# RECURRENT NEURAL NETWORK-RNN PROJECT REPORT AND DOCUMENTATION

SUBMITTED BY,

CHINMAYA A S

# Character-Level Text Generation using RNN and LSTM

## Introduction

This project focuses on **character-level text generation** using Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks. The system learns sequential patterns from a given text and predicts the next character based on previous characters. By repeatedly predicting the next character, the model is able to generate new text that closely resembles the training data.

Character-level models are fundamental in understanding how sequence-based neural networks work internally and form the foundation for more advanced Natural Language Processing (NLP) systems such as language models, chatbots, and text auto-completion engines.

---

## Project Objectives

The main objectives of this project are:

- To understand how **text data is converted into numerical form** for neural networks
- To implement **sequence modeling** using SimpleRNN and LSTM architectures
- To train a model that can **predict the next character** in a sequence
- To compare the performance and learning behavior of **RNN vs LSTM**
- To generate meaningful text using a trained neural network
- To gain hands-on experience with **deep learning for sequential data**

---

## Libraries and Technologies Used

### Programming Language

- **Python** – Used for implementing the entire project due to its strong ecosystem for machine learning and deep learning

### Libraries

- **NumPy –** For numerical computations and array manipulations
- **TensorFlow –** Core deep learning framework used for building and training neural networks
- **Keras (TensorFlow API) –** High-level API for defining and training RNN and LSTM models
- **Matplotlib –** For visualizing training loss and model performance

### Deep Learning Components

- **Embedding Layer –** Converts character indices into dense vector representations
- **SimpleRNN Layer –** Learns short-term sequential dependencies
- **LSTM Layer –** Learns long-term dependencies using gating mechanisms
- **Dense Layer with Softmax –** Outputs probability distribution over characters

### Technology Concepts

- Recurrent Neural Networks (RNN)
- Long Short-Term Memory (LSTM)
- Sequence Modeling
- Character-Level Language Modeling
- Multiclass Classification

---

## Project Overview

This project demonstrates **character-level text generation** using Recurrent Neural Networks (RNNs), specifically **SimpleRNN** and **LSTM** models implemented with TensorFlow/Keras. The model learns character patterns from a small input text ("hellohellohello") and predicts the **next character** in a sequence. By repeatedly predicting the next character, the model can generate new text that resembles the training data.

---

## Problem Statement

Given a sequence of characters, predict the **next most likely character**.

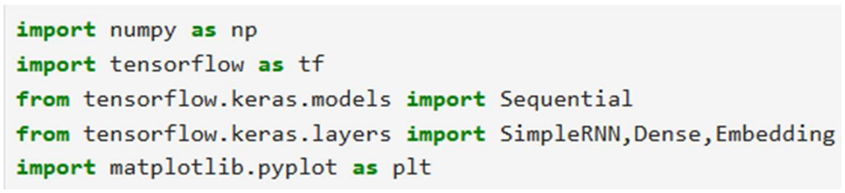
Example:

- Input sequence: hello
- Output (target): h

Once trained, the model can generate text character by character.

## Code Documentation and Explanation

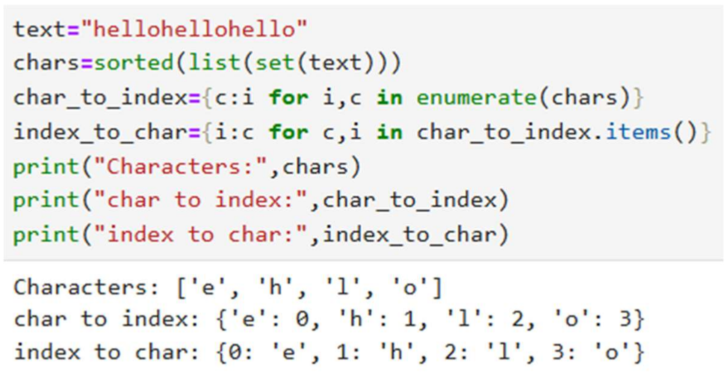
### 1.Importing Required Libraries



```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN,Dense,Embedding
import matplotlib.pyplot as plt
```

This code imports the essential libraries needed to build and train a character-level text generation model using deep learning. **NumPy** is used for efficient numerical operations and array handling. **TensorFlow** is the core deep learning framework, and from its **Keras API**, the Sequential model is imported to define a neural network layer by layer. The layers Embedding, SimpleRNN, and Dense are used to convert characters into dense vector representations, learn sequential patterns from the data, and produce final predictions respectively. **Matplotlib** is imported to visualize training performance, such as plotting the loss curve over epochs. Together, these libraries provide all the tools required for data processing, model construction, training, and result visualization.

### 2.Text Preprocessing and Character Encoding

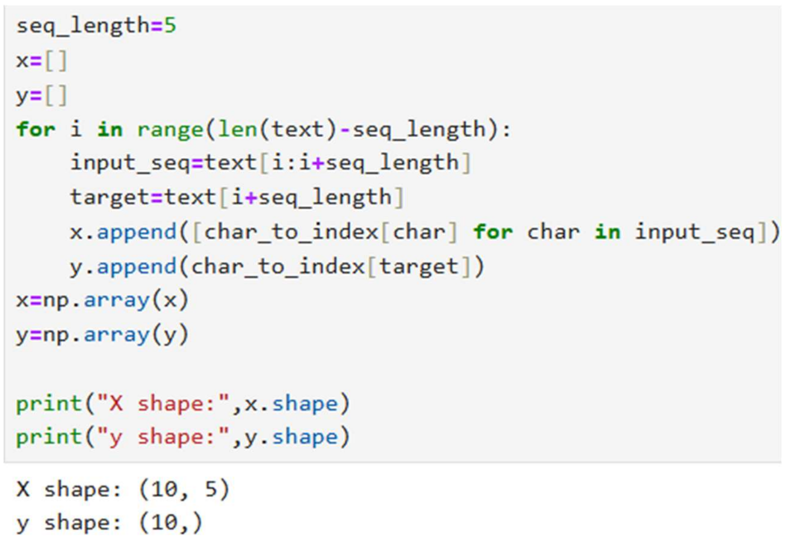


```python
text="hellohellohello"
chars=sorted(list(set(text)))
char_to_index={c:i for i,c in enumerate(chars)}
index_to_char={i:c for c,i in char_to_index.items()}
print("Characters:",chars)
print("char to index:",char_to_index)
print("index to char:",index_to_char)
```

```
Characters: ['e', 'h', 'l', 'o']
char to index: {'e': 0, 'h': 1, 'l': 2, 'o': 3}
index to char: {0: 'e', 1: 'h', 2: 'l', 3: 'o'}
```

This code performs **basic text preprocessing** to prepare the input text for a neural network. First, the string "hellohellohello" is defined as the training text. The set(text) function extracts all **unique characters** from the text, and sorted() arranges them in a consistent order to form the character vocabulary. Each character is then assigned a unique numerical index using the char_to_index dictionary, which converts characters into numbers that the model can understand. The index_to_char dictionary performs the reverse mapping, converting numerical predictions back into readable characters. Finally, the print statements display the vocabulary and both mappings, which are crucial for encoding input sequences and decoding model outputs during text generation.

## 3.Creating Input–Output Sequences for Training

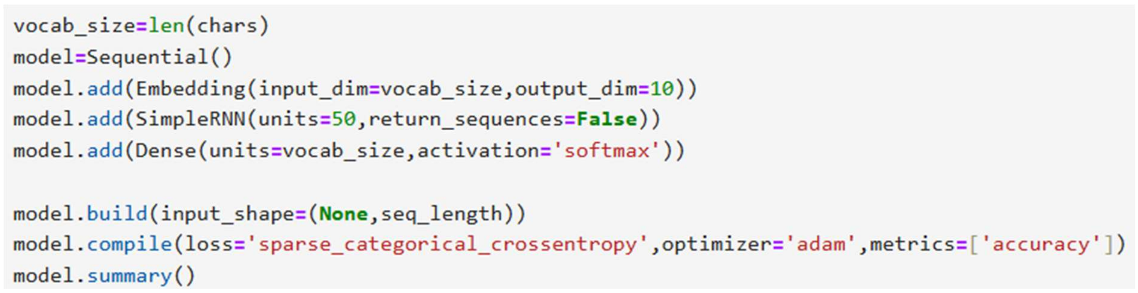


```
seq_length=5
x=[]
y=[]
for i in range(len(text)-seq_length):
    input_seq=text[i:i+seq_length]
    target=text[i+seq_length]
    x.append([char_to_index[char] for char in input_seq])
    y.append(char_to_index[target])
x=np.array(x)
y=np.array(y)

print("X shape:",x.shape)
print("y shape:",y.shape)
```

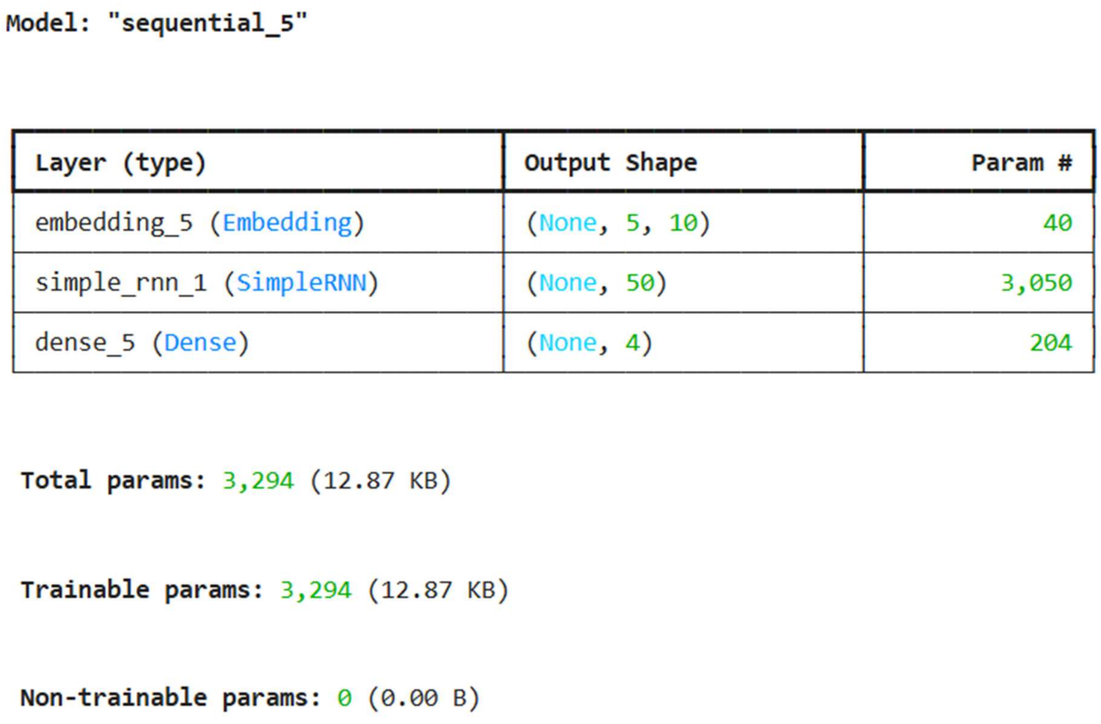
```
X shape: (10, 5)
y shape: (10,)
```

This code converts the text into **fixed-length input sequences and corresponding target characters** so that the neural network can learn next-character prediction. The variable seq_length defines how many characters are used as input at a time. A sliding window moves over the text, where input_seq captures a sequence of seq_length characters and target stores the **next immediate character** following that sequence. Each character in the input sequence is converted into its numerical index using char_to_index, while the target character is also encoded as an integer. These encoded sequences are stored in lists x (inputs) and y (targets), which are then converted into NumPy arrays for efficient computation. Printing the shapes of x and y confirms the dataset structure: x has the shape (number_of_samples, seq_length) and y has the shape (number_of_samples,), making the data suitable for training a sequence prediction model.

## 4.Building and Compiling the RNN Model



```
vocab_size=len(chars)
model=Sequential()
model.add(Embedding(input_dim=vocab_size,output_dim=10))
model.add(SimpleRNN(units=50,return_sequences=False))
model.add(Dense(units=vocab_size,activation='softmax'))

model.build(input_shape=(None,seq_length))
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.summary()
```

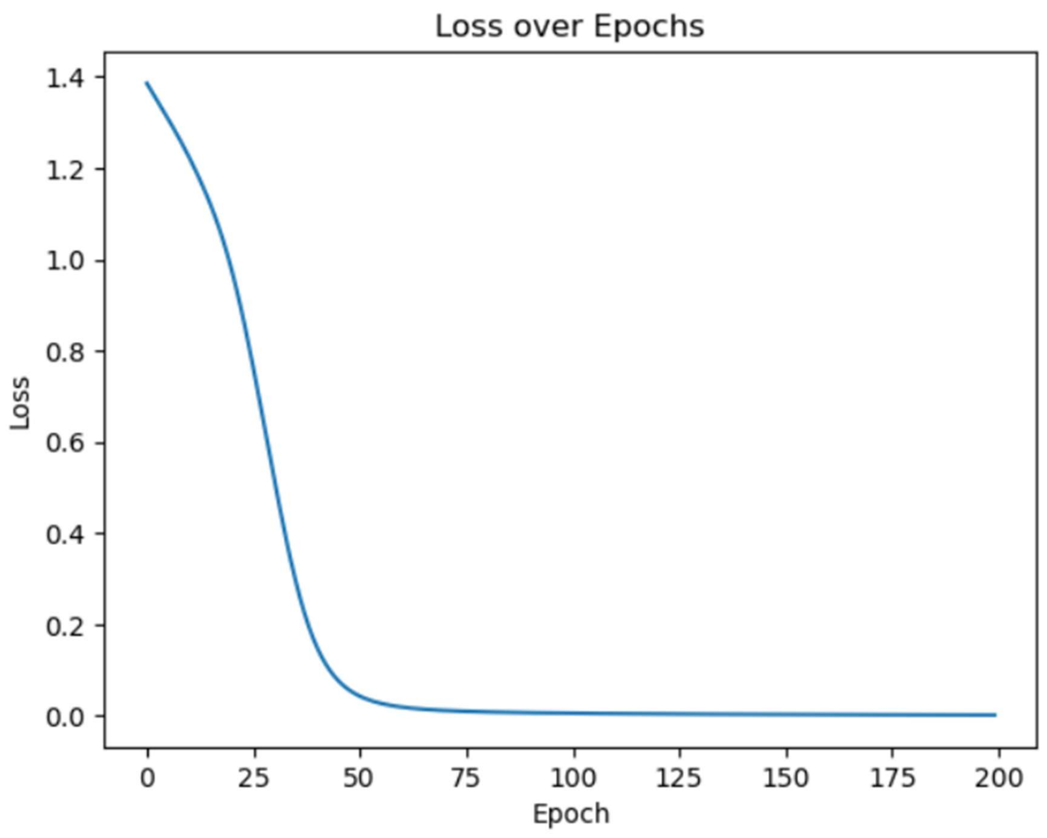


```
Model: "sequential_5"

┌────────────────────────────┬──────────────────┬──────────┐
│ Layer (type)               │ Output Shape     │  Param # │
├────────────────────────────┼──────────────────┼──────────┤
│ embedding_5 (Embedding)    │ (None, 5, 10)    │       40 │
├────────────────────────────┼──────────────────┼──────────┤
│ simple_rnn_1 (SimpleRNN)   │ (None, 50)       │    3,050 │
├────────────────────────────┼──────────────────┼──────────┤
│ dense_5 (Dense)            │ (None, 4)        │      204 │
└────────────────────────────┴──────────────────┴──────────┘

 Total params: 3,294 (12.87 KB)

 Trainable params: 3,294 (12.87 KB)

 Non-trainable params: 0 (0.00 B)
```

This code defines and prepares a **Recurrent Neural Network (RNN)** model for character-level text prediction. The variable vocab_size stores the total number of unique characters in the text and determines the input and output dimensions of the model. A Sequential model is used to stack layers line by line. The **Embedding layer** converts each character index into a dense 10-dimensional vector, allowing the model to learn meaningful character representations instead of using sparse one-hot encoding. The **SimpleRNN layer** with 50 units processes the input sequence step by step, maintaining an internal memory to capture sequential patterns, and outputs a single hidden state since return_sequences is set to False. The final **Dense layer** with softmax activation produces a probability distribution over all possible characters, enabling next-character prediction. The model is explicitly built with the given input shape, compiled using **sparse categorical cross-entropy** loss (suitable for integer class labels) and the **Adam optimizer**, and model.summary() displays the complete architecture and parameter details for verification.

