acc: 0.5368
[1.151287641928703, 0.53678024]
```

The testing accuracy of model 2 is only slightly more than Model 1. Though the accuracy increased only slightly from both Base Model and Model 1, there were significant changes made to certain hyperparameters such as dimensionality set to 150, maximum length set to 30, training samples set to 500,000, validation samples set to 40,000, maximum words set to 4,200,000. These are just minor changes compared to the build of the model, which is not changed much from Model 1. Hence, it justifies that the slight increase in testing accuracy between Model 1 and Model 2. Comparing the testing accuracy of Model 2, which is 53.67% and Model 1, which is has a testing accuracy of 50.51%. Model 2 has a higher testing accuracy, suggesting that it can do better in predictions.

Of the three word embedding models, I'd recommend Model 2 as it has a dimensionality of 150, compared to 25 on Base Model and 100 on Model 1, which suggests that it will be able to get a higher testing and prediction accuracy than both Base Model and Model 1. Although the lowest validation loss score for Model 2 is 1.1461 (Figure 2.2), it is still lower than Model 1 at 1.1660 (Figure 1.2) and Base Model at 1.2042 (Figure 0.2). I believe that the level of confidence here does not matter that much, since this is using probability to predict the sentiment of the sentences.

I believe that the accuracy of the model is more crucial as a model which has higher accuracy will have a higher chance of predicting the correct sentiment, compared to a model which picks the wrong sentiment with high confidence. Thus, the justification for my recommendation.

4. LSTM Model 1 (RNN Model – LSTM)

```
#Testing Base Model for the accuracy on Test Data
results = LSTM_Model1.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 1s 107us/sample - loss: 0.9903 - acc: 0.5931
[0.9902674125448377, 0.593067]
```

The testing accuracy is 59.30%. This is higher than the testing accuracy of Base Model as it has a testing accuracy of 50.31%. This might be affected by the difference in using Word Embedding as compared to RNN Models. The final validation accuracy is seen to also be lower at 51.44% for Base Model as compared to 51.50%. LSTM Model 1 hyperparameters such as maximum length, training samples, validation samples, maximum words and dimensionality as this model is using the same hyperparameters as Model 2.

5. LSTM Model 2 (RNN Model – LSTM)

```
#Testing Base Model for the accuracy on Test Data
results = LSTM_Model2.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 7s 765us/sample - loss: 1.0037 - acc: 0.5767
[1.0037204244302107, 0.57673323]
```

This model has a testing accuracy of 57.67%. This is a rather significant increase compared to the increases seen in the word embedding models. The testing accuracy for this higher compared to LSTM Model 1 as LSTM Model 1 has a testing accuracy of 59.30% compared to 57.67% of LSTM Model 2. There are no changes made to the hyperparameters such maximum length, training samples, validation samples, maximum words and dimensionality as this model uses the same one as Model 2.

I would not recommend LSTM Model 2 as a contender for the best model as it does not have a stable model and the testing accuracy is not that high as when compared to LSTM Model 1, LSTM Model 2 testing accuracy is lower than LSTM Model 1.

Instead, I would recommend LSTM Model 2 since it has a high validation and testing accuracy at 57.92% (Figure 4.2) and 59.30% respectively. The graph is also more stable in comparison to LSTM Model 2 and the validation loss graph is not as instable as LSTM Model 2.

6. GRU Model 1 (RNN Model – GRU)

```
#Testing Base Model for the accuracy on Test Data
results = GRU_Model1.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 1s 104us/sample - loss: 0.9866 - acc: 0.5964
[0.986611993920789, 0.5963572]
```

This GRU Model 1 has the exact same build as LSTM Model 1 as I would like to ensure a fair comparison between the testing accuracy of both LSTM Model 1 and GRU Model 1. This resulted in 59.63% testing accuracy for GRU Model 1 and 59.30% testing accuracy for LSTM Model 1. This meant that the LSTM Model was better able to reach a higher accuracy compared to a GRU Model within the same number of epochs.

Comparing the final validation accuracy of GRU Model 1 with LSTM Model 1, it was seen that LSTM Model 1 had the higher final validation accuracy at 59.49% while GRU Model 1 has the final validation accuracy of 60.31%. As the accuracy of GRU Model 1 is higher than LSTM Model 1, the added stability of the model by not having spikes and dips in accuracy allows it to be a great contender for the best model.

7. GRU Model 2 (RNN Model – GRU)

```
#Testing Base Model for the accuracy on Test Data
results = GRU_Model2.evaluate(X_test, y_test)
print(results)
```

```
8510/8510 [==============================] - 7s 770us/sample - loss: 1.0033 - acc: 0.5738
[1.0032957745214748, 0.57379556]
```

GRU Model 2 has a testing accuracy of 57.37%. It has a considerably low testing accuracy after all the fine tuning done to the model. This would still be a contender for the best model due to the model stability and the increasing validation accuracy from the validation accuracy graph. It might suggest that the GRU Model 2 can obtain a higher accuracy than what it has achieved now.

In this case, GRU Model 2 would not be a contender for the best model as the testing accuracy itself is already lower than GRU Model 1 and LSTM Model 1.

8. Conclusion for choosing the Best Model of DL Assignment 2 Part 1

Ultimately, it is a tough choice between GRU Model 1 and LSTM Model 1. Since GRU Model 1 has the higher testing accuracy, higher validation accuracy, the more stable validation accuracy extended graph and the more stable validation loss extended graph, the title of the best model has to be given to GRU Model 1 instead of LSTM Model 1.

Use Best Model to Make Prediction of DL Assignment 2 Part 1

1. Base Model (Word Embedding)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(base_model, items_l):
    prob = base_model.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(base_model, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.085093  0.071792  0.012335  0.023344  0.807436
```

The input text is taken in from the user and will be printed to show the user. The input used for all model predictions are “The ending of this show touched my heart, I love it so much.” The emoji I had associated this with was the heart emoji, which was the last one in the array mapping.

This can be used in real life context when people are giving feedbacks to certain things such as workshops or movies and output as emojis so that the host is able to see the percentage of people who liked, is neutral or disliked the show.

In the next line, the tokenizer was used to fit the input text of the user. It is then sequenced and padded and truncated. It will then be renamed as “text_input_edit” after it is sequenced and padded. The “text_input_edit” is then printed and shown in the output.

Next, the model will extract the data from mapping.csv. It will then locate the emotions and save it into “emoji_dictionary”. It will then print the emojis in the output. The next line will show the number of emojis in the array.

Next is the prediction function. The parameters included is “base_model” which can be changed as per model and “items_l”. The predict function is used for the base model, predicting the “text_input_edit” and saving the prediction as “prob”.

Once “prob” is put into “pd.Dataframe” which is a pandas data structure that aligns the rows and columns. The result is then saved as “pro_df”.

The “items_l” will be calculated for the probability using numpy argument max. This will be saved as “result”.

“prob_df” and “result” is used to predict the result of the sentiment and show the probability of the sentiment given by the model.

The emoji prediction for Base Model is the heart emoji, which the sentiment I expected the sentence to give out. The prediction accuracy for the heart emoji using Base Model is 80.74%, which is very high as it surpassed 80% prediction accuracy mark.

2. Model 1 (Word Embedding)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    1 2760   12    9  128 6808    4  111    5   15   24   21
    67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(model1, items_l):
    prob = model1.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(model1, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2       3         4
0  0.084695  0.056914  0.006034  0.0114  0.840958
```

Model 1 predicted the same input with a prediction accuracy of 84.09%. This is only a slight increase of 3.35% from Base Model. It is not a surprise as the validation and testing accuracy were also similar and did not differ much in terms of accuracy.

3. Model 2 (Word Embedding)

```
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(model2, items_l):
    prob = model2.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(model2, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.104138  0.039596  0.010805  0.015152  0.830309
```

Model 2 gave a prediction accuracy of 83.03% when predicting the same input. This was a slightly higher increase than compared to the difference between Base Model and Model 1. The increase of Base Model to Model 2 is 2.29% while decrease of Model 1 to Model 2 is 1.06%. There are a few factors that might cause this. The first one is that Model 2 has higher testing and validation accuracy compared to both Base Model and Model 1. The next factor is the change in dimensionality from 100 in Model 1 to 150 in Model 2. Model 2 correctly predicted the sentiment the sentence gave with a high accuracy.

4. LSTM Model 1 (RNN Model – LSTM)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(LSTM_Model1, items_l):
    prob = LSTM_Model1.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(LSTM_Model1, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.073328  0.006342  0.001789  0.001097  0.917444
```

LSTM Model 1 uses the exact same code as explained on Base Model. The same input was used, and it produced a result which predicts the sentiment of the model is love. This allows for the model to be tested equally as the models are all tested using the same input. The prediction accuracy for LSTM Model 1 is quite high at 91.74% compared to 83.08% of Model 2, 84.09% of Model 1 and 80.74% of Base Model.

5. LSTM Model 2 (RNN Model – LSTM)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(LSTM_Model2, items_1):
    prob = LSTM_Model2.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_1)
    result = items_1[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(LSTM_Model2, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.042867  0.003062  0.001344  0.001489  0.951238
```

This LSTM Model 2 is using the exact same code that was explained in Base Model. The one difference is that the made is the dimensionality of the code. This model obtained a 95.12% for the prediction accuracy, compared to LSTM Model 1 which is 91.74% and Model 2, at 83.08%. This shows that despite a higher accuracy for LSTM Model 1 and Model 2, LSTM Model 2 was still able to predict with an extremely high prediction accuracy.

6. GRU Model 1 (RNN Model – GRU)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(GRU_Model1, items_l):
    prob = GRU_Model1.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(GRU_Model1, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.061967  0.006303  0.001956  0.001651  0.928122
```

The GRU Model 1 has a prediction accuracy of 92.81%. This is higher compared to all the testing accuracy of other models. This can be due to several factors such as the type of RNN model ran. It is discovered that GRU networks are especially effective for the task of sentiment analysis as it has an ability to remember long term dependencies.

This GRU Model 1 did not attain the best prediction accuracy as I expected it to obtain. As when compared to LSTM Model 2, LSTM Model 2 obtained a testing accuracy of

95.12% compared to a testing accuracy if only 92.81% on GRU Model 1. This means that either the results from LSTM Model 2 prediction as shown before is a fluke that happened or that GRU Model 1 does not do as well in terms of prediction accuracy as I thought.

GRU networks are especially useful in large texts, compared to LSTM networks which are not able to converge to better accuracy. Hence, this factor alone has given the GRU models a huge advantage, which is shown here in the form of a very high prediction accuracy. Compared to LSTM Model 2 which has a prediction accuracy of 95.12% compared to 92.81% of GRU Model 1. The fact that the LSTM model was able to have a 2.31% advantage despite the GRU network technically having an advantage could mean that the results obtained for LSTM Model 2 was most likely to be a fluke. This is also further supported by the lower prediction accuracy obtained for LSTM Model 1, at only 91.74%, while a similarly built GRU Model 1 has a prediction accuracy of 92.81%.

7. GRU Model 2 (RNN Model – GRU)

```
# takes the user input
text_input = np.array([input()])
```

```
The ending of this show touched my heart, I love it so much.
```

```
# convert the user input into numeric tensor
tokenizer.fit_on_texts(text_input)
text_input = tokenizer.texts_to_sequences(text_input)
text_input_edit = preprocessing.sequence.pad_sequences(text_input, maxlen, truncating = "pre")
print(text_input_edit)
```

```
[[   0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    1 2760   12    9  128 6808    4  111    5   15   24
    21   67]]
```

```
# show the model output using predict function

import pandas as pd
df = pd.read_csv('mapping.csv', delimiter=',')
emoji_dictionary = df.loc[:,'emoticons'].to_dict()
print(emoji_dictionary)
print('A total of: ', len(emoji_dictionary), 'Emoji Icons')

import pandas as pd
def prediction(GRU_Model2, items_l):
    prob = GRU_Model2.predict(text_input_edit)
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result

prob_df, result = prediction(GRU_Model2, emoji_dictionary)
print('The prediction is: ', result, '\n\n', prob_df)
```

```
{0: '😍', 1: '😂', 2: '📷', 3: '🔥', 4: '❤'}
A total of:  5 Emoji Icons
The prediction is:  ❤

          0         1         2         3         4
0  0.048865  0.009439  0.001046  0.00159  0.93906
```

The prediction accuracy of this model is 93.90%. This is higher than the 92.81% prediction accuracy of GRU Model 1 but lower than the 95.12% of LSTM Model 2. The reason for this slight increase in prediction accuracy might be some hyperparameters in the model build. Comparing Model 2, which has a prediction accuracy of 83.03%, while GRU Model 2 has a prediction accuracy of 93.90%.

This clearly shows that the RNN network models are obtaining higher prediction accuracies compared to word embedding network models due to the algorithm in which it converges to determine the prediction.

8. Conclusion for Model Prediction of DL Assignment 2 Part 1

In conclusion, I was slightly incorrect for the model prediction as the model which gave the highest model prediction was LSTM Model 2 and not GRU Model 1. Perhaps it is the algorithm and the LSTM network itself which caused LSTM Model 2 to have a 2.31% more in prediction accuracy compared to GRU Model 1. Despite the higher validation accuracy of GRU Model 1, the LSTM Model 2 produced a higher prediction accuracy in the end.

Summary of DL Assignment 2 Part 1

In conclusion, LSTM Model 2 has a final validation accuracy of 57.92%, testing accuracy of 57.67%, and a prediction accuracy of 95.12%. Though the validation and testing accuracy were lower than most of the other models shown here, it managed to obtain the best prediction accuracy of 95.12% due to a couple of reasons.

Firstly, the hyperparameters are more favorable towards LSTM networks than GRU networks as there might be too much fine tuning involved.

Secondly, that the prediction accuracy output was a fluke from LSTM Model 2. I state this as a reason due to LSTM Model 1 prediction accuracy was lower than GRU Model 1 prediction accuracy.

I have also noticed when predicting LSTM Model 2 repeatedly for the next few times that the accuracy could not match that of the initial prediction accuracy of 95.12%. However, as this is a result predicted by the model, I am unable to close an eye on this initial finding and had to use this in this report.

Some of the suggestions for improvements can be adding in more layers for training as it increases the network, which perhaps might increase the overall accuracy of the model. By ensuring that fine tuning is properly done and not overdoing it is also an improvement to the model as I believe that LSTM Model 2 will be able to obtain a higher accuracy when properly fine-tuned and tested.

In conclusion, I am satisfied with the model and the progress it has made. I am happy with the prediction accuracy that LSTM Model 2 has predicted, as it shows that there are justified steps taken at every juncture to ensure that the next model would be better than the previous in terms of not only accuracy, but stability and confidence in predicting the sentiment.

This ends my Deep Learning Assignment 2 Part 1 Report on Sentiment Analysis Model.

Overview of DL Assignment Part 2

LSTM Base Model

The initial issue was the data processing and data cleaning. Data processing had some issues as I was not as familiar to this process. As for data cleaning, there was initially some issues with my code, hence, not all the unnecessary characters were cleaned out. But all these errors and difficulties has since been fixed. The prediction of characters was also an issue for me as I tried for quite some time to attempt to try the prediction, but after a week of trial and error, I was finally able to predict the character using an input field.

The objective was for me to build a good model with a reasonable accuracy. The first time I ran the model using LSTM, the model obtained an accuracy of around 50%. This was considered good enough for me to use as a starting point. Hence, I changed a few hyperparameters and it is now the LSTM Base Model. The base model for both LSTM and GRU will be fine-tuned to further increase the accuracy of the model.

My approach was to closely follow practical 8b and research as much as I could to gain more information on how to fine tune this model as there are many new hyperparameters such as window size, steps and so forth.

This graph below shows the key values recorded of the LSTM Base Model.

Recorded data type	Value	Epoch
Initial training accuracy	0.3629	01st Epoch, 1st Batch, Figure 1.1
Final training accuracy	0.6464	10th Epoch, 4th Batch, Figure 1.2
Initial validation accuracy	0.4295	01st Epoch, 1st Batch, Figure 1.1
Final validation accuracy	0.5757	10th Epoch, 4th Batch, Figure 1.2
Initial training loss	2.1833	01st Epoch, 1st Batch, Figure 1.1
Final training loss	1.1385	10th Epoch, 4th Batch, Figure 1.2
Initial validation loss	1.9217	01st Epoch, 1st Batch, Figure 1.1
Final validation loss	1.4331	10th Epoch, 4th Batch, Figure 1.2

As shown in the table, the values are very clearly defined for the training and validation. In the graph below for this model, it is seen that the training and validation graphs do not intersect. This has caused me confusion but after a few attempts, I have concluded that the graph was indeed correct and there was no need for the graph to look like a standard graph. The graph shows a final validation accuracy of only 57.57% and a final training accuracy of 64.64%. This is a rather decent start but requires fine tuning to ensure higher accuracy for future models.

GRU Base Model

There was no problem faced in the GRU Base Model as it is the exact same as LSTM Base Model. I did this to try and compare the differences between the two.

The objective of this model is to compare with LSTM Base Model, carefully analysis the training accuracy and loss as well as validation accuracy and loss. The stability of the model is also taken into consideration.

My approach was to mimic the conditions and environment at which LSTM Base Model was trained and predicted such that GRU Base Model would have a fair comparison.

Recorded data type	Value	Epoch
Initial training accuracy	0.4017	01st Epoch, 1st Batch, Figure 2.1
Final training accuracy	0.6055	10th Epoch, 4th Batch, Figure 2.2
Initial validation accuracy	0.4478	01st Epoch, 1st Batch, Figure 2.1
Final validation accuracy	0.5256	10th Epoch, 4th Batch, Figure 2.2
Initial training loss	2.0259	01st Epoch, 1st Batch, Figure 2.1
Final training loss	1.2656	10th Epoch, 4th Batch, Figure 2.2
Initial validation loss	1.8267	01st Epoch, 1st Batch, Figure 2.1
Final validation loss	1.6064	10th Epoch, 4th Batch, Figure 2.2

As seen from this table above, when compared to LSTM Base Model, the GRU Base Model does not have as high of a training and validation accuracy. The validation accuracy of GRU Base Model is 52.56% compared to 57.57% of LSTM Base Model. This suggests in general, the GRU Model should tend to have a lower accuracy when compared to the LSTM Models.

LSTM Model 1

One problem I faced when I tried to fine-tune the model was that when I attempted to add more dense layers to the model build, the model gave an error. The objective was to be able to create a model that uses the strengths of the LSTM network and allow it to obtain the best accuracy possible. The approach was to research and obtain more information as to apply it.

