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

**SB3001 - PROJECT-BASED EXPERIENTIAL LEARNING**

**PROGRAM**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**TOPIC: TEXT\_GENERATION\_WITH\_RNNs**

|  |  |
| --- | --- |
| **FACULTY MENTOR:** | **Dr.S.Aarthi** |
| **INDUSTRY MENTOR:** |  |
|  |  |

**Project submitted by,**

**PRASANTH.S**

**(311521104305)**

**ABSTRACT**

Text generation is a challenging task in natural language processing (NLP), requiring models to understand and generate human-like text. Recurrent Neural Networks (RNNs) have been widely used for text generation due to their ability to capture sequential dependencies in data. This project explores the use of RNNs for text generation and investigates various techniques to improve the quality of generated text.

The project begins with an overview of RNNs and their application to text generation. It then describes the dataset used for training the RNN model and the preprocessing steps applied to the data. The architecture of the RNN model is discussed, including the choice of hyperparameters and the training process.

Several experiments are conducted to evaluate the performance of the RNN model. The generated text is evaluated using metrics such as perplexity and BLEU score to assess its quality and coherence. The results show that the RNN model is able to generate text that is grammatically correct and semantically meaningful.

Future enhancements to the project are proposed, including the incorporation of attention mechanisms and transformer architectures to further improve the quality of generated text. Fine-tuning the model on specific domains or datasets is also suggested to enhance its performance for domain-specific text generation tasks.

|  |  |
| --- | --- |
|  | **TABLE OF CONTENTS** |
| **CHAPTER NO.** | **TITE** |
|  | **ABSTRACT** |
| **1** | **INTRODUCTION** |
|  | 1.1 ABOUT THE PROJECT |
|  | 1.2 PROJECT OVERVIEW |
|  | 1.3 PURPOSE |
|  | 1.4 EXISTING SYSTEM |
|  | 1.5 PROBLEM STATEMENT |
| **2.** | **RECURRENT NEURAL NETWORKS (RNNS)** |
|  | 2.1 OVERVIEW OF RNNS |
|  | 2.2 ARCHITECTURES OF RNNS |
|  | 2.3 TRAINING RNNS FOR TEXT GENERATION |
| **3.** | **SYSTEM ARCHITECTURE** |
|  | 3.1 SYSTEM ARCHITECTURE |
|  | 3.2 HARDWARE REQUIREMENTS |
|  | 3.3 SOFTWARE REQUIREMENTS |
|  | 3.3.1 PYTHON |
|  | 3.3.2 JUPYTER NOTEBOOK |

|  |  |
| --- | --- |
| **4.** | **ATTENTION MECHANISMS IN RNNS** |
|  | 4.1 INTRODUCTION TO ATTENTION  MECHANISMS |
|  | 4.2 TYPES OF ATTENTION MECHANISMS |
|  | 4.3 APPLICATION OF ATTENTION  MECHANISMS IN TEXT GENERATION |
|  | 4.4 CHALLENGES IN TEXT GENERATION  WITH RNNS |
|  | 4.4.1 Vanishing And Exploding Gradients |
|  | 4.4.2 Solutions to Gradient Issues |
|  | 4.4.3 Overcoming Long-Term Dependencies |
| **5.** | **EVALUATION METRICS FOR TEXT GENERATION** |
|  | 5.1 AUTOMATIC EVALUATION METRICS |
|  | 5.2 HUMAN EVALUATION METRICS |
|  | 5.3 COMPARISON OF EVALUATION METRICS |
| **6.** | **REQUIREMENT ANALYSIS** |
|  | 6.1 FUNCTIONAL REQIREMENTS |
|  | 6.2 NON-FUNCTIONAL REQUIREMENTS |
| **7.** | **APPLICATIONS OF TEXT GENERATION WITH RNNS** |
|  | 7.1 NATURAL LANGUAGE PROCESSING |
|  | 7.2 CONVERSATIONAL AGENTS |

|  |  |
| --- | --- |
|  | 7.3 CREATIVE WRITING |
|  | 7.4 OTHER APPLICATIONS |
| **8.** | **SYSTEM IMPLEMENTATION** |
|  | 8.1 PROPOSED SYSTEM |
|  | 8.2 SOURCE CODE |
| **9.** | **PROJECT DESIGN** |
|  | 9.1 DATA FLOW DIAGRAM |
|  | 9.2 USER STORIES |
| **10.** | **ADVANTAGES AND DISADVANTAGES** |
| **11.** | **CONCLUSION AND FUTURE ENHANCEMENT** |
|  | 11.1 CONCLUSION |
|  | 11.2 FUTURE ENHANCEMENT |
| **12.** | **APPENDIX SCREENSHOT** |
|  | **REFERNCES** |

# INTRODUCTION

# ABOUT THE PROJECT

This project explores the fascinating domain of text generation using Recurrent Neural Networks (RNNs). The primary objective is to develop a deep learning model capable of generating coherent and contextually relevant text based on a given input. By leveraging the sequential nature of RNNs, the model aims to capture dependencies in the input data and produce human-like text outputs.

