**Shakespeare Plays Text Generation By Using Recurrent Neural Networks**

**-2015556452**

**-Özgün**

**-carpathiarms@gmail.com**

1. **Introduction**

In this study, the main purpose is setting up a new text generator with Recurrent Neural Network and train that system to create a new play exactly like Shakespeare’s way. Text generation is a popular problem in Data Science and Machine Learning, and it is a suitable task for Recurrent Neural Networks. This study uses the Tensorflow library to build an RNN text generator and builds a high-level Application Programming Interface. The study has three main parts Dataset, RNN Text Generator, and Model Selector. First, you read the data(text) and process the text then vectorize it. After you build the model to train it. Configure the checkpoints and execute the training. Finally, you generate the text and restore the checkpoints.

1. **Material and Method**
2. **Google Shakespeare Plays Dataset**

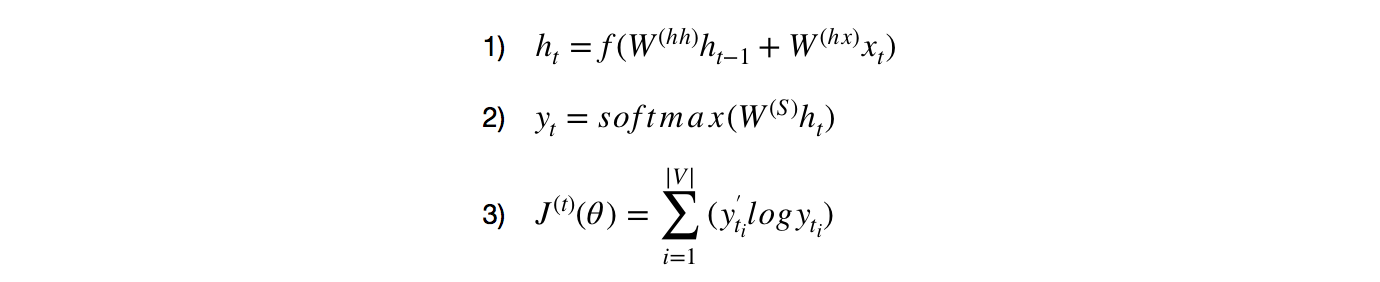
Result of my Google researches, I found an opportunity to use **Google Shakespeare Plays Dataset** (Which I attend into references. This dataset contains all of Shakespeare’s plays. Shakespeare’s Plays Dataset contains 1.115.394 rows of sentences. The data set consists of 715 users (characters of Shakespeare plays), where each example corresponds to a contiguous set of lines spoken by the character in a given play. Data set sizes: train: 16,068 examples, train 2,356 examples. Rather than holding out specific users, each user's examples are split across train and test so that all users have at least one example in train and one example in test. Characters that had less than 2 examples are excluded from the data set.

1. **Recurrent Neural Network**

A recurrent neural network (RNN) is a class of [artificial neural networks](https://en.wikipedia.org/wiki/Artificial_neural_network" \o "Artificial neural network) where connections between nodes form a [directed graph](https://en.wikipedia.org/wiki/Directed_graph" \o "Directed graph) along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from [feedforward neural networks](https://en.wikipedia.org/wiki/Feedforward_neural_networks" \o "Feedforward neural networks), RNNs can use their internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as unsegmented, connected [handwriting recognition](https://en.wikipedia.org/wiki/Handwriting_recognition" \o "Handwriting recognition)or [speech recognition](https://en.wikipedia.org/wiki/Speech_recognition" \o "Speech recognition).