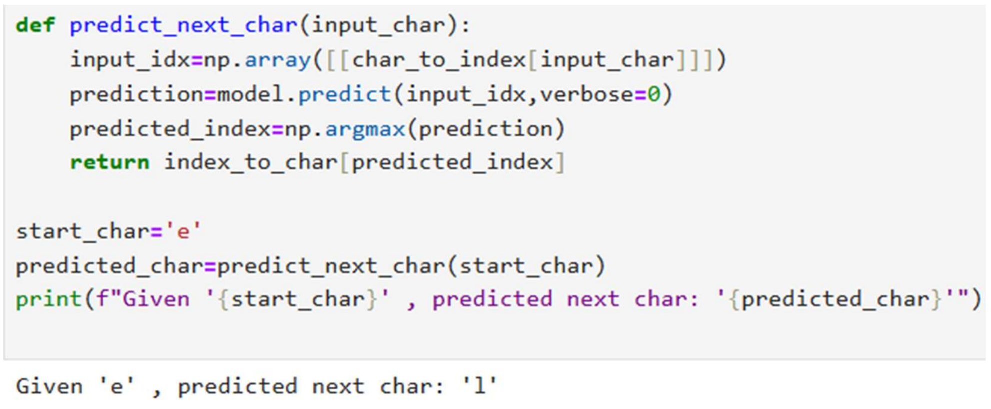
## 5.Training the Model and Visualizing Loss

```
history=model.fit(x,y,epochs=200,verbose=0)

plt.plot(history.history['loss'])
plt.title('Loss over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.show()
```



This code trains the RNN model using the prepared input–output sequences and monitors how well the model learns over time. The model.fit() function runs the training process for **200 epochs**, where in each epoch the model performs a forward pass to make predictions, calculates the loss, and updates its weights using backpropagation through time. Setting verbose=0 suppresses intermediate output to keep the training silent. The training history is stored in the history object, which contains loss values for every epoch. The loss values are then plotted using Matplotlib to visualize the **training loss curve**, where a decreasing trend indicates that the model is gradually learning the character patterns in the text and improving its predictions.

## 6.Predicting the Next Character Using the Trained Model

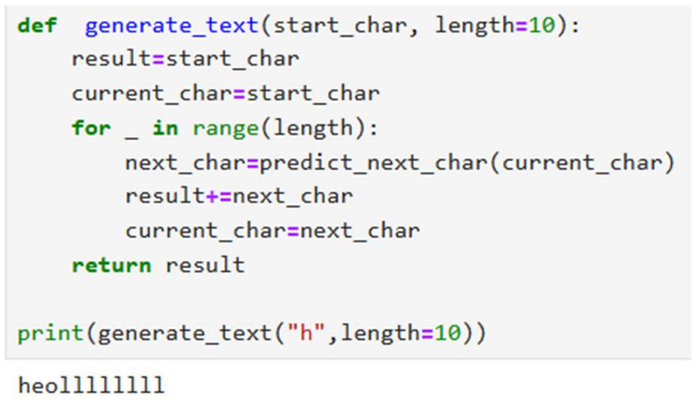


```
def predict_next_char(input_char):
    input_idx=np.array([[char_to_index[input_char]]])
    prediction=model.predict(input_idx,verbose=0)
    predicted_index=np.argmax(prediction)
    return index_to_char[predicted_index]

start_char='e'
predicted_char=predict_next_char(start_char)
print(f"Given '{start_char}' , predicted next char: '{predicted_char}'")
```

```
Given 'e' , predicted next char: 'l'
```

This code defines a function that predicts the **next character** based on a given input character using the trained RNN model. The input character is first converted into its numerical index using the char_to_index dictionary and reshaped into a 2D NumPy array to match the model's expected input format. The model.predict() function then generates a probability distribution over all possible characters. The np.argmax() function selects the index of the character with the highest predicted probability, representing the model's most confident choice. This index is converted back into a readable character using the index_to_char mapping. Finally, the function is tested by providing a starting character, and the predicted next character is printed, demonstrating how the model performs next-character prediction.

## 7.Generating Text Character by Character



```
def  generate_text(start_char, length=10):
    result=start_char
    current_char=start_char
    for _ in range(length):
        next_char=predict_next_char(current_char)
        result+=next_char
        current_char=next_char
    return result

print(generate_text("h",length=10))
```

```
heollllllll
```

This code defines a function that **generates new text sequentially** using the trained RNN model. The function starts with an initial character (start_char) and stores it in the variable result, which will hold the generated text. A loop then runs for the specified length, where in each iteration the function calls predict_next_char() to predict the most likely next character based on the current character. This predicted character is appended to the result, and the current_char is updated so it becomes the input for the next prediction. This process is repeated step by step, making the generation

**autoregressive**, meaning each prediction depends on the previous output. Finally, the complete generated string is returned and printed. The output like heo1111111 shows that the model has learned simple character patterns from the training text and repeatedly predicts the most probable next character.