Recorded data type	Value	Epoch
Initial training accuracy	0.2848	01st Epoch, Figure 3.1
Final training accuracy	0.6449	20th Epoch, Figure 3.2
Initial validation accuracy	0.3675	01st Epoch, Figure 3.1
Final validation accuracy	0.5751	20th Epoch, Figure 3.2
Initial training loss	2.5134	01st Epoch, Figure 3.1
Final training loss	1.1526	20th Epoch, Figure 3.2
Initial validation loss	2.1853	01st Epoch, Figure 3.1
Final validation loss	1.4169	20th Epoch, Figure 3.2

As seen from the table above, the final validation accuracy for LSTM Model 1 which has an accuracy of 57.52%, is higher than GRU Base Model at 52.56% but only slightly lower than LSTM Base Model at 57.57%. This shows that the model is a likely contender to be the best model.

GRU Model 1

The issue was that the accuracy was not as high as I would like to expect. I have tried all forms of fine-tuning, but the accuracy remains around the same. My objective is the same as LSTM Model 1, to utilize the strengths of the GRU network such that it can give the best prediction possible. The approach was to conduct research and test it.

Recorded data type	Value	Epoch
Initial training accuracy	0.3921	01st Epoch, Figure 4.1
Final training accuracy	0.6598	15th Epoch, Figure 4.2
Initial validation accuracy	0.4460	01st Epoch, Figure 4.1
Final validation accuracy	0.5836	15th Epoch, Figure 4.2
Initial training loss	2.0777	01st Epoch, Figure 4.1
Final training loss	1.0990	15th Epoch, Figure 4.2
Initial validation loss	1.8477	01st Epoch, Figure 4.1
Final validation loss	1.4186	15th Epoch, Figure 4.2

Overall, the model was done well, and it was higher than all the other models as the GRU network was able to be taken advantage on in this model. The final validation accuracy of GRU Model 1 is 58.36% compared to 57.51% of LSTM Model 1. Further fine tuning is needed to increase the accuracy of the model, but as of now, I am currently satisfied with the model accuracies as shown here.

Data Loading and Processing for DL Assignment 2 Part 2

1. Data Loading

```
# read in the text file, transforming everything to lower case
text = open('holmes.txt').read().lower()
print('The original text has ' + str(len(text)) + ' characters.\n')
```

```
The original text has 562436 characters.
```

Data Loading is used to open “holmes.txt” and change it to lowercase before saving it as “text”. The number of characters is then printed to be displayed as shown below.

2. Data Processing

```
### print out the first 2000 characters of the raw text to get a sense of what characters to remove
text[:2000]
```

```
"the adventures of sherlock holmes by sir arthur conan doyle\n\n   i. a scandal in bohemia\n  ii. the red-headed league\n iii
i. a case of identity\n  iv. the boscombe valley mystery\n   v. the five orange pips\n  vi. the man with the twisted lip\n v
ii. the adventure of the blue carbuncle\nviii. the adventure of the speckled band\n  ix. the adventure of the engineer's thu
mb\n   x. the adventure of the noble bachelor\n  xi. the adventure of the beryl coronet\n xii. the adventure of the copper b
eeches\n\n\nadventure i. a scandal in bohemia\n\ni.\n\nto sherlock holmes she is always the woman. i have seldom heard\nhim
mention her under any other name. in his eyes she eclipses\nand predominates the whole of her sex. it was not that he felt\n
any emotion akin to love for irene adler. all emotions, and that\none particularly, were abhorrent to his cold, precise but\
nadmirably balanced mind. he was, i take it, the most perfect\nreasoning and observing machine that the world has seen, but
```

This prints out the first 2000 characters of the raw, unedited text to be displayed.

```
# remove all '\n' and '\r' from text
text = text.replace('\n',' ')
text = text.replace('\r','')

text[:7000]
```

This command replaces ‘\n’ and ‘\r’ with a space (‘ ’). This will clean up much of the unnecessary characters and leave more common characters such as punctuations. This is done to allow the model, when predicting, to accurately obtain an accuracy that is as high as possible as well as complete the English words correctly.

```
def clean_text(text):
    punctuation = ['!', ',', '.', ':', ';', '?', '-', "'",' ']
    letters='abcdefghijklmnopqrstuvwxyz'
    letters=list(letters)
    for x in text:
        if x not in punctuation and x not in letters:
            text=text.replace(x,'')
    return(text)
```

This function is created to clean the text further other than just replacing '\n' and '\r' with spaces. This clears out all the unnecessary punctuations, leaving only the alphabets and exclamation mark ('!'), commas (','), full-stops ('.'), colon (':'), semi-colon (';'), question mark ('?'), dash ('-'), single-quote (" ' ") and space (' '), Any other characters listed in the text are removed.

```
# clean data using clean_text function
text = clean_text(text)
text[:2000]
```

```
"the adventures of sherlock holmes by sir arthur conan doyle     i. a scandal in bohemia   ii. the red-headed league  iii. a
case of identity   iv. the boscombe valley mystery    v. the five orange pips   vi. the man with the twisted lip  vii. the a
dventure of the blue carbuncle viii. the adventure of the speckled band   ix. the adventure of the engineer's thumb    x. th
e adventure of the noble bachelor   xi. the adventure of the beryl coronet  xii. the adventure of the copper beeches   adven
ture i. a scandal in bohemia  i.  to sherlock holmes she is always the woman. i have seldom heard him mention her under any
other name. in his eyes she eclipses and predominates the whole of her sex. it was not that he felt any emotion akin to love
for irene adler. all emotions, and that one particularly, were abhorrent to his cold, precise but admirably balanced mind. h
e was, i take it, the most perfect reasoning and observing machine that the world has seen, but as a lover he would have pla
ced himself in a false position. he never spoke of the softer passions, save with a gibe and a sneer. they were admirable th
```

This code will then clean the code and save it as text. It will then print out the first 2000 characters of the text as shown above.

```
# count the number of unique characters in the text
chars = sorted(list(set(text)))

# print some of the text, as well as statistics
print ("This document has " +  str(len(text)) + " total number of characters.")
print ("This document has " +  str(len(chars)) + " unique characters.")
```

```
This document has 556982 total number of characters.
This document has 35 unique characters.
```

The text is sorted and placed into a list to be saved as unique characters. This then prints out the total number of characters and unique characters found in the text after thorough cleaning. There are 556,982 total characters and 35 unique characters in total.

```
def generate_text_io(text, window_size):
    inputs = [] # store inputs
    labels = [] # stores label

    # Enter your code here:
    step = 3

    for i in range(0, len(text) - window_size, step):
        inputs.append(text[i: i + window_size])
        labels.append(text[i + window_size])
    print('Number of sequences:', len(inputs))
    #Number of sequences: 557272
    #Unique characters: 51
    return [inputs, labels]
```

This function “generate_text_io” was created to store the input and labels. The steps defined here are per model basis. In this case, this screenshot was taken from the GRU Base Model. It will then print the number of sequences and unique characters, which are 557,272 and 51, respectively. It will then return the inputs and labels.

```
# this dictionary is a function mapping each unique character to a unique integer
chars_to_indices = dict((c, i) for i, c in enumerate(chars))  # map each unique character to unique integer

# this dictionary is a function mapping each unique integer back to a unique character
indices_to_chars = dict((i, c) for i, c in enumerate(chars))  # map each unique integer back to unique character
```

This section of code maps each unique character to a unique integer.

```
# create a function 'encode_io_pairs' to perform one-hot encoding of inputs and labels
import numpy as np

def encode_io_pairs(text,window_size): # window_size determines # of characters in each input
    # Enter your code here:
    print('Vectorization...')
    sentence=generate_text_io(text,window_size)
    sentences=sentence[0]
    next_chars=sentence[1]
    x = np.zeros((len(sentences), window_size, len(chars)), dtype=np.bool)
    y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
    for i, sentence in enumerate(sentences):
        for t, char in enumerate(sentence):
            x[i, t, chars_to_indices[char]] = 1
            y[i, chars_to_indices[next_chars[i]]] = 1
    return x, y
```

This section of code performs creates the function “encode_io_pairs”, which encode the inputs and labels. The window size here determines the number of characters which are allowed per input.

```
# perform one-hot encoding of inputs and labels

window_size = 100

X, y = encode_io_pairs(text, window_size)
```

This code specifies the window size of the model. Window size is specified as the number of characters included in a sequence. The inputs and labels are then one-hot encoded using the “encode_io_pairs” function.

3. Data Splitting

```
# Split the X & y into train and test sets
from tensorflow.keras import preprocessing
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = 32)
print(X_train.shape)
print(len(X_train.shape))
```

```
(445505, 100, 35)
3
```

This section of the code shows the data and labels being split into training and testing set. The test size here is set to 20% of the data with a random state of 32. The shape and length of the training data is then printed.

Develop the Sentimental Analysis Models using Training Data for DL Assignment 2 Part 2

1. LSTM Base Model

a. Building the LSTM Base Model

```
from tensorflow.keras import layers

maxlen = 100
step = 3

LSTM_base_model = keras.models.Sequential()
LSTM_base_model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
LSTM_base_model.add(layers.Dense(len(chars), activation='softmax'))

from tensorflow.keras import optimizers

optimizer = optimizers.RMSprop(lr=0.001)
LSTM_base_model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
LSTM_base_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 128)               83968
_________________________________________________________________
dense (Dense)                (None, 35)                4515
=================================================================
Total params: 88,483
Trainable params: 88,483
Non-trainable params: 0
```

This is the build for the LSTM Base Model, which shows one LSTM layer and one dense layer, using “softmax” as the final activation function.

The optimizer used here is RMSprop, with a learning rate value of 0.001. It is set low this model has been run beforehand and the accuracy will increase and decrease quickly when the learning rate is set too high.

The loss function used is categorical crossentropy as the data and labels are now one-hot encoded. There is a total of 88,483 trainable parameters in this model.

The maximum length used here is 100, meaning that in a sequence, there is 100 characters. A step of 3 is used, meaning that after a sequence, the next sequence will start at 3 characters after the previous sequence. The lower the value of the steps, the higher the accuracy of the model in training, validation, and prediction accuracy.

b. Training the LSTM Base Model

```
import random
import sys

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

for epoch in range(1, 5):
    print('epoch', epoch)
    history = LSTM_base_model.fit(X_train, y_train,
              epochs = 10,
              batch_size=128)

    # Select a text seed at random
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [0.2, 0.5, 0.7, 1.0, 1.2]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)

        for i in range(1000):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, chars_to_indices[char]] = 1.

            preds = LSTM_base_model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

The “sample” function was used to draw from the model, a probability distribution over the next character given the available text, which in this case, is from “holmes.txt”.

It then reweighs the distribution to a certain temperature and samples the next character at random according to the updated weights distribution. It then adds a character at the back of the current available text to “predict” texts.

```
    history = LSTM_base_model.fit(X_train, y_train,
              epochs = 10,
              batch_size=128)
```

The epochs trained are very different from usual as these are trained in batches. So, the model will train 10 epochs. The batch size used is 128 and will not change for the other models.

```
for epoch in range(1, 5):

    for temperature in [0.2, 0.5, 0.7, 1.0, 1.2]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)
```

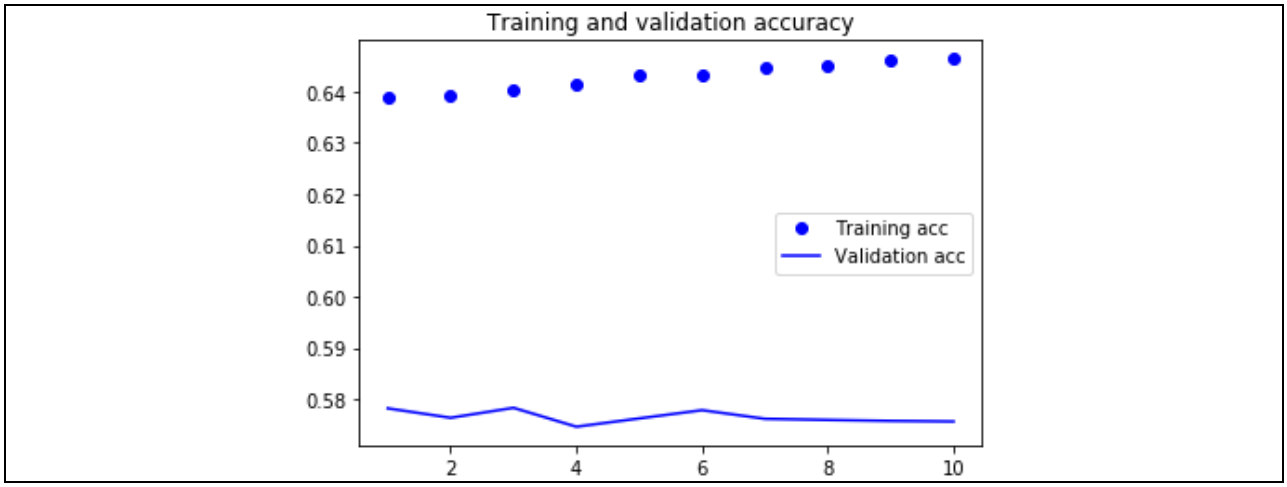
The total number of times trained is 4. Essentially, the model will train 40 epochs, but will print the temperature as specified in the model build after 10 epochs. The temperatures can be increased and decreased as per liking of the user. The accuracy and loss will not reset after 10 epochs as the model assumes that this is one continuous training.

```
        for i in range(1000):
```

The temperature printed will be texts that are predicted according to the temperature. A temperature too low or too high might result in a gibberish text while a decent and averaged temperature such as 0.5 will allow the model to display some English words while “predicting” the texts. The number of words is determined by the range as shown in the image above.

c. Plotting the graph of LSTM Base Model

Graph of Training and Validation Accuracy for LSTM Base Model

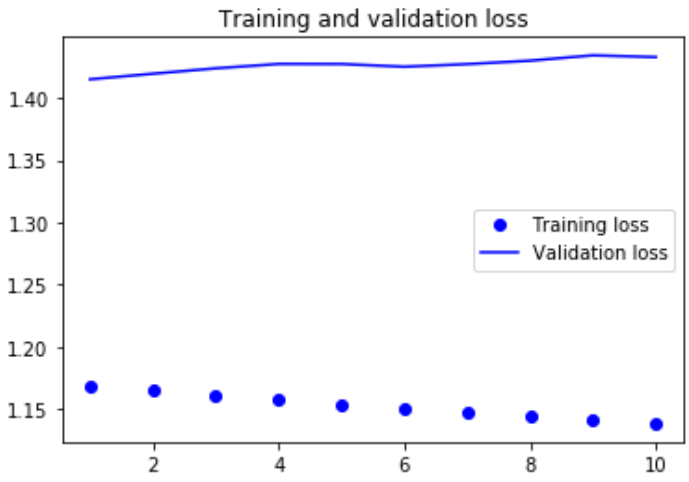


This graph is not like regular graphs where the validation graph initially starts at a higher accuracy than training accuracy. This graph here shows the final 10 epochs that were trained by the model. The initial and final accuracy are defined as the very first epoch and very last epoch of the model, and it applies to all models. The initial training accuracy is 36.29% (Figure 1.1) and final training accuracy is 64.64% (Figure 1.2). The initial validation accuracy and final validation accuracy are 42.95% (Figure 1.1) and 57.57% (Figure 1.2) respectively.

As seen from the graph above, the training accuracy is way too high the model can easily predict the next character as compared to the validation accuracy. One reason might be the validation split being only 20% of the original data, thus the model was not able to do much ensure that the validation accuracy is comparable to the training accuracy. Thus, this led to a graph like this to be produced. The graph is also not stable as seen from the uneven increase of the validation accuracy.

The training accuracy is seen to be increasing, but not in a smooth manner. Fine tuning can be used to improve the stability of future models.

Graph of Training and Validation Loss for LSTM Base Model



The training loss as seen here decreases steadily, with the initial training loss at 2.1833 (Figure 1.1) and the final training loss at 1.1385 (Figure 1.2). The validation loss graph shows an increase instead of a decrease. The initial validation loss is 1.9217 (Figure 1.1) and the final validation loss is 1.4331 (Figure 1.2). This shows a rather huge difference between the loss of training and validation loss. This shows that the model requires some fine-tuning in the hyperparameters to reduce the training and validation loss, which in turn could increase the confidence in the prediction of characters for the model.

```
Epoch 1/10
356404/356404 [==============================] - 28s 78us/sample - loss: 2.1833 - acc: 0.3629 - val_loss: 1.9217 - val_acc: 0.4295
```

Figure 1.1 (LSTM Base Model, 1st Epoch of 1st batch)

```
Epoch 10/10
356404/356404 [==============================] - 25s 70us/sample - loss: 1.1385 - acc: 0.6464 - val_loss: 1.4331 - val_acc: 0.5757
```

Figure 1.2 (LSTM Base Model, 10th Epoch of 4th batch)

2. GRU Base Model

a. GRU Base Model

```
# Build the Model
# Enter your code here:
from tensorflow.keras import layers

maxlen = 100
step = 3

GRU_base_model = keras.models.Sequential()
GRU_base_model.add(layers.GRU(128, input_shape=(maxlen, len(chars))))
GRU_base_model.add(layers.Dense(len(chars), activation='softmax'))

from tensorflow.keras import optimizers

optimizer = optimizers.RMSprop(lr=0.005)
GRU_base_model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
GRU_base_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 128)               63360
_________________________________________________________________
dense (Dense)                (None, 35)                4515
=================================================================
Total params: 67,875
Trainable params: 67,875
Non-trainable params: 0
```

The GRU Base Model also has a maximum length of 100 and a step of 3. The learning rate was different from LSTM Base Model as the model quickly increase and decrease in both training and validation accuracy. The total trainable parameters for this model are 67,875. This model is built similarly to LSTM Model 1 as I would like to make comparisons between the two. For example, even though both LSTM and GRU Base Model has 128 as their input nodes, the total trainable parameters was different between LSTM and GRU Base Model despite all hyperparameters kept the same. LSTM Base Model has 88,483 while GRU Base Model has 67,875 total trainable parameters. This shows that LSTM would create more parameters compared to GRU even if the hyperparameters are the same.

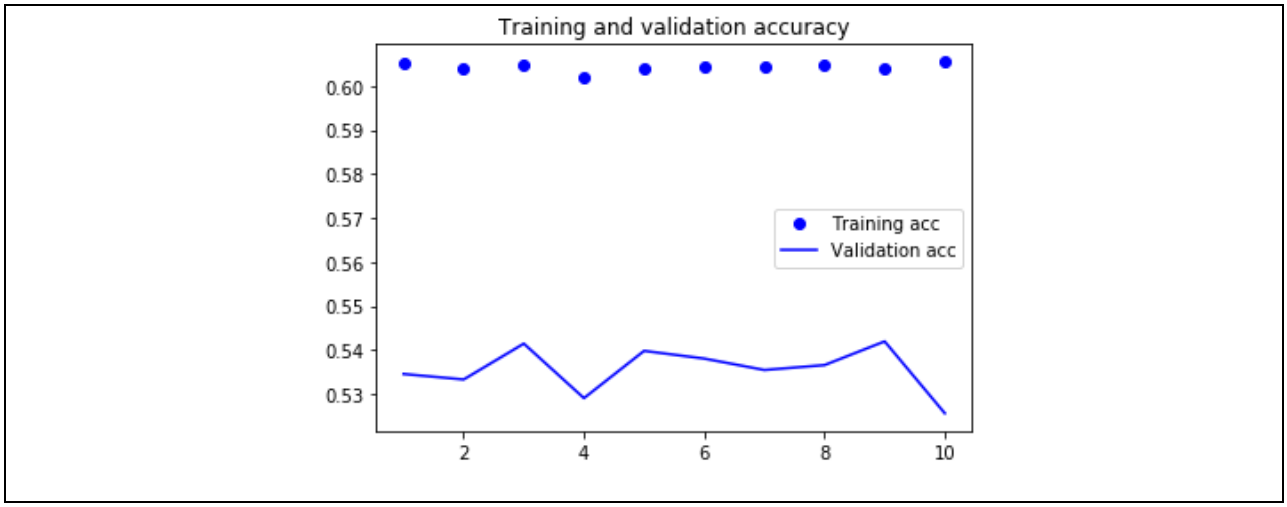
b. Training GRU Base Model