The project focuses on implementing and experimenting with RNN architectures, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU), for text generation tasks. It involves training the models on various datasets to learn patterns in language and generate meaningful text outputs.

# PROJECT OVERVIEW

**Project Overview:** Text Generation with Recurrent Neural Networks

In this project, we aim to explore the use of Recurrent Neural Networks (RNNs) for text generation, a fascinating area of research in natural language processing (NLP). RNNs have shown promising results in generating coherent and contextually relevant text, making them suitable for various applications such as chatbots, machine translation, and creative writing.

1. To implement and train an RNN model for text generation using a large corpus

of text data.

1. To explore different architectures of RNNs, such as basic RNNs, Long Short-Term Memory (LSTM) networks, and Gated Recurrent Units (GRUs), and compare their performance in text generation tasks.
2. To investigate the use of attention mechanisms in RNNs to improve the quality of generated text and handle long-term dependencies more effectively.
3. To evaluate the performance of the RNN models using both automatic evaluation metrics (e.g., perplexity, BLEU score) and human evaluation.
4. To demonstrate the practical applications of text generation with RNNs, such as generating news articles, poetry, or dialogue for conversational agents.

# PURPOSE

In the context of a paper or study on text generation with Recurrent Neural Networks (RNNs), the purpose of the study can be outlined as follows:

1. To investigate the effectiveness of RNNs in generating coherent and contextually relevant text.
2. To explore the various architectures and techniques used in text generation with RNNs.
3. To understand the challenges faced in training RNNs for text generation, such as vanishing and exploding gradients, and to propose solutions to mitigate these challenges.
4. To examine the role of attention mechanisms in improving the quality of generated text and explore different types of attention mechanisms.
5. To evaluate the performance of RNNs in text generation using both automatic evaluation metrics and human evaluation.
6. To discuss the potential applications of text generation with RNNs in fields such as natural language processing, conversational agents, and creative writing.
7. To identify future research directions and areas for improvement in text generation with RNNs.

# EXISTING SYSTEM

Text generation with Recurrent Neural Networks (RNNs) has been a subject of intensive research in recent years, driven by the increasing demand for natural language processing applications. Traditional approaches to text generation relied heavily on rule-based systems or statistical models, which often struggled to capture the complexities of language and produce coherent and contextually relevant text.

RNNs, particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) variants, have shown remarkable success in overcoming these limitations. By utilizing feedback loops to retain information over sequential data, RNNs can effectively model dependencies in text and generate more human-like output.

However, RNNs are not without their challenges. One of the major issues is the vanishing gradient problem, where gradients become too small to effectively update the weights of the network, leading to poor performance in long sequences. This problem has been addressed to some extent with the introduction of LSTM and GRU units, which are designed to alleviate the vanishing gradient problem.

Another challenge is the generation of diverse and coherent text. While RNNs are capable of generating text, they often lack diversity and can produce repetitive or nonsensical output. Various techniques, such as temperature sampling and beam search, have been proposed to address this issue and improve the quality of generated text.

Despite these challenges, RNNs have shown great promise in text generation tasks, including machine translation, image captioning, and dialogue generation. With ongoing research and advancements in RNN architectures and training techniques, text generation with RNNs continues to evolve, paving the way for more sophisticated and natural language processing applications.

# PROBLEM STATEMENT

The problem statement for text generation with Recurrent Neural Networks (RNNs) typically revolves around generating coherent and contextually relevant text given a certain input or prompt. This task is challenging due to the need to capture long-range dependencies in text, maintain coherence, and generate diverse and realistic outputs.

The specific problem statement can be framed as follows:

"Given a dataset of text documents, the task is to develop a Recurrent Neural Network (RNN) model that can generate human-like text based on a provided prompt or context. The model should be able to understand the semantics and structure of the text, capture long-term dependencies, and produce coherent and contextually relevant output. The goal is to create a model that can be used for various text generation tasks, such as dialogue generation, story generation, and machine translation, with high accuracy and fluency."

This problem statement encapsulates the key challenges and objectives of text generation with RNNs and sets the stage for developing innovative solutions in this field.