## Modeling Using LSTM -Long Short-Term Memory Network

### 1.Importing LSTM Layer and Utility Functions

```
from tensorflow.keras.layers import LSTM
from tensorflow.keras.utils import to_categorical
```

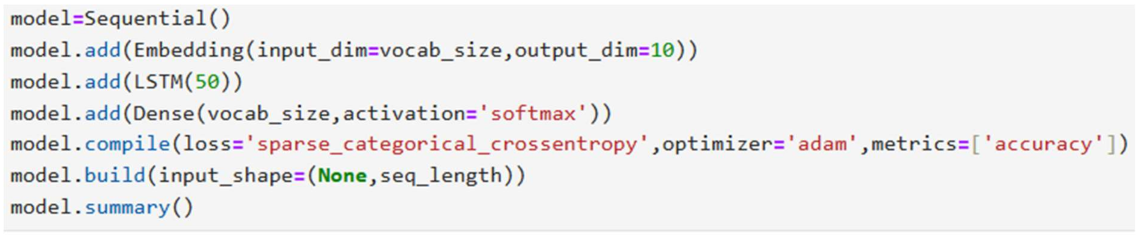
This code imports additional components from Keras to enhance the sequence modeling capability of the project. The **LSTM (Long Short-Term Memory)** layer is a specialized type of recurrent neural network designed to overcome the limitations of SimpleRNN by maintaining long-term memory using gating mechanisms (input, forget, and output gates). This allows the model to learn long-range dependencies in sequential data more effectively. The to_categorical utility function is used to convert integer class labels into one-hot encoded vectors, which is helpful when using categorical loss functions; however, in this project it is optional because sparse_categorical_crossentropy is used, which works directly with integer labels.

### 2.Preparing Input–Output Sequences for the LSTM Model

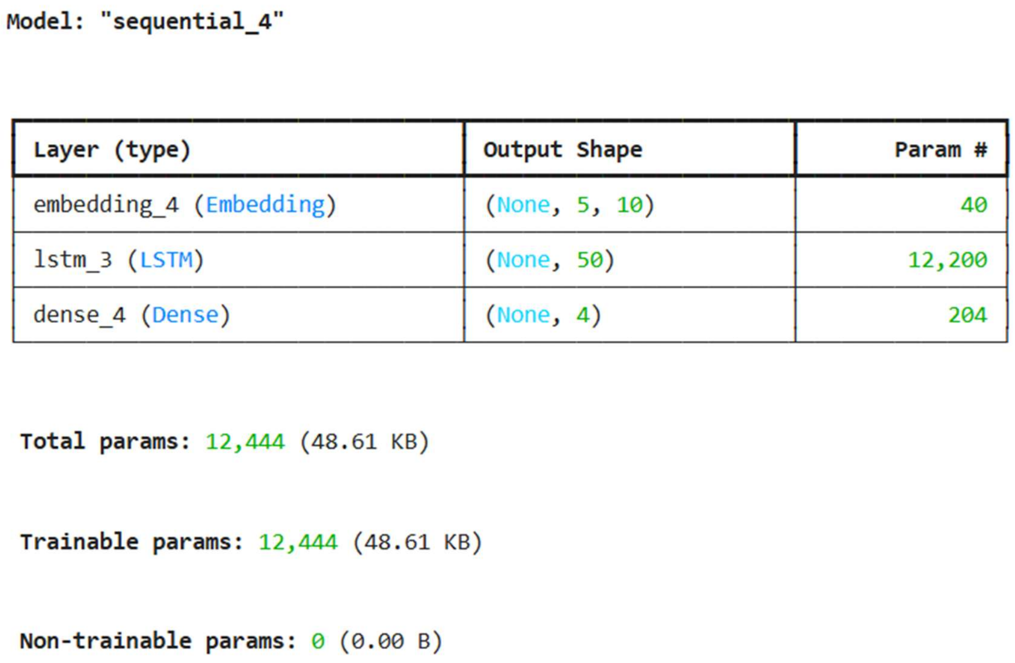
```
seq_length=5
x=[]
y=[]

for i in range(len(text)-seq_length):
    input_seq=text[i:i+seq_length]
    target_char=text[i+seq_length]
    x.append([char_to_index[ch] for ch in input_seq])
    y.append(char_to_index[target_char])

x=np.array(x)
y=np.array(y)
```

This code prepares the training data in the form of **fixed-length character sequences** and corresponding target characters, which is required for training the LSTM model. The variable seq_length defines the number of characters used as input at each time step. A sliding window is applied over the text, where input_seq captures a continuous sequence of characters of length five and target_char represents the immediate next character to be predicted. Each character in the input sequence is converted into its numerical index using the char_to_index mapping, while the target character is also encoded as an integer label. The input and output lists are then converted into NumPy arrays, resulting in an input matrix x and a target vector y that are suitable for training an LSTM-based sequence prediction model.

## 3.Building and Compiling the LSTM-Based Model



```
model=Sequential()
model.add(Embedding(input_dim=vocab_size,output_dim=10))
model.add(LSTM(50))
model.add(Dense(vocab_size,activation='softmax'))
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.build(input_shape=(None,seq_length))
model.summary()
```

