CISC 867 Project 3:

Building a Recurrent Language Model

Using a Neural Network

Prepared By:

[Esraa Elsayed] **ID: 21ESMS**

[Fatma Bleity] **ID: 21FKMA**

[Gehan Essa] **ID: 21GHHE**

Submitted to:

[Dr. Hazem Abbas]

Table of content

1. Introduction 6

1.1. Objective of the project 6

1.2. The Dataset used 6

2. The Libraries 7

3. Data Preparation 9

3.1. Load and Read Text Data 9

3.2. Describe the data 9

3.3. Clean and Preprocess the data.....10

4. Train the Language model 12

4.1. Load Training data (load sequences).....13

4.2. Encode Training data (encode sequence) 13

4.3. Separate Sequences into inputs and outputs 13

4.4. Define size of vocabulary and sequence size for the model.....14

4.5. Encode The outputs words 15

5. Build the Language Model 15

5.1. Build the Model 15

5.2. Compile & Fit Model 16

6. Save the Language Model..... 18

7. Use The Language Model 18

7.1. Load Data 18

7.2. Load Model 19

7.3. Generate Text 19

8. Trial_2 21

9. Trial_3 24

10. Trial_4 26

11. Trial_5 29

12. Results of all Trials 1, 2, 3, 4, 5..... 31

13. Conclusion 32

14. References 33

Table of figures:

Figure 1. Import libraries 7

Figure 2. Reading the data 9

Figure 3. Clean the text data 10

Figure 4. Print tokens after cleaning 10

Figure 5. Split text into sequence (each on consist of 50 words) 11

Figure 6. Save sequence data 12

Figure 7. Load training data (sequences).....13

Figure 8. Encode the input data 13

Figure 9. Separate data 13

Figure 10. Size of vocab 14

Figure 11. Length of sequence 14

Figure 12. Encode output words 15

Figure 13. Build the language model 16

Figures 14 Model summary 16

Figures 15 Compile and Fit the model 17

Figure 16. Result of fitting model for trial_1.....17

Figure 17. Save the model 18

Figures 18 Load data for generate the new text trials 18

Figure 19. Sequence size for the new text 19

Figure 20. Load our model to generate the new text 19

Figure 21. Select random line 19

Figure 22. Function to generate the new text.....20

Figure 23. Print the new text for trial_1.....21

Figure 24. Build the model for trial_2.....21

Figure 25. Compile and Fit model for Trial_2 22

Figure 26. Result of fitting model for trial_2 22

Figure 27. Print the new text for trial_2 23

Figure 28. Build the model for trial_3 24

Figure 29. Compile and Fit model for Trial_3 25

Figure 30. Result of fitting model for trial_3 25

Figure 31. Print the new text for trial_3 25

Figure 32. Build the model for trial_4 26

Figure 33. Compile and Fit model for Trial_4 27

Figure 34. Result of fitting model for trial_4 27

Figure 35. Print the new text for trial_4 28

Figure 36. Build the model for trial_5 29

Figure 37. Compile and Fit model for Trial_5 30

Figure 38. Result of fitting model for trial_5 30

Figure 39. Print the new text for trial_5 31

1. Introduction

1.1. Objective of the project

Build language model that can predict the probability of the next word in the sequence, based on the words already observed in the sequence. And use the learned language model to generate new text with similar statistical properties as the source text.

1.2. The Dataset Used

Use the Republic by Plato as the source text can be downloaded from

https://www.gutenberg.org/cache/epub/1497/pg1497.txt

2.The libraries

We used some important libraries in python to help us for training a RNN neural network that shown in the figure below.

```
import re
import nltk
from nltk.tokenize import word_tokenize
from keras.preprocessing.text import Tokenizer
from numpy import array
from keras.utils.np_utils import to_categorical
from keras.models import Sequential
from keras.layers import Embedding, LSTM, GRU, Dense
from pickle import dump
from keras.models import load_model
from random import randint
from keras.preprocessing.sequence import pad_sequences
import numpy as np
```

Figure 1. Import Libraries

- **re**

  The re module provides a set of powerful regular expression facilities, which allows you to quickly check whether a given string matches a given pattern (using the match function), or contains such a pattern (using the search function). [1]

- **NLTK**

  The Natural Language Toolkit (NLTK) is a platform used for building Python programs that work with human language data for applying in statistical natural language processing (NLP). It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. [2]

- **Numpy**

  The name “Numpy” stands for “Numerical Python”. It is the commonly used library. It is a popular machine learning library that supports large matrices and multi- dimensional data. It consists of in-built mathematical

functions for easy computations. Even libraries like TensorFlow use Numpy internally to perform several operations on tensors. Array Interface is one of the key features of this library.

- **Keras**

It is a powerful and easy-to-use free open source Python library for developing and evaluating deep learning models.

- **pickle**

Python pickle module is used for serializing and de-serializing a Python object structure. Any object in Python can be pickled so that it can be saved on disk. What pickle does is that it “serializes” the object first before writing it to file. Pickling is a way to convert a python object (list, dict, etc.). [3]

- **`random`**

Python Random module is an in-built module of Python which is used to generate random numbers. These are pseudo-random numbers means these are not truly random. This module can be used to perform random actions such as generating random numbers, print random a value for a list or string, etc. [4]

3. Data Preparation

3.1. Load and Read Text data

```
with open('republic_text.txt') as file:
    contents = file.read()
    print(contents)
```

```
Or a more general division into two parts may be adopted; the first
(Books I - IV) containing the description of a State framed generally in
accordance with Hellenic notions of religion and morality, while in the
second (Books V - X) the Hellenic State is transformed into an
ideal kingdom of philosophy, of which all other governments are the
perversions. These two points of view are really opposed, and the
opposition is only veiled by the genius of Plato. The Republic, like
the Phaedrus (see Introduction to Phaedrus), is an imperfect whole; the
higher light of philosophy breaks through the regularity of the
Hellenic temple, which at last fades away into the heavens. Whether this
imperfection of structure arises from an enlargement of the plan;
or from the imperfect reconcilement in the writer's own mind of the
struggling elements of thought which are now first brought together
by him; or, perhaps, from the composition of the work at different
times--are questions, like the similar question about the Iliad and
the Odyssey, which are worth asking, but which cannot have a distinct
answer. In the age of Plato there was no regular mode of publication,
and an author would have the less scruple in altering or adding to a
work which was known only to a few of his friends. There is no absurdity
in supposing that he may have laid his labours aside for a time, or
turned from one work to another; and such interruptions would be more
likely to occur in the case of a long than of a short writing. In all
attempts to determine the chronological order of the Platonic writings
on internal evidence, this uncertainty about any single Dialogue being
composed at one time is a disturbing element, which must be admitted
to affect longer works, such as the Republic and the Laws, more than
shorter ones. But, on the other hand, the seeming discrepancies of
the Republic may only arise out of the discordant elements which the
philosopher has attempted to unite in a single whole, perhaps without
being himself able to recognise the inconsistency which is obvious to
us. For there is a judgment of after ages which few great writers have
```

Figure 2. Reading the data

We use ‘with open ‘function to read the dataset file.

3.2. Describe the data

Here we use The Republic by Plato as the source text.

The Republic is the classical Greek philosopher Plato’s most famous work.
It is structured as a dialog (e.g. conversation) on the topic of order and justice within a city state. Can be downloaded from https://www.gutenberg.org/cache/epub/1497/pg1497.txt

3.3. Clean the data

According to **Garbage in Garbage out** phrase:

Raw text data contain unwanted or unimportant text due to which our results might not give efficient accuracy, and might make it hard to understand and analyze. So, proper pre-processing must be done on raw data.

The raw text must be converted into a sequence of tokens or words which can be used to train the model. Here we will apply some of text preprocessing techniques such as

1- Replace '–' with a white space so we can split words better.
2- Split words based on white space.
3- Removing all non-essential letters (Numbers and Punctuation).
4- Convert all characters to lowercase

```
nltk.download('punkt')
nltk.download('wordnet')

# function to preprocesse the summary text
def clean_text(contents):
    # replace '--' with a space ' '
    contents = contents.replace('--', ' ')

    # remove any special characters and punctuaton
    contents=re.sub('[^a-zA-Z]',' ',contents)

    # convert all words to lowercase
    contents=str(contents).lower()

    # tokenize the sentence
    contents=word_tokenize(contents)

    return contents  # return our text
```

Figure 3. Clean the text data

```
# clean document
tokens = clean_text(contents)
print(tokens[:10]) # print list of tokens that look cleaner than the raw text
print('Total number of Tokens: %d' % len(tokens)) # find out nuber of words in our text after applying preprocessing
print('Total number of Unique Tokens: %d' % len(set(tokens))) # find out nuber of vocabulary (unique words) in our text
```

```
['the', 'project', 'gutenberg', 'ebook', 'of', 'the', 'republic', 'by', 'plato', 'this']
Total number of Tokens: 217708
Total number of Unique Tokens: 10243
```

Figure 4. Print tokens after cleaning

To develop a language model from this text. We need to organize the text into sequences each of 50 input words and 1 output.

This can be accomplished by iterating over the list of tokens from token 51 on wards and recording the previous 50 tokens as a sequence, then continuing the procedure until the list of tokens.

```
# define length of our sequence
length_of_seq = 50+1
sequences = list()
for i in range(length_of_seq, len(tokens)):
    seq = tokens[i-length_of_seq:i]

    # To save these tokens as a lines, we'll convert them into space-separated strings
    line = ' '.join(seq)
    sequences.append(line)

print (sequences[:1]) # TO make sure that oue sequence is 50 word
print("----------------------------------------------")
print('Total number of Sequences: %d' % len(sequences))
```

```
['the project gutenberg ebook of the republic by plato this ebook is for the use of anyone anywhere at no cost and with almost no restrictions
----------------------------------------------
Total number of Sequences: 217657
```

Figure 5. Split text into sequence (each on consist of 50 words)

Here we can see that we will have exactly 217657 training patterns to fit our model.

Save the sequences to a separate file and load them later.

```
# function for saving lines of text to a file
def save_seq(lines, filename):
    data = '\n'.join(lines)
    f = open(filename, 'w')
    f.write(data)
    f.close()

# Call save_seq function and save our training sequences to the file 'republic_sequences.txt'
sequences_file = 'republic_sequences.txt'
save_seq(sequences, sequences_file)
```

Figure 6. Save sequence data

Now we have training data stored in the file ‘republic_sequences.txt‘

In ‘republic_sequences.txt‘file each line consist of 50 words

So, let’s go on to fitting a language model to this data.

4.Train the Language model

Train a statistical language model using a recurrent architecture from the prepared data that

a. Uses a distributed representation for words so that different words with similar meanings will have a similar representation.
b. Learns the representation at the same time as learning the model.
c. Learns to predict the probability for the next word using the context of the last 100 words

4.1. Load Training data (load sequences)

```
with open('republic_sequences.txt') as f:
    contents = f.read()
    # split data into separate training sequences by splitting based on new lines.
    new_lines = contents.split('\n')
```

Figure 7. Load training data (sequences)

4.2. Encode Training data (encode sequence)

According to the word embedding layer the input sequences should be made up of integers.

We can encode our input sequences by mapping each word to a unique number using Tokenizer class in the Keras.

```
tokenizer = Tokenizer()
# fit Tokenizer to encode all of the training sequences, converting each sequence from a list of words to a list of integers.
tokenizer.fit_on_texts(new_lines)
sequences = tokenizer.texts_to_sequences(new_lines)
```

Figure 8. Encode the input data

4.3. Separate Sequences into inputs and outputs

```
# separate sequences into input and output
sequences = array(sequences)
X = sequences[:,:-1]
y = sequences[:,-1]
```

Figure 9. Separate data

4.4. Define size of vocabulary and sequence size for the model

We need to know the size of the vocabulary for defining the embedding layer later. We can determine the vocabulary by calculating the size of the mapping dictionary.

The word index dictionary field on the Tokenizer object allows us to obtain the mapping of words to numbers.

```
# define size of vocabulary for using in embedding layer in the model.
size_of_vacab = len(tokenizer.word_index) + 1
size_of_vacab
```

```
10244
```

Figure 10. Size of vocab

Using the second dimension (number of columns) of the input data's structure is a decent generic method to indicate that. As a result, if the length of sequences changes when preparing data, you won't have to update this data loading function because it's general.

```
# The length of input sequences must be specified to the Embedding layer (50 word in each sequence)
seq_length = X.shape[1]
seq_length
```

```
50
```

Figure 11. Length of sequence

4.5. Encode the outputs words

After separating, we need to one hot encode the output word using to_categorical() that can be used to one hot encode the output words for each input-output sequence pair. This means converting it from an integer to a vector of 0 values, one for each word in the vocabulary, with a 1 to indicate the specific word at the index of the words integer value.

This is so that the model learns to predict the probability distribution for the next word and the ground truth from which to learn from is 0 for all words except the actual word that comes next.

```
from keras.utils.np_utils import to_categorical
y = to_categorical(y, num_classes=size_of_vacab)
```

Figure 12. Encode output words

5.Build the Language Model

5.1 Build the model (for trial_1)

We will use an Embedding Layer to learn the representation of words, and a Long ShortTerm Memory (LSTM) recurrent neural network to l earn to predict words based on their context.

A Dense fully connected layer connects to the LSTM hidden layers to interpret the features extracted from the sequence.

The output layer predicts the next word as a single vector the size of the vocabulary with a probability for each word in the vocabulary.

Here we build the model using 1 Embedding layer, 2 (LSTM) layer with 200 units in each layer and 2 Dense layers, one with 2000 units and the other is 1500 units.

```
# define model

model1 = Sequential()
model1.add(Embedding(size_of_vacab,50,input_length=seq_length))

model1.add(LSTM(200, return_sequences=True))
model1.add(LSTM(200))

model1.add(Dense(2000, activation='relu'))
model1.add(Dense(1500, activation='relu'))

model1.add(Dense(size_of_vacab, activation='softmax'))
print(model1.summary())
```

Figure 13. Build the language model

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 50, 50)            512200
_________________________________________________________________
lstm (LSTM)                  (None, 50, 200)           200800
_________________________________________________________________
lstm_1 (LSTM)                (None, 200)               320800
_________________________________________________________________
dense (Dense)                (None, 2000)              402000
_________________________________________________________________
dense_1 (Dense)              (None, 1500)              3001500
_________________________________________________________________
dense_2 (Dense)              (None, 10244)             15376244
=================================================================
Total params: 19,813,544
Trainable params: 19,813,544
Non-trainable params: 0
_________________________________________________________________
```

Figure 14. Model summary

5.2 Compile & Fit Model

The model is compiled specifying the categorical cross entropy loss because, the model is learning a multi-class classification and this is the suitable loss function for this type of problem.

Use The efficient Adam implementation to mini-batch gradient descent and accuracy is evaluated of the model.

```
# compile model
model1.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model1.fit(X, y, batch_size=128, epochs=60)
```

Figure 15. Compile and Fit the model

```
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2810 - accuracy: 0.9155
Epoch 44/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2630 - accuracy: 0.9209
Epoch 45/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2500 - accuracy: 0.9255
Epoch 46/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2516 - accuracy: 0.9243
Epoch 47/60
1701/1701 [==============================] - 39s 23ms/step - loss: 0.2318 - accuracy: 0.9305
Epoch 48/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.2234 - accuracy: 0.9324
Epoch 49/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.2200 - accuracy: 0.9342
Epoch 50/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2098 - accuracy: 0.9368
Epoch 51/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2017 - accuracy: 0.9393
Epoch 52/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.2009 - accuracy: 0.9391
Epoch 53/60
1701/1701 [==============================] - 39s 23ms/step - loss: 0.1898 - accuracy: 0.9420
Epoch 54/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.1885 - accuracy: 0.9436
Epoch 55/60
1701/1701 [==============================] - 39s 23ms/step - loss: 0.1791 - accuracy: 0.9453
Epoch 56/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.1830 - accuracy: 0.9447
Epoch 57/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.1670 - accuracy: 0.9495
Epoch 58/60
1701/1701 [==============================] - 38s 23ms/step - loss: 0.1687 - accuracy: 0.9491
Epoch 59/60
1701/1701 [==============================] - 39s 23ms/step - loss: 0.1718 - accuracy: 0.9484
Epoch 60/60
1701/1701 [==============================] - 38s 22ms/step - loss: 0.1569 - accuracy: 0.9521
<keras.callbacks.History at 0x7f83b402e5d0>
```

Figure 16. Result of fitting model for trial_1

6.Save the Language Model

The model is saved to the file'languagemodel.h' in the current working directory using the Keras model API.

We'll need the mapping of words to integers when we load the model to make predictions.

This is stored in the Tokenizer object, which we can also save using Pickle.

Here we saved the model to use it later for generate the text.

```
# save the model to file to use when generate the text
model1.save('language model1.h')
# save the tokenizer
dump(tokenizer, open('tokenizer1.pkl', 'wb'))
```

```
2022-04-05 17:48:31.248998: W tensorflow/python/util/util.cc:348] Sets are not currently considered sequences
```

Figure17. Save the model

7. Use the Language Model

Here we can use the model to generate new sequences of text that have the same statistical properties.

7.1. Load Data

We require the text so that we can choose a source sequence to feed into the model in order to generate a new text sequence

```
with open('republic_sequences.txt') as f:
    contents = f.read()
    lines = contents.split('\n')
```

Figure 18. Load data for generate the new text

We will need to specify the expected length of input. We can determine this from the input sequences by calculating the length of one line of the loaded data and subtracting 1 for the expected output word that is also on the same line.

```
seq_length = len(lines[0].split()) - 1
seq_length
```

```
50
```

Figure 19. Sequence size for the new text

7.2. Load Model

```
model1 = load_model('language model1.h')
```

```
# load the tokenizer
dump(tokenizer, open('tokenizer1.pkl', 'wb'))
```

Figure 20. Load our model to generate the new text

7.3. Generate the text

First, we will select a random line of text from the input text for generating the text.

```
# select the random line of the text data
_text = lines[randint(0,len(lines))]
print(_text + '\n')
```

```
or intruder very true suppose i said the study of philosophy to take the place of gymnastics and to be continued diligently and earnestly and exclusively for twice the number of years
```

Figure 21. Select random line

Second, the _text must be encoded to integers using tokenizer tha t we used
when training the model.

Third, the input sequence is going to get too long. So we need to truncate it to the desired length after the input sequence has been encoded to integers.
Keras provides the pad_sequences() function.

Forth, the model can predict the next word directly by calling model predict and np.argmax that will return the index of the word with the highest probability.

Fifth, map predicted word index to word.

```
# function to generate a sequence from a language model
def generate_new_seq(model, tokenizer, seq_length, _text, n_words):
    result = list()
    in_text = _text
    # generate a fixed number of words
    for _ in range(n_words):

        # the _text must be encoded to integers using tokenizer that we used when training the model.
        encoded = tokenizer.texts_to_sequences([in_text])[0]

        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')

        # here The model can predict the next word directly by calling model.predict and np.argmax that will return the index of the word with the highest probability.
        yhat = model.predict(encoded, verbose=0)
        yhat= np.argmax(yhat, axis=1)

        # Fifth, map predicted word index to word
        #To find the related word, we can look up the index in the Tokenizers mapping.
        out_word = ''
        for word, index in tokenizer.word_index.items():
            if index == yhat:
                out_word = word
                break

            # append to input
        in_text += ' ' + out_word
        result.append(out_word)
    return ' '.join(result)
```

Figure 22. Function to generate the new text

```
# generate new text
generated = generate_new_seq(model1, tokenizer, seq_length, _text, 100)
print(generated)
```

```
say five years i replied at the end of the time they must be sent down again into the den and compelled to hold any military or
other office which young men are qualified to hold in this way they will call their own advantage or the good the human creatur
e would be as far as he can be into one another the only original and in this other sphere we acknowledge that we could not sup
pose that a man is profited by persuasion and this he is afraid to be a debt which he had seen in his own
```

Figure 23. Print the new text for trial_1

8.Trial_2

As we made in the trial 1, we will go through the same steps except we will change our model.

So, in this trial_2, we will try build the model using 1 Embedding layer, 1 (LSTM) layer with 265 units in layer and 3 Dense layers, one with 2500 units and the other is 2000 units to see their effect on the predicted sequence of words.

```
# define model
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense

model2 = Sequential()
model2.add(Embedding(size_of_vacab, 50, input_length=seq_length))
model2.add(LSTM(256))
model2.add(Dense(2500, activation='relu'))
model2.add(Dense(2000, activation='relu'))
model2.add(Dense(size_of_vacab, activation='softmax'))
print(model2.summary())
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 50, 50)            512200
_________________________________________________________________
lstm_2 (LSTM)                (None, 256)               314368
_________________________________________________________________
dense_3 (Dense)              (None, 2500)              642500
_________________________________________________________________
dense_4 (Dense)              (None, 2000)              5002000
_________________________________________________________________
dense_5 (Dense)              (None, 10244)             20498244
=================================================================
Total params: 26,969,312
Trainable params: 26,969,312
Non-trainable params: 0
_________________________________________________________________
```

Figure 24. Build the model for trial_2

```
# compile model
model2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model2.fit(X, y, batch_size=128, epochs=60)
```

Figure 25. Compile and Fit model for Trial_2

```
Epoch 43/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1316 - accuracy: 0.9600
Epoch 44/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1228 - accuracy: 0.9626
Epoch 45/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1233 - accuracy: 0.9625
Epoch 46/60
1701/1701 [==============================] - 31s 19ms/step - loss: 0.1202 - accuracy: 0.9641
Epoch 47/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1149 - accuracy: 0.9655
Epoch 48/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1156 - accuracy: 0.9650
Epoch 49/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1131 - accuracy: 0.9659
Epoch 50/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1094 - accuracy: 0.9670
Epoch 51/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1050 - accuracy: 0.9683
Epoch 52/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1027 - accuracy: 0.9687
Epoch 53/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1049 - accuracy: 0.9682
Epoch 54/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.1008 - accuracy: 0.9698
Epoch 55/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0957 - accuracy: 0.9710
Epoch 56/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0991 - accuracy: 0.9700
Epoch 57/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0950 - accuracy: 0.9710
Epoch 58/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0931 - accuracy: 0.9721
Epoch 59/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0954 - accuracy: 0.9715
Epoch 60/60
1701/1701 [==============================] - 31s 18ms/step - loss: 0.0893 - accuracy: 0.9730
```

Figure 26. Result of fitting model for trial_2

```
# generate new text
generated = generate_new_seq(model2, tokenizer, seq_length, _text, 100)
print(generated)
```

```
last for they have nothing to say in this new game of which words are the counters and yet all the time they are in the right t
he observation is suggested to me by what is now occurring for any one of us might say that although in words he is not able to
meet you at each step of the argument he sees as a fact that the votaries of philosophy when they carry on the study not only i
n youth as a part of education but as the pursuit of their maturer years most of them become strange
```

Figure 27. Print the new text for trial_2

Here we found that the model's performance became better when using only one (LSTM) layer with 256 units and 2 Dense layers with 2500 units and the other is 2000.

9. Trial_3

As we made in the trial _1, we will go through the same steps except we will change our model.

So, in this trial_3, we will try build the model using 1 Embedding layer, 1 (GRU) layer with 128 units in layer and 5 Dense layers, one with 3000 units and the other is 2000 units,1000 units, 500 units, 200 units to see their effect on the predicted sequence of words.

```
# define model
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding, GRU

model3 = Sequential()
model3.add(Embedding(size_of_vacab, 50, input_length=seq_length))

model3.add(GRU(128))

model3.add(Dense(3000, activation='relu'))
model3.add(Dense(2000, activation='relu'))
model3.add(Dense(1000, activation='relu'))
model3.add(Dense(500, activation='relu'))
model3.add(Dense(200, activation='relu'))

model3.add(Dense(size_of_vacab, activation='softmax'))
print(model3.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 50, 50)            512200
_________________________________________________________________
gru (GRU)                    (None, 128)               69120
_________________________________________________________________
dense_6 (Dense)              (None, 3000)              387000
_________________________________________________________________
dense_7 (Dense)              (None, 2000)              6002000
_________________________________________________________________
dense_8 (Dense)              (None, 1000)              2001000
_________________________________________________________________
dense_9 (Dense)              (None, 500)               500500
_________________________________________________________________
```

Figure 28. Build the model for trial_3

```
# compile model
model3.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model3.fit(X, y, batch_size=128, epochs=60)
```

Figure 29. Compile and Fit model for Trial_3

```
1701/1701 [==============================] - 23s 14ms/step - loss: 1.9486 - accuracy: 0.5462
Epoch 44/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.9265 - accuracy: 0.5515
Epoch 45/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.8929 - accuracy: 0.5582
Epoch 46/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.8707 - accuracy: 0.5612
Epoch 47/60
1701/1701 [==============================] - 23s 14ms/step - loss: 1.8415 - accuracy: 0.5682
Epoch 48/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.7976 - accuracy: 0.5767
Epoch 49/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.7846 - accuracy: 0.5796
Epoch 50/60
1701/1701 [==============================] - 23s 14ms/step - loss: 1.7615 - accuracy: 0.5851
Epoch 51/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.7320 - accuracy: 0.5919
Epoch 52/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.7109 - accuracy: 0.5954
Epoch 53/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.6826 - accuracy: 0.6022
Epoch 54/60
1701/1701 [==============================] - 23s 14ms/step - loss: 1.6674 - accuracy: 0.6042
Epoch 55/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.6311 - accuracy: 0.6123
Epoch 56/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.6273 - accuracy: 0.6131
Epoch 57/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.6061 - accuracy: 0.6181
Epoch 58/60
1701/1701 [==============================] - 23s 14ms/step - loss: 1.5893 - accuracy: 0.6207
Epoch 59/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.5687 - accuracy: 0.6251
Epoch 60/60
1701/1701 [==============================] - 24s 14ms/step - loss: 1.5435 - accuracy: 0.6302
<keras.callbacks.History at 0x7f83b3524750>
```

Figure 30. Result of fitting model for trial_3

```
# generate new text
generated = generate_new_seq(model3, tokenizer, seq_length, _text, 100)
print(generated)
```

```
are deserving of progress friend socrates the dorian life is the best and the true method of the body only and in which a certa
in character is unknown by plato and then in a state which is ordered states neither can these be found in the city you say tha
t the philosopher was mistaken yes and there is an endless purgation of the body very true then is the name which you have been
describing by all the poets are both hard and educators of the sake of appearances is great in nursing up in institutions and r
emaining only a
```

Figure 31. Print the new text for trial_3

Here we found that the model's performance isn't better when using only one (GRU) layer with 128 units and 4 Dense layers with 3000 units , 2000 units,1000 units, 500 units and 200 units.

The model's accuracy is about 63%

10. Trial_4

As we made in the trial _1, we will go through the same steps except we will change our model.

So, in this trial_4, we will try build the model using 1 Embedding layer, 2 (GRU) layer with 256 units and 128 units in layer and 3 Dense layers, one with 2500 units and the other is 2000 units,1000 units to see their effect on the predicted sequence of words.

```
# define model
from keras.models import Sequential
from keras.layers import Dense, LSTM, Embedding, GRU

model4 = Sequential()
model4.add(Embedding(size_of_vacab, 50, input_length=seq_length))
model4.add(GRU(265, return_sequences=True))
model4.add(GRU(128))

model4.add(Dense(2500, activation='relu'))
model4.add(Dense(2000, activation='relu'))
model4.add(Dense(1000, activation='relu'))

model4.add(Dense(size_of_vacab, activation='softmax'))
print(model4.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 50, 50)            512200
_________________________________________________________________
gru_1 (GRU)                  (None, 50, 265)           252015
_________________________________________________________________
gru_2 (GRU)                  (None, 128)               151680
_________________________________________________________________
dense_12 (Dense)             (None, 2500)              322500
_________________________________________________________________
dense_13 (Dense)             (None, 2000)              5002000
_________________________________________________________________
dense_14 (Dense)             (None, 1000)              2001000
_________________________________________________________________
dense_15 (Dense)             (None, 10244)             10254244
=================================================================
```

Figure 32. Build the model for trial_4

```
# compile model
model4.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model4.fit(X, y, batch_size=128, epochs=60)
```

Figure 33. Compile and Fit model for Trial_4

```
Epoch 44/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.5743 - accuracy: 0.8403
Epoch 45/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.5529 - accuracy: 0.8468
Epoch 46/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.5462 - accuracy: 0.8477
Epoch 47/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.5295 - accuracy: 0.8527
Epoch 48/60
1701/1701 [==============================] - 42s 24ms/step - loss: 0.5188 - accuracy: 0.8564
Epoch 49/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4996 - accuracy: 0.8613
Epoch 50/60
1701/1701 [==============================] - 41s 24ms/step - loss: 0.4964 - accuracy: 0.8623
Epoch 51/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4785 - accuracy: 0.8657
Epoch 52/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4780 - accuracy: 0.8683
Epoch 53/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4647 - accuracy: 0.8714
Epoch 54/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4617 - accuracy: 0.8728
Epoch 55/60
1701/1701 [==============================] - 43s 25ms/step - loss: 0.4498 - accuracy: 0.8759
Epoch 56/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4343 - accuracy: 0.8798
Epoch 57/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4346 - accuracy: 0.8804
Epoch 58/60
1701/1701 [==============================] - 43s 25ms/step - loss: 0.4245 - accuracy: 0.8825
Epoch 59/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4195 - accuracy: 0.8844
Epoch 60/60
1701/1701 [==============================] - 42s 25ms/step - loss: 0.4012 - accuracy: 0.8896
<keras.callbacks.History at 0x7f80d4e12bd0>
```

Figure 34. Result of fitting model for trial_4

```
# generate new text
generated = generate_new_seq(model4, tokenizer, seq_length, _text, 100)
print(generated)
```

```
discuss how many there are some elementary artist of as thrasymachus may judge only each of them not in modern times we sometim
es need to imply that he is like a man who tells us that he is a good man who is the greatest one of them that is the inference
and when you want to keep a pruning hook safe then justice is useful to the individual and to the state but when you want to us
e it then the art of payment begins by his right men and not to lose their plan in the case he
```

Figure 35. Print the new text for trial_4

Here we found that the model's performance became better when using 2 (GRU) layer with 265 units 128 units in layers and 3 Dense layers with 3000 units , 2000 units,1000 units than using only one (GRU) layer with 128 units and 4 Dense layers with 3000 units , 2000 units,1000 units, 500 units and 200 units.

The model's accuracy is about 88%

11. Trial_5

As we made in the trial _1, we will go through the same steps except we will change our model.

So, in this trial_5, we will try build the model using 1 Embedding layer, 1 (GRU) layer with 256 units and 1(LSTM) layer with 128 units in layer and 3 Dense layers, one with 2500 units and the other is 2000 units,1000 units to see their effect on the predicted sequence of words.

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

model5 = Sequential()
model5.add(Embedding(size_of_vacab, 50, input_length=seq_length))

model5.add(GRU(256, return_sequences=True))
model5.add(LSTM(256))

model5.add(Dense(2500, activation='relu'))
model5.add(Dense(2000, activation='relu'))
model5.add(Dense(1000, activation='relu'))

model5.add(Dense(size_of_vacab, activation='softmax'))
print(model5.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 50, 50)            512200
_________________________________________________________________
gru_3 (GRU)                  (None, 50, 256)           236544
_________________________________________________________________
lstm_3 (LSTM)                (None, 256)               525312
_________________________________________________________________
dense_16 (Dense)             (None, 2500)              642500
_________________________________________________________________
dense_17 (Dense)             (None, 2000)              5002000
_________________________________________________________________
dense_18 (Dense)             (None, 1000)              2001000
_________________________________________________________________
dense_19 (Dense)             (None, 10244)             10254244
=================================================================
```

Figure 36. Build the model for trial_5

```
# compile model
model5.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit model
model5.fit(X, y, batch_size=128, epochs=60)
```

Figure 37. Compile and Fit model for Trial_5

```
1701/1701 [==============================] - 40s 24ms/step - loss: 0.5065 - accuracy: 0.8567
Epoch 44/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4926 - accuracy: 0.8611
Epoch 45/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4789 - accuracy: 0.8648
Epoch 46/60
1701/1701 [==============================] - 41s 24ms/step - loss: 0.4603 - accuracy: 0.8694
Epoch 47/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4551 - accuracy: 0.8721
Epoch 48/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4409 - accuracy: 0.8758
Epoch 49/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4255 - accuracy: 0.8791
Epoch 50/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4204 - accuracy: 0.8821
Epoch 51/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.4004 - accuracy: 0.8869
Epoch 52/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3964 - accuracy: 0.8871
Epoch 53/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3830 - accuracy: 0.8910
Epoch 54/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3839 - accuracy: 0.8914
Epoch 55/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3786 - accuracy: 0.8945
Epoch 56/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3593 - accuracy: 0.8985
Epoch 57/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3551 - accuracy: 0.9000
Epoch 58/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3564 - accuracy: 0.8997
Epoch 59/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3430 - accuracy: 0.9027
Epoch 60/60
1701/1701 [==============================] - 40s 24ms/step - loss: 0.3431 - accuracy: 0.9036
<keras.callbacks.History at 0x7f7cd23d5890>
```

Figure 38. Result of fitting model for trial_5

```
# generate new text
generated = generate_new_seq(model5, tokenizer, seq_length, _text, 100)
print(generated)
```

```
one form of style may be hard against those which you were going to receive the ideal polity though imperfectly that the just m
an seeks to have a share in the government of the other or again about death the same person is acknowledging the size of the s
tate but no city must be treated or not by external effect at last too will be made the most quick while in his ideal state wil
l hereafter be called a man very true he said and i think that you mean for the art of justice when deprived of their subjects
```

Figure 39. Print the new text for trial_5

Here we found that the model's performance became better when using 1 (GRU) layer with 265 units and 1(LSTM) layer with 128 units in layer and 3 Dense layers, one with 2500 units and the other is 2000 units, 1000 units than using 2 (GRU) layer with 265 units 128 units in layers and 3 Dense layers with 3000 units, 2000 units, 1000 units. The model's accuracy is about 90%.

12. Results of all Trials 1, 2, 3, 4, and 5.

After running 5 models each one with using different layers (LSTM or GRU or LSTM & GRU) and Dense layers with different number of units in each layers as mentioned above, I found that the best model is the model which consist of 1 Embedding layer, 1 (LSTM) layer with 265 units in layer and 3 Dense layers, one with 2500 units and the other is 2000 units.

The model's accuracy is about 97%.

13. Conclusion

Recurrent Neural Networks are the best type of the networks to deal with text data. Here we learned more how to use it, and how to develop word based language model using a word embedding.

From our trials, we can feel what the difference between GRU, and LSTM layers, and they effect on the model, and help the model to learn more due to their memory gate, and the mechanism they work on from the memory gate, and forget gate.

Besides, cleaning the input text and make good preprocessing on the input, helps in making the model more powerful to get good results. As this prevents the model from taking garbage data, so it gives garbage outputs if we didn't make good cleaning and preprocessing from starting in the model.

In addition to, choosing good hyper-parameters to be used while training the model help the model to be better and learn more from the input features. As we tried many different parameters until we reached our optimal hyper-parameters which were suitable to our steps in the cleaning the text input, and the steps through building the model.

14. References

[1] https://www.oreilly.com/library/view/python-standard-library/0596000960/ch01s08.html

[2] https://www.techopedia.com/definition/30343/natural-language-toolkit nltk#:~:text=The%20Natural%20Language%20Toolkit%20(NLTK)%20is%20a%20platform%20used%20for,stemming%2C%20tagging%20and%20semantic%20reasoning.

[3] https://www.geeksforgeeks.org/understanding-python-pickling example/#:~:text=Python%20pickle%20module%20is%20used,list%2C%20dict%2C%20etc.)

[4] https://www.geeksforgeeks.org/python-random-module