AUTOMATIC TEXT GENERATION

Mini Project report submitted
in partial fulfilment of the requirement for the degree of

Bachelor of Technology

By

Bhaskar Borah (Roll No. 180610026054)
Biki Deka (Roll No. 180810026012)

Under the guidance
of

Ankur Jyoti Sarma
Manash Pratim Bhuyan

DEPARTMENT OF ELECTRONICS & TELECOMMUNICATION ENGINEERING
ASSAM ENGINEERING COLLEGE
JALUKBARI- 781013, GUWAHATI

July, 2021

ASSAM ENGINEERING COLLEGE, GUWAHATI

CERTIFICATE

This is to certify that the report entitled "Automatic Text Generation" submitted by Bhaskar Borah (Roll No. 180610026054) and Biki Deka (Roll No. 180810026012) of B. Tech 6th semester, Electronics & Telecommunication Engineering, is an authentic work carried out by them under my supervision and guidance.

To the best of my knowledge, the matter embodied in the report has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Signature of Supervisor(s)
Name(s)
Department(s)
Assam Engineering College
July, 2021

DECLARATION

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that we have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Bhaskar Borah

(Signature)
(Bhaskar Borah)

(Roll No. 180610026054)

Biki Deka

(Signature)
(Biki Deka)

(Roll No. 180810026012)

Date: _26/07/2021_

ACKNOWLEDGMENT

We would like to express our deepest gratitude to Electronics and telecommunication Department of Assam Engineering College for providing us the opportunity to undertake this mini project on the title "Automatic Text Generation".

We acknowledge with sincere thanks to our project guide Ankur Jyoti Sarma and Manash Pratim Bhuyan for excellent guidance and selfless efforts. Without their co-operative attitude, constant inspiration and dedicated at each and every stage of this project it would not be possible to make this project complete.

We would also like to express our gratitude towards the Head of the Department of Electronics and Telecommunication Engineering Department, Mr. Dinesh Shankar Pegu Sir.
We are also grateful to other teachers of the department for contributing their inputs in this project

Lastly, we would like to thank our parents and friends for all the support without which this project couldn't have been completed in the given limited time frame. Also a special acknowledgement goes to our colleague who helped us in completing the project by exchanging interesting ideas and sharing their experience.

BHASKAR BORAH (ROLL NO. 180610026054)
BIKI DEKA (ROLL NO. 180810026012)

ABSTRACT

Today, Computers have influenced the life of human beings to a great extent. To provide communication between computers and humans, natural language techniques have proven to be very efficient way to exchange the information with less personal requirement. As demand continues to grow for individual online content, automated text generation is to become increasingly important. Text generation techniques can be applied for improving language models, machine translation, summarization and captioning.

Text can be generated by using Hidden Markov Models and Markov Chains but it is difficult to generate whole sentence using them so we have used Recurrent Neural Networks (RNNs) and with its variant Long Short Term Memory (LSTM) to develop language model that can generate whole new text word by word automatically.

Research work presented in this thesis focused on generation of language model by training different RNNs. In our project training of simple RNN and LSTM is done on a selected dataset. Lastly, after this all the output texts are compared to conclude which network generates more realistic text. The variation of training loss with iterations is also examined.

LIST OF FIGURES

CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Introduction

In this chapter we will discuss the area of our work, how is the present-day scenario with regard to the work area, what is our motivation to do this project, significance of the possible end result, objective of our work, main work objective, importance of the end result and organization of project report.

The project is to build a text generation deep learning model. Text generation, as a basic natural language processing task, has many applications, such as dialogue robots, machine translation, paraphrasing and so on. With the rise of deep learning, different neural networks are introduced to generate text. For example, researchers use the recurrent neural network (RNN), long Short-Term Memory (LSTM)[1] to train the language model because of its capability to process sequential data.

Nowadays people are using different and very powerful models in NLP and in text generation area. One very popular thing is to use Deep generative models. Generally, the popular techniques for the generation of text in deep learning era are Variational Auto-Encoders (VAEs) and Generative Adversarial Networks (GANs).

1.2 Motivation

While the field of text generation or natural language generation remains largely unexplored, there are already quite a few areas where we find its applications. Most popular use cases of natural language generation have emerged in business intelligence interpretation. Numerous business intelligence tool providers are incorporating NLG into their tools to enable report generation based on analytics data. There have also where a natural language generation has been helping businesses automate the processes that require the creation of information in natural language. An example is a content creation AI tool that is capable of writing as naturally as humans by building on short, manually provided writing prompts. Such systems, while not capable of perfectly replicating human ways of expression, can definitely put together long and coherent sequences of sentences. So we can see text generation is very important in present days and this is what motivate us to do this project.

Our objective is to build a deep learning model that can generate text relevant to the input text provided to the model. To simply say model which will generate some lines or paragraph of text based on the provided line of texts.

CHAPTER 2

LITERATURE REVIEW

This chapter will include discussion on the project title, present day scenario or recent development in the field and also the background history of natural language generation.

2.1 Project Title

Text generation [1] is sub field of NLP. It has many uses. Our main objective is to generate story like text based on line of text provided. The project will use sequential model LSTM for this purpose.

2.2 Literature Review

There has very good developments in this field recently. The most popular techniques for the generation of text in deep learning era are Variational Auto-Encoders (VAEs) and Generative Adversarial Networks (GANs).

- Variational Auto-Encoders (VAEs): The power of most deep learning model depends on cleanly labelled data. Since most of the data is unlabelled or unstructured in nature. The popular deep learning models require large quantities of structured data for training. Labelling the unstructured data is very time consuming. One way to address this issue is to make use of unsupervised methods to train on data without labels. Variational Auto-Encoders is one of the powerful deep generative model that works on unlabelled data.

- Generative Adversarial Networks (GANs) : GANs are the popular deep learning algorithm that takes up an adversarial approach, dissimilar from the conventional neural network. GANs contain two models that are trained in an adversarial way. First, the generator generates the data samples and discriminator which classifies these data samples as real (training data) or fake (generated by generator) . The goal of the generator is to generate samples that are very close to the true data so that it can fool the discriminator and the goal of the discriminator is to classify accurately these two types of data samples. Since it's like a game-theoretic approach, where the objective function is represented as a part of the minimax function. The discriminator *D* tries to maximize the objective function, while the generator *G* tries to minimize the objective function.

Background theory:

To say something or write something we don't start our thinking every second. As I am writing this, I write each word based on my understanding of previous words. I don't throw everything away and start thinking from scratch again. Our thoughts have persistence. Traditional neural networks can't do this, and it seems like a major shortcoming. Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.

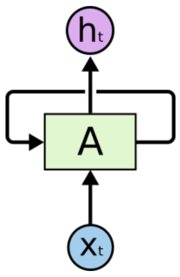


Fig 2.1 RNN cell

In the above diagram, a chunk of neural network, A, looks at some input x and outputs a value h. A loop allows information to be passed from one step of the network to the next.

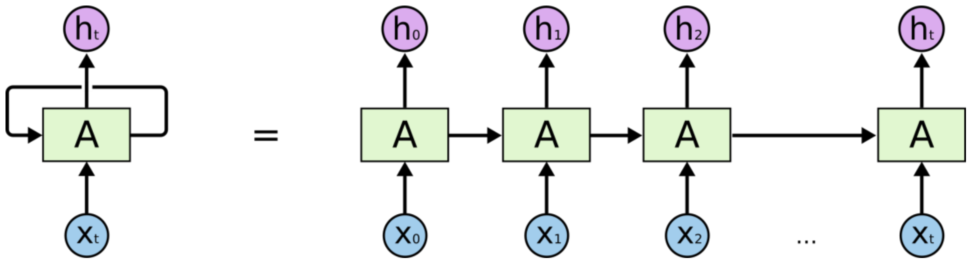


Fig 2.2 Unrolled RNN

RNNs can learn and remember the context of very recent data. But sometime we need more context, information from way long back. This problem is addressed by LSTM. Remembering information for long periods of time is practically their default behavior.

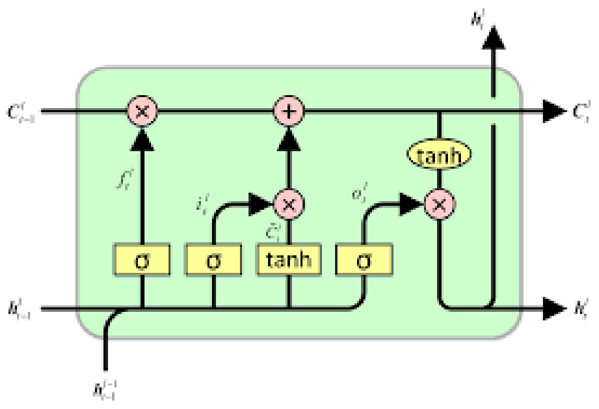


Fig 2.3: LSTM cell

The key to LSTMs is the cell state. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a point wise multiplication operation.

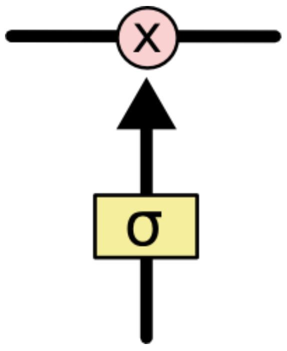


Fig 2.4: LSTM gate

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”. An LSTM has three of these gates, to protect and control the cell state.

Now to generate text which are relevant to the previous word or previous text this LSTM model will have to get trained on large amount of data. During the training the network will learn the representation of data. The network will be fed one word or character at a time, then it does some calculation and generates a hidden value (vector) and this hidden value is used to predict next character or word. The predicted output is then compared to real output and loss is generated which is backpropagated through the network to calibrate the parameter.

CHAPTER 3

METHODOLOGY

In this chapter we will be discussing the detailed methodology of the project, algorithms

3.1 Methodology

Here is the step-by-step procedure that we followed

3.1.1 Dataset

First of all, we needed to select the dataset and we took the book "The Yellow Wallpaper [3] by" Charlotte Perkins Gilman. This dataset is taken from "Project Gutenberg". Project Gutenberg is a library of over 60,000 free eBooks. This dataset was suitable because our goal was to generate story like text, so story books are preferred and the eBook was not huge so suitable according to the processing power we have.

3.1.2 Data pre-processing

- Every word is converted to lowercase so and not worry about capitalization.
- Tokenization of text. Tokenization is a common task in Natural Language Processing (NLP). Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or sub words. In our case it is words. Tokenization helps in building vocabulary and also removing unnecessary information.
- Remove punctuation, stop words using NLTK library in python. Stop words are frequently occurring words in English. These words don't carry much information. By eliminating them, model can focus on the important words.
- Define a vocabulary. The set of unique words used in the dataset or text corpus is referred to as the vocabulary. It is of two types one is word based and other is character based. In our case we defined a character-based vocabulary which consists of each unique character that is present in the processed data.
- A map of each vocabulary character to number is created.

- Divide the data into small sequences(X) used as inputs, output characters(Y). Map them to numbers according to the vocabulary and also shape those sequences as required by the neural network.
- Then batched are created where each batch contain some number of sequences.

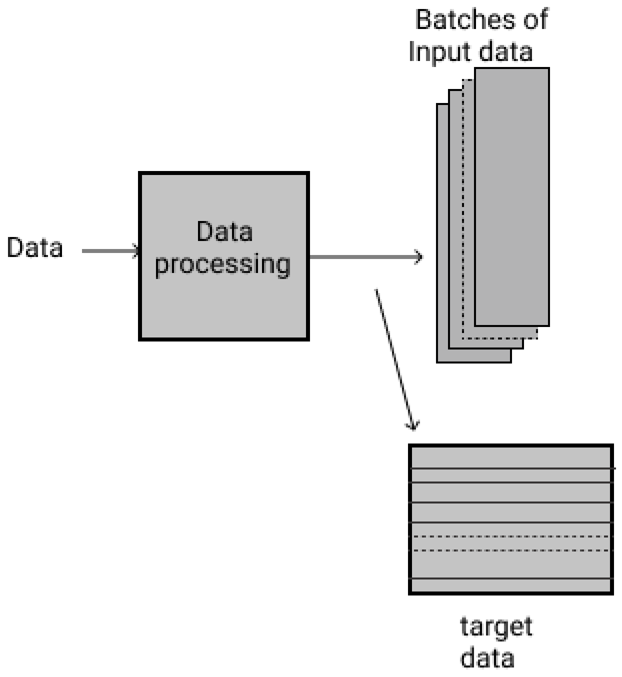
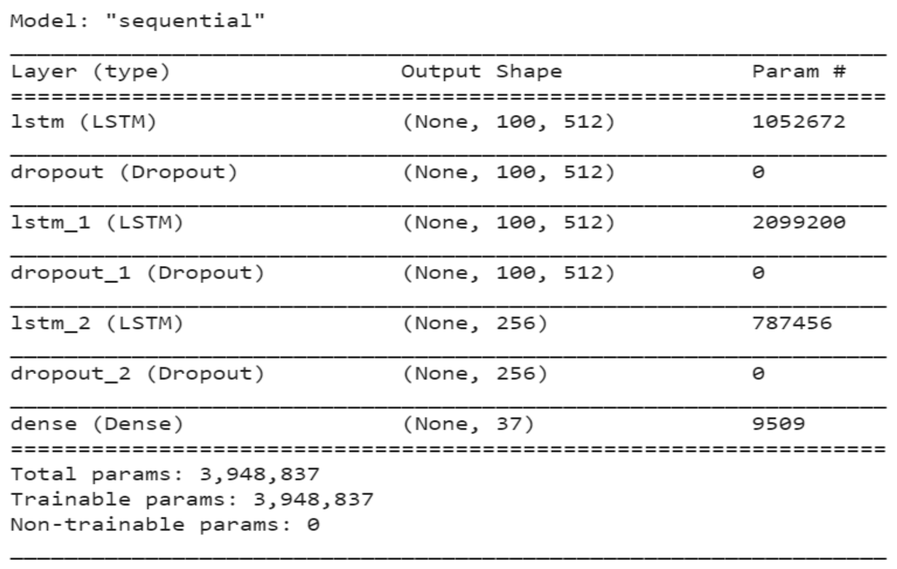


Fig 3.1: Data converted to vectors

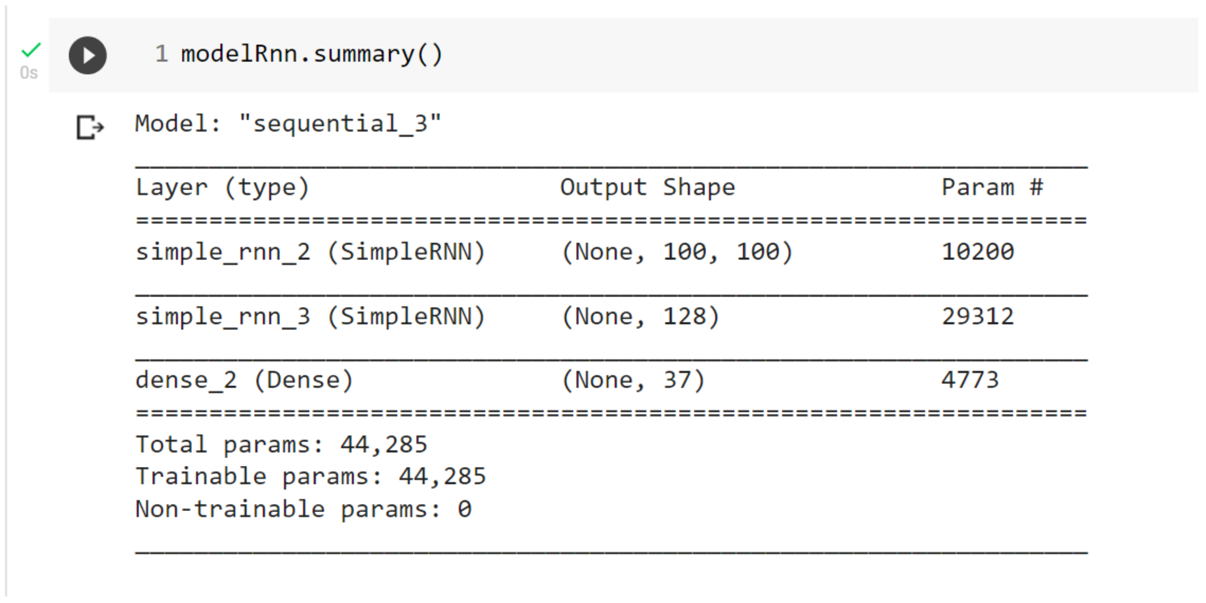
3.1.3 Define the neural network

We have used RNN, LSTM sequential neural network. Building models in Python alone is hard so using framework make it easy. These days there are different popular framework like Pytorch, Keras, TensorFlow exist. We choose Keras because it is easier to use. Keras API uses TensorFlow in the backend.



