Aim: Write a Python program to show Back Propagation Network for XOR function with binary input and output

Software requirement for Python: Jupyter Notebook

Theory:

The XOR function is a binary function that takes two binary inputs and returns a binary output based on the logical operation of "exclusive OR." The truth table for the XOR function (Fig.1.) is as follows:

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

Fig.1. The truth table for the XOR function

To create a backpropagation neural network for XOR function, we first need to define the architecture of the network. We can use a feedforward neural network with one hidden layer. The input layer will have two neurons, representing the two binary inputs. The hidden layer will have two or more neurons with a sigmoid activation function. Finally, the output layer will have one neuron with a sigmoid activation function to represent the binary output.

To train the network, we need a set of training data consisting of the binary inputs and outputs for the XOR function. We can use stochastic gradient descent with backpropagation to train the network. The backpropagation algorithm involves the following steps:

1. Feed the input values forward through the network and calculate the output.

2. Calculate the error between the output and the target output.

3. Calculate the derivative of the error with respect to the output neuron.

4. Calculate the derivative of the output neuron with respect to the hidden layer neurons.

5. Calculate the derivative of the hidden layer neurons with respect to the input layer neurons.

6. Use the chain rule to multiply the derivatives together to obtain the partial derivatives of the error with respect to the weights in the network.

7. Update the weights in the network using the partial derivatives and the learning rate.

8. Repeat steps 1-7 for each training example in the dataset, adjusting the weights after each example.

9. Repeat steps 1-8 for a fixed number of epochs or until the error on the training data is minimized.

By adjusting the weights in the network using backpropagation, the network will learn to approximate the XOR function for binary inputs and outputs.

Procedure:

This code implements a backpropagation neural network for solving the XOR function with binary input and output.

First, the sigmoid activation function and its derivative are defined. Then, the XOR function dataset is created and the weights and biases are initialized with random values.

The network is trained using a loop that performs forward propagation, backpropagation, and weight and bias updates for a specified number of iterations. The learning rate is also set in this loop.

Finally, the trained network is tested by applying the sigmoid function to the dot product of the input with the weights and adding the bias for each layer. The predicted output is printed by rounding the test output to the nearest integer.

Conclusion: Thus, we have studied Back Propagation Neural Network for XOR function with binary input and output.

Program Code and Output

```
import numpy as np
# Define the sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Define the derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
# Define the XOR function dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
# Set the random seed for reproducibility
np.random.seed(42)
# Initialize the weights and biases with random values
weights1 = np.random.rand(2, 2)
bias1 = np.random.rand(1, 2)
weights2 = np.random.rand(2, 1)
bias2 = np.random.rand(1, 1)
# Set the learning rate and number of iterations
learning_rate = 0.1
num_iterations = 10000
# Training loop
for iteration in range(num_iterations):
    # Forward propagation
    layer1_output = sigmoid(np.dot(X, weights1) + bias1)
    layer2_output = sigmoid(np.dot(layer1_output, weights2) + bias2)
```

```
  # Backpropagation
    layer2_error = y - layer2_output
    layer2_delta = layer2_error * sigmoid_derivative(layer2_output)
    layer1_error = np.dot(layer2_delta, weights2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1_output)

    # Update weights and biases
    weights2 += np.dot(layer1_output.T, layer2_delta) * learning_rate
    bias2 += np.sum(layer2_delta, axis=0, keepdims=True) * learning_rate
    weights1 += np.dot(X.T, layer1_delta) * learning_rate
    bias1 += np.sum(layer1_delta, axis=0, keepdims=True) * learning_rate

# Test the trained network
test_output = sigmoid(np.dot(sigmoid(np.dot(X, weights1) + bias1), weights2) + bias2)
print("Predicted Output:")
print(test_output.round())
```

OUTPUT OF THE CODE

Predicted Output:

```
[[0.]
 [1.]
 [1.]
 [0.]]
```