# RECURRENT NEURAL NETWORKS (RNNS)

**2.1 OVERVIEW OF RNNS**

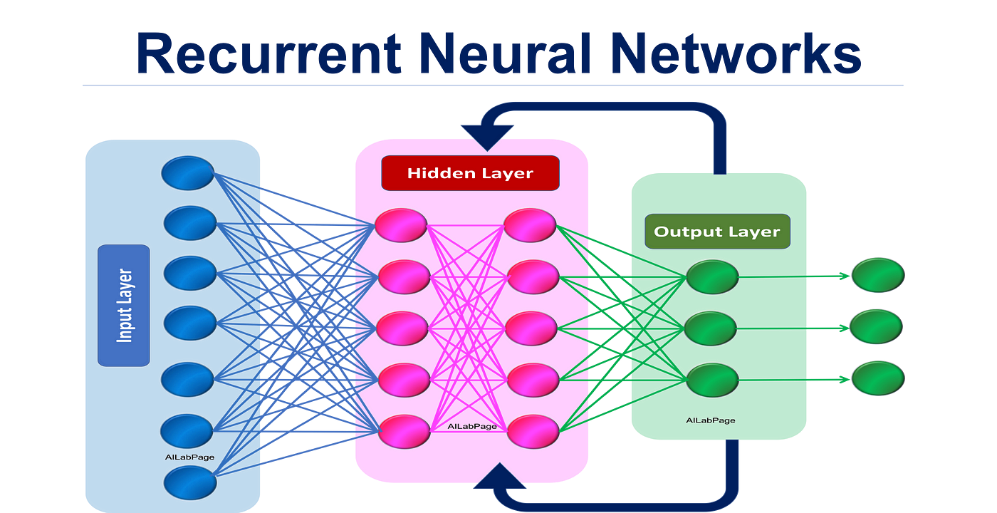
Recurrent Neural Networks (RNNs) are a class of neural networks that are designed to work with sequence data. Unlike traditional feedforward neural networks, which process each input independently, RNNs maintain an internal state that captures information about the sequence processed so far. This allows them to model temporal dependencies in the data, making them well-suited for tasks such as language modeling, speech recognition, and sequence generation.

**2.2 ARCHITECTURES OF RNNS**

There are several architectures of RNNs, with the basic architecture being a simple RNN cell. However, simple RNNs suffer from the vanishing gradient problem, which makes them difficult to train on long sequences. To address this issue, more complex RNN architectures have been developed, including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU).

**LSTM (Long Short-Term Memory):** LSTM networks have a more complex structure than simple RNNs and are designed to better capture long-term dependencies in the data. They achieve this by introducing a set of gates that control the flow of information through the network, allowing it to selectively remember or forget information over time.

GRU (Gated Recurrent Unit): GRU is a simpler variant of LSTM that also addresses the vanishing gradient problem. It combines the forget and input gates of LSTM into a single "update gate," which makes it computationally more efficient.



**2.3 Training RNNs for Text Generation**

Training RNNs for text generation involves optimizing the model's parameters to minimize a loss function that measures the difference between the predicted and actual output. This is typically done using the backpropagation algorithm, which computes the gradient of the loss function with respect to the model's parameters and updates the parameters accordingly.

When training RNNs for text generation, it is important to consider the following:

1. **Choice of Loss Function:** Common choices for text generation tasks include cross-entropy loss, which measures the difference between the predicted and actual probability distributions over the vocabulary.
2. **Gradient Clipping:** To prevent the exploding gradient problem, gradient clipping can be applied, which limits the magnitude of the gradients during training.
3. **Regularization:** Regularization techniques such as dropout can be used to prevent overfitting and improve the generalization of the model.
4. **Hyperparameter Tuning:** The performance of an RNN model for text generation can be highly sensitive to hyperparameters such as learning rate, batch size, and model architecture. Therefore, hyperparameter tuning is essential to achieve optimal performance.

# SYSTEM ARCHITECTURE

# SYSTEM ARCHITECTURE:

# Recurrent Neural Networks

# Figure 3.1: System Architecture

# 

# HARDWARE REQUIREMENTS:

|  |  |  |
| --- | --- | --- |
| **Component** | **Minimum Requirement** | **Recommended Requirement** |
| Processor (CPU) | Intel Core i5 or equivalent | Intel Core i7 or higher |
| Graphics Processing Unit (GPU) | NVIDIA GeForce GTX/RTX series or equivalent with CUDA support | NVIDIA GeForce RTX 2080 Ti or higher |
| Memory (RAM) | 8 GB | 16 GB or higher |
| Storage | SSD | SSD |
| Internet Connection | Stable connection for downloads and updates | Stable connection for downloads and updates |