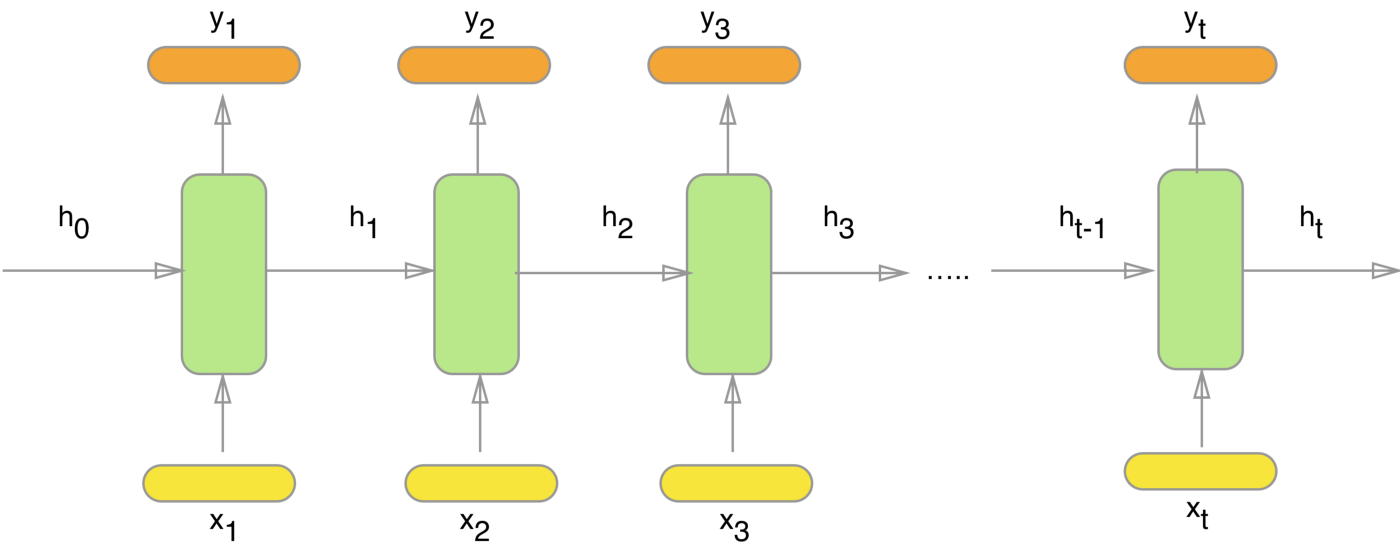
The term “recurrent neural network” is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is [finite impulse](https://en.wikipedia.org/wiki/Finite_impulse_response" \o "Finite impulse response) and the other is [infinite impulse](https://en.wikipedia.org/wiki/Infinite_impulse_response" \o "Infinite impulse response). Both classes of networks exhibit temporal [dynamic behavior](https://en.wikipedia.org/wiki/Dynamic_system" \o "Dynamic system). A finite impulse recurrent network is a [directed acyclic graph](https://en.wikipedia.org/wiki/Directed_acyclic_graph" \o "Directed acyclic graph) that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a [directed cyclic graph](https://en.wikipedia.org/wiki/Directed_cyclic_graph" \o "Directed cyclic graph) that can not be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of [long short-term memory](https://en.wikipedia.org/wiki/Long_short-term_memory" \o "Long short-term memory) networks (LSTMs) and [gated recurrent units](https://en.wikipedia.org/wiki/Gated_recurrent_unit" \o "Gated recurrent unit). This is also called Feedback Neural Network.