```
import random
import sys

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

for epoch in range(1, 5):
    print('epoch', epoch)
    history = GRU_base_model.fit(X_train, y_train,
              epochs = 10,
              batch_size=128)

    # Select a text seed at random
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [0.2, 0.5, 0.7, 1.0, 1.2]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)

        for i in range(1000):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, chars_to_indices[char]] = 1.

            preds = GRU_base_model.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

The training code for GRU Base Model is the same as LSTM Model 1. The number of epochs, the number of batches the epoch ran, the temperature at which the characters are predicted and the number of characters that are printed for each temperature all remain the same as LSTM Base Model.

c. Plotting the graph of GRU Base Model

Graph of Training and Validation Accuracy for GRU Base Model



The training accuracy of GRU Base Model is not that stable, as seen by the decrease then increase of the training graph. It is rather low as compared to LSTM Base Model, with the LSTM Base Model final training accuracy at 64.64% (Figure 1.2) compared to only 60.55% (Figure 2.2) of the GRU Base Model. The validation accuracy of GRU Base Model is completely not stable as seen from the graph above as it randomly increases and decreases. The initial validation accuracy of GRU Base Model is 44.78% (Figure 2.1) and the final validation accuracy is 52.56% (Figure 2.2) . Comparing the final validation accuracy of LSTM Base Model to GRU Base Model, the former has a validation accuracy of 57.57% (Figure 1.2) compared to 52.56% (Figure 2.2) of the latter. This might perhaps be due to certain algorithm differences while processing the validation data.

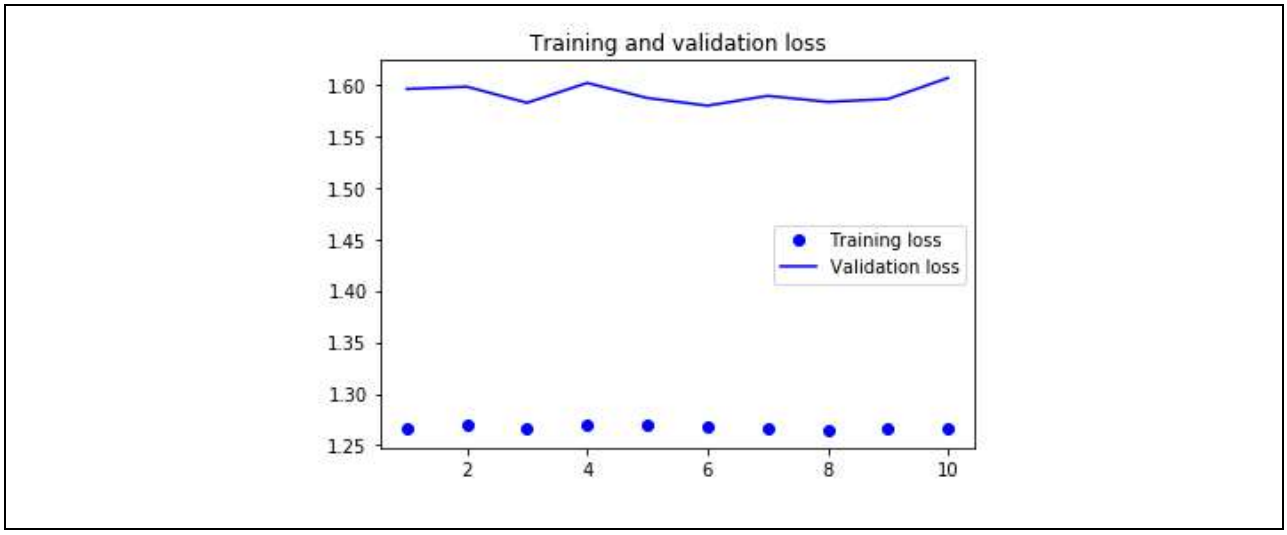
As the LSTM network has two separate gates, the Forget Gate which determines which part of the previous cell state to retain, and the Input Gate that determines the amount of new memory to be added. These two gates are independent to one another and the new information added will not be mixed up in the forget gate.

While in the GRU network, the Update Gate is responsible for retaining information from previous memory and control which new memory to be added. This means that the gates are not independent of one another.

There are also fewer numbers of weights and parameters to update for the GRU networks compared to LSTM networks, hence, the model might seem more unstable as compared to the LSTM Base Model training accuracy graph as the weights and parameters are not updated as frequently and to the similar extent seen on the LSTM networks.

More fine tuning could be done such as increasing the window size and maximum length of text in a sequence, as well as increasing the batch size when training the model. But as I have tried many other methods, there seems to be a limit on much fine-tuning can be used to improve the accuracy of this model.

Graph of Training and Validation Loss for GRU Base Model



The training loss is seen to be stable in the graph above. The final training graph loss score of GRU Base Model is 1.2656 (Figure 2.2) compared to LSTM Base Model at 1.1385 (Figure 1.2). The validation graph here seems to also be unstable due to the spikes and dips in accuracy as seen above. The validation loss tend to hover around the value of 1.58 and occasionally spike to 1.60. In this model, the final validation loss score is 1.6064 (Figure 2.2). Comparing this to the final validation loss score of LSTM Base Model, the LSTM Base Model has a final validation loss score of 1.43311 (Figure 1.2). This shows that an LSTM network is able to perform better in terms of accuracy when it comes to character prediction due to the ability to be more accurate on a dataset using longer sequences.

This model can be further improved by changing certain hyperparameters according to the strengths of each model. As seen from the comparison and analysis of the GRU and LSTM Base Model, the LSTM network has more trainable parameters and allows for greater accuracy when working with large datasets. While the GRU network is faster, works better with smaller sequences, and does not produce an accuracy as high as an LSTM network. Thus, the models built upon this knowledge will be tuned to their specific strengths.

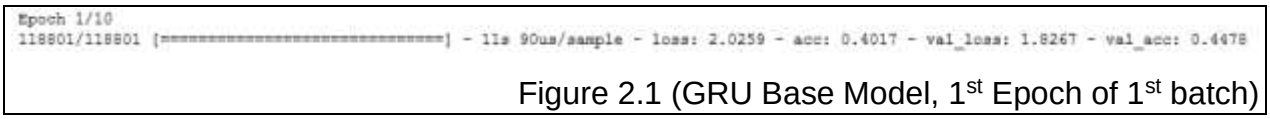


Figure 2.1 (GRU Base Model, 1st Epoch of 1st batch)

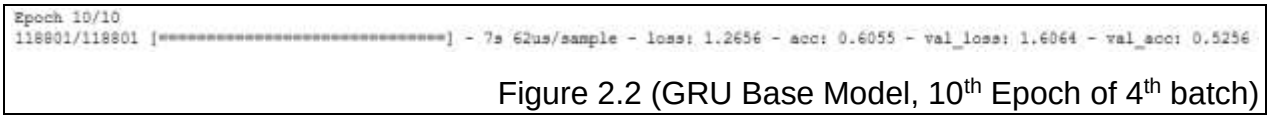


Figure 2.2 (GRU Base Model, 10th Epoch of 4th batch)

3. LSTM Model 1

a. Building LSTM Model 1

```
# Build the Model
# Enter your code here:
from tensorflow.keras import layers

maxlen = 300
step = 1

LSTM_model1 = keras.models.Sequential()
LSTM_model1.add(layers.LSTM(256, input_shape=(maxlen, len(chars))))
LSTM_model1.add(layers.Dense(len(chars), activation='softmax'))

from tensorflow.keras import optimizers

optimizer = optimizers.RMSprop(lr=0.001)
LSTM_model1.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
LSTM_model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 256)               299008
_________________________________________________________________
dense (Dense)                (None, 35)                8995
=================================================================
Total params: 308,003
Trainable params: 308,003
Non-trainable params: 0
```

The maximum length done here is 300 as compared to 100 in LSTM Base Model. The step is also set to 1 as compared to 3 in the LSTM Base Model. This will help to increase the accuracy of the model as the next sequence will start 1 character after the previous sequence has ended. The number of nodes has also increased from 128 to 256, this will give this LSTM Model more trainable parameters compared to the LSTM Base Model as this has 308,003 total trainable parameters as compared to 88,483 on the LSTM Base Model. The learning rate was changed from 0.005 on LSTM Base Model to 0.001 as a learning rate of 0.005 will overfit the model significantly faster.

