```
import numpy as np

# Input features for AND gate
X = np.array([[0, 0],  # Input (0, 0)
              [0, 1],  # Input (0, 1)
              [1, 0],  # Input (1, 0)
              [1, 1]]) # Input (1, 1)

# Output for AND gate (Expected output)
Y = np.array([[0],  # Output for (0, 0)
              [0],  # Output for (0, 1)
              [0],  # Output for (1, 0)
              [1]]) # Output for (1, 1)

# Initialize weights and bias
w = np.random.randn(2, 1)  # Two weights (for two inputs)
b = np.random.randn()      # Bias

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

# Define the binary cross-entropy loss function (for monitoring the loss)
def binary_cross_entropy(Y_actual, Y_predicted):
    return -np.mean(Y_actual * np.log(Y_predicted) + (1 - Y_actual) * np.log(1 - Y_predicted))

# Hyperparameters
learning_rate = 0.1
epochs = 2000  # You may need to adjust this to ensure convergence

# Training loop
for epoch in range(epochs):
    # Forward pass: compute the output
    z = np.dot(X, w) + b        # Linear combination
    y_pred = sigmoid(z)         # Sigmoid activation (probabilities)

    # Compute the loss
    loss = binary_cross_entropy(Y, y_pred)

    # Backward pass (Gradient Descent)
    dz = y_pred - Y              # Derivative of loss w.r.t. z
    dw = np.dot(X.T, dz) / X.shape[0]  # Gradient w.r.t weights
    db = np.sum(dz) / X.shape[0]       # Gradient w.r.t bias

    # Update weights and bias using gradient descent
    w -= learning_rate * dw
    b -= learning_rate * db

    # Print loss every 100 epochs for monitoring
    if epoch % 100 == 0:
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {loss:.4f}")

# Final weights and bias after training
print("\nFinal weights and bias:")
print("Weights:", w)
print("Bias:", b)

# Testing the model
test_X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Test input for AND gate
test_pred = sigmoid(np.dot(test_X, w) + b)  # Predictions using learned weights and bias
test_class = (test_pred > 0.5).astype(int)  # Classify based on threshold of 0.5

# Display the test results
print("\nTest Predictions:")
print("Input:", test_X)
print("Predicted Probabilities:", test_pred.flatten())
print("Predicted Classes:", test_class.flatten())
```

```
Epoch 1/2000, Loss: 0.7017
Epoch 101/2000, Loss: 0.4305
Epoch 201/2000, Loss: 0.3382
Epoch 301/2000, Loss: 0.2825
Epoch 401/2000, Loss: 0.2445
Epoch 501/2000, Loss: 0.2163
Epoch 601/2000, Loss: 0.1943
Epoch 701/2000, Loss: 0.1765
Epoch 801/2000, Loss: 0.1617
Epoch 901/2000, Loss: 0.1492
Epoch 1001/2000, Loss: 0.1385
Epoch 1101/2000, Loss: 0.1292
Epoch 1201/2000, Loss: 0.1210
```

```
Epoch 1301/2000, Loss: 0.1138
Epoch 1401/2000, Loss: 0.1074
Epoch 1501/2000, Loss: 0.1016
Epoch 1601/2000, Loss: 0.0964
Epoch 1701/2000, Loss: 0.0917
Epoch 1801/2000, Loss: 0.0874
Epoch 1901/2000, Loss: 0.0835

Final weights and bias:
Weights: [[4.28023764]
 [4.28288377]]
Bias: -6.613719704331315

Test Predictions:
Input: [[0 0]
 [0 1]
 [1 0]
 [1 1]]
Predicted Probabilities: [0.00134003 0.08860114 0.08838769 0.87538139]
Predicted Classes: [0 0 0 1]
```