Prac9

```
import numpy as np

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

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

class BackpropagationNN:
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes
        self.learning_rate = learning_rate

        self.weights_input_hidden = np.random.rand(self.input_nodes, self.hidden_nodes) - 0.5
        self.weights_hidden_output = np.random.rand(self.hidden_nodes, self.output_nodes) - 0.5
        self.bias_hidden = np.random.rand(1, self.hidden_nodes) - 0.5
        self.bias_output = np.random.rand(1, self.output_nodes) - 0.5

    def forward_propagate(self, X):
        self.hidden_input = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = sigmoid(self.hidden_input)
        self.final_input = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.final_output = sigmoid(self.final_input)
        return self.final_output

    def backpropagate(self, X, y):
        output_error = y - self.final_output
        output_delta = output_error * sigmoid_derivative(self.final_output)

        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden_output)

        self.weights_hidden_output += self.learning_rate * np.dot(self.hidden_output.T, output_delta)
        self.bias_output += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_input_hidden += self.learning_rate * np.dot(X.T, hidden_delta)
        self.bias_hidden += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

# XOR Training Data
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([[0],
              [1],
              [1],
              [0]])

# Create and Train Neural Network
nn = BackpropagationNN(input_nodes=2, hidden_nodes=4, output_nodes=1, learning_rate=1)

print("\nTRAINING PROCESS:")
reached_exact_output = False
iteration = 0
```

```
while not reached_exact_output:
    iteration += 1
    outputs = nn.forward_propagate(X)
    nn.backpropagate(X, y)

    print(f"\nITERATION {iteration}")
    reached_exact_output = True

    for inp, expected, output in zip(X, y, outputs):
        difference = expected - output
        gradient_output = sigmoid_derivative(output)
        print(f"INPUT: {inp}, EXPECTED: {expected}, OUTPUT: {output}, GRADIENT OUTPUT:
{gradient_output}")

        if not np.isclose(output, expected, atol=0.01):
            reached_exact_output = False
```