Backpropogation

```
import numpy as np

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


class NeuralNetwork:
  def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
    self.input_size = input_size
    self.hidden_size1 = hidden_size1
    self.hidden_size2 = hidden_size2
    self.output_size = output_size

    self.weights_input_hidden1 = np.random.rand(self.input_size, self.hidden_size1)
    self.bias_hidden1 = np.zeros((1, self.hidden_size1))

    self.weights_hidden1_hidden2 = np.random.rand(self.hidden_size1, self.hidden_size2)
    self.bias_hidden2 = np.zeros((1, self.hidden_size2))

    self.weights_hidden2_output = np.random.rand(self.hidden_size2, self.output_size)
    self.bias_output = np.zeros((1, self.output_size))

  def forward(self, X):
    self.hidden1_output = sigmoid(np.dot(X, self.weights_input_hidden1) + self.bias_hidden1)
    self.hidden2_output = sigmoid(np.dot(self.hidden1_output, self.weights_hidden1_hidden2) + self.bias_hidden2)
    self.output = sigmoid(np.dot(self.hidden2_output, self.weights_hidden2_output) + self.bias_output)
    return self.output

  def backward(self, X, y, output):
    error = y - output

    d_output = error * sigmoid_derivative(output)
    error_hidden2 = d_output.dot(self.weights_hidden2_output.T)
    d_hidden2 = error_hidden2 * sigmoid_derivative(self.hidden2_output)

    error_hidden1 = d_hidden2.dot(self.weights_hidden1_hidden2.T)
    d_hidden1 = error_hidden1 * sigmoid_derivative(self.hidden1_output)

    self.weights_hidden2_output += self.hidden2_output.T.dot(d_output)
    self.bias_output += np.sum(d_output, axis=0, keepdims=True)

    self.weights_hidden1_hidden2 += self.hidden1_output.T.dot(d_hidden2)
    self.bias_hidden2 += np.sum(d_hidden2, axis=0, keepdims=True)

    self.weights_input_hidden1 += X.T.dot(d_hidden1)
    self.bias_hidden1 += np.sum(d_hidden1, axis=0, keepdims=True)

  def train(self, X, y, epochs):
      for epoch in range(epochs):
        output = self.forward(X)
        self.backward(X, y, output)
        loss = np.mean(np.square(y - output))
        if epoch % 100 == 0:
          print(f"Epoch {epoch}, Loss: {loss:.4f}")


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


input_size = X.shape[1]
hidden_size1 = 4
hidden_size2 = 4
output_size = y.shape[1]
model = NeuralNetwork(input_size, hidden_size1, hidden_size2, output_size)
model.train(X, y, epochs=1000)
```

```
Epoch 0, Loss: 0.4118
Epoch 100, Loss: 0.2498
Epoch 200, Loss: 0.2497
Epoch 300, Loss: 0.2495
Epoch 400, Loss: 0.2492
Epoch 500, Loss: 0.2485
Epoch 600, Loss: 0.2471
Epoch 700, Loss: 0.2431
Epoch 800, Loss: 0.2294
Epoch 900, Loss: 0.1964
```

```
test_output = model.forward(X)
print("Test Output:")
print(test_output)
```

```
Test Output:
[[0.14504423]
 [0.64114697]
 [0.64174524]
 [0.64338423]]
```