NGEE ANN POLYTECHNIC

School of InfoComm Technology

Deep Learning Assignment

Diploma in CSF / FI / IT

April 2020 Semester

ASSIGNMENT 2

(40% of DL Module)

Submission Deadline:

Presentation: 11th Aug 2020 (Tuesday), 11:59PM

Report: 23rd Aug 2020 (Friday), 11:59PM

Tutorial Group :	~~P01~~ / **P02** / ~~P03 /P04~~
Student Name :	**NG CHIN TIONG RYAN**
Student Number :	

Table of Contents

Problem 1

1. Overview

The Problem

The main goal of this assignment is to find the mapping between sentiments in tweets to corresponding emoticons specified. As the dataset file (dataset.csv) is rather large, it would be a rather tedious and time-consuming task to do it manually. Using Recurrent Neural Networks helps to simplify this process since they are less likely to make mistakes when mapping sentiments to their corresponding emoticons as compared with humans.

The problem is a multi-class, single label type of classification as there are five possible emoticons which a sentiment can be mapped to, with the condition that each sentiment supplied can only belong to one category.

The Objective

Recurrent Neural Networks are able to pick up patterns in natural language structure. Based on the different sentiments supplied, the model should be able to better associate words with the corresponding sentiment, perhaps by using the frequency of the words that appear in the training dataset for a particular sentiment type.

The main objective of this assignment is to experiment with various Recurrent Neural Network (RNN) architectures to determine which model is the best for mapping new sentiments provided to the model and generate the most probable emoticon to represent that sentiment. In my opinion, a best model should:

- Not overfit too quickly
- Have good accuracy of above 62% (which is the average of the models)
- Be able to correctly predict most of the corresponding emoticons based on the sentiment supplied

The Approach

I chose to experiment from non-pretrain models before transitioning to using pretrained models, as from my past experience with using pretrained RNNs achieved a higher test and validation accuracies as compared with non-pretrained models. I started off with a base model for each unique model architecture type (Non-Pretrained LSTM, Non-Pretrained GRU, Pretrained LSTM and Pretrained GRU) and then tweaked the various hyperparameters accordingly in order to achieve the highest possible accuracy for each type of model. For the Pretrained Models I experimented with different embedding dimensions of 50, 100 and 200 in order to verify if this has any effect on the final test accuracy of the model.

I then supplied a list of new sentiments in order to assess the predictions of the best shortlisted model of the different architectures. Based on the results from the testing, I then went on to find out why the model does well/does not do so well in accurately predicting the most appropriate emoticon, which was decided before testing and used the criteria as described above in the objective to determine my best model.

2. Data Loading & Processing

Data Loading

The first step in this problem is to load the data from the dataset. Firstly, dataset.csv (which contains the tweets and the corresponding sentiment labels from 0 to 4) and mapping.csv (which contains the labels and the corresponding emoticons), as well as the Assignment_2_p1.ipynb must be placed in the same base directory.

```
# 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
```

Thereafter I made use of the given function as above in order to convert the mapping.csv into a dictionary with 5 key-value pairs. The key is the label for the sentiment and the value is the corresponding emoticon as shown above.

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

```
['Been friends since 7th grade. Look at us now we all following our dreams doing what we love and…\n'
 'This is what it looks like when someone loves you unconditionally oh Puppy Brother. #htx…\n'
 "RT @user this white family was invited to a Black barbecue and i've never laughed so hard in my life\n"
 ...
 'Meet Olive. Our new #GreatDane   #DogsOfDenver #Dane #DogLove @ Dream Denver\n'
 '"I talk gray, I don\'t keep it white and black" : @user @ Three Rivers Park District -…\n'
 'When his baby comes to visit. #cheflife #chefdogs #chanceboudreaux #bloodhoundpuppy…\n']
[0 1 1 ... 4 2 4]
```

Another method (.values) as shown above is used to extract all the sentiments and the corresponding labels and store them in two separate lists—one for the sentiments in the variable texts and another for the labels. This helps to simplify the process of processing later on.

```
# 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
```

Thereafter the length of each data sample is compared with one another to determine the sentiment with the greatest number of words and subsequently the word count (in this case the maximum number of words in the ‘TEXT’ column is 34) is recorded in the variable max_len and displayed to the user. This will be used supplied as an argument when tokenizing the data and the labels into vectors

Data Processing

```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

max_words = 10000

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=max_len)

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

X = data
y = labels
```

```
Found 54076 unique tokens.
Shape of data tensor: (42546, 34)
Shape of label tensor: (42546,)
```

A tokenizer is used in this step to tokenize the words into numeric vectors. This is essential because the models can only be trained with numeric data and thus there is a need to map the individual words in the texts list to a numeric tensor in order for the end product to be a 2D Tensor (shown later), which can then be fed into the input embedding layer.

42456 in this case represents the number of records present in the dataset.csv file (excluding the header row). It is 42456 x 34 for the data since the data is already padded with zeros until there are 34 integers are in the data array and 42456 x 1 for the labels since there is only 1 integer for each of the labels and thus it is a 1D array.

Code Breakdown and Explanations

The tokenizer breaks down the dataset into words and identifying all of the unique words using the fit_on_texts method and thereafter text_to_sequences converts the word to the corresponding index if it exists in the list of 10,000 words.

The output below shows the result of printing sequences[0] (result of text_to_sequences):

```
[145, 95, 539, 1913, 2063, 91, 17, 96, 79, 25, 31, 2322, 41, 744, 381, 58, 25, 15, 263]
```

For instance, the first example has 19 unique words, where each of the word indexes as specified above.

word_index allows us to map the words to the indexes as a dictionary of key-value pairs as shown below. This allows the model to be able to retrieve the words later.


```
{'the': 1, 'user': 2, '': 3, 'my': 4, 'i': 5, 'to': 6, 'a': 7, 'and': 8, 'this': 9, 'in': 10, 'you': 11, 'of': 12, 'with': 13, 'for': 14, 'love': 15, '
is': 16, 'at': 17, '…': 18, 'new': 19, 'on': 20, 'so': 21, 'me': 22, 'amp': 23, 'it': 24, 'we': 25, 'happy': 26, 'was': 27, 'day': 28, 'be': 29, 'that
': 30, 'all': 31, 'when': 32, 'your': 33, 'from': 34, 'out': 35, 'night': 36, 'just': 37, 'are': 38, 'one': 39, 'by': 40, 'our': 41, 'york': 42, 'have
': 43, 'today': 44, 'up': 45, 'time': 46, 'but': 47, 'best': 48, 'these': 49, "i'm": 50, 'park': 51, 'like': 52, 'get': 53, 'birthday': 54, 'beautiful
': 55, 'last': 56, 'got': 57, 'what': 58, 'california': 59, 'good': 60, 'beach': 61, "it's": 62, 'family': 63, 'see': 64, 'little': 65, 'great': 66, 'm
uch': 67, 'city': 68, 'her': 69, 'thank': 70, 'some': 71, 'favorite': 72, 'life': 73, 'back': 74, 'thanks': 75, 'had': 76, 'center': 77, 'not': 78, 'no
w': 79, 'do': 80, 'here': 81, 'tonight': 82, 'first': 83, 'amazing': 84, 'home': 85, 'university': 86, 'how': 87, 'girl': 88, 'always': 89, 'go': 90, '
look': 91, 'more': 92, 'he': 93, 'as': 94, 'friends': 95, 'us': 96, 'an': 97, 'fun': 98, 'she': 99, 'texas': 100, 'repost': 101, 'lol': 102, 'can': 10
3, "can't": 104, 'no': 105, 'if': 106, 'about': 107, 'nyc': 108, 'who': 109, 'school': 110, 'heart': 111, 'know': 112, 'morning': 113, 'baby': 114, 'ch
ristmas': 115, 'weekend': 116, "don't": 117, 'people': 118, 'they': 119, 'year': 120, 'his': 121, 'come': 122, '2': 123, 'two': 124, 'w': 125, 'friend
': 126, 'world': 127, 'san': 128, 'show': 129, 'florida': 130, 'too': 131, 'lit': 132, 'high': 133, 'will': 134, 'house': 135, 'lake': 136, 'right': 13
7, 'fire': 138, 'place': 139, 'los': 140, 'off': 141, 'ever': 142, 'state': 143, 'only': 144, 'been': 145, 'big': 146, 'has': 147, 'never': 148, 'still
': 149, 'make': 150, 'angeles': 151, 'the…': 152, 'chicago': 153, 'work': 154, 'tbt': 155, 'way': 156, 'made': 157, 'or': 158, 'girls': 159, 'south': 1
60, 'miss': 161, 'bar': 162, '1': 163, 'man': 164, 'dinner': 165, 'music': 166, 'him': 167, 'than': 168, 'there': 169, 'take': 170, 'date': 171, 'bette
r': 172, 'ready': 173, 'another': 174, 'am': 175, 'them': 176, 'live': 177, 'photo': 178, 'hot': 179, 'oh': 180, 'guy': 181, 'old': 182, 'miami': 183,
```


pad_sequences allows us to pad the list of words with 0s until the list contains 34 integers (as defined in max_len). This the first example will become:

```
[    0     0     0     0     0     0     0     0     0     0     0     0     0     0
     0   145    95   539  1913  2063    91    17    96    79    25    31  2322    41
   744   381    58    25    15   263]
```

Thereafter, the labels were also transformed into a list. Following that I proceeded to explicitly define **X** (the data) and **y** (the labels) for the purpose of later use.

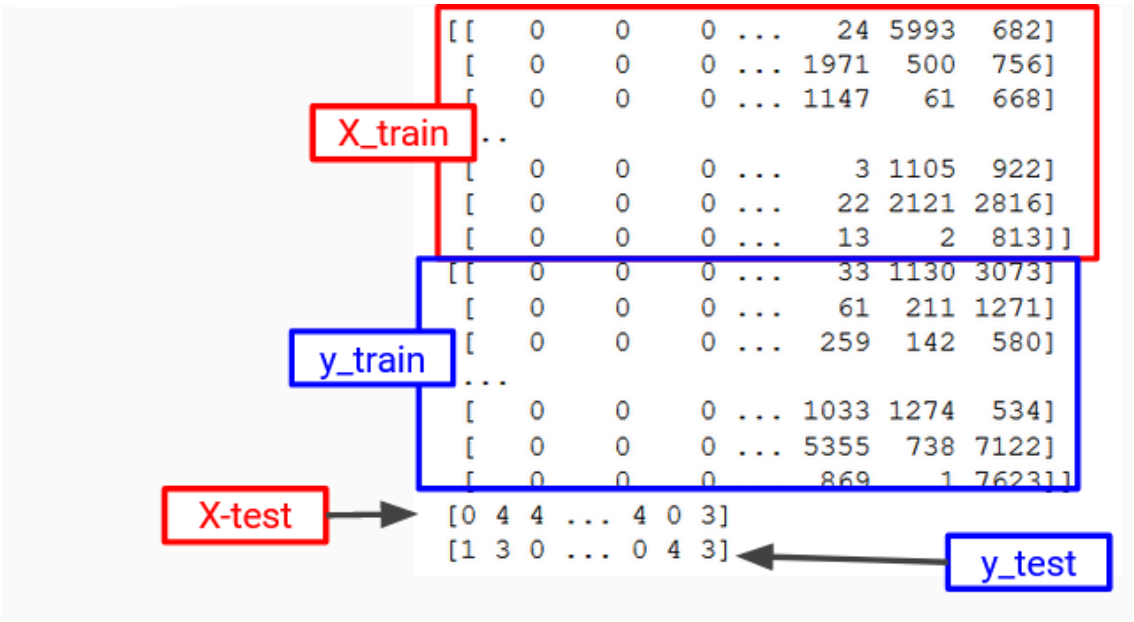
Data Sampling

```
# Split the X & y into train and test sets
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 = 40)
```

The dataset is thereafter further split into X_train, X_test, y_train and y_test with a test split of 0.2 (i.e. 80% of the dataset will be set aside for training and the remaining 20% will be used for testing). The random state provided is a seed to the pseudorandom generator). I also initialized the random state, which ensures that the random split will always be the same and guarantees that some of the training data will not become the test data if I were to rerun the training again. This is important in ensuring that all the models get trained on the same training data and assessed by the same testing data to ensure fair experiments as using a different set of training or testing data might have an impact on the test accuracy of my models and since this is one of my evaluation criteria, I should ensure that there is fairness in both testing and training of the models.

Random sampling in this case is important to ensure an even spread of the data across both training and test datasets, so that this will not skew the results of the neural network training. For example, if the dataset had all the emoji 0 and 1 labels and data at the top and all the emoji 4 labels and data at the bottom (i.e. alphabetically sorted by labels), the model may do very poorly on the test set because it is trained on and biased towards the upper portion of the dataset.

After the sampling is conducted, the data will be split as follows:



3. Develop the Sentiment Analysis Models using Training Data and Model Evaluation using Testing Data

All the models consisted of an input embedding layer, which takes care of the word embedding for me. For the non-pretrained models, I kept the embedding dimension the same at 50. However, for the Pretrained Models, since I used the glove Pretrained word embeddings, I had the option of using 50, 100 and 200 as dimensions to experiment with.

Model #1 NonPT LSTM RMSprop Base

For the first model, I used a Long-Short Term Memory (LSTM) model with an embedding input layer. max_words is specified as an argument here to specify the number of unique tokens, which in this case is 10,000 as specified earlier. embedding _dim represents the dimensionality of the embedding supplied to the model for training, which in this case will be 50 for all the non-pretrained models and will vary (50, 100 or 200) for pretrained models. Input_length is specified to be max_len, which allows for the shape of the Dense layers to be computed and in this case, is 34.

```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

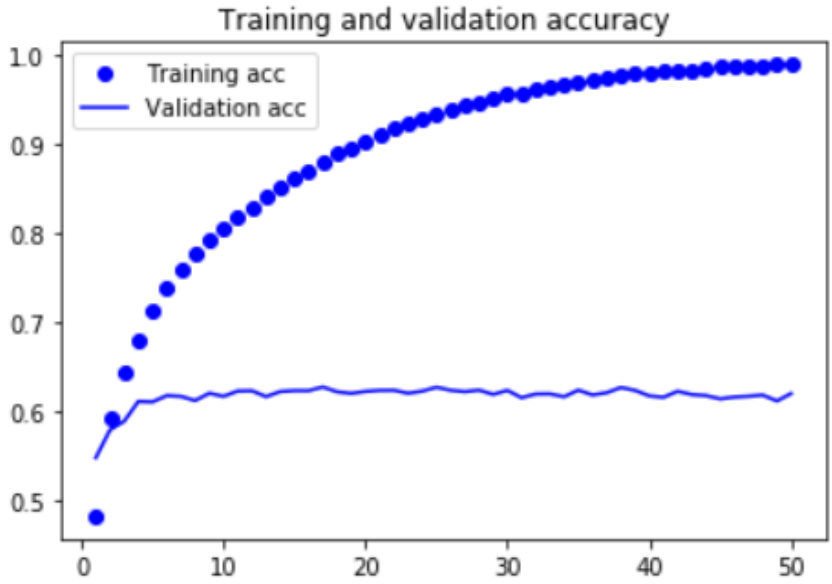
I also used 2 LSTM layers of 32 nodes each, as well as a last dense layer of 5 nodes since there are 5 possible labels, hence the model should be built such that the probabilities of the 5 emoticons can be later predicted in the evaluation section. The argument ‘return sequences = True’ is required for all but the last LSTM and Dense layers because it enables the output to be fed in as input to the next LSTM layer. A softmax activation function is used in this case since the problem is a multi-class, single label type of classification.

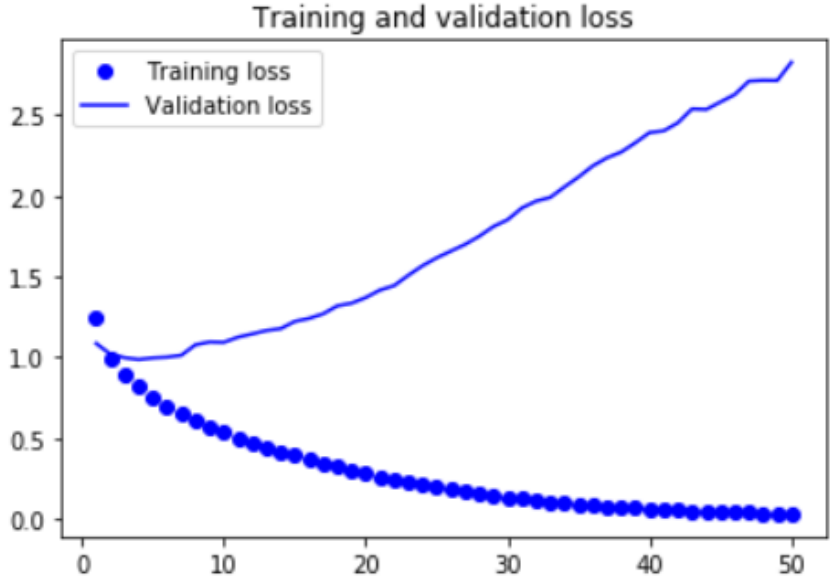
```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=50,
                    batch_size=128,
                    validation_split=0.2)
```

I used a Root Mean Squared Propagation (RMSprop) optimizer for this first base model, with a loss function of sparse categorical crossentropy (scce). This loss function outputs a single probability for a given target index instead of giving a list of probability values (unlike what

categorical crossentropy does for one-hot encoded labels—which is a list of probability values). Since I am intending to word embed the data, the usage of sparse categorical crossentropy is appropriate. In addition, my research suggests that scce should be used when the classes are mutually exclusive, which is true in this case.

A metric of accuracy is used since this is the primary way which we are using to assess the model's performance. In the model.fit() function, I specified the training data and labels—X_train, y_train, with a batch size of 128 and a validation split of 0.2 (80% will go to training the data and 20% will perform validation whilst the model is being trained). I ran the model for 50 epochs as shown in the graphs below.





The model overfits quite rapidly at around 4 epochs and thereafter the validation accuracy plateaus at about 62%, with the training accuracy plateauing later at about 40 epochs and achieving a training accuracy of almost 100%. The loss curve also shot up rather rapidly after plateauing at around 5 epochs.

To evaluate the model, I made use of the model.evaluate() function included in the Keras library as follows, parsing in the testing data and labels as well as verbose = 2, which will make the output display more information (more verbose output as the name implies).

```
results = model.evaluate(X_test, y_test, verbose=2)
results # returns the loss value and accuracy
```

The model achieved a final test accuracy of 62.75% and a loss score of 2.77.

```
8510/8510 - 2s - loss: 2.7668 - acc: 0.6275

[2.7667604046338594, 0.6274971]
```

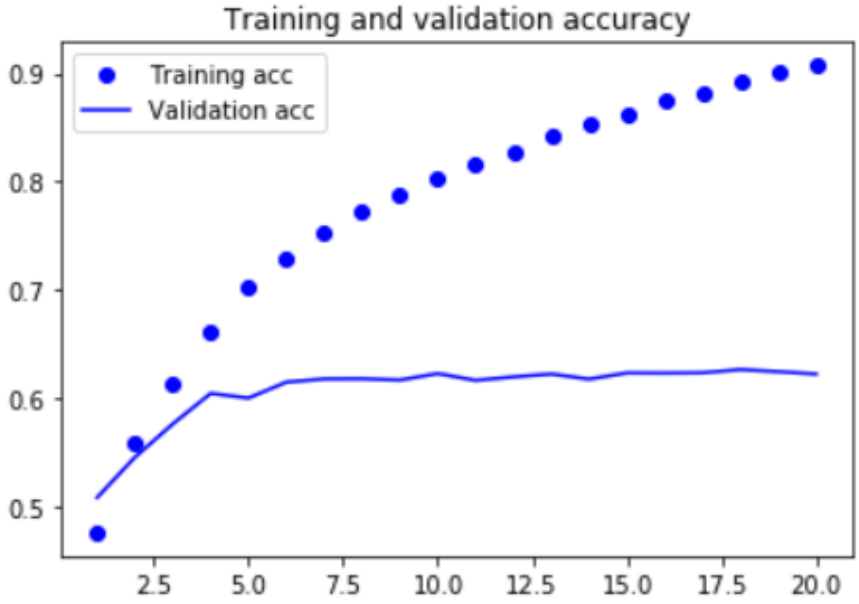
Model #2 NonPT LSTM RMSprop Add 1 layer

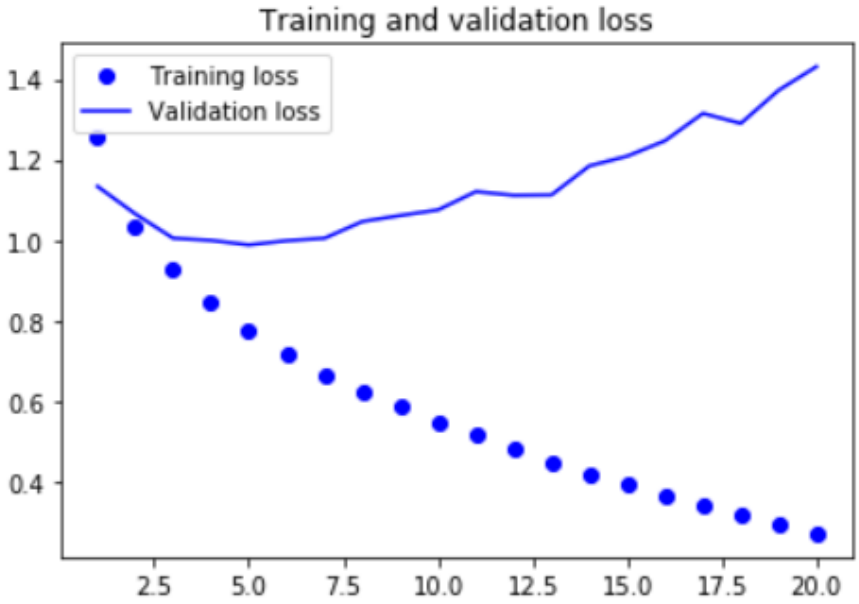
For the next model, I decided to experiment with adding more layers to see if this would increase the model's accuracy. Again, I used 50 as the embedding dimension. For this model, I included a second LSTM layer with 64 nodes because I thought that this would help to improve the accuracy since this would allow the model to pick up more patterns and complex features of the training data.

```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(64, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

I also decided to decrease the number of nodes in accordance with the Universal Workflow of Machine Learning, since the previous model had already overfitted quite early on, as well as to save computing power in training of the future models. Besides those, I kept the rest of the hyperparameters the same.

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2)
```





However, the model did not perform as expected. It overfitted still at 4 epochs, just like the previous model and actually achieved a lower test accuracy of 62.59%. The loss curve plateaus at around 5 epochs before starting to rise again gradually at 8 epochs and the final loss score of this curve is 1.4174, which is significantly lower than that of the previous model.

```
8510/8510 - 2s - loss: 1.4174 - acc: 0.6259

[1.4173936721160745, 0.6258519]
```

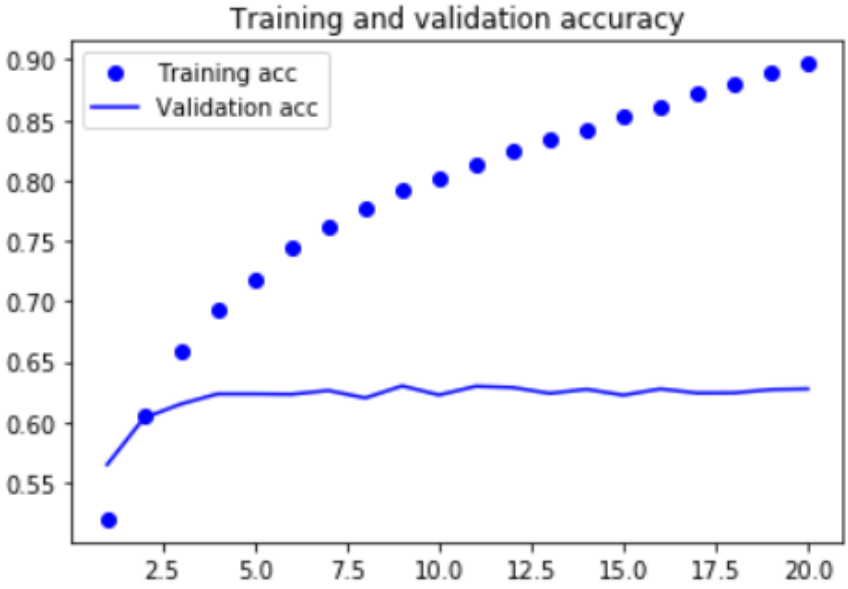
Model #3 NonPT LSTM RMSprop Add L2 reg; change batch size

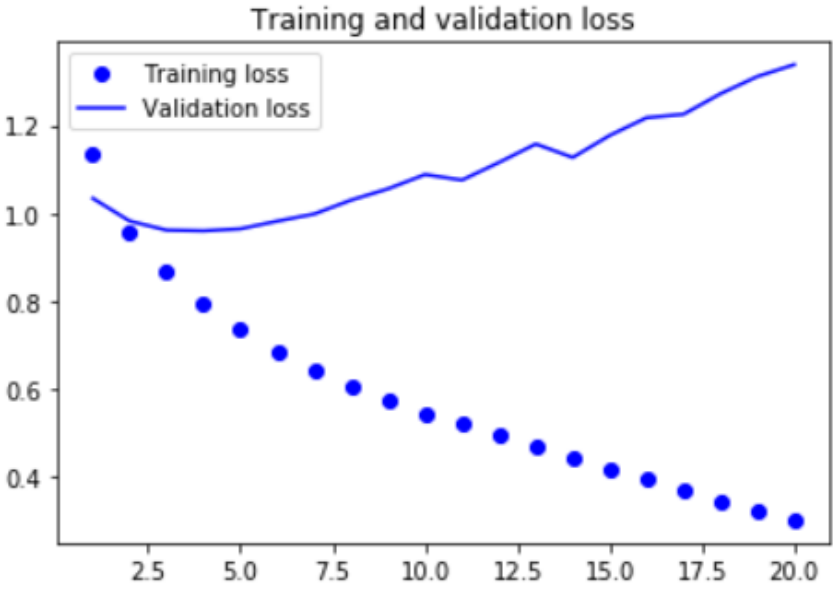
For this model, I chose to instead increase the nodes at the last LSTM layer from 32 to 64. I also added an L2 recurrent regularizer with a magnitude of 1e-3, with the hope that this could help to slow down overfitting of the model.

```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True, recurrent_regularizer=regularizers.l2(0.001)))
model.add(LSTM(64))
model.add(Dense(5, activation='softmax'))
model.summary()
```

I also decreased the batch size down to 32 from the initial 128 in order to assess the effect of this hyperparameter on the final test accuracy of the model.

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2)
```





The model overfitted again at around 4 epochs so I concluded that perhaps the regularization did not help much in this aspect. The model started plateauing after 5 epochs with a validation accuracy of around 62%. The validation loss also rises more sharply than the previous model after 6 epochs. The model achieved a final test accuracy of 62.54%, which is a deprovement from the previous 62.59% and the initial base model score of 62.75%. However, the model achieved a lower loss score of 1.3138, down from the previous 1.4174.

```
8510/8510 - 3s - loss: 1.3138 - acc: 0.6254

[1.3138317877761625, 0.6253819]
```

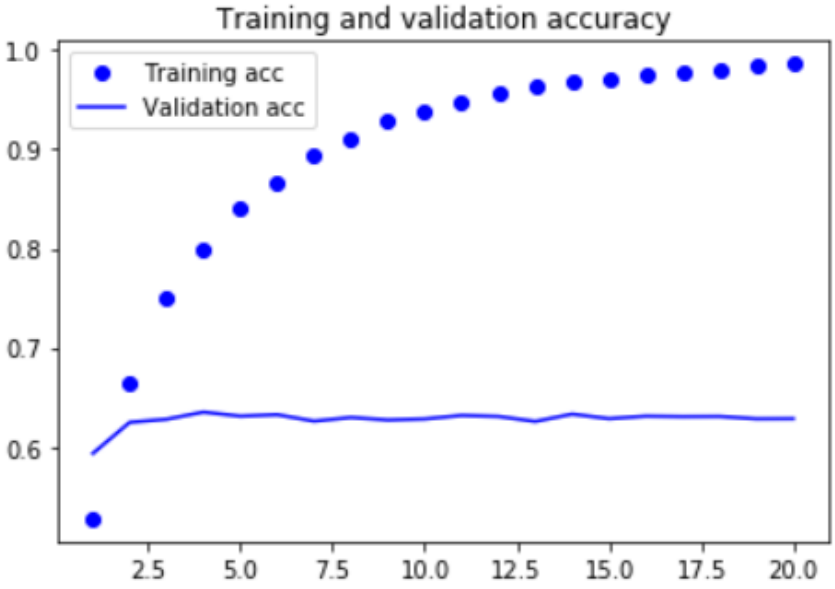
Model #4 NonPT LSTM Adam Base

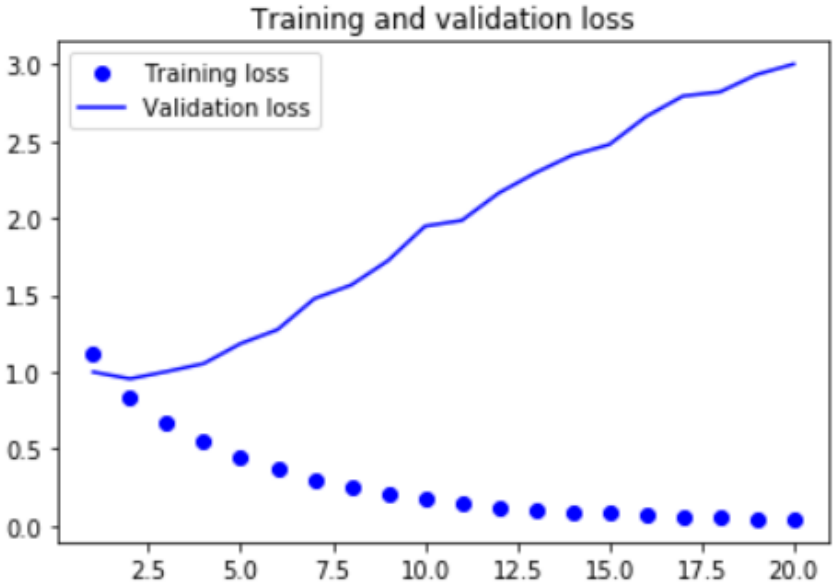
Thereafter, I decided to experiment with a different optimiser. In this case I used the Adam optimiser because my research suggested that it has a reputation of achieving higher test accuracies but at the same time tends to overfit more quickly. For this model I used a similar model architecture to the LSTM RMSprop Base Model, with an embedding dimension of 50.

```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

I also chose to keep the batch size at 32. In addition, I decided to just run to the Adam base model for 20 epochs since I already expected the model to overfit rather quickly.

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=32,
                    validation_split=0.2)
```





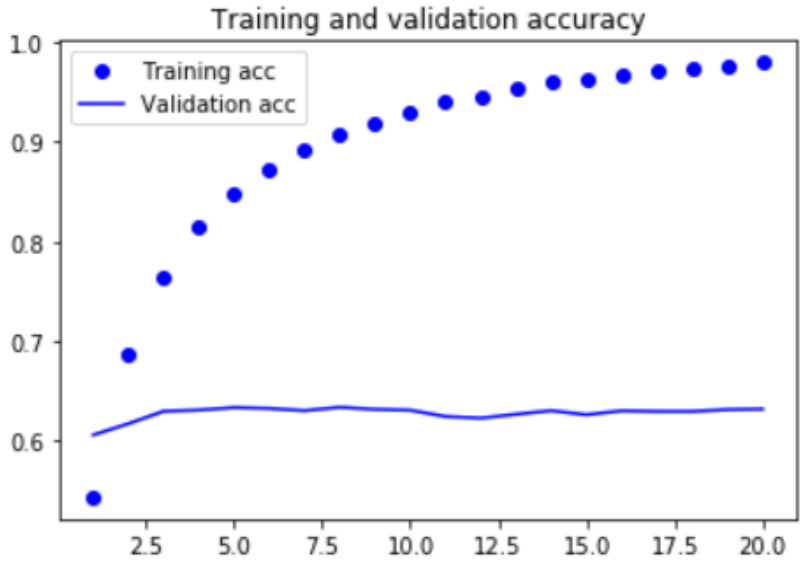
As expected, the model overfits rather quickly at around 3 epochs and the validation accuracy plateaus at around 63%, although the training accuracy continues to increase until almost 100%. The validation loss starts rising quite rapidly after 2 epochs, with the model achieving a final loss score of 2.9142, which is higher than the RMSprop base model's final loss. This model also achieved a higher test accuracy of 63.36% up from the RMSprop's 62.75%.

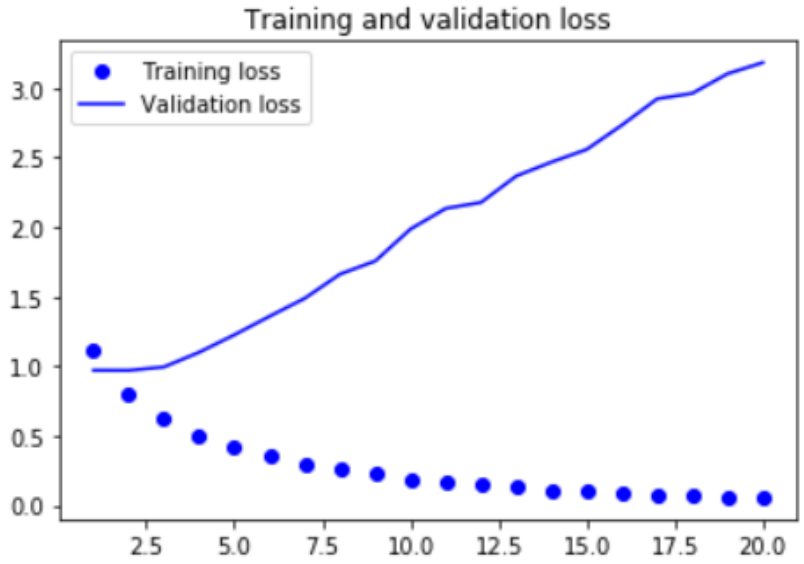
```
8510/8510 - 2s - loss: 2.9142 - acc: 0.6336

[2.9142338340907203, 0.6336075]
```

Model #5 NonPT LSTM Adam Add L2 reg

In this model, I added L2 regularization with a magnitude of 1e-2 in order to assess the impact on this hyperparameter on the final test accuracy of the model. Besides that, all the other hyperparameters, including the batch size and number of epochs that the model had were kept constant.

```
# Build the Model
embedding_dim = 100

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True, recurrent_regularizer=regularizers.l2(0.01)))
model.add(LSTM(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```





This model still overfitted at around 3 epochs despite the regularization put into place. It also achieved a worse test accuracy as compared with the Adam Base model of 62.94%, down from 63.36%. The model also had a higher loss score of 3.1623, up from the previous model's score of 2.9142, with the validation loss rising quite rapidly after the model overfits.