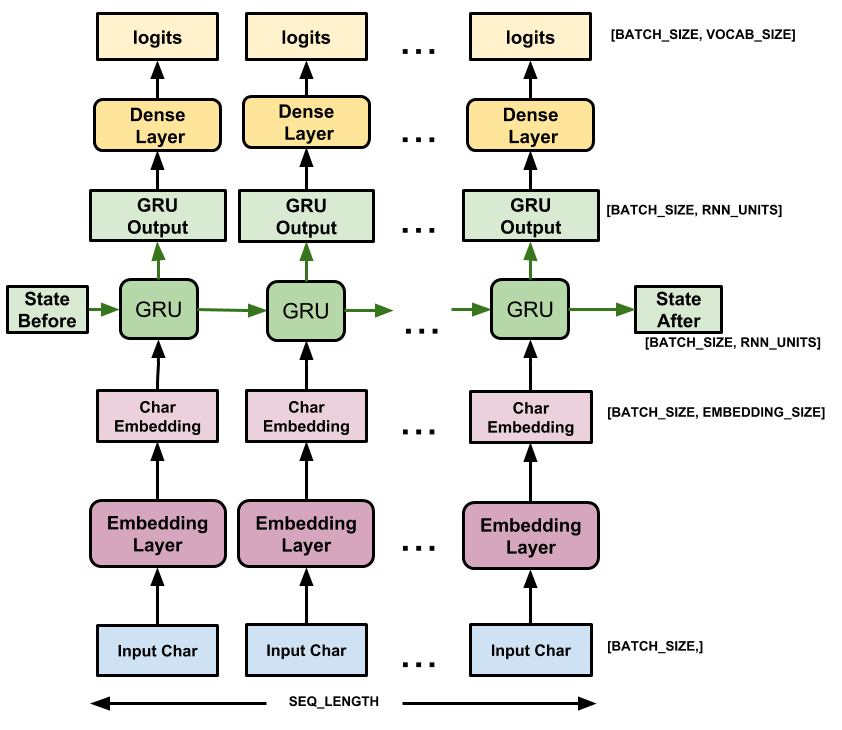


1. Holds information about the previous words in the sequence. As you can see, h\_t is calculated using the previous h\_(t-1) vector and current word vector x\_t*.* We also apply a non-linear activation function f(usually [tanh](https://theclevermachine.wordpress.com/tag/tanh-function/" \t "_blank) or [sigmoid](https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Sigmoid_function.html)) to the final summation. It is acceptable to assume that h\_0is a vector of zeros.
2. Calculates the predicted word vector at a given time step t. We use the [softmax function](https://www.youtube.com/watch?v=mlaLLQofmR8" \t "_blank) to produce a (V,1) vector with all elements summing up to 1. This probability distribution gives us the index of the most likely next word from the vocabulary.

3) Uses the [cross-entropy](https://www.youtube.com/watch?v=tRsSi_sqXjI" \t "_blank) loss function at each time step t to calculate the error between the predicted and actual word.



For each character the model looks up the embedding, runs the GRU one timestep with the embedding as input, and applies the dense layer to generate logits predicting the log-likelihood of the next character:



1. **Python Codes**

#I had import tensorflow, numpy, os and time libraries here

import tensorflow as tf

import numpy as np

import os

import time

#With that path I downloaded the Shakespeare dataset online

path\_to\_file = tf.keras.utils.get\_file('shakespeare.txt', 'https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt')

# It is for first look of dataset I’m reading the text here

text = open(path\_to\_file, 'rb').read().decode(encoding='utf-8')

print ('Line Count of text: {} characters'.format(len(text)))

#Here I made a counter to show the row number

count=0

for line in text:

    count=count+1

print(count)

# I took a look at the first 1000 characters in text

print(text[:1000])

# The unique characters in the file we have 65 unique characters here

vocab = sorted(set(text))

print ('{} unique characters'.format(len(vocab)))

# From that unique characters I did a mapping and created 2 look up tables one for mapping characters to numbers and other one for numbers to characters..

char2idx = {u:i for i, u in enumerate(vocab)}

idx2char = np.array(vocab)

text\_as\_int = np.array([char2idx[c] for c in text])

# I displayed that characters

print('{')

for char,\_ in zip(char2idx, range(20)):

    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))

print('  ...\n}')

#Here I’m showing how the first 13 characters from the text are mapped to integers

print ('{} ---- characters mapped to int ---- > {}'.format(repr(text[:13]), text\_as\_int[:13]))

# The maximum length sentence we want for a single input in characters. Each input will contain 100 characters because I made seg\_length=100 so it will be shifted to the right seg\_length+1. For instance seg\_length is 4 and my text is Hello my input will be Hello and target will be ello

seq\_length = 100

examples\_per\_epoch = len(text)//(seq\_length+1)

# This is my training examples or targets

char\_dataset = tf.data.Dataset.from\_tensor\_slices(text\_as\_int)

for i in char\_dataset.take(5):

  print(idx2char[i.numpy()])

#This is the batch method. It is easily convert individual characters to sequences of size that we want

sequences = char\_dataset.batch(seq\_length+1, drop\_remainder=True)

for item in sequences.take(5):

  print(repr(''.join(idx2char[item.numpy()])))