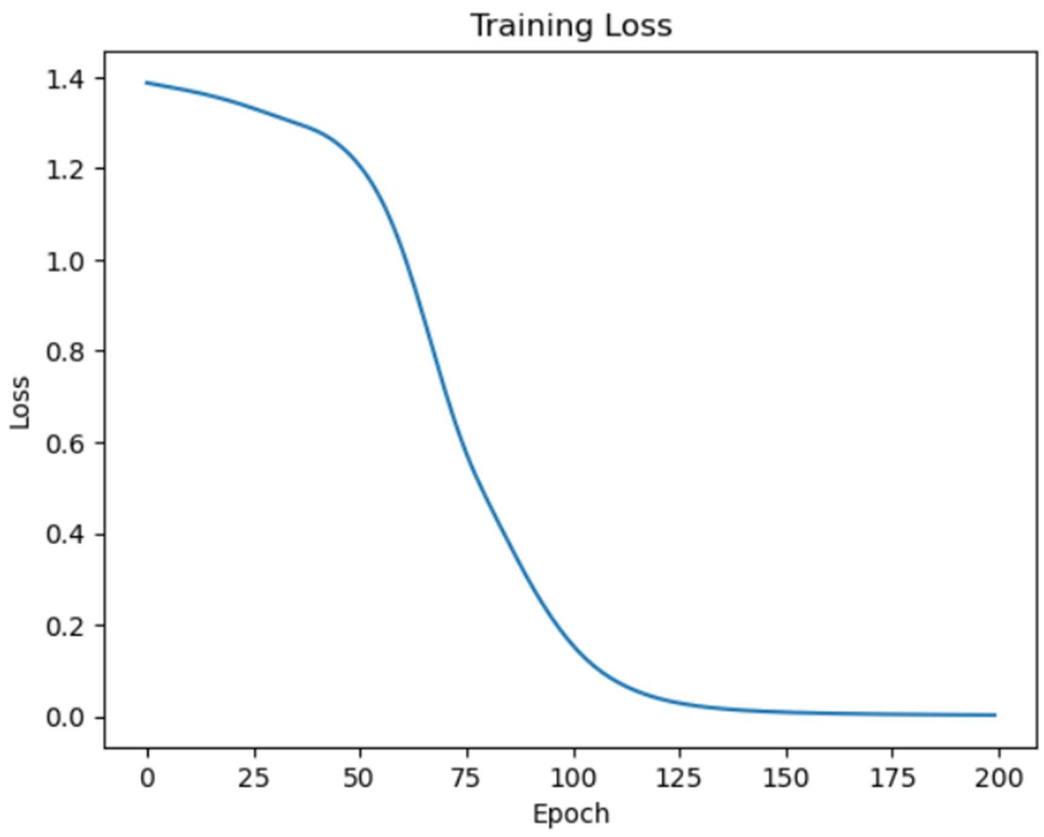


```
Model: "sequential_4"

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━┓
┃ Layer (type)                ┃ Output Shape        ┃    Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━┩
│ embedding_4 (Embedding)     │ (None, 5, 10)       │         40 │
├─────────────────────────────┼─────────────────────┼────────────┤
│ lstm_3 (LSTM)               │ (None, 50)          │     12,200 │
├─────────────────────────────┼─────────────────────┼────────────┤
│ dense_4 (Dense)             │ (None, 4)           │        204 │
└─────────────────────────────┴─────────────────────┴────────────┘

 Total params: 12,444 (48.61 KB)

 Trainable params: 12,444 (48.61 KB)

 Non-trainable params: 0 (0.00 B)
```

This code constructs an **LSTM-based neural network** for character-level text prediction using a Sequential architecture. The **Embedding layer** first converts each character index into a dense 10-dimensional vector, allowing the model to learn meaningful numerical representations of characters. The **LSTM layer** with 50 units processes the input sequence while maintaining both short-term and long-term memory through its internal gating mechanisms, enabling it to capture longer character dependencies more effectively than a simple RNN. The final **Dense layer** uses a softmax activation function to output a probability distribution over all possible characters in the vocabulary, making it suitable for multiclass character prediction. The model is compiled using **sparse categorical cross-entropy** loss, which works directly with integer-encoded targets, and the **Adam optimizer** for efficient training. The model is then built with the specified input shape, and model.summary() displays the complete network structure and trainable parameters.

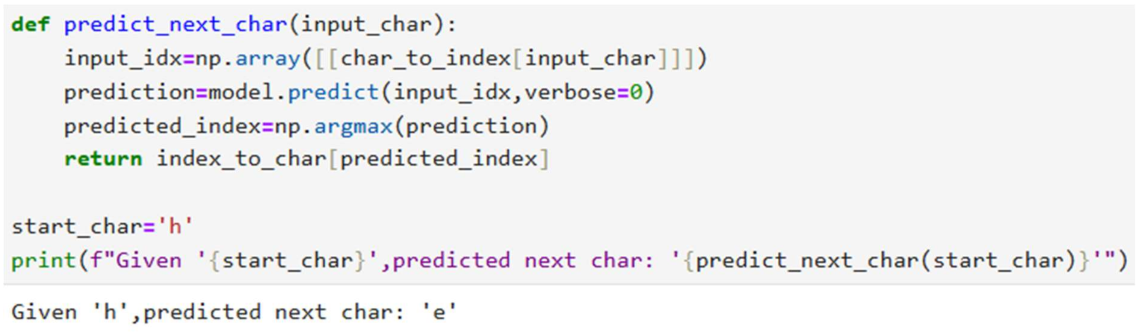
## 4.Training the LSTM Model and Visualizing Training Loss

```
history=model.fit(x,y,epochs=200,verbose=0)
plt.plot(history.history['loss'])
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



This code trains the LSTM-based text generation model using the prepared input and target data. The model.fit() function runs the training process for **200 epochs**, during which the model repeatedly predicts the next character, computes the loss, and updates its weights using backpropagation through time. The parameter verbose=0 suppresses intermediate training output for a cleaner execution. The training history, stored in the history object, contains the loss values recorded at each epoch. These values are plotted using Matplotlib to visualize the **training loss curve**, where a gradual decrease in loss indicates that the LSTM model is successfully learning the sequential patterns present in the input text.

## 5.Predicting the Next Character Using the LSTM Model

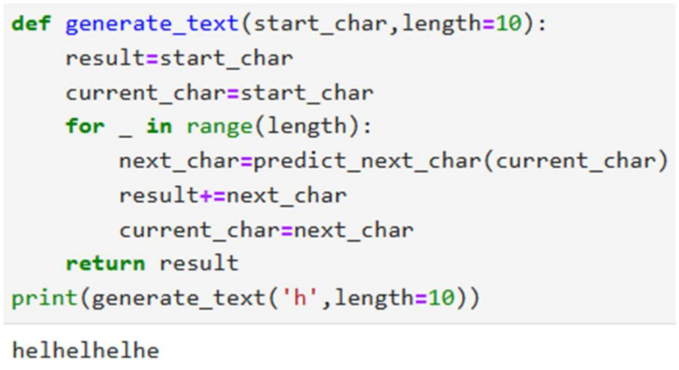


