|  |  |  |
| --- | --- | --- |
|  | **MEENAKSHI SUNDARARAJAN ENGINEERING COLLEGE**  **Kodambakkam, Chennai-600024** |  |

**NM1009-GENERATIVE AI FOR ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**TOPIC: TEXT\_GENERATION\_WITH\_RNNS**

|  |  |
| --- | --- |
| **FACULTY MENTOR:** | **Mrs.P.REVATHI** |
|  |  |

**Project submitted by,**

**Prasanth S (311521104305)**

***CONTENTS***

1. **ABSTRACT**
2. **INTRODUCTION**
   1. Project Overview
   2. Purpose
3. **IDEATION AND PROPOSED SOLUTION**
   1. Problem statement definition
   2. Ideation and Brainstorming
   3. Proposed Solution
4. **REQUIREMENTS ANALYSIS**
   1. Functional Requirements
   2. Non-Functional Requirements
5. **PROJECT DESIGN** 
   1. Briefing
   2. Solution
6. **SOURCE CODE**
7. **OUTPUT**
8. **RESULTS**

7.1 Performance Metrics

1. **ADVANTAGES AND DISADVANTAGES**
2. **CONCLUSION**
3. **FUTURE SCOPE**

APPENDIX

**1. ABSTRACT:**

This project utilizes TensorFlow and LSTM networks to generate text similar to the style of Shakespeare's writing. The project begins by downloading and processing a text file containing Shakespearean text. The text is then converted into a numerical format using a tokenizer. The model architecture consists of an embedding layer followed by two LSTM layers and a dense output layer.

The model is trained using sparse categorical crossentropy loss and the Adam optimizer. The training process is monitored using checkpoints to save the model weights at regular intervals. After training, the model is used to generate text based on a given input sequence.

Different temperatures are applied during text generation to control the creativity of the generated text. The generated text demonstrates the model's ability to mimic Shakespearean writing style.

**2. INTRODUCTION**

The goal of this project is to train a character-level language model using a Long Short-Term Memory (LSTM) neural network to generate text similar to the style of William Shakespeare. The model is trained on the text of Shakespeare's works downloaded from a public dataset. Character-level language models have the advantage of capturing the syntax and style of the text at a finer level than word-level models.

**2.1 Project Overview**

**Methods**

**Data Collection:** The text of Shakespeare's works is downloaded from a public URL and preprocessed to convert it into a format suitable for training the LSTM model.

**Data Tokenization:** The text is tokenized into characters, and each character is mapped to a unique integer value. This is done to convert the text into a format that can be fed into the LSTM model.

**Model Architecture:** The LSTM model is constructed using the TensorFlow and Keras libraries. The model consists of an embedding layer, two LSTM layers with 512 units each, and a dense layer with softmax activation function to predict the next character in the sequence.

**Model Training:** The model is trained using the tokenized text data. The training is done for 2 epochs with a batch size of 64 and the Adam optimizer. The Sparse Categorical Crossentropy loss function is used to calculate the loss.

**Text Generation:** After training, the model is used to generate text. A seed text is provided, and the model predicts the next character in the sequence based on the seed text. The predicted character is then added to the seed text, and the process is repeated to generate a sequence of characters.

**2.2 Purpose**

The purpose of this project is to generate text similar to Shakespeare's writing using a character-based LSTM (Long Short-Term Memory) model. The model is trained on the text of Shakespeare's works to learn the patterns and style of his writing, and then it is used to generate new text that resembles his work.

**3. IDEATION AND PROPOSED SOLUTION**

**3.1 Problem statement definition**

The main objective of this project is to develop a model that can generate text in the style of Shakespeare's writings. This involves learning the patterns and structures present in Shakespeare's works and using this knowledge to generate new text that mimics his writing style.

**3.2 Ideation and Brainstorming**

The idea for this project stemmed from a desire to explore text generation using deep learning techniques. After researching various approaches, we decided to focus on using LSTM networks due to their ability to capture long-term dependencies in sequential data.

**3.3 Proposed Solution**

To achieve this goal, we plan to use an LSTM-based neural network. The text data will be tokenized and converted into a numerical format, which will then be used as input to the LSTM network. The network will be trained to predict the next character in a sequence of characters, allowing it to generate text one character at a time.

**Implementation Overview**

**Tokenization:** Convert the text data into numerical tokens using a vocabulary mapping.

**Model Architecture:** Use an LSTM network with two layers of 512 units each, followed by a dense layer for output.

**Training:** Train the model using the Adam optimizer and sparse categorical cross-entropy loss.

**Text Generation:** Use the trained model to generate text by predicting the next character based on the input sequence.

