**Shakespearean Echo**

**Text Generation with Recurrent Neural Networks: A Case Study on Shakespeare's Dataset**

**Abstract**

This project builds a Recurrent Neural Network (RNN) model using Long Short-Term Memory (LSTM) layers to generate text resembling Shakespeare's writing. The dataset is Shakespeare's complete works, and the model aims to predict the next word in a sequence of words. The model successfully learns patterns in the text but faces overfitting, as it starts repeating stanzas after generating 60–65 words. This issue arises due to the model memorizing patterns instead of generalizing them. Various solutions like regularization, reducing the vocabulary size, and batch processing were explored to overcome this limitation.

**Objective**

The primary objective is to develop a deep learning model capable of generating coherent and contextually relevant text in the style of Shakespeare. The model should mimic the structure, language, and rhythm of the original texts, while addressing overfitting issues that lead to repetition in generated sequences. This project explores how LSTMs can help in generating longer sequences while avoiding repeated phrases or stanzas.

**Literature Survey**

Recurrent Neural Networks (RNNs) have been a popular choice for sequence generation tasks, including text generation. LSTM networks, introduced by Hochreiter and Schmidhuber (1997), were designed to solve long-term dependency problems in standard RNNs. Later studies, such as by Karpathy et al. (2015), demonstrated the success of LSTM models in generating human-like text for literature and code. Various techniques like dropout (Srivastava et al., 2014) and regularization have been proposed to combat overfitting in RNNs. Despite their success, one major issue remains: overfitting and repetition, especially when models generate long sequences.

**Problem Statement**

Generating text sequences is a challenging task, as the model needs to remember previous words while predicting the next word in the sequence. The primary problem encountered in this project is overfitting, where the model memorizes sequences from the training data instead of learning general patterns. As a result, the generated text starts repeating words or stanzas after 60–65 words, reducing the model’s creativity and generalizability. This project focuses on addressing this limitation to improve the quality of the generated text.

**Existing Issues**

The key issue is overfitting, particularly visible when the model begins generating repetitive text after a certain number of words (60–65 words in this case). Overfitting occurs because the model learns to memorize specific sequences from the training data rather than generalizing patterns. Additionally, managing long-range dependencies in text sequences is difficult for LSTMs, especially when generating creative text like Shakespeare's works.

Solutions explored to mitigate overfitting:

- Limiting Vocabulary Size: By restricting the number of unique words the model can learn, the model is forced to generalize across words instead of memorizing exact patterns.

- Batch Processing: A generator function is used to yield batches of input-output pairs during training, which helps in providing the model with varied sequences.

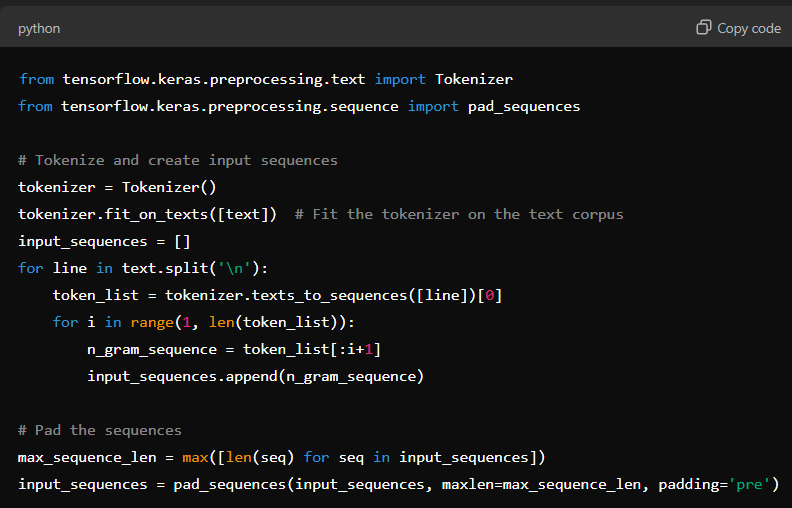
- Reduced Sequence Length: Limiting the maximum sequence length reduces the risk of overfitting to long phrases.

**Module Explanation**

The project is divided into the following modules:

1. **Data Processing Module**

The input data, which is Shakespeare's text, is split into sequences of words. Each sequence is tokenized, converting words into corresponding integer values based on a predefined vocabulary. These tokenized sequences are then padded to ensure they have the same length, which is necessary for batch processing during training.