```
8510/8510 - 1s - loss: 3.1623 - acc: 0.6294

[3.162336896085011, 0.6293772]
```

Thus, I concluded that I might need to experiment with other hyperparameters to get an idea of how to tune the model to achieve the highest test accuracy possible.

Model #6 NonPT LSTM Adam 1 layer; increase batch size

Since the regularization did not help to mitigate underfitting or improve the test accuracy as seen in the previous model, I decided to play around with different hyperparameters. For this model, I decided to remove one LSTM layer and decrease the number of nodes in the remaining LSTM layer, which still has L2 regularization in place. I also increased the training batch size to 1024, with the hope of decreasing the time spent on training per epoch.

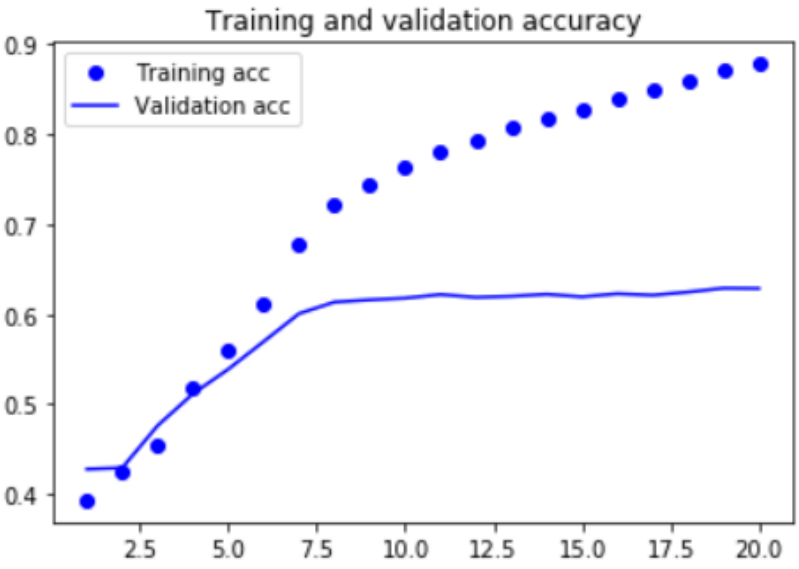
```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(16, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.summary()
```

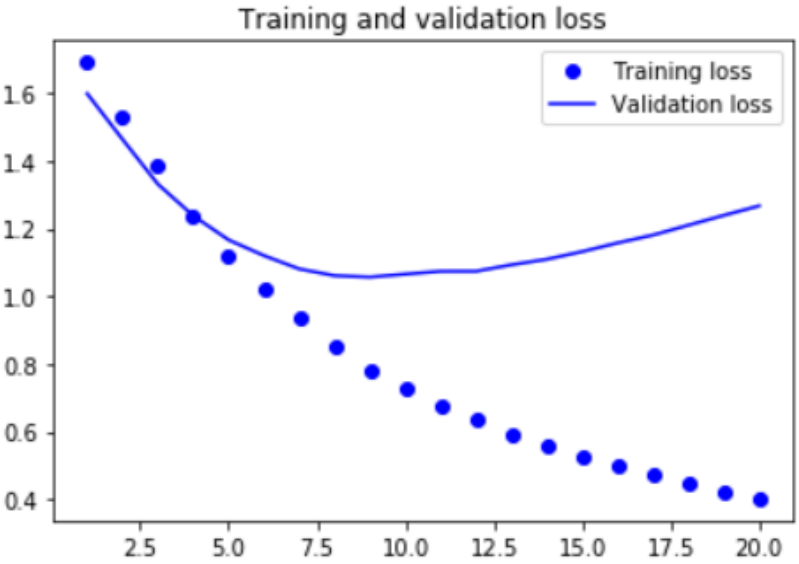
```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=1024,
                    validation_split=0.2)
```

```
Epoch 2/20
27228/27228 [==============================] - 13s 490us/sample - loss: 0.8030 - acc: 0.6878 - val_loss: 0.9703 - v
al_acc: 0.6175
Epoch 3/20
27228/27228 [==============================] - 14s 517us/sample - loss: 0.6287 - acc: 0.7645 - val_loss: 0.9948 - v
al_acc: 0.6300
Epoch 4/20
27228/27228 [==============================] - 14s 517us/sample - loss: 0.5076 - acc: 0.8138 - val_loss: 1.1003 - v
al_acc: 0.6312
Epoch 5/20
27228/27228 [==============================] - 15s 534us/sample - loss: 0.4193 - acc: 0.8484 - val_loss: 1.2257 - v
al_acc: 0.6337
Epoch 6/20
27228/27228 [==============================] - 15s 543us/sample - loss: 0.3517 - acc: 0.8712 - val_loss: 1.3589 - v
al_acc: 0.6328
```

As you can see the model trains much faster as compared to the previous model with a batch size of 32 (above), which averages about 14 seconds per epoch, as opposed to the current model (below), which averages 1 second per epoch.

```
Epoch 2/20
27228/27228 [==============================] - 1s 41us/sample - loss: 1.5282 - acc: 0.4256 - val_loss: 1.4633 - val
_acc: 0.4295
Epoch 3/20
27228/27228 [==============================] - 1s 45us/sample - loss: 1.3851 - acc: 0.4548 - val_loss: 1.3310 - val
_acc: 0.4764
Epoch 4/20
27228/27228 [==============================] - 1s 44us/sample - loss: 1.2349 - acc: 0.5190 - val_loss: 1.2377 - val
_acc: 0.5118
Epoch 5/20
27228/27228 [==============================] - 1s 43us/sample - loss: 1.1205 - acc: 0.5591 - val_loss: 1.1662 - val
_acc: 0.5391
Epoch 6/20
27228/27228 [==============================] - 1s 40us/sample - loss: 1.0229 - acc: 0.6116 - val_loss: 1.1199 - val
_acc: 0.5698
```





This model overfitted much later at around 7 epochs, which I would consider an accomplishment compared to the previous models which overfitted in the range of 2 to 4 epochs. The validation accuracy plateaus after 8 epochs at around 63% and it reaches a final test accuracy of 63.41%. As for the validation loss, it plateaus after 8 epochs and then starts to rise gradually again after 13 epochs and achieves a final loss score of 1.2497.

As mentioned earlier, the model also takes a much shorter time to train as compared to the other models, which greatly reduces the computational power required for training.

This model exceeded my expectations in the aspects of achieving a higher test accuracy and a low loss score and thus will be one of the models I would evaluate later on as the best model as a representative for the Non-Pretrained LSTM architectures.

```
8510/8510 - 2s - loss: 1.2497 - acc: 0.6341

[1.2497192720688888, 0.63407755]
```

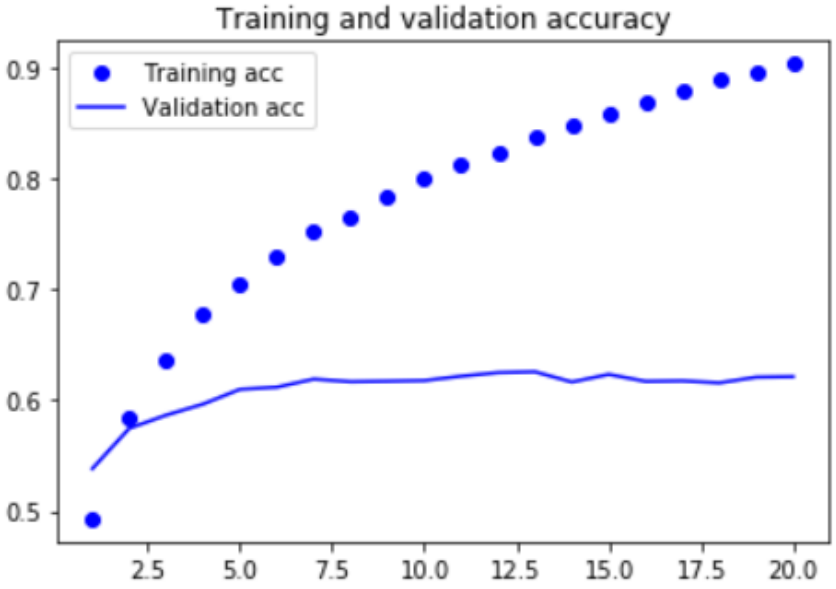
Model #7 NonPT GRU RMSprop Base

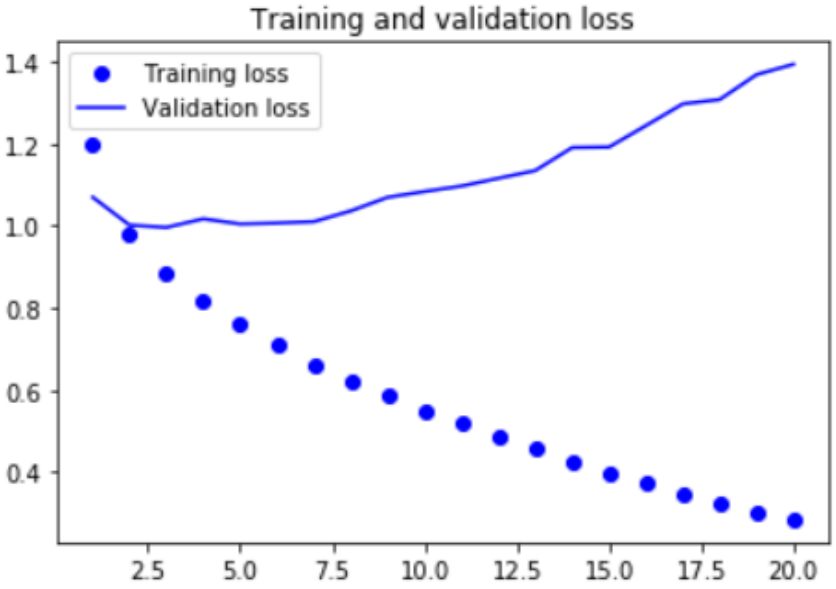
Thereafter I moved on to experiment with Non-Pretrained Gated Recurrent Unit (GRU) architectures. For this set of neural network architectures, I used an embedding dimension of 50 as I did with the Non-Pretrained LSTMs. The input parameters are the same as the LSTM layer since both architecture types use the same embedding layer to embed the data. I used 2 layers of 32-node GRUs.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, return_sequences=True))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

For this base model, I used an RMSprop optimiser with a batch size of 128 and ran the model for 20 epochs.

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2)
```





The model overfitted more quickly as compared with the Non-Pretrained LSTM Base Model with the RMSprop optimiser at 2 epochs, as opposed to the latter's 4 epochs. It also achieved a lower test accuracy compared to both the Non-Pretrained LSTM RMSprop and Adam Base Models, with a final test accuracy of 62.44%. The final loss score of the curve is 1.3792, which is significantly lower than the loss scores of previous base models, which were all above 2.0.

```
8510/8510 - 3s - loss: 1.3792 - acc: 0.6244

[1.3791578104857412, 0.62444186]
```

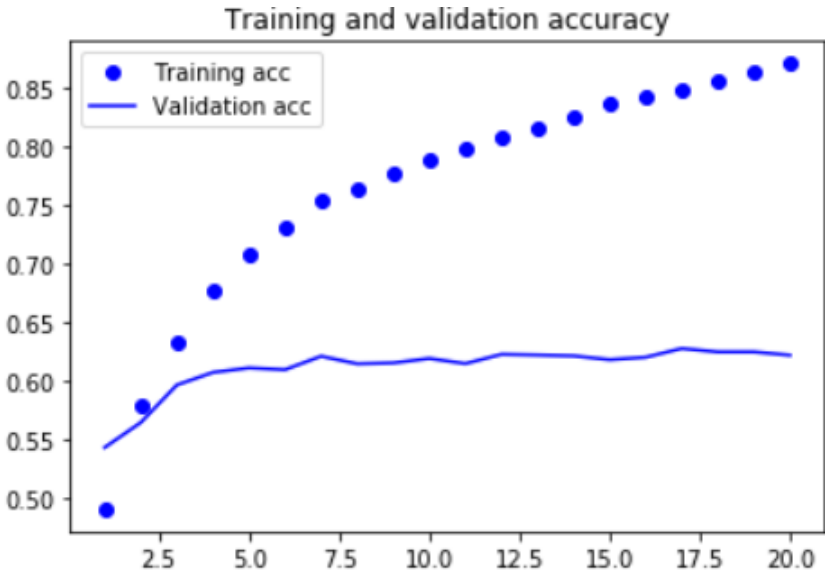
Model #8 NonPT GRU RMSprop add dropout

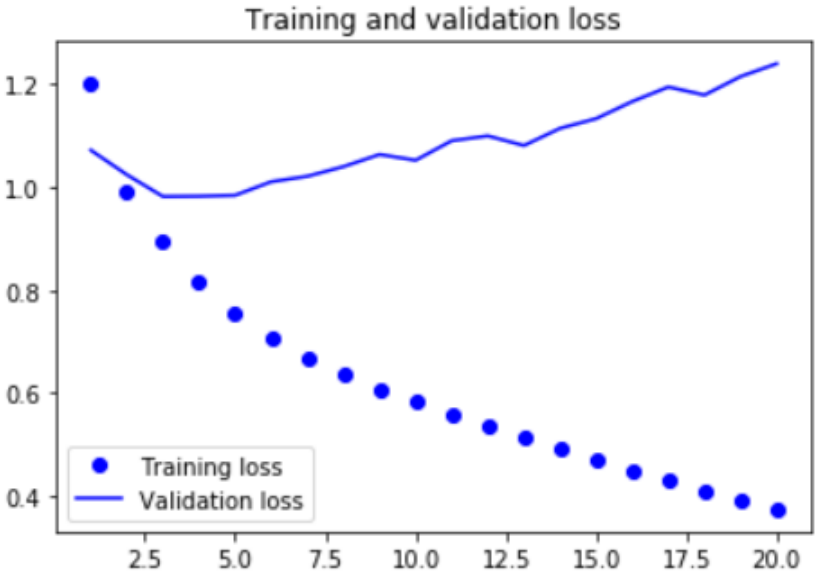
With the experience of regularization not working out for the LSTM architectures, I decided to instead experiment with dropout for the GRU-based models. For this model, I added one additional layer and applied dropout, as well as recurrent dropout to 2 of the 3 LSTM layers, with the other hyperparameters such as the batch size and the number of epochs kept constant. Both the magnitudes of the dropout and recurrent dropout were set to 0.1 for the purpose of preliminary testing and I planned to subsequently increased them in the upcoming models.

Reasons for using Recurrent dropout

While dropout is applied to the individual nodes of the layers, recurrent dropout is applied on the output of the nodes, which allows the model's performance to be improved without compromising the ability to capture long-term patterns. There patterns are a primary characteristic of Recurrent Neural Network Architectures like GRUs and I initially hoped that this would help improve model performance (G, 2018).

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```





I did see a slight improvement for this model in terms of a small reduction of the loss score by about 0.1 to 1.2268 and a slight increase in accuracy to 62.56%, up from the previous 62.44% in the base model. The rate at which the model overfitted was also enhanced slightly to 3 epochs from the previous 2 epochs.

```
8510/8510 - 6s - loss: 1.2268 - acc: 0.6256

[1.226808656592767, 0.6256169]
```

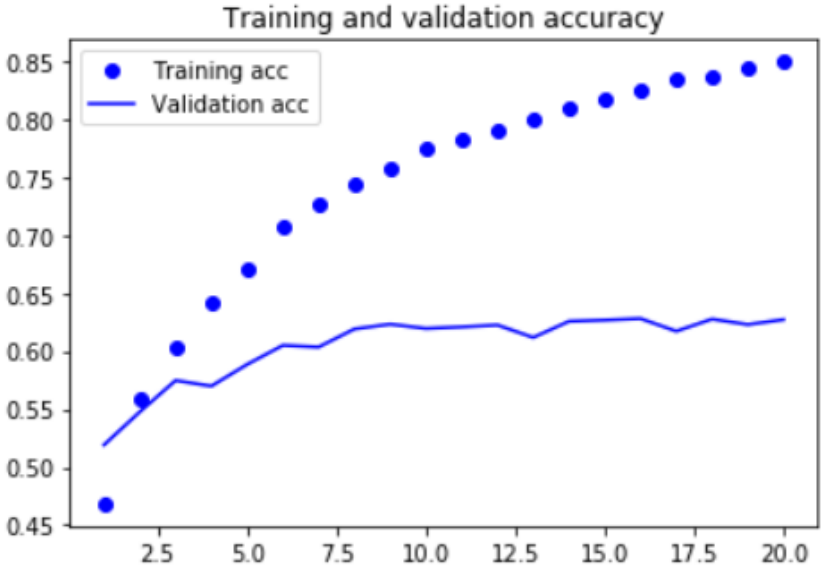
Model #9 NonPT GRU RMSprop increase dropout; change batch size

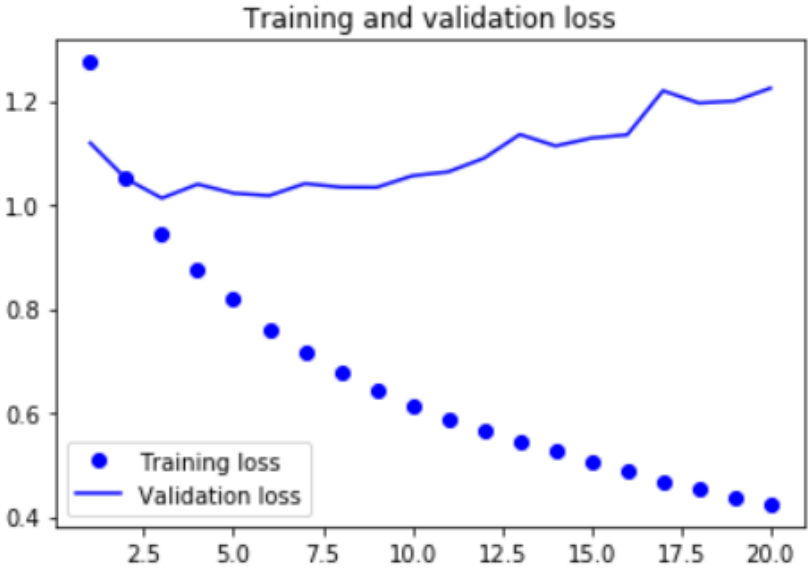
Based on my past experience with the LSTM architecture, I found that when I decreased the number of layers and increased the batch size, this would lead to a better test accuracy and the model overfitting later so I decided to experiment if using similar hyperparameter tuning would also lead to better results in the GRU architectures. For this model, I used 2 layers of GRU with 32 nodes each and increased both the dropout and recurrent dropout intensities to 0.2 up from 0.1, in the hope that the model will overfit less quickly.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

I also used a batch size of 256, up from the previous 128 in the base model and ran the model for 20 epochs.

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=256,
                    validation_split=0.2)
```





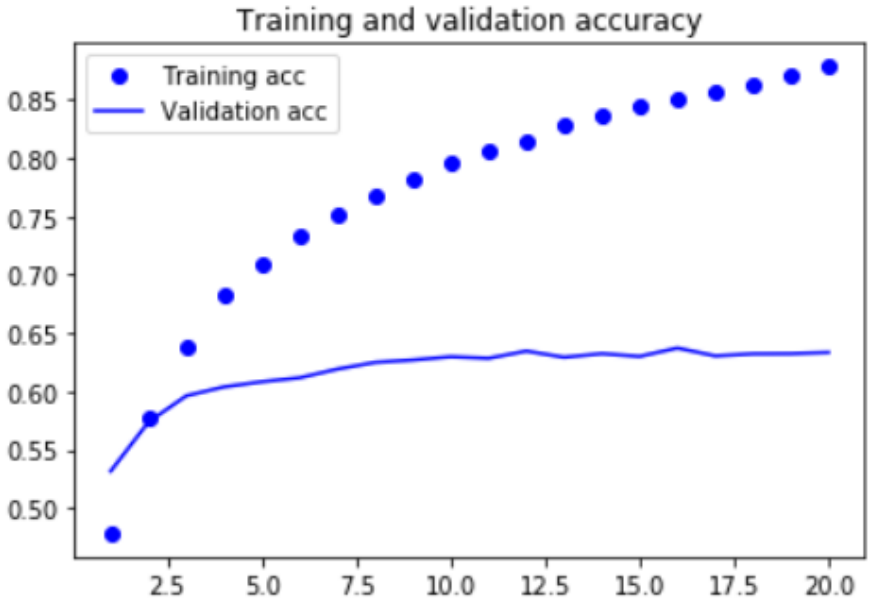
The model did not perform as expected as it achieved an even lower test accuracy of 61.77%, which is way worse than most of the models I have trained thus far. However, the model overfits at around 3 epochs this time round and its test loss score is also much lower as compared with previous models, at 1.2148.

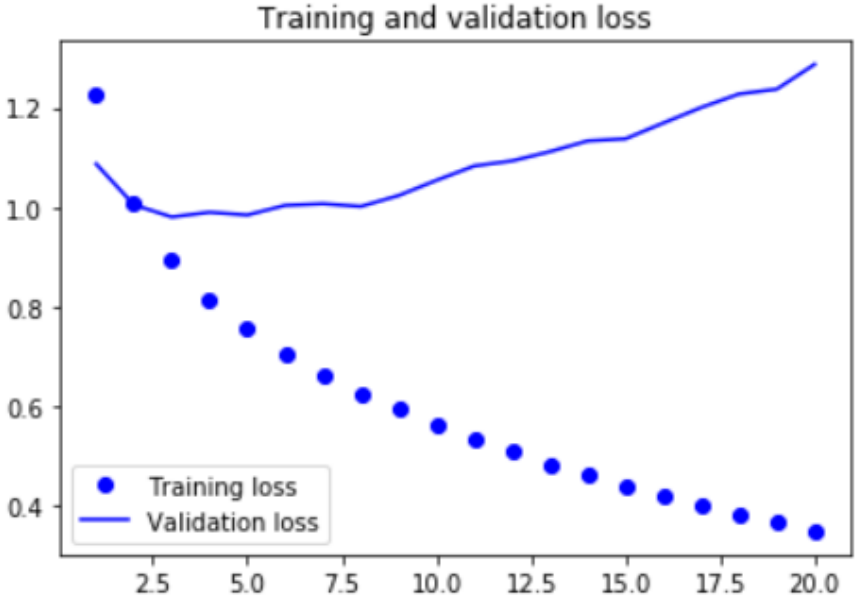
```
8510/8510 - 4s - loss: 1.2148 - acc: 0.6177

[1.2147616273507948, 0.61774385]
```

Model #10 NonPT GRU RMSprop increase nodes

Since decreasing the number of layers did not help, I tried to instead increase the number of nodes for the sole GRU layer from 32 to 64. I also used dropout and recurrent dropout with a magnitude of 0.1 on this layer, with the rest of the hyperparameters kept constant as the previous model.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(64, dropout = 0.1, recurrent_dropout = 0.1))
model.add(Dense(5, activation='softmax'))
model.summary()
```





The model now overfits at 3 epochs with the highest final test accuracy of the RMSprop-based GRU architectures of 62.88% but achieved a higher test loss score of 1.2677, which is higher the previous 2 iterations of fine tuning (1.2268 and 1.2148 respectively). Since most of the hyperparameters I tuned did not lead to a significant improvement of the model's performance by increasing the test accuracy by more than 1%, I decided to transition to experimenting with the Adam optimiser, hoping that I could achieve better results.

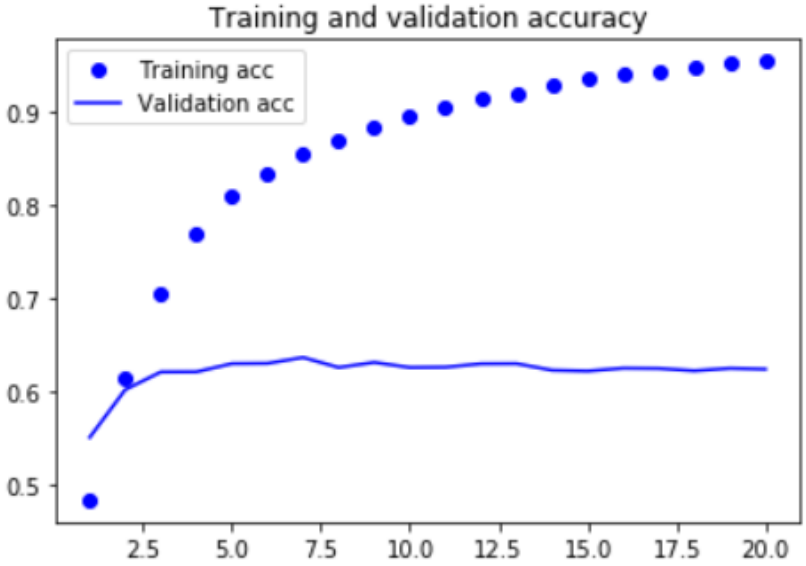
```
8510/8510 - 4s - loss: 1.2677 - acc: 0.6288

[1.2676673594989172, 0.62878966]
```

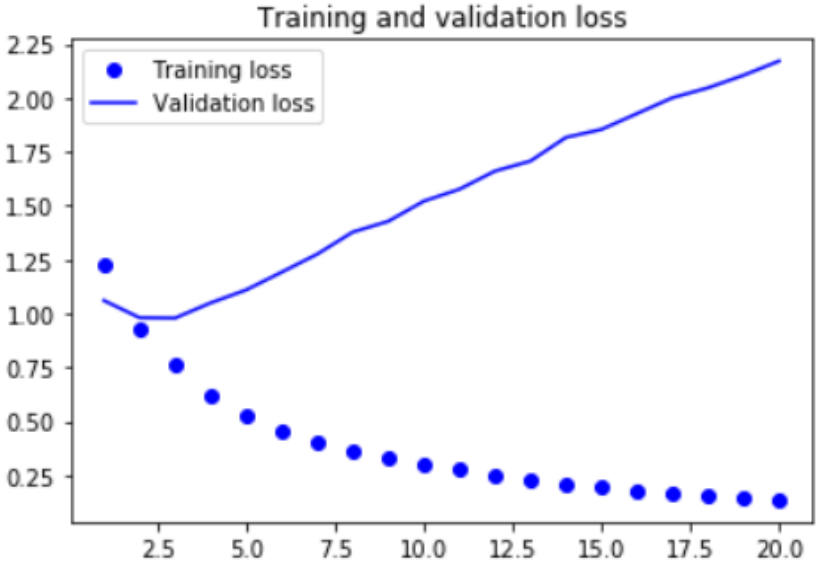
Model #11 NonPT GRU Adam Base

Besides using the Adam optimizer, I decided to keep the rest of the model architecture (number of layers and nodes, embedding dimension) and hyperparameters (batch size, epochs etc.) the same as the previous base models in order to assess whether the test accuracy is drastically impacted by the use of other optimizers besides RMSprop.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, return_sequences=True))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.summary()
```

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2)
```





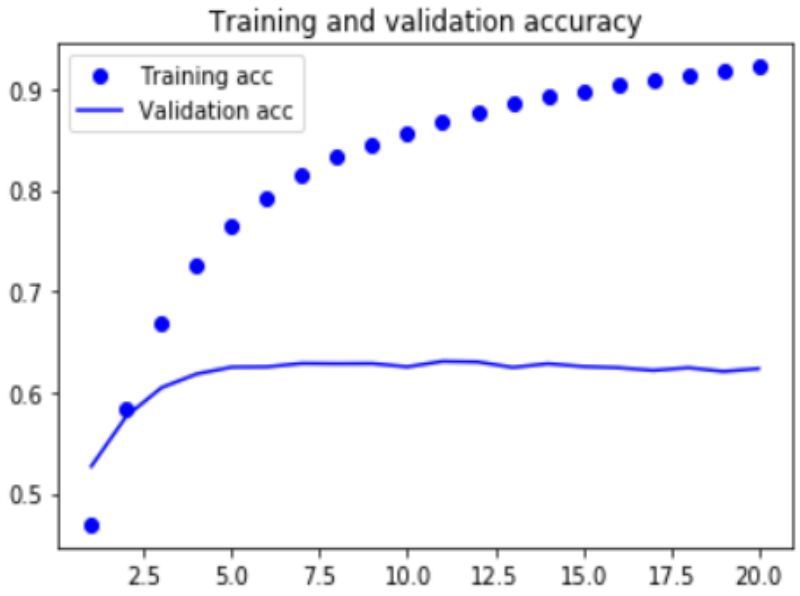
This model overfitted at 3 epochs, similar to the RMSprop Base model. However, it recorded a higher test accuracy of 63.03%, which is about 0.5% more than the GRU RMSprop Base. However, this came at a cost of a higher test loss score of 2.0838, up from 1.3792. The increase in the loss is also rather rapid after 3 epochs, as the loss score rises at a rate of approximately 0.07 per epoch.

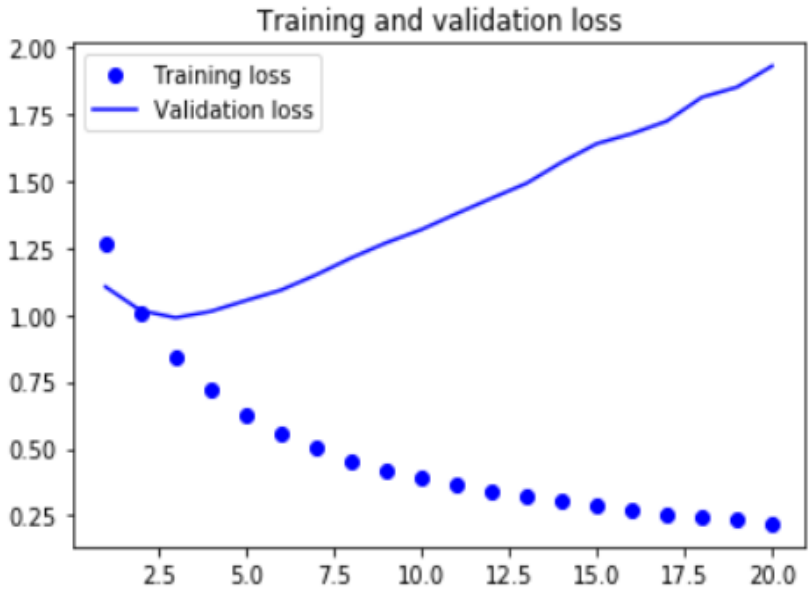
```
8510/8510 - 4s - loss: 2.0838 - acc: 0.6303

[2.083842955072674, 0.6303173]
```

Model #12 NonPT GRU Adam increase dropout

I then decided to reduce the number of GRU layers from 2 to 1 and also increased the magnitude of dropout to 0.2, with the rest of the hyperparameters kept constant from the GRU Adam Base model. I hoped that with these changes made, this model will be able to outperform the base model in terms of test accuracy.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

embedding_dim = 50
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, dropout = 0.2, recurrent_dropout = 0.2))
model.add(Dense(5, activation='softmax'))
model.summary()
```





This model did not meet my expectations and overfitted at around 3 epochs. It also had a poorer test accuracy of 62.29%, which is a drop from the previous 62.3%. I felt that I had already experimented with many Non-Pretrained GRUs and since there were not many significant improvements in accuracy, I would just move on to experiment with non-Pretrained models and select the highest performing Non-Pretrain Adam GRU as my best model for these set of architectures.

```
8510/8510 - 4s - loss: 1.8981 - acc: 0.6229

[1.898114839563919, 0.6229142]
```

Pretrained Dimension Imports

The code below shows how I managed to load the glove Pretrained Word Embedding. Since there are 3 different embedding dimensions (50, 100 and 200), which are already downloaded in the zip file, I decided to experiment with all of them to assess if they would affect the final test accuracy of the models.

Since every word is represented by an n dimensional vector, where n is the embedding dimension and the values in the array are separated by whitespace (‘ ’), therefore line.split() can be used to separate the values. Thereafter, the word itself is saved in the variable ‘word’ and the n arbitrary floating-point values are saved in the variable values as a numpy array. Thereafter, the word and the corresponding numpy array are added as key value pairs into the embeddings_index dictionary.

```
import os
glove_dir = 'C:/Users/maste/Desktop/DL/Week 7 RNN 2/glove.6B'

embeddings_index = {}
#file will be .50d, .100d or .200d based on dimension
f = open(os.path.join(glove_dir, 'glove.6B.50d.txt'), encoding="utf8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()
print('Found %s word vectors.' % len(embeddings_index))

# 50, 100 or 200
embedding_dim = 50

embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # Words not found in embedding index will be all-zeros.
            embedding_matrix[i] = embedding_vector
```

```
Found 400000 word vectors.
```

The embedding matrix allows us to form a mapping to be drawn from the word indexes with the corresponding numpy arrays, which will be used later for training the models. The for loop allows the set of n values to be loaded and saved into the embedding matrix based on the order in which the words appear in the training dataset.

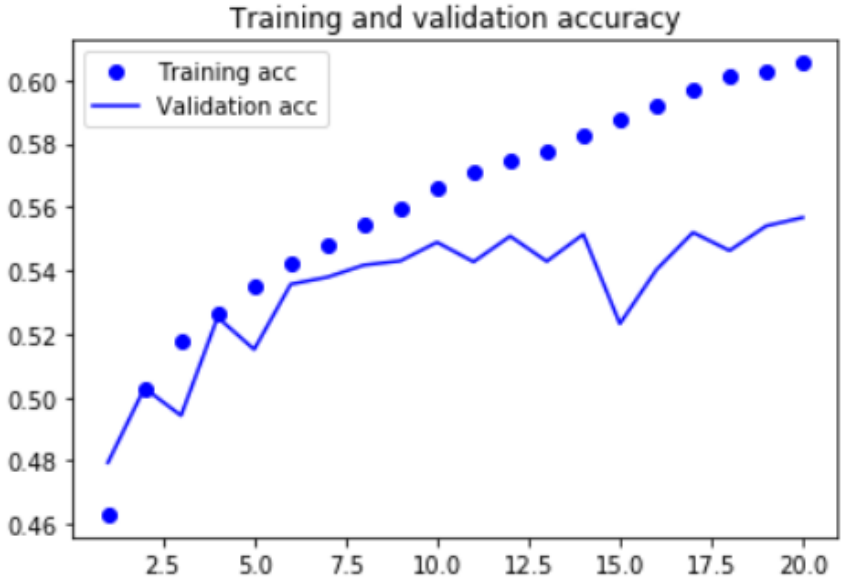
Model #13 PT LSTM RMSprop Base glove50

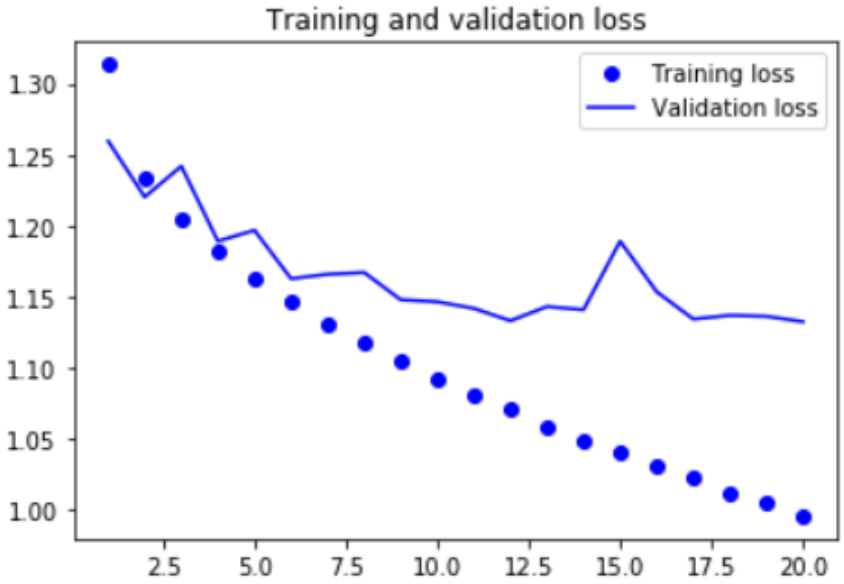
Thereafter I moved on to experiment with pretrained models. For this first base model, I used an embedding dimension of 50. The model also consisted of 2 LSTM layers with 32 nodes each and the final Dense layer with 5 nodes. I also specified the embedding matrix as an argument for the set_weights() method so that the list of values can be set based on the list of values supplied from the glove file. All the values in the embedding matrix are currently frozen for this model, meaning that backpropagation does not update them and thus, they are not trainable.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, return_sequences=True))
model.add(LSTM(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
model.summary()
```

Similar to the Non Pretrained Base Models, I used an initial batch size of 128 and trained the model for 20 epochs, with the validation split remaining the same at 0.2 (80% training data and labels, 20% validation data and labels).