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 100, 512)          1052672
_________________________________________________________________
dropout (Dropout)            (None, 100, 512)          0
_________________________________________________________________
lstm_1 (LSTM)                (None, 100, 512)          2099200
_________________________________________________________________
dropout_1 (Dropout)          (None, 100, 512)          0
_________________________________________________________________
lstm_2 (LSTM)                (None, 256)               787456
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 37)                9509
=================================================================
Total params: 3,948,837
Trainable params: 3,948,837
Non-trainable params: 0
_________________________________________________________________
```

Fig 3.2 : LSTM architecture used



```
modelRnn.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_2 (SimpleRNN)     (None, 100, 100)          10200
_________________________________________________________________
simple_rnn_3 (SimpleRNN)     (None, 128)               29312
_________________________________________________________________
dense_2 (Dense)              (None, 37)                4773
=================================================================
Total params: 44,285
Trainable params: 44,285
Non-trainable params: 0
_________________________________________________________________
```

Fig 3.3: RNN architecture used

3.1.4 Training

Models have different configurable variables that change its capability, we trained models with different configurations and kept the best performing model.

- For training the model is provided a sequence of 100 characters, then after processing all 100 characters it outputs 101th character which is compared with the actual truth to calculate the loss with a loss function. This calculated loss is then backpropagated through the network to update the parameters so that next time model performs better.
- Training is done with different epoch value, where epoch is number of times entire data is iterated during training.

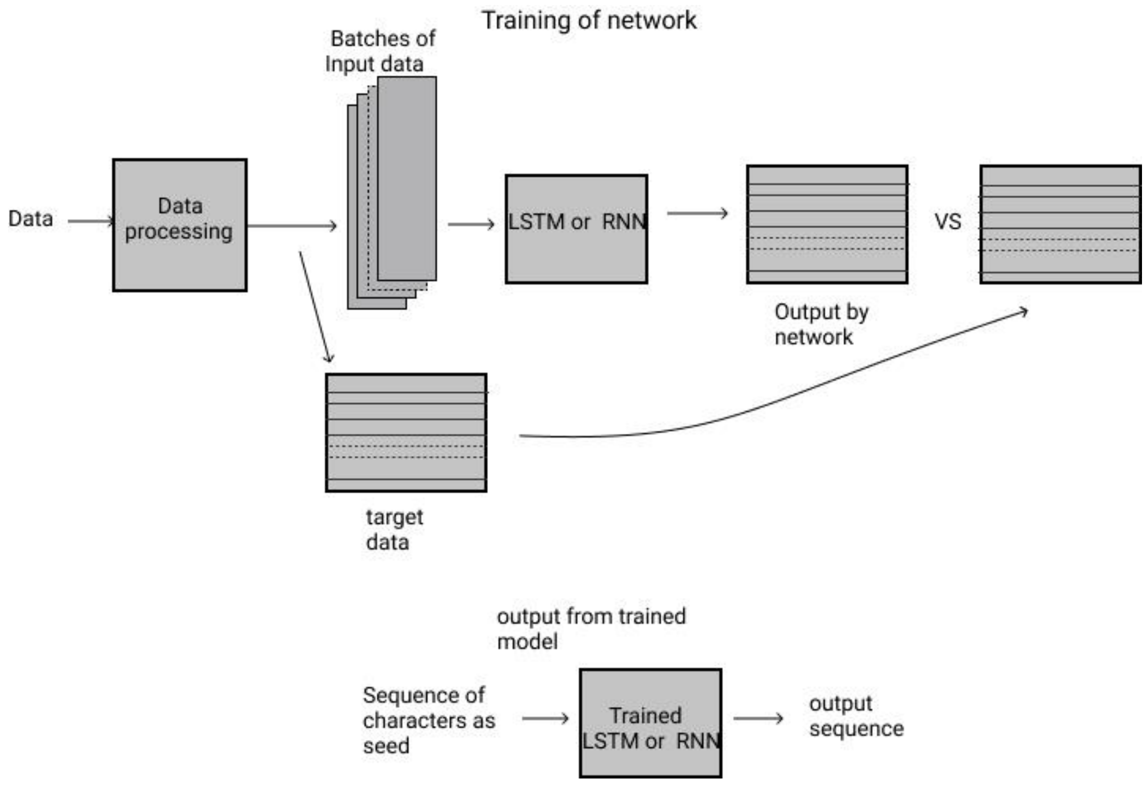


Fig 3.4: Functional block diagram

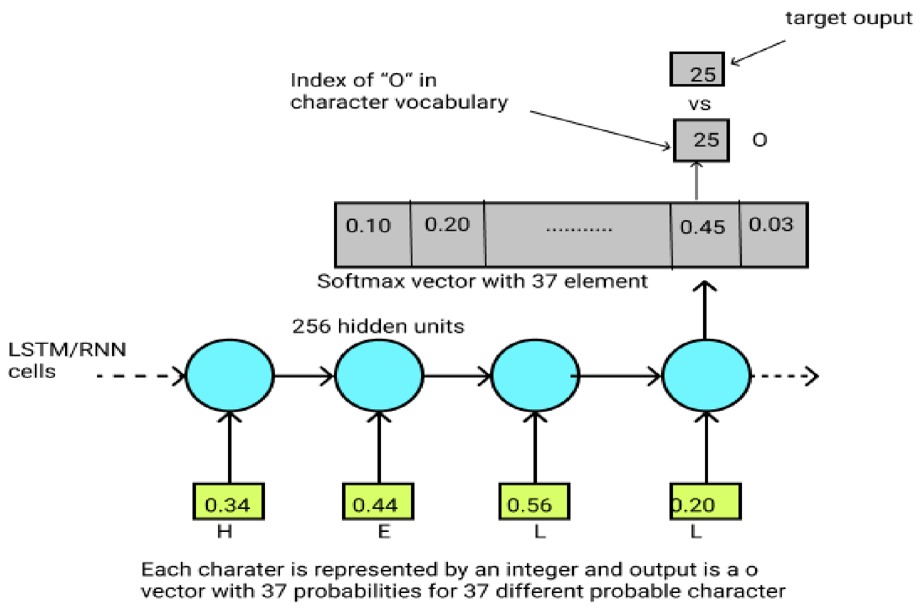

target ouput
25
vs
Index of "O" in character vocabulary
25
O
0.10
0.20
....
0.45
0.03
Softmax vector with 37 element
256 hidden units
LSTM/RNN cells
0.34
H
0.44
E
0.56
L
0.20
L
Each charater is represented by an integer and output is a o vector with 37 probabilities for 37 different probable character

Fig 3.5: Block diagram showing working of LSTM/RNN

3.1.5 Testing

After training of a model, it is provided a line of text to generate sequential text based on the provided input. Then the generated text of the model is analyzed by human to see the how the model is performing. This way we will train the two different models RNN and LSTM with different configurations and keep the one with best result.

3.1.6 Step regarding how the neural network generate output

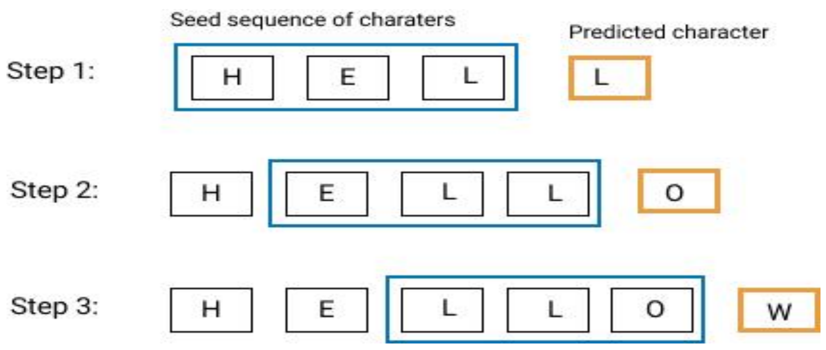


Fig 3.6: Output generation

3.2 Tools used

Software tools:

- Google colab Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education.
- Streamlit is an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science.

Programming language:

- Python : It is an easy-to-use programming language, it has a rich eco system of machine learning libraries which are used to build and train machine learning model easily.

Framework:

- Keras : Keras is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.
- NLTK: NLTK is one of the leading platforms for working with human language data and Python, the module NLTK is used for natural language processing. NLTK is literally an acronym for Natural Language Toolkit.

Dataset:

- The Yellow Wallpaper by Charlotte Perkins Gilman

CHAPTER 4

RESULT ANALYSIS

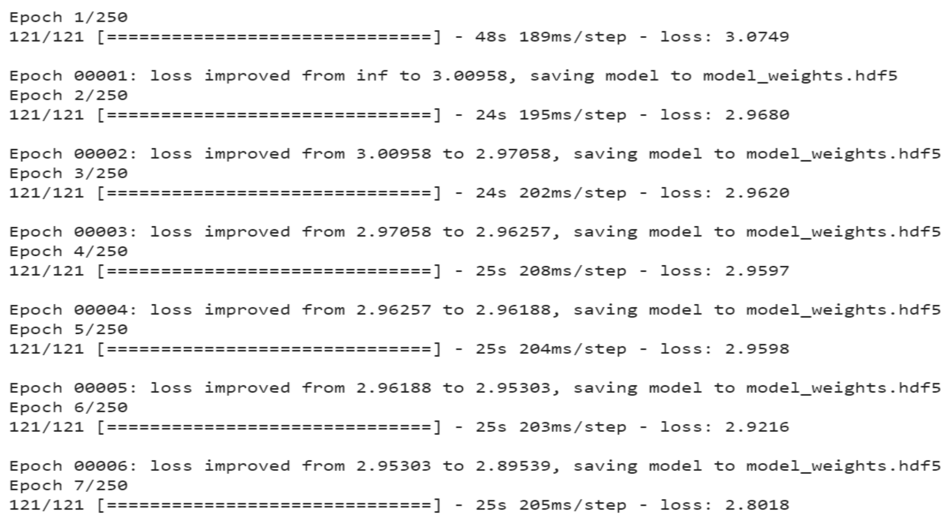
In this chapter we will be discussing the detailed results obtained after training the LSTM and RNN models.

4.1 Result Analysis

Here we will discuss the obtained results separately for both the models.

4.1.1 LSTM model

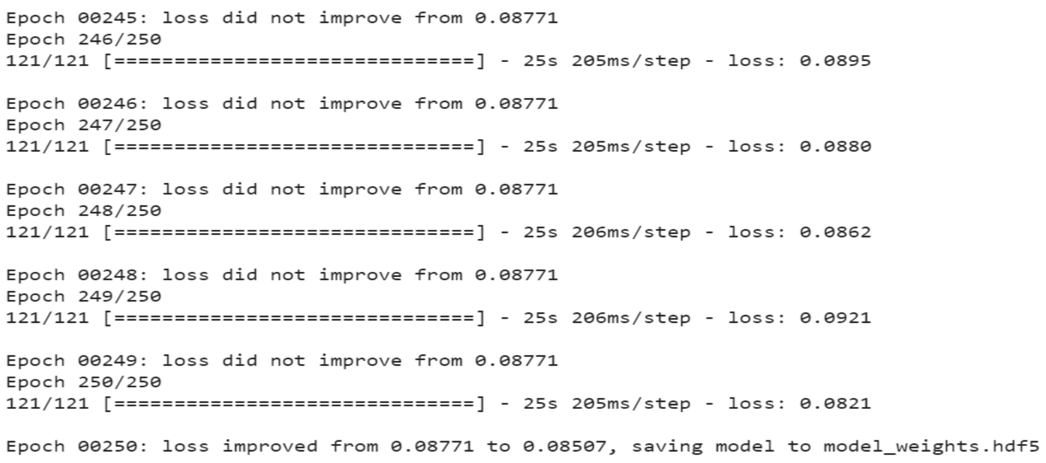
The LSTM model has trained upto 250 epochs, for which the batch size is 256.The loss value began with 3.00958.



