Third deep learning project

(Language model using recurrent neural network)

Supervised by: prof. Hazem Abbas

Team members:

Zeyad Tarek Mohamed

Mahmoud Adel Khorshed

Ahmed Salama

Queen’s university, School of computing

April 5, 2022

Introduction

A language model is nothing more than a probability distribution over words or word sequences. In practice, a language model predicts the likelihood of a given word sequence being "valid." In this situation, validity does not refer to grammatical validity at all. It implies that it is like how humans talk (or, more precisely, write) — which is what the language model learns.

Problem Formulation

In this problem, we want to predict what’s the next word after we write a sentence, we want to build a language model which can predict what's the next word and what's the probability of this word being the next word, The model will consider the context of the last 50 words of a particular sentence and predict the next possible word. We will be using methods of natural language processing, language modeling, and deep learning. We will start by analyzing the data followed by the pre-processing of the data. We will then tokenize this data and convert them into sequences and finally, we'll build and train some deep learning models and compare their performance with each other.

Data: dirty text file have **119112** words and **7490** unique words.

Input: **118632*50** training sequences. (**118632** here represent the number of samples and **50** represent the number of input words)

Output: **118632** training sequences. (Each value represents that output word).

Data mining function:

1- Load and read the text data.
2- Analyzing and pre-processing the data.
3- Build and train the models
4- Classification and prediction
5- Get insights from the results.

Challenges:

1- These types of problems require a huge amount of data and what we have here is a small one.
2- This is an open-ended problem so we have a lot of model architectures that we may want to try and choose the best one.

3- Non-accurate results due to a lack of information.
4- We use recurrent models with a lot of recurrent units to memorize the information, so we need huge number of resources, especially **RAM**, at least **(16 GIGABYTE of RAM).**

Impact:

Solving kind of this problem will help people a lot when they search for something on the internet and they don't remember the exact context of the sentence, also when they send an email or while they're using chats, that will save much time.

Experimental protocol

1- Load and read the data.
2- Understand the nature of this data
3- Pre-processing the text data

   3.1 Remove punctuation from the text file.
   3.2 Remove non-alphabetic words
   3.3 Tokenize the text data.
   3.4 Convert each token to numerical value.
   3.5 Organize the numerical data into 118,632 training sequences each sequence has 51 values.
   3.6 Save 50 value from each sequence in a variable and they will represent the input data and save the last numerical value from each sequence in a variable, and they will represent the output data.
   3.7 Convert the output feature to one hot encoded data.

4- Start building the models
5- Train each model.
6- Plotting the performance graph
7- Start predicting in the context of last 100 words.
8- Get insights from the results.

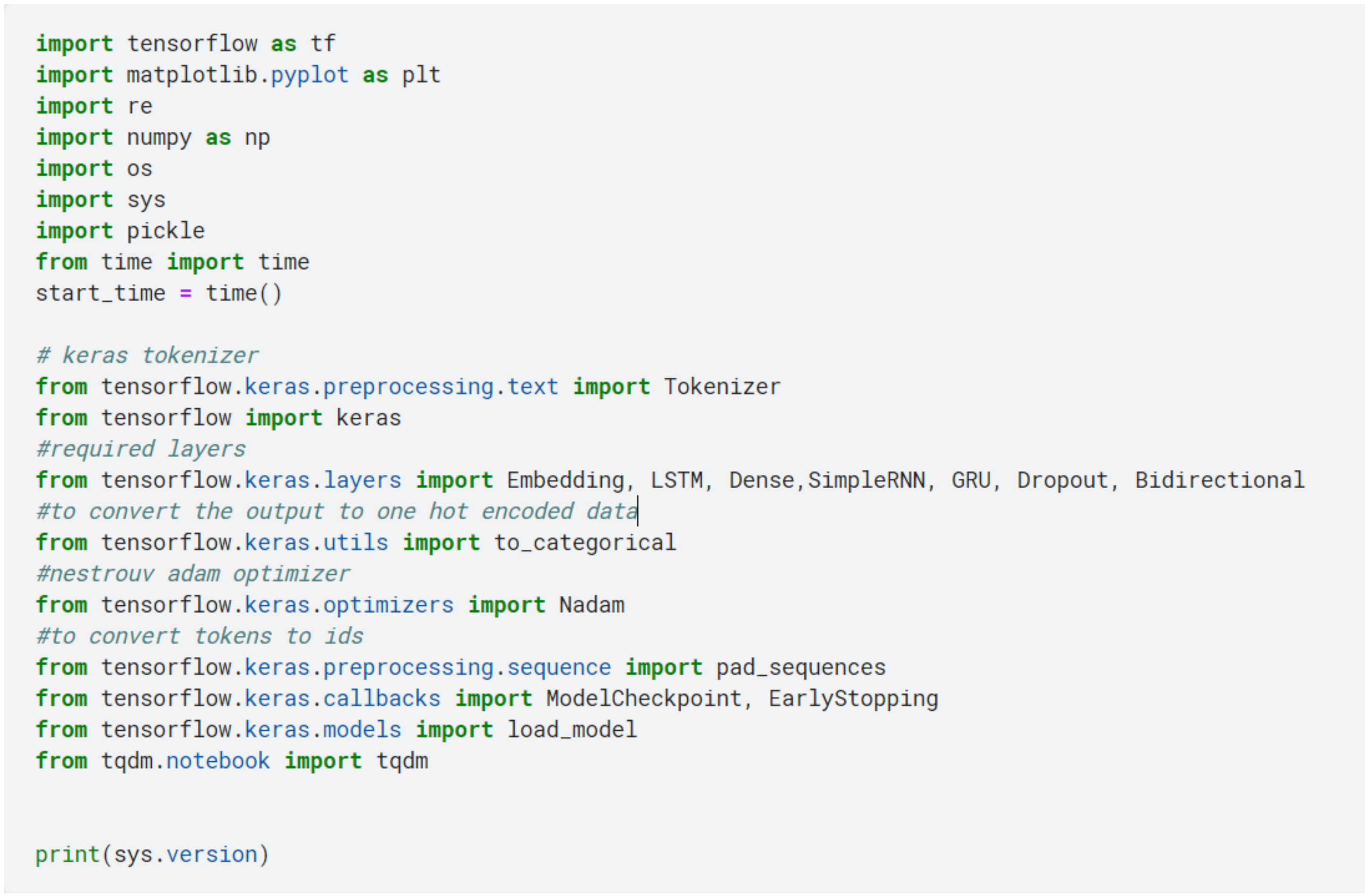
Code Explanation

This code ran on Kaggle environment using **Nvidia Tesla P100 GPUs** to train the deep learning models and **16GB of RAM**, Note: less than these resources the model may crash because I used 1000 units in each recurrent layer.

So please try to run it on Kaggle environment.

First cell: I imported all required packages.

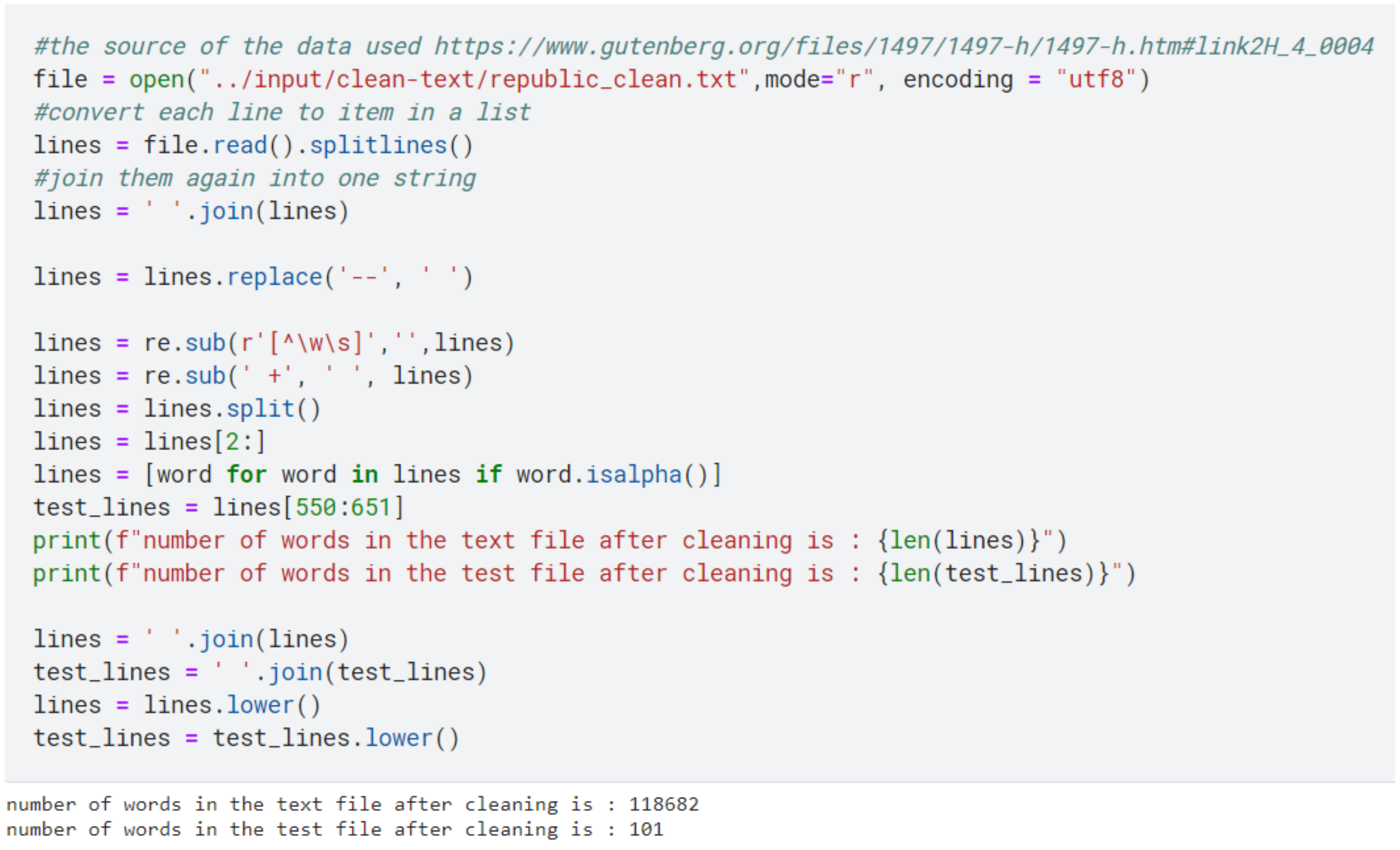
You can see what each package will do for us.



```
import tensorflow as tf
import matplotlib.pyplot as plt
import re
import numpy as np
import os
import sys
import pickle
from time import time
start_time = time()

# keras tokenizer
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow import keras
#required layers
from tensorflow.keras.layers import Embedding, LSTM, Dense,SimpleRNN, GRU, Dropout, Bidirectional
#to convert the output to one hot encoded data
from tensorflow.keras.utils import to_categorical
#nestrouv adam optimizer
from tensorflow.keras.optimizers import Nadam
#to convert tokens to ids
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.models import load_model
from tqdm.notebook import tqdm


print(sys.version)
```

Second cell:

I Started loading and reading the data and I clean the text file; I removed the punctuation, and I removed the non-alphabetic words, then I took **101** words from that file to use them as test data, then I convert the training data and test data to lower case.