#For all sequences we shift and dublicated it from input and target text by using the map method to apply a simple function to each batch

def split\_input\_target(chunk):

    input\_text = chunk[:-1]

    target\_text = chunk[1:]

    return input\_text, target\_text

dataset = sequences.map(split\_input\_target)

#Here I printed a first example of input and target value.

for input\_example, target\_example in  dataset.take(1):

  print ('Input data: ', repr(''.join(idx2char[input\_example.numpy()])))

  print ('Target data:', repr(''.join(idx2char[target\_example.numpy()])))

#All vectors are working as one time step. For instance input step 0 recevies the index for “F” and it is trying to predict the character for index “i” as teh next character. It is repeating the same steps but only one thing different is RNN considers previous step context in addition to current character input that we use.

for i, (input\_idx, target\_idx) in enumerate(zip(input\_example[:5], target\_example[:5])):

    print("Step {:4d}".format(i))

    print("  input: {} ({:s})".format(input\_idx, repr(idx2char[input\_idx])))

    print("  expected output: {} ({:s})".format(target\_idx, repr(idx2char[target\_idx])))

#We need to create training batches so we use tf.data to split the text into meaningful sequences. Before modelling we need to shuffle the data and put it into bathces.

#I’m defining batch size here

BATCH\_SIZE = 64

#We need buffer size to shuffle the dataset tensorflow data maintains a buffer to shuffle elements

BUFFER\_SIZE = 10000

dataset = dataset.shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE, drop\_remainder=True)

dataset

#Now it is the time that I need to build my model

#so I need to define length of the vocabulary in chars, embedding dimension and number of RNN units

vocab\_size = len(vocab)

embedding\_dim = 256

rnn\_units = 1024

#tf.keras.Sequential is going to define the model. The input layer is tf.keras.layers.Embedding and it will train and map the numbers to a vector with embedding\_dig dimension.

#tf.keras.layers.GRU is a type of RNN size units=rnn\_units.

#tf.keras.layers.Dense is the output layer with vocab\_size outputs.

def build\_model(vocab\_size, embedding\_dim, rnn\_units, batch\_size):

  model = tf.keras.Sequential([

    tf.keras.layers.Embedding(vocab\_size, embedding\_dim,

                              batch\_input\_shape=[batch\_size, None]),

    tf.keras.layers.GRU(rnn\_units,

                        return\_sequences=True,

                        stateful=True,

                        recurrent\_initializer='glorot\_uniform'),

    tf.keras.layers.Dense(vocab\_size)

  ])

  return model

model = build\_model(

  vocab\_size = len(vocab),

  embedding\_dim=embedding\_dim,

  rnn\_units=rnn\_units,

  batch\_size=BATCH\_SIZE)

#It is time to try my model so I need to check shape of my output

for input\_example\_batch, target\_example\_batch in dataset.take(1):

  example\_batch\_predictions = model(input\_example\_batch)

  print(example\_batch\_predictions.shape, "# (batch\_size, sequence\_length, vocab\_size)")

# here length of input is 100 but model can be run on any length (I will Show the results on Results and Discussions section)

model.summary()

#To get real predictions from the model we need to sample from the output to get actual character indices.

sampled\_indices = tf.random.categorical(example\_batch\_predictions[0], num\_samples=1)

sampled\_indices = tf.squeeze(sampled\_indices,axis=-1).numpy()

#I need to see the timestep, a prediction of the next character index

sampled\_indices

#By this untrained model we decode these to see the text predicted.

print("Input: \n", repr("".join(idx2char[input\_example\_batch[0]])))

print()

print("Next Char Predictions: \n", repr("".join(idx2char[sampled\_indices ])))

#Now I’m going to train the model. We are able to say that it is like standart classiffication problem.