```
Epoch 1/250
121/121 [==============================] - 48s 189ms/step - loss: 3.0749

Epoch 00001: loss improved from inf to 3.00958, saving model to model_weights.hdf5
Epoch 2/250
121/121 [==============================] - 24s 195ms/step - loss: 2.9680

Epoch 00002: loss improved from 3.00958 to 2.97058, saving model to model_weights.hdf5
Epoch 3/250
121/121 [==============================] - 24s 202ms/step - loss: 2.9620

Epoch 00003: loss improved from 2.97058 to 2.96257, saving model to model_weights.hdf5
Epoch 4/250
121/121 [==============================] - 25s 208ms/step - loss: 2.9597

Epoch 00004: loss improved from 2.96257 to 2.96188, saving model to model_weights.hdf5
Epoch 5/250
121/121 [==============================] - 25s 204ms/step - loss: 2.9598

Epoch 00005: loss improved from 2.96188 to 2.95303, saving model to model_weights.hdf5
Epoch 6/250
121/121 [==============================] - 25s 203ms/step - loss: 2.9216

Epoch 00006: loss improved from 2.95303 to 2.89539, saving model to model_weights.hdf5
Epoch 7/250
121/121 [==============================] - 25s 205ms/step - loss: 2.8018
```

Fig 4.1: Loss at the beginning of training LSTM

The loss went upto 0.0857 at the end of 250 epoch.



```
Epoch 00245: loss did not improve from 0.08771
Epoch 246/250
121/121 [==============================] - 25s 205ms/step - loss: 0.0895

Epoch 00246: loss did not improve from 0.08771
Epoch 247/250
121/121 [==============================] - 25s 205ms/step - loss: 0.0880

Epoch 00247: loss did not improve from 0.08771
Epoch 248/250
121/121 [==============================] - 25s 206ms/step - loss: 0.0862

Epoch 00248: loss did not improve from 0.08771
Epoch 249/250
121/121 [==============================] - 25s 206ms/step - loss: 0.0921

Epoch 00249: loss did not improve from 0.08771
Epoch 250/250
121/121 [==============================] - 25s 205ms/step - loss: 0.0821