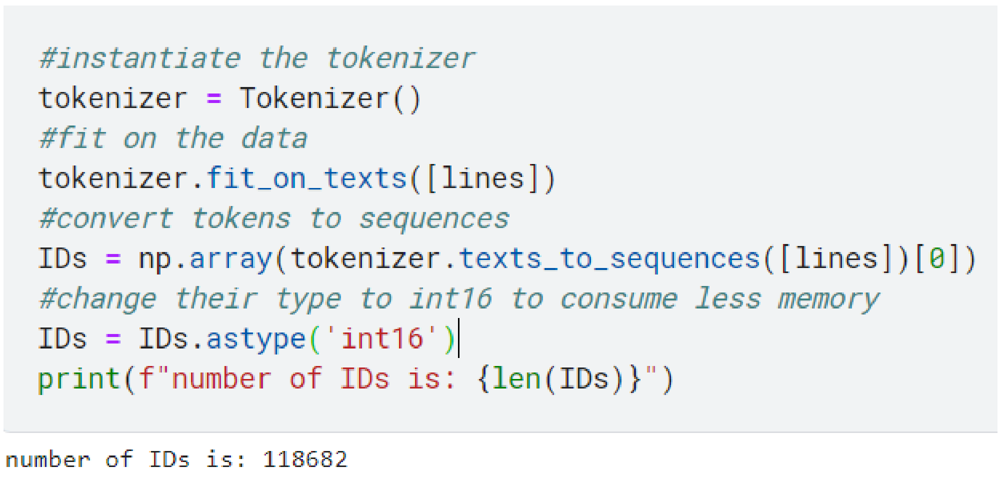


```python
#the source of the data used https://www.gutenberg.org/files/1497/1497-h/1497-h.htm#link2H_4_0004
file = open("../input/clean-text/republic_clean.txt",mode="r", encoding = "utf8")
#convert each line to item in a list
lines = file.read().splitlines()
#join them again into one string
lines = ' '.join(lines)

lines = lines.replace('--', ' ')

lines = re.sub(r'[^\w\s]','',lines)
lines = re.sub(' +', ' ', lines)
lines = lines.split()
lines = lines[2:]
lines = [word for word in lines if word.isalpha()]
test_lines = lines[550:651]
print(f"number of words in the text file after cleaning is : {len(lines)}")
print(f"number of words in the test file after cleaning is : {len(test_lines)}")

lines = ' '.join(lines)
test_lines = ' '.join(test_lines)
lines = lines.lower()
test_lines = test_lines.lower()
```

```
number of words in the text file after cleaning is : 118682
number of words in the test file after cleaning is : 101
```

Third cell:

Here I continued pre-processing the text data, I tokenized the text data using keras tokenizer and I convert each **token** (word) to **ID** (numerical value).



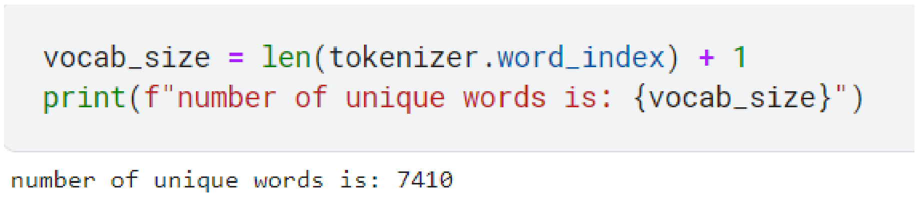
```python
#instantiate the tokenizer
tokenizer = Tokenizer()
#fit on the data
tokenizer.fit_on_texts([lines])
#convert tokens to sequences
IDs = np.array(tokenizer.texts_to_sequences([lines])[0])
#change their type to int16 to consume less memory
IDs = IDs.astype('int16')
print(f"number of IDs is: {len(IDs)}")
```

```
number of IDs is: 118682
```

Note: The number of **IDs** is equal to the number of words in the text file.

Fourth Cell:

I just saw how many unique words in the text file.

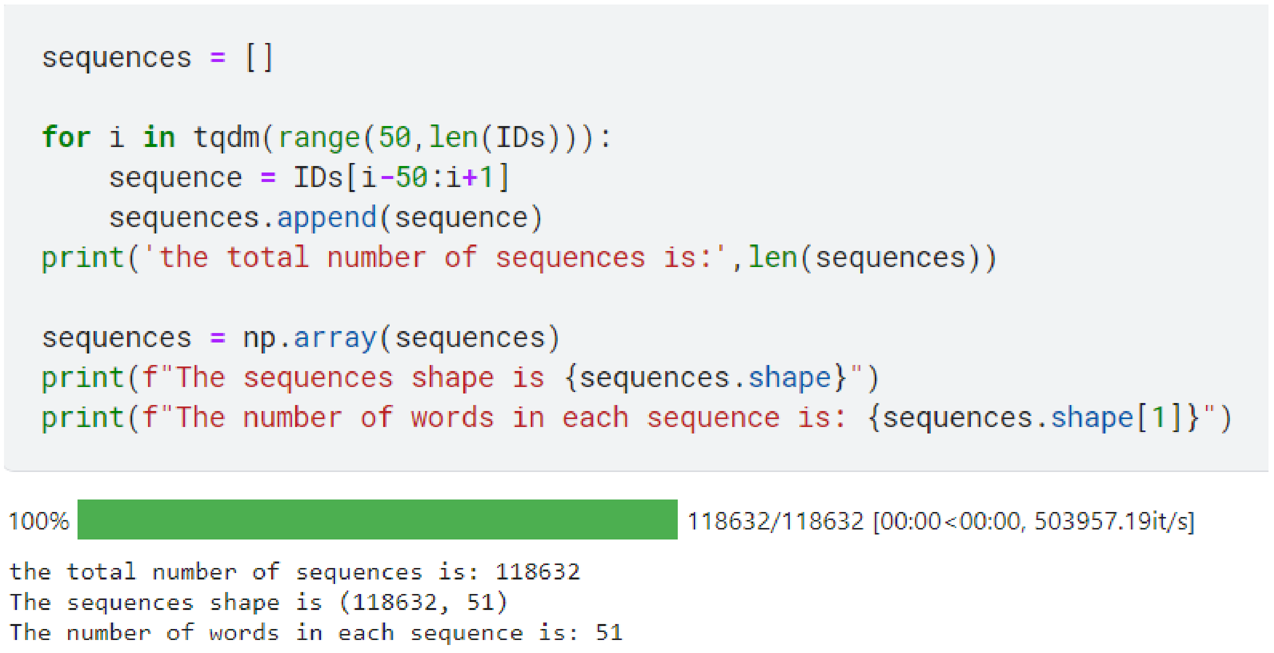


```python
vocab_size = len(tokenizer.word_index) + 1
print(f"number of unique words is: {vocab_size}")
```

```
number of unique words is: 7410
```

Fifth cell (important step):

I organized the **IDs** into sequences each sequence has **51** values.

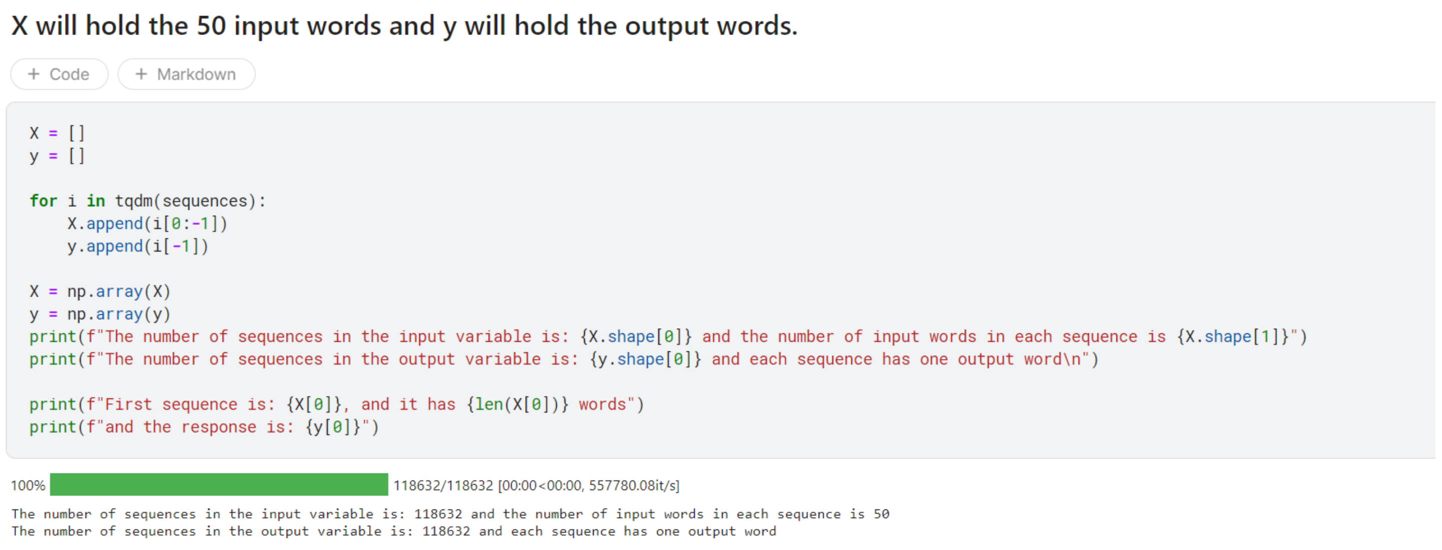


```python
sequences = []

for i in tqdm(range(50,len(IDs))):
    sequence = IDs[i-50:i+1]
    sequences.append(sequence)
print('the total number of sequences is:',len(sequences))

sequences = np.array(sequences)
print(f"The sequences shape is {sequences.shape}")
print(f"The number of words in each sequence is: {sequences.shape[1]}")
```

```
100% 118632/118632 [00:00<00:00, 503957.19it/s]
the total number of sequences is: 118632
The sequences shape is (118632, 51)
The number of words in each sequence is: 51
```

Sixth cell:

I stored the **50** values in each sequence in one variable, and they will represent the input data, and the last word will be the predicted word and I stored them in another variable.

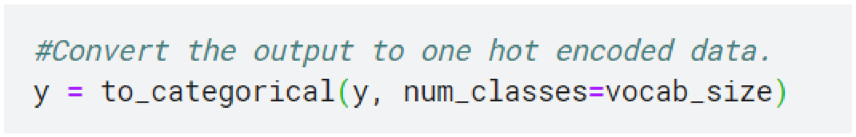

X will hold the 50 input words and y will hold the output words.

```python
X = []
y = []

for i in tqdm(sequences):
    X.append(i[0:-1])
    y.append(i[-1])

X = np.array(X)
y = np.array(y)
print(f"The number of sequences in the input variable is: {X.shape[0]} and the number of input words in each sequence is {X.shape[1]}")
print(f"The number of sequences in the output variable is: {y.shape[0]} and each sequence has one output word\n")

print(f"First sequence is: {X[0]}, and it has {len(X[0])} words")
print(f"and the response is: {y[0]}")
```

```
100% 118632/118632 [00:00<00:00, 557780.08it/s]
The number of sequences in the input variable is: 118632 and the number of input words in each sequence is 50
The number of sequences in the output variable is: 118632 and each sequence has one output word
```

Seventh cell:

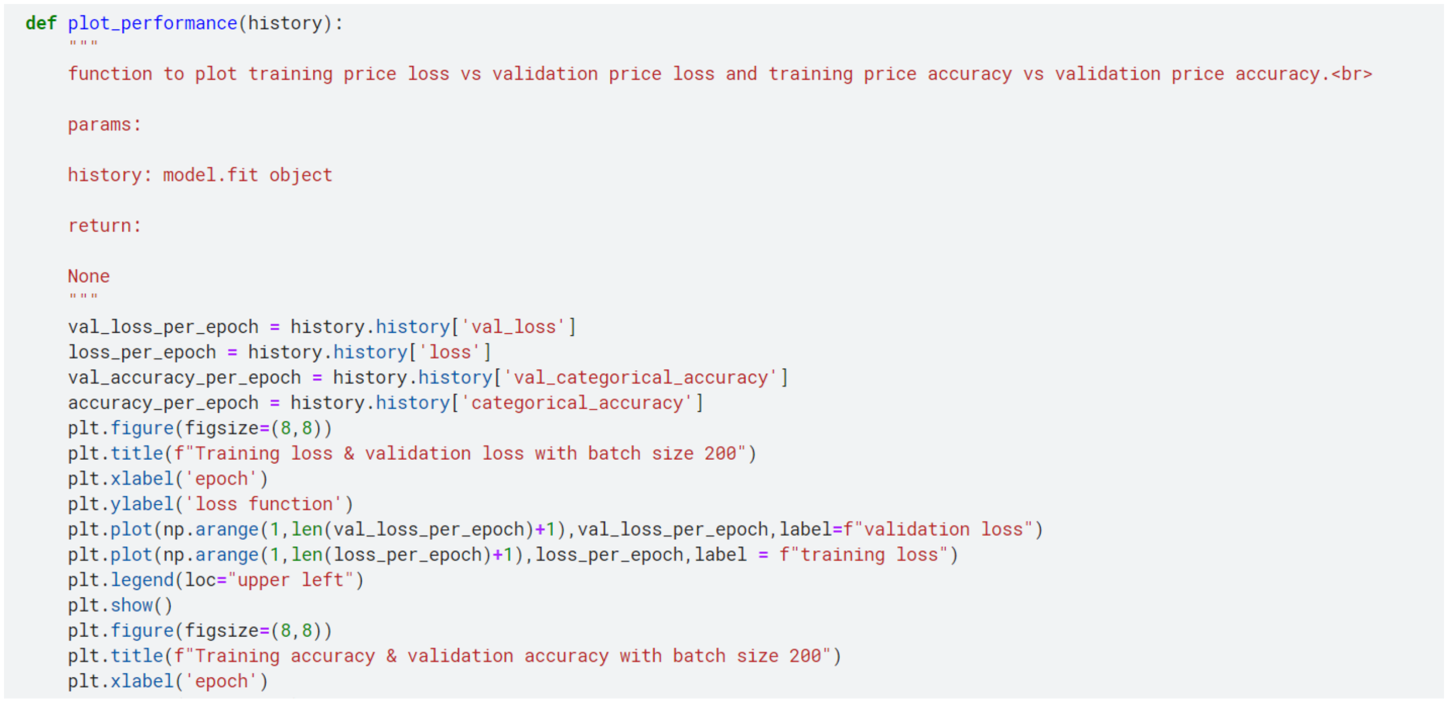
Convert output words to one hot encoded data because one hot encoded output is better with multi-class classification problem.



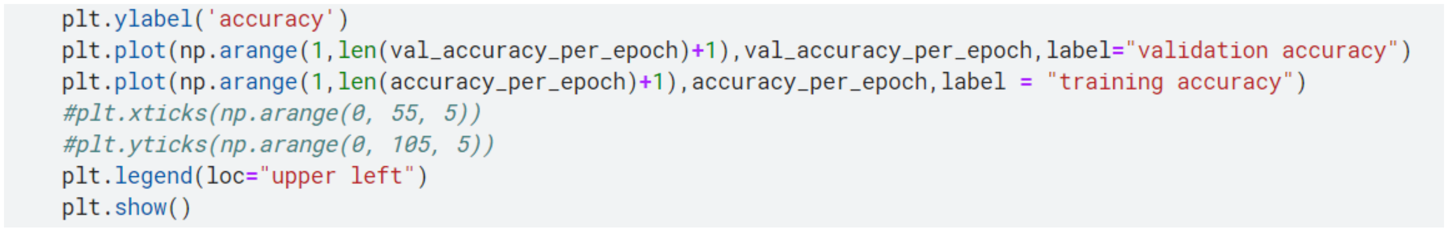
```python
#Convert the output to one hot encoded data.
y = to_categorical(y, num_classes=vocab_size)
```

Eighth cell:

Function to plot the validation loss and training loss over epochs and validation accuracy and training accuracy over epochs for one model at time, and it takes the history of the model as parameter.



```python
def plot_performance(history):
    """
    function to plot training price loss vs validation price loss and training price accuracy vs validation price accuracy.<br>

    params:

    history: model.fit object

    return:

    None
    """
    val_loss_per_epoch = history.history['val_loss']
    loss_per_epoch = history.history['loss']
    val_accuracy_per_epoch = history.history['val_categorical_accuracy']
    accuracy_per_epoch = history.history['categorical_accuracy']
    plt.figure(figsize=(8,8))
    plt.title(f"Training loss & validation loss with batch size 200")
    plt.xlabel('epoch')
    plt.ylabel('loss function')
    plt.plot(np.arange(1,len(val_loss_per_epoch)+1),val_loss_per_epoch,label=f"validation loss")
    plt.plot(np.arange(1,len(loss_per_epoch)+1),loss_per_epoch,label = f"training loss")
    plt.legend(loc="upper left")
    plt.show()
    plt.figure(figsize=(8,8))
    plt.title(f"Training accuracy & validation accuracy with batch size 200")
    plt.xlabel('epoch')
```

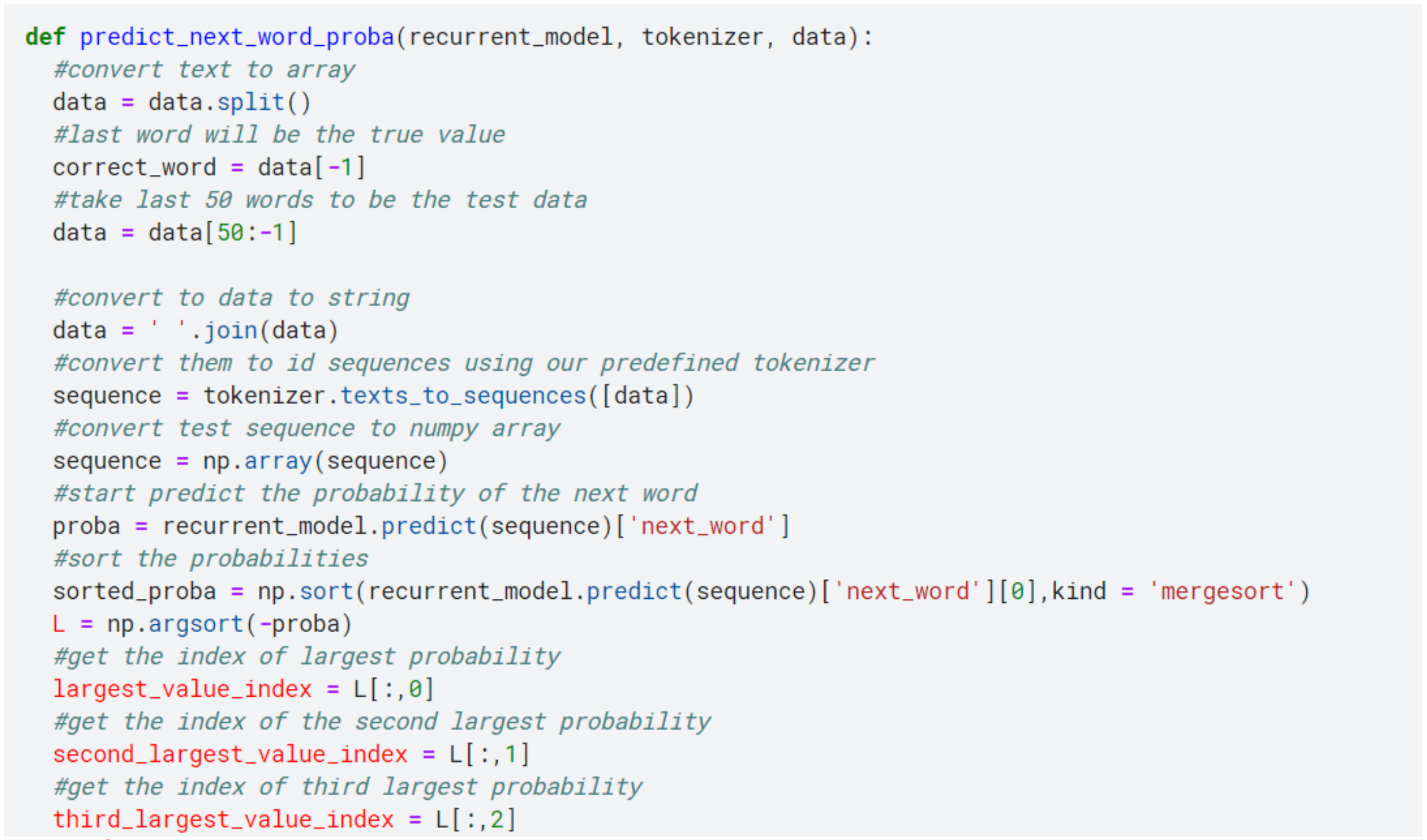


```python
    plt.ylabel('accuracy')
    plt.plot(np.arange(1,len(val_accuracy_per_epoch)+1),val_accuracy_per_epoch,label="validation accuracy")
    plt.plot(np.arange(1,len(accuracy_per_epoch)+1),accuracy_per_epoch,label = "training accuracy")
    #plt.xticks(np.arange(0, 55, 5))
    #plt.yticks(np.arange(0, 105, 5))
    plt.legend(loc="upper left")
    plt.show()
```

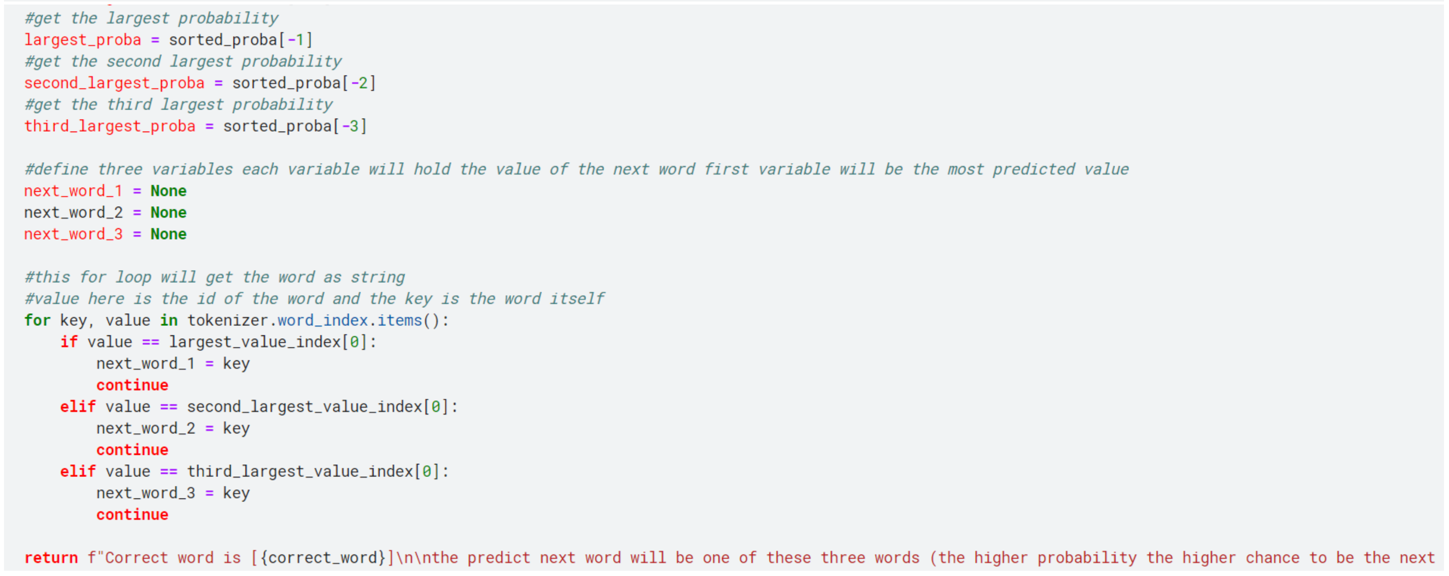
Nineth cell:

Function to predict the probabilities of all unique words and take the three highest (and maybe the correct next word will be one of these three words.), and it will print to probable 3 next word and their probabilities.