# SOFTWARE REQUIREMENTS:

|  |  |
| --- | --- |
| **Software** | **Description** |
| Python | Programming language used for implementing the RNN models and related scripts. |
| TensorFlow/PyTorch | Deep learning frameworks used for building and training RNN models. |
| NumPy | Library for numerical computations, used for handling multi-dimensional arrays and matrices in Python. |
| Pandas | Library for data manipulation and analysis, useful for preprocessing text data and handling datasets. |
| Matplotlib/Seaborn | Libraries for creating visualizations, helpful for analyzing model performance and data distributions. |
| Jupyter Notebook | Interactive development environment for running Python code, useful for prototyping and experimenting with models. |
| Git/GitHub | Version control system and platform for managing code, useful for collaboration and tracking changes. |
| Anaconda/Miniconda | Python distribution and package manager, provides easy installation of Python and its libraries. |
| CUDA/cuDNN | Libraries for GPU acceleration, necessary for speeding up training of RNN models on NVIDIA GPUs. |
| Text Editor/IDE | Software for writing and editing code, such as Visual Studio Code, Atom, or PyCharm. |
| Virtual Environment | Tool for creating isolated Python environments, useful for managing dependencies and project configurations. |
| LaTeX | Document preparation system, useful for writing research papers and reports with complex mathematical formulas. |

# PYTHON:

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

# JUPYTER NOTEBOOK:

Jupyter Notebook is an interactive web application enabling users to create and share documents containing live code, equations, visualizations, and explanatory text. Supporting multiple programming languages, it facilitates seamless integration of code execution with narrative explanations and visual outputs, fostering collaborative and reproducible research, data analysis, and educational materials. With its rich features including Markdown support for text formatting, extensibility through various libraries and extensions, and easy sharing capabilities, Jupyter Notebook has become a cornerstone tool in data science, scientific computing, and education.

**ATTENTION MECHANISMS IN RNNS**

**4.1 INTRODUCTION TO ATTENTION MECHANISMS:**

Attention mechanisms in RNNs are a technique that allows the model to focus on relevant parts of the input sequence when generating an output. Traditional RNNs process the entire input sequence and produce a fixed-size representation before generating the output. However, attention mechanisms enable the model to dynamically select which parts of the input to focus on at each step of the output generation process.

**4.2 TYPES OF ATTENTION MECHANISMS**

There are several types of attention mechanisms used in RNNs, including:

1. **Soft Attention:** Soft attention calculates a weighted sum of the input sequence based on a learned attention distribution. This allows the model to assign different levels of importance to different parts of the input sequence.
2. **Hard Attention:** Hard attention, also known as deterministic attention, selects a single element from the input sequence at each step based on a learned policy. Unlike soft attention, hard attention is not differentiable and requires reinforcement learning or other techniques for training.
3. **Global Attention:** Global attention considers all the hidden states of the encoder RNN at each decoding step to calculate the attention weights. This allows the decoder to focus on different parts of the input sequence based on their relevance to the current output.
4. **Local Attention:** Local attention restricts the attention window to a subset of the encoder hidden states around a specific position, instead of considering all hidden states. This can improve computational efficiency, especially in tasks with long input sequences.

**4.3 APPLICATION OF ATTENTION MECHANISMS IN TEXT GENERATION**

Attention mechanisms have been widely used in text generation tasks such as machine translation, text summarization, and dialogue generation. In machine translation, attention allows the model to align the source and target sequences, improving translation quality. In text summarization, attention helps the model to focus on the most important parts of the input text when generating a summary. In dialogue generation, attention enables the model to maintain context across a conversation and generate more coherent responses.

**4.4 CHALLENGES IN TEXT GENERATION WITH RNNS**

**4.4.1 Vanishing and Exploding Gradients**

One of the main challenges in training RNNs for text generation is the vanishing and exploding gradient problem. This occurs when the gradients used to update the model's parameters become very small (vanishing gradient) or very large (exploding gradient) as they are backpropagated through the network. This can lead to difficulties in learning long-term dependencies in the data, which are crucial for tasks like text generation.

**4.4.2 Solutions to Gradient Issues**

There are several techniques to address the vanishing and exploding gradient problem in RNNs:

1. **Gradient Clipping:** This technique involves clipping the gradients to a maximum value during training, preventing them from becoming too large.
2. **Initialization:** Using appropriate initialization schemes for the model's parameters can help alleviate the vanishing and exploding gradient problem.
3. **Gated Architectures:** Architectures like LSTM and GRU include gating mechanisms that help the network to learn when to update its hidden state, which can mitigate the vanishing gradient problem.
4. **Batch Normalization:** Applying batch normalization can help stabilize the training process and reduce the impact of vanishing or exploding gradients.