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2)
```





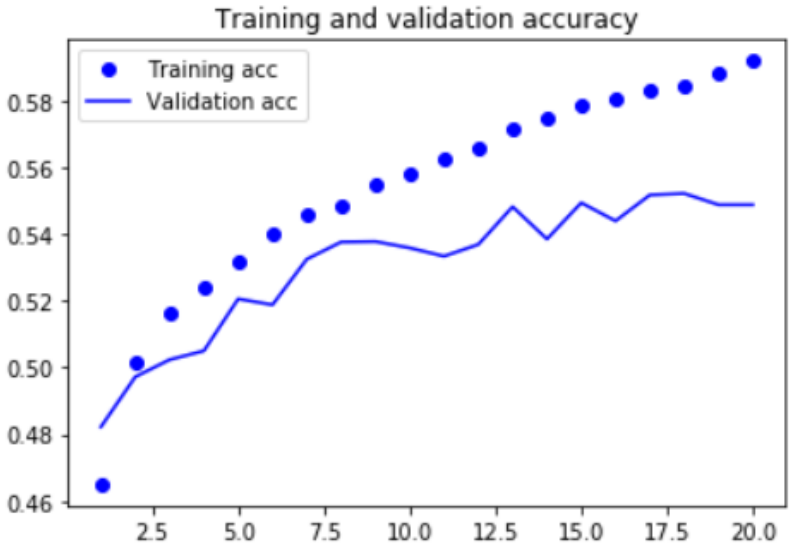
This model overfitted quite slowly at around 10 epochs and there were many fluctuations in the validation accuracy although it started to plateau at around 17 epochs. The training accuracy continues to rise steadily even after the model has overfitted. However, the model achieved quite a poor test accuracy of 54.71%, which is even lower than the Non-Pretrained LSTM and GRU models. I thought that this could be due to the fact that the embedding matrix was frozen and thus the values used for training were not tuned by backpropagation and tailored to the words in the dataset.

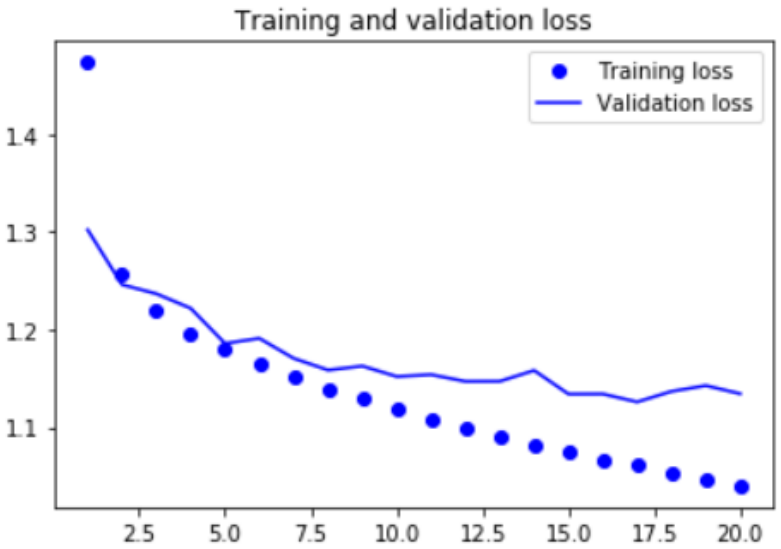
```
8510/8510 - 3s - loss: 1.1464 - acc: 0.5471

[1.146439734637107, 0.54712105]
```

Model #14 PT LSTM RMSprop glove50 FT1

Before unfreezing the values in the embedding matrix for them to be updated, I decided to just experiment with the effects of some fine tuning on the model's accuracy. I added some L2 regularization for both the LSTM layers with a magnitude of 1e-2, while keeping the rest of the hyperparameters constant.

```
# Build the Model
embedding_dim = 50

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
model.summary()
```





The model overfitted slightly earlier at around 8 epochs compared with the base model's, but it also achieved a higher test accuracy of 54.79% and a slightly lower loss score of 1.1437. Thus, I concluded that having regularization helped to improve model accuracy.

```
8510/8510 - 2s - loss: 1.1437 - acc: 0.5479

[1.1436806535888924, 0.5479436]
```

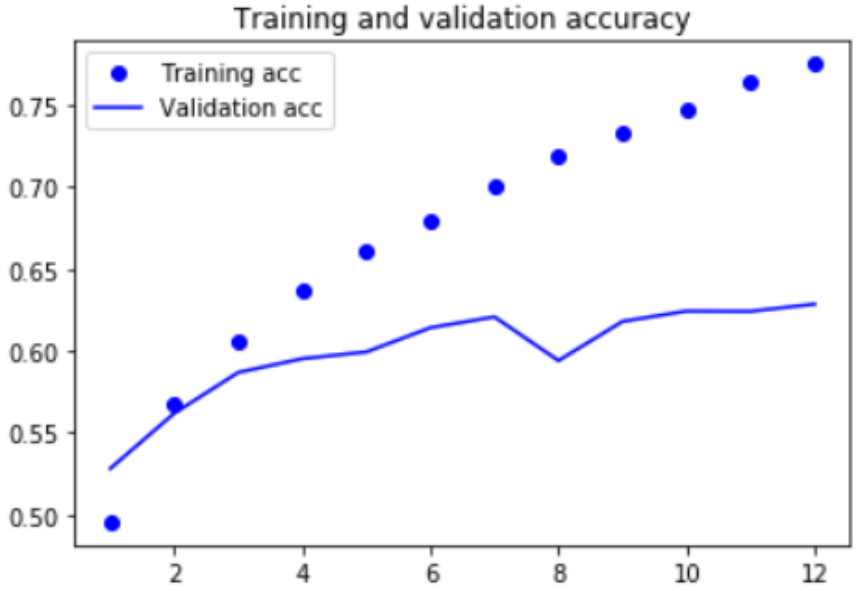
Model #15 PT LSTM RMSprop glove100 (FT2)

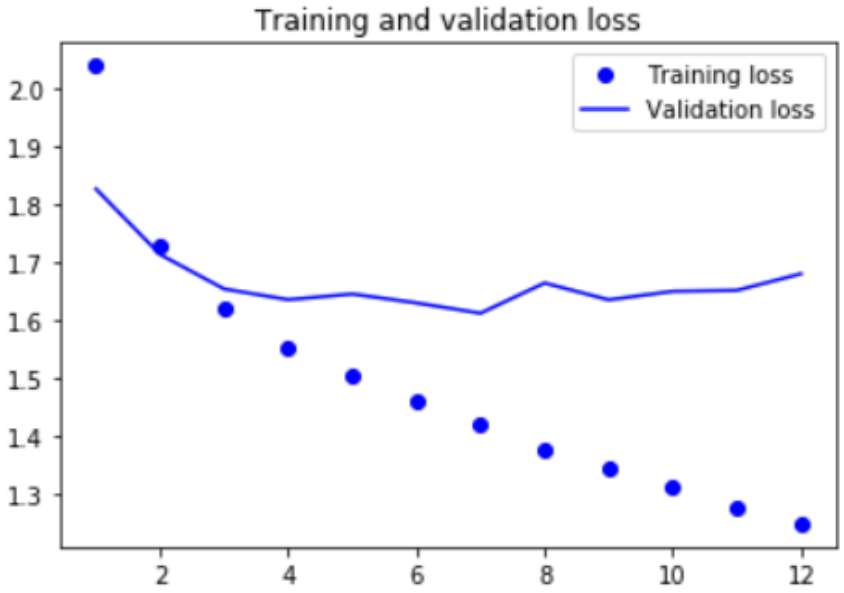
Thereafter, I proceeded to make some of the layers to be trainable, which allows for the embedding matrix to be updated and in turn help to increase the test accuracy of the model. I also added an additional layer of 32 nodes and tweaked the magnitude of the L2 regularization to slow down overfitting. I used an embedding dimension of 100 for this model.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.02),return_sequences=True))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.02),return_sequences=True))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[2].trainable = False
model.summary()
```

I was not sure of what to expect in terms of model performance so I ran it for 12 epochs to see if it would underfit or overfit and had plans to thereafter increase the number of epochs for training if the latter was true.

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=12,
                    batch_size=128,
                    validation_split=0.2)
```





This model overfitted rather rapidly at about 3 epochs so I did not need to increase the number of training epochs. This is in spite of the similar pattern from the base model in which the training accuracy continues to rise gradually even after the model has overfitted. It improved significantly from the previous model and achieved a test accuracy of 63.03%.

```
8510/8510 - 4s - loss: 1.6788 - acc: 0.6309

[1.6788121832523728, 0.6309048]
```

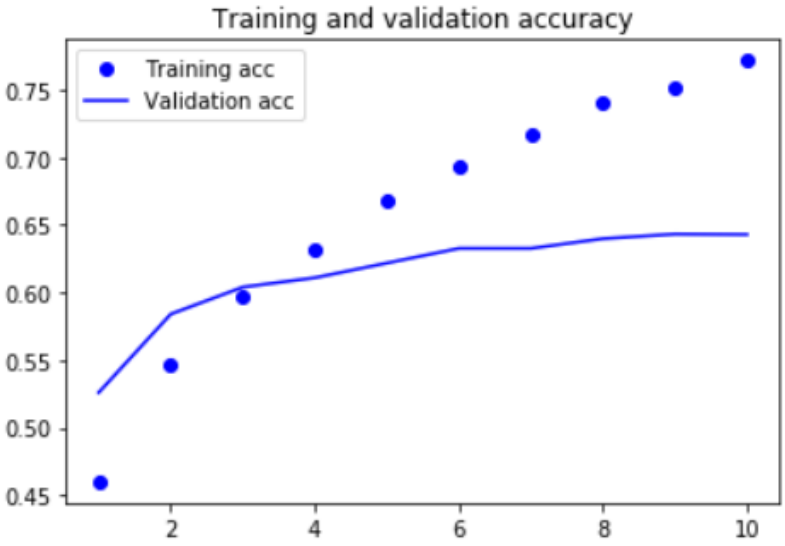
Model #16 PT LSTM RMSprop glove200 increase nodes in layers, add dropout (FT3)

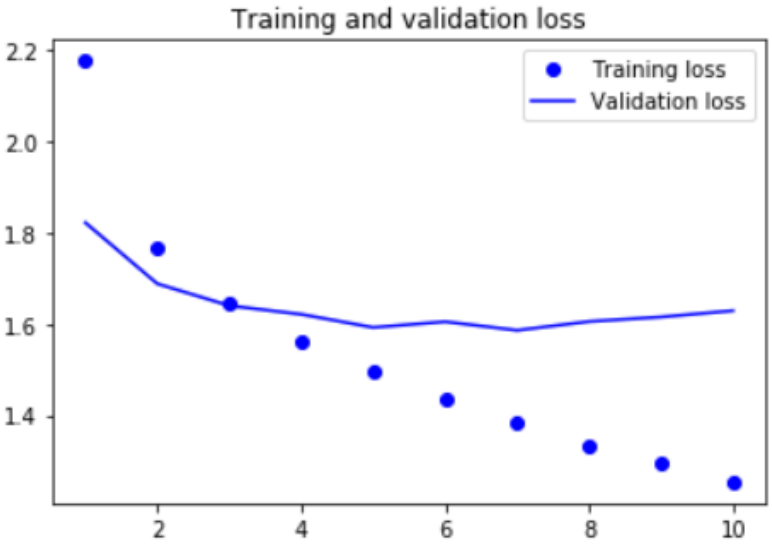
For this model I increased the nodes in 2 of the 3 LSTM layers to 64 up from 32 and also added dropout and recurrent dropout, both with a magnitude of 0.2 since I expected the model to overfit much more quickly. I also changed the number of trainable layers in this instance and increased the embedding dimension to 200, up from the previous 100.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM, Dropout
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

I ran this model for 10 epochs instead of the previous 20 as I expected it to overfit much more quickly.

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```





As expected, the model overfitted at 4 epochs and the validation accuracy curve plateaued quite quickly after the overfitting occurred, at about 63%. Similarly, the validation loss plateaued at about 1.6. The model achieved a test accuracy of 63.43%.

```
8510/8510 - 5s - loss: 0.9920 - acc: 0.6343

[0.9920371968373569, 0.63431257]
```

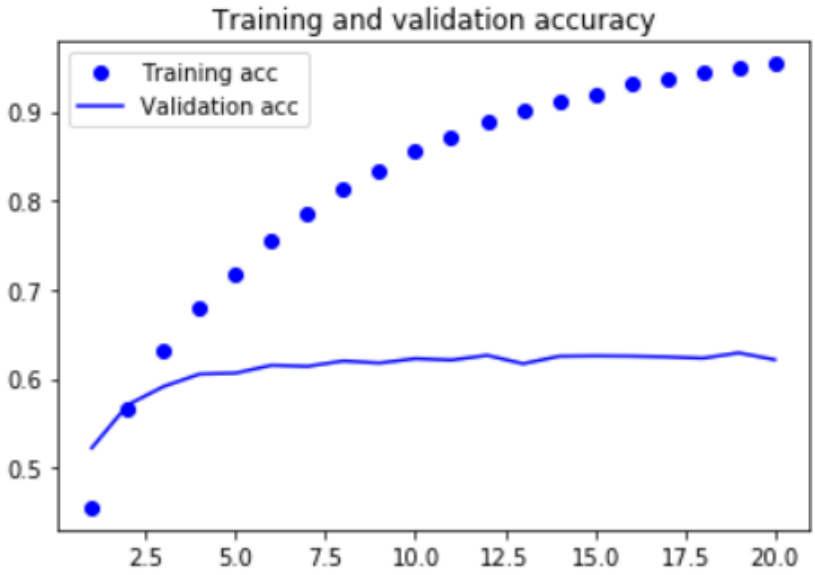
Model #17 PT LSTM Adam glove100 Base

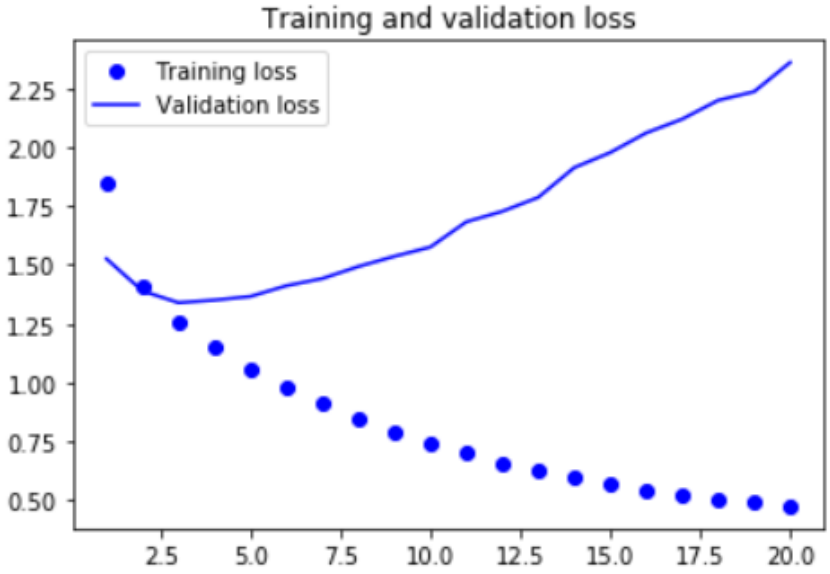
Thereafter, I chose to change the optimiser to Adam. For this model I used an embedding dimension of 100, with 3 LSTM layers, each with 32 nodes and with L2 recurrent regularization with a magnitude of 1e-2. I also unfroze some of the layers in order to allow the weights to be updated through backpropagation, which would better tailor the weights towards training the model on the dataset.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

I ran the model for 20 epochs with a batch size of 128.

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2)
```





This model overfits rather quickly at around 3 epochs and thereafter the validation accuracy plateaus at about 63% while the training accuracy continues to rise and plateaus only at about 95%. The validation loss curve plateaus before rising quite quickly. The model achieves a test accuracy of 62.74% and a rather high loss score of 2.2992.

```
8510/8510 - 4s - loss: 2.2992 - acc: 0.6274

[2.2991861951897485, 0.62737954]
```

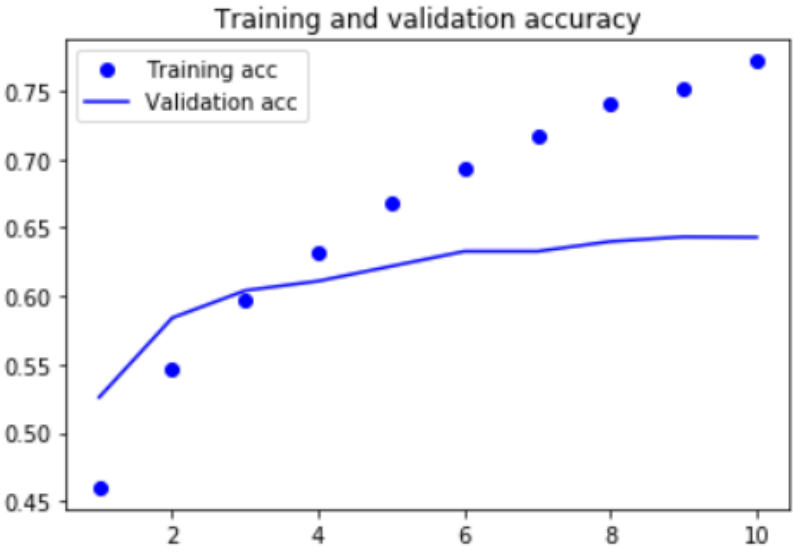
Model #18 PT LSTM Adam glove200 add reg and dropout

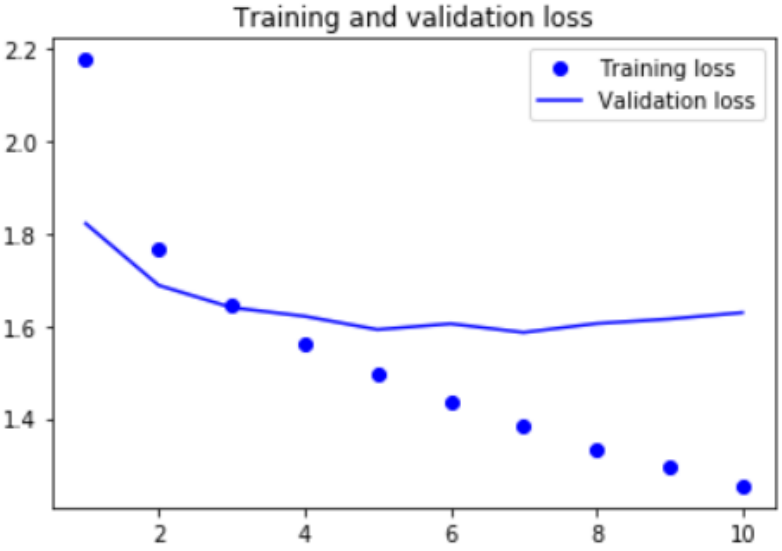
I went on to then increase the dimension of the embedding to 200 as well as the magnitude of the dropout and recurrent dropout to 0.2. I hoped that together with the L2 regularization already put into place, this would hopefully help to slow down the overfitting of the model.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM, Dropout
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

Since the model already overfits quite quickly, I decided to reduce the number of epochs to 10.

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```





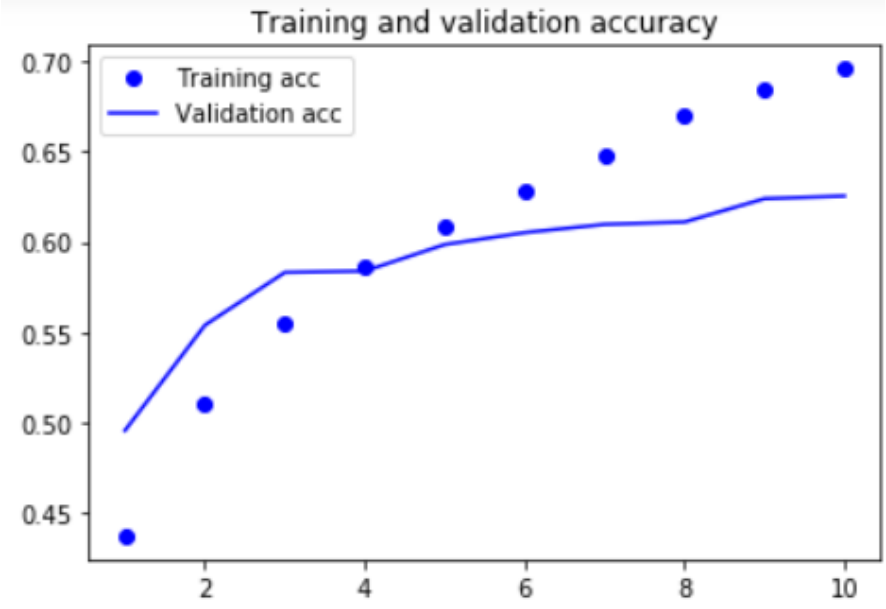
Unfortunately, the overfitting was only slightly delayed in this instance and the model now overfits at 4 epochs, which is only slightly later as compared with the previous 3 epochs. On the bright side, the model did achieve a better test accuracy of 63.63% and a loss score of 1.6351.

```
8510/8510 - 7s - loss: 1.6351 - acc: 0.6363

[1.6350606343440808, 0.6363102]
```

Model #19 PT LSTM Adam glove200 FT1

Thereafter, I also tried to add a layer and further increase the magnitude of dropout and recurrent dropout to 0.3 for 3 of the 4 LSTM layers as shown in the code below, with the test of the hyperparameters kept constant for a fair test to assess whether this fine-tuning helps to improve model performance.

```
# Build the Model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Flatten, Dense, LSTM, Dropout
from tensorflow.keras import regularizers
model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.3, recurrent_dropout = 0.3))
model.add(LSTM(64, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.3, recurrent_dropout = 0.3))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01),return_sequences=True, dropout = 0.3, recurrent_dropout = 0.3))
model.add(LSTM(32, recurrent_regularizer=regularizers.l2(0.01)))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```



This however resulted in the model overfitting at 5 epochs, only slightly later than the 4 epochs from the previous iteration of fine tuning. However, this had a detrimental effect on the test accuracy which fell to 62.02%.

```
8510/8510 - 8s - loss: 1.6291 - acc: 0.6202

[1.6290599204678653, 0.62021154]
```

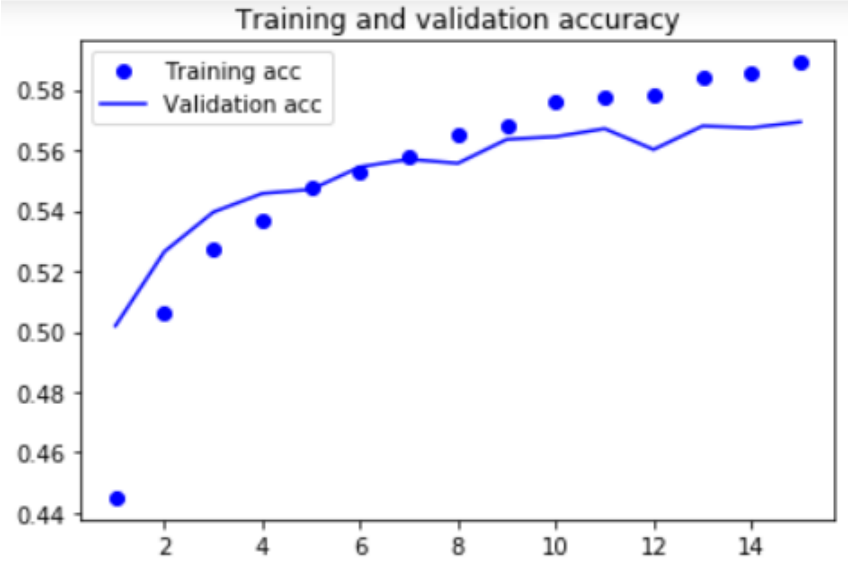
I then decided to move on to try out the Pretrained GRU architectures since I was satisfied with the results of my experiments to optimize the Pretrained LSTM Base Models in order to improve the test accuracies and reduce overfitting. From the various LSTM architectures, I selected Model 18 for further evaluation for the best model later on.

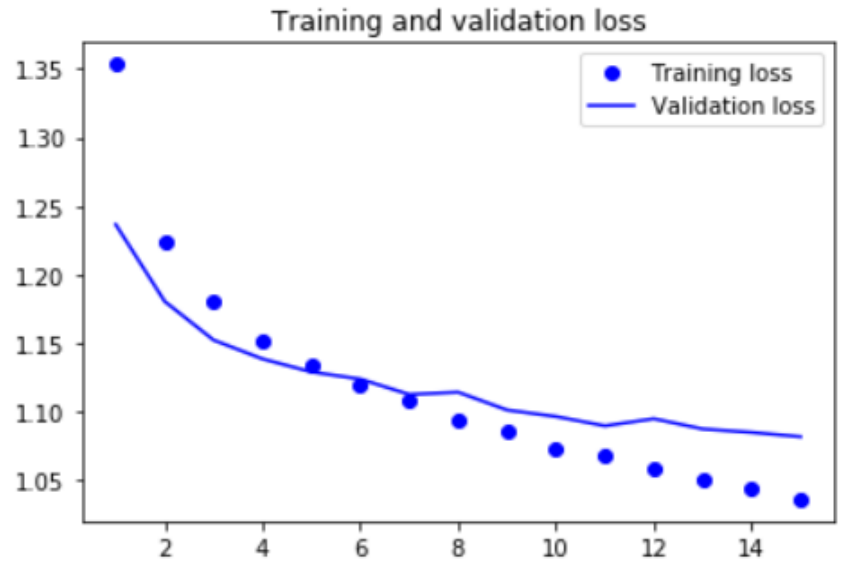
Model #20 PT GRU Adam glove100 Base

For the first Pretrained GRU Model, I decided to use an embedding dimension of 100. I also chose to include dropout and recurrent dropout with a magnitude of 0.2 and freeze the trainable parameters for this base model. Similar to the previous base models, I used 2 layers of GRU with 32 nodes each. In addition, I trained the model for 15 epochs as I expected it to overfit more quickly than if I were to use RMSprop.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(32, return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=15,
                    batch_size=128,
                    validation_split=0.2)
```





Surprisingly, the model overfitted at about 12 epochs, which is even later than the LSTM RMSprop Base Model. The rise in the training accuracy is also much gentler in this instance, with it achieving a loss of 1.0892 and a test accuracy of 55.98%, which is also higher than the LSTM RMSprop Base (Model 13)

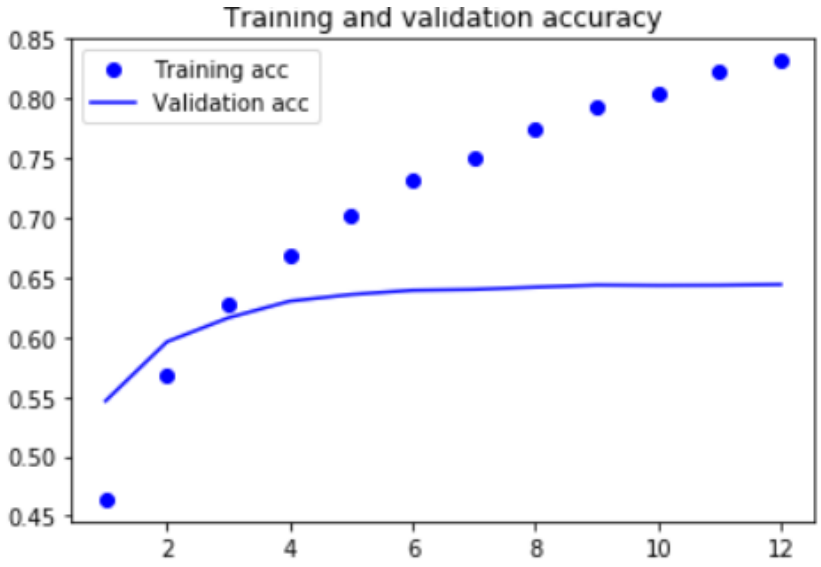
```
8510/8510 - 4s - loss: 1.0892 - acc: 0.5598

[1.0892318515463806, 0.559812]
```

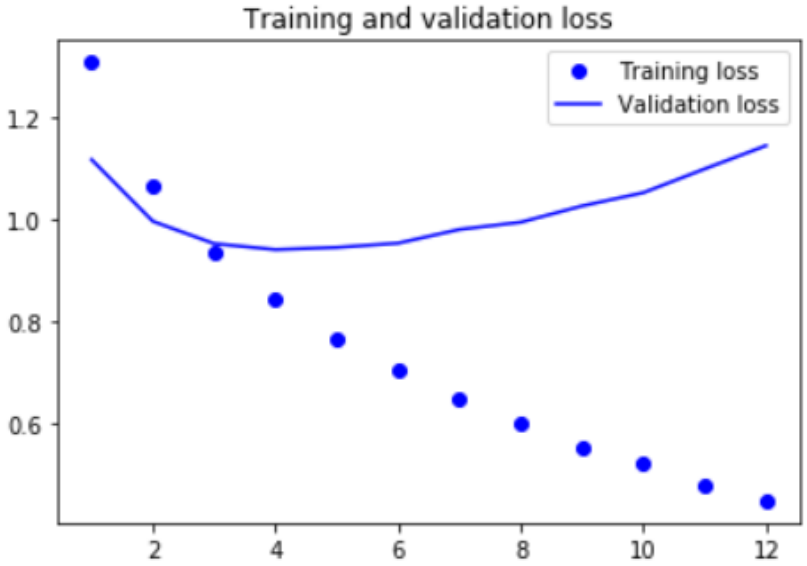
Model #21 PT GRU Adam glove200 FT1

For the first iteration of fine tuning on the base model, I decided to add an additional layer with 64 nodes and dropout and recurrent dropout with magnitudes of 0.2 to slow down overfitting caused by the increase in the number of layers and nodes. In addition, I also unfroze some of the layers for to update the weights through backpropagation with the use of the sparse categorical crossentropy loss function. I expected this model to overfit rather quickly thus I only trained it for 12 epochs instead of the previous 15.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(64, return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(GRU(32, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=12,
                    batch_size=128,
                    validation_split=0.2)
```





As expected, the model overfits at 3 epochs, similar to what happened when I unfroze the layers for training in the Pretrained LSTMs. The validation accuracy plateaus at 64% after 4 epochs and the validation loss decreases, plateaus before rising gradually again. This model achieved a test accuracy of 64.37%, which is higher than all the fine-tuning iterations using the Pretrained LSTM architectures. It also achieves a lower loss score of 1.1327.

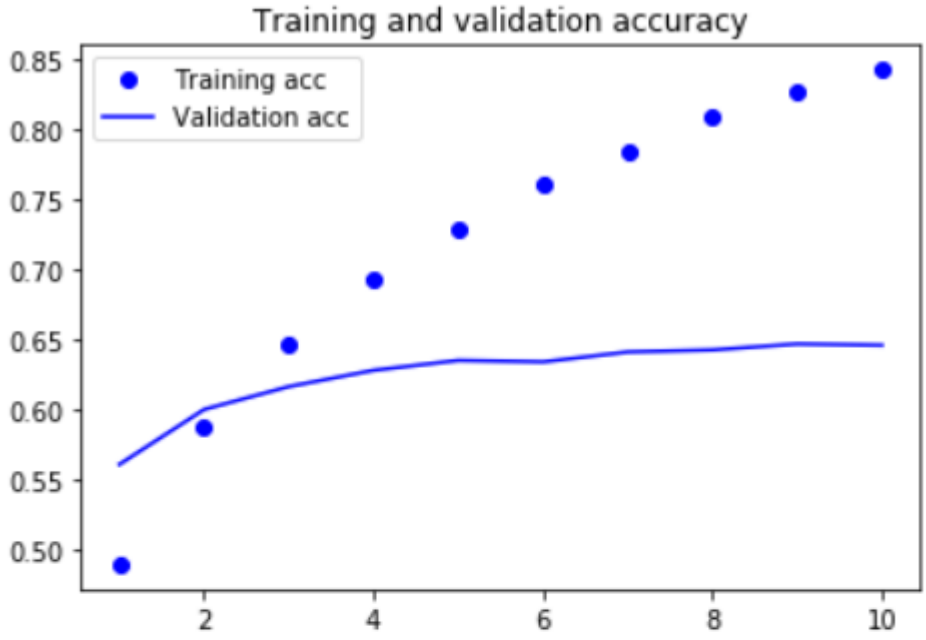
```
8510/8510 - 6s - loss: 1.1327 - acc: 0.6437

[1.1326846046537407, 0.6437133]
```

Model #24 PT GRU Adam FT2

For this model, I increased the number of nodes further and decreased the number of epochs to 10, since I expected a faster rate of overfitting. For the second layer I increased it up to 128 nodes, whilst keeping the rest of the hyperparameters constant.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(64, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(128, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```



Again, the model overfits at 3 epochs but it does achieve a slightly higher accuracy of 64.54% and a slightly lower loss of 1.1292 as compared to the first iteration of fine tuning the Pretrained Adam GRU network architecture.

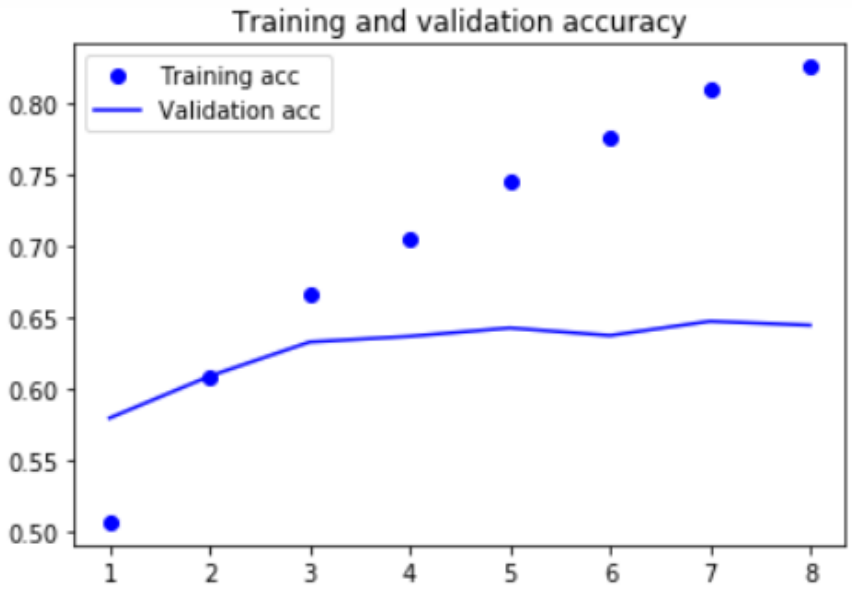
```
8510/8510 - 6s - loss: 1.1292 - acc: 0.6454

[1.1291882836580556, 0.6453584]
```

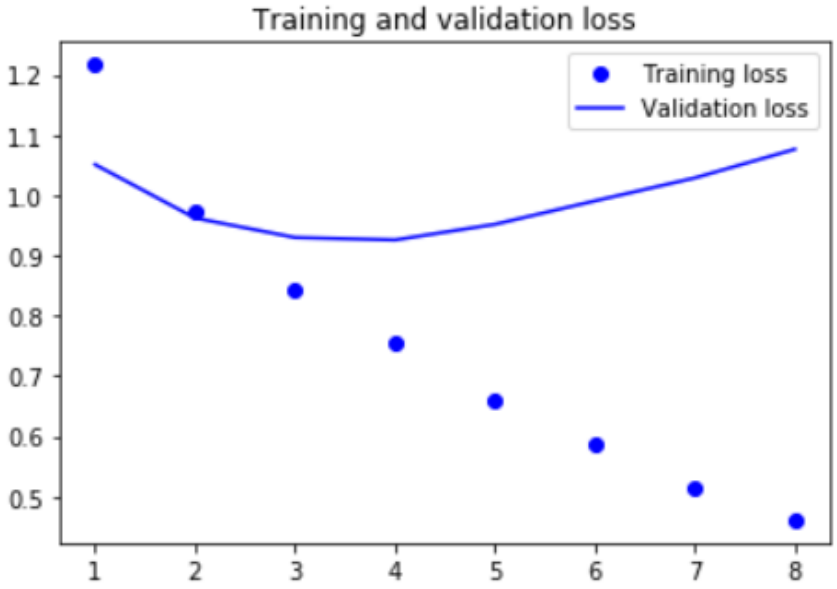
Model #25 PT GRU Adam FT3

Since I observed that increasing the number of nodes has a positive impact on both the test loss score and accuracy, I decided to further increase the number of nodes once more (I double the nodes in 2 of the 3 LSTM layers from the base model), whilst decreasing the nodes to 8 as I expected it to overfit more quickly.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(128, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(256, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=8,
                    batch_size=128,
                    validation_split=0.2)
```





The model however, did not overfit more quickly, but as hinted previously, the model achieved its highest test accuracy of 64.69% and a loss of 1.0703. I was satisfied with my fine tuning since the models all achieved test accuracies above the average of models so far, which is about 62%.