Epoch 00250: loss improved from 0.08771 to 0.08507, saving model to model_weights.hdf5
```

Fig 4.2: Loss at the end of training LSTM

We have also made a plot for the epoch vs loss graph.

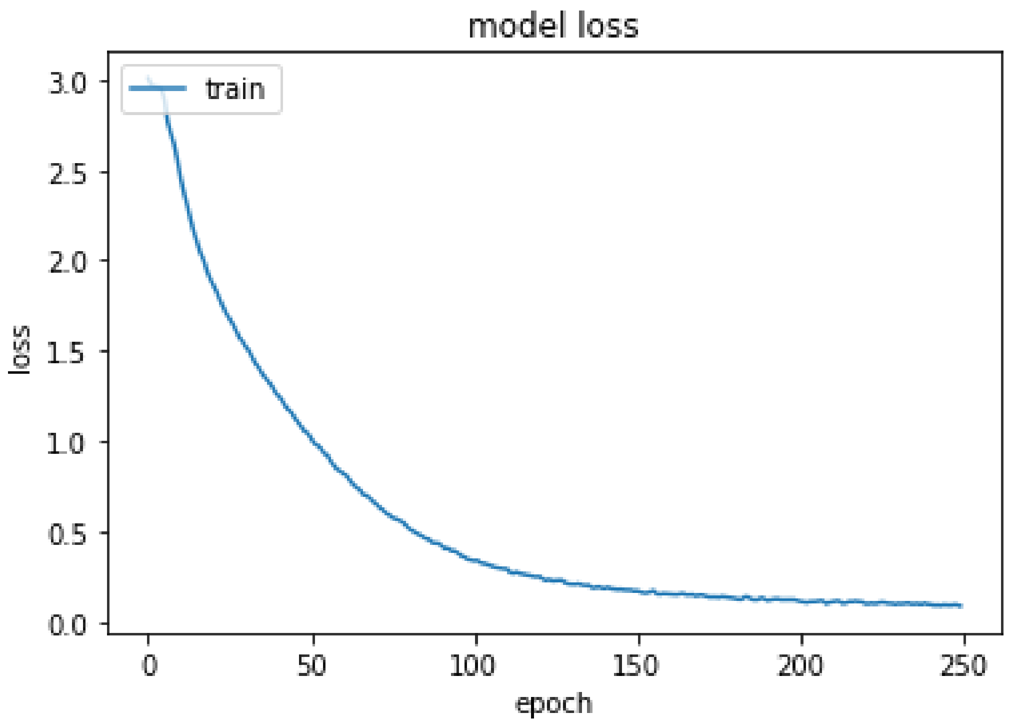
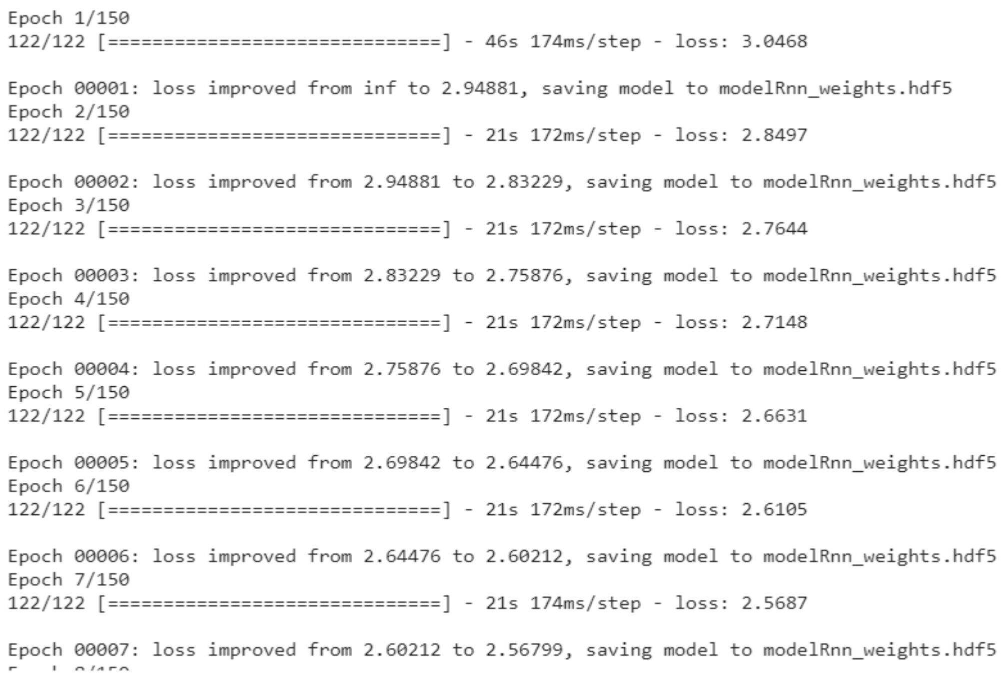


Fig 4.3: Loss vs Epoch graph for LSTM model training

4.1.2 RNN model

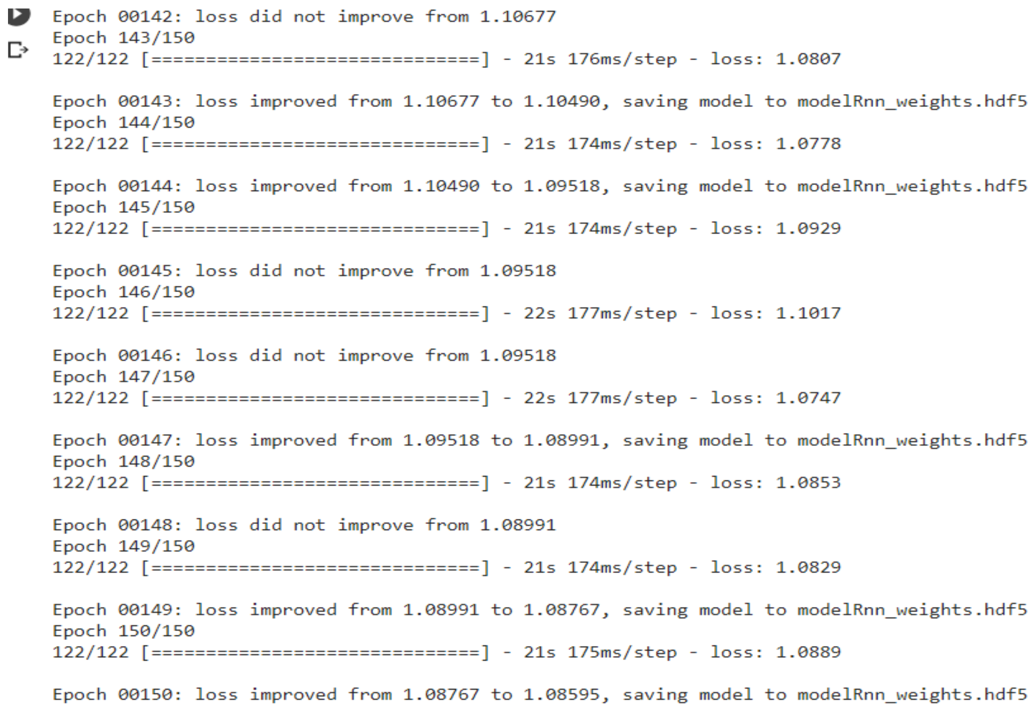
The RNN model has trained upto 150 epochs, for which the batch size is 256.The loss value began with 3.0468.



```
Epoch 1/150
122/122 [==============================] - 46s 174ms/step - loss: 3.0468

Epoch 00001: loss improved from inf to 2.94881, saving model to modelRnn_weights.hdf5
Epoch 2/150
122/122 [==============================] - 21s 172ms/step - loss: 2.8497

Epoch 00002: loss improved from 2.94881 to 2.83229, saving model to modelRnn_weights.hdf5
Epoch 3/150
122/122 [==============================] - 21s 172ms/step - loss: 2.7644

Epoch 00003: loss improved from 2.83229 to 2.75876, saving model to modelRnn_weights.hdf5
Epoch 4/150
122/122 [==============================] - 21s 172ms/step - loss: 2.7148

Epoch 00004: loss improved from 2.75876 to 2.69842, saving model to modelRnn_weights.hdf5
Epoch 5/150
122/122 [==============================] - 21s 172ms/step - loss: 2.6631

Epoch 00005: loss improved from 2.69842 to 2.64476, saving model to modelRnn_weights.hdf5
Epoch 6/150
122/122 [==============================] - 21s 172ms/step - loss: 2.6105

Epoch 00006: loss improved from 2.64476 to 2.60212, saving model to modelRnn_weights.hdf5
Epoch 7/150
122/122 [==============================] - 21s 174ms/step - loss: 2.5687

Epoch 00007: loss improved from 2.60212 to 2.56799, saving model to modelRnn_weights.hdf5
```

Fig 4.4: Loss at the beginning of training RNN

The loss went upto 1.08767 at the end of 150 epoch.