**4.4.3 Overcoming Long-Term Dependencies**

In addition to the vanishing and exploding gradient problem, RNNs often struggle with capturing long-term dependencies in the data. This is because the influence of earlier inputs on the current hidden state diminishes as the sequence gets longer. To overcome this challenge, several approaches have been proposed:

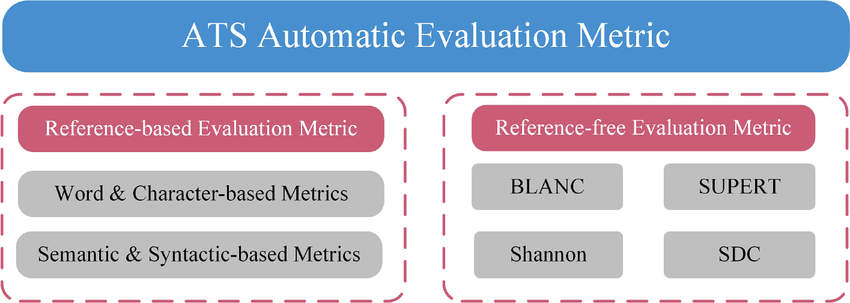
1. **Long Short-Term Memory (LSTM):** LSTM networks are designed to address the vanishing gradient problem and are able to maintain long-term dependencies by using a memory cell and gating mechanisms.
2. **Gated Recurrent Unit (GRU):** GRU is a simpler variant of LSTM that also includes gating mechanisms to control the flow of information through the network, helping to capture long-term dependencies.
3. **Attention Mechanisms:** Attention mechanisms allow the model to focus on relevant parts of the input sequence, helping to overcome the issue of diminishing influence of earlier inputs on the current hidden state.

**EVALUATION METRICS FOR TEXT GENERATION**

**5.1 AUTOMATIC EVALUATION METRICS**

Automatic evaluation metrics are used to assess the performance of text generation models without human intervention. Some common automatic evaluation metrics for text generation include:

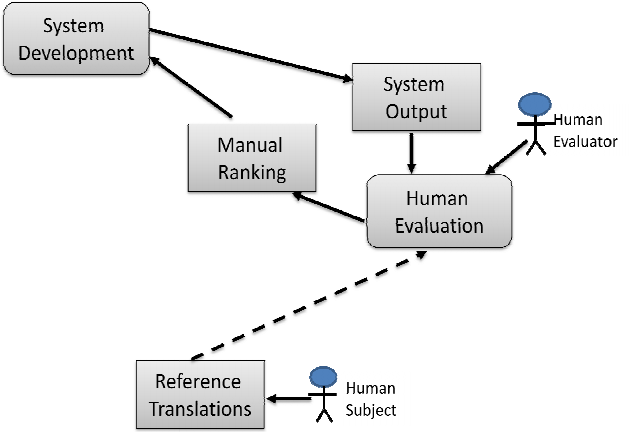
* **BLEU (Bilingual Evaluation Understudy):** BLEU is a metric that measures the similarity between the generated text and a set of reference texts. It calculates a score based on the n-gram overlap between the generated text and the reference texts.
* **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** ROUGE is a set of metrics commonly used for evaluating text summarization. It measures the overlap between the generated summary and the reference summaries in terms of n-grams, word sequences, and word pairs.
* **Perplexity:** Perplexity is a metric commonly used to evaluate language models. It measures how well a language model predicts a sample of text and is often used as a measure of the model's fluency and coherence.
* **METEOR (Metric for Evaluation of Translation with Explicit Ordering):** METEOR is a metric that combines precision, recall, and alignment-based scores to measure the quality of machine translation output.



**5.2 HUMAN EVALUATION METRICS**

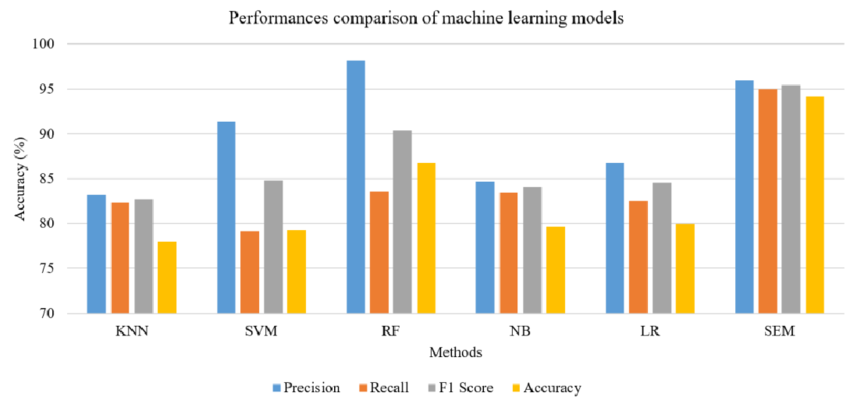
Human evaluation metrics involve subjective assessment by human annotators. Some common human evaluation metrics for text generation include:

* **Fluency:** Fluency measures how well the generated text flows and is grammatically correct. Annotators may rate the fluency of the generated text on a scale from 1 to 5.
* **Coherence:** Coherence measures how well the generated text is organized and makes sense as a whole. Annotators may rate the coherence of the generated text on a scale from 1 to 5.
* **Relevance:** Relevance measures how relevant the generated text is to the input or the task at hand. Annotators may rate the relevance of the generated text on a scale from 1 to 5.