#I had to attach an optimizer and a loss function. tf.keras.losses.sparse\_categorical\_crossentropy is my loss function.Cause of my model returns logits I had to set the from\_logits flag.

def loss(labels, logits):

  return tf.keras.losses.sparse\_categorical\_crossentropy(labels, logits, from\_logits=True)

example\_batch\_loss  = loss(target\_example\_batch, example\_batch\_predictions)

print("Prediction shape: ", example\_batch\_predictions.shape, " # (batch\_size, sequence\_length, vocab\_size)")

print("scalar\_loss:      ", example\_batch\_loss.numpy().mean())

#tf.keras.optimizers.Adam has default arguments and loss function so we configure the training procedure using the tf.keras.Model.compile method.

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

#So Im creating a directory where the checkpoints will be saved

checkpoint\_dir = './training\_checkpoints'

#Name of that checkpoint files

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt\_{epoch}")

checkpoint\_callback=tf.keras.callbacks.ModelCheckpoint(

    filepath=checkpoint\_prefix,

    save\_weights\_only=True)

#here I’m executing the training.Epochs is our referance so It will be doing this 10 times.

EPOCHS=10

history = model.fit(dataset, epochs=EPOCHS, callbacks=[checkpoint\_callback])

#as far as it is finished I need to generate a text

tf.train.latest\_checkpoint(checkpoint\_dir)

#I’m setting the batch size 1 to keep prediction step simple

model = build\_model(vocab\_size, embedding\_dim, rnn\_units, batch\_size=1)

model.load\_weights(tf.train.latest\_checkpoint(checkpoint\_dir))

model.build(tf.TensorShape([1, None]))

model.summary()

#this part is so important cus here I’m setting up a prediction loop so it immitates and make pharagraphs Shakespeare like writing vocabulary.

def generate\_text(model, start\_string):

  #Evaluation step (generating text using the learned model)

  #Number of characters to generate

  num\_generate = 1000

  #Converting our start string to numbers (I’m doing vectorizing here)

  input\_eval = [char2idx[s] for s in start\_string]

  input\_eval = tf.expand\_dims(input\_eval, 0)

  #Empty string to store our results

  text\_generated = []

  #Low temperatures give results in more predictable text.

  #Higher temperatures give results in more surprising text.

  temperature = 1.0

  #Here I put the batch size 1

  model.reset\_states()

  for i in range(num\_generate):

      predictions = model(input\_eval)

      #I’m removing the batch dimension

      predictions = tf.squeeze(predictions, 0)

      # We use a categorical distribution to predict the character returned by the model

      predictions = predictions / temperature

      predicted\_id = tf.random.categorical(predictions, num\_samples=1)[-1,0].numpy()

      #We pass the predicted character as the next input to the model along with the previous hidden state

      input\_eval = tf.expand\_dims([predicted\_id], 0)

      text\_generated.append(idx2char[predicted\_id])

  return (start\_string + ''.join(text\_generated))

#finally I’m printing the generated text Shakespeare like writing result here

print(generate\_text(model, start\_string=u"ROMEO: "))