```
def predict_next_char(input_char):
    input_idx=np.array([[char_to_index[input_char]]])
    prediction=model.predict(input_idx,verbose=0)
    predicted_index=np.argmax(prediction)
    return index_to_char[predicted_index]

start_char='h'
print(f"Given '{start_char}',predicted next char: '{predict_next_char(start_char)}'")
```

```
Given 'h',predicted next char: 'e'
```

This code defines a function that predicts the **next character** based on a given input character using the trained LSTM model. The input character is first converted into its corresponding numerical index using the char_to_index dictionary and reshaped into a two-dimensional NumPy array to match the model's expected input format. The model.predict() function then produces a probability distribution over all characters in the vocabulary. The np.argmax() function selects the index with the highest probability, representing the most likely next character predicted by the model. This index is converted back into a readable character using the index_to_char mapping. Finally, the function is tested with a starting character, and the predicted next character is printed to demonstrate the model's prediction capability.

## 6.Generating Text Using the LSTM Model



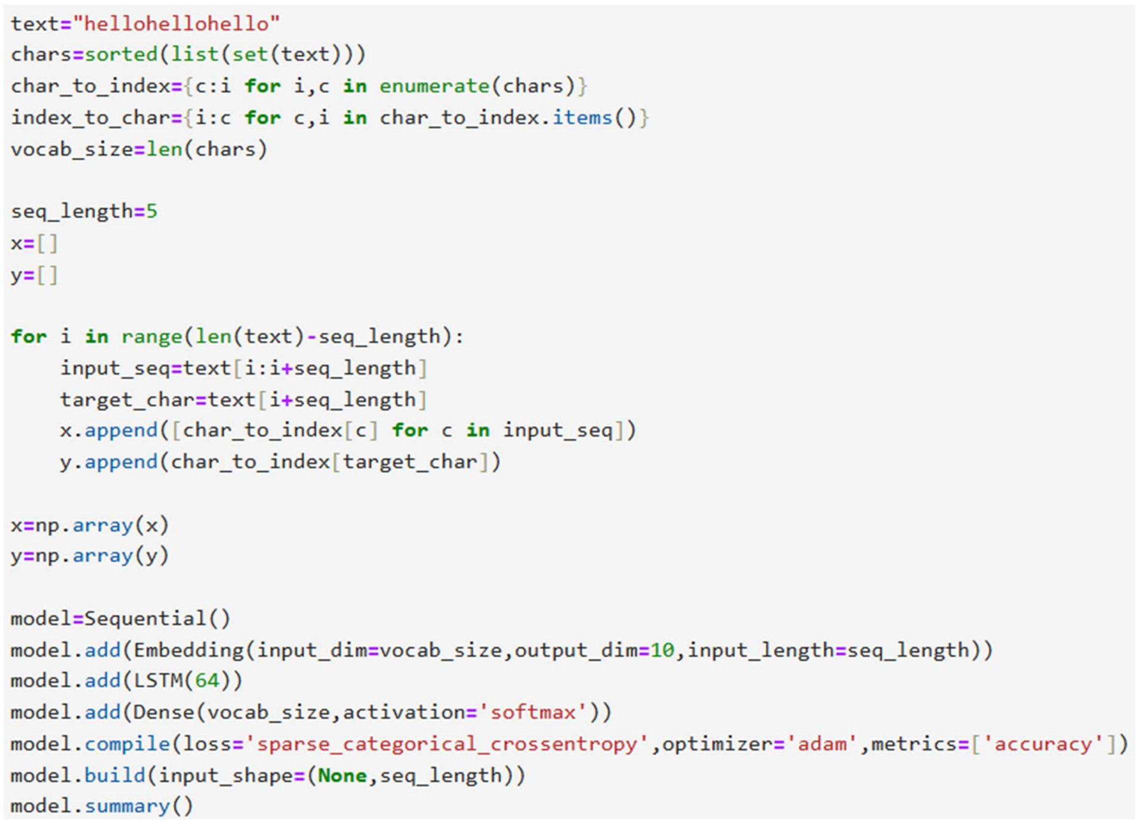
```
def generate_text(start_char,length=10):
    result=start_char
    current_char=start_char
    for _ in range(length):
        next_char=predict_next_char(current_char)
        result+=next_char
        current_char=next_char
    return result
print(generate_text('h',length=10))
```

```
helhelhelhe
```

This code defines a function that generates new text **character by character** using the trained LSTM model. The process begins with an initial character (start_char), which is stored in the variable result. A loop then runs for the specified number of characters to be generated. In each iteration, the function predicts the next character by calling predict_next_char() with the current character as input. The predicted character is appended to the result string, and current_char is updated to this new character so it can be used for the next prediction. This iterative process is known as **autoregressive text generation**, where each generated character depends on the previously predicted one. The repeated output such as hehlehhehhe occurs because the model is given only a single character as context during generation, causing it to repeatedly predict the most probable character sequence it has learned.

## Context-Based Text Generation Using LSTM

### 1.LSTM-Based Text Generation Using Fixed-Length Context