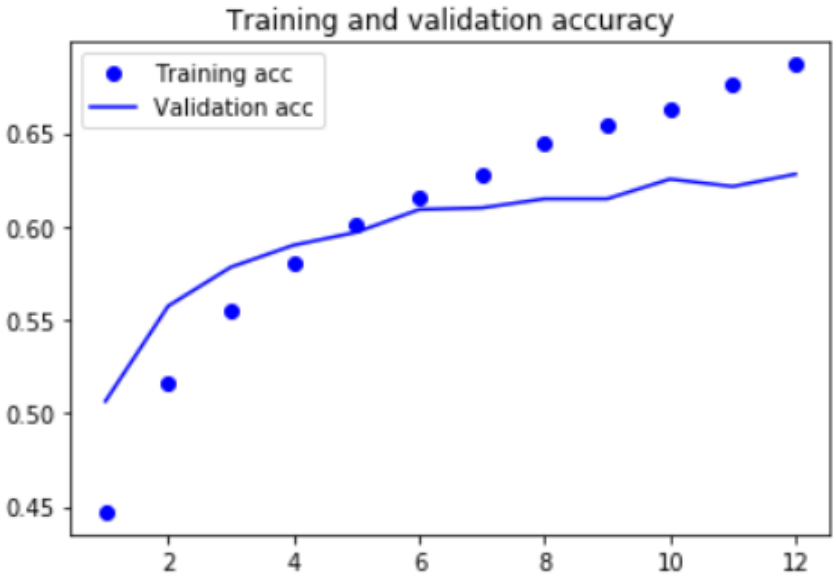
```
8510/8510 - 7s - loss: 1.0703 - acc: 0.6469

[1.0703108439014044, 0.646886]
```

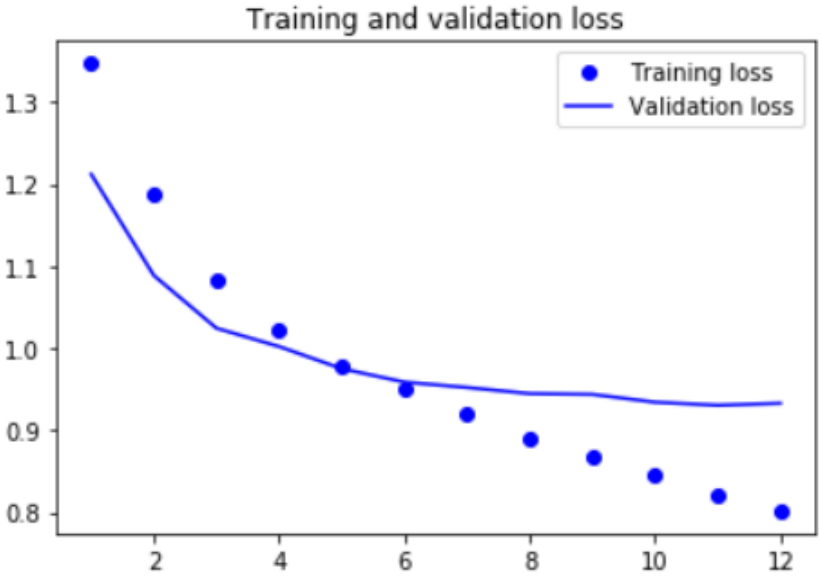
Model #22 PT GRU RMSprop glove100 Base

Thereafter, I proceeded on to assess if the RMSprop optimiser could achieve similar results to its Adam counterpart. For this I used similar hyperparameters and network architecture as the first iteration of Adam Fine Tuning (Model 21).

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(64, return_sequences=True, dropout = 0.2, recurrent_dropout = 0.2))
model.add(GRU(32, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=12,
                    batch_size=128,
                    validation_split=0.2)
```





The model overfitted later at about 7 epochs but it achieved a worse test accuracy of 62.78%, which is below the average of 64% for the fine-tuned models with the Adam optimiser. However, it achieved an even lower loss score of 0.9265.

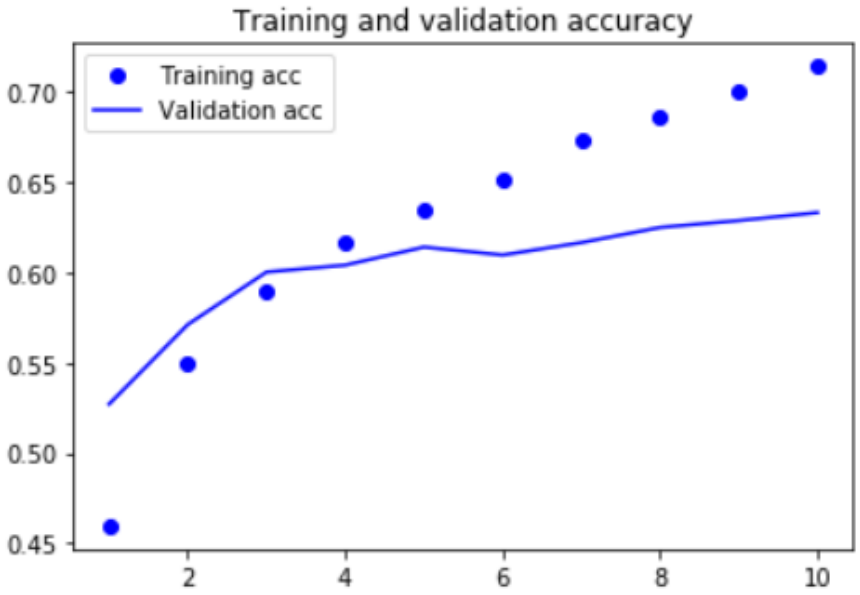
```
8510/8510 - 6s - loss: 0.9265 - acc: 0.6278

[0.9265411731499202, 0.6278496]
```

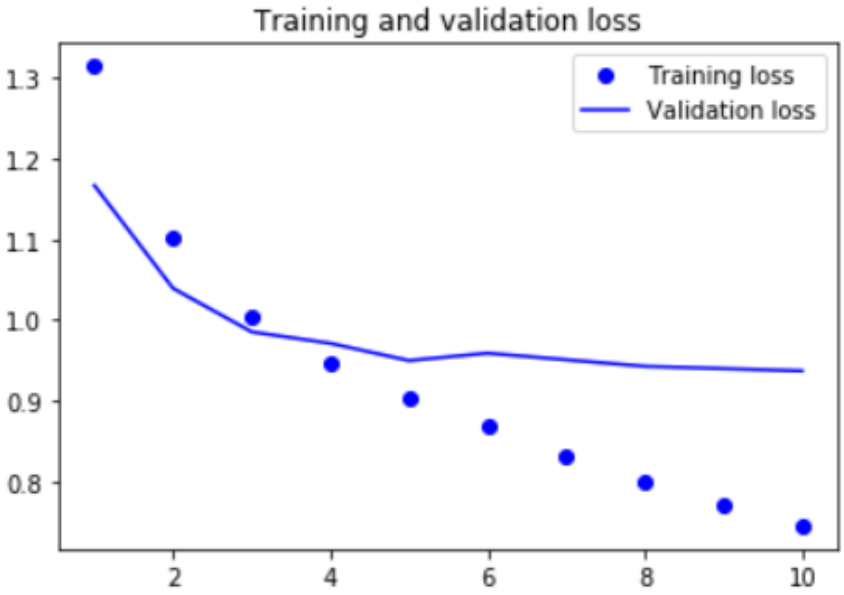
Model #23 PT GRU RMSprop glove100 FT1

Thereafter, I decided to reduce the number of layers to 2 with the hope of increasing the test accuracy. For this model, I reduced the number of epochs to 10 since I had also increased the number of nodes at the same time.

```
from tensorflow.keras.layers import GRU, Embedding, Dense
from tensorflow.keras.models import Sequential

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=max_len))
model.add(GRU(128, return_sequences=True, dropout = 0.1, recurrent_dropout = 0.1))
model.add(GRU(32))
model.add(Dense(5, activation='softmax'))
model.layers[0].set_weights([embedding_matrix])
model.layers[1].trainable = False
model.summary()
```

```
# Train the Model
model.compile(optimizer='rmsprop',
              loss='sparse_categorical_crossentropy',
              metrics=['acc'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```





This model achieved a decent test accuracy of 63.16%, even though there was a slightly detrimental effect on the loss which increased by 0.01 to 0.9330. It also overfits now at 4 epochs compared to the previous 7 epochs.

```
8510/8510 - 4s - loss: 0.9330 - acc: 0.6316

[0.9329513919479558, 0.63160986]
```

For the Pretrained GRU architectures, I shortlisted Model 25 for further evaluation since it has the highest test accuracy of 64.69% and an average test loss score out of all the models.

General Observations in model training

I observed that:

- GRU models tend to have a higher test accuracy for similar model architecture (number of nodes and layer) compared to LSTMs.
- Using Adam optimiser leads to models generally overfitting more quickly.
- Use of multilayered recurrent dropout and dropout in Pretrained GRUs helps to increase test accuracy
- Use of L2 regularization and freezing layers in Pretrain LSTMs helps to slow down overfitting
- Embedding Dimension of Pretrained layers does not have much impact on test accuracy of the model

The tweaking of the model hyperparameters had made it clearer to me that I should focus more on the points listed above in the future when training such sentiment analysis models since they enable me to make better and more informed decisions to optimizing their performance for the best results.

4. Evaluating the best test accuracy models based on predefined criteria and Using the Best Model to Make Prediction

Based on the three different distinct network architectures I have experimented with (Non-Pretrained LSTM, Non-Pretrained GRU, Pretrained LSTM and Pretrained GRU), I have shortlisted the best model for each architecture type for my consideration towards the best model, which I would ultimately use for generating emoticons based on new sentiment supplied. In my opinion, a best model should:

- Not overfit too quickly
- Have good test accuracy of above 62% (which is the average of the models)
- Be able to correctly predict most of the corresponding emoticons based on the sentiment supplied

Shortlisted Models

- Model 6 (Non-Pretrained LSTM): Test accuracy 63.41%
- Model 11 (Non-Pretrained GRU): Test accuracy 62.97%
- Model 18 (Pretrained LSTM): Test accuracy 63.63%
- Model 25 (Pretrained GRU): Test accuracy 64.69%

To ensure that there is no bias in picking the best model, I decided to use a points system—1 point for the best model for each condition, 2 points for the second model and so on. Then I proceeded to calculate the average points and the model with the lowest number of points would be my best model.

Steps taken to conduct evaluation

```
from tensorflow.keras import models
model = models.load_model('text_model_25.h5')
```

Firstly, the saved model is loaded into the jupyter notebook.

```
# takes the user input
text_input = np.array([input("Enter some text here: ")])
```

```
Enter some text here: I am so happy today!
```

Thereafter the user is prompted to enter the sentiment to be used to predict the label.

```
# convert the user input into numeric tensor
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

def input_to_tensor(text_input):
    tokenizer.fit_on_texts(text_input)
    sequences = tokenizer.texts_to_sequences(text_input)
    data = pad_sequences(sequences, maxlen=max_len)
    return data

def prediction(model, data, emoji_dictionary):
    prob = model.predict(data)
    pro_df = pd.DataFrame(prob, columns = emoji_dictionary)
    result = emoji_dictionary[np.argmax(prob)]
    return pro_df, result
```

The input_to_tensor function is then used to tokenize the user's text_ input into a tensor. Thereafter it is fed into pad_sequences in order to locate the particular word's index, if it is in the list of 10,000 unique words that were initially defined in the data processing step. Thereafter the individual words are positioned a numpy array based on the index of the word according to the positioning as supplied in the word input. (e.g. 'I' is at index 5 in the list of 10,000 words and it is the first character of the text_input). Since "I am so happy today!" has only a length of 5 words, the tensor will be padded with 29 zeros as shown below since the initial max length is defined as 34.

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   5 174  21  26  44]]
```

Thereafter, the returned tensor (data) as well as emoji_dictionary is fed into another function predicta() which predicts and retrieves the emoticon based on the label with the highest probability for a given sentiment. It also returns a probability dataframe pro_df which consists of the labels and the corresponding probabilities, which is also useful later in assessing the certainty of the model's prediction.

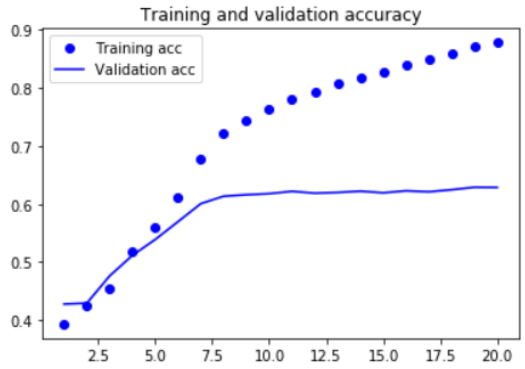
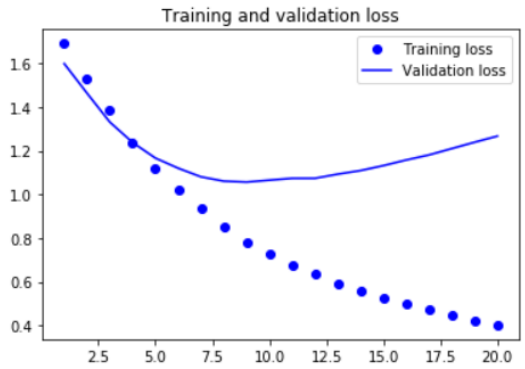
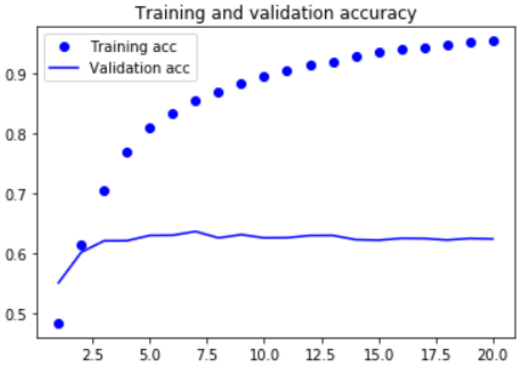
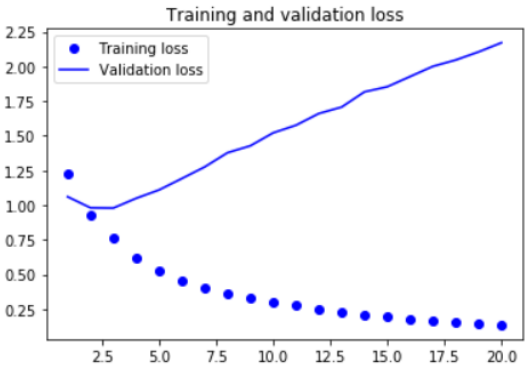
The code below illustrates how the functions were called and displaying of the output as well as the probability dataframe to the user.

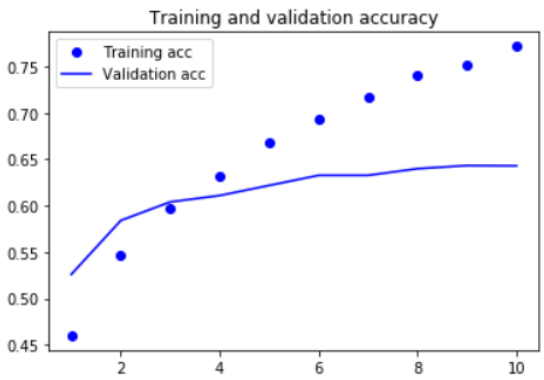
```
# show the model output using predict function
data = input_to_tensor(text_input)
pro_df, result = prediction(model, data, emoji_dictionary)
print(f'The prediction for {text_input} is: {result}\n\n', pro_df)
```

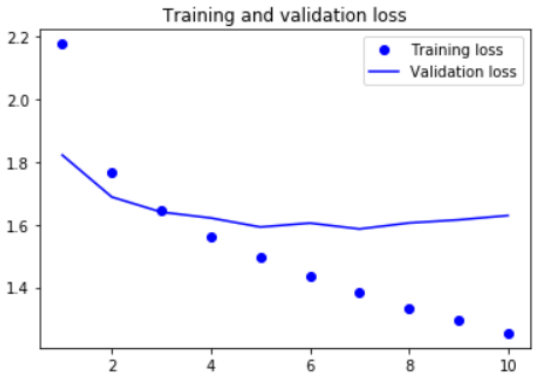
With my evaluation code written and tested successfully, I then proceeded to evaluate the models based on the criteria I previously formulated.

Condition 1: Not overfit too quickly

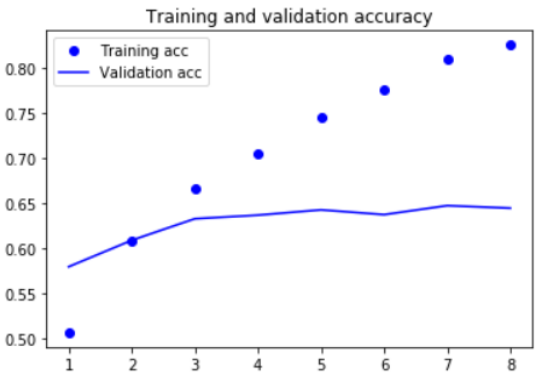
Based on the accuracy and loss curves of the various models, I was able to determine which model actually overfitted more quickly as compared to other models.

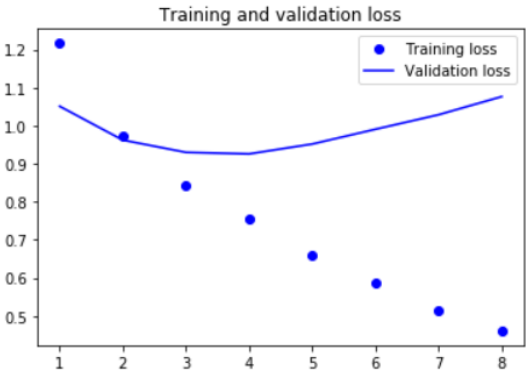
Model 6	Model 11
  Overfits at about 7 epochs	  Overfits at about 2 epochs
Model 18	Model 25





Overfits at about 4 epochs





Overfits at about 3 epochs

The best model in this case should take the largest number of epochs to overfit. From the comparison above, Model 6 takes the longest of 7 epochs to overfit, followed by Model 18 at 4 epochs and thereafter Model 25 at 3 epochs and finally Model 11 at 2 epochs. I feel that it is important to follow the universal workflow of machine learning when regularizing my models to ensure that they do not generalize too much to the training data as a result of overfitting.

Model 6: 1; Model 11: 4; Model 18: 2, Model 25: 3

Condition 2: Have good test accuracy of above 62%

All the models meet this criterion. The model with the highest test accuracy is Model 25, with 64.69%, followed by Model 18 with 63.63% and Model 6 with 63.41% and finally Model 11 with a test accuracy of 62.97%. I set a benchmark of 62% because this is the average accuracy which most achieved.

Model 6: 3; Model 11: 4; Model 18: 2, Model 25: 1

Condition 3: Be able to correctly predict most of the corresponding emoticons based on the sentiment supplied

For this segment I randomly generated some input sentiments to be passed into the various models to determine if it could predict the label that I had decided beforehand, which is to be mapped to the particular emoticon.

Test texts:

I am so happy today! ❤

So excited!!! 😍

Let's take a picture here 📷

I'm in Love... 😍 or ❤

Cooking my lunch now 🔥

Flaming hot oven 🔥

HAHAHAHAHA so funny! 😂

Heart of stone ❤

Cold Black Icy Heart ❤

Ready the camera and Say Cheese 📷

For sentiments like “I’m in Love” I allowed for 2 possible correct emoticon labels because I felt that both were equally appropriate labels, since natural language can be subjective at times. I also included a correct label certainty column in my evaluation tables. A Red Fill is given for an incorrect label prediction and a Green Fill is given for a correct prediction to allow me to identify correct and incorrect predictions more quickly.

A score is given to the model based on the number of correct labels it predicts as compared with the total number of prediction tasks (out of 10 instances).

Model 6 (3/10)

Sentiment	Model Prediction	Correct Label	Correct Label Certainty (%)
I am so happy today!	❤️	❤️	61.5
So excited!!!	😂	😍	20.0
Let's take a picture here	😍	📷	7.3
I'm in Love...	❤️	❤️ or 😍	47.1
Cooking my lunch now	🔥	🔥	66.5
Flaming hot oven	😍	🔥	4.5
HAHAHAHAHA so funny!	❤️	😂	1.0
Heart of stone	😂	❤️	15.4
Cold Black Icy Heart	😍	❤️	27.6
Ready the camera and Say Cheese	😂	📷	2.1

Model 6 did worse than expected in correctly labelling the new sentiments provided (It gets only 3 out of 10 labels correct). The certainty of the model in prediction varies widely for both the correct and incorrect labels. I expected it to do better on the new test data since it has a test accuracy of above 63%.

The model predicts 3 of the emoticons as labels thrice and it gets the black heart emoticon as the correct output for the most number of times.

Model 11 (5/10)

Sentiment	Model Prediction	Correct Label	Correct Label Certainty (%)
I am so happy today!	❤️	❤️	86.4
So excited!!!	😍	😍	96.4
Let's take a picture here	❤️	📷	0.0
I'm in Love...	😍	❤️ or 😍	50.1
Cooking my lunch now	😂	🔥	1.1
Flaming hot oven	❤️	🔥	0.5
HAHAHAHAHA so funny!	😍	😂	0.0
Heart of stone	❤️	❤️	99.6
Cold Black Icy Heart	❤️	❤️	98.4
Ready the camera and Say Cheese	😍	📷	0.0

Model 11 achieved an average evaluation score of 5/10. However, the model is always very certain with its prediction. For 3 out of the 4 correctly predicted emoticon labels, the certainty is above 95% and for all of the incorrectly predicted labels, the certainty is below 2%. In the case for “I'm in Love” the model predicts almost a 50-50 for the ❤️ and 😍 emoticons respectively as shown below.

```
The prediction for ["I'm in Love..."] is: 😍

          0         1         2         3         4
0  0.50094   0.02875   0.00038  0.001832  0.468098
```

Model 11 predicts the black heart emoticon (❤️) most frequently (5 out of 10 times). Out of these 5 instances, only 3 of them are correct. This corresponds to the test accuracy of 62.97%.

Model 18 (4/10)

Sentiment	Model Prediction	Correct Label	Correct Label Certainty (%)
I am so happy today!	❤️	❤️	65.9
So excited!!!	😍	😍	38.5
Let's take a picture here	😂	📷	5.7
I'm in Love...	❤️	❤️ or 😍	46.6
Cooking my lunch now	❤️	🔥	1.1
Flaming hot oven	😂	🔥	1.8
HAHAHAHAHA so funny!	😂	😂	41.8
Heart of stone	😂	❤️	24.7
Cold Black Icy Heart	🔥	❤️	27.2
Ready the camera and Say Cheese	😍	📷	6.3

Model 18 did worse than expected in correctly labelling the new sentiments provided (It gets only 4 out of 10 labels correct). The certainty of the model in prediction varies widely for both the correct and incorrect labels. I expected it to do better on the new test data since it has a test accuracy of above 63%.

Model 18 predicts the tears of joy emoticons (😂) in 4 out of 10 times, However, out of these 4 instances, only one is a correct label prediction. It also predicts the black heart emoticon correctly (❤️) 2 out of the 3 times.

Model 25 (6/10)

Sentiment	Model Prediction	Correct Label	Correct Label Certainty (%)
I am so happy today!	❤️	❤️	49.4
So excited!!!	😍	😍	52.0
Let's take a picture here	❤️	📷	4.26
I'm in Love...	❤️	❤️ or 😍	49.1
Cooking my lunch now	🔥	🔥	37.3
Flaming hot oven	😍	🔥	2.16
HAHAHAHAHA so funny!	😂	😂	67.7
Heart of stone	❤️	❤️	95.1
Cold Black Icy Heart	😍	❤️	41.3
Ready the camera and Say Cheese	😍	📷	9.7

Model 25 does quite well in correctly labelling the new sentiments provided (It gets 7 out of 10 labels correct). The certainty of the model in prediction varies widely for the correct labels. For 3 out of 4 of the incorrect labels, the certainty for the actual correct label is very low.

Model 25 predicts the black heart emoticon (❤️) correctly the greatest number of times (3 out of 4) and based on the frequency of the predictions, it is best at predicting that particular emoticon.

For this criteria, Model 25 did the best with 6 out of 10 correct labels, followed by Model 11 with 5 correct emoticons, Model 18 with 4 labels and finally Model 6 with 3 correct emoticons.

Model 6: 4; Model 11: 2; Model 18: 3, Model 25: 1

Final Tabulated Results

Model 6: (1+3+4) /4 = 2

Model 11: (4+4+2) /4 = 2

Model 18: (2+2+3) /4 = 1.75

Model 25: (3+1+1) /4 = 1.25

Hence, I conclude that Model 25 is the best model since it has the lowest number of points based on the scoring given in meeting the evaluation criteria.

Reasons for the poor performance of Models 6 & 18

The poor performance of these models on new test data may be due to several different factors. Firstly, both models 6 and 18 have the same (core) LSTM architecture. Based on the accuracies of the models that I have trained, the test accuracies of GRUs are generally much better than their LSTM counterparts, thus resulting in them not faring well when working with new test data.

Table 1: Classification accuracies of gru compared with other models

MODEL	AUC SCORE
tf-idf	0.89588
Word2Vec(vetor average)	0.55564
Word2Vec(Kmeans dictionary)	0.80140
LSTM	0.95707
GRU	0.970000

GRUs are also found to converge faster as compared to their LSTM counterparts. Some researchers did experiments using the IMDB dataset and they found that GRUs outperform LSTMs even though the GRU model was trained for a fewer number of epochs (Shamim Biswas, 2015). In their research, they used a slightly different metric to assess the model—AUC, which in this case refers to the area under the curve.

For Model 6, since I used a batch size of 1024 that may have led high test accuracy, which might in turn caused the model to be unable to perform generalization as it has become too optimized towards the training dataset, thus leading to poorer performance on the new test data (newly supplied text sentiments) and resulting in its poor performance.

5. Summary

After all the testing and evaluating of the models, as well as doing some research that gave me insight to how the model architectures relate to the model's test accuracy, test loss score and performance on new test data, I decided to take some time to reflect on some of the improvements I could have made when training models, which could ultimately lead to even better models and also more findings for this sentiment analysis problem.

Firstly, while evaluating my models, I found that out of all my models and all the 10 new test sentiments, none of the predicted emoticons were emoticon 2. Upon further investigation, I discovered that this has got to do with the dataset.

Reasons for Emoticon 2 (📷) not being predicted as a label

TEXT	Label
Been friends since 7th grade. Look at us now we all following our dreams doing what we	0
This is what it looks like when someone loves you unconditionally oh Puppy Brother.	1
RT @user this white family was invited to a Black barbecue and i've never laughed so hard	1
	2
	0
Story On Saturday 136 Collins Ave, Miami Beach, FL 33139 Ladies Contact Me To Join Me	3
Dragged my loving husband to a live episode of #hellofromthemagictavern ï¸ @ Javits	4
#AladdinSF was shining, shimmering, and splendid! Such a talented cast @ SHN: Broadway	0
	4
These have been SUCH A HIT ! READY TO RIDE Harley Davidson (TONS of cities from	3
LINK IN MY BIO #boombap #90s #webster #mcgels #naturaldoc #southbronx #realhiphop	3
	3
	4
	3
	4
#THIS . ï¸ This specially formulated hormone replacement cream was crucial for my	4
	4
Ry Lounge is def the move tomorrow rylounge #pullup #lucid #rylounge @ Ry Restaurant	3
	1
	0
	3
Some things I can never get tired of! ï¸#radiocitymusichall #citylights #nyc #nycviews	4
	0
	4
	4
Houston we have a .....! Every airport in the world should have one of these... i want to	0
	1
	0

dataset

Since there are very few tweets with the corresponding label for emoticon number 2 in the csv file, this may have skewed the predictions of labels, leading to none of the models ever predicting it as the label for the new sentiments supplied.

This could have been circumvented by adding more entries for that particular emoticon in order to ensure that there is no misrepresentation and all the emoticons stand a fair chance of being learnt during the training phase and subsequently identified as a label in the testing phase. Perhaps if I had the time to play around with the dataset, I would be able to also ensure that the number of entries for each different emoticon category is equal to eliminate the initial bias present in the training dataset.

Secondly, based on some of my observations that I had after training my models and finding the relationships between tuning certain hyperparameters in relation to the test accuracy of the model, I found that:

- GRU models tend to have a higher test accuracy for similar model architecture (number of nodes and layer) compared to LSTMs.
- Using Adam optimizer leads to models generally overfitting more quickly.
- Use of multilayered recurrent dropout and dropout in Pretrained GRUs helps to increase test accuracy
- Use of L2 regularization and freezing layers in Pretrain LSTMs helps to slow down overfitting
- Embedding Dimension of Pretrained layers does not have much impact on test accuracy of the model

Based on this feedback, I could try to further improve the Pretrained GRU models, especially that with the Adam optimizer, since it has been proven to have test accuracies of above 64% (3 out of 4 models). Perhaps I could also play around with the learning rate instead of just using the default learning rate. I should also try to tweak the batch size more often to test my hypothesis that a large batch size will have a negative impact on the test accuracies. Perhaps I could also try stronger magnitudes of dropout or recurrent dropout which could help slow down overfitting in the Adam models. I could also try other optimizers like Stochastic Gradient Descent (SGD) with Momentum and also try different embedding dimensions for not just the Pretrained, but also the Non-Pretrained models.

Thirdly, perhaps I could also try out more pretrained word embeddings like FastText or Word2Vec like I did in Assignment 1, where I had experiment with different pretrain architectures instead of just sticking to one word embedding type. This would enable me to get a greater spread of model architectures and perhaps even better results than the glove pretrained word embedding.

Problem 2

6. Overview

The Problem

The main goal of this assignment is to find the optimal neural network model to predict and generate the next character after a given sequence of word(s). As Natural Language is still a work-in-progress area in the field of Deep Learning, there is still a lot of room of improvement. What I am trying to do is merely a simulation of how Natural Language--in this case English could be like when it is generated by Recurrent Neural Networks, which may simulate how toddlers may try to come up with new words. However, adult humans are still undoubtedly better in generating characters, which are put together to construct words and subsequently, phrases and sentences. Humans are also much better at detecting tone, emotions and style in language and this is something that software has yet to achieve.

The problem is a multi-class, single label type of classification as there are 35 possible next characters that the model can predict and the condition that the model can only be generated one character at a time.

The Objective

Recurrent Neural Networks are able to pick up patterns in natural language structure like the spelling and positioning of alphabets and characters used to form new words, phrases and sentences. Based on the structure of the text input supplied, the model tries to predict the probability of the next character. In this case, since temperature is used to either create deterministic or probabilistic outcomes, there may be different outputs every time, even if the model is fed with the same input.

The main objective of this assignment is to build a character generation model to generate a semi-coherent English sentence from scratch. In my opinion, the best model should:

- Have a high final epoch training accuracy
- When I visual assess the words generated at **last epoch** for
    - Temperature = 0.5 and
    - Temperature = 1.0,

  The model should have a high percentage of english words since the objective is to predict a semi-coherent sentence.
- Appropriate punctuations placements
- Distributions of words and frequency of English words (check for repetition)
- Be able to generate a label that can possibly be used construct and English word given a new input

The Approach

I chose to experiment with models based on their different architectures (LSTM and GRUs) and I also played around with different window sizes. For the three different window sizes that I experimented with (100, 200 and 300), I started off with a base model for each unique model architecture type and thereafter proceeded to conduct hyperparameter tuning in order to improve the model's performance.

I then went on to shortlist a few models for further evaluation based on high final epoch training accuracy, as well as results from the visual assessment of the words. Based on the results from the testing, I then went on to find out why the model does well/does not do so well in accurately generating the next character for a given input supplied.

7. Data Loading & Processing

Data Loading

The first step in this problem is to load the data from the dataset, which in this case is holmes.txt. The function below is used to read the dataset file and transform all the uppercase alphabets into lowercase using the .lower() function.