1. **Results and Discussions**

**First 1000 characters of dataset**

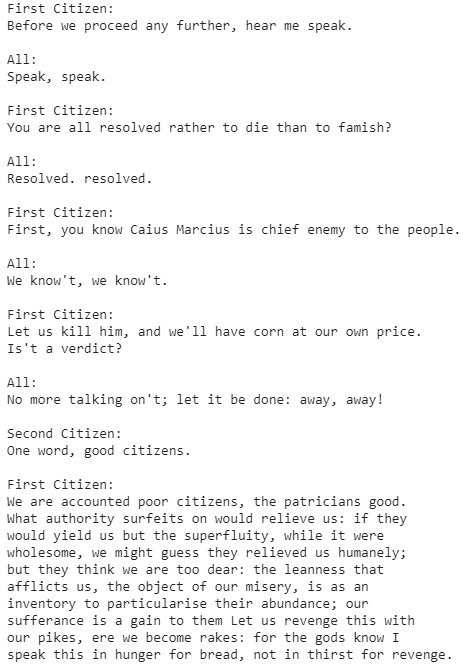


Figure 1

**Mapped characters as indexes from 0 to len(unique) (we have 65 unique characters)**

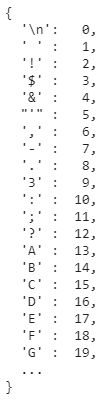


Figure 2

**First 13 characters from the text are mapped to integers**



Figure 3

**Training Example**

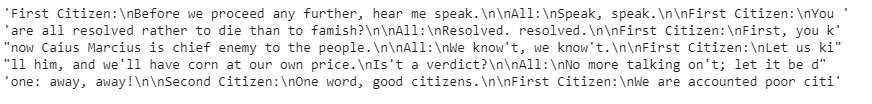


Figure 4

**First Example Input and Target Value**



Figure 5

**One Time Step Results**

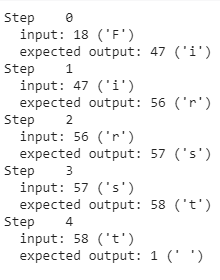


Figure 6

**Model Summary**

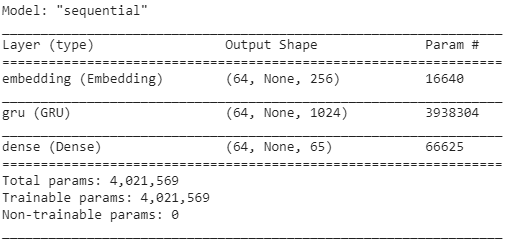


Figure 7

**Prediction Of Next Character Index**

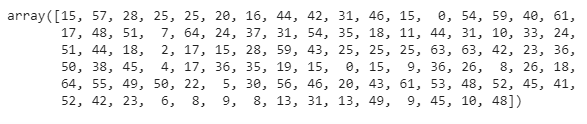


Figure 8

**Decode Of Text Predicted By Untrained Model**

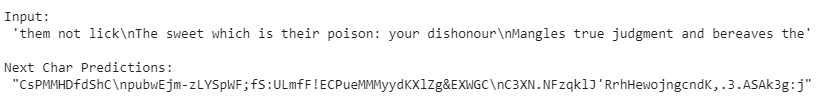


Figure 9

**Epochs Loss Results**

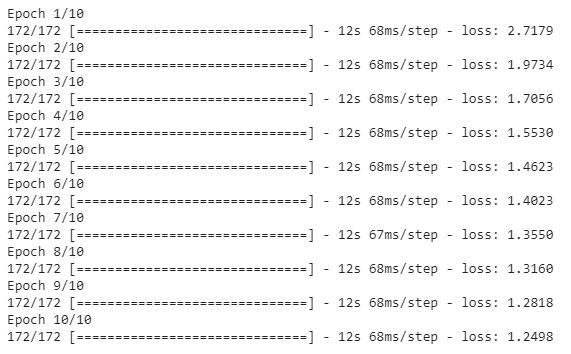


Figure 10

**Checkpoint Model Summary**

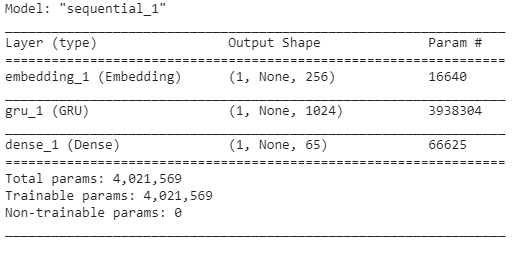


Figure 11

**Final Result Of The Generated Shakespeare Like Style Text**

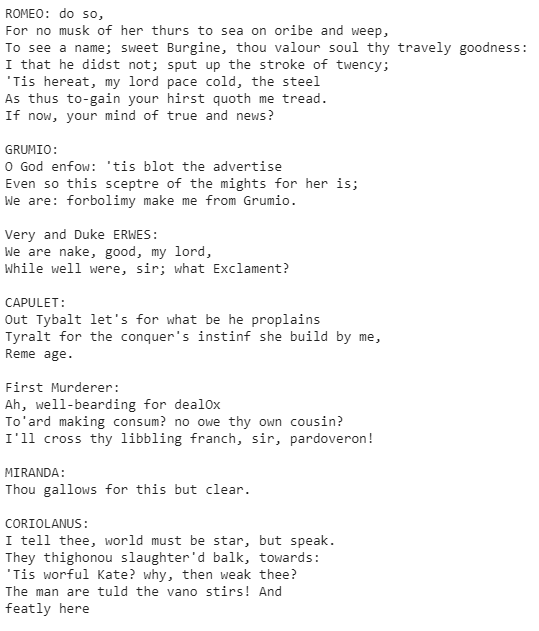


Figure 12

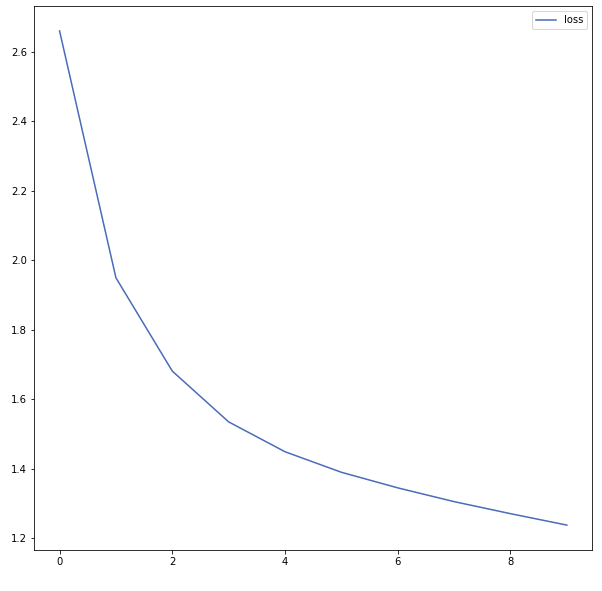
**Result**

To sum up that experiment code, part generates the text like that, It starts by choosing a start string then It initialize the RNN and set the number of characters to generate.It gets the prediction distribution of the next char using the first string and the RNN state.Then algorithm uses a categoric distribution to calculate the index of predicted character.It uses that character as our next input to the model. And RNN state returned by the model is fed back into the model so that it now has more context, instead than only one character.After predicting the next character, the modified RNN states are again fed back into the model, which is how it learns as it gets more context from previously predicted characters.

When you look at the generated text you will see the model knows when to capitalize, make paragraphs and imitates a Shakespeare like writing vocabulary with a small number of training .If you want to improve the results , you have to train it longer like “EPOCHS=30”. You are also able to experiment with a different start string, or try adding another RNN layer to improve the results.