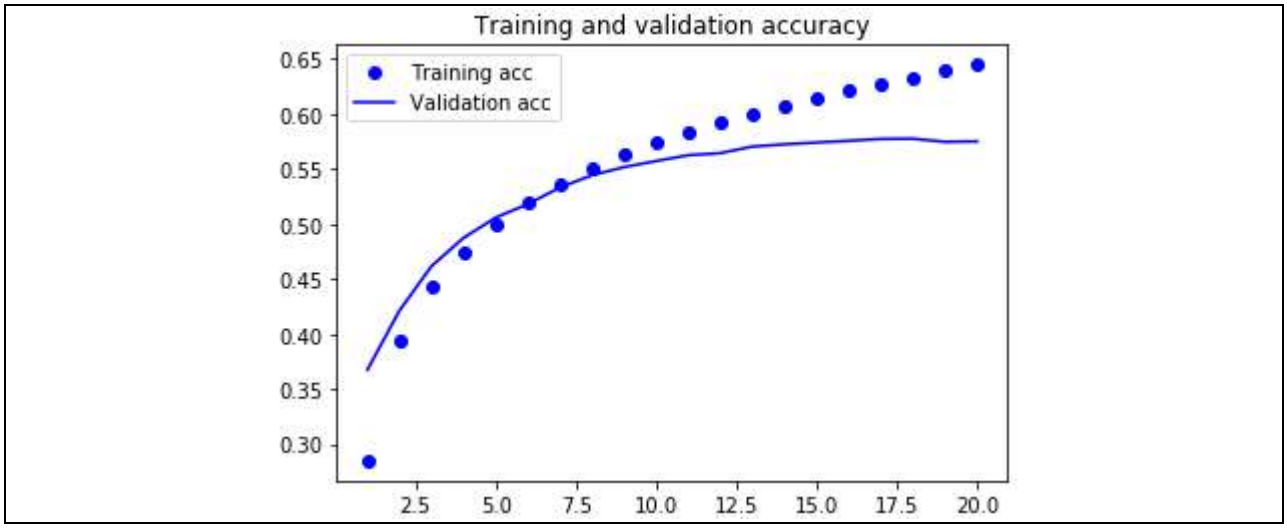
b. Training LSTM Model 1

```
import random
import sys

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

for epoch in range(1, 2):
    print('epoch', epoch)
    history = LSTM_model1.fit(X_train, y_train,
              epochs = 20,
              batch_size=1024,
              validation_split = 0.2)

    # Select a text seed at random
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [1.5, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)

        for i in range(50):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, chars_to_indices[char]] = 1.

            preds = LSTM_model1.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

This is the training build for LSTM Model 1. The number of epochs trained is only 20 overall, afterwards, the text will be “predicted” according to temperature. The number of characters that will be “predicted” is 50 characters. The batch size was increased here to reduce the number of times it will update the weights internally.

c. Plotting the graph of LSTM Model 1

Graph of Training and Validation Accuracy for LSTM Model 1

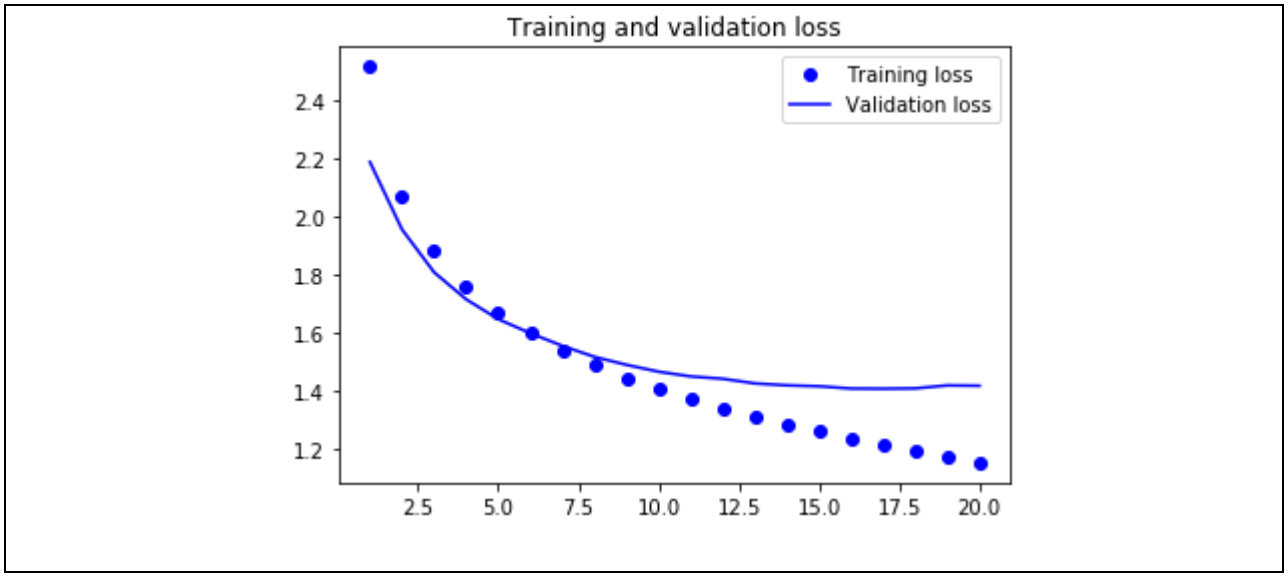


The training and accuracy graph as seen is stable. The training accuracy starts at a lower point compared to the validation, slowly increasing stably as the validation accuracy begins to flatten. The initial training and validation accuracy are 28.48% and 36.75% respectively (Figure 3.1). The model overfitted on the 6th epoch onwards. This then sees the training accuracy increasing constantly as compared to validation accuracy, which begins to flatten.

The final training and validation accuracy are 64.49% and 57.51% respectively (Figure 3.2). The validation accuracy is seen to be quite high as I stopped the training when the validation accuracy is at its highest around the 20th epoch. This form of regularization is called Early Stopping, where a model training is cut short to increase the accuracy and reduce the overfitting of the model. Compared to LSTM Base Model, the final training accuracy of LSTM Base Model is higher than LSTM Model 1 at 64.64% (Figure 1.2) and 64.49% (Figure 3.2) respectively. But keep in mind that LSTM Model 1 training accuracy could be higher if early stopping regularization was not implemented.

The final validation accuracy was also seen to higher on LSTM Base Model at 57.57% (Figure 1.2) as compared to LSTM Model 1 at 57.51% (Figure 3.2). However, the shape of the model and the overall stability of LSTM Model 1 is more favorable compared to LSTM Base Model.

Graph of Training and Validation Loss for LSTM Model 1



The loss graph for both training and validation are smooth and there seems to be no spikes or dips in scores. The training loss initially started out at 2.5134 (Figure 3.1), which was considerably higher than the initial validation loss at 2.1853 (Figure 3.1). The validation loss decreased at a faster rate, then slowing down on the 7th epoch onwards as seen from the graph above. The training loss decreased constantly. The final training and validation loss are 1.1526 and 1.4169, respectively (Figure 3.2). When compared to LSTM Base Model, the final training loss is higher on LSTM Base Model at 1.9217 (Figure 1.2) compared to LSTM Model 1 at 1.1526 (Figure 3.2). The final validation loss is higher on LSTM Base Model at 1.4331 (Figure 1.2) compared to 1.4169 of LSTM Model 1 (Figure 3.2). The lower loss increases the confidence of the model when predicting characters.

In general, this model has greatly improved in term of the shape of the graph and the general stability of the model. Although the training and validation accuracy did not improve much, the model is much better able to predict characters as compared to the LSTM Base Models. More fine tuning can be done, such as adding more layer to the model build. But I kept encountering errors while doing so. Thus, I have refrained from doing that.

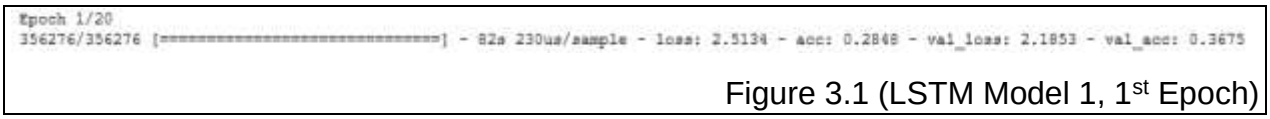


Figure 3.1 (LSTM Model 1, 1st Epoch)

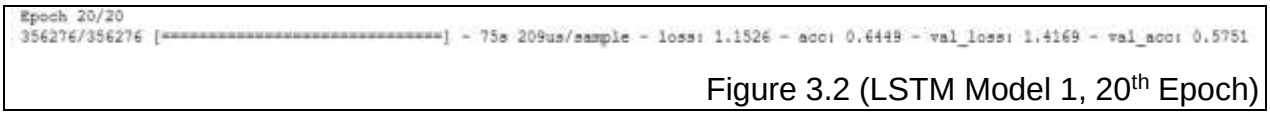


Figure 3.2 (LSTM Model 1, 20th Epoch)

4. GRU Model 1

a. Building GRU Model 1