```
# 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 562439 characters.
```

Data Processing and X/y split

Thereafter, the data is cleaned using several methods. Firstly, the newline characters (\n) and the carriage return (\r). Then I created a function to only extract characters based on set of the allowable characters, which are lowercase alphabets and punctuation marks. This is implemented using an if statement, which is used to check whether each character in the dataset is in the punctuation list or the string of letters and they are saved to a new string char_list as long as they fulfil one of these criteria.

```
def clean_text(text):
    punctuation = ['!', ',', '.', ':', ';', '?', '-', "'",' ']
    letters='abcdefghijklmnopqrstuvwxyz'
    char_list = ""
    # Enter your code here:
    for i in text:
        if i in punctuation or i in letters:
            char_list += i
    return char_list
```

The characters remaining after the cleaning are thereafter sorted and displayed in the chars list.

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

```
[' ', '!', "'", ',', '-', '.', ':', ';', '?', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v
', 'w', 'x', 'y', 'z']
This document has 544340 total number of characters.
This document has 35 unique characters.
```

```
# 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
```

Thereafter, a dictionary mapping is created between characters and the corresponding integer indexes in the chars list.

We are now ready to move on to generate the inputs and the labels using generate_text_io. Firstly, two lists to store the input and labels respectively are initialised. Thereafter, a for loop loops through the entire dataset to extract the sentence (input or X) based on the window size, specified by the argument maxlen. For each input, the function will also extract the corresponding label (in this case the character after the length of the window size). Thereafter, 3 characters thereafter will be skipped and the loop will thereafter extract the next input and labels starting from there. Using step allows us to create new training data and thus is more useful, as compared with just starting right after the label.

```
step = 3

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

    # Enter your code here:
    for i in range(0, len(text) - maxlen, step):
        inputs.append(text[i: i + maxlen])
        labels.append(text[i + maxlen])
    print('Number of sequences:', len(inputs))

    # List of unique characters in the corpus
    chars = sorted(list(set(text)))
    print('Unique characters:', len(chars))
    # Dictionary mapping unique characters to their index in `chars`
    char_indices = dict((char, chars.index(char)) for char in chars)

    return inputs, labels, char_indices, chars
```

Now, we are ready to perform one-hot encoding on the inputs and labels. Window size is also specified as an argument as this is important in determining the initialized 3D input vector, which will be first filled with zeros using the np.zeros() method as shown below. Thereafter, depending on the particular positioning of the character based on its sequence in the input (e.g. e is the second character in “hello world”) as well as the index of the character in the chars dictionary, a “1” replaces the 0, which indicates the presence of a particular character. In this way, one hot encoding is performed and a similar process is repeated for the corresponding labels.

```
import numpy as np

# create a function 'encode_io_pairs' to perform one-hot encoding of inputs and labels
def encode_io_pairs(text, maxlen, inputs): # window_size determines # of characters in each input

    # Enter your code here:
    print('Vectorization...')
    x = np.zeros((len(inputs), maxlen, len(chars)), dtype=np.bool)
    y = np.zeros((len(inputs), len(chars)), dtype=np.bool)
    for i, sentence in enumerate(inputs):
        for t, char in enumerate(sentence):
            x[i, t, char_indices[char]] = 1
        y[i, char_indices[labels[i]]] = 1
    return x, y
```

The code below shows how I used different window sizes and shows the inputs and labels after it is returned from generate_text_io and after it is one-hot encoded using the encode_io_pairs function. In this case, False represents a ‘0’ and True represents a ‘1’.

```
#window_size = 100 # Models 1-7
window_size = 200 # Models 8-14
#window_size = 300
inputs, labels, char_indices, chars = generate_text_io(text, window_size)
print(inputs[0] + '\n' + labels[0])
X, y = encode_io_pairs(text, window_size, inputs)
print(X[0])
print(y[0])
```

```
Number of sequences: 181380
Unique characters: 35
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 boscomb
e valley mystery   v. the five orange pips  vi.

Vectorization...
[[False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]
 ...
 [False False False ... False False False]
 [False False False ... False False False]
 [False False False ... False False False]]
[ True False False False False False False False False False False False
 False False False False False False False False False False False False
 False False False False False False False False False False False False]
```

One hot encoding is used in this case since the number of unique possible character labels are small thus this representation is suitable to be used in model training.

Since there are different probability distributions for each batch of inputs and temperature is used in this case to either create deterministic or probabilistic outcomes, thus another function is needed to sample an index from the probability array and this will ultimately help to determine the corresponding label and hence output the character predicted, which will be shown later on in the report. In this case 1.0 is used as the default temperature value.

```
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)
```

Sampling is important in this case because the output (or character generated), is not only determined based on the label with highest probability, because temperature is taken into account in this case.

8. Develop the Character Generation Model

All the models have a Dense layer at the end. The number of output nodes in this case will be determined by the length of the chars list, which is in this case, 35. However it is a bad practice if we just write 35 here in case, we want to change the number of allowable characters in future. The activation function used in all models is categorical crossentropy since the problem is a multi-class, single label type of classification. The optimizers used will be defined later on in the respective models and accuracy is used as a metric for model evaluation and this also allows for both the training accuracy and loss curves to be plotted later on.

```
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
```

I also decided that since I would be training multiple models and using all of them for prediction and further evaluation would be time consuming (as elaborated in the next step of this Problem), I would shortlist best 5 models for prediction with the following criteria:

- Have a high final epoch training accuracy
- When I visual assess the words generated at **last epoch** for
    - Temperature = 0.5 and
    - Temperature = 1.0,

  The model should have a high percentage of english words since the objective is to predict a semi-coherent sentence.

In addition, since more coherent sentences generally will have more instances of appropriate placement of punctuation, I decided to just shortlist the models which have a high percentage of english words for predictions of the labels later on in this report.

Model #1 LSTM RMSprop Base Win Size 100

For the first model, I used a Window Size of 100. I chose an RMSprop optimiser, with a single layer of LSTM with 128 nodes. The initial learning rate is set at 1e-2. The input shape is set to the product of the window_size (which is 100 for this model) and the length of the chars list (which is 35).

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

The code below shows the fitting of the model, as well as the generation of the 400 characters repeatedly with the use of the for loop based on the different temperatures. I also included some additional variables and loops which help to collect and store the accuracy and loss scores for plotting later on.

```
import random
import sys

acc_list= []
loss_list = []

for epoch in range(1, 6):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=128,
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

The model is fitted and ran for 5 epochs. In this case X which represents the inputs, y which represents the labels, a batch size of 128 and epochs is set to 1 since the for loop already helps us simulate training for 5 epochs in this case.

From the dataset, a random start index is generated. Thereafter, the generated text will contain the 400 characters generated based on the starting index.

Code for Plotting (refer to code above): acc_list and loss_list are firstly initialized to collect the accuracy and loss values at each iteration of the loop as shown in the code above.

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

            preds = 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()


graph_acc = []
for i in acc_list:
    for j in i:
        graph_acc.append(j)

graph_loss = []
for i in loss_list:
    for j in i:
        graph_loss.append(j)
```

Thereafter for temperatures of 0.2, 0.5, 1.0 and 1.2, 400 characters will be generated. To accomplish this, firstly one hot encoding is performed on the inputs. Thereafter the sample function is called to make a prediction based on the probability distribution generated by calling predict() on the one hot encoded input array. Thereafter the next character is generated and this will be added to the original generated_text and the first character of the original is removed (indicating a shift by 1 character).

Code for Plotting (refer to code above): I initially found out that the individual accuracy and loss values appended to the lists as 1D arrays themselves (code snippet below), which made the lists 2D arrays so I needed to find a way to remove and re-add them to another list to make it 1D, which is accepted as input into the plt.plot() function.

```
In [20]: print(acc_list)
[[0.41414666], [0.5016206], [0.53146946], [0.54662263], [0.55652267]]

In [21]: print(loss_list)
[[2.0013530449513093], [1.6765556476123724], [1.5782622638687547], [1.523333394588021], [1.4854275606713825]]
```

Plotting Code

```
#Plotting training graphs
%matplotlib inline
import matplotlib.pyplot as plt

epochs = range(1, len(graph_acc) + 1)

plt.plot(epochs, graph_acc, 'bo', label='Training acc')
plt.title('Training accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, graph_loss, 'bo', label='Training loss')
plt.title('Training loss')
plt.legend()

plt.show()
```

The code used for the plotting of the accuracy and loss curves are largely similar compared to Problem 1, except that there is no validation loss or accuracy, since there is no validation data and labels allocated for this problem. I also had to use len(graph_acc), which is the length of the cleansed training accuracy list which I used for the plotting instead of the variable 'epochs' which is set to 1 and so it will not work iterating through the list with the help of the range function in this case.

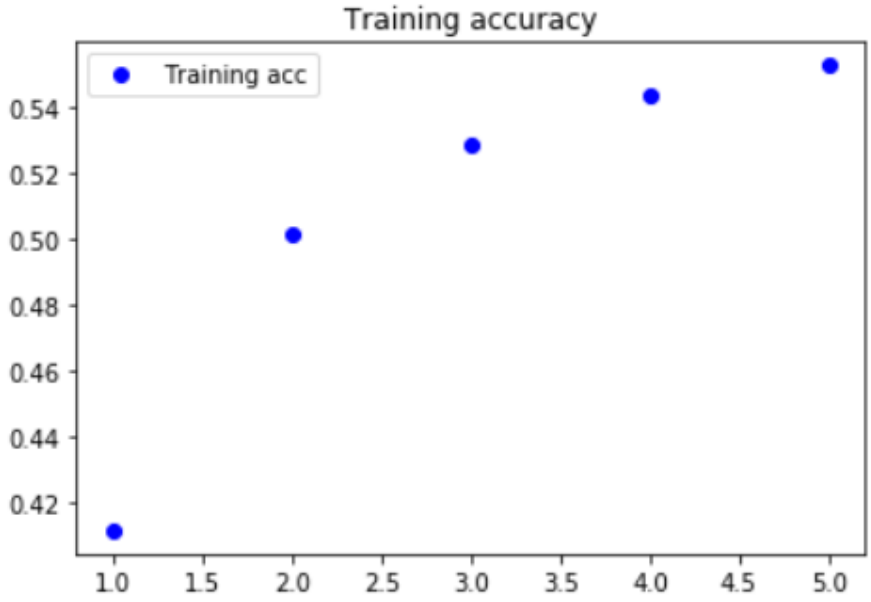
Note that the fitting and plotting code are all the same for all of the models, except for models when I changed the batch size when fitting the model as part of hyperparameter tuning. In those cases, I will include a code snippet of the batch size.

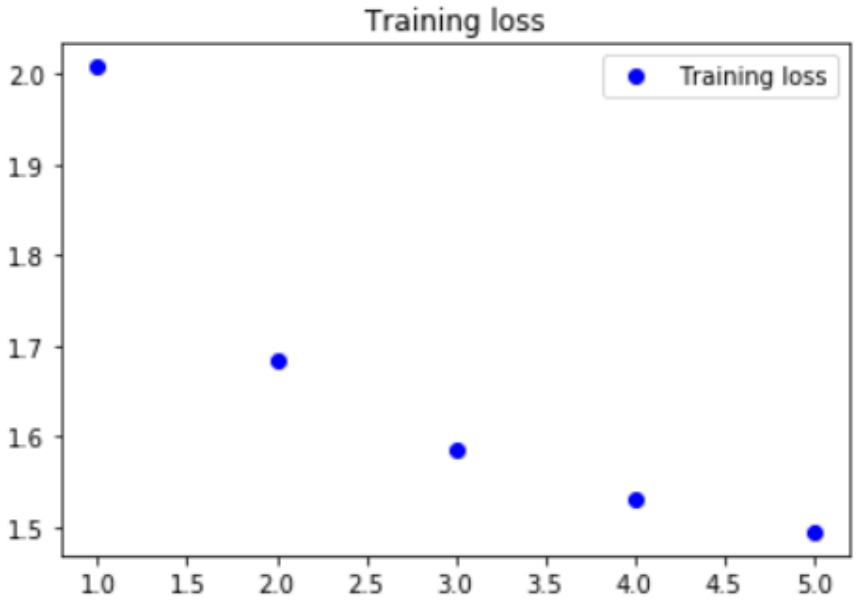
```
------ temperature: 0.5
 have seen to the street. i have been to the lady stain to her to get to the station. i have a street. the laing. i could see when i have considerate wit
h a street, on the bed about to the stant, and she had a fawn and such standing offered to come to be as we should see it and she has been an a point-roo
m with me, whise conding to the place of the light to come, said holmes. this she was a dress. i have profession to the windows were where i had been int
o the back of the strong with my company
------ temperature: 1.0
 i have profession to the windows were where i had been into the back of the strong with my company wasen hegrient, pervelanogrese pon ontened hy. you wi
nd, there wore. she had can carns of whealer could was assicing of his faies and pack of it srowly and by fattle to discut of thescaim, when to do you ex
pleaed. i would thentasykeen, tood to this adgertrest, as i gentyenow bedaged up the passing. visiting should, said he would come. she was  whom you have
emend, does, then clear these, howes, an
```

After briefly scanning through the words generated by Model 1, I found that

- At temperature = 0.5, approximately 90% of the words are english
- At temperature = 1.0, approximately 65% of the words are english

I was fairly impressed by the results of this model since it was able to generate rather coherent English sentences. Thereafter, I moved on to shortlist the model based on its final epoch training accuracy.





As seen from the accuracy curve produced by the model, the training accuracy starts to increase at a slower rate starting at 4 epochs, which may indicate that it may start plateauing had I trained it further for over 5 epochs. This may indicate that the model is soon to be reaching its maximum possible training accuracy.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 24s 134us/sample - loss: 1.4945 - acc: 0.5528
```

As seen from the output below, the model took an average of 24 seconds for training per epoch and it achieved a training accuracy of 55.28% and a loss score of 1.4945 after 5 epochs of training.

Model #2 LSTM RMSprop Add layer Win Size 100

For this second model, I decided to start simple with the fine tuning and simply add a layer of 128 nodes to the LSTM Model. I kept the rest of the hyperparameters the same, including the initial batch size of 128.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(128, return_sequences=True, input_shape=(window_size, len(chars))))
model.add(layers.LSTM(128))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

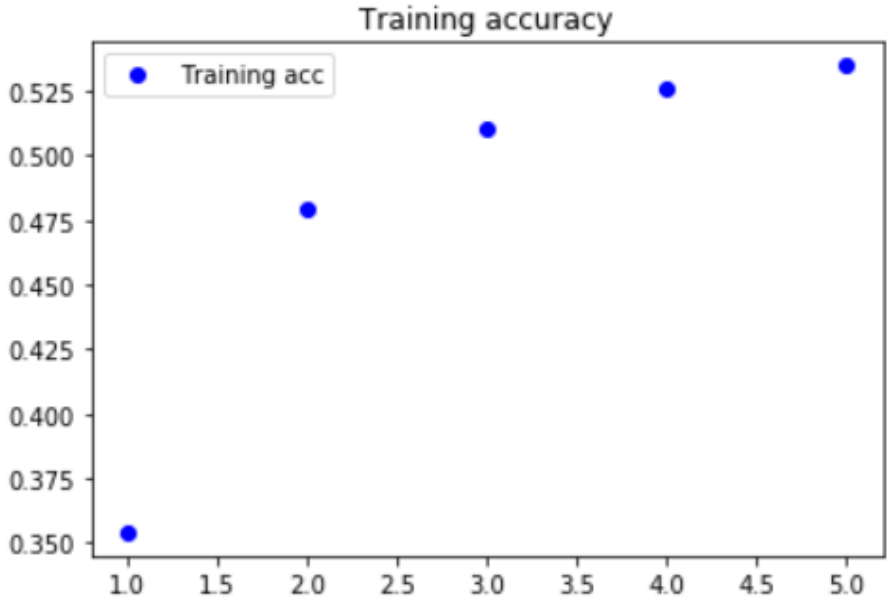
I was honestly unsure of what to expect from this model since it was my first time experimenting with the hyperparameters.

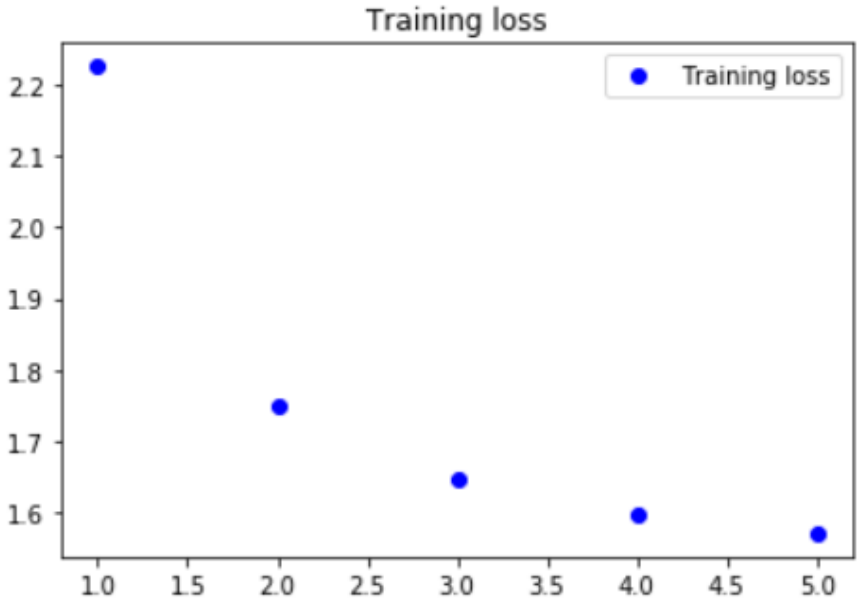
```
------ temperature: 0.5
the way to have had seven such a case and my commantion and the commanted the start the stark the word stone a clain the disge, the story. then i was mor
e the commantion and the came that she say it at heark have been and in the concice--that and your short the complished suggess but safe it as say we hav
e summest well, and i have been severing in the man was the passed the shatter and back as in you as i was sime the case, said he was and of sat the coro
nes and when we do you smck. i had set in
------ temperature: 1.0
you as i was sime the case, said he was and of sat the corones and when we do you smck. i had set inthe innot attont.it was excepts and youghtthicked by
the falter bumge shars ahaught a strange have carkcivel, holmeswing anotheree of sever the care, who isso, goodcallang adlast anddrear.mychurly gone, sii
veing buming find the miming, you sot as her stepresid awobfiving chaisably heard. she can bebefore? fuld on figure.ithin limed, perhaps luablechas was t
hick, by been eldbarncicaly when i had ca
```

After briefly scanning through the words generated by Model 2, I found that

- At temperature = 0.5, approximately 75% of the words are english
- At temperature = 1.0, approximately 50% of the words are english

Thus, in terms of generating the semi-coherent English sentence, model 2 performed much worse as compared to its base model (Model 1).





Model 2 also achieved a lower final epoch training accuracy of 53.5%, which is almost 2% lower as compared to Model 1. It also has a higher loss score of 1.5706 and takes more than twice as long to train as compared to model 1, with an average training time of 52s per epoch. However, the similarity drawn from the 2 models is that the accuracy starts increasing at a decreasing rate, suggesting that maximum accuracy of the model may be reached soon.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 52s 286us/sample - loss: 1.5706 - acc: 0.5350
```

Since increasing the number of layers resulted in a poorer final epoch accuracy and had a negative effect on the coherence of the words generated, I decided to remove the additional layer and test out other hyperparameters instead.

Model #3 LSTM Adam Base Win Size 100

For this model, the general architecture is exactly the same as the Base Model (Model 1), with the exception of the optimiser. I switched from an RMSprop to an Adam optimiser to assess if this would have a positive effect on both the final epoch accuracy, as well as the percentage of English words, which determines the coherence of the English sentence.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

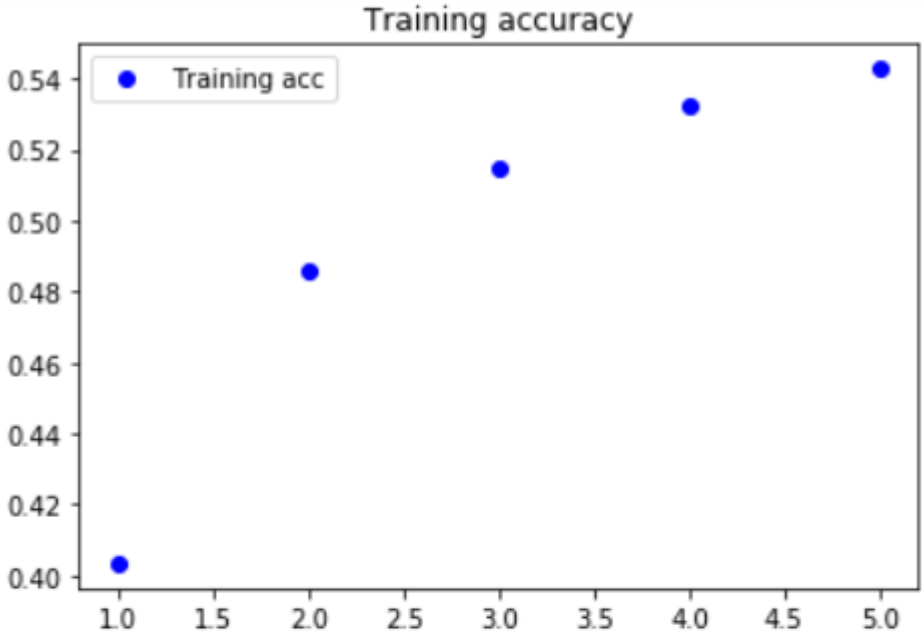
I kept the batch size the same as Model 1 at 128.

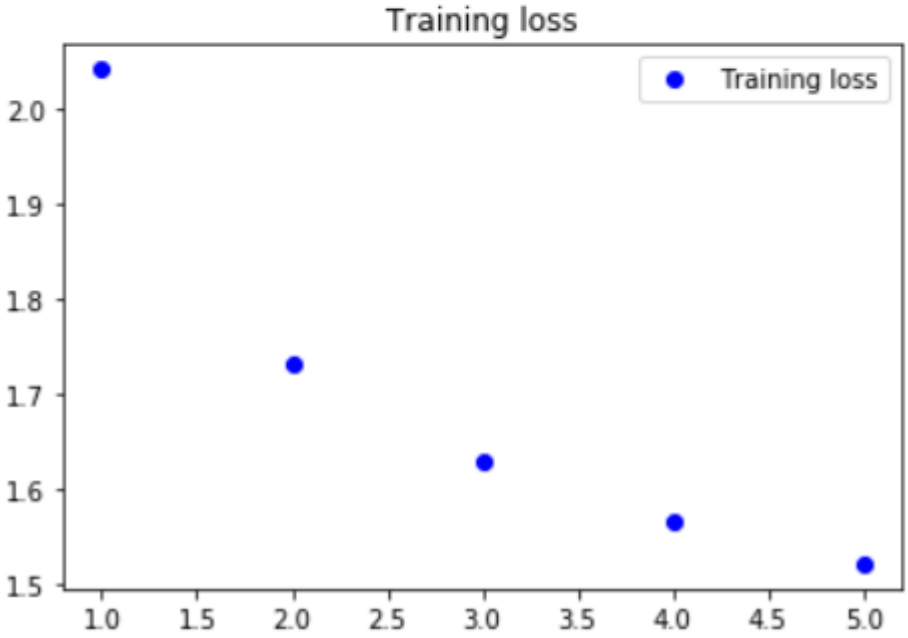
```
------ temperature: 0.5
d started to me and stood the stark and stood and there is the man and stood and the started his hands. i do the passed at the hands. we was there was so
me once me of the ways of the finition which is the clawing been sat a gentleman, mr. more in the stone of the moten upon the way have early on the closi
culary little and clear. there was a recaully there would seen as i was house the carriagenent and were for us his man was for minutely and she has alrea
red up, and not the man and to a munding
------ temperature: 1.0
 and were for us his man was for minutely and she has alreared up, and not the man and to a munding andwhich if very nature up either inethens. you would
to uster, and he stepped out, to mr. starking objecta was of which is glanching?and 'you cannot a corck. i relick into the clawthy pare struke of a myste
ry. mored. we may not aga, but but lies to arrocter. me, but it windows as to the swimbub of the busing all the stout positions depore the sat enting up
of which you do.it can undarkly befores
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 90% of the words are english
- At temperature = 1.0, approximately 60% of the words are english

Compared to the RMSprop Base Model, this model had fewer english words in the last two lines generated at temperature 1.0, which I found interesting. I concluded that varying the optimiser could have such a drastic effect on the coherence of the language generated.





For this model, the model seems to only start plateauing at 4 epochs, as there is a noticeable difference of about a 2% increase in accuracy compared with the 1.5% from epoch 3 to epoch 4, as compared with the previous models using the RMSprop optimizers. After looking at the final epoch accuracy, which is 54.28%, I hypothesized that there is indeed a positive correlation between the final epoch accuracy as well as the percentage of english words in the generated 400 characters.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 28s 152us/sample - loss: 1.5214 - acc: 0.5428
```

Model #4 LSTM Adam Add layer, increase learning rate Win Size 100

Thereafter, I tried to fine tune the Adam model by experimenting with a slightly different architecture. For this model, I used 2 layers of 64 LSTM nodes, because from model 2, I realised that adding more nodes would have a negative effect on both the accuracy as well as the coherence of the text generated. Thus, I reduced the number of nodes to 64, hoping that this would not severely affect the accuracy of the model. I also increased the learning rate to 5e-2 from 1e-2, with the batch size still kept the same.

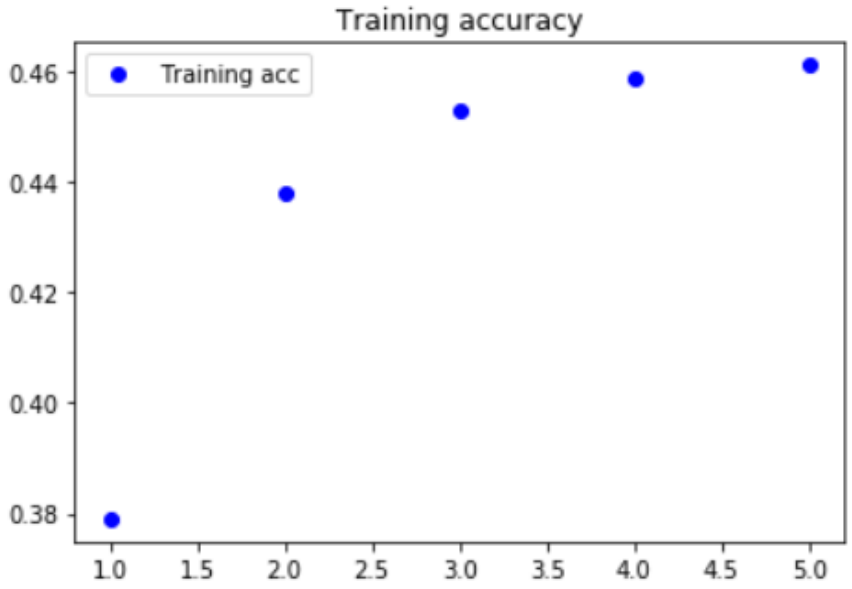
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(64, return_sequences=True, input_shape=(window_size, len(chars))))
model.add(layers.LSTM(64))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.Adam(lr=0.05)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

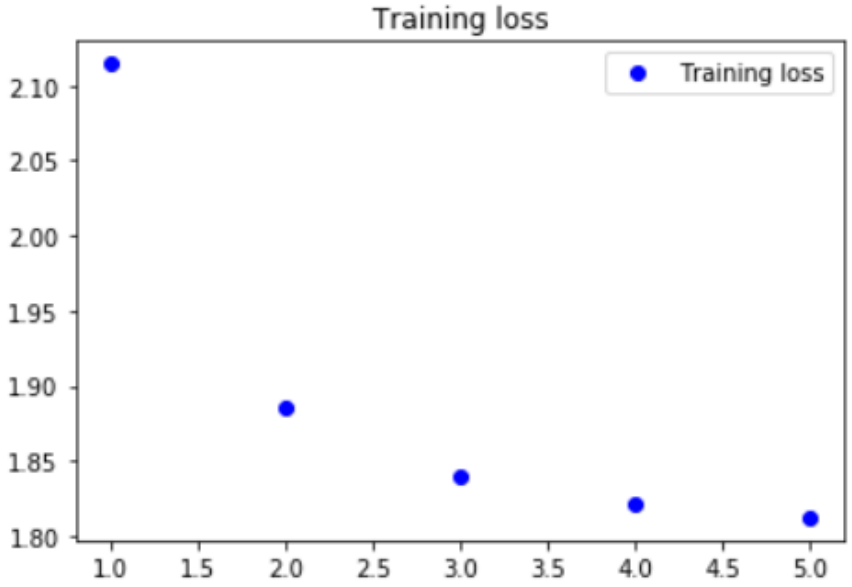
```
------ temperature: 0.5
in the has the clair the sure to the pound the compart and man and pain the would in the compain the mied the bank felly as are what it to be into feliou
s back a priewy to the dress for the would of the comperfrant to the positing at the proper be in what when the compand to the offarked that the doubt ha
nded him as said is a round cire who the tinded the dember to the time to the think him to the round it to the prufess that i mert a shall maniad as him
the could the more have been the state in
------ temperature: 1.0
 round it to the prufess that i mert a shall maniad as him the could the more have been the state inthere be jusicairfour everlay a clainded before wereh
all him has insque.dealuites of tine you could inscectrrew of thome an stase but i came outne that i carnerenturn it irune a know, he hadsis it?he wux i
five streep rutrarried sided at the would perhaps a gupeds writorlyfor in his is where not aen.yes, shag, the rusiatry, have maducly a urn, a tooups:to t
he buiecine they lar a the roke-o leottio
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 60% of the words are english
- At temperature = 1.0, approximately 45% of the words are english

The text generated started to have more variety in this instance as the model tries to generate “new” english words, but the model deproved further from base model in terms of its performance so I decided to instead tune other hyperparameters, in the search for the best character generation model.





The model's curves have a similar shape to the RMSprop optimizers and the training accuracy starts increasing at a decreasing rate starting at 3 epochs. This model achieved the worst last epoch training accuracy so far of 46.13% and quite a high loss score of 1.8122.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 29s 159us/sample - loss: 1.8122 - acc: 0.4613
```

Model #5 GRU RMSprop Base Win Size 100

Since adding additional layers and changing the learning rate or optimiser did not work out as intended, I decided to then go on to explore using Gated Recurrent Units (GRUs) instead. For the base model I have a layer of 128 nodes and an RMSprop optimiser with a learning rate of magnitude 1e-2, as I was trying to compare if there was progress made comparing the LSTM and GRU base models. I kept the Window Size at 100.

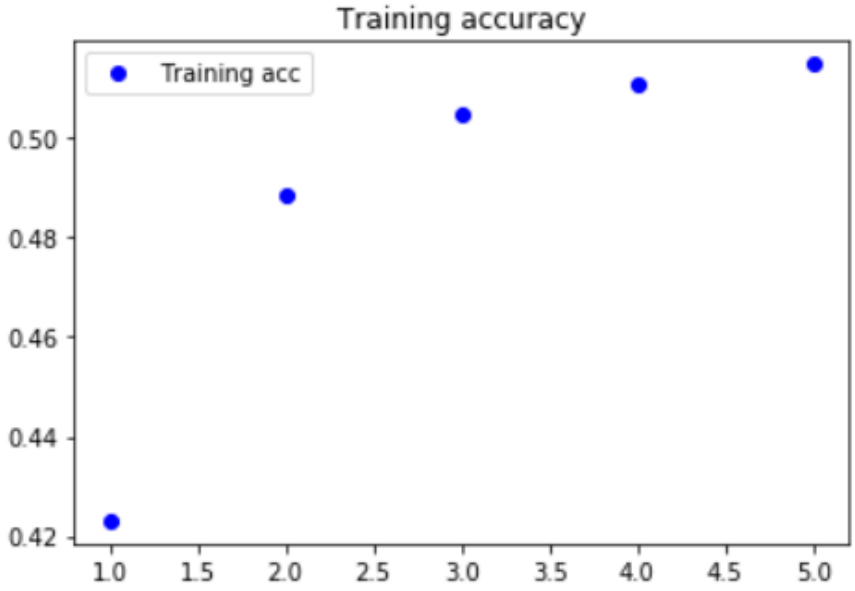
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

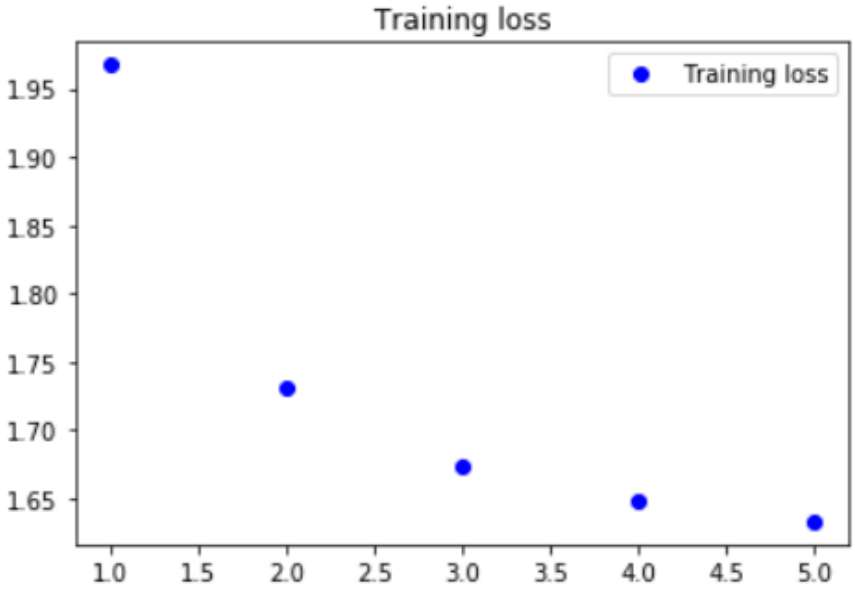
```
------ temperature: 0.5
 in the pain of in the stand in the station of the stand with a station of the stand windibbull masted upon which i shall and so my since, sherrow stand,
in with a most man in the and of a break of bettered from the as in the some with the course very some station. i was not mr. she was a droves and she co
uld have not started to give to a coust some had the standuch sit of the said home in his in the pass of the was are and comparines of the indown than in
the statter and so the some as in the s
------ temperature: 1.0
the pass of the was are and comparines of the indown than in the statter and so the some as in the stair of in spall nawress. in cried the foors whavill
greenus. will see no firstduces what is found she could a gleatchen.to therenow, s if r hattwey whichsideas whise you ask he fushearew, said he much mean
of this timpessionivand with alons of the ofthere, anditnebelive bromenihas necklonglea to blover with that i was found as he have admrat in awring only
for our could such i aptee of me, what w
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 85% of the words are english
- At temperature = 1.0, approximately 55% of the words are english

The GRU model fared slightly worse in this aspect as compared with the LSTM Base Models. This is perhaps due to GRUs having lower performance for the same model architecture due to it merging the hidden and cell states and thus it has 2 new sets of weights which can be used to better optimize model performance (Koupanou, 2019).