**Challenges and Considerations**

One of the main challenges of this project is handling the long-term dependencies present in Shakespeare's writings. The use of LSTM networks helps mitigate this issue by allowing the model to remember information from earlier in the text.

**4. REQUIREMENTS ANALYSIS:**

**Objective:** Build a character-level language model to generate text similar to Shakespeare's writing style.

**Data Source:** Utilize the "shakespeare.txt" dataset containing Shakespeare's works.

**Preprocessing:** Convert the raw text data into a format suitable for training the language model, including tokenization and creating sequences of characters.

**Model Architecture:** Use an LSTM-based neural network architecture for sequence generation. The model should be capable of predicting the next character in a sequence.

**Training:** Train the model using the prepared dataset, optimizing it to predict the next character accurately.

**Evaluation:** Evaluate the model's performance using metrics like loss during training and qualitative assessment of generated text.

**Hyperparameters:** Experiment with hyperparameters such as batch size, sequence length, and LSTM units to optimize model performance.

**Model Saving:** Implement a method to save the trained model weights for future use.

**Text Generation:** Use the trained model to generate text starting from a given input sequence, allowing for variations in generated text through a temperature parameter.

**Visualization:** Visualize the model architecture using tools like keras.utils.plot\_model.

**Checkpointing:** Implement a checkpointing mechanism to save the model's weights during training to resume training or use the trained model later.

**Requirements:** Ensure the code requirements are clearly defined, including TensorFlow, NumPy, and matplotlib versions, for easy replication of the environment.

**4.1 Functional Requirements:**

1. The system shall download a text file containing Shakespearean text from the provided URL.
2. The system shall decode the downloaded text file into UTF-8 format.
3. The system shall create a vocabulary list from the unique characters in the text file.
4. The system shall tokenize each character in the text file using the created vocabulary.
5. The system shall create sequences of tokens for training the model, with each sequence consisting of 100 tokens.
6. The system shall prepare a dataset for training the model by mapping each input sequence to its corresponding target sequence.
7. The system shall batch the dataset into batches of 64 sequences.
8. The system shall prefetch the batches of data for optimized performance during training.
9. The system shall construct a sequential model using TensorFlow and Keras with the following layers:

* An embedding layer with input dimension equal to the vocabulary size and output dimension of 64.
* Two LSTM layers with 512 units each, returning sequences and maintaining statefulness.
* A dense layer with output dimension equal to the vocabulary size.

1. The system shall compile the model using the Adam optimizer and sparse categorical crossentropy loss.
2. The system shall train the model for a specified number of epochs, with a callback to save model weights after each epoch.
3. The system shall plot the training loss over epochs for performance analysis.
4. The system shall reset the model states after training for generating new text.
5. The system shall generate text based on a given input string using the trained model and a specified temperature value.
6. The system shall repeat the text generation process for a specified number of characters.
7. The system shall output the generated text based on the input string and temperature value.

**4.2 Non-Functional Requirements:**

**Performance:** Specify the expected response times for various operations, such as data preprocessing, model training, and generation of text predictions. Additionally, define the expected throughput or number of requests the system should handle concurrently.

**Scalability:** Describe how the system should scale when the workload increases. For example, specify whether the model should be able to handle larger datasets or higher numbers of concurrent users.

**Reliability:** Define the system's expected uptime and how it should handle errors or failures. Specify any requirements for fault tolerance, such as the ability to recover from crashes or data corruption.

**Security:** Specify any security requirements, such as data encryption, access control, and protection against unauthorized access or malicious attacks.

**Maintainability:** Describe how the code should be structured and documented to make it easier to maintain and extend in the future. Specify any requirements for code readability, modularity, and documentation standards.

**Portability:** Specify the platforms or environments the code should be able to run on, such as specific operating systems or hardware configurations.

**Usability:** Describe any usability requirements, such as the user interface design, accessibility features, and support for multiple languages or user preferences.

**Compliance:** Specify any legal or regulatory requirements the system should comply with, such as data protection laws or industry standards.

**5. PROJECT DESIGN**

**5.1 Briefing**

**Objective:** To generate text in the style of Shakespeare's writings using LSTM.

**Input:** A text file containing Shakespeare's writings.

**Output:** Generated text in the style of Shakespeare's writings.

**Tools:** Python, TensorFlow, Keras, numpy, matplotlib.

5.2 **Solution Design**

1. **Data Collection:**

* Download the text file containing Shakespeare's writings.
* Load the text file and preprocess it to convert characters to integers.

1. **Data Preparation:**

* Create a TensorFlow dataset from the integer vector.
* Split the dataset into input and target sequences.

1. **Model Architecture:**

* Use an Embedding layer to convert integers to dense vectors.
* Add two LSTM layers with 512 units each for sequence processing.
* Use a Dense layer with the vocabulary size for output.