```
# Build the Model
# Enter your code here:
from tensorflow.keras import layers

maxlen = 50
step = 1

GRU_model1 = keras.models.Sequential()
GRU_model1.add(layers.GRU(256, input_shape=(maxlen, len(chars))))
GRU_model1.add(layers.Dense(len(chars), activation='softmax'))

from tensorflow.keras import optimizers

optimizer = optimizers.RMSprop(lr=0.0005)
GRU_model1.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
GRU_model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
gru (GRU)                    (None, 256)               225024
_________________________________________________________________
dense (Dense)                (None, 35)                8995
=================================================================
Total params: 234,019
Trainable params: 234,019
Non-trainable params: 0
```

The maximum length set here is only 50 as explained earlier in GRU Base Model, the GRU network works better in shorter sequences. Hence, in this model, this is done to maximize the benefits of the model. The step is set to the value of 1. The number of input nodes remains similar to LSTM Model 1 at 256. The optimizer value here is set to 0.0005 as I have tried with 0.001 and it was still overfitting very quickly. Thus, a lower value was set, and this value works nicely. There are 234,019 total trainable parameters. This is less than LSTM Model 1 although the same input nodes of 256 was used. This shows that the LSTM network has a set of algorithms that generates the total trainable parameters differently as compared to GRU network.

b. Training GRU Model 1

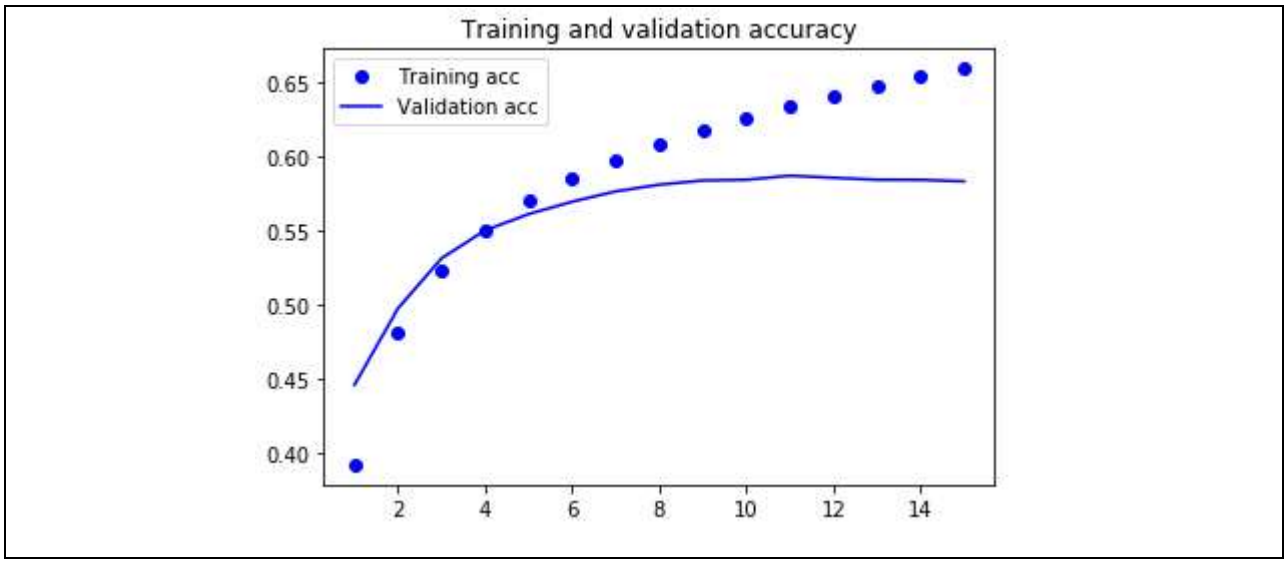
```
import random
import sys

def sample(preds, temperature=0.5):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)

for epoch in range(1, 2):
    print('epoch', epoch)
    history = GRU_model1.fit(X_train, y_train,
              epochs = 15,
              batch_size=64,
              validation_split = 0.2)

    # Select a text seed at random
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [0.4, 0.45, 0.5, 0.55, 0.6]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)

        for i in range(200):
            sampled = np.zeros((1, maxlen, len(chars)))
            for t, char in enumerate(generated_text):
                sampled[0, t, chars_to_indices[char]] = 1.

            preds = GRU_model1.predict(sampled, verbose=0)[0]
            next_index = sample(preds, temperature)
            next_char = chars[next_index]

            generated_text += next_char
            generated_text = generated_text[1:]

            sys.stdout.write(next_char)
            sys.stdout.flush()
        print()
```

The epochs here used is only 15 as a form of regularization called Early Stopping. The number of characters “predicted” in this case is 200.

The batch size is reduced to 64 as I would like for the model to update the internal weights more often since this uses a lesser number of maximum number of characters per sequence.

c. Plotting the graph of GRU Model 1

Graph of Training and Validation Accuracy for GRU Model 1



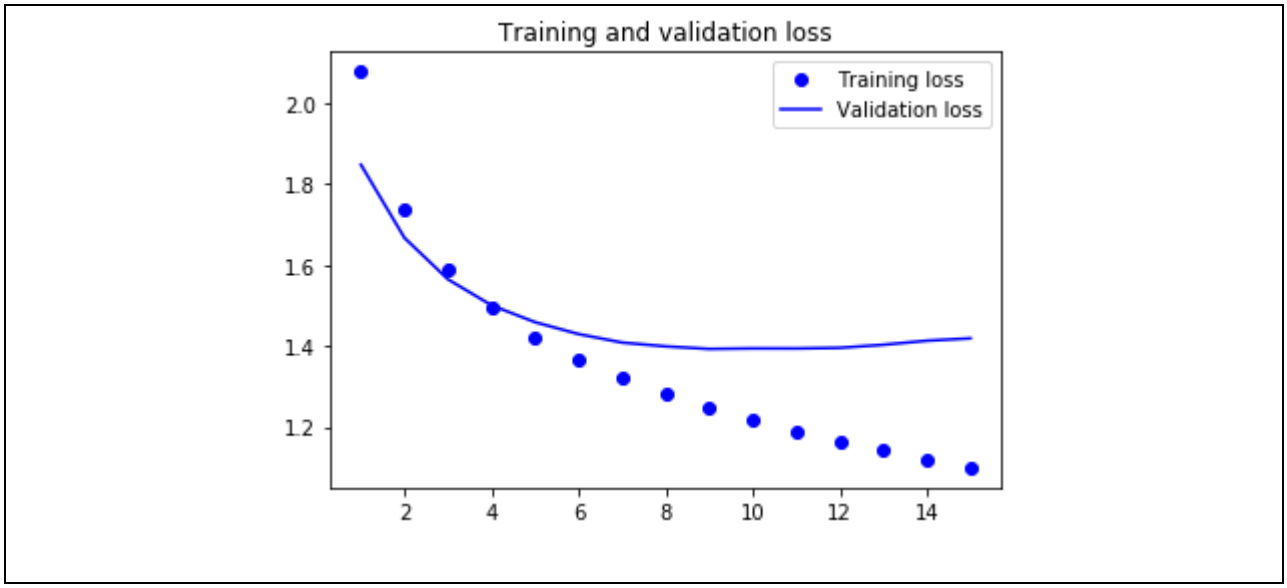
The training and validation accuracy graph as shown above is stable. The training accuracy starts at a lower point at 39.21% (Figure 4.1) compared to the validation accuracy, which starts at 44.60% (Figure 4.1). As the model continues to train, the validation accuracy flattens while the training accuracy continues to increase. This model overfitted on the 4th epoch as shown from the graph above. The final training and validation accuracy are 65.98% and 58.36% respectively (Figure 4.2).

Comparing this to the final validation accuracy of GRU Base Model which is 52.56% (Figure 2.2), this model has a higher final validation accuracy at 58.36% (Figure 4.2).

Comparing to the final validation accuracy of LSTM Model 1 at 57.51% (Figure 3.2), GRU Model 1 has a higher final validation accuracy at 58.36% (Figure 4.2).

Comparing GRU Model 1 to LSTM Model 1, the final validation accuracy of GRU Model 1 is 58.36% (Figure 4.2) compared to 57.51% (Figure 3.2) of LSTM Model 1. The final validation accuracy of GRU Model 1 is higher perhaps due to the algorithm of the GRU network. It can also be due to other factors.

Graph of Training and Validation Loss for GRU Model 1



The loss graph is seen to be stable and does not have any spikes or dips in losses. The loss score for the training and validation graph are 2.0777 and 1.8477 (Figure 4.1). as expected, the training loss score was initially higher than the validation loss score. The training loss score then decreased steadily. The validation loss decreased slowly and eventually flattens and increase slightly as shown in the graph. The final training and validation loss score were 1.0900 and 1.4186 respectively (Figure 4.2).

Comparing this to GRU Base Model, the final training validation loss score of GRU Base Model was higher at 1.6064 (Figure 2.2) compared to 1.4186 (Figure 4.2) of GRU Model 1. The validation loss graph of GRU Model 1 is also far better compared to the GRU Base Model. As it is far more stable. Also comparing this model to LSTM Model 1, the final validation loss score is higher on GRU Model 1 at 1.4186 (Figure 4.2) compared to 1.4169 of LSTM Model 1 (Figure 3.2).

This model seems to be more reliable in terms of accuracy and prediction in texts. My approach to choosing the model best suited for prediction is solely based on the accuracy of the validation model. Since GRU Model 1 has the highest validation accuracy of all the models. This model would be the model I would pick to predict the texts.

GRU Model 1 has the added benefit of having a rather low validation loss score, which would increase the confidence of the model when predicting the characters. This model is also fine-tuned such that the model is stopped when the highest validation accuracy is obtained by using Early Stopping. Hence, this model was extensively fine-tuned and would be suited to be the best model to make prediction on.