This model achieved a training accuracy of 51.49%, which is poorer as compared to LSTM Base Models. It also achieved a loss score of 1.6333 after being ran for 5 epochs. I also noticed that GRUs generally take less time to train per epoch and this perhaps is due to the simpler operation performed in the GRU structure.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 23s 125us/sample - loss: 1.6333 - acc: 0.5149
```

Model #6 GRU RMSprop add layer & dropout Win Size 100

In this case I made some changes to the base model. I added an additional layer with 64 nodes and added dropout and recurrent dropout of magnitude 0.1 in this case to help to circumvent the model from reaching its maximum training accuracy too quickly.

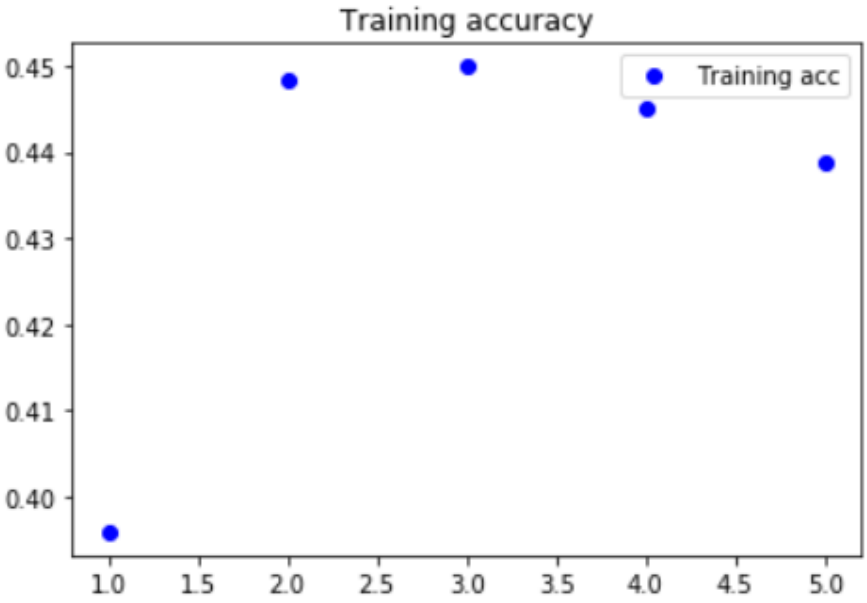
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.GRU(128,return_sequences=True, input_shape=(window_size, len(chars))))
model.add(layers.GRU(64, dropout = 0.1, recurrent_dropout = 0.1))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

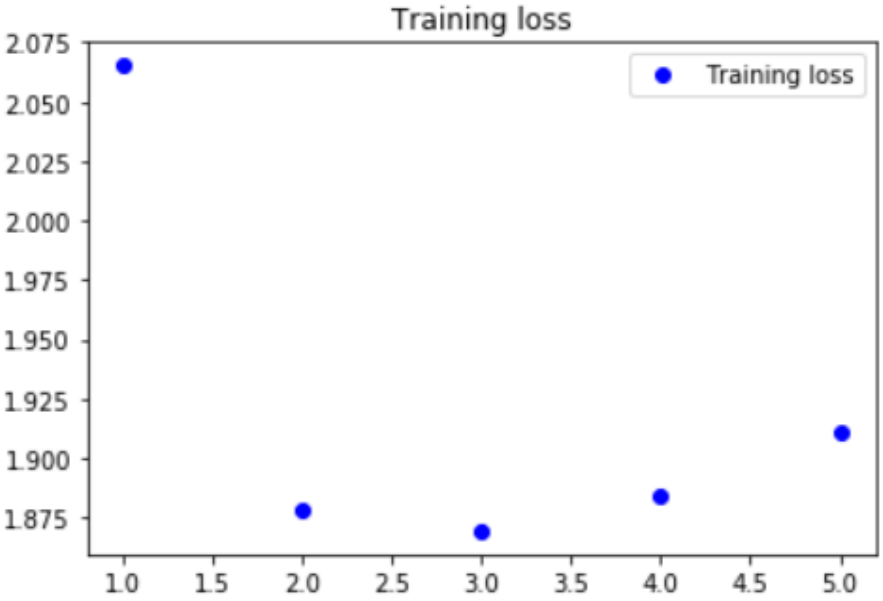
```
------ temperature: 0.5
ome the may the canding the come the come the mad the come the clair the come the come the may the cears. the mark may competing the cotiress which a can
g the may me the marring may the the pothing a the can not the come that it and mark it have ce can this him the had may reasit. the come a my of the cla
in may clinien the mastir the contion the contint the clair my a mation to the colarting the may compish that is my have it is custicurs. the matter math
erg in the crain the clair you come to me
------ temperature: 1.0
ay compish that is my have it is custicurs. the matter matherg in the crain the clair you come to me the case was watsat the littubevert of mewn mingiof
yout.yes ing itof, mucts and poeting him that, handear gromo an moy andce.when mine destition toley tion tried.itis icappwight the did at it impoint, iwh
oad it is nencil my at you reasing etelarce lymay it.uher, in and they coulnt! cor caning tele cuitestor khorywrime men ththat. ce cair thick a dainaning
cent at timeg'tatios thatin i ceathiwh i
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 65% of the words are english
- At temperature = 1.0, approximately 45% of the words are english

There was also much more repetition of words as seen in lines 1 and 3 at temperature = 0.5 in this case and the model performed worse as compared with the GRU Base Model (Model 5) in generative semi-coherent English sentences.





In this case, the model has seemed to have reached its maximum training accuracy quite early on and thereafter the training accuracy starts to decrease. Since this might have been caused by either the increase in the number of nodes present, I decided to further explore this phenomenon later on. The model has a poor final epoch training accuracy of 43.88% and achieves quite a high loss score of 1.9106.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 252s 1ms/sample - loss: 1.9106 - acc: 0.4388
```

Model #7 GRU Adam Win Size 100

Since using dropout with the increased number of layers did not help to improve model accuracy, I instead decided to try a different optimiser to check if this would have the same effect (a lower final epoch training accuracy) on the GRU model as it had on the LSTM architectures. For this I kept the rest of the hyperparameters the same, including the batch size of 128, the Window size of 100 and ran it for 5 epochs, with the exception of the optimizer I used, which is Adam instead of RMSprop.

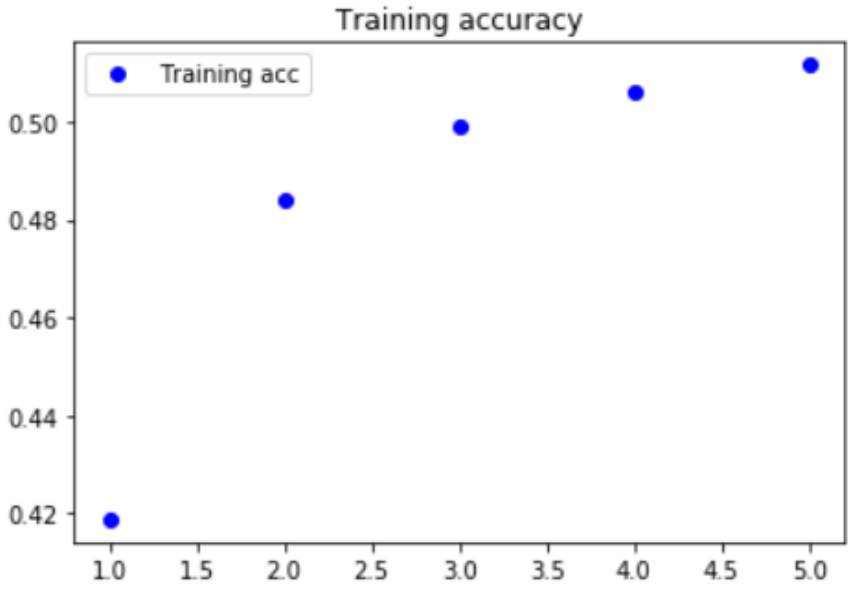
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

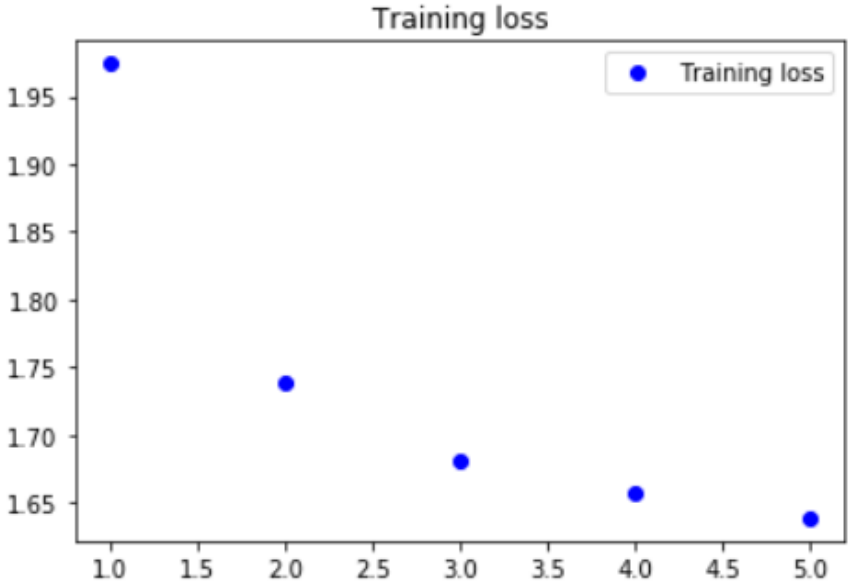
```
------ temperature: 0.5
in which he day which was the man which was the carriage to me the look and was a fear which we was it were to him. i did you are in the dank who might b
e in the morning a very advention of the daught and side to me the land strengh and the figgor which as the bere with the dark where we should one the da
ught of live the light two in the daught and past the far she was a fear the ground dressing to the door her figaring it with the gester and the wards an
d the down the other the police and get w
------ temperature: 1.0
o the door her figaring it with the gester and the wards and the down the other the police and get weus mese ear dout in clack. so soan ise stone in kill
of luster seemone me matogress.cltandas wemcitied common and the ride, theugh me mach. no known to where mares in the colonel ania scurhing to behouchand
ovelone a i in me nots. he was we little an eed his waileage in st, as which sey tin! miss he coldming it.yoursobraid, suth one we can so down, which sad
marry buling be to me.i had while you
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 90% of the words are english
- At temperature = 1.0, approximately 65% of the words are english

This model generally did much better in terms of the generation of english words and also had less repetitions observed as compared to the models that were compiled with the RMSprop optimizer.





However, the final epoch training accuracy was slightly lower than that of the RMSprop Base Model at 51.17% as I expected. This model was quite different from the general trend I inferred from the other models where higher final epoch training accuracy generally leads to better performance in generating coherent english sentences.

```
epoch 5
Train on 181414 samples
181414/181414 [==============================] - 20s 112us/sample - loss: 1.6384 - acc: 0.5117
```

Since I tried out several hyperparameters already, I decided to try out a different window size of 200 instead of 100 (which requires rerunning generate_text_io and encode_pairs_io).

Model #8 LSTM RMSprop Base Win Size 200

For this base model I used an LSTM layer with 128 nodes with an RMSprop optimiser and a learning rate of 1e-2. I ran the model for 5 epochs with a batch size of 128. In this case I also changed the window size to 200 as I am required to experiment with different window sizes as part of my approach to this problem.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

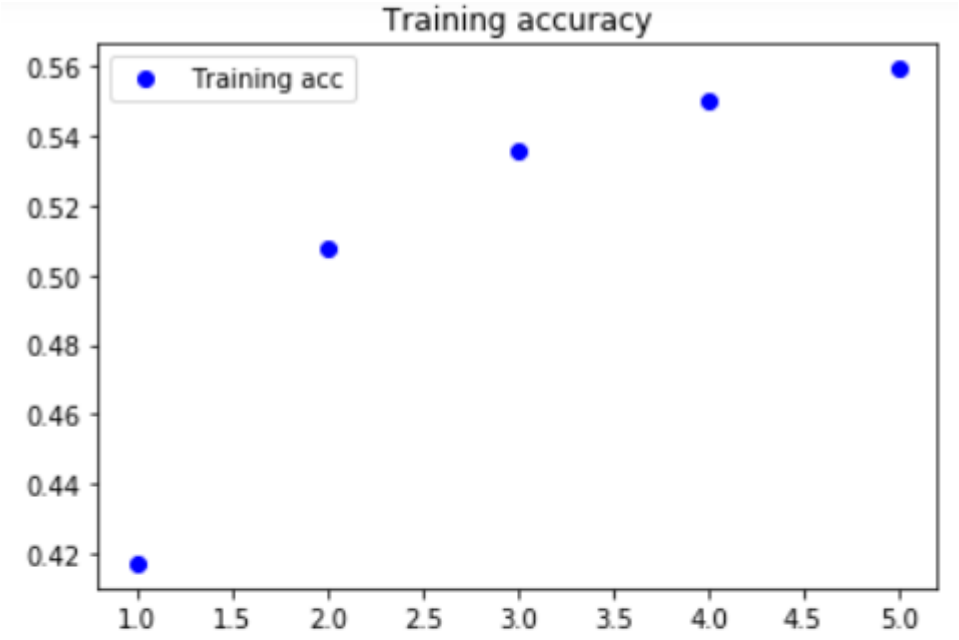
```
for epoch in range(1, 6):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=128,
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

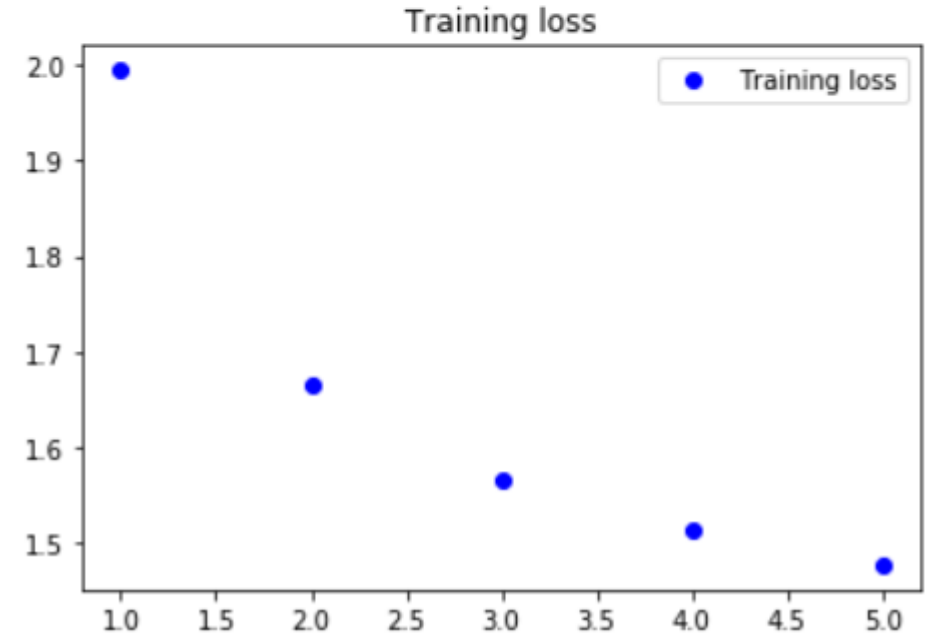
```
------ temperature: 0.5
 think that i think that i think the last man in my day stone the strange of the care of the carried and the leagual of the other way and the brought of the strange of the stawed the stant of the care. who is the some side of the stares and that it was but her some grey gave a thrugg lear a matter that they was a man marker your step of his armshord when i have far a start and the confider of the intiller. he was on the clair of a little man who street of a few what you have throw it as be no drifting his change to mean large companion as whetide with me the light, and he was attered to surpre
------ temperature: 1.0
intiller. he was on the clair of a little man who street of a few what you have throw it as be no drifting his change to mean large companion as whetide with me the light, and he was attered to surpredural mectsome toom, yet away ineversament, raven that yourevisely man; said holmes, said he,  then-cecto us a time to bushes wisless he right?me should only basorver per againce, and could he.'that herely irater to fetter inkant rads, it was a tadi the ratter, but i gonebaggered snous man,i can oylught there was murded frand of an brisey on onerward for the t becapisaw late but lugily. standa tte
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 85% of the words are english
- At temperature = 1.0, approximately 60% of the words are english

I would consider this model's performance to be average in this aspect as compared with the other base models.





The model achieved the highest last epoch accuracy of all the models so far of 55.91% and the lowest loss score of 1.4773, although the performance in generating semi-coherent english sentence was rather average. It was also another model which didn't follow my hypothesis of the correlation between the final epoch accuracy as well as the percentage of english words in the generated 400 characters. However, with the other hyperparameters kept constant, the model was still able to achieve a slightly higher accuracy as compared to the RMSprop LSTM model trained using the smaller window size of 100.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 46s 251us/sample - loss: 1.4773 - acc: 0.5591
```

Model #9 LSTM RMSprop Change learning rate, add dropout Win Size 200

For this model, I chose to keep the number of layers at 1 and add a lower magnitude of dropout to see if this would have a positive effect on the final epoch training accuracy as well as the quality of the words generated. I also increased the magnitude of the learning rate to 5e-2.

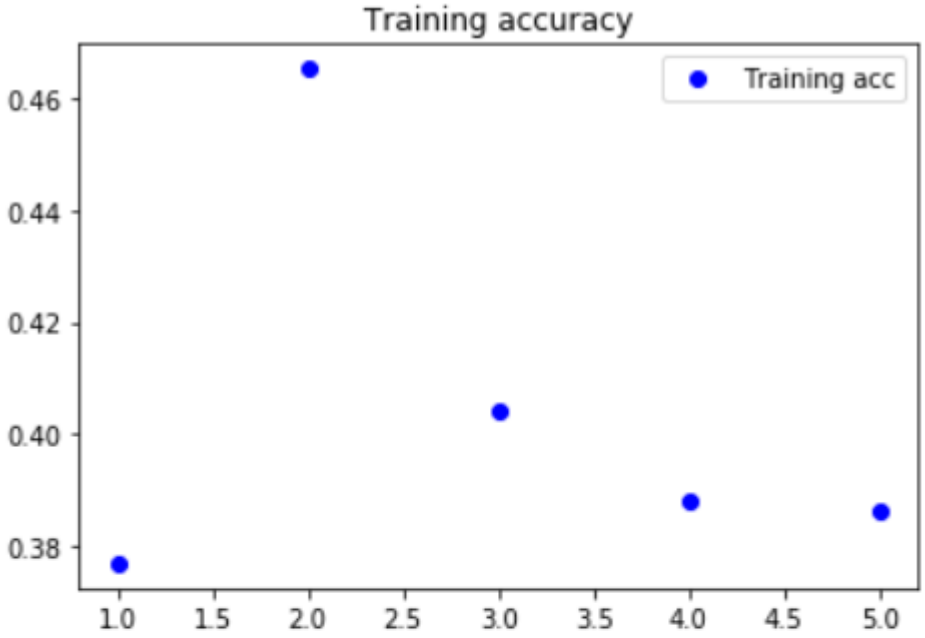
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(128, dropout = 0.005, recurrent_dropout = 0.005, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.RMSprop(lr=0.05)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

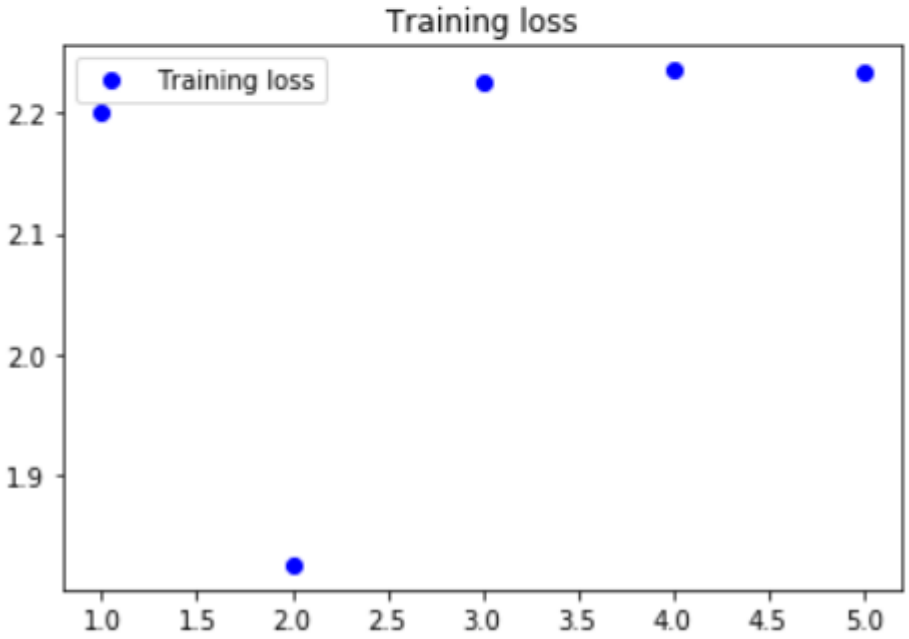
```
------ temperature: 0.5
so so the so the sooe state oou s st is the it she sole she so the stathe s st st to so so the st the so so st st that is y ofooot who she so the sound s
he she sat the so the sound the st the se some shole as aor the lts weple to se sh shall alook was a mol he gord be the so to the oshou st the so be had
stad hor sfost bat i  what succe a sile come liched to had of t pormestors  the shall seess the seed. comme awas stat his oft we por he she mo notwer the
side to the some a wing, be the saies s could and this sace can be ad a sook se and the case that a chow the sound the to some so the stard
------ temperature: 1.0
e seed. comme awas stat his oft we por he she mo notwer the side to the some a wing, be the saies s could and this sace can be ad a sook se and the case
that a chow the sound the to some so the stard mpor perad in ly hadrolkedt she could atsply, dinso s. he wo st. bluseco, yarss nougheprass ved someeethin
g, gote, a sedagw bat it she coule deighof ocliesnsewhe mockmand, sor rofu oson his heow shealconou wothter walo ttll ticessablirk os your mass tell. fo
n?it the casimvat to ho. persimer ofputtear to se ton the bey chadlholmessalasaydian, and himeahellgiato noweve poski, flis-o ncstigerce to go
```

However, this did not turn out as expected. After briefly scanning through the words generated by the model, I found that

- At temperature = 0.5, approximately 45% of the words are english and there are lots of repetitions on all the lines generated
- At temperature = 1.0, approximately 20% of the words are english

Thus, I concluded that even a very low magnitude of both dropout and recurrent dropout had such a drastic and negative effect on the coherence of the text generated by the model.





The training accuracy of the training accuracy rose very sharply from 1 to 2 epochs and subsequently plunged to about 46.5% and then plunged at 3 epochs before further tapering off and plateauing later on. The model achieved the lowest last epoch accuracy so far of 38.6% and a pretty high loss score of 2.2344.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 477s 3ms/sample - loss: 2.2344 - acc: 0.3860
```

Model #12 LSTM RMSprop change learning rate Win Size 200

For this model I experimented with increasing the number of nodes in the sole LSTM layer from 128 to 512 to see if this will positively affect the accuracy. I also changed the learning rate to a smaller magnitude of 1e-3, down from the previous 1e-2 in the LSTM base model.

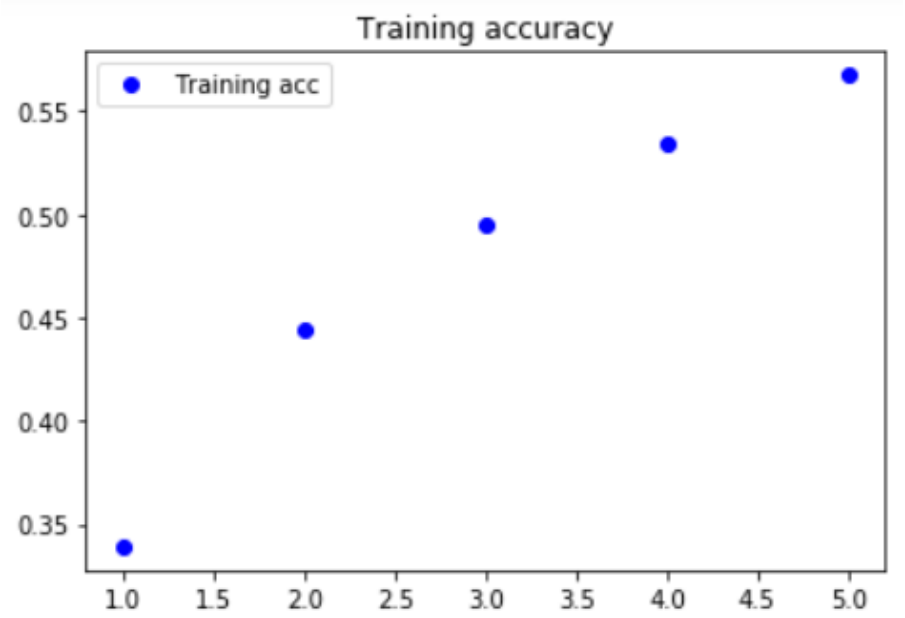
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(512, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

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

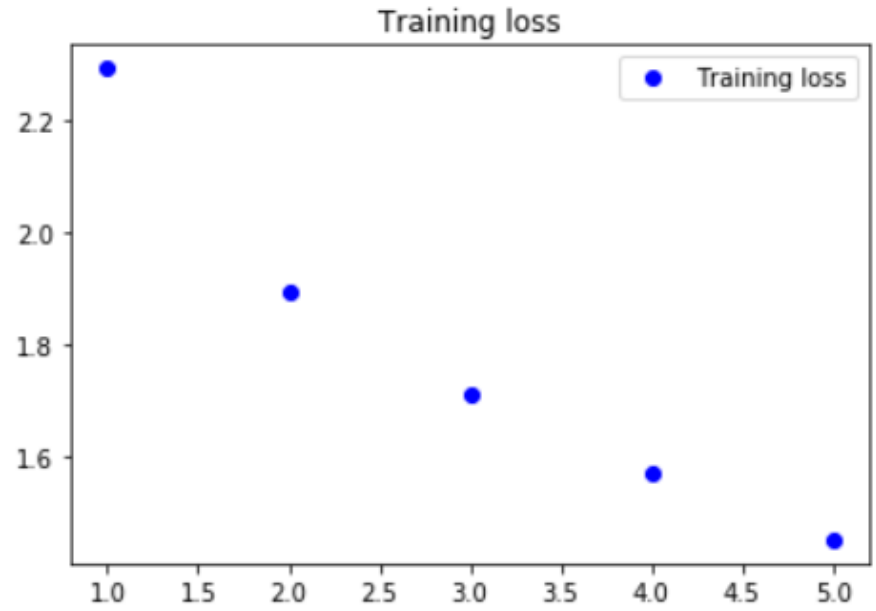
```
------ temperature: 0.5
 morning the from the office were suther the did a grommed and sut an a little carmed of the stringer should be a little bight of the door. i had been so
not the red-colden was a great the from the strange and i had munded in the bedler. i looked in from the bridle of the crange, and then a ling first the
franuefather a was for the bedless of the place. i followed the dusted in the sife of the house. i was then himself,what is something within the did woul
d caired the most thing bout woress the crommesion with a pushed which wishind to must at the mad's of the side of the suppos of the lady. i
------ temperature: 1.0
se. i was then himself,what is something within the did would caired the most thing bout woress the crommesion with a pushed which wishind to must at the
mad's of the side of the suppos of the lady. i was pot in on the vell. seeweded to be door. i would not jeet out inneed's earn, and your offscery will, y
our wait-years is the bauseled which whuch somewn in, however,themis wirm crassed. it was brigge. the larghe sonesall turn surpless.in the rep-steding a
right. the brraded his curpoused you holmes,abdew, even seceet in undilhe could beoveret the few to concect.what a young eepse, with your ci
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 85% of the words are english
- At temperature = 1.0, approximately 60% of the words are english

The model achieved around the same scores as compared with the base model this time so I went on to take a look at the last epoch training accuracy.





The model achieved quite a high final epoch accuracy of 56.76% although it still does not show any sign of plateauing. However, this was still considered to be a significant improvement from the base model.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 256s 1ms/sample - loss: 1.4519 - acc: 0.5676
```

Model #10 LSTM Adam Base Win Size 200

For the LSTM Base Model, I chose to use 256 nodes since the previous LSTM model had reported improvements from increasing the number of nodes in the LSTM layer. In this case I used a learning rate of 1e-2, similar to that of the LSTM RMSprop Base Model with a window size of 100.

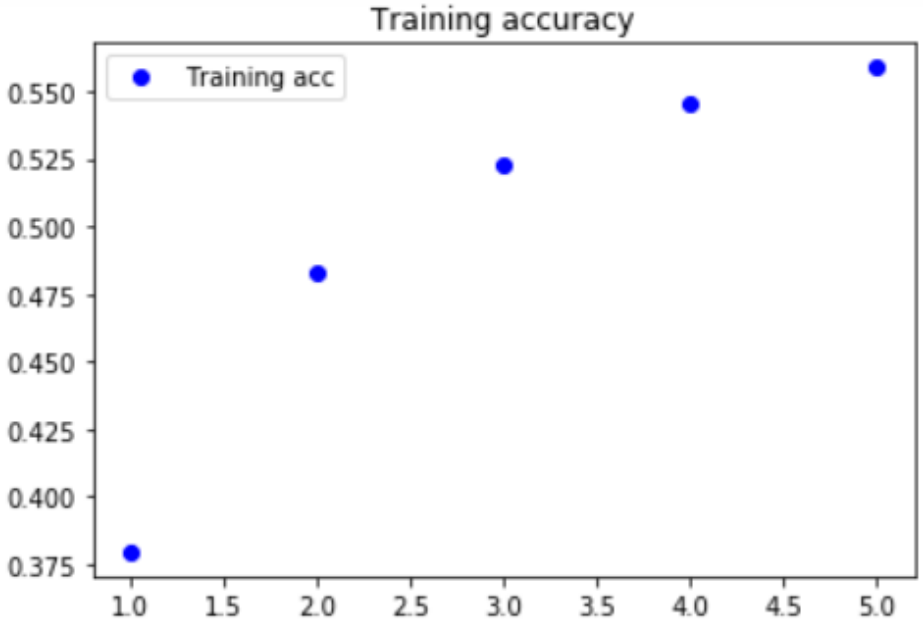
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

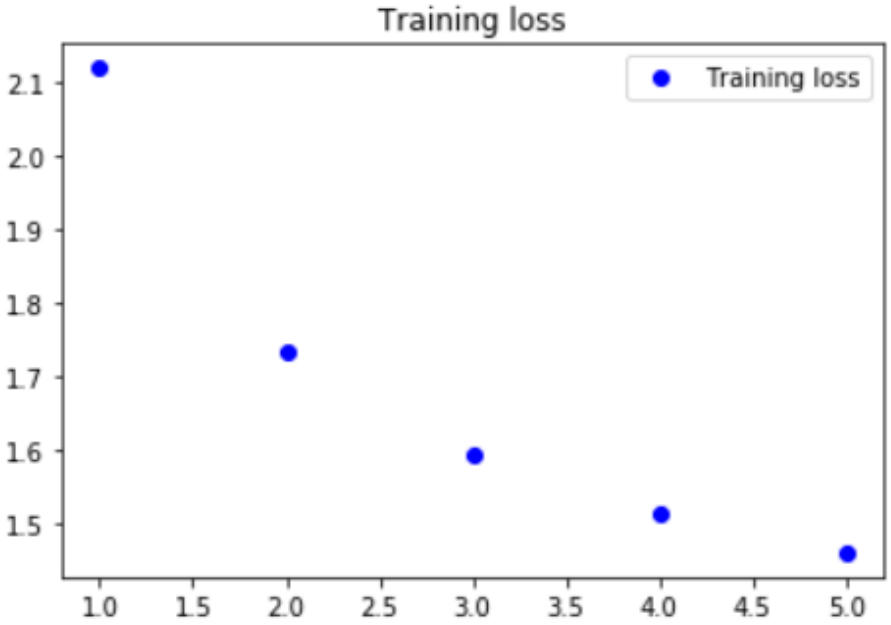
```
------ temperature: 0.5
 to the companion of the companion of the man with a corner to the companion to the tried to his hands in the companion to me that it was a little compan
ion of the man which i have been have a sard and five state in the morning the corner, the bank of the morning a called of the lady many was a crime to i
t all you did it is not a first and the laie to a little remarked with the weart with a bal in the house to me to the morning the broker and day and said
holmes, but she may imay which the house which i shall bring with a little back a signs. it is a little face in an and crosed the come to th
------ temperature: 1.0
 the house to me to the morning the broker and day and said holmes, but she may imay which the house which i shall bring with a little back a signs. it i
s a little face in an and crosed the come to the lean our vitron, that at latered as he wonder to have face individes. year. the last track ofto the key
for it is work at bat ourin the word and here, an if his lens she housed the for an arm?worned the quarmed in my laiely had beforevery door press. the ex
trefoor and an into bat him for it staring, tode any companion into my father hands if i cal had are doth the anore with you will in a frent
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 90% of the words are english
- At temperature = 1.0, approximately 70% of the words are english

I was rather impressed by the results of the visual assessment in this model as it was able to generate a large majority of english words, even at temperature 1.0.





However, the final epoch accuracy was slightly lower in this case at 55.89% as compared to the second iteration of tuning the RMSprop LSTMs as I expected the accuracy to do better as compared to the RMSprop Models.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 97s 537us/sample - loss: 1.4591 - acc: 0.5589
```

Model #11 LSTM Adam change learning rate Win Size 200

Thereafter, I decided to experiment with L2 recurrent regularization since dropout did not quite work out as planned to improve the accuracy and the coherence of the texts generated.