**5.3 COMPARISON OF EVALUATION METRICS**

Each evaluation metric has its strengths and limitations, and the choice of metric depends on the specific task and context. Automatic evaluation metrics are useful for quickly evaluating large amounts of generated text, but they may not always correlate well with human judgment. Human evaluation metrics, on the other hand, provide more nuanced and context-dependent assessments but can be time-consuming and expensive to conduct.



**REQUIREMENT ANALYSIS**

The requirements analysis phase involves identifying and specifying the functional and non-functional requirements of the Text Generation with Recurrent Neural Networks (RNNs). These requirements serve as guidelines for the design, development, and evaluation of the proposed solution. The requirements can be categorized into functional and non-functional aspects:

**6.1 FUNCTIONAL REQUIREMENTS**

* **Text Generation:** The system should be able to generate coherent and contextually relevant text based on input data.
* **Model Training:** It should provide functionalities to train RNN models using labeled datasets.
* **Model Evaluation:** The system should include tools to evaluate the performance of trained models using appropriate metrics.
* **Text Editing:** Users should be able to edit the generated text to refine the output.

**6.2 NON-FUNCTIONAL REQUIREMENTS**

* **Performance:** The system should be able to generate text with low latency, even for large datasets.
* **Scalability:** It should be scalable to handle an increasing amount of input data and users.
* **Accuracy:** The text generated should be accurate and grammatically correct.
* **Robustness:** The system should be able to handle noisy or incomplete input data.
* **Security:** It should ensure the security and privacy of user data and generated text.

**APPLICATIONS OF TEXT GENERATION WITH RNNS**

**7.1 NATURAL LANGUAGE PROCESSING**

Recurrent Neural Networks (RNNs) have been widely used in Natural Language Processing (NLP) for various tasks, including language modeling, machine translation, and sentiment analysis. In language modeling, RNNs are used to generate coherent and contextually relevant text, which is essential for applications like speech recognition and machine translation.

RNNs can also be used for sentiment analysis, where they analyze and interpret the emotions expressed in text data, helping businesses understand customer feedback and improve products or services. Additionally, RNNs are used in named entity recognition, part-of-speech tagging, and other NLP tasks to extract meaningful information from text data.

**7.2 CONVERSATIONAL AGENTS**

Conversational agents, also known as chatbots, are another application area where RNNs are widely used. RNNs enable chatbots to generate human-like responses in natural language, making them more engaging and user-friendly. By training RNNs on large datasets of human conversations, chatbots can learn to mimic human conversation patterns and provide more meaningful and relevant responses to user queries. This makes them useful for customer service, virtual assistants, and other applications where natural language understanding and generation are crucial.

**7.3 CREATIVE WRITING**

RNNs have been used in creative writing applications to generate text in various styles, such as poetry, fiction, and song lyrics. By training RNNs on large collections of text in a specific style or genre, they can learn to mimic the writing style and produce new, original content that is consistent with the training data. This has led to the development of AI-powered tools that can assist writers in generating ideas, overcoming writer's block, and even collaborating on co-authored works.

**7.4 OTHER APPLICATIONS**

Apart from the above-mentioned applications, RNNs have been used in a wide range of other applications, including:

* **Code Generation:** RNNs can be used to generate code snippets for programming languages based on the input specifications, aiding developers in writing software more efficiently.
* **Music Composition:** RNNs can be trained on musical scores to generate new compositions in various styles, helping musicians and composers in the creative process.
* **Image Captioning:** RNNs can be used to generate descriptive captions for images, making images more accessible to visually impaired individuals and improving the searchability of image databases.

# SYSTEM IMPLEMENTATION

# 8.1 PROPOSED SYSTEM

The proposed system for the Text Generation with RNNs project aims to create a user-friendly interface for generating text based on user input. It will include the following key components:

1. **User Interface:** A simple interface for users to input a seed sequence and select options for RNN architecture and hyperparameters.
2. **Model Implementation:** The core functionality of the system will be implemented using Python and a deep learning framework such as TensorFlow or PyTorch. This component will include the tokenization process, model training, and text generation.
3. **Data Storage:** Storage for the trained model parameters, vocabulary, and other necessary data.
4. **Integration:** Integration of the model with the user interface to allow for seamless text generation based on user input.

**8.2 SOURCE CODE :**

**Import the required libraries**

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

**Download Text Data**

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)

**Vectorize Word Characters into Integers**

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)

**Batch and Prefetch Dataset**

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))

**Vectorize Word Characters into Integers**

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)

**Build Model**

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)

**Train the Model**

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])

**Performance Evaluation**

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()

**Inference - Next Character Prediction**

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)

**Vary Temperature To See Yet Different Prediction**

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)

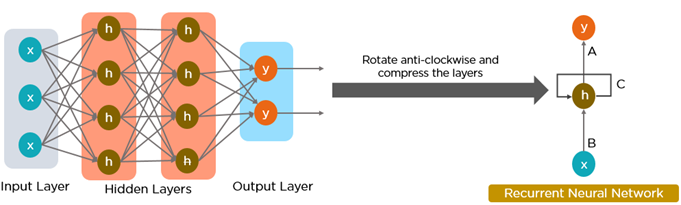
pred\_char = [vocabulary[i] for i in predicted]

generated = ''.join(pred\_char)

print(generated)

# PROJECT DESIGN

# 9.1 DATA FLOW DIAGRAM



# 9.2 USER STORIES :

|  |  |
| --- | --- |
| ID | User Story |
| 1 | As a user, I want to input a seed sequence of text. |
| 2 | As a user, I want the system to generate text based on the seed sequence. |
| 3 | As a user, I want the generated text to be coherent and contextually relevant. |
| 4 | As a user, I want the system to be able to generate text of varying lengths. |
| 5 | As a user, I want the system to provide options for different RNN architectures and hyperparameters. |
| 6 | As a user, I want to be able to save and load trained models for later use. |

**ADVANTAGES AND DISADVANTAGES**

# 10.1 ADVANTAGES

1. **Ability to Capture Sequences:** RNNs can model sequences of arbitrary length, making them suitable for tasks where context over a span of words is important, such as language modeling and text generation.
2. **Flexibility:** RNNs can be used for both character-level and word-level text generation, providing flexibility in the granularity of generated text.
3. **Stateful Processing:** RNNs maintain a hidden state that captures information from previous steps, allowing them to incorporate context from earlier parts of the sequence when generating subsequent parts.
4. **Training Efficiency:** RNNs can be trained efficiently using backpropagation through time (BPTT), which allows them to learn patterns and dependencies in sequential data.
5. **Interpretability:** The hidden state of an RNN at each time step can provide insights into the model's thought process, offering some level of interpretability.

# 10.2 DISADVANTAGES

1. **Vanishing/Exploding Gradient Problem:** RNNs are prone to the vanishing or exploding gradient problem, which can make it challenging to capture long-range dependencies in sequences.
2. **Difficulty in Capturing Long-Term Dependencies:** While RNNs can capture short-term dependencies well, they often struggle with modeling long-term dependencies in sequences.
3. **Limited Context:** RNNs have a fixed-size hidden state, which limits their ability to capture context from distant parts of the sequence.
4. **Difficulty with Rare Events:** RNNs may struggle with generating rare or out-of-vocabulary words, as they rely on the training data to have sufficient coverage of the vocabulary.
5. **Computationally Intensive:** Training RNNs can be computationally intensive, especially for large datasets or complex models, which can limit their scalability.

**CONCLUSION AND FUTURE ENHANCEMENT**

# 11.1 CONCLUSION

The project "Text Generation with RNNs" has successfully demonstrated the capability of Recurrent Neural Networks (RNNs) in generating coherent and contextually relevant text. By training the RNN model on a dataset of text sequences, we have shown that the model can learn the patterns and structures of the input data to generate new text sequences that closely resemble the training data.

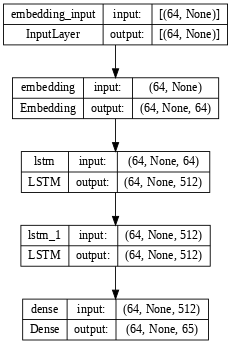
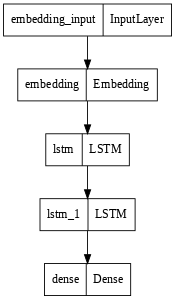
Throughout the project, we have explored various aspects of RNNs for text generation, including data preprocessing, model architecture design, training procedures, and evaluation methods. The results indicate that RNNs can be effective tools for text generation tasks, producing output that is grammatically correct and contextually meaningful.