Note: more explanation in the test part.



```
def predict_next_word_proba(recurrent_model, tokenizer, data):
  #convert text to array
  data = data.split()
  #last word will be the true value
  correct_word = data[-1]
  #take last 50 words to be the test data
  data = data[50:-1]

  #convert to data to string
  data = ' '.join(data)
  #convert them to id sequences using our predefined tokenizer
  sequence = tokenizer.texts_to_sequences([data])
  #convert test sequence to numpy array
  sequence = np.array(sequence)
  #start predict the probability of the next word
  proba = recurrent_model.predict(sequence)['next_word']
  #sort the probabilities
  sorted_proba = np.sort(recurrent_model.predict(sequence)['next_word'][0],kind = 'mergesort')
  L = np.argsort(-proba)
  #get the index of largest probability
  largest_value_index = L[:,0]
  #get the index of the second largest probability
  second_largest_value_index = L[:,1]
  #get the index of third largest probability
  third_largest_value_index = L[:,2]
```



```
#get the largest probability
largest_proba = sorted_proba[-1]
#get the second largest probability
second_largest_proba = sorted_proba[-2]
#get the third largest probability
third_largest_proba = sorted_proba[-3]

#define three variables each variable will hold the value of the next word first variable will be the most predicted value
next_word_1 = None
next_word_2 = None
next_word_3 = None

#this for loop will get the word as string
#value here is the id of the word and the key is the word itself
for key, value in tokenizer.word_index.items():
    if value == largest_value_index[0]:
        next_word_1 = key
        continue
    elif value == second_largest_value_index[0]:
        next_word_2 = key
        continue
    elif value == third_largest_value_index[0]:
        next_word_3 = key
        continue

return f"Correct word is [{correct_word}]\n\nthe predict next word will be one of these three words (the higher probability the higher chance to be the next
```

Start building, training and testing the models.

I built six recurrent models, to test them on the test data and choose the best one.

All the models have common things and I'll explain them in the next page.

- **Embedding layer:** Represent words as semantically meaningful dense real-valued vectors and uses a distributed representation for words so that different words with similar meanings will have a similar representation. and its shape will be **(the number of unique words * input length)**.

- **Dense layer:** dense layer after recurrent layer to add more trainable parameters and to improve the accuracy.

- **Output layer:** will have 7410 neurons (number of unique words) each neuron represents the probability of the next word (highest probability will be the next word)

- All models have **Adam optimizer with nesterov momentum** and learning rate **0.001**, and I used it instead of **Adam optimizer with momentum**, because **Nadam** will reach to the optimal minimum faster than **Adam optimizer**.

- All models have **categorical_crossentropy** as **loss function method** and **categorical_accuracy** to calculate the accuracy since the output is **one-hot-encoded data**.

- All models will be optimized using **50 epochs** and **200 batch size** to make the training process faster.

- All models will be trained on **GPU**.

- In the training process there's an **early stopping monitor** to ensure that the validation accuracy increases after every epoch otherwise it will stop after **5 epochs**.

- Each model will be saved in the **h5 file** format, and this file will be the best trained model with **best validation accuracy** to use it again in the prediction step.

- The data will be divided into the **training data (80% of the whole data)** and the **validation data (20% of the whole data)** just before the training process.

- The difference between each model is, that is each model have different recurrent neural networks between the embedding layer and the dense layer.

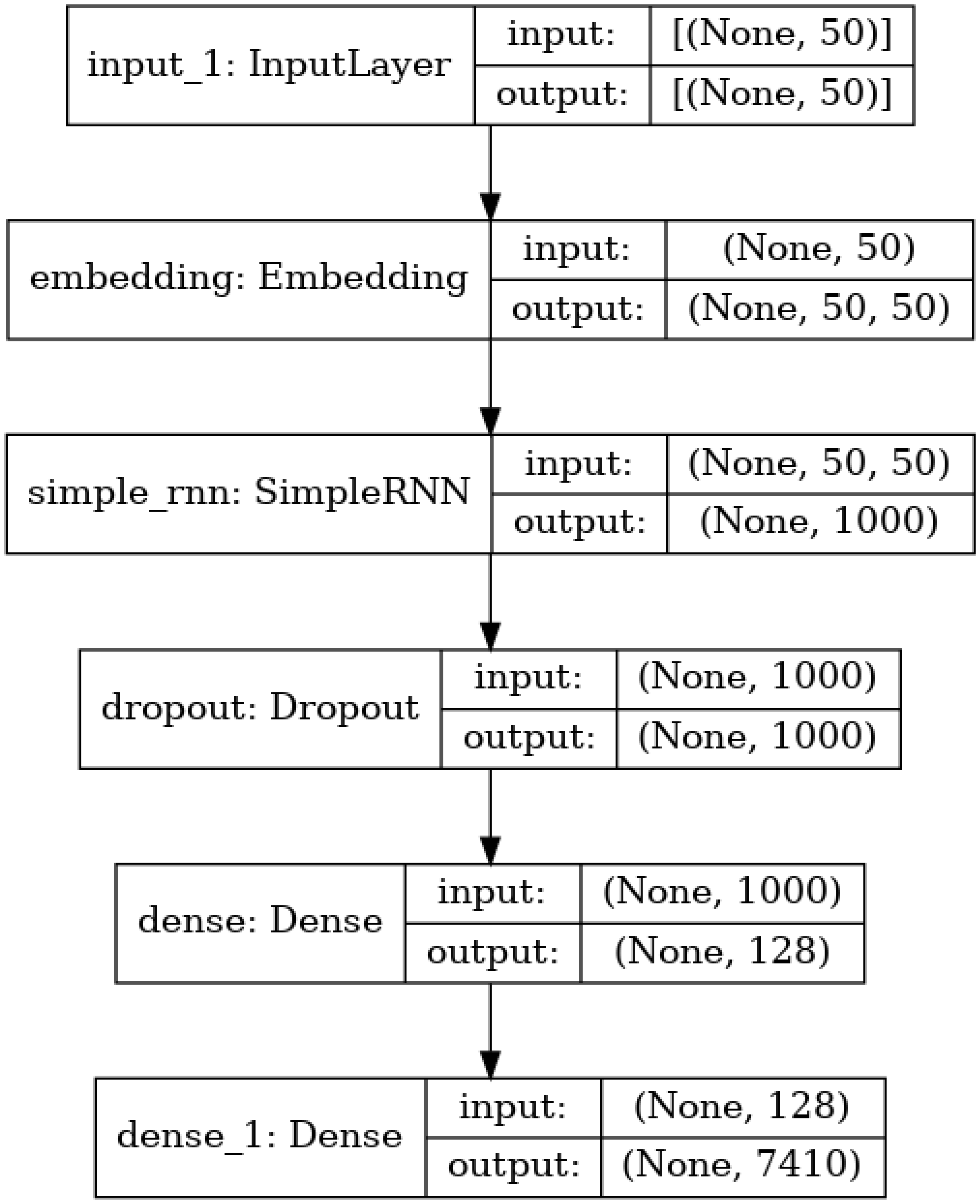
- The test data used to test all models is: input data: but at my age i can hardly get to the city and th erefore you should come oftener to the piraeus for let me tell you that the more the pleasures of the body fade away the greater to me is the pleasure and charm of conversation do not then deny my request but make our house your resort and keep company with these young men we are old friends and you will be quite at home with us i replied there is nothing which for my part i like better cephalus than conversing with aged men for i regard them .....

And the output word is **‘as’**

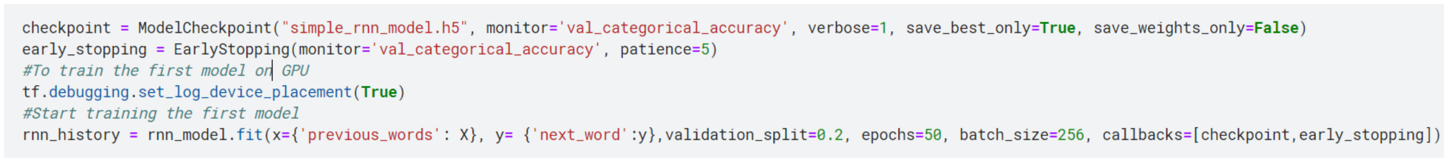
Building the first model

In the first model I used a simple recurrent neural network in order to see if this kind of problems can be handled by the simplest recurrent neural networks or not.

Total params: 2,505,518
Trainable params: 2,505,518
Non-trainable params: 0



Training the first model



```
checkpoint = ModelCheckpoint("simple_rnn_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=5)
#To train the first model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the first model
rnn_history = rnn_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stopping])
```

Total time taken to finish the training process was: 15 minutes and 32 seconds.

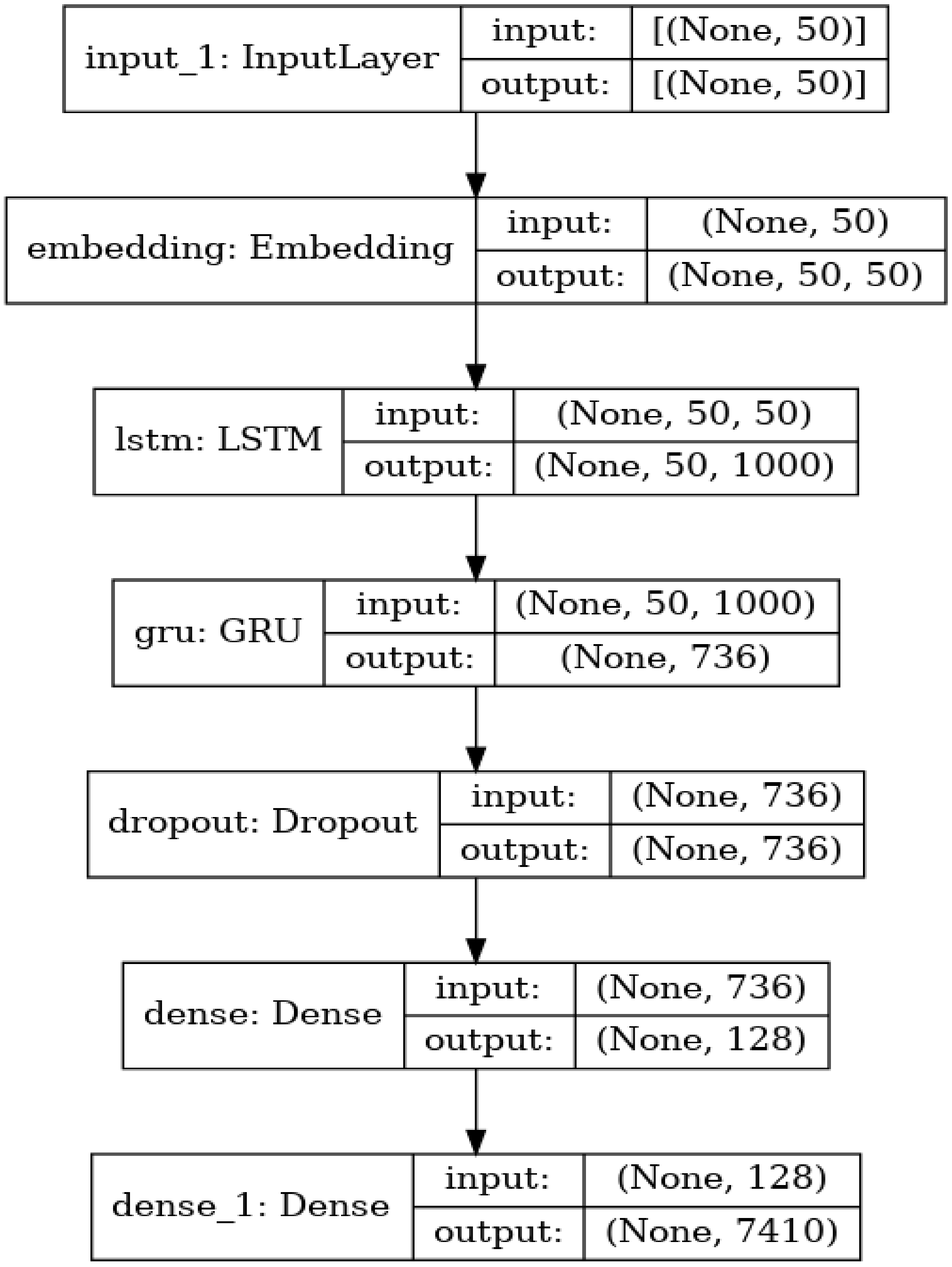
Building the second model

In this model I used a LSTM network with 1000 units followed by GRU network with 736 units because they have gates to forget data and memorize the others also, they are better in resisting the vanishing gradient and gradient explosion problems.

