Title: - Backpropagation Feedforward Neural Network

Aim/Objective: - To implement and demonstrate a backpropagation feedforward neural network for training and testing on sample data.

Software Required:

- Python programming environment (Jupyter Notebook, Google Colab, or any Python IDE)
- NumPy library
- Matplotlib library

Hardware Required:

- 4GB RAM
- Intel i3 or higher / AMD equivalent
- 120 GB SSD
- GPU for accelerated computations (if using deep learning frameworks)

Theory:
A feedforward neural network is an artificial neural network where connections between nodes do not form a cycle. It consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction—forward—from input to output.

Backpropagation is a training algorithm that adjusts weights and biases by minimizing the error through gradient descent. It has the following steps:

1. **Forward Pass:** Compute the predicted output.
2. **Error Calculation:** Find the difference between actual and predicted outputs.
3. **Backward Pass:** Use the gradient of the error to adjust weights and biases.
4. **Repeat:** Iterate until the error is minimized.

Procedure:

1. Define input and expected output values.
2. Initialize weights and biases randomly.
3. Define the activation function (Sigmoid for non-linearity).
4. Perform the forward pass to compute predictions.
5. Compute the error and use backpropagation to adjust weights.
6. Repeat for multiple epochs until the network converges.
7. Test the trained network on sample data.

Code:

```
import numpy as np

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Sample Training Data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Inputs
y = np.array([[0], [1], [1], [0]])  # Expected Outputs

# Network Architecture
input_neurons = 2
hidden_neurons = 2
output_neurons = 1

# Randomly Initialize Weights and Biases
np.random.seed(42)
hidden_weights = np.random.uniform(size=(input_neurons, hidden_neurons))
hidden_bias = np.random.uniform(size=(1, hidden_neurons))
output_weights = np.random.uniform(size=(hidden_neurons, output_neurons))
output_bias = np.random.uniform(size=(1, output_neurons))

# Training Parameters
learning_rate = 0.5
epochs = 10000

# Training Loop
for epoch in range(epochs):
    # Forward Pass
    hidden_input = np.dot(X, hidden_weights) + hidden_bias
    hidden_output = sigmoid(hidden_input)
    final_input = np.dot(hidden_output, output_weights) + output_bias
    final_output = sigmoid(final_input)

    # Compute Error
    error = y - final_output

    # Backpropagation
    d_output = error * sigmoid_derivative(final_output)
    error_hidden = d_output.dot(output_weights.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    # Update Weights and Biases
    output_weights += hidden_output.T.dot(d_output) * learning_rate
```

```
    output_bias += np.sum(d_output, axis=0, keepdims=True) * learning_rate
    hidden_weights += X.T.dot(d_hidden) * learning_rate
    hidden_bias += np.sum(d_hidden, axis=0, keepdims=True) * learning_rate

# Testing the trained network
print("Trained Neural Network Output:")
hidden_input = np.dot(X, hidden_weights) + hidden_bias
hidden_output = sigmoid(hidden_input)
final_input = np.dot(hidden_output, output_weights) + output_bias
final_output = sigmoid(final_input)
print(final_output.round())
```

Output:

```
Trained Neural Network Output:
[[0.]
 [1.]
 [1.]
 [0.]]
```

Observations:

- The neural network successfully learns the given dataset.
- The output approximates the expected values after sufficient training.
- Backpropagation effectively reduces the error over multiple iterations.

Conclusion:

A backpropagation feedforward neural network can learn and approximate complex patterns using gradient descent. The experiment demonstrates how weights and biases are updated iteratively to minimize error, proving the effectiveness of multi-layer perceptrons in solving non-linearly separable problems.