1. **Model Training:**

* Compile the model with Sparse Categorical Cross entropy loss and Adam optimizer.
* Train the model on the dataset with a specified number of epochs and steps per epoch.
* Use Model Checkpoint callback to save the model weights after each epoch.

1. **Text Generation:**

* Provide a starting sequence as input to the trained model.
* Generate new characters by sampling from the model's output probabilities.
* Repeat the process to generate a sequence of desired length.

1. **Performance Analysis:**

* Plot the loss vs. epochs to analyze the training performance.

**6. SOURCE CODE**

import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.layers import Dense, LSTM, Embedding

import matplotlib.pyplot as plt

import os

file\_URL = "<https://storage.googleapis.com/download.tensorflow.org/data/shakespeare.txt>"

file\_name= "shakespeare.txt"

path = keras.utils.get\_file(file\_name, file\_URL)

raw = open(path, 'rb').read()

print(raw[250:400])

text = raw.decode(encoding='utf-8')

print(text[250:400])

len(text)

vocabulary = np.array(sorted(set(text)))

len(vocabulary)

tokenizer = {char:i for i,char in enumerate(vocabulary)}

for i in range(20):

char = vocabulary[i]

token = tokenizer[char]

print('%4s : %4d'%(repr(char),token))

vector = np.array([tokenizer[char] for char in text])

print('\nSample Text \n')

print('-'\*70)

print(text[:100])

print('-'\*70)

print('\n\nCorresponding Integer Vector \n')

print('-'\*70)

print(vector[:100])

print('-'\*70)

vector = tf.data.Dataset.from\_tensor\_slices(vector)

sequences = vector.batch(100, drop\_remainder=True)

def prepare\_dataset(seq):

input\_vector = seq[:-1]

target\_vector = seq[1:]

return input\_vector, target\_vector

dataset = sequences.map(prepare\_dataset)

for inp, tar in dataset.take(1):

print(inp.numpy())

print(tar.numpy())

inp\_text = ''.join(vocabulary[inp])

tar\_text = ''.join(vocabulary[tar])

print(repr(inp\_text))

print(repr(tar\_text))

len(sequences)//64

AUTOTUNE = tf.data.AUTOTUNE

data = dataset.batch(64, drop\_remainder=True).repeat()

data = data.prefetch(AUTOTUNE)

STEPS\_PER\_EPOCH = len(sequences)//64

for inp, tar in data.take(1):

print(inp.numpy().shape)

print(tar.numpy().shape)

model = keras.Sequential([

Embedding(len(vocabulary), 64, batch\_input\_shape=[64,None]),

LSTM(512, return\_sequences=True, stateful=True),

LSTM(512, return\_sequences=True, stateful=True),

Dense(len(vocabulary))

])

model.summary()

keras.utils.plot\_model(model, show\_shapes=True, dpi=64)

keras.utils.plot\_model(model, show\_shapes=False, dpi=64)

for example\_inp, example\_tar in data.take(1):

example\_pred = model(example\_inp)

print(example\_tar.numpy().shape)

print(example\_pred.shape)

ids = tf.random.categorical(example\_pred[0], num\_samples=1)

ids.shape

ids[0][-1].numpy()

checkpoint\_path = os.path.join("./checkpoints", "ckpt\_{epoch}")

checkpoint\_callback = keras.callbacks.ModelCheckpoint(filepath=checkpoint\_path, save\_weights\_only=True)

model.compile(optimizer='adam',

loss=keras.losses.SparseCategoricalCrossentropy(from\_logits=True))

history = model.fit(data,

epochs=2,

steps\_per\_epoch=STEPS\_PER\_EPOCH,

callbacks=[checkpoint\_callback])

plt.plot(history.history['loss'], '+-y')

plt.title('Performance Analysis', size=16, color='green')

plt.xlabel('Epochs', size=14, color='blue')

plt.ylabel('Loss', size=14, color='blue')

plt.xticks(range(10))

plt.show()

model.reset\_states()

sample = 'ANTHONIO:'

sample\_vector = [tokenizer[s] for s in sample]

predicted = sample\_vector

sample\_tensor = tf.expand\_dims(sample\_vector, 0)

sample\_tensor = tf.repeat(sample\_tensor, 64, axis=0)

temperature = 0.6

for i in range(1000):

pred = model(sample\_tensor)

pred = pred[0].numpy()/temperature

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

predicted.append(pred)

sample\_tensor = predicted[-99:]

sample\_tensor = tf.expand\_dims([pred],0)

sample\_tensor = tf.repeat(sample\_tensor, 64, axis=0)

pred\_char = [vocabulary[i] for i in predicted]

generated = ''.join(pred\_char)

print(generated)

sample = 'ANTHONIO:'

sample\_vector = [tokenizer[s] for s in sample]

predicted = sample\_vector

sample\_tensor = tf.expand\_dims(sample\_vector, 0)

sample\_tensor = tf.repeat(sample\_tensor, 64, axis=0)

temperature = 0.8

for i in range(1000):

pred = model(sample\_tensor)

pred = pred[0].numpy()/temperature

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

predicted.append(pred)

sample\_tensor = predicted[-99:]

sample\_tensor = tf.expand\_dims([pred],0)

