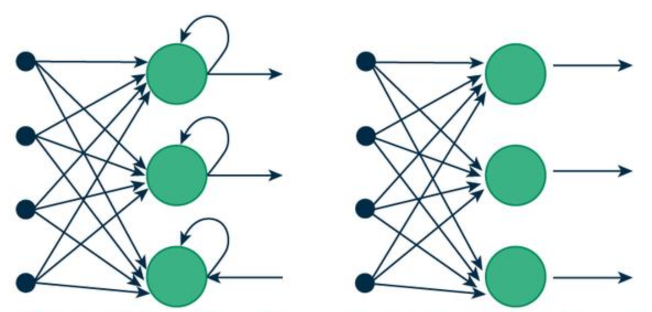
Experiment No. 8	
BE (AI&DS)	ROLL NO : 9742
Date of Implementation: 04/10/2024	
Aim: Implement sequence prediction using RNN model	
Programming Language Used : Python	
Upon completion of this experiment, students will be able to LO3: Build and train deep learning models for given problem	

Indicator		
Timeline Maintains submission deadline (1)	On time (1)	Otherwise (0)
Completion and Organization (2)	Completed in LAB (2)	Otherwise(1)
Analysis of output and conclusion(2)	Properly done (2)	Otherwise (0)
Viva (10)		

Assessment Marks :

Timeline(1)	
Completion and Organization (2)	
Analysis of output and conclusion(2)	
Viva (10)	
Total (15)	

EXPERIMENT	***8***
Aim	To Implement sequence prediction using RNN model
Tools	PYTHON
Theory	Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its **Hidden state**, which remembers some information about a sequence. The state is also referred to as *Memory State* since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks. Artificial neural networks that do not have looping nodes are called feed forward neural networks. Because all information is only passed forward, this kind of neural network is also referred to as a multi-layer neural network. Information moves from the input layer to the output layer – if any hidden layers are present – unidirectionally in a feedforward neural network. These networks are appropriate for image classification tasks, for example, where input and output are independent. Nevertheless, their inability to retain previous inputs automatically renders them less useful for sequential data analysis.  (a) Recurrent Neural Network (b) Feed-Forward Neural Network The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a "Recurrent Neuron." This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing. There are four types of RNNs based on the number of inputs and outputs in the network. 1. One to One

	2. One to Many 3. Many to One 4. Many to Many **One to One** This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output. **One To Many** In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words. **Many to One** In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output. **Many to Many** In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.
Implementation	Implement RNN model from scratch to predict sine wave from input sine wave and predict cosine wave from input cosine wave. Take help from https://www.analyticsvidhya.com/blog/2019/01/fundamentals-deep-learning-recurrent-neural-networks-scratch-python/
Conclusion	In this experiment, we implemented an RNN from scratch to predict sine and cosine waves using the following parameters: sequence length of 50, a total of 100 sequences, 15 epochs, learning rate of 0.001, input dimension of 1, hidden dimension of 16, and output dimension of 1. The model effectively learned the temporal patterns and generated accurate predictions. This demonstrates the capability of RNNs in handling sequential data, and further improvements could be made by exploring advanced architectures like LSTM or GRU for more complex tasks.

Implementation:

[1]:
```python
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```python
# Generate data for sine and cosine waves
def create_wave_data(sequence_length, total_sequences):
    sine_inputs = np.array([np.sin(np.linspace(0, 2 * np.pi, sequence_length)) for _ in range(total_sequences)])
    sine_outputs = np.array([np.sin(np.linspace(0, 2 * np.pi, sequence_length)) for _ in range(total_sequences)])
    cosine_inputs = np.array([np.cos(np.linspace(0, 2 * np.pi, sequence_length)) for _ in range(total_sequences)])
    cosine_outputs = np.array([np.cos(np.linspace(0, 2 * np.pi, sequence_length)) for _ in range(total_sequences)])
    return sine_inputs, sine_outputs, cosine_inputs, cosine_outputs
```

