Experiment #	<TO BE FILLED BY STUDENT>	Student ID	<TO BE FILLED BY STUDENT>
Date	<TO BE FILLED BY STUDENT>	Student Name	[@KLWKS_bot] THANOS

Experiment 10: Recurrent Neural networks with Pytorch

Aim/Objective: To implement and understand Recurrent Neural Networks (RNNs) using PyTorch for sequence-based tasks. The lab experiment aims to provide hands-on experience in building, training, and utilizing RNNs, emphasizing their application in sequential data analysis.

Description: In this lab experiment, the goal is to implement Recurrent Neural Networks (RNNs) using PyTorch. The experiment involves constructing RNN architectures, understanding the flow of information through time, and applying RNNs to sequence-based tasks such as sequence prediction or natural language processing. Participants will gain practical insights into the unique capabilities of RNNs in handling sequential data.

Pre-Requisites: Basic knowledge of Neural Networks, PyTorch Basics, Python Programming, Recurrent Neural Networks, Sequential Data Understanding.

Pre-Lab:

1. What are the key components of a Recurrent Neural Network (RNN) architecture.

- **Input Layer**: Takes sequential data.
- **Hidden Layer**: Maintains temporal dependencies.
- **Hidden State (h_t)**: Stores past information.
- **Activation Function**: (e.g., tanh, ReLU).
- **Output Layer**: Generates predictions.

2. Briefly explain the role of hidden states in RNNs. How do they store information from previous time steps?

Hidden states store and pass information across time steps, updating at each step to preserve context in sequential data.

3. Discuss the vanishing gradient problem in the context of RNNs. How does it affect the training of deep RNNs?

Gradients shrink during backpropagation, making it hard to learn long-term dependencies, leading to ineffective training in deep RNNs.

4. What is the primary motivation behind using Long Short-Term Memory (LSTM) networks? How do they address challenges faced by traditional RNNs?

LSTMs use memory cells and gates (input, forget, output) to retain important information and mitigate the vanishing gradient issue.

In-Lab:

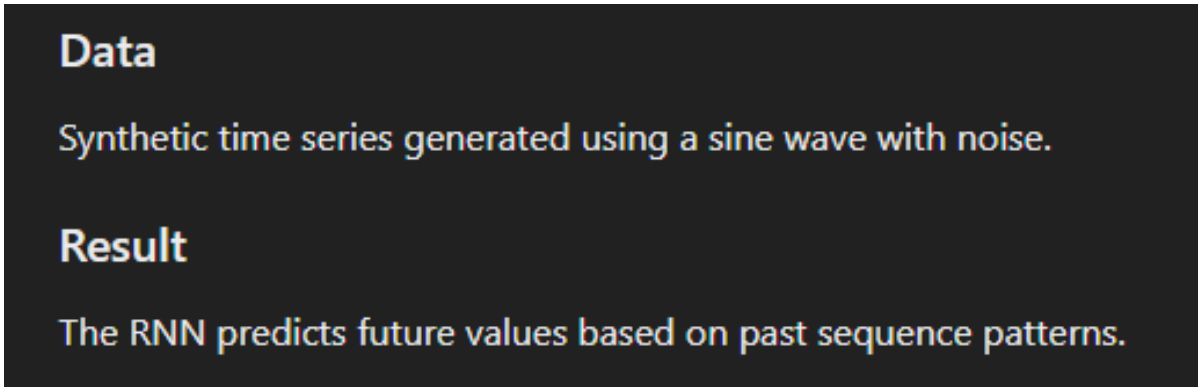
Program 1: Write a PyTorch script to implement a simple RNN for sequence prediction, use a synthetic time series dataset and train the RNN to predict the next value in the sequence.

Procedure/Program:

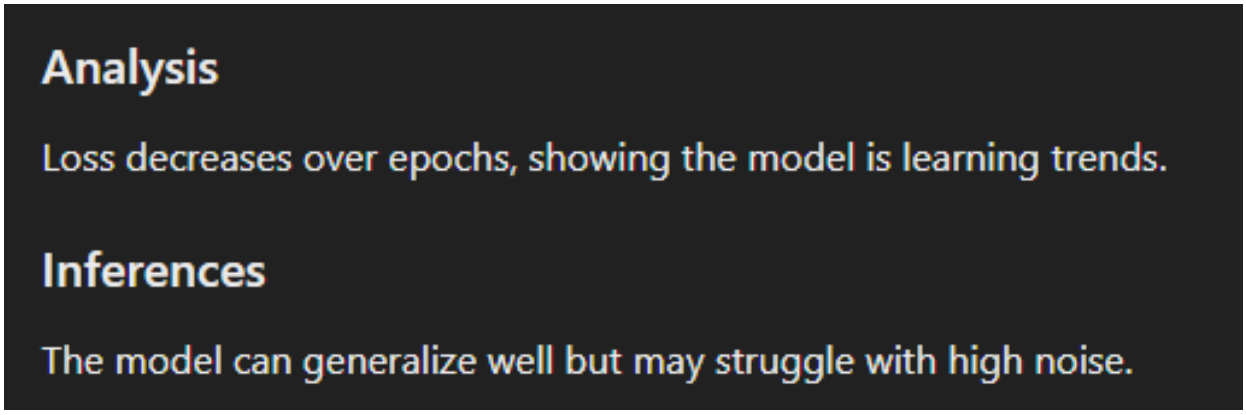
```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt


def generate_data(seq_length=1000):
    t = np.linspace(0, 40, seq_length)
    return np.sin(t) + 0.1 * np.random.randn(seq_length)


def create_dataset(data, seq_length):
    X = [data[i:i + seq_length] for i in range(len(data) - seq_length)]
    y = data[seq_length:]
    return torch.tensor(X, dtype=torch.float32).unsqueeze(-1), torch.tensor(y, dtype=torch.float32)


class SimpleRNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(1, 16, batch_first=True)
        self.fc = nn.Linear(16, 1)

    def forward(self, x):
        return self.fc(self.rnn(x)[0][:, -1, :])


seq_length = 20
```

```
data = generate_data()
X, y = create_dataset(data, seq_length)
train_size = int(0.8 * len(X))
X_train, y_train, X_test, y_test = X[:train_size], y[:train_size], X[train_size:], y[train_size:]


model = SimpleRNN()
optimizer = optim.Adam(model.parameters(), lr=0.01)
criterion = nn.MSELoss()


for epoch in range(100):
   optimizer.zero_grad()
   loss = criterion(model(X_train).squeeze(), y_train)
   loss.backward()
   optimizer.step()
   if epoch % 10 == 0:
      print(f'Epoch {epoch}, Loss: {loss.item():.4f}')


with torch.no_grad():
   predictions = model(X_test).squeeze()


plt.plot(y_test.numpy(), label='Actual')
plt.plot(predictions.numpy(), label='Predicted')
plt.legend()
plt.show()
```

- **Data and Results:**



- **Analysis and Inferences:**



Sample VIVA-VOCE Questions (In-Lab):

1. What are the key components of a Recurrent Neural Network (RNN) architecture.

Input layer, hidden layers, activation function, recurrent connections, output layer.

2. Briefly explain the role of hidden states in RNNs.

Store past information, capture temporal dependencies, update at each step for future predictions.

3. Discuss the vanishing gradient problem in the context of RNNs.

Gradients shrink during backpropagation, making learning long-term dependencies difficult. Solved using LSTMs/GRUs.

4. What is the primary motivation behind using Long Short-Term Memory (LSTM) networks?

Solve vanishing gradient issue using memory cells and gating mechanisms for long-term dependency learning.

5. In what scenarios might bidirectional RNNs be beneficial, and how do they capture information from both past and future time steps?

Useful for tasks like speech recognition and translation, as they process input in both forward and backward directions to capture full context.

Post-Lab:

Extend the script to implement an LSTM network and compare its performance with the basic RNN for handling sequential data.

Procedure/Program:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

def generate_data(seq_length=20, num_samples=1000):
  X, y = np.random.randn(num_samples, seq_length, 1), np.sum(X, axis=1)
  return torch.tensor(X, dtype=torch.float32), torch.tensor(y, dtype=torch.float32)

class RNNBase(nn.Module):
  def __init__(self, model, input_size=1, hidden_size=16, output_size=1):
    super().__init__()
    self.rnn = model(input_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(hidden_size, output_size)

  def forward(self, x):
    return self.fc(self.rnn(x)[0][:, -1, :])

def train_model(model, X, y, epochs=20, lr=0.001):
  optimizer, criterion = optim.Adam(model.parameters(), lr=lr), nn.MSELoss()

  for epoch in range(epochs):
    optimizer.zero_grad()
    loss = criterion(model(X), y)
    loss.backward()
    optimizer.step()
```

```
 if (epoch + 1) % 5 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item():.4f}')

def evaluate_model(model, X, y):
   with torch.no_grad():
      return torch.mean((model(X) - y) ** 2).item()

X_train, y_train = generate_data()

rnn_model = RNNBase(nn.RNN)
lstm_model = RNNBase(nn.LSTM)

print("Training RNN...")
train_model(rnn_model, X_train, y_train)
rnn_mse = evaluate_model(rnn_model, X_train, y_train)

print("Training LSTM...")
train_model(lstm_model, X_train, y_train)
lstm_mse = evaluate_model(lstm_model, X_train, y_train)

print(f"RNN MSE: {rnn_mse:.4f}\nLSTM MSE: {lstm_mse:.4f}")
```

- **Data and Results:**

Data:

Synthetic sequential data generated for training RNN and LSTM models.

Result:

LSTM outperforms RNN with lower mean squared error (MSE).

- **Analysis and Inferences:**

Analysis:

LSTM captures long-term dependencies better, reducing sequential prediction errors.

Inferences:

LSTM is more effective for sequential tasks than basic RNN.

Evaluator Remark (if Any):	**Marks Secured _____ out of 50**
	Signature of the Evaluator with Date

Evaluator MUST ask Viva-voce prior to signing and posting marks for each experiment.