```
Epoch 1/15
356436/356436 [==============================] - 37s 104us/sample - loss: 2.0777 - acc: 0.3921 - val_loss: 1.8477 - val_acc: 0.4460
```

Figure 4.1 (GRU Model 1, 1st Epoch)

```
Epoch 15/15
356436/356436 [==============================] - 35s 97us/sample - loss: 1.0990 - acc: 0.6598 - val_loss: 1.4186 - val_acc: 0.5836
```

Figure 3.1 (GRU Model 1, 15th Epoch)

Use Best Model to Make Prediction for DL Assignment 2 Part 2

The prediction input

This input was taken from Sherlock Holmes: The Complete Novel and Stories Volume 1.

The excerpt was used instead of the entire text as I did not want to have too many overly complicated words inputted. All the models use the same prediction input.

The prediction of words and characters are shown to justify my actions and show why I chose GRU Model 1 over the other models.

Encoding the input

```
# one-hot encode the user input
# Enter your code here:
new_window_size = 50

X, y = encode_io_pairs(text_input, new_window_size)
```

```
Vectorization...
Number of sequences: 10876
```

This code is used to one hot encode the input text using the function “encode_io_pairs”.

1. LSTM Base Model

```
# show the model output using predict function
# Enter your code here:
import random
import sys
import numpy as np

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


for epoch in range(1, 2):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    LSTM_base_model.fit(X, y,
              batch_size=128,
              epochs=1)

    # Select a text seed at random
    start_index = random.randint(0, len(text_input) - window_size - 1)
    generated_text = text_input[start_index: start_index + window_size]
    print('--- Generating with seed: "' + generated_text + '"')

    sys.stdout.write(generated_text)

    for i in range(15):
        sampled = np.zeros((1, window_size, len(chars)))
        for t, char in enumerate(generated_text):
            sampled[0, t, chars_to_indices[char]] = 1.

        preds = LSTM_base_model.predict(sampled, verbose=0)[0]
        next_index = sample(preds, temperature)
        next_char = chars[next_index]

        generated_text += next_char
        generated_text = generated_text[1:]

        sys.stdout.write(next_char)
        sys.stdout.flush()
        print(end="")
```

These are the parameters set to predict GRU Base Model.

```
5499/5499 [==============================] - 0s 67us/sample - loss: 1.6231 - acc: 0.5261
```

LSTM Base Model has a prediction accuracy of 52.61%. This is not that high in terms of prediction rate. Due to this accuracy, the character generated will be not that great as seen later.

```
university of london, and proceeded to netley to go throug"
```

This is the generated text from LSTM Base Model. It stops at “throug” to allow the model to predict the next character.

```
and proceeded to netley to go through me mirey and
```

And as seen in the figure above, the model predicted the correct character “h” to complete the word “through”. It also predicted the words “me” and “and”. I only allowed the model to predict 15 characters, hence not many words are predicted.

2. GRU Base Model

```
# show the model output using predict function
# Enter your code here:
import random
import sys
import numpy as np

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


for epoch in range(1, 2):
    print(' --epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    GRU_base_model.fit(X, y,
              batch_size=128,
              epochs=1)

    # Select a text seed at random
    start_index = random.randint(0, len(text_input) - window_size - 1)
    generated_text = text_input[start_index: start_index + window_size]
    print('--- Generating with seed: "' + generated_text + '"')

    #for temperature in [0.2, 0.5, 1.0, 1.2]:
        #print('------ temperature:', temperature)
    sys.stdout.write(generated_text)

    # We generate 400 characters
    for i in range(15):
        sampled = np.zeros((1, window_size, len(chars)))
        for t, char in enumerate(generated_text):
            sampled[0, t, chars_to_indices[char]] = 1.

        preds = GRU_base_model.predict(sampled, verbose=0)[0]
        next_index = sample(preds, temperature)
        next_char = chars[next_index]

        generated_text += next_char
        generated_text = generated_text[1:]

        sys.stdout.write(next_char)
        sys.stdout.flush()
        print(end="")
```

These are the parameters set to predict GRU Base Model.

```
1833/1833 [==============================] - 0s 65us/sample - loss: 1.7323 - acc: 0.4986
```

The accuracy on GRU Base Model is only 49.86% as seen from the figure above. This is also lower compared to the LSTM Base Model which has a prediction accuracy of 52.61%. This difference in accuracy could be due to the algorithm differences of LSTM and GRU networks. Both LSTM and GRU Base Model has been trained on longer sequences. This puts the GRU network at a disadvantage due to it able to produce a greater accuracy when the characters in the sequences are shorter. Thus, explaining the accuracy for GRU Base Model when compared to LSTM Base Model.

```
with a great train of wounded sufferers, to "
```

This is the text that is generated from GRU Base Model. It stops at a spacebar this time.

```
with a great train of wounded sufferers, to they, listed. n
```

The model was able to predict “they” and “listed”. This model was also only allowed to predict 15 characters, hence the short prediction.

3. LSTM Model 1

```
# show the model output using predict function
# Enter your code here:
import random
import sys
import numpy as np

def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


for epoch in range(1, 2):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    LSTM_model1.fit(X, y,
              batch_size=1024,
              epochs=1)

    # Select a text seed at random
    start_index = random.randint(0, len(text_input) - window_size - 1)
    generated_text = text_input[start_index: start_index + window_size]
    print('--- Generating with seed: "' + generated_text + '"')

    sys.stdout.write(generated_text)

    for i in range(400):
        sampled = np.zeros((1, window_size, len(chars)))
        for t, char in enumerate(generated_text):
            sampled[0, t, chars_to_indices[char]] = 1.

        preds = LSTM_model1.predict(sampled, verbose=0)[0]
        next_index = sample(preds, temperature)
        next_char = chars[next_index]

        generated_text += next_char
        generated_text = generated_text[1:]

        sys.stdout.write(next_char)
        sys.stdout.flush()
        print(end="")
```

T These are the hyperparameters set for LSTM Model 1 for the prediction.

```
10626/10626 [==============================] - 2s 206us/sample - loss: 1.5435 - acc: 0.5443
```

LSTM Model 1 has a prediction accuracy of 54.43%, which is a considerable increase compared to 52.61% from LSTM Base Model. This prediction accuracy is also not as high as I would expect. But extensive fine tuning has been done to the model but there was no difference in terms of accuracy.

```
i was so weak and emaciated that a medical board det"
```

This is the text that was generated by the model LSTM Model 1. The model stopped on the alphabet “t”.

```
i was so weak and emaciated that a medical board deticles, and he stounged my handled and examination
```

The predicted text as a continuation from the generated text was “ticles” to form a word “deticles”. This is not an actual word in the English dictionary. The other words predicted are “and”, “he”, “stounged”, “my”, “handled”, “and” and “examination”. The words predicted are all correct and valid English words, but the fact that the next character predicted was incorrect shows that there are still hyperparameters to be changed to increase the prediction accuracy of characters.

4. GRU Model 1

```
# show the model output using predict function
# Enter your code here:
import random
import sys
import numpy as np

def sample(preds, temperature=0.5):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)


for epoch in range(1, 2):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    GRU_model1.fit(X, y,
              batch_size=64,
              epochs=1)

    # Select a text seed at random
    start_index = random.randint(0, len(text_input) - window_size - 1)
    generated_text = text_input[start_index: start_index + window_size]
    print('--- Generating with seed: "' + generated_text + '"')

    sys.stdout.write(generated_text)

    for i in range(400):
        sampled = np.zeros((1, window_size, len(chars)))
        for t, char in enumerate(generated_text):
            sampled[0, t, chars_to_indices[char]] = 1.

        preds = GRU_model1.predict(sampled, verbose=0)[0]
        next_index = sample(preds, temperature)
        next_char = chars[next_index]

        generated_text += next_char
        generated_text = generated_text[1:]

        sys.stdout.write(next_char)
        sys.stdout.flush()
        print(end="")
```

These are the hyperparameters set for GRU Model 1 for the prediction.

```
10876/10876 [==============================] - 1s 90us/sample - loss: 1.5519 - acc: 0.5507
```

The prediction accuracy for GRU Model 1 has an accuracy of 55.07%. This is the highest of all the prediction accuracy of the models thus far.

```
                    i should like to meet him, i said. if i am to l"
```

This shows the characters generated by the model. The model stopped at the character "l".

```
i should like to meet him, i said. if i am to like a few company as to the bed-head in the mad a strange lamp of the approach in
```

This shows the prediction of the GRU Model 1. It continues the sentence and completes the entire sentence without an error.

5. Conclusion for Model Prediction for DL Assignment 2 Part 2

Hence, I have chosen the correct model that is best able to predict. The approach used was accuracy, but the output prediction was shown there such that it can be used to justify the case. This has also concluded that when fine-tuned accordingly, the GRU network is able to outclass the LSTM network in terms of accuracy and reliability in terms of predicting characters.

Summary of DL Assignment 2 Part 2

In summary, I have learnt from part 2 of this assignment that the LSTM and GRU networks have their own respective strengths. Each have their own pros and cons, and each problem should be handled using different models and approaches such that the problem can be solved in the shortest time possible.

Reference for DL Assignment 2 Part 2

Excerpt from Sherlock Holmes: The Complete Novels and Stories Volume I. (n.d.). Retrieved August 21, 2020, from https://www.penguinrandomhouse.ca/books/42602/sherlock-holmes-the-complete-novels-and-stories-volume-i-by-sir-arthur-conan-doyle/9780553212419/excerpt