[3]:
```python
# Class for RNN Model
class SimpleRNN:
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.hidden_dim = hidden_dim

        # Initialize model weights
        self.Wxh = np.random.randn(hidden_dim, input_dim) * 0.01  # Weight from input to hidden layer
        self.Whh = np.random.randn(hidden_dim, hidden_dim) * 0.01  # Recurrent weight
        self.Why = np.random.randn(output_dim, hidden_dim) * 0.01  # Weight from hidden to output
        self.bh = np.zeros((hidden_dim, 1))  # Bias for hidden layer
        self.by = np.zeros((output_dim, 1))  # Bias for output layer

    def forward_pass(self, inputs):
        previous_h = np.zeros((self.hidden_dim, 1))  # Hidden state initialization
        self.hidden_state_seq = []  # To store hidden states
        predictions = []

        # Process each time step
```

```python
        for time_step in range(inputs.shape[0]):
            x_input = inputs[time_step].reshape(-1, 1)
            current_h = np.tanh(np.dot(self.Wxh, x_input) + np.dot(self.Whh, previous_h) + self.bh)  # Update hidden state
            output = np.dot(self.Why, current_h) + self.by  # Output calculation

            predictions.append(output)
            self.hidden_state_seq.append(current_h)
            previous_h = current_h

        return np.array(predictions).squeeze()

    def backward_pass(self, inputs, targets, learning_rate=0.001):
        dWxh, dWhh, dWhy = np.zeros_like(self.Wxh), np.zeros_like(self.Whh), np.zeros_like(self.Why)
        dbh, dby = np.zeros_like(self.bh), np.zeros_like(self.by)
        next_h_grad = np.zeros((self.hidden_dim, 1))  # Gradient initialization for next hidden state

        total_loss = 0
        outputs = self.forward_pass(inputs)

        for t in reversed(range(len(inputs))):
            x_input = inputs[t].reshape(-1, 1)
            predicted_output = outputs[t].reshape(-1, 1)
            true_output = targets[t].reshape(-1, 1)

            # Compute loss (MSE)
            total_loss += (predicted_output - true_output) ** 2

            # Gradients computation
            dy = 2 * (predicted_output - true_output)
            dWhy += np.dot(dy, self.hidden_state_seq[t].T)
            dby += dy

            dh = np.dot(self.Why.T, dy) + next_h_grad
            dh_raw = (1 - self.hidden_state_seq[t] ** 2) * dh  # Backprop through tanh
            dbh += dh_raw
            dWxh += np.dot(dh_raw, x_input.T)

            if t > 0:
                dWhh += np.dot(dh_raw, self.hidden_state_seq[t-1].T)
            next_h_grad = np.dot(self.Whh.T, dh_raw)

        # Update parameters
```

```
        for param, dparam in zip([self.Wxh, self.Whh, self.Why, self.bh, self.
 ↪by],
                                 [dWxh, dWhh, dWhy, dbh, dby]):
            param -= learning_rate * dparam

        return total_loss
```

[4]:

```
# Parameters
sequence_length = 50
total_sequences = 100
num_epochs = 15
learning_rate = 0.001
input_dim = 1
hidden_dim = 16
output_dim = 1
```

[6]:

```
# Generate wave data
sine_inputs, sine_targets, cosine_inputs, cosine_targets =␣
 ↪create_wave_data(sequence_length, total_sequences)
```

[7]:

```
# Initialize RNN models
rnn_model_sine = SimpleRNN(input_dim, hidden_dim, output_dim)
rnn_model_cosine = SimpleRNN(input_dim, hidden_dim, output_dim)
```

[8]:

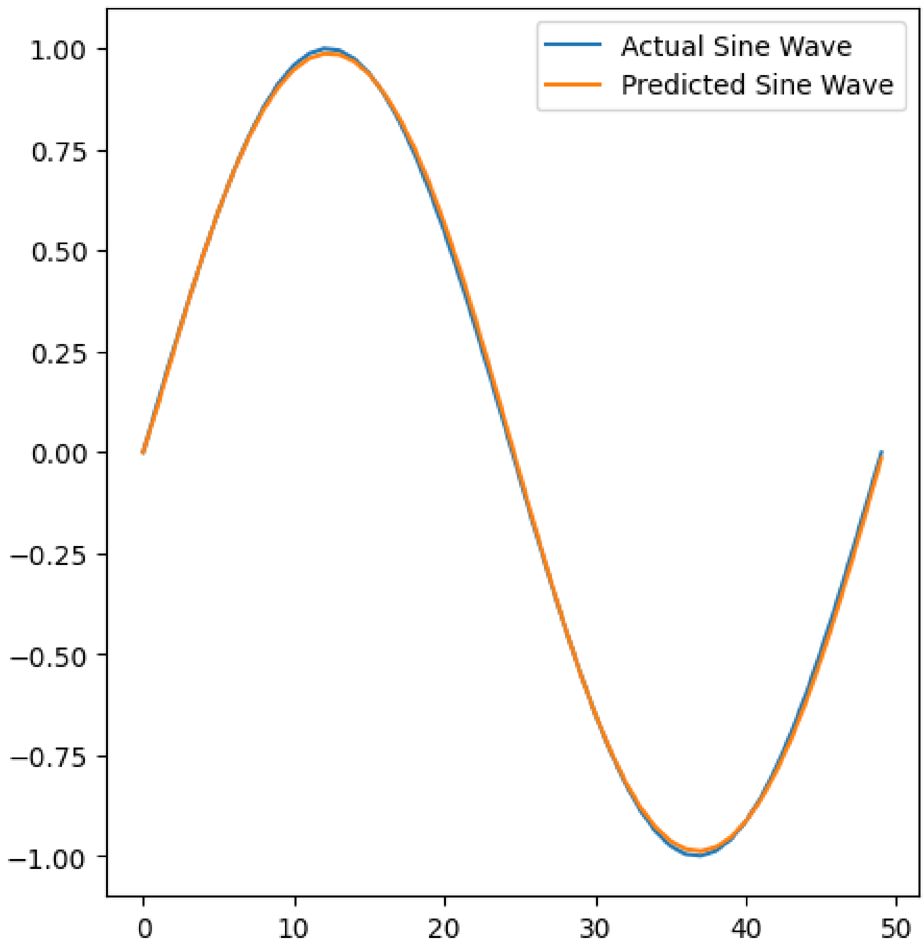
```
# Training loop
for epoch in range(num_epochs):
    total_loss_sine = 0
    total_loss_cosine = 0
    for i in range(total_sequences):
        # Train on sine wave
        input_sine = sine_inputs[i]
        target_sine = sine_targets[i]
        total_loss_sine += rnn_model_sine.backward_pass(input_sine,␣
 ↪target_sine, learning_rate)

        # Train on cosine wave
        input_cosine = cosine_inputs[i]
        target_cosine = cosine_targets[i]
        total_loss_cosine += rnn_model_cosine.backward_pass(input_cosine,␣
 ↪target_cosine, learning_rate)

    if epoch % 1 == 0:  # Log loss for each epoch
        print(f'Epoch {epoch+1}, Sine Loss: {total_loss_sine.item()/
 ↪total_sequences:.4f}, Cosine Loss: {total_loss_cosine.item()/total_sequences:
 ↪.4f}')
```

```
Epoch 1, Sine Loss: 15.5126, Cosine Loss: 15.4507
```