Total params: 9,462,230

Trainable params: 9,462,230

Non-trainable params: 0



Training the second model

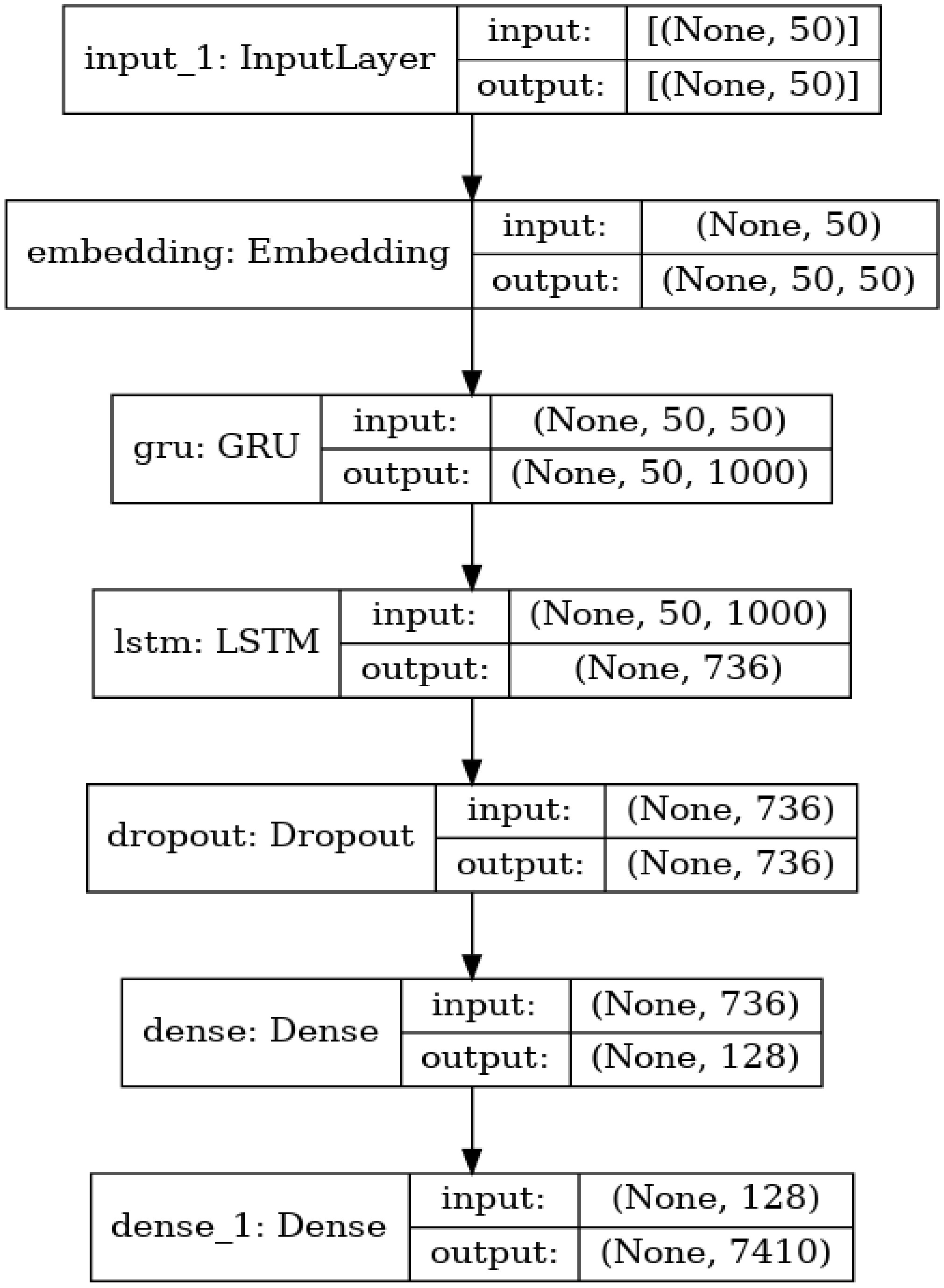
```
checkpoint = ModelCheckpoint("second_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=5)
#To train the second model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the second model.
second_history = second_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stopp
```

Total time taken to finish the training process was: 15 minutes and 31 seconds.

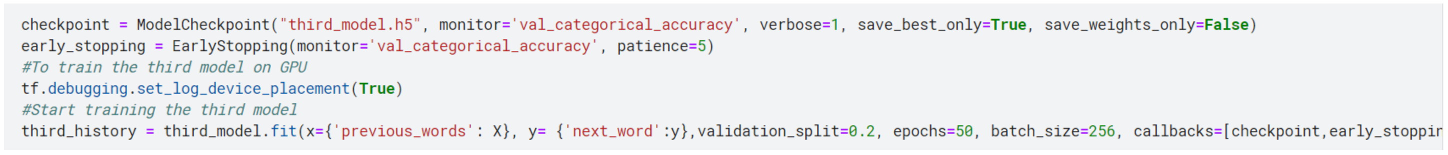
Building the third model

In this model I used a GRU network followed by LSTM network to see if the change in orders will improve accuracy or not.

Total params: 9,690,454
Trainable params: 9,690,454
Non-trainable params: 0



Training the third model



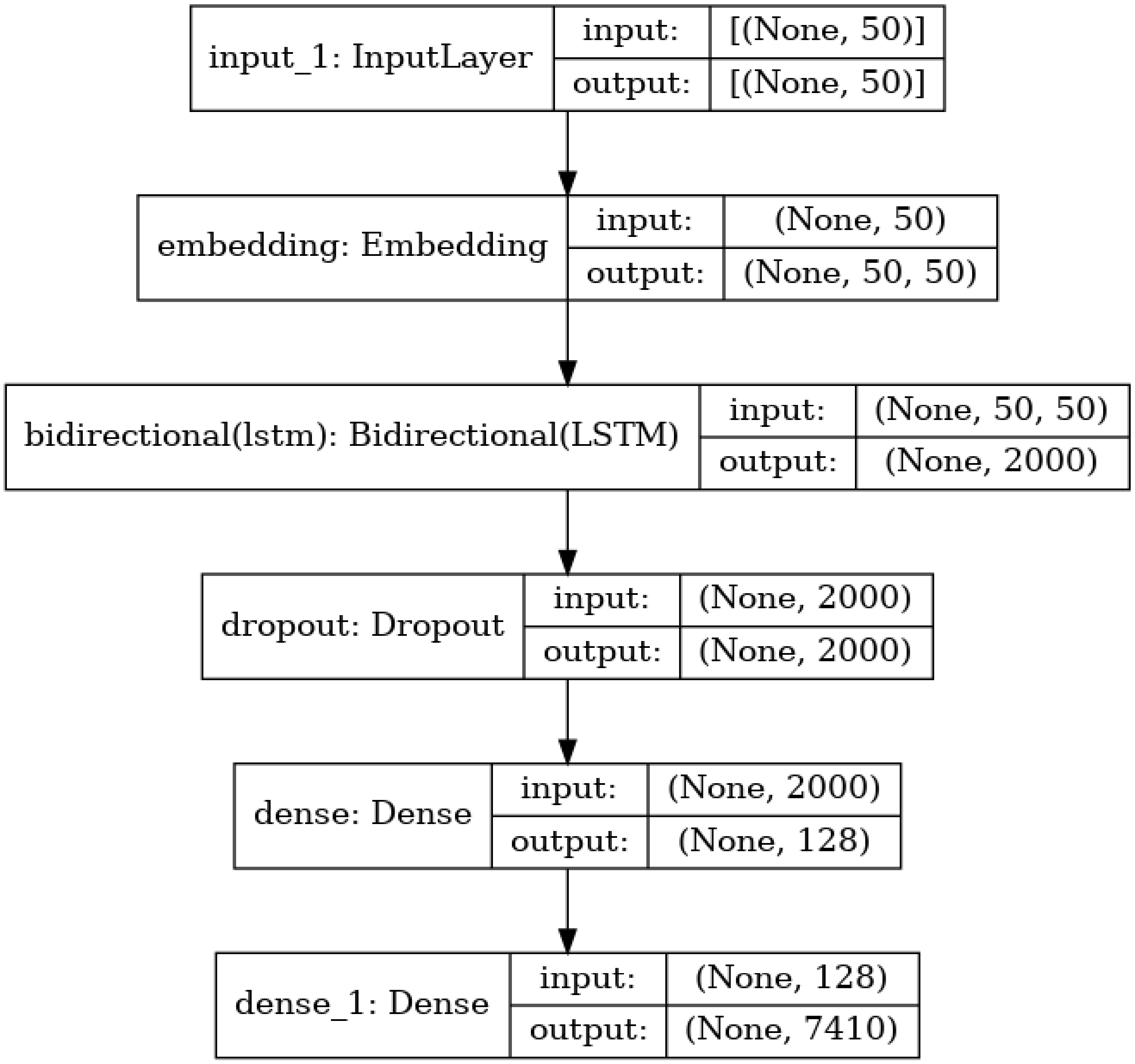
```
checkpoint = ModelCheckpoint("third_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=5)
#To train the third model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the third model
third_history = third_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stoppi
```

Total time taken to finish the training process was: 23 minutes and 25 seconds.

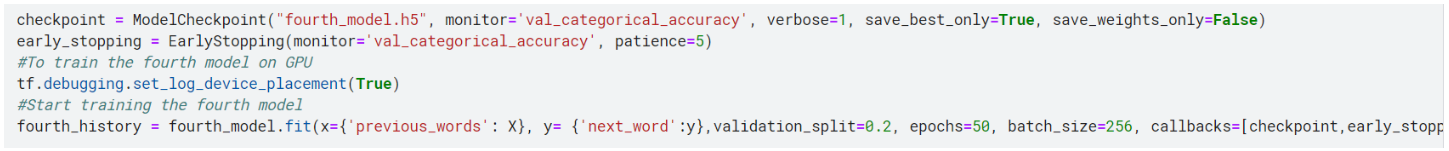
Building the fourth model

In this model I used a Bidirectional LSTM network with 1000 units in order to have and memorize the sequence information in both directions, backwards (future to past) and forward (past to future), in order to improve the accuracy.

