Title: - Backpropagation Neural Network for XOR Function

Aim/Objective: - To implement and demonstrate a backpropagation neural network for solving the XOR problem with binary inputs and outputs.

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
- GPU for accelerated computations (if using deep learning frameworks)
- 120 GB SSD

Theory:

The XOR (Exclusive OR) function is a binary classification problem where the output is true (1) if the inputs are different and false (0) if they are the same. A simple perceptron cannot solve XOR since it is not linearly separable.

A **backpropagation neural network** is a multi-layer perceptron (MLP) that uses gradient descent and backpropagation to adjust weights, enabling the network to learn non-linear patterns like XOR. The network consists of:

- **Input Layer:** Two neurons (for the two binary inputs).
- **Hidden Layer:** At least two neurons with non-linear activation (e.g., Sigmoid).
- **Output Layer:** One neuron producing binary output (0 or 1).

Backpropagation works by computing the error between predicted and actual outputs, propagating the error backward, and adjusting weights accordingly using **gradient descent**.

Procedure:

1. Define the XOR input-output pairs.
2. Initialize the network architecture (input, hidden, and output layers).
3. Assign random weights and biases.
4. Use the **Sigmoid** activation function for non-linearity.
5. Compute the forward pass to obtain predictions.

6. Compute the error between predicted and actual outputs.
7. Use backpropagation to update weights using gradient descent.
8. Repeat the training process until the network converges.
9. Test the trained network on XOR inputs and observe outputs.

Code:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# XOR Inputs and Outputs
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Initialize weights and biases
input_layer_neurons = 2
hidden_layer_neurons = 2
output_layer_neurons = 1

np.random.seed(42)  # For reproducibility
hidden_weights = np.random.uniform(size=(input_layer_neurons, 
hidden_layer_neurons))
hidden_bias = np.random.uniform(size=(1, hidden_layer_neurons))
output_weights = np.random.uniform(size=(hidden_layer_neurons, 
output_layer_neurons))
output_bias = np.random.uniform(size=(1, output_layer_neurons))

# Training parameters
learning_rate = 0.5
epochs = 10000

for epoch in range(epochs):
    # Forward Propagation
    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 XOR Neural Network Results:")
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 XOR Neural Network Results:
[[0.]
 [1.]
 [1.]
 [0.]]
```

Observations:

- The neural network successfully learns the XOR function after sufficient training.
- The outputs approximate the expected values of XOR (0, 1, 1, 0).
- Backpropagation effectively adjusts weights to reduce error.

Conclusion:

A simple multi-layer perceptron with backpropagation successfully models the XOR function, demonstrating that non-linear activation functions and hidden layers are essential for solving non-linearly separable problems. This experiment highlights the importance of **gradient descent** and **error propagation** in training neural networks.