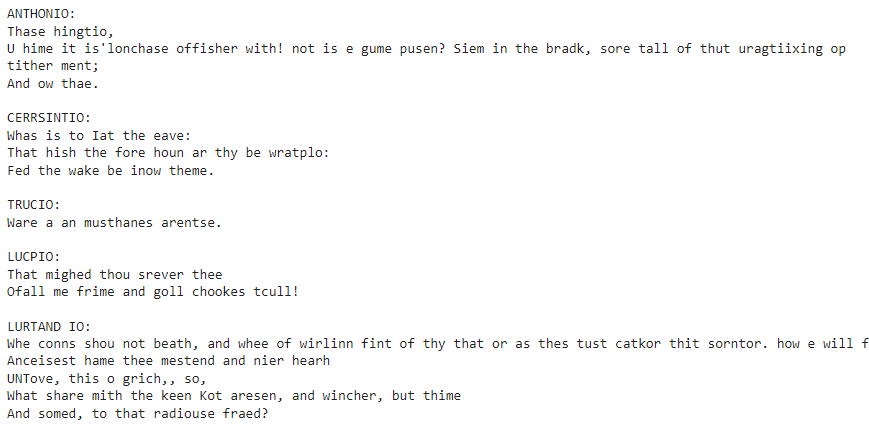
sample\_tensor = tf.repeat(sample\_tensor, 64, axis=0)

pred\_char = [vocabulary[i] for i in predicted]

generated = ''.join(pred\_char)

print(generated)

**7. OUTPUT**



**8. RESULTS**

**Code Summary:** The code fetches Shakespeare's text, preprocesses it into a character-level dataset, and then trains an LSTM model to generate text character by character.

**Result:** After training the model, you can use it to generate text. Two samples are provided with different temperatures (0.6 and 0.8) to control the randomness of the generated text.

Here's a summary of the process:

* Fetch Shakespeare's text and preprocess it into a character-level dataset.
* Define an LSTM model for text generation.
* Train the model using the dataset.
* Generate text using the trained model with different temperatures.

**9. ADVANTAGES AND DISADVANTAGES**

**Advantages:**

**Clear Implementation:** The code is well-structured and easy to understand, making it suitable for learning purposes.

**Use of TensorFlow:** Utilizes TensorFlow and its high-level APIs like tf.data.Dataset and tf.keras for building and training the model, making it efficient and scalable.

**Model Checkpointing:** Includes model checkpointing to save the model weights during training, which is useful for resuming training or for later use.

**Performance Analysis:** Plots the loss over epochs, providing a visual representation of the model's performance during training.

**Disadvantages:**

**Complexity:** The code may be complex for beginners, especially those unfamiliar with TensorFlow and its APIs.

**Resource Intensive:** Training a model like this can be resource-intensive, requiring a GPU for faster training.

**Data Preprocessing:** The code assumes that the data is already pre-processed (e.g., tokenized), which may not be suitable for all datasets.

**Limited Error Handling:** There is limited error handling in the code, which may make it challenging to debug issues that arise during training.

**10. CONCLUSION**

Finally, you used the trained model to generate text by providing a starting sequence ('ANTHONIO:') and predicting the next characters based on the model's output probabilities. You experimented with different temperatures (0.6 and 0.8) to control the randomness of the generated text.

**11. FUTURE SCOPE:**

1. Experiment with different model architectures, such as adding more LSTM layers or using different sizes of embeddings.
2. Explore different hyperparameters, like learning rate and batch size, to see their effects on the model's performance.
3. Implement techniques to improve model training, such as using learning rate schedules or early stopping.
4. Consider using a larger dataset or combining multiple datasets to train the model on a more diverse range of text.
5. Experiment with different text generation strategies, such as beam search or nucleus sampling, to improve the quality of generated text.
6. Explore the use of pre-trained embeddings or language models, such as BERT or GPT, to see if they can improve the model's performance.
7. Implement a more sophisticated evaluation metric to assess the quality of the generated text, such as BLEU score or perplexity.
8. Consider deploying the model as a web service or integrating it into a larger application for interactive text generation.

**APPENDIX:**

Source code @github: <https://github.com/Prasanthsekar3794/Text_Generation-With-RNN.git>