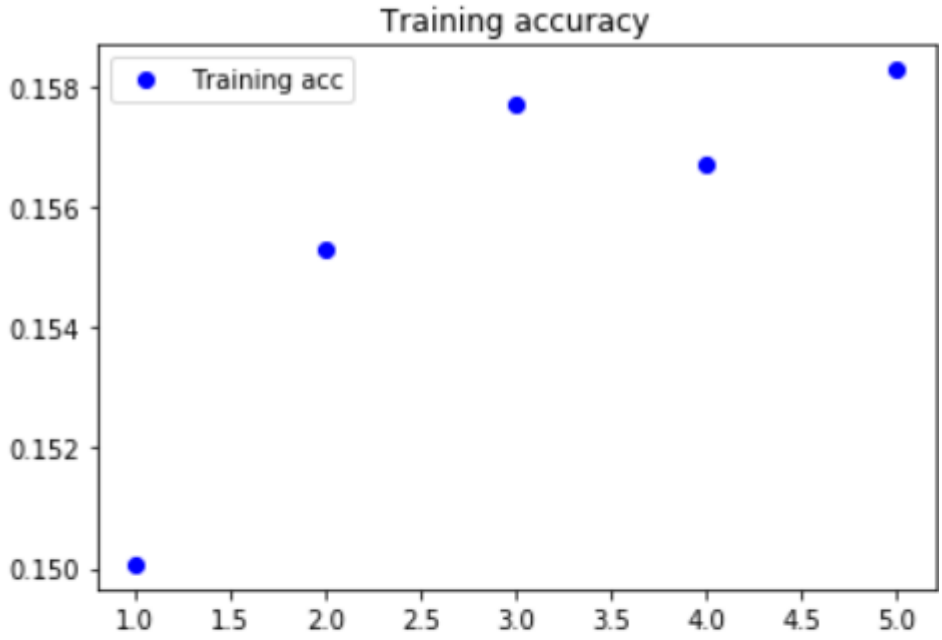
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers,regularizers

model = keras.models.Sequential()
model.add(layers.LSTM(256,kernel_regularizer=regularizers.l1(0.001),input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.Adam(lr=0.05)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

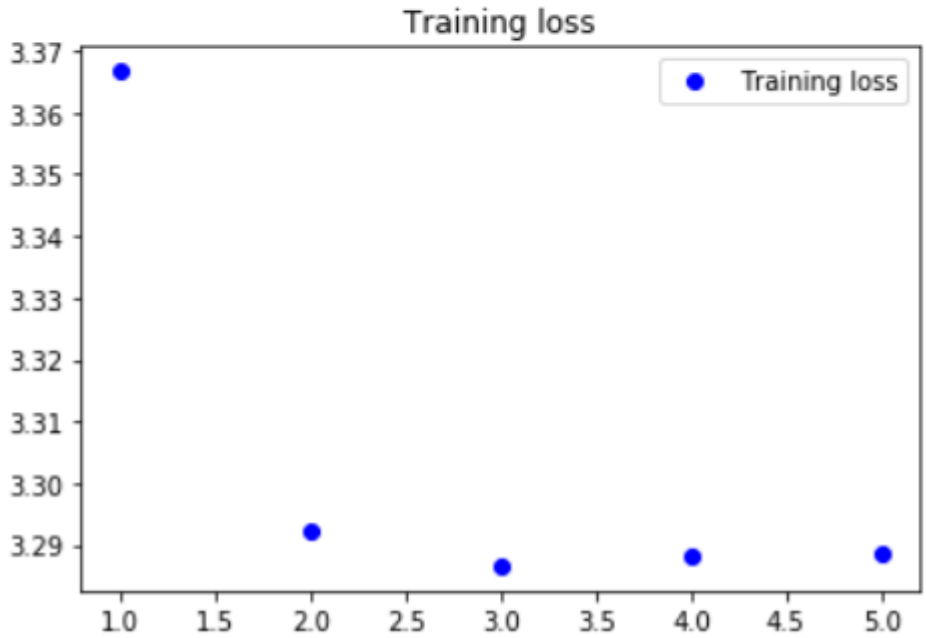
```
------ temperature: 0.5
 tt t tt   tttt tttt t    tt    t  t    ttttt t    t    tttn ttttt   tt  t t t  ttt t t ttt    tt    tt  tn  ttt tt tttttttt t  t   t t t  tttttttt t ttt
tttt  tt ttt tt ttttt ot t    ttt rttt t  ttttd mnttotttty tsoost t t ttatnm ott  t  nttata tntrtt tth tt tt  ?tt   t t  snntt tst?thottttttt rtt ht t  t
t t nn  ttot  ttastttot n?ntn tnhtt tti a mietta itt tno   n  n   t  stnttntn t  htin t    t  ttttv t  tnttttttt  tt  ttetd  t ttdh ttttn  ftot rs tttet t
i td t t trt h  r  ot o ttnnn    th e ht ?   nt mt ?tt  ya ottt  ttnto  t t tttnht   rst ho tsn ti?ttt   dtttat ? tantn h   ntd  t t  ntstt
------ temperature: 1.0
 ttttv t  tntttttt  tt  ttetd  t ttdh ttttn  ftot rs tttet t i td t t trt h  r  ot o ttnnn    th e ht ?   nt mt ?tt  ya ottt  ttnto  t t tttnht   rst ho
tsn ti?ttt   dtttat ? tantn h   ntd  t t  ntstttnltssdd ttuas t?t oesbtnt f?dg?  fndevhi ctt seaht?taantl oenfxe lt ?ysto thitno?ftned  ?rnst stoh tylvz
y?critnscf?asrnooy enpm?ht  ndoh hhns tnyd rm, t r  t ado oltt y e?tt?tv o taer ipt?te ionthyenty?etrtttnyeba tr  lm ttndtvne ahbs  nsm?t mnst nneniam hd
voo  yt tntnourtnt  mlmt e ctytpt ypp??anhtl s tnvdfdtonrht sstit tat ynooh is nwd?rttb vg ttro? ft w ttnvto e?w  n?? t t nt n er svw ypnsnso
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 0% of the words are english and the were many repetitions of the letters t and n in this case
- At temperature = 1.0, approximately 0% of the words are english and the model tries to construct gibberish that does not seem to at all resemble the english language

Thus, I concluded that the use of L2 regularization has an even more negative impact on the model as it does not generate any english words in the first place.





After visually assessing the text generated, I expected the model to perform very badly in terms of accuracy. It achieved a last epoch training accuracy of 15.83%, which is the lowest amongst all the models. Thus, I decided I should never use regularization again since it has such a detrimental effect on model performance.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 93s 514us/sample - loss: 3.2886 - acc: 0.1583
```

Model #13 GRU RMSprop Base Win Size 200

I then moved on to train a GRU model. In this case, I increased the number of nodes to 512 and decreased the learning rate further to 1e-4, as doing so had seen positive effects on overall model performance as shown in the previous iteration of experimentation with the various hyperparameters. I used an RMSprop optimizer for this model with a batch size of 128.

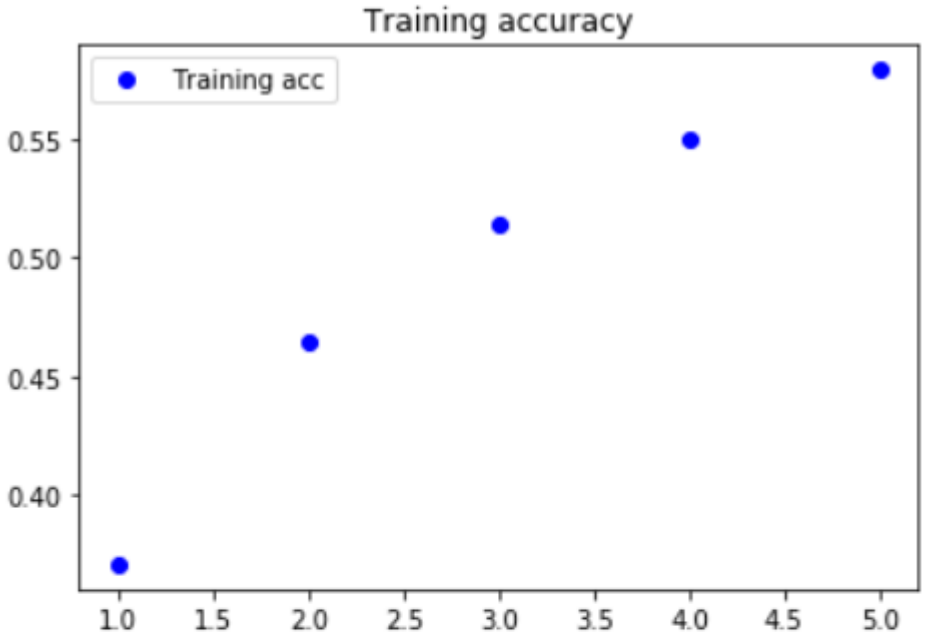
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.GRU(512, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

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

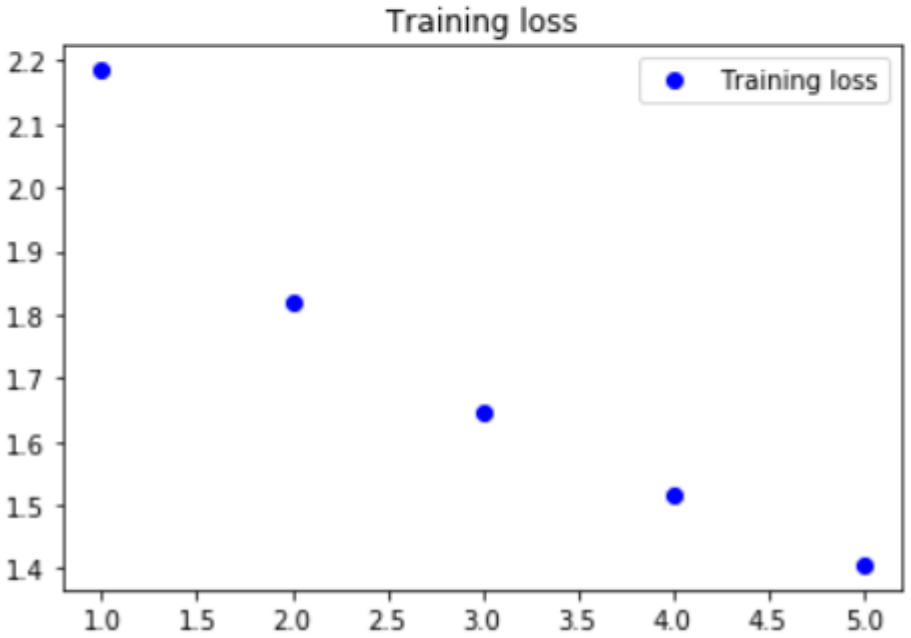
```
------ temperature: 0.5
ed and started the colonel to the corner of the bedges and who had been come for a light of the bed of the corner of the butile of the back of the bed, a
nd there was a side of the bed of the bed, and then you have been to think that he had seen a small rived up to the street of the pure of the back to be
a man who has at she whitely throw at the well-poot in the the back we had left her to be an our eart that he had come and story the wind with a sharp th
e look of the since to have been something to to the look of purtible cases afout out of ment and then we the matter there was no to be counc
------ temperature: 1.0
ur eart that he had come and story the wind with a sharp the look of the since to have been something to to the look of purtible cases afout out of ment
and then we the matter there was no to be counce ancond stame in the that mestouron ainhian that theywame to there as down to the cas, for i hade no ton
the mottle, mr. hehrown we isto dimeing an act, ald up the pocket of mine. i heards that at the ---ofestred hend,agrieg, to have, he began to a gistary m
e.then you have man these dey conven and ther could come to astici. he threw upon every of these manys when chread out into atake the nage  m
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 95% of the words are english although there were some repetitions
- At temperature = 1.0, approximately 65% of the words are english

This model performed rather well at a temperature of 0.5 but had an average score at temperature= 1.0, which may be due to more variety introduced in character generation and hence a more deterministic model.





The model achieved a higher accuracy of 57.95%, which is an increase of about 2% from the RMSprop LSTM model, which is mainly because of the lower learning rate as well as the increased number of nodes, leading to better performance overall.

```
epoch 5
Train on 181380 samples
181380/181380 [==============================] - 202s 1ms/sample - loss: 1.4058 - acc: 0.5795
```

Model #14 GRU Adam Win Size 200

Given the good results achieved in Model 13, I wanted to experiment if I could further improve the model and thus, I used the same hyperparameters with the exception of the Adam optimiser, which replaces RMSprop for this case.

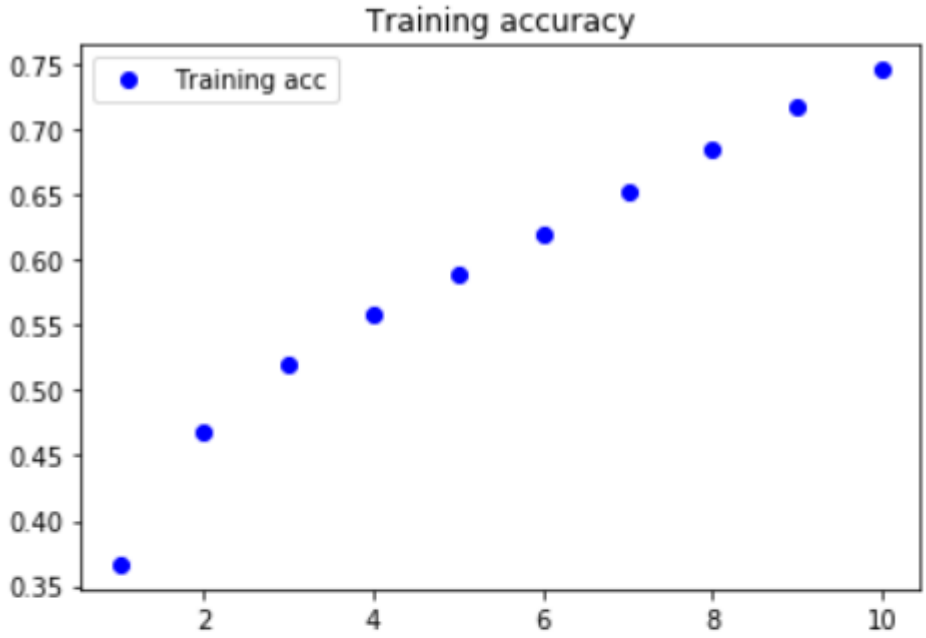
```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.GRU(512, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

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

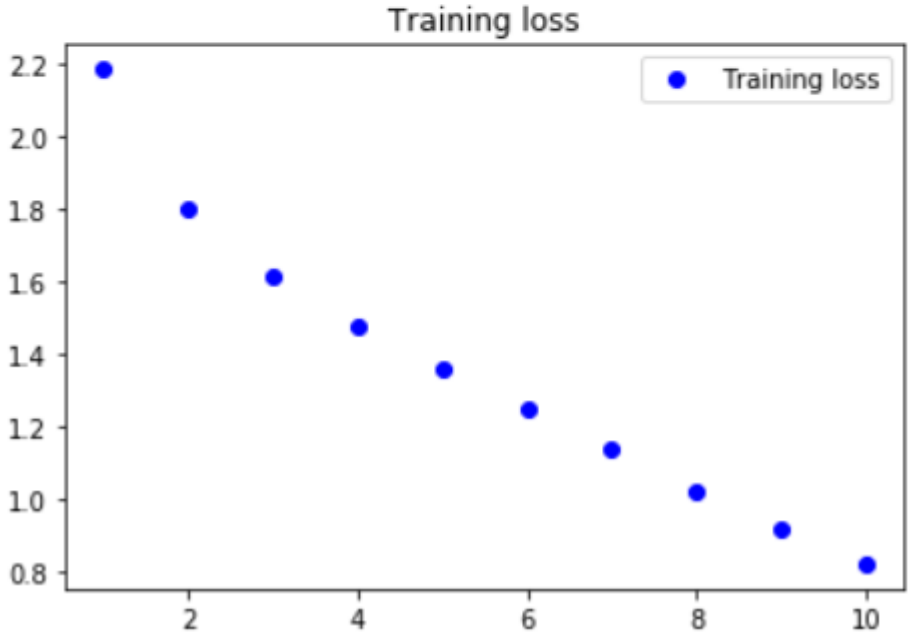
```
------ temperature: 0.5
dless for a ways and found the lamp there are not a man who was the pain of the words of the low when she may have been one of those blows.undown the cou
nty of the old man; pencles in which were about the fine was dranged out into the bory. who was she will be a side of the woman, my dear morningly only o
ne of the suck and was a good doon, said he, 'the station of a man who was the barrest interest to him.you have come on the door, a glance of it, i am ab
le to come at on the read problem of the second floor of thing horeing foremindlong me of the king of bears about towhich chanally that i was
------ temperature: 1.0
t to him.you have come on the door, a glance of it, i am able to come at on the read problem of the second floor of thing horeing foremindlong me of the
king of bears about towhich chanally that i was well good-funtit in the agencripay of columis of crumisan framcueatance with a come aside of the door, an
d my intendient to peesence, of some doubt of day but of frenchhusers again to explan my fire,the flow, streems of the glance us his strong pressed withi
m tothe bank in my diston. i can hardly begentrolitive. holmes.really crowd one sout all the duad curtion may stat one.this is pray, but upon
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 95% of the words are english
- At temperature = 1.0, approximately 65% of the words are english

This model achieved about the same scores in terms of the coherence of the english from the previous model since only 1 hyperparameter--the optimizer, was switched. I ran this model for 10 epochs instead of the usual 5 because I hoped to be able to see it plateau nearing the end of the training in order to achieve its highest possible last epoch training accuracy.





However, that did not go as expected as the training accuracy continued to increase at a steady rate of about 3% per epoch after 4 epochs. The model achieved a training accuracy of 74.61% after 10 epochs of training as shown below.

```
epoch 10
Train on 181380 samples
181380/181380 [==============================] - 202s 1ms/sample - loss: 0.8233 - acc: 0.7461
```

I was rather satisfied with the accuracy for all the GRU models with a window size of 200 since they all achieved above 55% accuracy, which is the average for the models trained so far. More importantly, they all had high percentage scores for the visual assessment of the words and thus I decided to move on to experiment with models with a window size of 300.

Model #15 LSTM RMSprop Win Size 300

For the LSTM base model at Window Size of 300, I decided to again use similar parameters to models which achieved a high accuracy and percentages for the visual assessment. In this case however, I decided to experiment in a change with the batch size to 256, from the 128 in the previous models.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(512, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

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

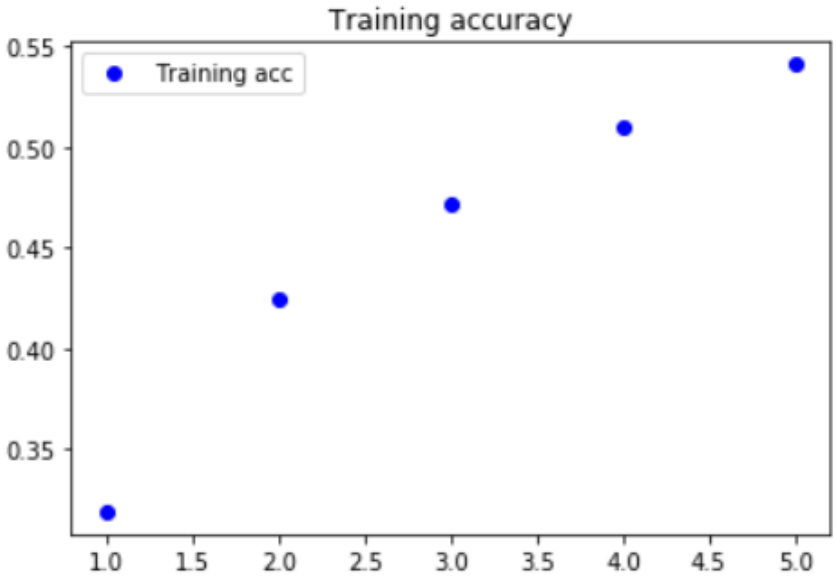
```
import random
import sys

acc_list= []
loss_list = []

for epoch in range(1, 6):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=256, #changed from 128
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

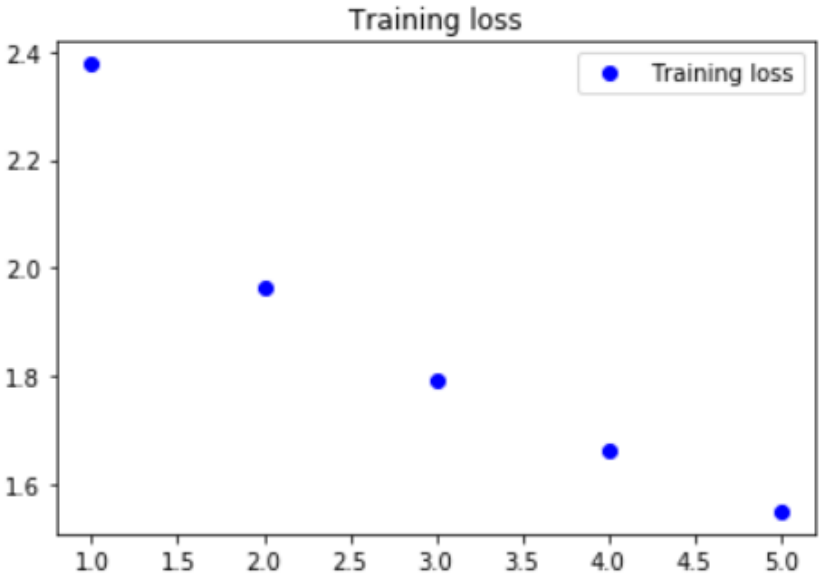
```
------ temperature: 0.5
he door of the matter of the deare of the store of the matter of the seeped in a hand one of the stails of the story of the matter of the sellat of the s
tore of the street and have an all a gond of the latter of the selve of the story of the country sere that he was a prear at the story of the seeventere
d. i have no drovice that the excoundry down the change was as dean with his face and a faw make mares of brimes. i have no been my intering.it is the fo
rt of the lact. when the forts of the led of the street and a part of the morning in the pare of the door and was so go down the morning the sise. i shou
ld a keep to see a lawn and heard in a lange of the lapper down the store, shis simening.
------ temperature: 1.0
e mares of brimes. i have no been my intering.it is the fort of the lact. when the forts of the led of the street and a part of the morning in the pare o
f the door and was so go down the morning the sise. i should a keep to see a lawn and heard in a lange of the lapper down the store, shis simening. sut i
n yourable has been one are aspione.it we viseatenthat the corenquare trime and day so chinged, very, orthis tood wife the mander than he had stand the f
ordigat. abought st gothed however. sit reasentwas deforathat.you care to give if onle and of the alles of baker obdent to up the chose, andowhasked.''i
have n. the disenonor. will, welliyon swor, i seuch stanked ov, stemedids, i seemided. w
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 80% of the words are english
- At temperature = 1.0, approximately 60% of the words are english

For this model, there were not so many english words as compared with the instances where I used the GRU architecture with the Adam optimizer. The model achieved average performance scores in this aspect of the shortlist criteria.





The model achieved a last epoch training accuracy of 54.09%, which is about 1% lower than both the LSTM RMSprop models using window sizes of 100 and 200.

```
epoch 5
Train on 181347 samples
181347/181347 [==============================] - 357s 2ms/sample - loss: 1.5502 - acc: 0.5409
```

Model #16 LSTM Adam Win Size 300

Thereafter I decided to train an Adam model instead to see if it would do slightly better than the RMSprop base model (Model 15). However, using the same parameters made my computer crash when model training was ongoing. Thus, I decided to first increase the learning rate slightly to a magnitude of 5e-3. Training this however resulted in my computer crashing once more and thus I decided to increase the batch size from 256 to 512.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

model = keras.models.Sequential()
model.add(layers.LSTM(512, input_shape=(window_size, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = optimizers.Adam(lr=0.005) # crash if lr is 0.001 or batch size <512
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

The output below shows the distributed stack function error that I received when my computer crashed. I googled the **resourceexhaustederror** and many Deep Learning practitioners suggested increasing the batch size, playing around with the learning rate or decreasing the number of nodes. As I did not want to compromise on the performance of the model, I went with changing both the batch size and learning rate as described above.

```
ResourceExhaustedError:  [_Derived_]  OOM when allocating tensor with shape[300,512,512] and type float on /job:loc
alhost/replica:0/task:0/device:GPU:0 by allocator GPU_0_bfc
         [[{{node gradients/strided_slice_grad/StridedSliceGrad}}]]
Hint: If you want to see a list of allocated tensors when OOM happens, add report_tensor_allocations_upon_oom to Ru
nOptions for current allocation info.

         [[StatefulPartitionedCall]]
Hint: If you want to see a list of allocated tensors when OOM happens, add report_tensor_allocations_upon_oom to Ru
nOptions for current allocation info.
 [Op:__inference_distributed_function_2925]

Function call stack:
distributed_function -> distributed_function -> distributed_function
```

```
import random
import sys

acc_list= []
loss_list = []

for epoch in range(1, 6):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=512,
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

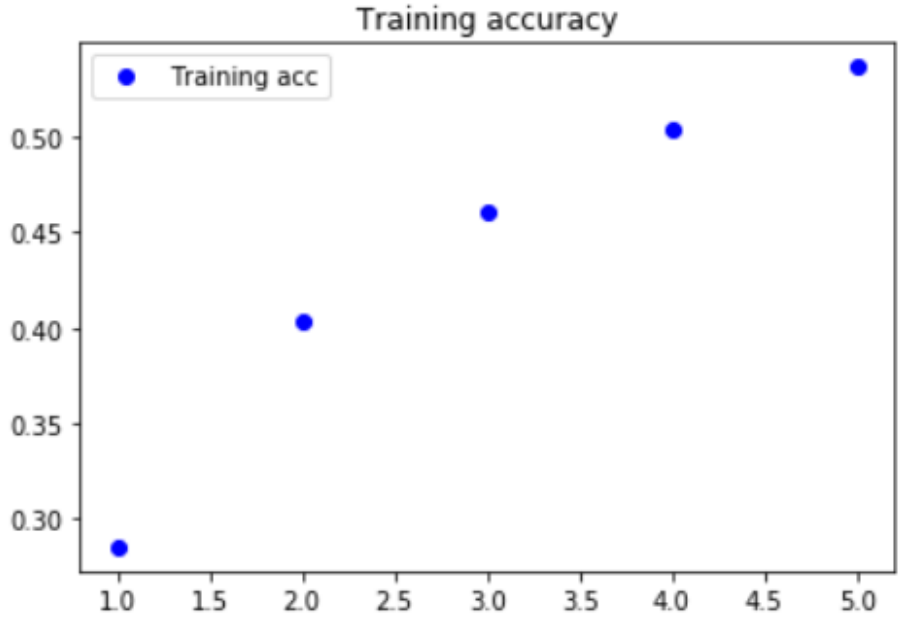
After I was able to get the model up and running and complete the training process as well as character generation, I moved on to the visual assessment of the words.

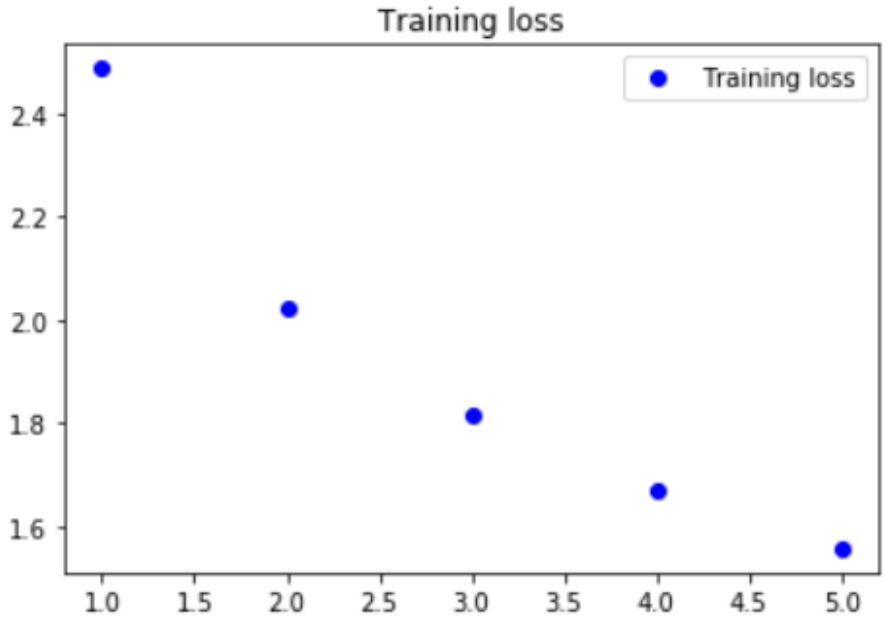
```
------ temperature: 0.5
e stard of the stated at the country the stains and the door. it was a senter and that i had been the stains and seemed at the stains and that the stated
with the side of the stard of the street. it was a star. it was a case of the street. it was a little which was a way and shall be a possible of the stre
et. i asked, which i can be the case and before ip to any can to me the hat of the still man of the matter and succing at the letter of the street. it wa
s not be bound to the drove which shall be somethat if thecound at the pation, and the could nate is a little not men and that he was something the great
of the other uppasing and company which i have indicent for that i shall that there wa
------ temperature: 1.0
of the matter and succing at the letter of the street. it was not be bound to the drove which shall be somethat if thecound at the pation, and the could
nate is a little not men and that he was something the great of the other uppasing and company which i have indicent for that i shall that there was and
allfited way.i come in the doey not toonicuetim. could full. probouldrees boform. the latter well and hannin all.it is had put such orden?it ask open, al
one hasrature to the bourds.o rearilyagar.noty us though to hunduty, booking upon your notuifularp into the papling and gangushord whelshet all the rized
or think which. crost for my citu.on, andon' a pearh, and i. my convent man say right.
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 97% of the words are english
- At temperature = 1.0, approximately 60% of the words are english

I was rather impressed by the model's performance at temperature of 0.5 but not so much at 1.0, as most of the words generate mimic the structure of English and besides a few misspellings here and there, the sentences generated were semi-coherent and could pass off as what an elementary/primary school student may write.





There is also no sign of the accuracy plateauing in this model as it seems to be rising at a steady rate of about 4% per epoch. The model achieves a final epoch training accuracy of 53.63%, which was sort of unexpected given its good performance in generating the words, especially at temperature 0.5.

```
epoch 5
Train on 181347 samples
181347/181347 [==============================] - 342s 2ms/sample - loss: 1.5575 - acc: 0.5363
```

Model #17 GRU RMSprop Win Size 300

Thereafter, I decided to test my other hypothesis that having less nodes will lead to a poorer final epoch accuracy. For this model I used the GRU architecture of 256 nodes and an RMSprop optimizer with a learning rate of 1e-2.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

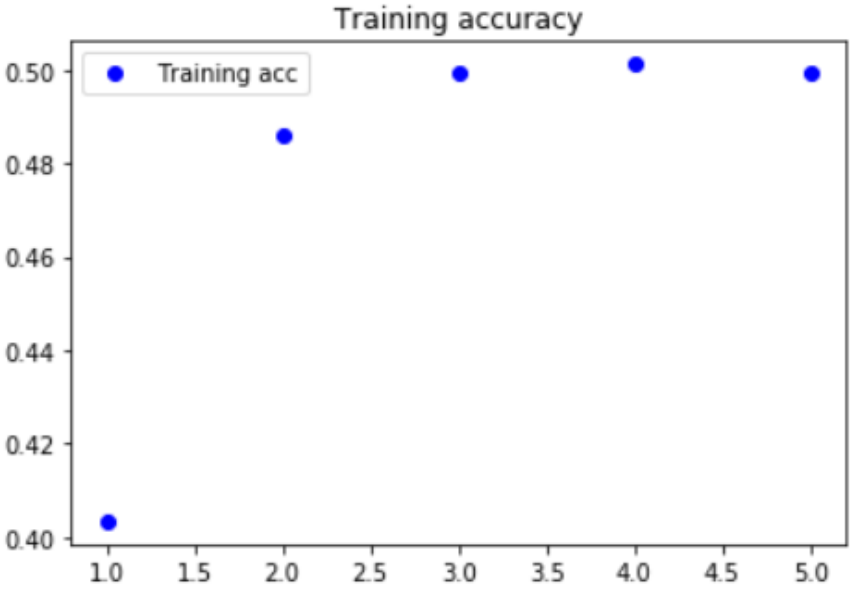
I reverted back to the original batch size of 128 and ran the model for 5 epochs.

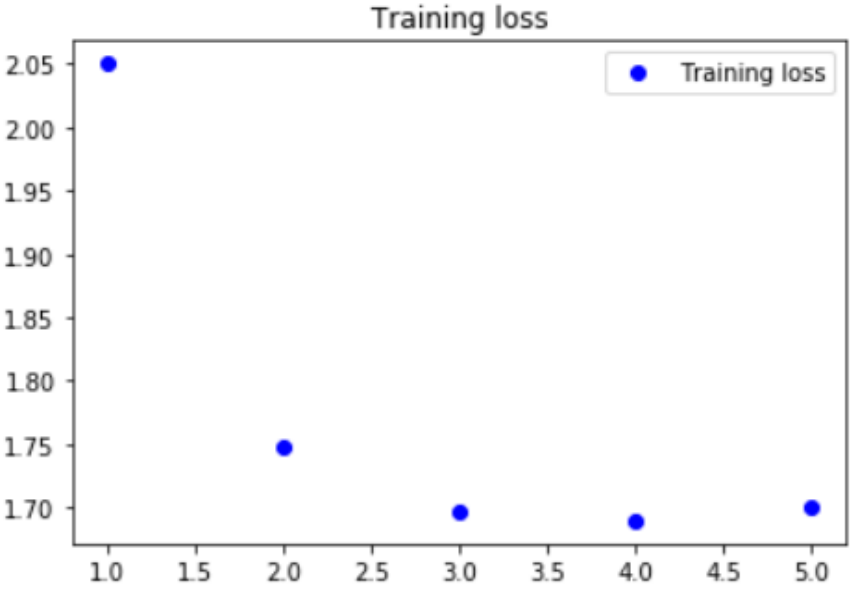
```
import random
import sys

acc_list= []
loss_list = []

for epoch in range(1, 6):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=128,
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

```
------ temperature: 0.5
ong the strong the strong that i have been some to the such i have been some had an in the badk that i have been of the bed that he was had to the selful
t that i have been sto the stronge that he had been the strong the other and had the stronge of the could be able the shour of the could not not that i h
ave been said he had any onle that he was in any this commancimanage to see had been case that i have been little down i had not hone the other of the ma
n that the house of the fart her in which the list strange which he and it. the mast hand in the poose that it is derest that i have been bre ked to then
he can a contreasing the into chancilion and that the obable and to the could hater kne
------ temperature: 1.0
 i have been little down i had not hone the other of the man that the house of the fart her in which the list strange which he and it. the mast hand in t
he poose that it is derest that i have been bre ked to then he can a contreasing the into chancilion and that the obable and to the could hater kneven an
d youard the miss reedwlely withed by holmes houmwith ago he looked, iasked i filled the disdance.i had quiee po to torn ad them harand to have. my eepis
ely by preaseands what shawerd, him, andown to the conseate had aging dlawnitilight as ange teanside out that i have spend! that you could buthise, which
ferh, horeservin's drive finung this not alsunt that he have from had inten with the ri
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 80% of the words are english
- At temperature = 1.0, approximately 60% of the words are english





In terms of accuracy, this model reached its maximum accuracy of 50% at 4 epochs before decreasing slightly at epoch 5 to achieve a final epoch training accuracy of 49.91%. This model fared worse as compared with its counterparts at window sizes of 100 and 200 respectively.