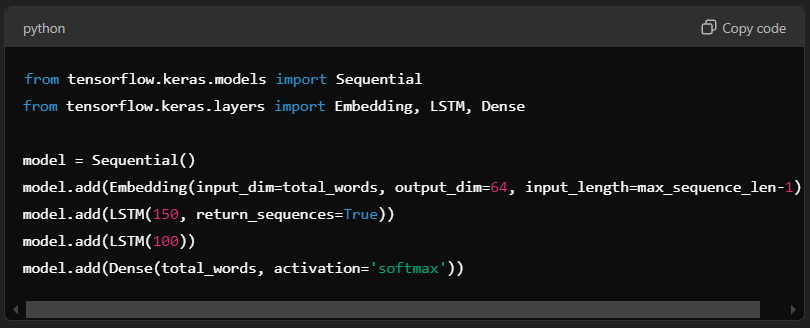
In this code:

- `texts\_to\_sequences` tokenizes the text, converting each word into a corresponding integer.

- `pad\_sequences` ensures that all input sequences have the same length by adding padding where necessary.

**2. Model Architecture**

The model is built using an embedding layer, two LSTM layers, and a dense output layer with softmax activation. The LSTM layers allow the model to capture long-range dependencies in the text.



Here:

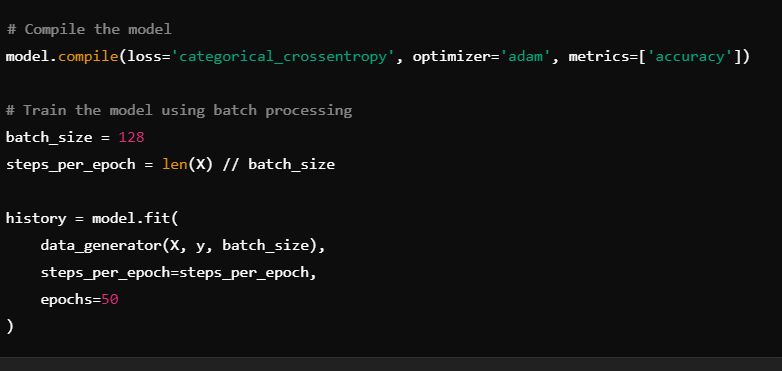
- The Embedding layer maps words to dense vectors, which are more meaningful representations.

- Two LSTM layers allow the model to learn patterns over longer sequences.

- The Dense layer with `softmax` activation is used for predicting the next word in the sequence.

**3. Training and Evaluation**

The model is compiled with categorical cross-entropy loss and the Adam optimizer. The overfitting problem is addressed by limiting the vocabulary size and the sequence length.

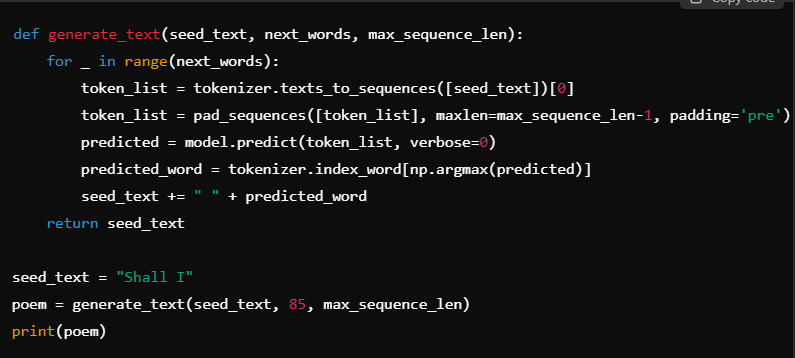


The `data\_generator` function batches the input data during training to improve the generalization of the model.

**4. Implementation with Demo**

The model was trained to predict the next word in a sequence, generating text in Shakespeare's style. However, after generating 60–65 words, the model starts repeating itself, indicating overfitting.

Code for Text Generation:



In this example:

- The function generates text by predicting the next word repeatedly based on the seed text.

- After generating a certain number of words, overfitting manifests as repeated stanzas.

**Demo Output**

Using a seed text such as "Shall I compare thee to a summer's day", the model can generate text for up to 60–65 words before overfitting kicks in.

**Output:**

Shall I be gone go on me if i do i see thee slave to you as i have a palace with a year or bloody villain branch will be the instrument and she protector ho drop up her pains to be so far off in't and i will be gone and then i will not be so stern as i do not a gentleman in this care for my vow i have a gentleman to command it is not all uncleanliness uncleanliness uncleanliness uncleanliness uncleanliness uncleanliness

**Limitations**

Despite achieving good results initially, the model overfits and repeats itself after generating 60–65 words. This is addressed through techniques such as limiting vocabulary size, using batch processing, and exploring alternative architectures like GRU or Transformer models for future improvements.