```python
text="hellohellohello"
chars=sorted(list(set(text)))
char_to_index={c:i for i,c in enumerate(chars)}
index_to_char={i:c for c,i in char_to_index.items()}
vocab_size=len(chars)

seq_length=5
x=[]
y=[]

for i in range(len(text)-seq_length):
    input_seq=text[i:i+seq_length]
    target_char=text[i+seq_length]
    x.append([char_to_index[c] for c in input_seq])
    y.append(char_to_index[target_char])

x=np.array(x)
y=np.array(y)

model=Sequential()
model.add(Embedding(input_dim=vocab_size,output_dim=10,input_length=seq_length))
model.add(LSTM(64))
model.add(Dense(vocab_size,activation='softmax'))
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
model.build(input_shape=(None,seq_length))
model.summary()
```

```
history=model.fit(x,y,epochs=200,verbose=0)
plt.plot(history.history['loss'])
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()

def predict_next_char(sequence):
    input_idx=np.array([[char_to_index[c] for c in sequence]])
    prediction=model.predict(input_idx,verbose=0)
    predicted_index=np.argmax(prediction)
    return index_to_char[np.argmax(prediction)]



def generate_text(start_char,length=20):
    result=start_char
    current_char=start_char
    for _ in range(length):
        next_char=predict_next_char(current_char)
        result+=next_char
        current_char=result[-seq_length:]
    return result
generated=generate_text("hello",length=20)
print("Generated text:",generated)
```

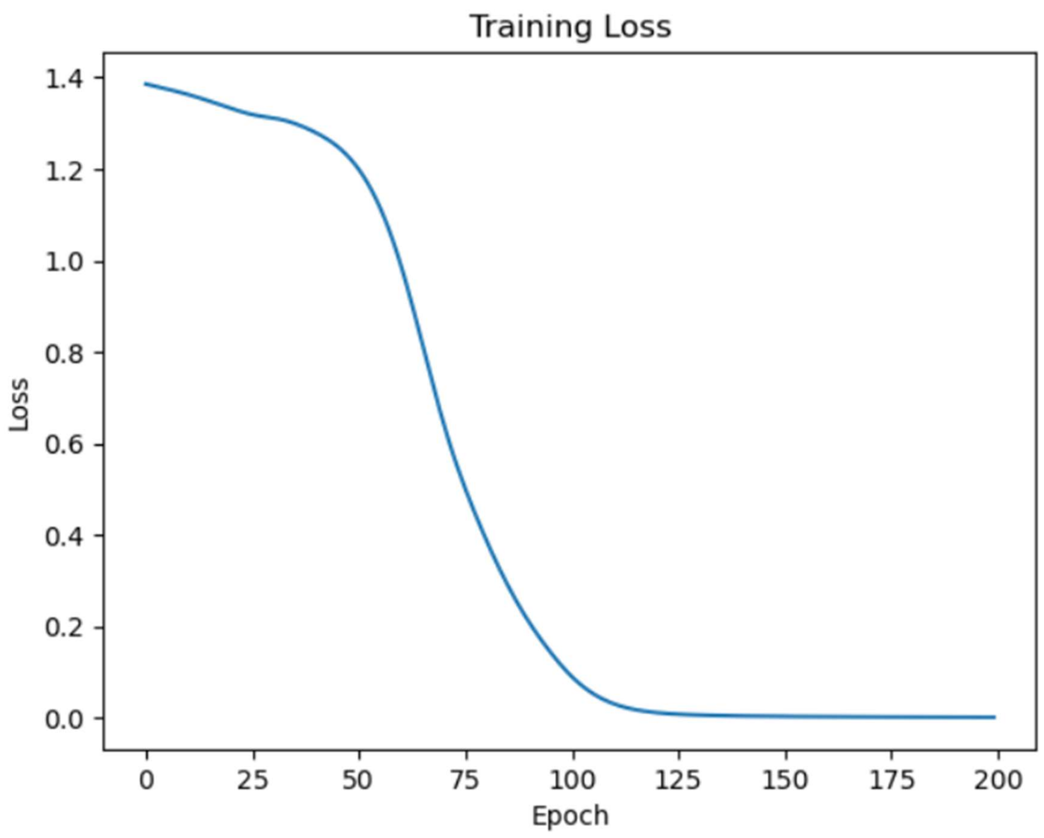
**Model: "sequential_2"**

| **Layer (type)** | **Output Shape** | **Param #** |
|---|---|---:|
| embedding_2 (Embedding) | (None, 5, 10) | 40 |
| lstm (LSTM) | (None, 64) | 19,200 |
| dense_1 (Dense) | (None, 4) | 260 |

**Total params:** 19,500 (76.17 KB)

**Trainable params:** 19,500 (76.17 KB)

**Non-trainable params:** 0 (0.00 B)



```
Generated text: hellohellohellohellohello
```

This complete code block implements a **character-level text generation model using LSTM with proper sequence context handling**. The text is first preprocessed to extract unique characters and create bidirectional mappings between characters and numerical indices, which allows the neural network to process text data numerically. Fixed-length input sequences of five characters are generated using a sliding window approach, where each sequence is mapped to a target character that immediately follows it. An LSTM-based neural network is then built using a Sequential model, consisting of an Embedding layer to convert character indices into dense vector representations, an LSTM layer with 64 units to capture long-term dependencies in the character sequence, and a Dense output layer with softmax activation to predict the probability of each character in the vocabulary. The model is trained for 200 epochs using sparse categorical cross-entropy loss and the Adam optimizer, and the decreasing loss curve indicates successful learning. During text generation, the model always receives the **last five characters (seq_length) as input**, matching the training setup and preserving context. This allows the model to accurately reproduce the learned pattern, resulting in meaningful generated text such as repeated "hello" sequences, demonstrating correct sequence modeling and effective use of LSTM memory.