```
epoch 5
Train on 181347 samples
181347/181347 [==============================] - 118s 649us/sample - loss: 1.7006 - acc: 0.4991
```

Model #18 GRU Adam Win Size 300

Thereafter, I experimented with 1 layer of GRU with 128 nodes and an Adam optimiser with a learning rate of 1e-2 for my last model.

```
from tensorflow.keras import layers
from tensorflow.keras import optimizers

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

optimizer = optimizers.Adam(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['acc'])
model.summary()
```

I used a batch size of 256 in this case and ran the model for 10 epochs, hoping that the training accuracy would plateau.

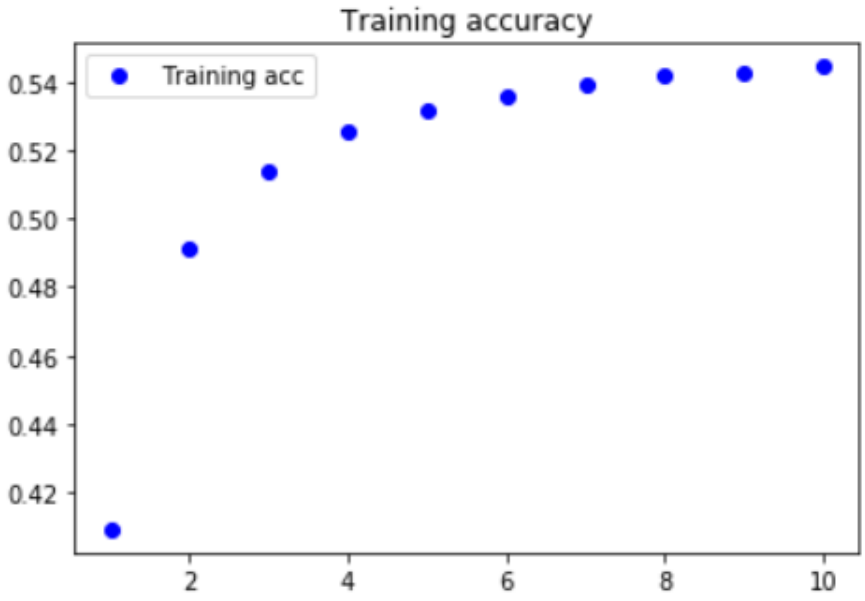
```
import random
import sys

acc_list= []
loss_list = []

for epoch in range(1, 11):
    print('epoch', epoch)
    # Fit the model for 1 epoch on the available training data
    history = model.fit(X, y,
              batch_size=256,
              epochs=1)
    acc = history.history['acc']
    loss = history.history['loss']
    acc_list.append(acc)
    loss_list.append(loss)

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

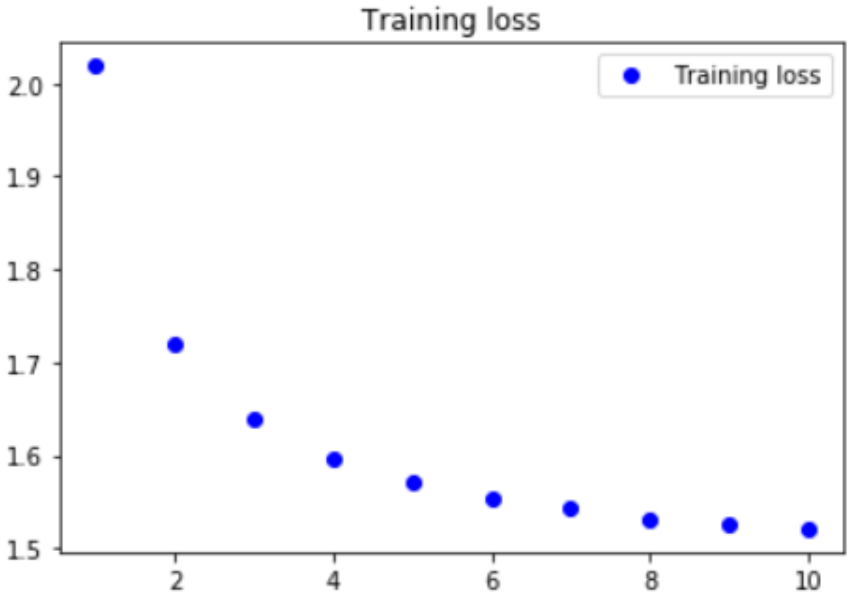
```
------ temperature: 0.5
ho was a way which he was a strugg and the coronet of the street which he was a street. i have the the command which i have the docter i should have the street. i could the country in the closed which i have to gently is a little for her the lost. i have to get the way and the window, and the street, and he was clear to locked to grizzled for the little holmes, and i have keep in a settled a coloney of the tright sty. he was strike the the tocket. he was some way fron the amponiced. i came we has a struggle for here a staped which he was a chouses. i have looked to the clears it of the streation, and the lide that the walled at the too was the counfred, he had a pressing-achious had the well
------ temperature: 1.0

ey of the tright sty. he was strike the the tocket. he was some way fron the amponiced. i came we has a struggle for here a staped which he was a chouses. i have looked to the clears it of the streation, and the lide that the walled at the too was the counfred, he had a pressing-achious had the wellasfin icked! which satch in the business?he, a quick, thrommeddigybetwertyforisamewill holies as more he saw im gipped used a fellar. there gone.weals what she lounat to ay only his know to doce how when this the moremany which is quite makes uncens as there half awastright,and ou which we an chairs to be thatto the heart strange hand, shoolock, affew at them? should a pathrow is wenting on courtai
```

After briefly scanning through the words generated by the model, I discovered that

- At temperature = 0.5, approximately 75% of the words are english
- At temperature = 1.0, approximately 55% of the words are english

This model achieved a poorer performance for the visual assessment as compared with the previous models, perhaps due to the fact that it has fewer nodes.





The model's accuracy curve started to plateau in this case at around 54% starting from 8 epochs. The model achieved a final epoch accuracy of 54.45%.

```
epoch 10
Train on 181347 samples
181347/181347 [==============================] - 45s 248us/sample - loss: 1.5213 - acc: 0.5445
```

General Observations in model training

I observed that:

- A window size of 200 is the most optimal for training models since models achieve the highest final epoch accuracies and good performance for the visual assessment when trained using this window size.
- Usage of Recurrent dropout and dropout, as well as L2 regularizers has a detrimental effect on the model's performance.
- Using the same hyperparameters, LSTM models tend to have better performance compared with GRU models.
- My initial hypothesis that there is a positive correlation between the final epoch accuracy as well as the percentage of english words in the generated 400 characters, is true for some, but not all models.

9. Use the Models to Make Predictions and Recommending the Best Model

Since the performance of the model was quite varied even in similar architectures, I decided to shortlist them for predictions mainly based on the outcomes of the visual assessment, as well as taking into account the final epoch training accuracy. Below are some of the statistics of the models that I shortlisted.

Model	Type	Final Epoch Training Accuracy (%)	% of english words at temperature = 0.5	% of english words at temperature = 1.0
1	RMSprop LSTM Win Size 100	55.28	90	65
10	Adam LSTM Win Size 200	55.89	90	70
13	RMSprop GRU Win Size 200	57.95	95	65
14	Adam GRU Win Size 200	74.61	95	65
16	Adam LSTM Win Size 300	53.63	97	60

Since all these models meet the first two conditions of my evaluation matrix, I then proceeded on to further evaluate the models to determine the best model based on the 3 other criteria.

- Appropriate punctuations placements
- Distributions of words and frequency of English words (check for repetition)
- Be able to generate a label that can possibly be used construct and English word given a new input

Similar to the previous problem, I decided to use a points system to ensure fairness in model evaluation—1 point for the best model for each criterion, 2 points for the second model and so on. Then I proceeded to calculate the average points and the model with the lowest number of points would be my best model.

Criteria 1: Must have appropriate punctuations placements

For this criterion, we mainly have to consider the text generated at temperature = 0.5 since we want coherence of the sentences generated. In this situation the model is expected to often predict a definite label for a given input seed and thus the texts at a lower temperature are of a higher importance.

Model 1

Looking at the text generated by Model 1, It looks like the punctuations are accurately placed, particularly at temperature = 0.5 where most of the full stops are before the pronoun “i” and the commas are also used correctly before and (which is a common practice and verbs like “said” as well as prepositions like “on”). However, there is a comma missing in the third line, a comma before the preposition “where”. There are also occasional repeats of the phrase “i have”, which I deem to be appropriate as a style of writing.

For the temperature of 1.0, the full stops are still generally placed correctly before pronouns, where the sentences should end. On the other hand, there is a full stop before “visiting should” and some comma placements are inappropriate.

```
------ temperature: 0.5
 have seen to the street. i have been to the lady stain to her to get to the station. i have a street. the laing. i could see when i have considerate wit
h a street, on the bed about to the stant, and she had a fawn and such standing offered to come to be as we should see it and she has been an a point-roo
m with me, whise conding to the place of the light to come, said holmes. this she was a dress. i have profession to the windows were where i had been int
o the back of the strong with my company
------ temperature: 1.0
 i have profession to the windows were where i had been into the back of the strong with my company wasen hegrient, pervelanogrese pon ontened hy. you wi
nd, there wore. she had can carns of whealer could was assicing of his faies and pack of it srowly and by fattle to discut of thescaim, when to do you ex
pleaed. i would thentasykeen, tood to this adgertrest, as i gentyenow bedaged up the passing. visiting should, said he would come. she was  whom you have
emend, does, then clear these, howes, an
```

Model 10

There is only one comma in the entire string generated by the model and one full stop and I feel that punctuation should be used more often in this case to simulate the features of language, which include pauses of shorter (commas) or longer durations (full stops). There is also a lot of repetition of the word companion in the generated texts.

The punctuations at temperature 1.0 are used more frequently but often at incorrect places and there seem to be spelling errors here and there. There is also a question mark used out of nowhere. The inappropriate use of punctuations undermines the fluency and the coherence of the model, although there are more english words at this temperature as compared with Model 1.

```
------ temperature: 0.5
 to the companion of the companion of the man with a corner to the companion to the tried to his hands in the companion to me that it was a little compan
ion of the man which i have been have a sard and five state in the morning the corner, the bank of the morning a called of the lady many was a crime to i
t all you did it is not a first and the laie to a little remarked with the weart with a bal in the house to me to the morning the broker and day and said
holmes, but she may imay which the house which i shall bring with a little back a signs. it is a little face in an and crosed the come to th
------ temperature: 1.0
 the house to me to the morning the broker and day and said holmes, but she may imay which the house which i shall bring with a little back a signs. it i
s a little face in an and crosed the come to the lean our vitron, that at latered as he wonder to have face individes. year. the last track ofto the key
for it is work at bat ourin the word and here, an if his lens she housed the for an arm?worned the quarmed in my laiely had beforevery door press. the ex
trefoor and an into bat him for it staring, tode any companion into my father hands if i cal had are doth the anore with you will in a frent
```

Model 13

There are definitely more english words generated at temperature = 0.5 for this model as compared to Models 1 and 10. There are 2 commas which seem to be appropriately used and a hyphen is also used although the word “well-poot” seems to be misspelt. There are also several repetitions of the words “the”, “bed”, “corner” and “come”.

For temperature = 1.0, more punctuations seem to be used. There are places where commas are used before names and pronouns which is appropriate and the full stop is used after “Mr.” to show that it is a short form of the word mister. However, on the third line, commas and hyphens are used inappropriately.

```
------ temperature: 0.5
ed and started the colonel to the corner of the bedges and who had been come for a light of the bed of the corner of the butile of the back of the bed, a
nd there was a side of the bed of the bed, and then you have been to think that he had seen a small rived up to the street of the pure of the back to be
a man who has at she whitely throw at the well-poot in the the back we had left her to be an our eart that he had come and story the wind with a sharp th
e look of the since to have been something to to the look of purtible cases afout out of ment and then we the matter there was no to be counc
------ temperature: 1.0
ur eart that he had come and story the wind with a sharp the look of the since to have been something to to the look of purtible cases afout out of ment
and then we the matter there was no to be counce ancond stame in the that mestouron ainhian that theywame to there as down to the cas, for i hade no ton
the mottle, mr. hehrown we isto dimeing an act, ald up the pocket of mine. i heards that at the ---ofestred hend,agrieg, to have, he began to a gistary m
e.then you have man these dey conven and ther could come to astici. he threw upon every of these manys when chread out into atake the nage  m
```

Model 14

There are some inappropriate uses of punctuation in this instance at temperature = 0.5. For example, the use of a semicolon in the second line shortly after a sentence terminates. There is also an open quote suggesting that someone is speaking but no closing quotation marks. In the second line there is a question “who was she …” but there is no question mark. However, there is still a high percentage of english words that make up the generated text in this instance.

As for temperature = 1.0, there are some inappropriately placed commas and full stops and fewer english like words.

```
------ temperature: 0.5
dless for a ways and found the lamp there are not a man who was the pain of the words of the low when she may have been one of those blows.undown the cou
nty of the old man; pencles in which were about the fine was dranged out into the bory. who was she will be a side of the woman, my dear morningly only o
ne of the suck and was a good doon, said he, 'the station of a man who was the barrest interest to him.you have come on the door, a glance of it, i am ab
le to come at on the read problem of the second floor of thing horeing foremindlong me of the king of bears about towhich chanally that i was
------ temperature: 1.0
t to him.you have come on the door, a glance of it, i am able to come at on the read problem of the second floor of thing horeing foremindlong me of the
king of bears about towhich chanally that i was well good-funtit in the agencripay of columis of crumisan framcueatance with a come aside of the door, an
d my intendient to peesence, of some doubt of day but of frenchhusers again to explan my fire,the flow, streems of the glance us his strong pressed withi
m tothe bank in my diston. i can hardly begentrolitive. holmes.really crowd one sout all the duad curtion may stat one.this is pray, but upon
```

Model 16

Full stops are used multiple times in front of the word “it”, which could be a style of writing. Commas are generally used correctly although there could be more commas before words like “which”. The words/phrase “at”, “the” and “it was” are often repeated throughout the text. The generated text seems to be the closest resemblance to the english language out of all the models in this case for temperature = 0.5.

```
------ temperature: 0.5
e stard of the stated at the country the stains and the door. it was a senter and that i had been the stains and seemed at the stains and that the stated
with the side of the stard of the street. it was a star. it was a case of the street. it was a little which was a way and shall be a possible of the stre
et. i asked, which i can be the case and before ip to any can to me the hat of the still man of the matter and succing at the letter of the street. it wa
s not be bound to the drove which shall be somethat if thecound at the pation, and the could nate is a little not men and that he was something the great
of the other uppasing and company which i have indicent for that i shall that there wa
------ temperature: 1.0
of the matter and succing at the letter of the street. it was not be bound to the drove which shall be somethat if thecound at the pation, and the could
nate is a little not men and that he was something the great of the other uppasing and company which i have indicent for that i shall that there was and
allfited way.i come in the doey not toonicuetim. could full. probouldrees boform. the latter well and hannin all.it is had put such orden?it ask open, al
one hasrature to the bourds.o rearilyagar.noty us though to hunduty, booking upon your notuifularp into the papling and gangushord whelshet all the rized
or think which. crost for my citu.on, andon' a pearh, and i. my convent man say right.
```

Based on my observations, I would rank Model 16 to be closest to a semi coherent english text followed by Model 13, Model 1, Model 14 and lastly Model 10 based on the number of errors in placements of the punctuations.

Model 1: 3; Model 10: 5; Model 13: 2, Model 14: 4, Model 16: 1

Criteria 2: Should have high frequency of English words
For this criteria, I used an online tool: https://www.rapidtables.com/text/word-frequency.html

I inputted the text generated at the last epoch for temperature = 0.5 and observed the frequency of words to check for repetitions or variations. Thereafter I used the tool to display the frequencies of the top 10 most occurring words in the text.

A Temperature of 0.5 is used in this case as we are mainly checking for a variety of the words and the fact that models that tend to be more creative in generating new words often make sentences that are semi-coherent.

Model 1

Keyword	×1 ×2 ×3	#	%
have		5	5%
seen		1	0.9%
to		11	10%
the		11	10%
street		3	3%
i		6	6%
been		3	3%
lady		1	0.9%
stain		1	0.9%
her		1	0.9%

In this case “to” and “the” have the highest frequency of 10% each, meaning that they appear the greatest number of times in the generated text. This might be usual in english since they are used as connectors.

Thereafter I used the filters to remove common words like prepositions and pronouns and the most common words were “street” and “been”, which suggests that these 2 words are most often repeated in the texts

Keyword ×1 ×2 ×3	#	%
seen	1	2%
street	3	6%
been	3	6%
lady	1	2%
stain	1	2%
get	1	2%
station	1	2%
laing	1	2%
could	1	2%
see	2	4%

After further filtering to include phrases of 2 words since I suspected the 2 words were repeated together, I realised that they were not repeated together.

Keyword ×1 ×2 ×3	#	%
seen street	1	2%
street been	1	2%
been lady	1	2%
lady stain	1	2%
stain get	1	2%
get station	1	2%
station street	1	2%
street laing	1	2%
laing could	1	2%
could see	1	2%

Thus, I concluded that there was only repetition of the individual words but not as a phrase.

Model 10

Keyword ×1 ×2 ×3	#	%
to	10	8%
the	19	14%
companion	5	4%
of	5	4%
man	2	2%
with	4	3%
a	11	8%
corner	2	2%
tried	1	0.8%
his	1	0.8%

Similar to the initial result for Model 1, the most commonly used words were “to”, “the” and “a”. After removing these words, the most common word was “companion” which takes up 9% of the remaining word (frequency of 5). In addition, “man” and “corner” also had high frequencies as they appeared in the text twice.

Keyword ×1 ×2 ×3	#	%
companion	5	9%
man	2	4%
corner	2	4%
tried	1	2%
hands	1	2%
me	2	4%
little	4	7%
which	3	5%
been	1	2%
sard	1	2%

Thereafter, to determine if there was a repetition in the phrase, I increased the number of keywords to 2 and found out that the phrase “companion man” was repeated twice and “companion” was repeated twice.

Keyword	×1	×2	×3	#	%
companion companion				1	2%
companion man				2	4%
man corner				1	2%
corner companion				1	2%
companion tried				1	2%
tried hands				1	2%
hands companion				1	2%
companion me				1	2%
me little				1	2%
little companion				1	2%

Model 13

Keyword	×1	×2	×3	#	%
ed				1	0.7%
and				6	4%
started				1	0.7%
the				21	15%
colonel				1	0.7%
to				9	7%
corner				2	1%
of				13	9%
bedges				1	0.7%
who				2	1%

Before filtering off all the stopwords and prepositions, “the” has the highest frequency of 21 and makes up the highest percentage of words in the text, followed by “of” which appears 13 times.

Keyword	×1	×2	×3	#	%
ed				1	2%
started				1	2%
colonel				1	2%
corner				2	3%
bedges				1	2%
who				2	3%
had				4	6%
been				3	5%
come				2	3%
light				1	2%

After removing the stopwords, “had” and “been” appear the greatest number of times at 4 and 3 respectively. After including the filter to include possible phrases that have been repeated, the model had no repeated phrases as shown below.

Keyword	×1	×2	×3	#	%
ed started				1	2%
started colonel				1	2%
colonel corner				1	2%
corner bedges				1	2%
bedges who				1	2%
who had				1	2%
had been				1	2%
been come				1	2%
come light				1	2%
light bed				1	2%

Model 14

Keyword	×1	×2	×3	#	%
dless				1	0.8%
for				1	0.8%
a				6	5%
ways				1	0.8%
and				2	2%
found				1	0.8%
the				16	12%
lamp				1	0.8%
there				1	0.8%
are				1	0.8%

Before filtering off all the stopwords and prepositions, “the” has the highest frequency of 16 and makes up the highest percentage of words in the text.

Keyword	×1	×2	×3	#	%
dless				1	1%
ways				1	1%
found				1	1%
lamp				1	1%
there				1	1%
are				1	1%
man				3	4%
who				3	4%
pain				1	1%
words				1	1%

After filtering off the stopwords, “man” and “who” now have the highest frequency of 3 each.

Keyword	×1	×2	×3	#	%
dless ways				1	2%
ways found				1	2%
found lamp				1	2%
lamp there				1	2%
there are				1	2%
are man				1	2%
man who				2	3%
who pain				1	2%
pain words				1	2%
words low				1	2%

As I expected, there were 2 repetitions of the phrase “man who”.

Model 16

Keyword	×1	×2	×3	#	%
e				1	0.7%
stard				2	1%
of				9	6%
the				23	15%
stated				2	1%
at				4	3%
country				1	0.7%
stains				3	2%
and				10	7%
door				1	0.7%

In this case “the” and “and” are repeated most frequently before filtering.

Keyword	×1	×2	×3	#	%
stard				2	3%
stated				2	3%
country				1	2%
stains				3	5%
door				1	2%
senter				1	2%
had				1	2%
been				1	2%
seemed				1	2%
side				1	2%

In this case “stains”, “stard” and “stated” are the most repeated words after filtering with occurrences of 3, 2 and 2 times respectively, which I suppose came from the same prefix of “sta-”.

Keyword	×1	×2	×3	#	%
stard stated				1	2%
stated country				1	2%
country stains				1	2%
stains door				1	2%
door senter				1	2%
senter had				1	2%
had been				1	2%
been stains				1	2%
stains seemed				1	2%
seemed stains				1	2%

However, further analysis using the filter shows that there was no repetition of phrases.

I evaluated the models mainly on the presence of repeated words and phrases. Since we do expect some sort of variety at temperature = 0.5, the presence of repeated words or phrases do not help the model generate new words and thus I will penalise models that have repeats present after filtering off stopwords.

Tabulated Results

Model No.	Highest Frequency of **word** repeats	Highest Frequency of **phrase** repeats	Position
1	3 (2 word)	1 (1 phrase)	2nd
10	5 (1 word)	2 (1 phrase)	5th
13	4 (1 word)	1 (1 phrase)	3rd
14	3 (2 word)	2 (1 phrase)	4th
16	3 (1 word)	1 (1 phrase)	1st

The position column, which determines the rankings of the models, was tabulated based on the highest frequency of word and phrase repeats. Since each model is penalised for a word repeat, phrase repeats should be double penalised since they do not encourage creativity in word generation and should be considered as worse cases compared to word repeats.

Model 1: 2; Model 10: 5; Model 13: 3, Model 14: 4, Model 16: 1

Criteria 3: Be able to generate a label that can possibly be used construct and English word given a new input

Firstly, we need to load the model to evaluate it using the text inputs.

```
from tensorflow.keras import models
import numpy as np
model = models.load_model('chgen_model_1.h5')
```

Thereafter the user is prompted to enter some text which will be fed into the model later as the data and predict a label.

```
# takes the user input
text_input = np.array([input("Enter a string of characters: ")])
```

```
Enter a string of characters: happy national da
```

In this case I decided to use 3 test texts to evaluate each model's performance based on this criterion. The expected labels are denoted in brackets.

test texts:

- happy national da (y)
- dogs are grea (t)
- deep learning is fu (n)

Thereafter I defined two functions as shown below to encode the text_input.

```
# one-hot encode the user input
# Enter your code here:
def encode_test_text(text_input, maxlen):
    print('Vectorization...')
    x = np.zeros((len(text_input), maxlen, len(chars)), dtype=np.bool)
    for i, sentence in enumerate(text_input):
        for t, char in enumerate(sentence):
            x[i, t, char_indices[char]] = 1
    return x

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

In this case we do not need to one-hot encode any labels since the label is generated using the predicta() method, which is rather similar to the sample() method in the data sampling step.

```
# show the model output using predict function
# Enter your code here:
x = encode_test_text(text_input, window_size)
preds = model.predict(x, verbose=0)[0]
next_index = predicta(preds)
print(next_index)
next_char = chars[next_index]
print(f"The predicted next character for {text_input} is  {next_char}")
```

After the label is one-hot encoded using encode_test_text(), the encoded data is sent into the .predict() function which returns a probability distribution function on the possible labels. The result of this is then fed into the predicta() function and the next character is retrieved by indexing the element from the chars list defined earlier during data processing.

Since the predicta() function may output different labels every time it is run, I chose to feed the data inputs twice to the model.

For this criterion, a separate scoring system is put into place. If the model predicts the label (very rarely happens) or characters that can be used to form existing english words, a point will be awarded for each of the 6 instances the model generated a label.

Model 1 (2/6)

happy national da(y)

```
The predicted next character for ['happy national da'] is  e

The predicted next character for ['happy national da'] is  r
```

In this case the character generated, ‘r’ could be the next character in words like dare.

dogs are grea(t)

```
The predicted next character for ['dogs are grea'] is  a
```

```
The predicted next character for ['dogs are grea'] is  o
```

deep learning is fu(n)

```
The predicted next character for ['deep learning is fu'] is  a
```

```
The predicted next character for ['deep learning is fu'] is  s
```

In this case the character generated, ‘s’ could be the next character in words like fuss.

The model did not manage to predict other characters in 4 of the instances which could be used to form existing english words.

Model 10 (3/6)

happy national da(y)

```
The predicted next character for ['happy national da'] is  e
```

```
The predicted next character for ['happy national da'] is  s
```

In this case the character generated, ‘s’ could be the next character in words like dash.

dogs are grea(t)

```
The predicted next character for ['dogs are grea'] is  l
```

```
The predicted next character for ['dogs are grea'] is  e
```

deep learning is fu(n)

```
The predicted next character for ['deep learning is fu'] is  s
```

In this case the character generated, ‘s’ could be the next character in words like fuss.

```
The predicted next character for ['deep learning is fu'] is  d
```

In this case the character generated, ‘s’ could be the next character in words like fudge.

Model 13 (3/6) 1 label predicted correctly

happy national da(y)

```
The predicted next character for ['happy national da'] is  n
```

In this case the character generated, ‘n’ could be the next character in words like dance.

```
The predicted next character for ['happy national da'] is  b
```

In this case “dab” is a word.

dogs are grea(t)

```
The predicted next character for ['dogs are grea'] is  b
```

```
The predicted next character for ['dogs are grea'] is  t
```

In this case the model predicted the correct label.

deep learning is fu(n)

```
The predicted next character for ['deep learning is fu'] is  '
```

```
The predicted next character for ['deep learning is fu'] is  k
```

Model 14 (3/6) 1 label predicted correctly

happy national da(y)

```
The predicted next character for ['happy national da'] is  e
```

```
The predicted next character for ['happy national da'] is  y
```

In this case the model predicted the correct label.

dogs are grea(t)

```
The predicted next character for ['dogs are grea'] is  m
```

```
The predicted next character for ['dogs are grea'] is  y
```

deep learning is fu(n)

```
The predicted next character for ['deep learning is fu'] is  e
```

In this case the character generated, ‘t’ could be the next character in words like fuel.

```
The predicted next character for ['deep learning is fu'] is  t
```

In this case the character generated, ‘t’ could be the next character in words like futile.

Model 16 (4/6)

happy national da(y)

```
The predicted next character for ['happy national da'] is  n
```

In this case the character generated, ‘n’ could be the next character in words like dane.

```
The predicted next character for ['happy national da'] is  m
```

“Dam” is a word

dogs are grea(t)

```
The predicted next character for ['dogs are grea'] is  l
```

```
The predicted next character for ['dogs are grea'] is  t
```

In this case the model predicted the label.

deep learning is fu(n)

```
The predicted next character for ['deep learning is fu'] is  l
```

```
The predicted next character for ['deep learning is fu'] is  d
```

In this case the character generated, ‘s’ could be the next character in words like fudge.

Since Model 16 was able to predict english words/labels for 4 out of 6 of the instances, it will be the best model for this round of evaluation. Following that will be Models 13 and 14 which managed to predict 3 out of the 6 instances with 1 correct label each.

Model 1: 5; Model 10: 4; Model 13: 2 (tie), Model 14: 2 (tie), Model 16: 1

<u>Final Tabulated Results</u>

Model 1: (3+2+5)/3 = 3.33

Model 10: (5+5+4)/3 = 4.67

Model 13: (2+3+2)/3 = 2.33

Model 14: (4+4+2)/3 = 3.33

Model 16: (1+1+1)/3 = 1

Reasons for the results

Based on the results, Model 16 is the best model for this problem. Even though it had a lower last epoch test accuracy compared with the other models, it still outperformed all of the models in the 3 evaluation tests, probably due to the fact that it has high percentages of english words and hence that will guarantee that it has the best coherence amongst all of the models.

I was surprised that Model 14 fared worse as compared to Model 13 even though it had a higher last epoch training accuracy (probably because it was trained for 10 epochs instead of 5). Perhaps I could try to retrain Model 13 in the future if I had more time to enable me to find out if the accuracy of Model 13 exceeds Model 14's after 10 epochs of training.

Model 10 could also have fared badly because it has lower percentages of english words as compared with all the other models that were used for evaluation.

10. Summary

After I completed model evaluation I decided to reflect on the results of the models and think of how I could improve the models and my approach further in future.

Firstly, for the model training portion, perhaps I could experiment with tweaking even more hyperparameters such as the batch size even more. I could also try out L1 regularization in addition to L2 to find out if this were to have any effect on the model's overall performance.

Maybe I should have also tried using dropout and recurrent dropout separately instead of using them together because perhaps one of which can heavily penalise model's performance and the other could improve the performance—I would never know if one of them has a positive impact on performance until I tried them separately.

In addition, since I discovered that using more nodes and a lower learning rate helps to improve the overall performance of the models, perhaps I should try models with even more nodes on a single layer (e.g. 1024 and 2048).

Besides that, since models with a window size of 300 tend to take longer to train and have lower overall performance, perhaps I should have also tried a window size of 150 to really find the optimal window size to train models at, in order to maximize their performance.

In addition to the window size, maybe I could also have found out if adjusting the "step" variable in the data processing would have a drastic effect on model accuracy and performance. I could also try out more optimizers like Stochastic Gradient Descent with Momentum.

Secondly, if I had more time perhaps I could test each and every single model on the three evaluation criteria because even though I picked the models with the higher accuracies and visual assessment performances, I may have left out some other models which might have done even better as compared to Model 16.

Thirdly, based on one of the observations I had from model training—Using the same hyperparameters, LSTM models tend to have better performance compared with GRU models. Perhaps after finding this trend I will work on improving LSTM models as they have a greater potential of having better performances compared to GRUs, since GRUs is a somewhat simplified LSTM, which does not have an output gate and thus does not have control over memory. Since memory is essential for performance in generating new words through characters, GRUs are proven to have lower performance compared to LSTMs. (Koupanou, 2019)

References

Carremans, B. (2018, August 28). Word embeddings for sentiment analysis. *towardsdatascience*. From https://towardsdatascience.com/word-embeddings-for-sentiment-analysis-65f42ea5d26e

G, A. (2018, June 22). A review of Dropout as applied to RNNs. *medium*. From https://medium.com/@bingobee01/a-review-of-dropout-as-applied-to-rnns-72e79ecd5b7b

IJAS, A. H. (2019, November 2). Build a simple predictive keyboard using python and Keras. From https://medium.com/analytics-vidhya/build-a-simple-predictive-keyboard-using-python-and-keras-b78d3c88cffb

Koupanou, N. (2019, September 20). Exploring wild west of natural language generation — from n-gram and RNNs to Seq2Seq. *towardsdatascience*. From https://towardsdatascience.com/exploring-wild-west-of-natural-language-generation-from-n-gram-and-rnns-to-seq2seq-2e816edd89c6

RNN Regularization: Which Component to Regularize? (2019, November). *Stack Overflow*. From https://stackoverflow.com/questions/48714407/rnn-regularization-which-component-to-regularize

Shamim Biswas, E. C. (2015, April). Sentiment Analysis with Gated Recurrent Units. *Krishi Sanskriti Publications*. From https://www.krishisanskriti.org/vol_image/21Jul201503071043%20%20%20%20%20Faiyz%20Ahmad%202%20%20%20%20%20%20%20%20%20%2059-63.pdf

Shreyas, P. (2019, April 2). Sentiment analysis for text with Deep Learning. *towardsdatascience*. From https://towardsdatascience.com/sentiment-analysis-for-text-with-deep-learning-2f0a0c6472b5

Tanner, G. (2018, October 30). Generating text using a Recurrent Neural Network. *towardsdatascience*. From https://towardsdatascience.com/generating-text-using-a-recurrent-neural-network-1c3bfee27a5e