```
Epoch 00142: loss did not improve from 1.10677
Epoch 143/150
122/122 [==============================] - 21s 176ms/step - loss: 1.0807

Epoch 00143: loss improved from 1.10677 to 1.10490, saving model to modelRnn_weights.hdf5
Epoch 144/150
122/122 [==============================] - 21s 174ms/step - loss: 1.0778

Epoch 00144: loss improved from 1.10490 to 1.09518, saving model to modelRnn_weights.hdf5
Epoch 145/150
122/122 [==============================] - 21s 174ms/step - loss: 1.0929

Epoch 00145: loss did not improve from 1.09518
Epoch 146/150
122/122 [==============================] - 22s 177ms/step - loss: 1.1017

Epoch 00146: loss did not improve from 1.09518
Epoch 147/150
122/122 [==============================] - 22s 177ms/step - loss: 1.0747

Epoch 00147: loss improved from 1.09518 to 1.08991, saving model to modelRnn_weights.hdf5
Epoch 148/150
122/122 [==============================] - 21s 174ms/step - loss: 1.0853

Epoch 00148: loss did not improve from 1.08991
Epoch 149/150
122/122 [==============================] - 21s 174ms/step - loss: 1.0829

Epoch 00149: loss improved from 1.08991 to 1.08767, saving model to modelRnn_weights.hdf5
Epoch 150/150
122/122 [==============================] - 21s 175ms/step - loss: 1.0889

Epoch 00150: loss improved from 1.08767 to 1.08595, saving model to modelRnn_weights.hdf5
```

Fig 4.5: Loss at the end of training RNN

Similarly, we have also made a plot for the epoch vs loss graph for this model.

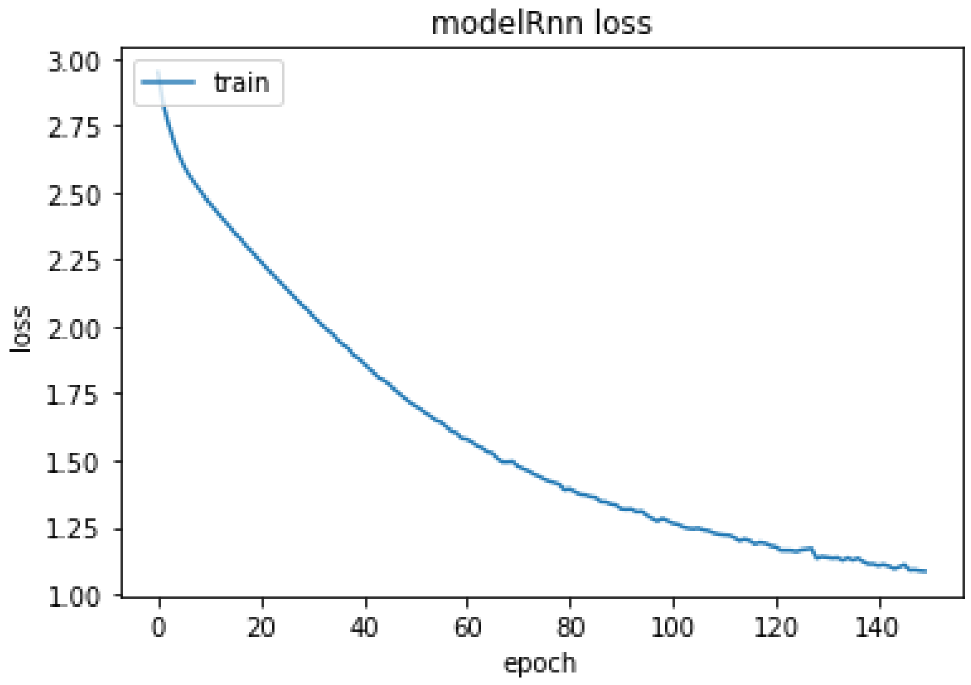


Fig 4.6: Loss vs Epoch graph for RNN model training

4.2 Significance of the results obtained

After training both the deep learning models with the designated parameters, we compared their final loss values, and also generated some texts using both of these models and analyzed both the obtained results.

For LSTM model:

Random seed which we feed into the model was:

“expression used lie awake child get entertainment terror blank walls plain furniture children could”

The generated text was:

“ find toy store remember kindly wink knobs big old bureau used one chair always seemed like strong friend used feel things looked fierce could always hop chair safe furniture room worse inharmonious however bring downstairs suppose used playroom take nursery things wonder never saw ravages children made wall paper said torn spots sticketh closer brother must perseverance well hatred floor scratched gouged splintered plaster dug great heavy bed found room looks wars mind bit paper comes john siste ”

For the RNN model:

Random seed which we feed into the model was:

“d sub pattern different s ”

The generated text was:

“ oopedty oirtirted mparen mentkrid bseaptent hwmtseght usevictanu prualtn fbseldluenee tlare muse tom nrto srotedt suojict gutenberg tm mlssso arranr line toulddgt faer avenghb huow begt kay wiuse comler sne bnario ttidballmy tay meu intt digois shovght pu hrmu gxer haan sfltleacoamt 1 g fen innidir gute bere eive mey aauc wint rnotguee hafa uie caroirg thuer seruor rirho coeli tarseres awcycmt ebyf moundarie ptrjrgmy duceusins sar corthiaet rolotlmec maies sllnk uuetn tnti shmngr derirtbecblt dp ”

Now it may as well happen that the predicted sentence may not be syntactically valid but the overall gist of the sentence can be easily understood. Therefore by analyzing the both the generated outputs and looking at final loss value for both of the models, we decided to select the LSTM model for our final text generation web app which has been made through streamlit and hosted in streamlit.

CHAPTER 5
CONCLUSION AND FUTURE SCOPE OF WORK

In this chapter we will discuss the conclusion and the future scope of the project that we have carried out.

5.1 Summary of the work

Our project is to build an automatic text generator from scratch, using deep learning models. We started this project by studying about automatic text generation and the required algorithms. During the study about algorithms, we studied about neural networks like Recurrent Neural Networks (RNN) and Long Short Term Memory (LSTM). Then we selected our dataset from Project Gutenberg, which is an e-book named “The Yellow Wallpaper” by Charlotte Perkins Gilman. Then we tried different algorithms for our project, which were RNN and LSTM. Then from these two models we selected the best. As our dataset is not huge and we were limited by both computing power and knowledge on machine learning, so building a sophisticated model which can generate meaningful output was a challenging task. As a result the output sentences from the model are not very meaningful but still the model can generate proper English words.

5.2 Conclusions

So after training and selecting the best model, it is hosted on the web with the help of streamlit. Streamlit is an open-source Python library that makes it easy to create and share beautiful, custom web apps for machine learning and data science. In the hosted web app, the users have to give a meaningful sentence and based on that sentence the model will generate some output. Right now the number of characters the model will generate is limited to 1000 characters. We have not trained the network on huge amount of different types of text data, so the model only learnt the structure and context of the dataset we provided which was not very big. That is why the output sentences are not very meaningful, but still the model can generate proper words which can be found in English dictionary.

One may also think that the predictions made by the model looks somewhat trivial, but it has to be kept in mind that prior to Text Generation our device was incapable of forming novel sentences, in-fact it was unaware of even the basic alphabets needed to construct words.

5.3 Future scope of work

LSTMs are very powerful. But task like meaningful and usable text generation is very complicated and it takes a huge amount of data (would be couple of gigabytes). So a much more sophisticated model like (Attention model) with huge amount of data along with more computation power might give better results. One may include grammar rules to make the text syntactically correct. The output text can be made a new story by training the models on different stories. One can also try to generate musical lyrics by training the model with some music files

REFERENCES

[1]. Karpathy.github.io. 2015. *The Unreasonable Effectiveness of Recurrent Neural Networks*. [online] Available at: <http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

[2]. A. Sherstinsky, “Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network,” *Physica D: Nonlinear Phenomena*, vol. 404, p. 132306, 2020.

[3]. Perkins, C., 1999. *The Yellow Wallpaper*. 1st ed. [ebook] Available at: <https://www.gutenberg.org/ebooks/1952> [Accessed 25 July 2021].

ANNEXURES

Code Used:

Automatic Text Generation

```
#importing dependencies import numpy as np

import sys

from nltk.tokenize import RegexpTokenizer from nltk.corpus import stopwords

from keras.models import Sequential,load_model

from keras.layers import Dense, Dropout, LSTM, SimpleRNN from keras.utils import np_utils

from keras.callbacks import ModelCheckpoint import nltk

nltk.download('stopwords')

import matplotlib.pyplot as plt import keras.callbacks
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data... [nltk_data]          Package
stopwords is already up-to-date!
```

Downloading Dataset

```
import os.path import shutil

from google.colab import drive

if  not  os.path.exists('/content/gdrive'): drive.mount('/content/gdrive')
```

```
DOWNLOAD_LOCATION = '/content/'

DRIVE_DATASET_FILE                              =                              '/content/gdrive/My
Drive/AutoTextGen/The_Yellow_Wallpaper_by_Charlotte_
shutil.copy(DRIVE_DATASET_FILE, DOWNLOAD_LOCATION)

print('Successfully downloaded the dataset')
```

```
Mounted at /content/gdrive

Successfully downloaded the dataset
```

```
DOWNLOAD_LOCATION = '/content/'

