Experiment 4

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 sigmoid(x) * (1 - sigmoid(x))

# Mean Squared Error loss function and its derivative
def mse_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def mse_loss_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size

class NeuralNetwork:
    def __init__(self, input_size, hidden1_size, hidden2_size,
output_size):
        # Initialize weights and biases
        self.W1 = np.random.randn(input_size, hidden1_size)
        self.b1 = np.zeros((1, hidden1_size))
        self.W2 = np.random.randn(hidden1_size, hidden2_size)
        self.b2 = np.zeros((1, hidden2_size))
        self.W3 = np.random.randn(hidden2_size, output_size)
        self.b3 = np.zeros((1, output_size))

    def forward(self, X):
        # Forward propagation
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = sigmoid(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = sigmoid(self.z2)
        self.z3 = np.dot(self.a2, self.W3) + self.b3
        self.a3 = sigmoid(self.z3)
        return self.a3

    def backward(self, X, y, learning_rate):
        # Compute the gradients for backpropagation
        m = y.shape[0]

        # Output layer
```

```
        d_a3 = mse_loss_derivative(y, self.a3) *
sigmoid_derivative(self.z3)
        d_W3 = np.dot(self.a2.T, d_a3) / m
        d_b3 = np.sum(d_a3, axis=0, keepdims=True) / m

        # Second hidden layer
        d_a2 = np.dot(d_a3, self.W3.T) * sigmoid_derivative(self.z2)
        d_W2 = np.dot(self.a1.T, d_a2) / m
        d_b2 = np.sum(d_a2, axis=0, keepdims=True) / m

        # First hidden layer
        d_a1 = np.dot(d_a2, self.W2.T) * sigmoid_derivative(self.z1)
        d_W1 = np.dot(X.T, d_a1) / m
        d_b1 = np.sum(d_a1, axis=0, keepdims=True) / m

        # Update weights and biases
        self.W1 -= learning_rate * d_W1
        self.b1 -= learning_rate * d_b1
        self.W2 -= learning_rate * d_W2
        self.b2 -= learning_rate * d_b2
        self.W3 -= learning_rate * d_W3
        self.b3 -= learning_rate * d_b3

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            y_pred = self.forward(X)

            # Compute loss
            loss = mse_loss(y, y_pred)
            if epoch % 100 == 0:
                print(f'Epoch {epoch}, Loss: {loss}')

            # Backward pass
            self.backward(X, y, learning_rate)

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])  # XOR problem

    # Initialize and train the neural network
nn = NeuralNetwork(input_size=2, hidden1_size=4, hidden2_size=4,
output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

    # Test the neural network
predictions = nn.forward(X)
print("Predictions:\n", predictions)
```

Output:

```
Epoch 0, Loss: 0.27470341494333644
Epoch 100, Loss: 0.2581455971756465
Epoch 200, Loss: 0.2520815594287613
Epoch 300, Loss: 0.2500711822433604
Epoch 400, Loss: 0.24940564884007405
Epoch 500, Loss: 0.24916536488433053
Epoch 600, Loss: 0.24905772185718672
Epoch 700, Loss: 0.24899128072178972
Epoch 800, Loss: 0.2489375844063635
Epoch 900, Loss: 0.2488877697886518
Predictions:
 [[0.48398123]
 [0.52804258]
 [0.47320395]
 [0.51074506]]
```