Total params: 9,990,518
Trainable params: 9,990,518
Non-trainable params: 0



Training the fourth model

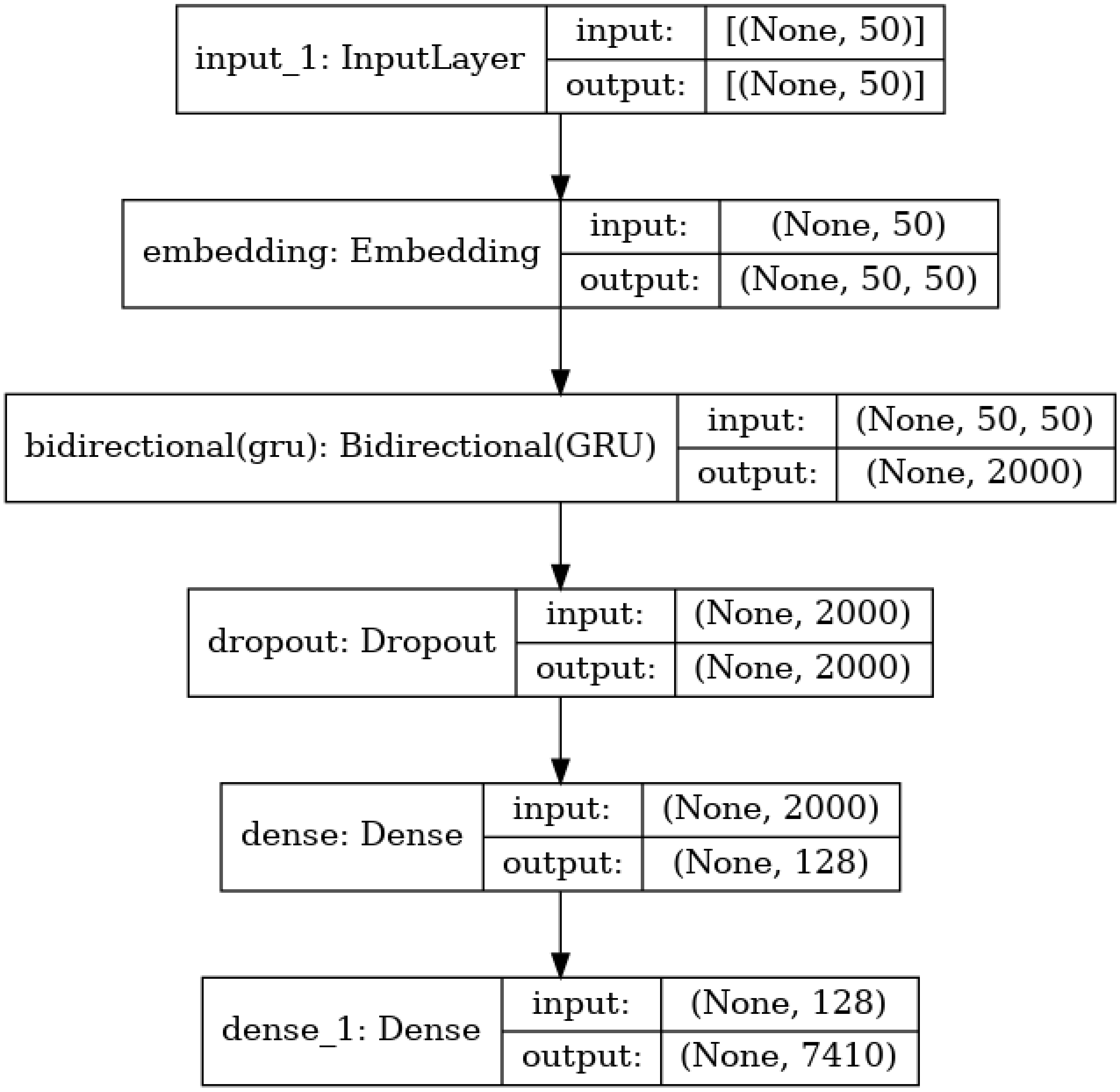

```
checkpoint = ModelCheckpoint("fourth_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=5)
#To train the fourth model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the fourth model
fourth_history = fourth_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stopp
```

Total time taken to finish the training process was: 34 minutes and 10 seconds.

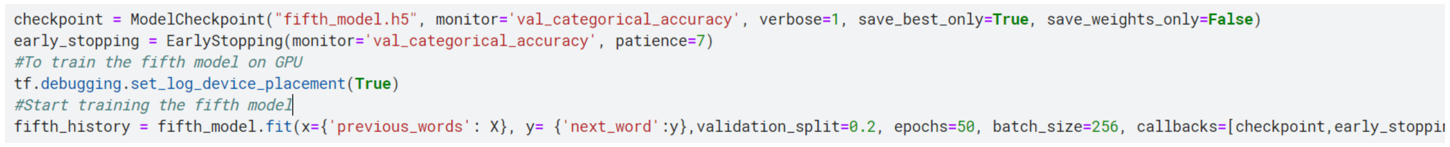
Building the fifth model

In this model I used a Bidirectional GRU network with 1000 units instead of Bidirectional LSTM because it’s more efficient than LSTM network, and to see if that will improve the accuracy or not.

Total params: 7,894,518
Trainable params: 7,894,518
Non-trainable params: 0



Training the fifth model



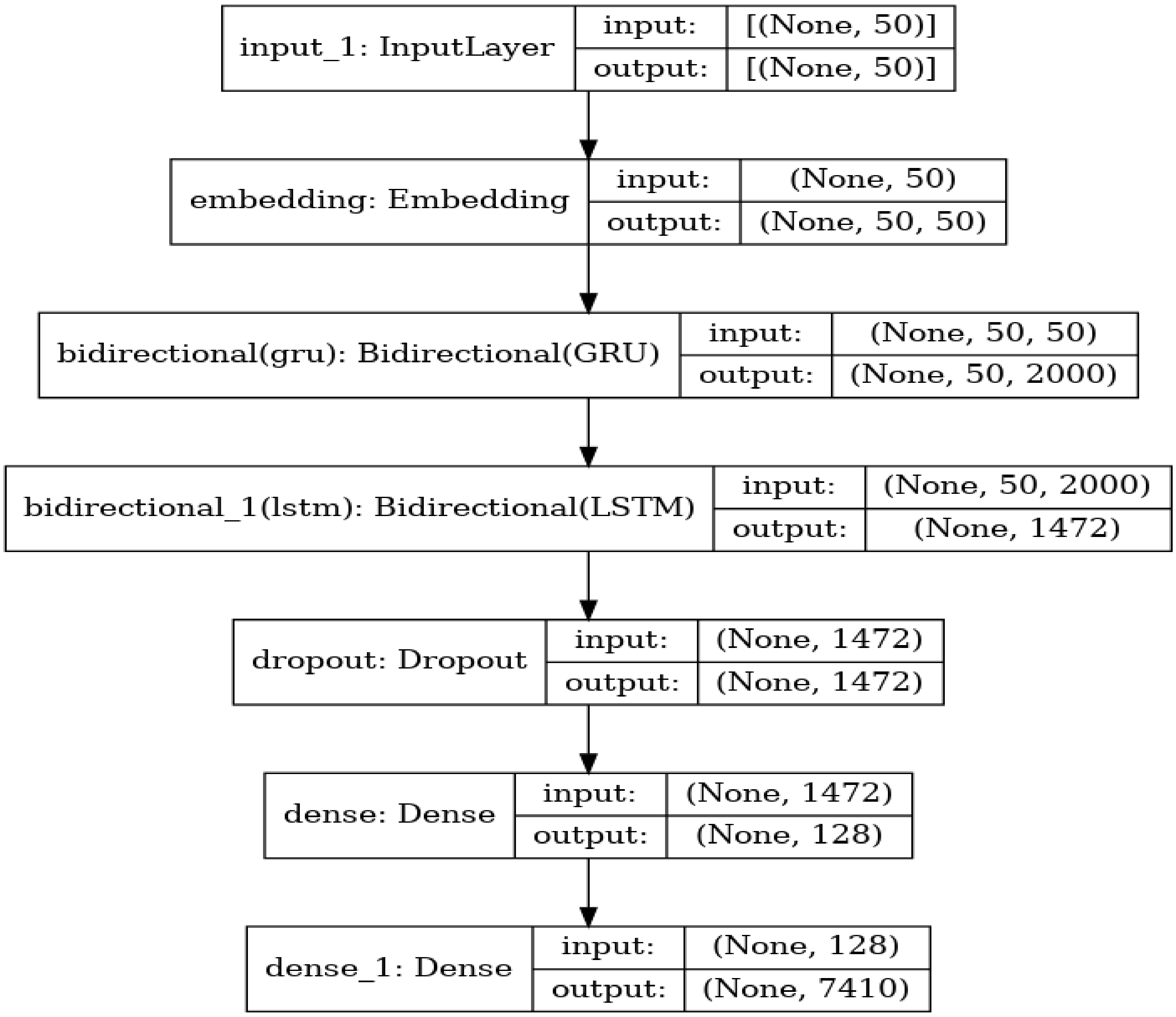
```
checkpoint = ModelCheckpoint("fifth_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=7)
#To train the fifth model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the fifth model
fifth_history = fifth_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stoppi
```

Total time taken to finish the training process was: 16 minutes and 2 seconds

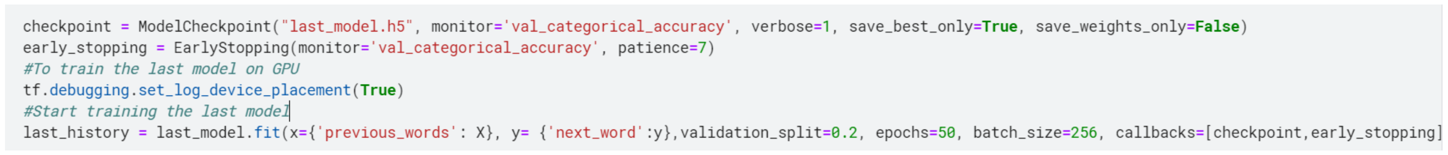
Building the last model

Since the bidirectional models was the best ones so, I tried to make a more complex model by using a bidirectional GUR network with 1000 units followed by bidirectional LSTM network with 1000 units, to see if more complex models can handle this kind of problems or not or it will be the same as simple ones.

Total params: 23,942,390
Trainable params: 23,942,390
Non-trainable params: 0



Training the last model



```
checkpoint = ModelCheckpoint("last_model.h5", monitor='val_categorical_accuracy', verbose=1, save_best_only=True, save_weights_only=False)
early_stopping = EarlyStopping(monitor='val_categorical_accuracy', patience=7)
#To train the last model on GPU
tf.debugging.set_log_device_placement(True)
#Start training the last model
last_history = last_model.fit(x={'previous_words': X}, y= {'next_word':y},validation_split=0.2, epochs=50, batch_size=256, callbacks=[checkpoint,early_stopping]
```

Total time taken to finish the training process was: 30 minutes and 40 seconds.

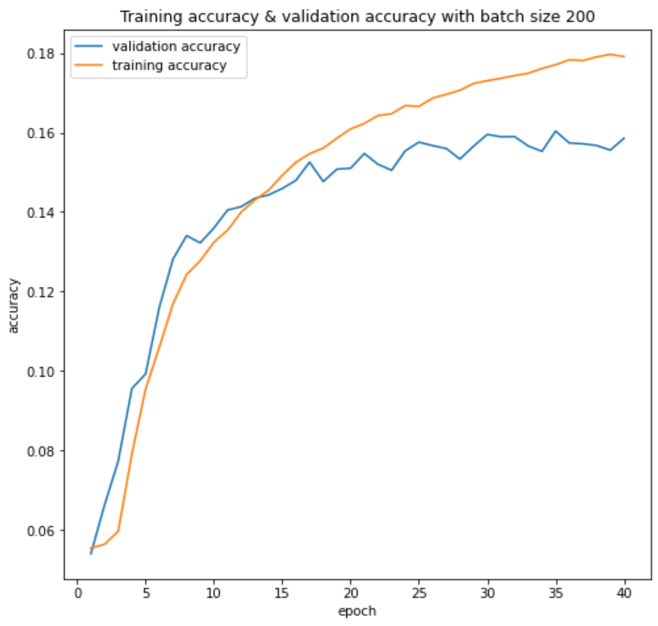
Results and comparisons

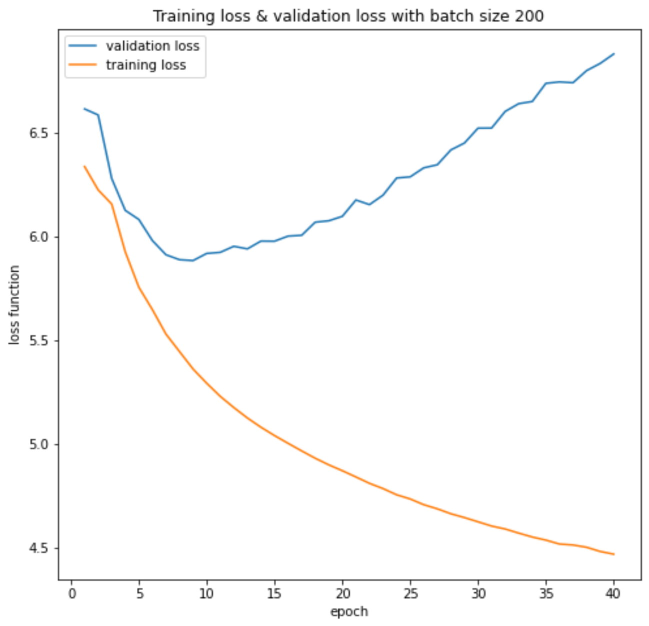
First model results

Best training loss: 4.4681
Best validation loss: 5.8874

Best training accuracy: 0.1791
Best validation accuracy: 0.16037

Performance curve and convergence curve for the first model





Results intuition: Here we find that the simplest model not the best in handling kind of this problems, but it can deal with it.

Start testing the first model

```
#load the best model which gave us the best validation accuracy to predict the test data.
rnn_model = load_model('simple_rnn_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(rnn_model, tokenizer, test_lines))
```

The output will be:

Correct word is [as]

the predict next word will be one of these three words **(the higher probability the higher chance to be the next word)**

the predicted next word is **[and]** with the largest probability **0.11699999868869781**
the next word could be **[in]** with the second largest probability **0.04899999871850014**
and it could be **[to]** with the third largest probability **0.035000000014901161**

That means the next word will be **"and"** although the correct word is **"as".**

I'll add more comments in all **results and comparisons** sections.

Second model results

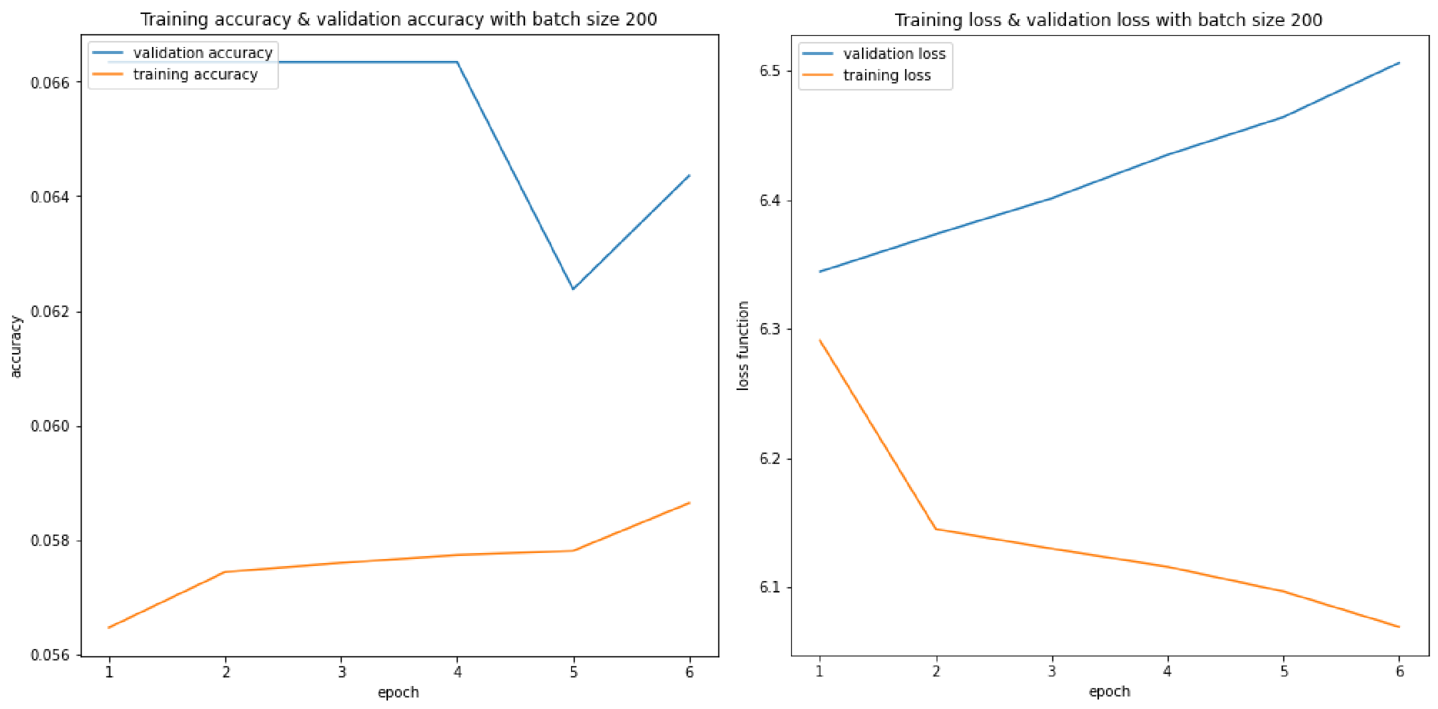
Best training loss: 6.0692

Best validation loss: 6.3443

Best training accuracy: 0.0586

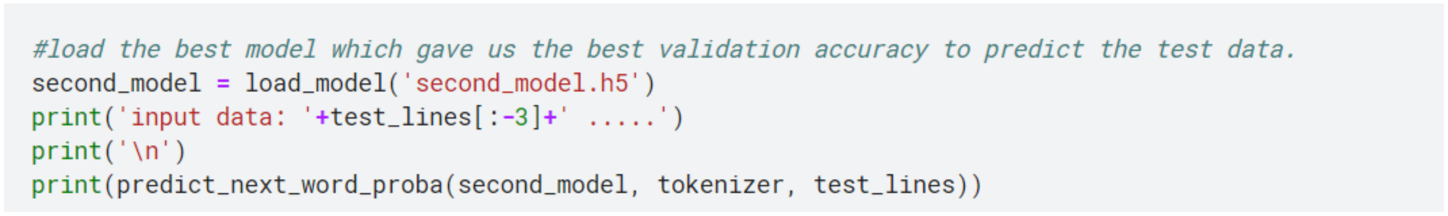
Best validation accuracy: 0.06634

Performance curve and convergence curve for the second model



Results intuition: one of the worst models you may use for this kind of problems, the models start overfitting from the first epoch even there's a dropout layer between GRU layer and the dense layer.

Start testing the second model



```
#load the best model which gave us the best validation accuracy to predict the test data.
second_model = load_model('second_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(second_model, tokenizer, test_lines))
```

The output will be:

Correct word is **[as]**

the predict next word will be one of these three words **(the higher probability the higher chance to be the next word)**

the predicted next word is **[the]** with the largest probability **0.035999998450279236**

the next word could be **[and]** with the second largest probability **0.029999999329447746**

and it could be **[of]** with the third largest probability **0.026000000536441803**

Third model results

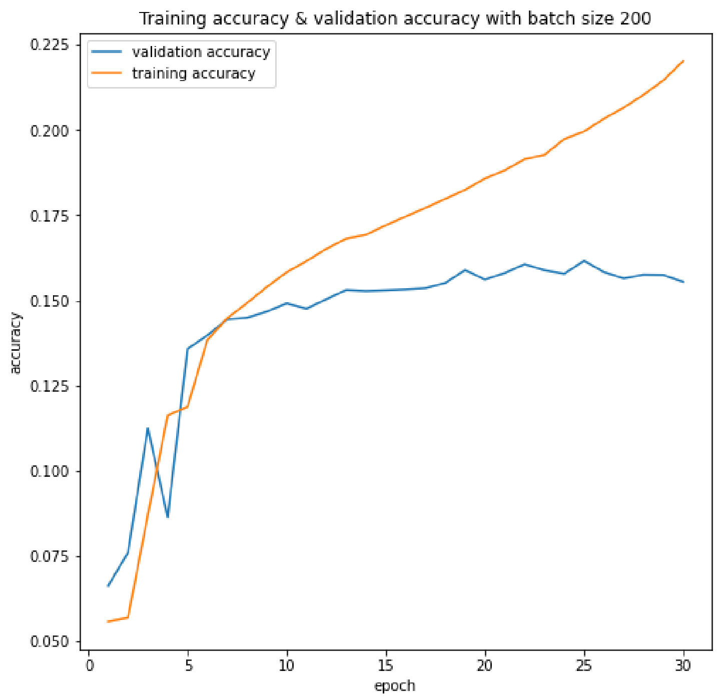
Best training loss: 4.1857

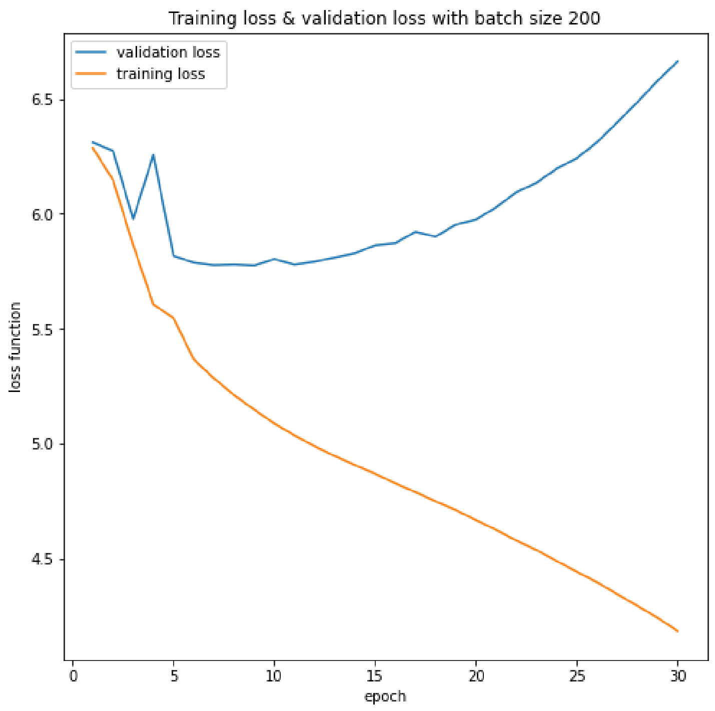
Best validation loss: 5.7769

Best training accuracy: 0.2202

Best validation accuracy: 0.16159

Performance curve and convergence curve for the third model





Results intuition: From this model results I found that the order of layers can flip the game although they are the same layers but what I did is I reverse the order and that's make the accuracy and the loss much better than the previous model.

Start testing the third model

```
#load the best model which gave us the best validation accuracy to predict the test data.
third_model = load_model('third_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(third_model, tokenizer, test_lines))
```

The output will be:

Correct word is **[as]**

the predict next word will be one of these three words **(the higher probability the higher chance to be the next word)**

the predicted next word is **[in]** with the largest probability **0.08299999684095383**

the next word could be **[and]** with the second largest probability **0.050999999046325684**

and it could be **[to]** with the third largest probability **0.03799999877810478**

Fourth model results

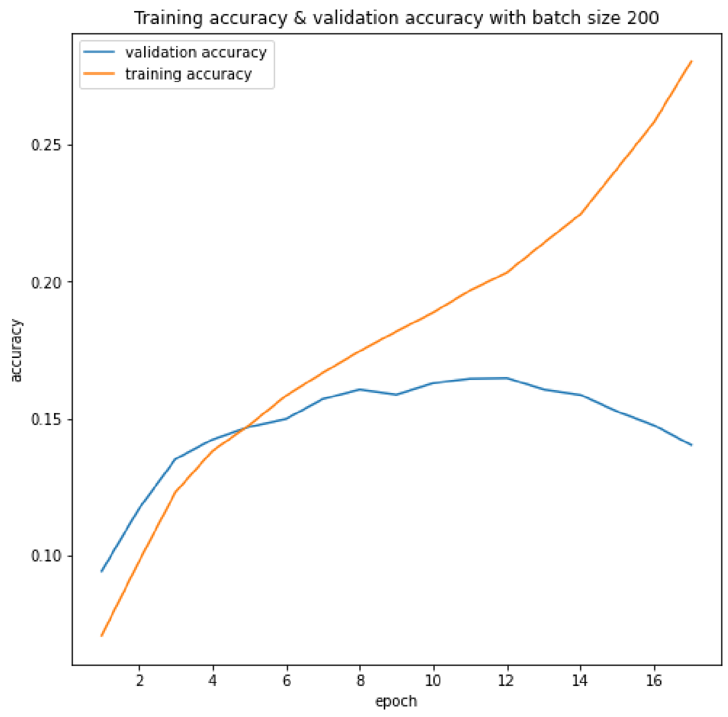
Best training loss: 3.4581

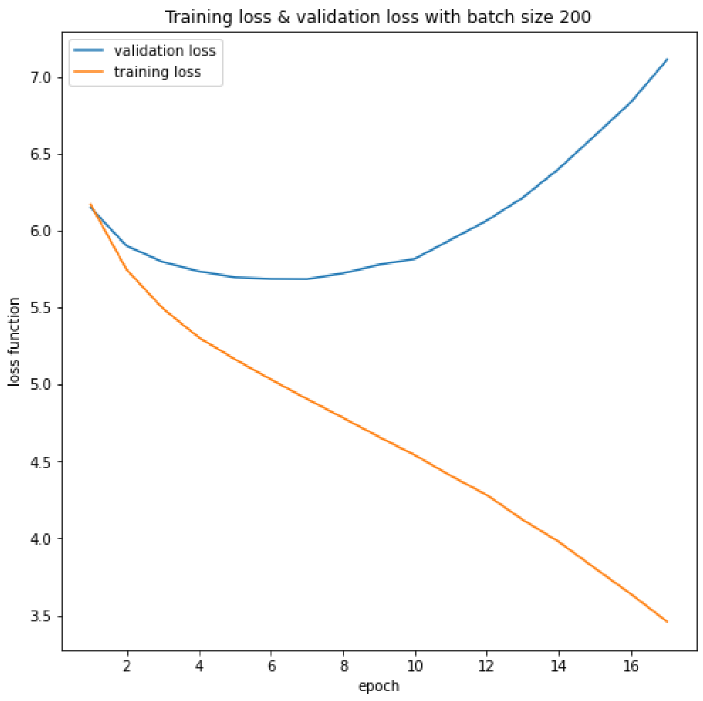
Best validation loss: 5.6837

Best training accuracy: 0.2803

Best validation accuracy: 0.16479

Performance curve and convergence curve for the fourth model





Results intuition: Using one bidirectional LSTM with 2000 units was better choice than GRU layer with 1000 unit followed by LSTM layer with 1000 and it got better accuracy.