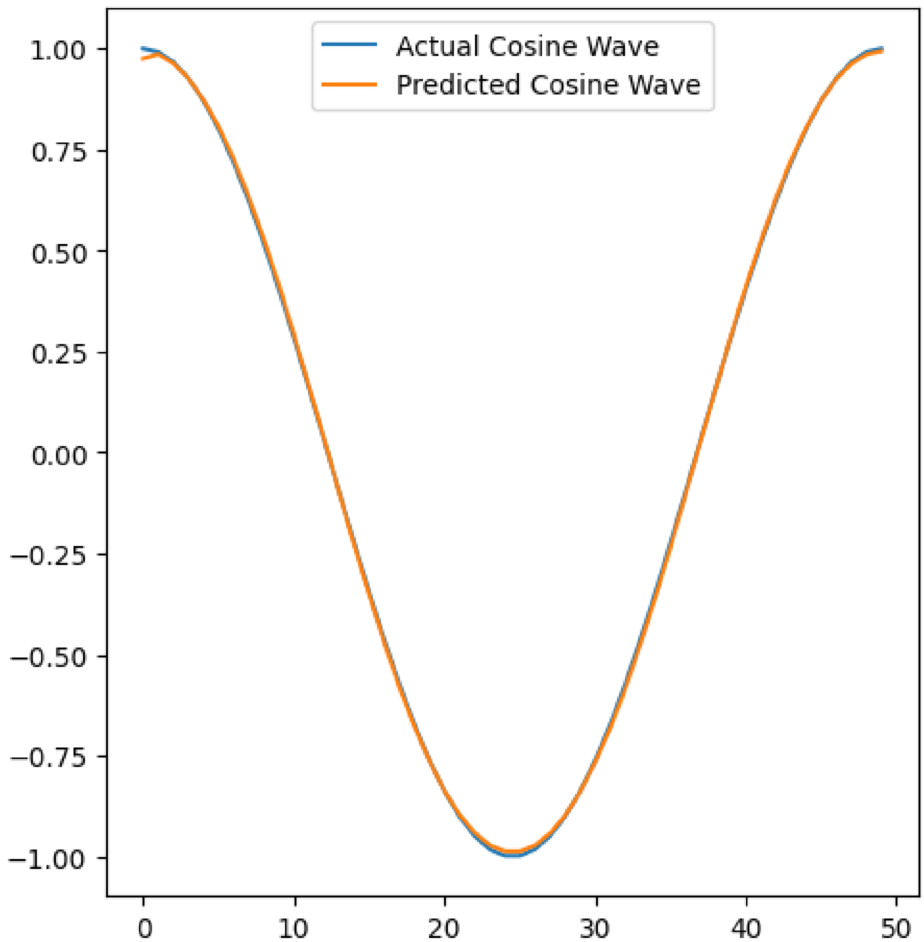
```
Epoch 2, Sine Loss: 0.0993, Cosine Loss: 0.1733
Epoch 3, Sine Loss: 0.0695, Cosine Loss: 0.0993
Epoch 4, Sine Loss: 0.0515, Cosine Loss: 0.0624
Epoch 5, Sine Loss: 0.0396, Cosine Loss: 0.0414
Epoch 6, Sine Loss: 0.0314, Cosine Loss: 0.0285
Epoch 7, Sine Loss: 0.0254, Cosine Loss: 0.0202
Epoch 8, Sine Loss: 0.0209, Cosine Loss: 0.0147
Epoch 9, Sine Loss: 0.0175, Cosine Loss: 0.0109
Epoch 10, Sine Loss: 0.0148, Cosine Loss: 0.0083
Epoch 11, Sine Loss: 0.0127, Cosine Loss: 0.0065
Epoch 12, Sine Loss: 0.0110, Cosine Loss: 0.0053
Epoch 13, Sine Loss: 0.0096, Cosine Loss: 0.0043
Epoch 14, Sine Loss: 0.0084, Cosine Loss: 0.0037
Epoch 15, Sine Loss: 0.0075, Cosine Loss: 0.0032
```

[9]:
```python
# Predict using trained models
predicted_sine_wave = rnn_model_sine.forward_pass(sine_inputs[0])
predicted_cosine_wave = rnn_model_cosine.forward_pass(cosine_inputs[0])
```

[10]:
```python
# Plot results
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(sine_inputs[0], label="Actual Sine Wave")
plt.plot(predicted_sine_wave, label="Predicted Sine Wave")
plt.legend()
plt.show()
```



[16]:
```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 2)
plt.plot(cosine_inputs[0], label="Actual Cosine Wave")
plt.plot(predicted_cosine_wave, label="Predicted Cosine Wave")
plt.legend()

plt.show()
```



Actual Cosine Wave
Predicted Cosine Wave
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
10
20
30
40
50


```
[ ]:
```