Let’s make process a bit simple. Firstly, we set up the Tensorflow and other libraries.Then download the Shakespeare dataset.As soon as I download the dataset I read the data and displayed results.After I had to process to text so I vectorized the text then I created a prediction task. These weren’t enough so I created a training example and target. Tensorflow is working with batch to train so that I created a training batch with 100 batch\_size. Then I built the model and tried it out.How I trained it ? I attached an optimizer and a loss function and configure the checkpoints then executed the training. Finally, I restored the checkpoint and made a prediction loop to generate a Shakespeare-like written text.

**Visualising Loss Function**



Actually when you genereate text you don’t need to visualise or use mode lplot but I wanted to see the loss function decrease by time on the graphic.To see that I just added the libraries which called pandas:

import pandas as pd

import matplotlib.pyplot as plt

Then I set the size of it and added the loss function on it :

plt.figure(figsize=(10,10))

plt.plot(history.history["loss"])

plt.legend(["loss"])

plt.show()

In the end I get the graph as above.

**It was a great opportunity and exciting to work with deep learning. It expanded my horizon. Thanks to these two tasks I decided to advance in my career as Data Scientist and kind regards to my Machine Learning lecturer Assoc. Prof. Umut ORHAN to give me that opportunity.**

1. **References**

* (**Google Shakespeare Plays Dataset**) <https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt>
* <https://www.tensorflow.org/guide/checkpoint>
* <https://www.tensorflow.org/guide/keras/overview>
* <https://www.thepythoncode.com/article/text-generation-keras-python>
* <https://towardsdatascience.com/learn-how-recurrent-neural-networks-work-84e975feaaf7>
* <https://en.wikipedia.org/wiki/Recurrent_neural_network>
* Assoc. Prof. Dr. Umut ORHAN’s PHD Machine Learning Slides