DRIVE_DATASET_FILE    =    '/content/gdrive/My    Drive/AutoTextGen/model_weights.hdf5'
#adjust  path/n shutil.copy(DRIVE_DATASET_FILE, DOWNLOAD_LOCATION)

print('Successfully downloaded the dataset')


file = open("The_Yellow_Wallpaper_by_Charlotte_Perkins_Gilman.txt").read()


#function to preprocess text def tokenize_words(input):

# lowercase everything to standardize it input = input.lower()


#removing punctuation

tokenizer = RegexpTokenizer(r'\w+') tokens = tokenizer.tokenize(input)
```

```
filtered = filter(lambda token: token not in stopwords.words('english'), tokens) return " ".join(filtered)



#preprocess the text

processed_text = tokenize_words(file)



#sorting the list of chars

chars = sorted(list(set(processed_text)))

char_to_num = dict((c, i) for i, c in enumerate(chars))



print(chars)


[' ', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f', '



import pickle
```

```
with  open('Chars.pkl','wb')  as  f: pickle.dump(chars,f)
```

```
#getting the number of input length and vocab size input_len = len(processed_text)
vocab_len = len(chars)
print ("Total number of characters:", input_len) print ("Total vocab:", vocab_len)
```

```
Total number of characters: 31121 Total vocab: 37
```

```
#seting up a sequence seq_length = 100
x_data = [] y_data = []
```

```
#converting into numeric data
for  i  in  range(0,  input_len  -  seq_length,  1):
# Input is the current character plus desired sequence length in_seq = processed_text[i:i + seq_length]
```

```
# Out sequence is the initial character plus total sequence length out_seq = processed_text[i + seq_length]
```

# We now convert list of characters to integers based on x_data.append([char_to_num[char] for char in in_seq])

y_data.append(char_to_num[out_seq])

#number of input sequences n_patterns = len(x_data)

print ("Total Patterns:", n_patterns)

Total Patterns: 31021

with open('Xdata.pkl','wb') as f2: pickle.dump(x_data,f2)

#converting input sequences into numpy array

X = np.reshape(x_data,(n_patterns, seq_length, 1)) X = X/float(vocab_len)

# one hot encoding our label data

y = np_utils.to_categorical(y_data)

#printing X print(X)

```
...

[[[0.7027027 ]

[0.75675676]

[0.67567568]

...

[0.37837838]

[0.        ]

[0.78378378]]


[[0.75675676]

[0.67567568]

[0.54054054]

...

[0.        ]

[0.78378378]

[0.81081081]]


[[0.67567568]

[0.54054054]


[0.40540541]

...
```

```
[0.78378378]

[0.81081081]

[0.2972973 ]]


...


[[0.83783784]

[0.81081081]

[0.40540541]

...

[0.40540541]

[0.32432432]

[0.67567568]]


[[0.81081081]

[0.40540541]

[0.64864865]

...

[0.32432432]

[0.67567568]

[0.67567568]]


[[0.40540541]
```

```
[0.64864865]

[0.32432432]

...

[0.67567568]

[0.67567568]

[0.56756757]]]
```

```
#print y data print(y)
```

```
[[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.  0.]

 [0.   0.  0.  ...   0.  0.  0.]]
```

```
#LSTM Model

model = Sequential()

model.add(LSTM(512, input_shape=(X.shape[1], X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(512,  return_sequences=True)) model.add(Dropout(0.2))

model.add(LSTM(256))

model.add(Dropout(0.2))

model.add(Dense(y.shape[1],  activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='adam')


model.summary()
```

```
Model: "sequential"
_________________________________________________________________

Layer  (type)   Output Shape  Param #
=================================================================
        lstm (LSTM)              (None,      100,  512)      1052672
        _________________________________________________________
        dropout (Dropout)        (None,      100,  512)      0
        _________________________________________________________
        lstm_1 (LSTM)            (None,      100,  512)      2099200
        _________________________________________________________
        dropout_1 (Dropout)      (None,      100,  512)      0
        _________________________________________________________
        lstm_2 (LSTM)            (None,      256)            787456
        _________________________________________________________
        dropout_2 (Dropout)      (None,      256)            0
        _________________________________________________________
        dense (Dense)            (None,      37)             9509
=================================================================
Total params: 3,948,837

Trainable params: 3,948,837
```

Non-trainable params: 0

---

filepath = "model_weights.hdf5"

checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True, mode=' desired_callbacks = [checkpoint]

#training the model

History = model.fit(X, y, epochs=250, batch_size=256, callbacks=desired_callbacks)

```
Epoch 1/250

121/121 [==============================] - 48s 189ms/step - loss: 3.0749


Epoch 00001: loss improved from inf to 3.00958, saving model to model_weights.hdf5 Epoch 2/250

121/121 [==============================] - 24s 195ms/step - loss: 2.9680


Epoch 00002: loss improved from 3.00958 to 2.97058, saving model to model_weights.hd Epoch 3/250

121/121 [==============================] - 24s 202ms/step - loss: 2.9620
```

```
Epoch 00003: loss improved from 2.97058 to 2.96257, saving model to model_weights.hd Epoch
4/250

121/121 [==============================] - 25s 208ms/step - loss: 2.9597


Epoch 00004: loss improved from 2.96257 to 2.96188, saving model to model_weights.hd Epoch
5/250

121/121 [==============================] - 25s 204ms/step - loss: 2.9598


Epoch 00005: loss improved from 2.96188 to 2.95303, saving model to model_weights.hd Epoch
6/250

121/121 [==============================] - 25s 203ms/step - loss: 2.9216


Epoch 00006: loss improved from 2.95303 to 2.89539, saving model to model_weights.hd Epoch
7/250

121/121 [==============================] - 25s 205ms/step - loss: 2.8018


Epoch 00007: loss improved from 2.89539 to 2.76860, saving model to model_weights.hd Epoch
8/250

121/121 [==============================] - 25s 204ms/step - loss: 2.7178


Epoch 00008: loss improved from 2.76860 to 2.70141, saving model to model_weights.hd Epoch
9/250

121/121 [==============================] - 25s 204ms/step - loss: 2.6557
```

```
Epoch 00009: loss improved from 2.70141 to 2.64137, saving model to model_weights.hd Epoch
10/250

121/121 [==============================] - 25s 203ms/step - loss: 2.5783


Epoch 00010: loss improved from 2.64137 to 2.55474, saving model to model_weights.hd Epoch
11/250

121/121 [==============================] - 25s 204ms/step - loss: 2.4663


Epoch 00011: loss improved from 2.55474 to 2.45362, saving model to model_weights.hd Epoch
12/250

121/121 [==============================] - 25s 204ms/step - loss: 2.3952


Epoch 00012: loss improved from 2.45362 to 2.37410, saving model to model_weights.hd Epoch
13/250

121/121 [==============================] - 25s 203ms/step - loss: 2.3299


Epoch 00013: loss improved from 2.37410 to 2.30528, saving model to model_weights.hd Epoch
14/250

121/121 [==============================] - 25s 204ms/step - loss: 2.2283


Epoch 00014: loss improved from 2.30528 to 2.22832, saving model to model_weights.hd Epoch
15/250

121/121 [==============================] - 25s 204ms/step - loss: 2.1746
```

```
filename = "model_weights.hdf5" model.load_weights(filename)

model.compile(loss='categorical_crossentropy', optimizer='adam')



filename  =  "model_weights.hdf5" md = load_model(filename)

md.compile(loss='categorical_crossentropy', optimizer='adam')



num_to_char = dict((i, c) for i, c in enumerate(chars))



start  =  np.random.randint(0,  len(x_data)  -  1) pattern = x_data[start]

print("Random Seed:")

print("\"",  ''.join([num_to_char[value]  for  value  in  pattern]),  "\"")
```

```
Random Seed:

" expression used lie awake child get entertainment terror blank walls plain furniture
```

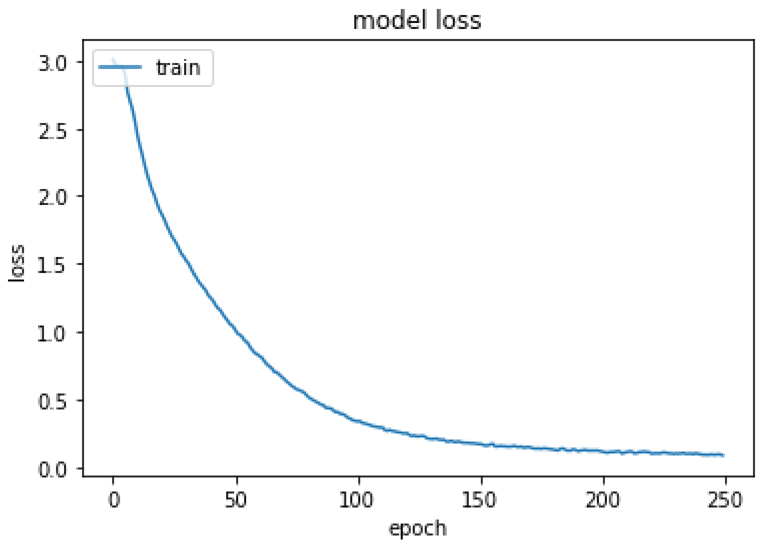
```
for i in range(500):
```

```
x  =  np.reshape(pattern,  (1,  len(pattern),  1)) x = x / float(vocab_len)

prediction = model.predict(x, verbose=0) index = np.argmax(prediction)

result = num_to_char[index] sys.stdout.write(result)

pattern.append(index)

pattern  =  pattern[1:len(pattern)]
```

find toy store remember kindly wink knobs big old bureau used one chair always seemed l

```
print(History.history.keys()) dict_keys(['loss'])
```

```
plt.plot(History.history['loss']) plt.title('model loss')

plt.ylabel('loss') plt.xlabel('epoch')

plt.legend(['train'], loc='upper left') plt.show()
```



```
#RNN Model

modelRnn = Sequential()

modelRnn.add(SimpleRNN(100,                    input_shape=(X.shape[1],                    X.shape[2]),
return_sequences=True))


modelRnn.add(SimpleRNN(128))

modelRnn.add(Dense(y.shape[1],  activation='softmax'))

modelRnn.compile(loss='categorical_crossentropy', optimizer='adam')



modelRnn.summary()


Model: "sequential_3"

 ____________________________________________________________________

Layer  (type)   Output Shape  Param #
=================================================================
        simple_rnn_2 (SimpleRNN)     (None,      100,   100)          10200
        ____________________________________________________________________
        simple_rnn_3 (SimpleRNN)     (None,      128)                 29312
        ____________________________________________________________________
        dense_2 (Dense)              (None,      37)                  4773
=================================================================
Total params: 44,285

Trainable params: 44,285
```

```
Non-trainable params: 0
```

```
filepath = "modelRnn_weights.hdf5"

checkpoint = ModelCheckpoint(filepath, monitor='loss', verbose=1, save_best_only=True,
mode=' desired_callbacks = [checkpoint]
```

```
#training the model

HistoryRnn = modelRnn.fit(X, y, epochs=150, batch_size=256, callbacks=desired_callbacks)
122/122 [==============================] - 21s 174ms/step - loss: 1.2126

Epoch 00112: loss improved from 1.22049 to 1.21928, saving model to modelRnn_weights
Epoch 113/150

122/122 [==============================] - 21s 175ms/step - loss: 1.1863


Epoch 00113: loss improved from 1.21928 to 1.21092, saving model to modelRnn_weights
Epoch 114/150

122/122 [==============================] - 21s 175ms/step - loss: 1.1810


Epoch 00114: loss improved from 1.21092 to 1.19871, saving model to modelRnn_weights
Epoch 115/150

122/122 [==============================] - 21s 174ms/step - loss: 1.2106
```

```
Epoch 00115: loss did not improve from 1.19871 Epoch 116/150

122/122 [==============================] - 21s 175ms/step - loss: 1.1845


Epoch 00116: loss did not improve from 1.19871 Epoch 117/150

122/122 [==============================] - 21s 174ms/step - loss: 1.1879


Epoch 00117: loss improved from 1.19871 to 1.18664, saving model to modelRnn_weights
Epoch 118/150

122/122 [==============================] - 21s 173ms/step - loss: 1.1883


Epoch 00118: loss did not improve from 1.18664

Epoch 119/150

122/122 [==============================] - 21s 174ms/step - loss: 1.1748


Epoch 00119: loss did not improve from 1.18664 Epoch 120/150

122/122 [==============================] - 21s 172ms/step - loss: 1.1708


Epoch 00120: loss improved from 1.18664 to 1.18054, saving model to modelRnn_weights
Epoch 121/150

122/122 [==============================] - 21s 172ms/step - loss: 1.1529
```

```
Epoch 00121: loss improved from 1.18054 to 1.17648, saving model to modelRnn_weights
Epoch 122/150

122/122 [==============================] - 21s 173ms/step - loss: 1.1463


Epoch 00122: loss improved from 1.17648 to 1.16431, saving model to modelRnn_weights
Epoch 123/150

122/122 [==============================] - 22s 177ms/step - loss: 1.1471


Epoch 00123: loss did not improve from 1.16431 Epoch 124/150

122/122 [==============================] - 21s 175ms/step - loss: 1.1412


Epoch 00124: loss improved from 1.16431 to 1.16392, saving model to modelRnn_weights
Epoch 125/150

122/122 [==============================] - 21s 175ms/step - loss: 1.1560


Epoch 00125: loss improved from 1.16392 to 1.15981, saving model to modelRnn_weights
Epoch 126/150

122/122 [==============================] - 21s 176ms/step - loss: 1.1578
```

```
filenameRnn = "modelRnn_weights.hdf5" modelRnn.load_weights(filenameRnn)

modelRnn.compile(loss='categorical_crossentropy', optimizer='adam')
```

```
start  =  np.random.randint(0,  len(x_data)  -  1) pattern = x_data[start]
```

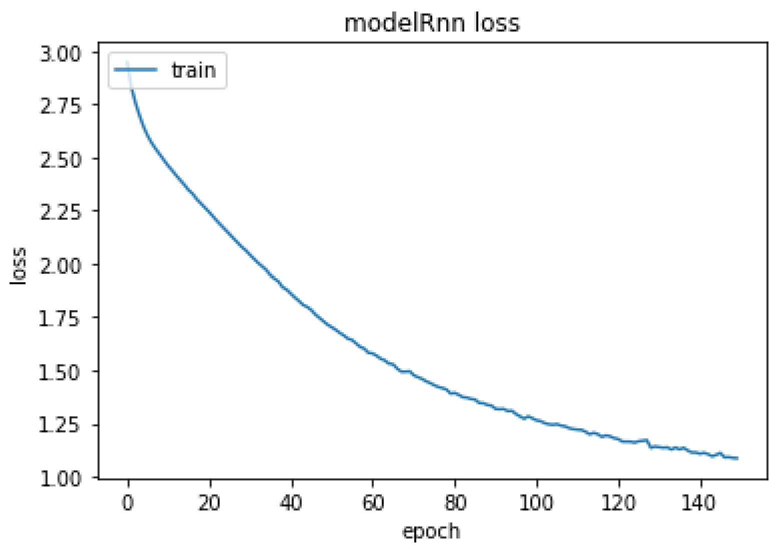
```
print("Random Seed:")

print("\"", ''.join([num_to_char[value]  for  value  in  pattern]),  "\"")
```

```
Random Seed:

" ed permission copyright holder work copied distributed anyone united states without p
```

```
for i in range(500):

x  =  np.reshape(pattern,  (1,  len(pattern),  1)) x = x / float(vocab_len)

prediction = modelRnn.predict(x, verbose=0) index = np.argmax(prediction)

result = num_to_char[index]


sys.stdout.write(result)

pattern.append(index)
pattern  =  pattern[1:len(pattern)]
```

```
   ge  say  womn  lissle  woukd  cerierantes  minhtedet  guten  erontsatiog  carafee
  dinuriout  pek
```

```
plt.plot(HistoryRnn.histor
y['loss'])
plt.title('modelRnn loss')
plt.
ylab
el('l
oss'
)
plt.x
labe
l('ep
och'
)
plt.legend(['train'],
loc='upper left')
plt.show()
```


modelRnn loss
train
loss
epoch