Detailed Explanation of the Code:

This code implements a simple neural network to solve the XOR (Exclusive OR) problem. Let's break down the code step by step:

1. Importing Libraries:

Python

```
import numpy as np
```

This line imports the numpy library, which is commonly used for scientific computing in Python. It provides functions for working with arrays, matrices, and other mathematical operations.

2. Sigmoid Activation Function:

Python

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

This defines a function named sigmoid. This function takes a number x as input and applies the sigmoid activation function to it. The sigmoid function is a mathematical function that maps any real number between negative infinity and positive infinity to a value between 0 and 1. It's often used in neural networks to introduce non-linearity.

3. Sigmoid Derivative:

Python

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

This defines another function named sigmoid_derivative. This function calculates the derivative of the sigmoid function with respect to its input x. The derivative is used in the backpropagation algorithm for training the neural network.

4. XOR Dataset:

Python

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

Here, we define the XOR dataset using two NumPy arrays:

- X: This represents the input data with four rows (one for each data point) and two columns (representing the two features). Each row represents an input vector for the neural network.
- y: This represents the desired output for each input in X. Here, it defines the target values for the XOR operation on the corresponding inputs.

5. Neural Network Architecture:

Python

```
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
```

These lines define the architecture of the neural network:

- input_layer_size: This specifies the number of neurons in the input layer (2 in this case, corresponding to the two features in the input data).
- hidden_layer_size: This specifies the number of neurons in the hidden layer (2 in this case).
- output_layer_size: This specifies the number of neurons in the output layer (1 in this case, as we want a single output value).

6. Weights and Biases Initialization:

Python

```
weights1 = np.random.uniform(size=(input_layer_size,
hidden_layer_size))
bias1 = np.random.uniform(size=(1, hidden_layer_size))
weights2 = np.random.uniform(size=(hidden_layer_size,
output_layer_size))
bias2 = np.random.uniform(size=(1, output_layer_size))
```

These lines initialize the weights and biases for the neural network. Weights represent the

connections between neurons in different layers, and biases are added to the activation of each neuron.

- weights1: This is a NumPy array of size (input_layer_size, hidden_layer_size) representing the weights between the input layer and the hidden layer. It's initialized with random values using np.random.uniform.
- bias1: This is a NumPy array of size (1, hidden_layer_size) representing the biases for the neurons in the hidden layer. It's also initialized with random values.
- weights2: This is a NumPy array of size (hidden_layer_size, output_layer_size) representing the weights between the hidden layer and the output layer.
- bias2: This is a NumPy array of size (1, output_layer_size) representing the bias for the output neuron.

7. Learning Rate and Epochs:

Python

```
learning_rate = 0.1
epochs = 10000
```

These lines define two important hyperparameters for training the neural network:

- learning_rate: This controls how much the weights and biases are adjusted during each training iteration. Here, it's set to 0.1.
- epochs: This specifies the number of times the entire training dataset will be passed through the network for training. Here, it's set to 10000.

8. Training Loop (Backpropagation) - Continued:

The core part of the code lies within the for loop iterating over a number of epochs. This loop represents the training process of the neural network. Let's break down what happens inside the loop:

- **Forward Propagation:**
  - hidden_layer_output = sigmoid(np.dot(X, weights1) + bias1): This line performs the forward propagation step for the hidden layer. It calculates the weighted sum of the inputs (X) multiplied by the weights (weights1) and adds the bias (bias1). Then, it applies the sigmoid activation function (sigmoid) to this sum to get the activation values for the hidden layer neurons.
  - output_layer_output = sigmoid(np.dot(hidden_layer_output, weights2) + bias2): This line performs the forward propagation for the output layer. It calculates the weighted sum of the hidden layer outputs (hidden_layer_output) multiplied by the weights (weights2) and adds the bias (bias2). Then, it applies the sigmoid activation function again to get the final output of the network.
- **Backpropagation:**

- error = y - output_layer_output: This calculates the error between the desired output (y) and the actual output (output_layer_output) of the network.
- output_layer_delta = error * sigmoid_derivative(output_layer_output): This calculates the delta value for the output layer. The delta represents the contribution of the error to the change in weights and biases for this layer. It considers the error (error) and the derivative of the activation function (sigmoid_derivative) applied to the output layer activations.
- hidden_layer_error = output_layer_delta.dot(weights2.T): This calculates the error for the hidden layer. It backpropagates the error from the output layer to the hidden layer, considering the weights between them (weights2.T).
- hidden_layer_delta = hidden_layer_error * sigmoid_derivative(hidden_layer_output): This calculates the delta value for the hidden layer. Similar to the output layer, it considers the backpropagated error (hidden_layer_error) and the derivative of the activation function applied to the hidden layer activations.

- **Updating Weights and Biases:**
  - weights2 += hidden_layer_output.T.dot(output_layer_delta) * learning_rate: This updates the weights between the hidden layer and the output layer based on the learning rate (learning_rate), the transpose of the hidden layer outputs (hidden_layer_output.T), and the output layer delta (output_layer_delta).
  - bias2 += np.sum(output_layer_delta, axis=0, keepdims=True) * learning_rate: This updates the bias for the output neuron based on the learning rate and the sum of the output layer delta across all data points.
  - weights1 += X.T.dot(hidden_layer_delta) * learning_rate: This updates the weights between the input layer and the hidden layer based on the learning rate, the transpose of the input data (X.T), and the hidden layer delta.
  - bias1 += np.sum(hidden_layer_delta, axis=0, keepdims=True) * learning_rate: This updates the biases for the hidden layer neurons based on the learning rate and the sum of the hidden layer delta across all data points.

9. Prediction:

Python

```
predictions = sigmoid(np.dot(sigmoid(np.dot(X, weights1) + bias1), weights2) + bias2)
print(predictions.round())
```

After training, this code performs prediction on the original input data (X). It again performs forward propagation through the network using the trained weights and biases. Finally, it applies the sigmoid function to the final output and rounds the values to get the predicted outputs for the XOR function. The print statement displays these rounded predictions, which ideally should match the target values in y.

This code demonstrates a basic implementation of a neural network with a single hidden layer,

trained using backpropagation to solve the XOR problem. By iteratively adjusting the weights and biases based on the error, the network learns to map the input patterns to the desired outputs.