Ex-10 Lab Manual: Implement a Recurrent Neural Network (RNN) for IMDB Movie Review Classification

---

1. Objective

To implement and train a **Recurrent Neural Network (RNN)** for classifying movie reviews as **positive or negative** using the **IMDB dataset**. The RNN model will process sequential text data and learn relationships between words over time.

2. Introduction to RNNs

A **Recurrent Neural Network (RNN)** is a type of neural network that is designed to handle **sequential data**, making it well-suited for tasks such as text classification, speech recognition, and time-series forecasting. Unlike traditional feedforward neural networks, RNNs **retain information from previous inputs**, allowing them to capture the context in text sequences.

Key Features of RNNs:

- Maintain memory of past inputs through **hidden states**.
- Process variable-length sequences efficiently.
- Suitable for NLP tasks like **text classification, machine translation, and sentiment analysis**.

In this lab, we will implement an **RNN using Long Short-Term Memory (LSTM) cells**, which help mitigate the vanishing gradient problem common in traditional RNNs.

---

3. System Requirements

Hardware Requirements:

- Computer with at least **4GB RAM** (8GB recommended)
- GPU support for faster training (optional but recommended)

Software Requirements:

- Python (>=3.6)
- TensorFlow/Keras
- NumPy, Matplotlib (for data processing and visualization)

Install the required libraries using:

```
pip install numpy tensorflow matplotlib
```

4. Step-by-Step Procedure

Step 1: Import Required Libraries

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import imdb

from tensorflow.keras.preprocessing.sequence import pad_sequences

from tensorflow.keras.models import Sequential

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

import matplotlib.pyplot as plt
```

Step 2: Load and Preprocess IMDB Dataset

The IMDB dataset consists of **50,000 movie reviews**, labeled as **positive (1)** or **negative (0)**.

```
# Define parameters

vocab_size = 10000  # Number of unique words to consider

max_length = 100  # Maximum words per review

embedding_dim = 32  # Size of word embeddings


# Load IMDB dataset (only the top 10,000 words)

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=vocab_size)


# Pad sequences to ensure uniform length

x_train = pad_sequences(x_train, maxlen=max_length, padding='post')

x_test = pad_sequences(x_test, maxlen=max_length, padding='post')
```

Step 3: Define the RNN Model with LSTM Layers

```
model = Sequential([

  Embedding(input_dim=vocab_size, output_dim=embedding_dim,
input_length=max_length),

  LSTM(64, return_sequences=False),  # LSTM layer with 64 units
```

```
    Dense(1, activation='sigmoid')  # Binary classification output
])


model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Step 4: Train the Model

```
history = model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_test, y_test))
```

Step 5: Evaluate the Model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")
```

Step 6: Plot Training History

```
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

---

5. Observations and Results

- The **RNN with LSTM** successfully classifies IMDB movie reviews as positive or negative.
- Training accuracy should improve over epochs, while validation accuracy indicates generalization.
- The **LSTM layer** captures long-range dependencies, improving classification performance compared to traditional models.

---

6. Troubleshooting & Optimization

- **Overfitting?** Apply dropout layers to reduce overfitting.

- **Low accuracy?** Increase the LSTM units or embedding dimensions.
- **Slow training?** Use a GPU for faster computation.

---

7. Additional Tasks

- Experiment with different **RNN architectures**, such as **GRUs**.
- Use **pre-trained word embeddings** like GloVe or Word2Vec.
- Extend the model to **multi-class classification** with more sentiment categories.

---

8. Conclusion

RNNs, especially **LSTMs**, are powerful for text classification tasks like sentiment analysis. They capture contextual meaning and dependencies between words, making them highly effective for NLP applications.

End of Lab Manual

what is Recurrent Neural Network

Recurrent Neral Network (RNN) - A Detailed Explanation

1. Introduction

A **Recurrent Neural Network (RNN)** is a type of artificial neural network designed to process **sequential data**, such as text, speech, and time series data. Unlike traditional **feedforward neural networks**, RNNs have a mechanism to retain memory of previous inputs, making them useful for tasks where **context and order** matter.

Example Use Cases:

✅ **Text classification** (e.g., sentiment analysis, spam detection)
✅ **Speech recognition** (e.g., Siri, Google Assistant)
✅ **Machine translation** (e.g., Google Translate)
✅ **Time series forecasting** (e.g., stock price prediction)

---

2. Why Use RNNs?

Traditional neural networks, like **fully connected (dense) networks**, treat each input independently, **ignoring sequence relationships**. RNNs solve this problem by introducing a **hidden state** that **remembers past inputs**, enabling the model to understand sequential dependencies.

Comparison of Neural Networks

Model Type	Processes Sequences?	Remembers Context?	Common Use Case
Feedforward NN	❌ No	❌ No	Image classification
Recurrent NN	✅ Yes	✅ Yes	Text & speech processing
LSTM / GRU	✅ Yes	✅ Better	Long-term dependencies

---

3. How RNNs Work

An RNN processes an **input sequence step-by-step**, maintaining a hidden state hth_tht that gets updated at each step based on the **current input** and the **previous hidden state**.

Mathematical Representation

At each time step t:

$$h_t = f(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

Where:

- h_t is the hidden state at time t
- W_h and W_x are weight matrices
- x_t is the current input
- b is a bias term
- f is an activation function (usually **tanh** or **ReLU**)

---

4. Types of RNN Architectures

4.1 Basic RNN

- The simplest form of RNN, but suffers from the **vanishing gradient problem**, making it hard to capture long-term dependencies.

4.2 Long Short-Term Memory (LSTM)

- A special type of RNN that solves the vanishing gradient problem using **gates** (forget, input, and output gates).
- Ideal for **long-range dependencies** in text and speech data.

4.3 Gated Recurrent Unit (GRU)

- A simplified version of LSTM with fewer parameters, making it faster.
- Works well for short and medium-length sequences.

---

5. Advantages & Limitations of RNNs

Feature	RNNs	LSTMs/GRUs
Handles sequences?	✅ Yes	✅ Yes
Short-term memory?	✅ Good	✅ Excellent
Long-term memory?	❌ Weak	✅ Strong
Training efficiency	✅ Fast	❌ Slower (more parameters)

🔴 **Limitations of Basic RNNs:**

- **Vanishing Gradient Problem**: Difficulty in learning long-term dependencies.
- **Exploding Gradients**: Large gradients can make the model unstable.
- **Slow Training**: Sequential processing limits parallel computation.

---

6. Implementing an RNN in Python (Using TensorFlow/Keras)

Step 1: Import Dependencies

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

Step 2: Define the RNN Model

```
model = Sequential([
    Embedding(input_dim=5000, output_dim=32, input_length=100),  # Word embeddings
    SimpleRNN(64, return_sequences=False),  # RNN Layer
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```