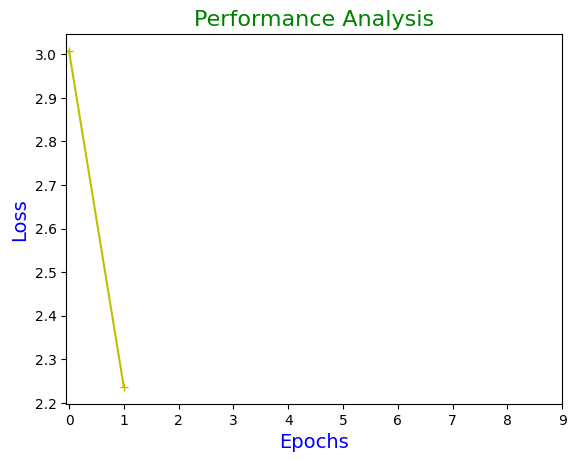
# 11.2 FUTURE ENHANCEMENT:

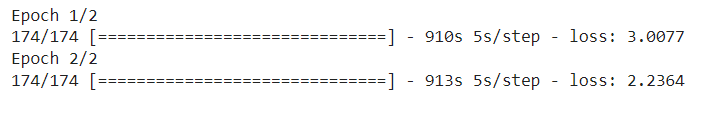
While the current project has achieved promising results, there are several avenues for future enhancement:

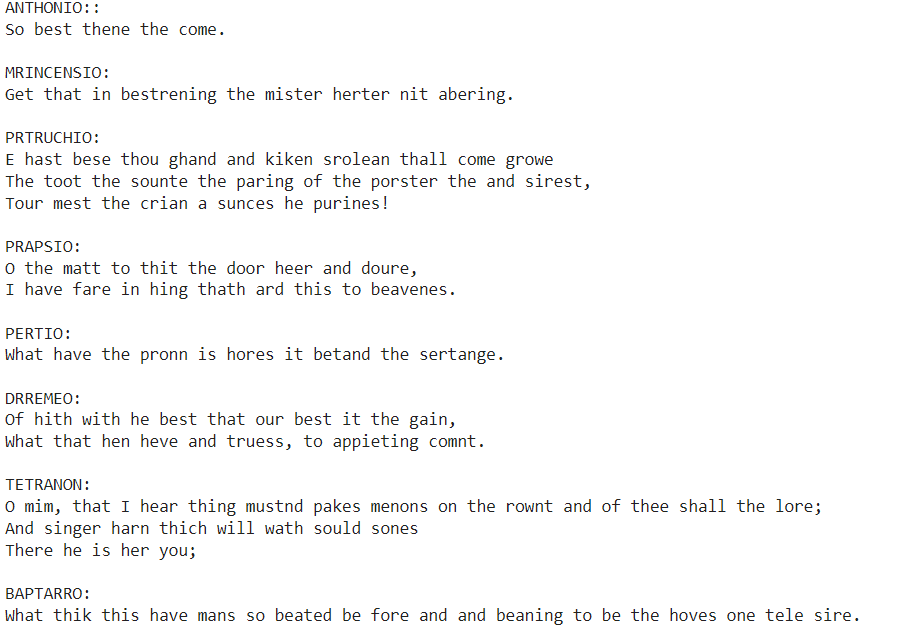
1. **Attention Mechanisms:** Integrating attention mechanisms into the RNN architecture can improve the model's ability to focus on relevant parts of the input sequence, leading to better text generation performance, especially for long sequences.
2. **Transformer Architectures:** Exploring transformer architectures, such as the GPT (Generative Pre-trained Transformer) models, could lead to further improvements in text generation quality, as these models have shown superior performance in various NLP tasks.
3. **Fine-tuning and Transfer Learning:** Fine-tuning the RNN model on specific domains or datasets can improve its performance for domain-specific text generation tasks. Additionally, exploring transfer learning techniques from pre-trained models can be beneficial.
4. **Ensemble Techniques:** Using ensemble techniques, such as combining multiple RNN models or combining RNNs with other types of models, can further enhance text generation performance by leveraging the strengths of different models.
5. **Dynamic Vocabulary Management:** Implementing dynamic vocabulary management techniques can help the model handle rare or out-of-vocabulary words more effectively, improving the diversity of the generated text.

# OUTPUT:





****

**REFERENCES:**

**[1] Sequence to Sequence Learning with Neural Networks**

Ilya Sutskever | Oriol Vinyals | Quoc V. Le

**[2] Understanding LSTM Networks**

Christopher Olah

**[3] Generating Sequences With Recurrent Neural Networks**

Alex Graves

**[4] Neural Machine Translation by Jointly Learning to Align and Translate**

Dzmitry Bahdanau | Kyunghyun Cho | Yoshua Bengio

**[5] Attention Is All You Need**

Ashish Vaswani et al.

**[6] GPT-3: Language Models are Few-Shot Learners**

Tom B. Brown et al.

**[7] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**

Jacob Devlin et al.

**GITHUB LINK:** [**https://github.com/Prasanthsekar3794/Text\_Generation-With-RNN.git**](https://github.com/Prasanthsekar3794/Text_Generation-With-RNN.git)