Start testing the fourth model

```
#load the best model which gave us the best validation accuracy to predict the test data.
fourth_model = load_model('fourth_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(fourth_model, tokenizer, test_lines))
```

The output will be:

Correct word is **[as]**

the predict next word will be one of these three words **(the higher probability the higher chance to be the next word)**

the predicted next word is **[to]** with the largest probability 0.07500000298023224

the next word could be **[in]** with the second largest probability 0.05299999937415123

and it could be **[and]** with the third largest probability 0.04899999871850014

Fifth model results

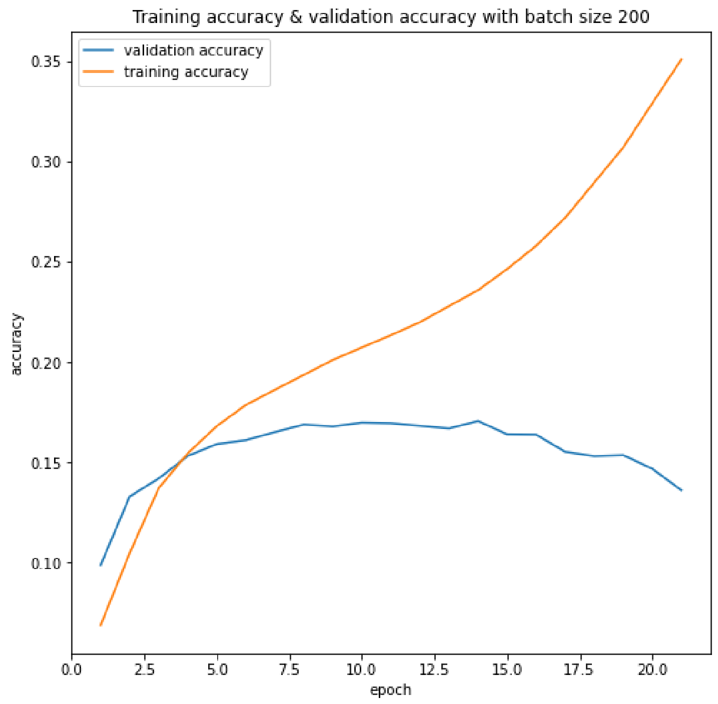
Best training loss: 3.0848

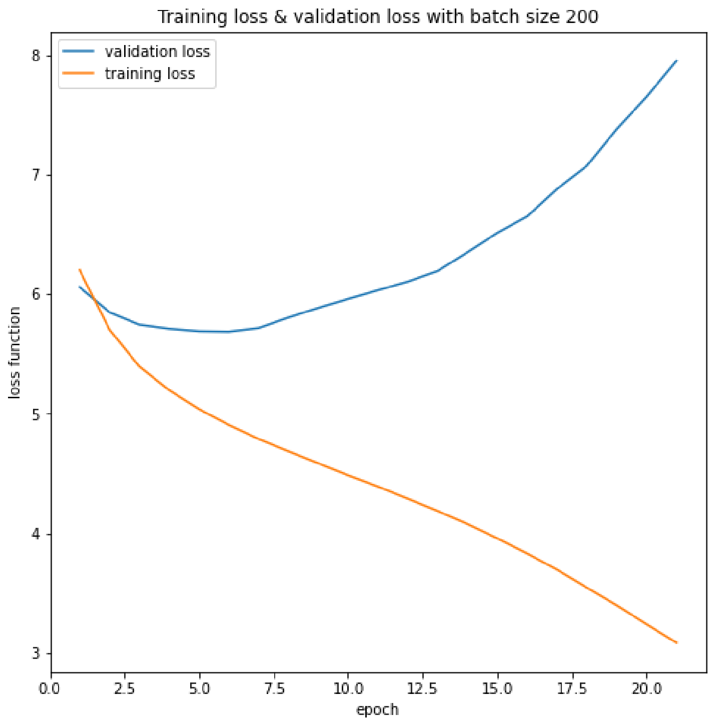
Best validation loss: 5.6880

Best training accuracy: 0.3509

Best validation accuracy: 0.17069

Performance curve and convergence curve for the fifth model





Results intuition: This trial was one of the best trials after the next model, it got the best training accuracy, and it was faster than previous trail in training time

Start testing the fifth model

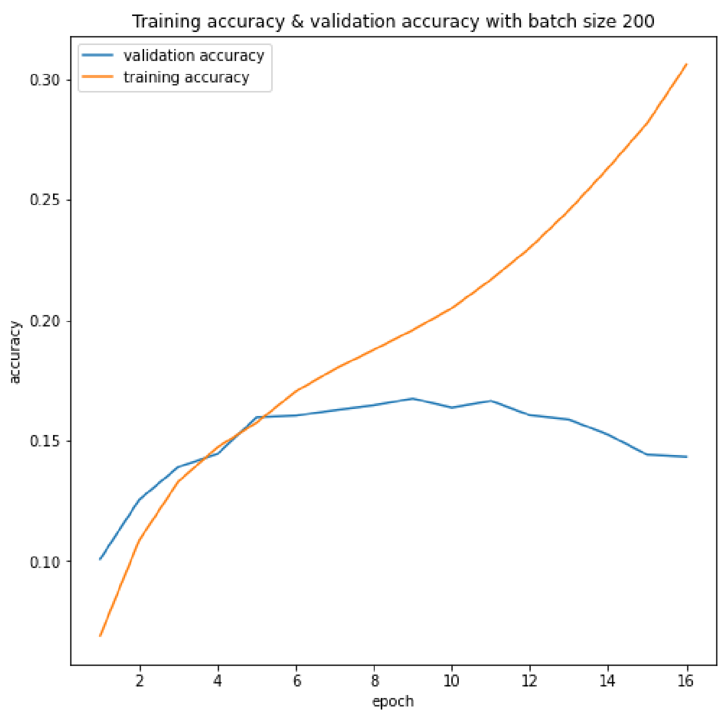
```
#load the best model which gave us the best validation accuracy to predict the test data.
fifth_model = load_model('fifth_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(fifth_model, tokenizer, test_lines))
```

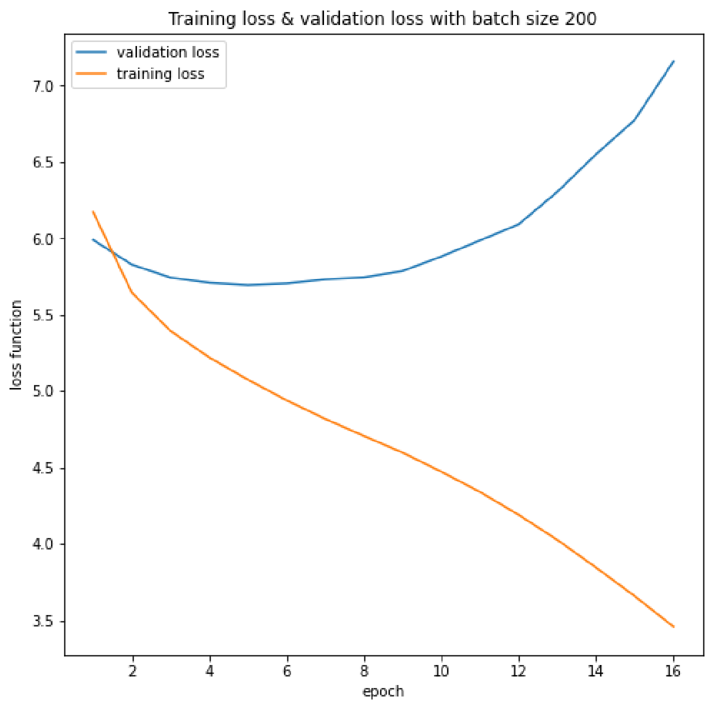
The output will be:

Sixth model results

Best training loss: 3.4577
Best validation loss: 5.6933
Best training accuracy: 0.3061
Best validation accuracy: 0.1713

Performance curve and convergence curve for the sixth model





Results intuition: Actually, merging bidirectional GRU with bidirectional LSTM give us the best intuition which the best method to solve this problem is to use complex bidirectional models to save the patterns from future to past and from past to future and this model was the best model compared to all previous models.

Start testing the sixth model

```
#load the best model which gave us the best validation accuracy to predict the test data.
last_model = load_model('last_model.h5')
print('input data: '+test_lines[:-3]+' .....')
print('\n')
print(predict_next_word_proba(last_model, tokenizer, test_lines))
```

The output will be: Correct word is **[as]**

the predict next word will be one of these three words **(the higher probability the higher chance to be the next word)**

the predicted next word is **[and]** with the largest probability **0.07900000363588333**
the next word could be **[to]** with the second largest probability **0.05299999937415123**
and it could be **[that]** with the third largest probability **0.05000000074505806**

All models' results comparison

	Training loss	Validation loss	Training accuracy	Validation accuracy
First model	4.4681	5.8874	17.91%	16.04%
Second model	6.0692	6.3443	5.86%	6.634%
Third model	4.1857	5.7769	22.02%	16.159%
Fourth model	3.4581	5.6837	28.03%	16.497%
Fifth model	3.0848	5.6880	35.09%	17.09%
Last model	3.4577	5.6933	30.61%	17.31%

Why these results and outputs has low accuracy?

I found that we don't have to have large validation or training accuracy to tell if the model is good or not, because we don't have one word to be the only word to come after specific sentence.

Ex: we have the test sentence that we used to test our models and the correct value was **'as'** and it was the predicted word, but it doesn't have to be **'as'**, it can be **'and'** or **'to'** like some models decided as long as it does not conflict with the English grammars.

Ex 2: test data: deep neural ….,

The predicted word in this example could be **'models'** or **'networks'** so it doesn't have to be one specific value.

And that what gave us low training and categorical accuracy.

Conclusion

Can we increase the accuracy of our models and predicted the exact correct word?

Yes, we can do that by following some steps:

1. Increase the number of sequences, The more data you train, the better results and accuracy you will get, but we'll need more computational resources.

2. Use more complex recurrent models and take in consideration the overfitting problem.

3. Use pre-trained models like GPT-3.

REF

https://towardsdatascience.com/the-beginners-guide-to-language-models-aa47165b57f9

https://towardsdatascience.com/next-word-prediction-with-nlp-and-deep-learning-48b9fe0a17bf