CODE:

```
# Import numpy for numerical computations
import numpy as np

# Define the sigmoid activation function
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Define the derivative of the sigmoid function
def sigmoid_prime(z):
    return sigmoid(z) * (1 - sigmoid(z))

# Define the XOR input and output data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input
Y = np.array([[0], [1], [1], [0]]) # Output

# Define the network parameters
input_size = 2 # Number of input neurons
hidden_size = 2 # Number of hidden neurons
output_size = 1 # Number of output neurons
learning_rate = 0.1 # Learning rate
epochs = 10000 # Number of training iterations

# Initialize the network weights and biases randomly
W1 = np.random.randn(hidden_size, input_size) # Weights from input to hidden
layer
W2 = np.random.randn(output_size, hidden_size) # Weights from hidden to output
layer
b1 = np.random.randn(hidden_size, 1) # Bias for hidden layer
b2 = np.random.randn(output_size, 1) # Bias for output layer

# Train the network using back propagation
for epoch in range(epochs):
    # Forward propagation
    Z1 = np.dot(W1, X.T) + b1 # Linear combination for hidden layer
    A1 = sigmoid(Z1) # Activation for hidden layer
    Z2 = np.dot(W2, A1) + b2 # Linear combination for output layer
    A2 = sigmoid(Z2) # Activation for output layer

    # Compute the cost function
    cost = -np.sum(Y.T * np.log(A2) + (1 - Y.T) * np.log(1 - A2)) / 4

    # Print the cost every 1000 iterations
    if epoch % 1000 == 0:
        print(f"Cost at epoch {epoch}: {cost}")

    # Backward propagation
    dZ2 = A2 - Y.T # Derivative of cost with respect to Z2
    dW2 = np.dot(dZ2, A1.T) / 4 # Derivative of cost with respect to W2
    db2 = np.sum(dZ2, axis=1, keepdims=True) / 4 # Derivative of cost with
respect to b2
```

```
    dA1 = np.dot(W2.T, dZ2) # Derivative of cost with respect to A1
    dZ1 = dA1 * sigmoid_prime(Z1) # Derivative of cost with respect to Z1
    dW1 = np.dot(dZ1, X) / 4 # Derivative of cost with respect to W1
    db1 = np.sum(dZ1, axis=1, keepdims=True) / 4 # Derivative of cost with
respect to b1

    # Update the weights and biases
    W1 = W1 - learning_rate * dW1
    W2 = W2 - learning_rate * dW2
    b1 = b1 - learning_rate * db1
    b2 = b2 - learning_rate * db2

# Test the network on the input data
print("Testing the network:")
for x, y in zip(X, Y):
    z1 = np.dot(W1, x) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(W2, a1) + b2
    a2 = sigmoid(z2)
    print(f"Input: {x}, Output: {a2}, Expected: {y}")
```

OUTPUT:

```
Cost at epoch 0: 0.8087653546919911
Cost at epoch 1000: 0.6928640969315474
Cost at epoch 2000: 0.6891986159390826
Cost at epoch 3000: 0.6555015616619227
Cost at epoch 4000: 0.4936893510006434
Cost at epoch 5000: 0.16409711265687588
Cost at epoch 6000: 0.07009952726682814
Cost at epoch 7000: 0.042476010340162004
Cost at epoch 8000: 0.030065840730146676
Cost at epoch 9000: 0.023137544172582708
Testing the network:
Input: [0 0], Output: [[0.01811426 0.01811426]], Expected: [0]
Input: [0 1], Output: [[0.98015443 0.97508017]], Expected: [1]
Input: [1 0], Output: [[0.98409284 0.98190037]], Expected: [1]
Input: [1 1], Output: [[0.01859382 0.01768678]], Expected: [0]
```

CODE For Graph:

```
# Import matplotlib library
import matplotlib.pyplot as plt

# Plot the input and output data
plt.plot(X, Y, 'o', label='Data') # Plot the data points as circles
plt.plot(X, A2.T, '-', label='Model') # Plot the model output as a line

# Add labels, title, legend, and grid
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('Back Propagation Network for XOR Function')
plt.legend()
plt.grid()

# Show the graph
plt.show()
```

OUTPUT: