Munaga Sai Snehitha BL.EN.U4CE21126 CSE B

In [24]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from math import exp
from sklearn.model_selection import train_test_split
import io
from sklearn import datasets
from sklearn import metrics
from sklearn.neural_network import MLPClassifier
from sklearn.neural_network import MLPRegressor
import seaborn as sns
```

In [25]:
```
# Define the input and output variables for the AND gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 0, 0, 1])
```

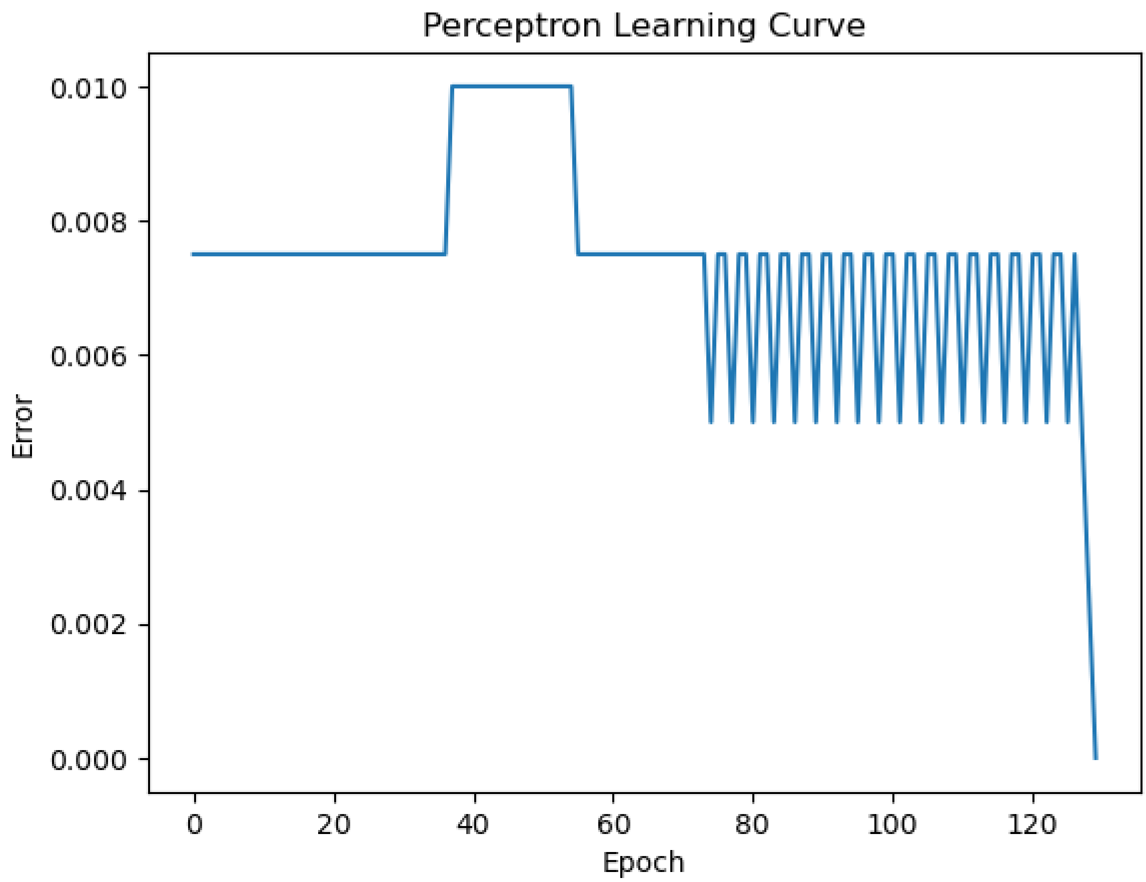
In [26]:
```
# Define the step activation function
def step_activation(x):
    return 0 if x < 0 else 1
# Define the Bi-Polar Step activation function
def bipolar_step_activation(x):
    return -1 if x < 0 else 1
# Define the Sigmoid activation function
def sigmoid_activation(x):
    return 1 / (1 + np.exp(-x))
# Define the ReLU activation function
def relu_activation(x):
    return max(0, x)
```

In [27]:
```
# Define the initial weights and learning rate
W0 = 10
W1 = 0.2
W2 = -0.75
alpha = 0.05

# Define the maximum number of epochs and convergence error
max_epochs = 1000
convergence_error = 0.002

# Initialize the error and epoch lists
errors = []
epoch_list = []
```

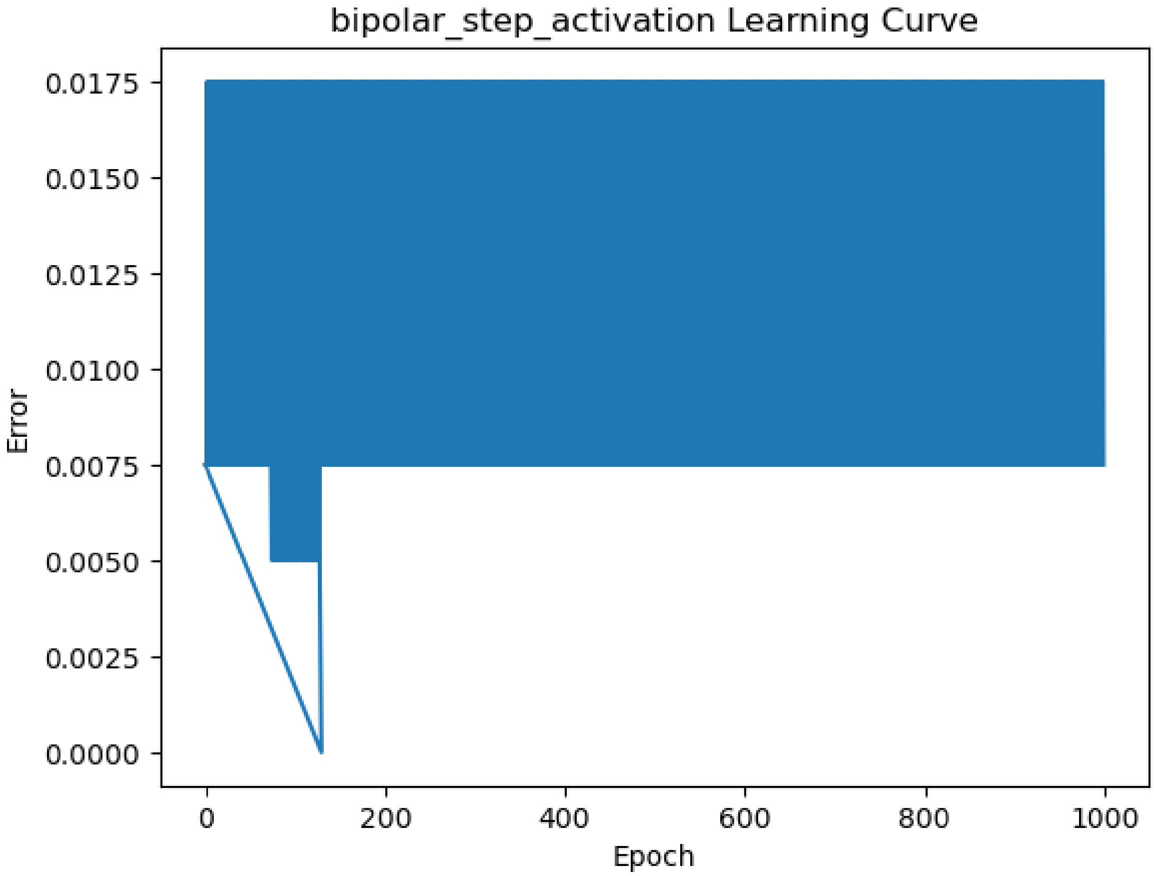
In [28]:

```
# Train the perceptron model step activation
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = step_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("Converged after", epoch, "epochs")
        break
print(f"Final Weights: W0 = {W0}, W1 = {W1}, W2 = {W2}")
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Perceptron Learning Curve')
plt.show()
```

```
Converged after 129 epochs
Final Weights: W0 = -0.10000000000000765, W1 = 0.1000000000000001, W2 = 0.050
00000000000032
```

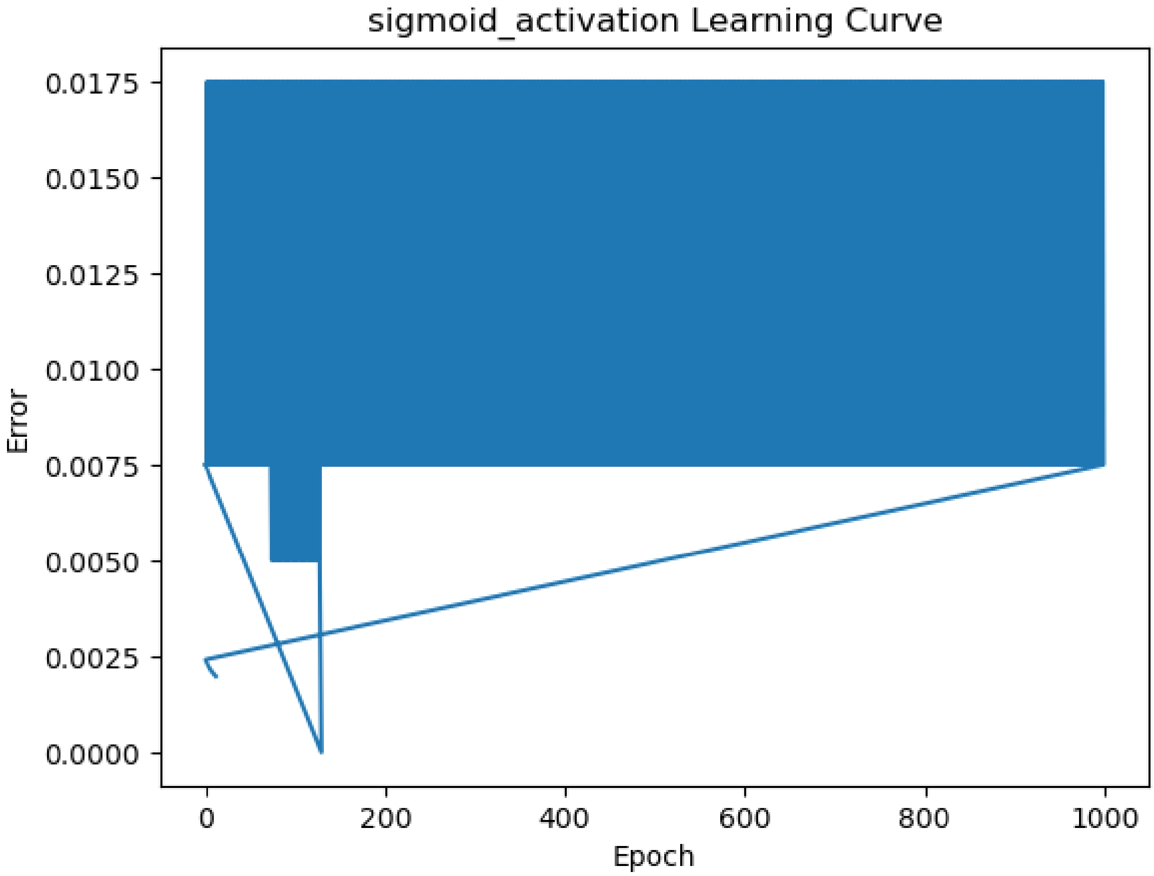

Perceptron Learning Curve
0.010
0.008
0.006
0.004
0.002
0.000
Error
0
20
40
60
80
100
120
Epoch

In [29]:
```
# Train the perceptron model step activation
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = bipolar_step_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("bipolar_step_activation after", epoch, "epochs")
        break
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('bipolar_step_activation Learning Curve')
plt.show()
```


bipolar_step_activation Learning Curve
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
Error
0
200
400
600
800
1000
Epoch

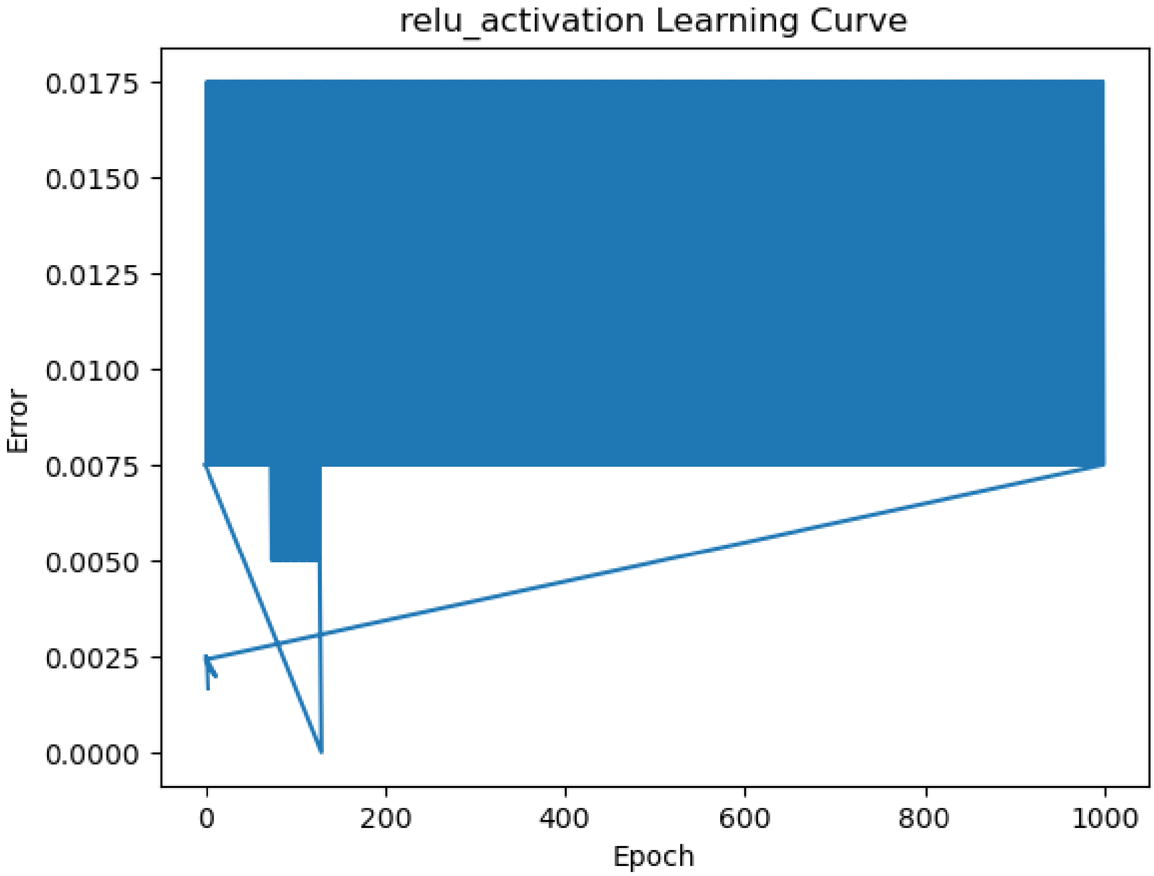
In [30]:
```
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = sigmoid_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("sigmoid_activation: Converged after", epoch, "epochs")
        break

# Plot the epochs against the error values
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('sigmoid_activation Learning Curve')
plt.show()
```

```
sigmoid_activation: Converged after 12 epochs
```


sigmoid_activation Learning Curve
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
Error
0
200
400
600
800
1000
Epoch

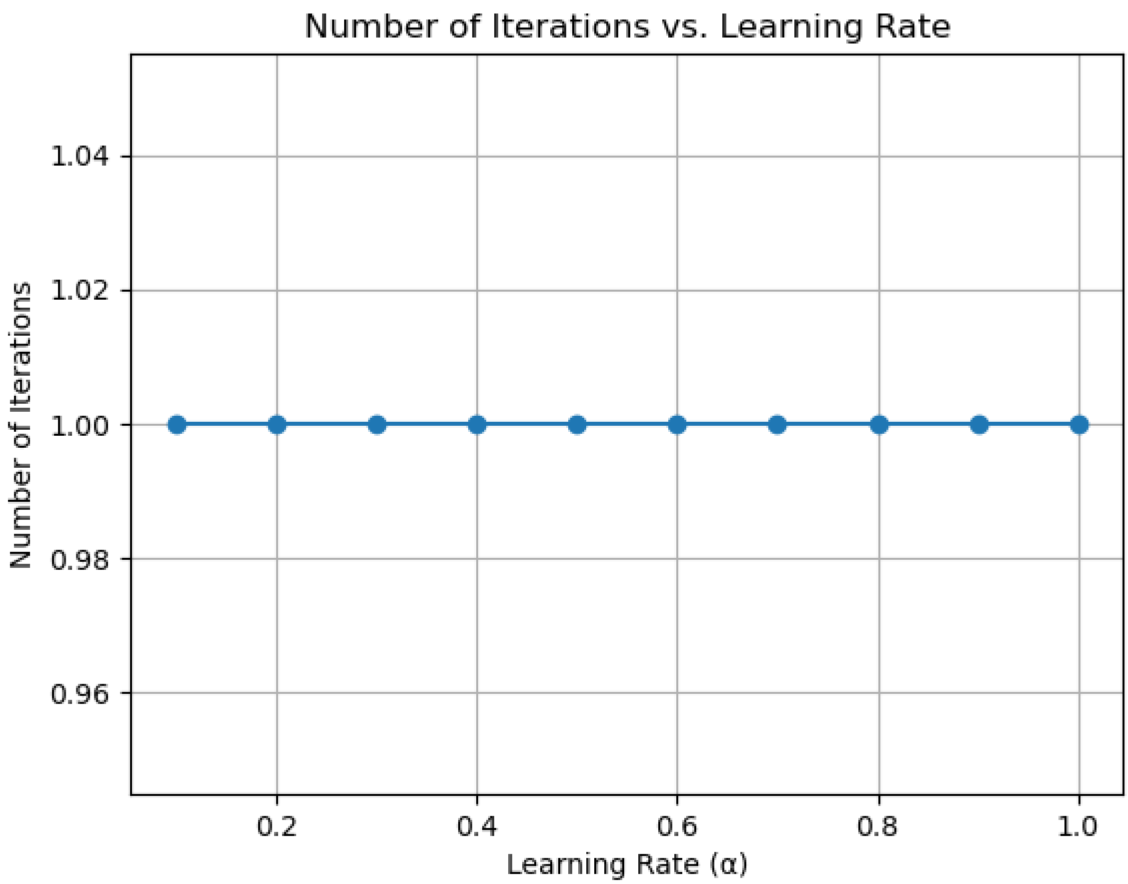
In [31]:
```
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = relu_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("relu_activation: Converged after", epoch, "epochs")
        break

# Plot the epochs against the error values
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('relu_activation Learning Curve')
plt.show()
```

```
relu_activation: Converged after 3 epochs
```


relu_activation Learning Curve
Error
Epoch
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0
200
400
600
800
1000

In [ ]:
```
# Initialize the learning rates and corresponding iteration counts
learning_rates = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
iteration_counts = []

# Perform the experiment for each learning rate
for alpha in learning_rates:
    # Initialize weights for each learning rate
    W0_temp = W0
    W1_temp = W1
    W2_temp = W2

    # Initialize the error and epoch lists
    errors = []

    # Train the perceptron model
    for epoch in range(max_epochs):
        error = 0
        for i in range(len(X)):
            # Calculate the predicted output
            weighted_sum = W0_temp + W1_temp * X[i][0] + W2_temp * X[i][1]
            y_pred = step_activation(weighted_sum)

            # Calculate the error
            delta = alpha * (y[i] - y_pred)

            # Update the weights
            W0_temp += delta
            W1_temp += delta * X[i][0]
            W2_temp += delta * X[i][1]

            # Calculate the sum-square-error
            error += delta**2

        # Check for convergence
        if error <= convergence_error:
            break

    # Append the number of iterations to the iteration_counts list
    iteration_counts.append(epoch + 1)  # Add 1 to account for 0-based indexin
```

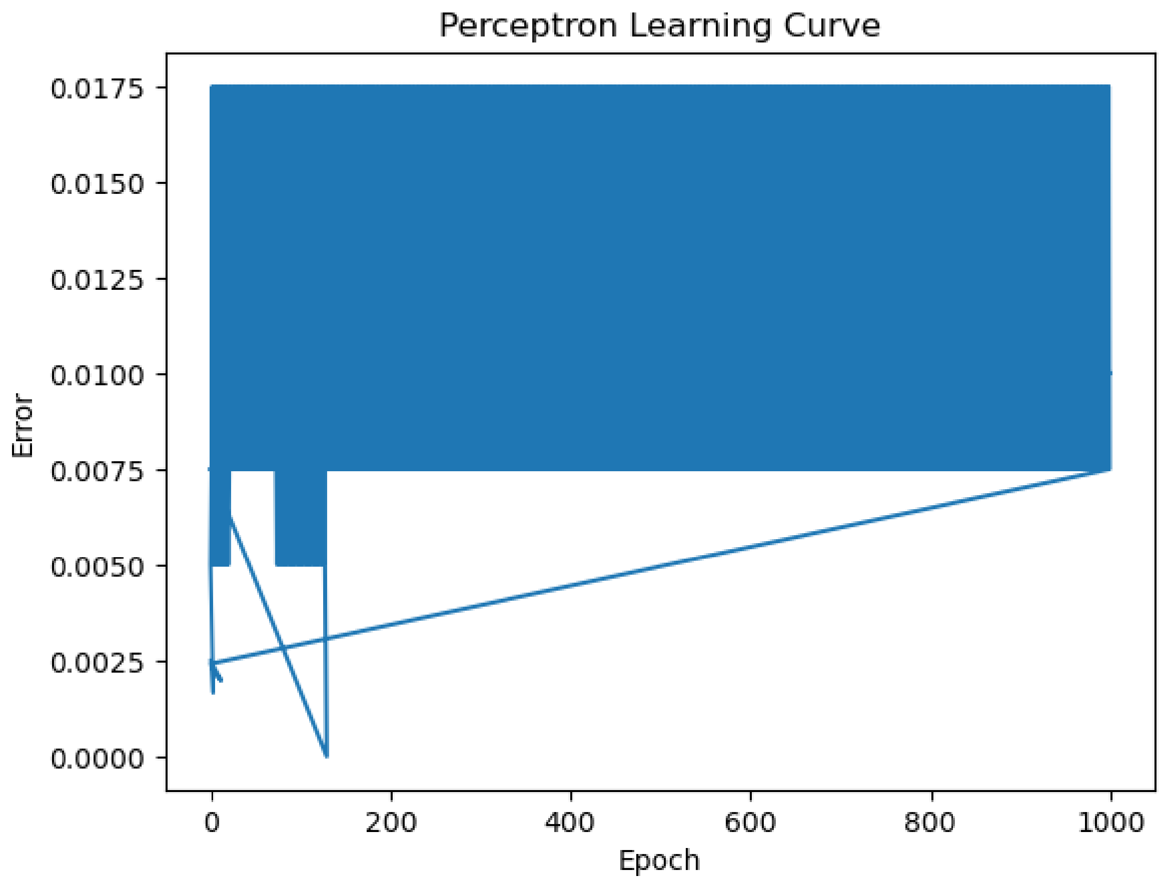
In [32]:
```
# Plot the learning rate vs. number of iterations
plt.plot(learning_rates, iteration_counts, marker='o')
plt.xlabel('Learning Rate (α)')
plt.ylabel('Number of Iterations')
plt.title('Number of Iterations vs. Learning Rate')
plt.grid()
plt.show()
```



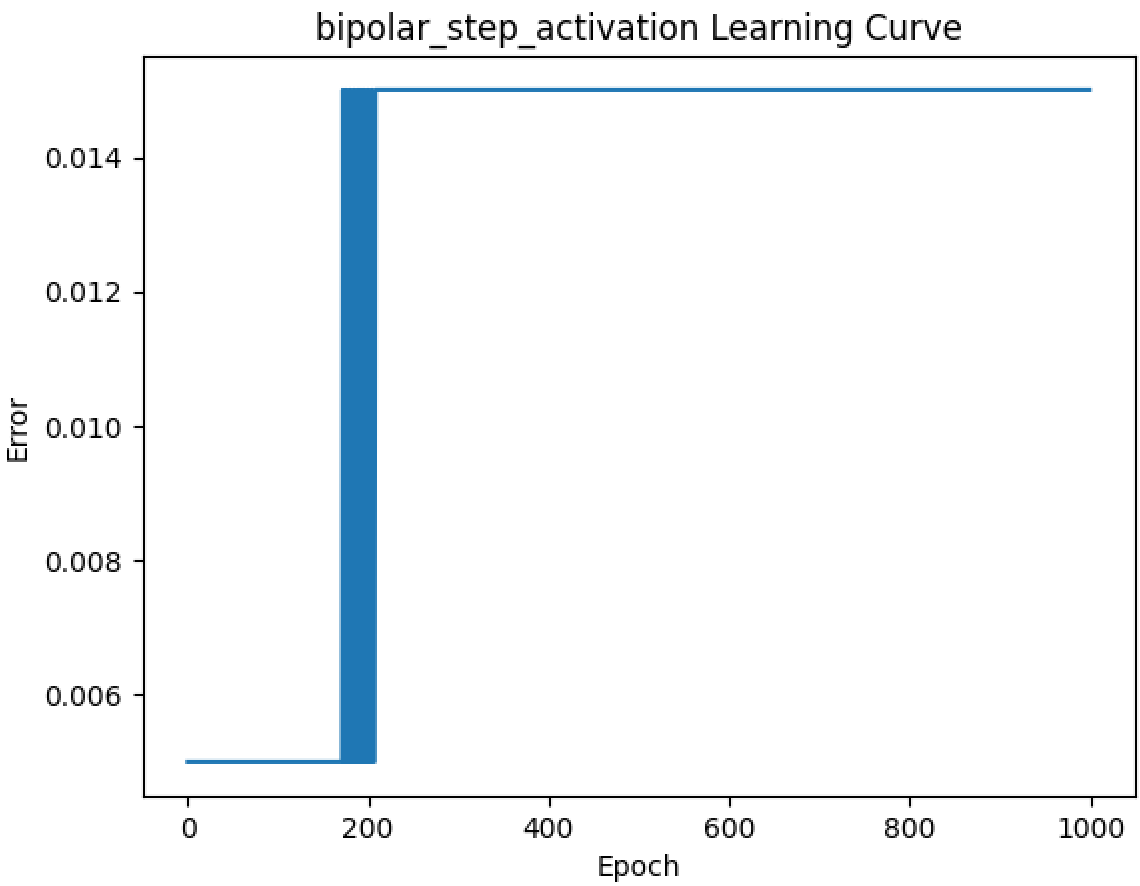
In [ ]:
```
#for XOR gate
```

In [33]:
```
# Define the input and output variables for the XOR gate
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([0, 1, 1, 0])
```

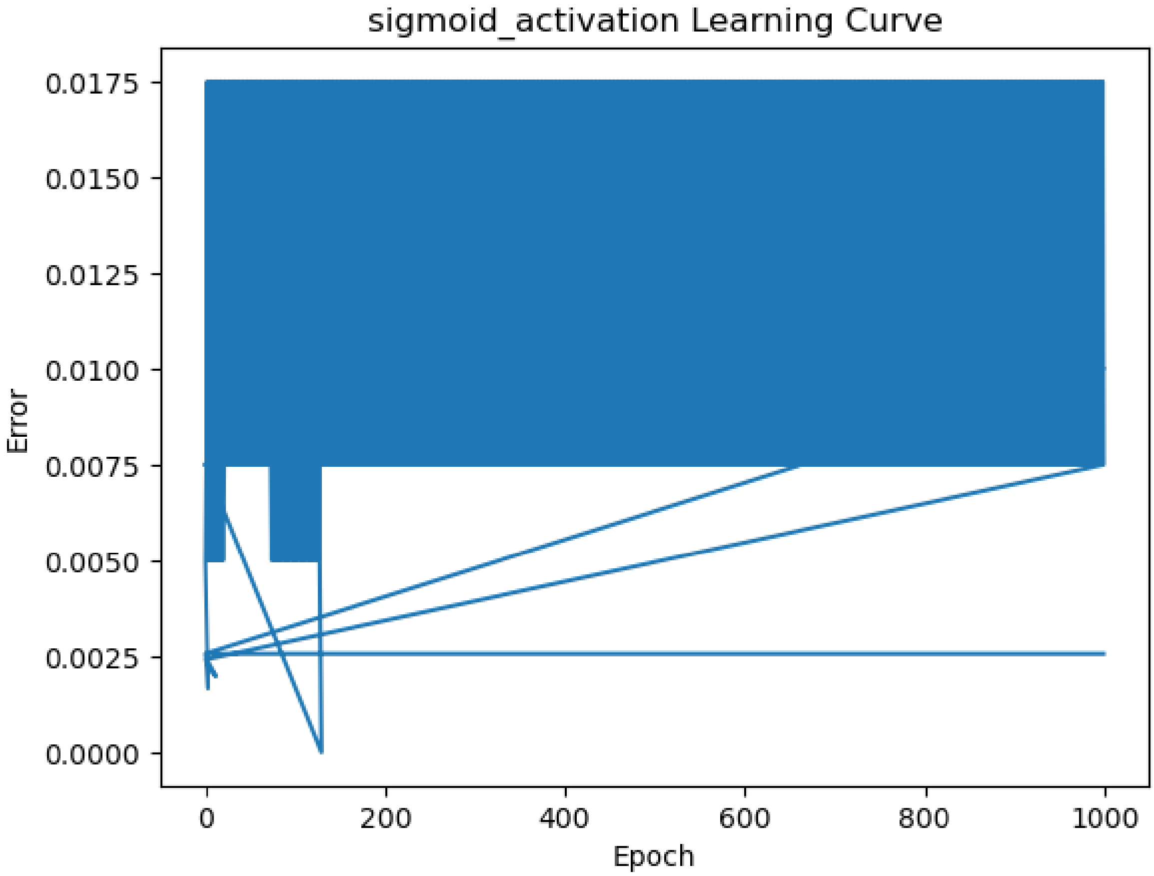
In [34]:
```
# Train the perceptron model step activation
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = step_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("Converged after", epoch, "epochs")
        break
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Perceptron Learning Curve')
plt.show()
```


Perceptron Learning Curve
Error
Epoch
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
200
400
600
800
1000

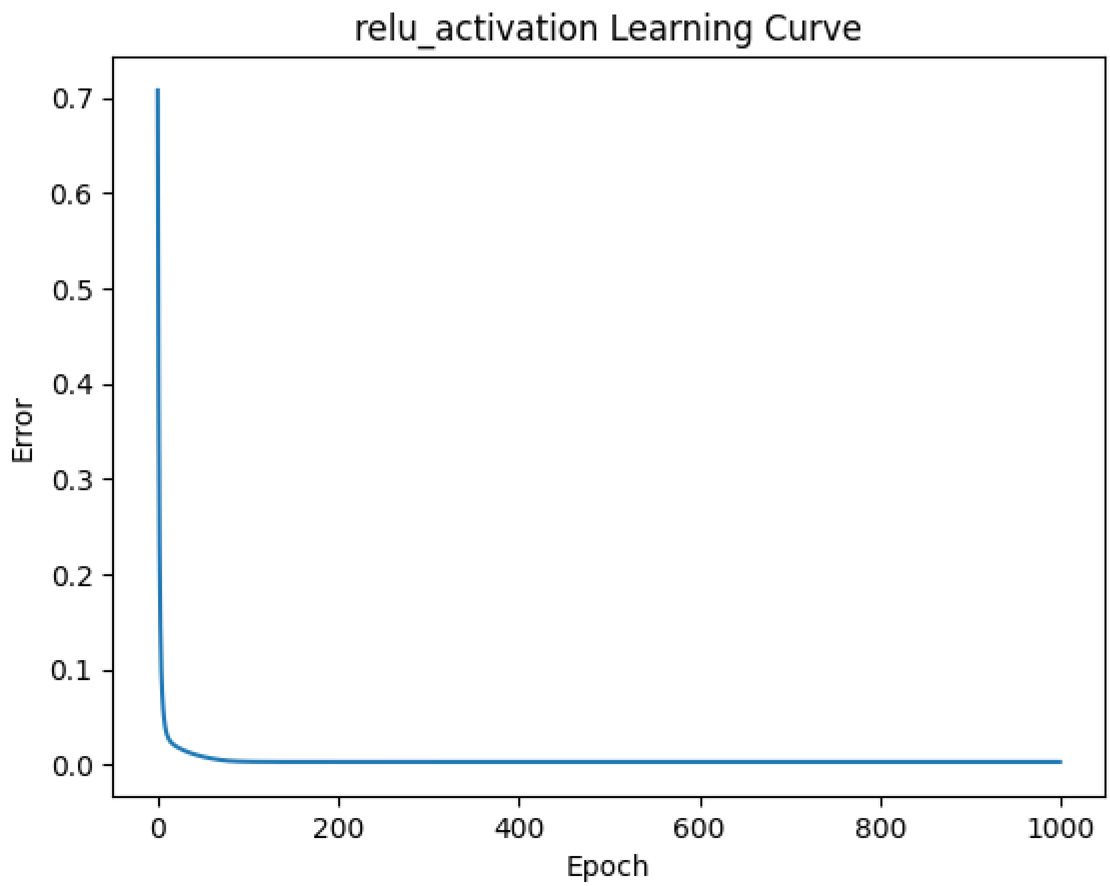
In [ ]:
```
# Train the perceptron model step activation
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = bipolar_step_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("bipolar_step_activation after", epoch, "epochs")
        break
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('bipolar_step_activation Learning Curve')
plt.show()
```


bipolar_step_activation Learning Curve
Error
0.014
0.012
0.010
0.008
0.006
Epoch
0
200
400
600
800
1000

In [35]:
```
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = sigmoid_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("sigmoid_activation: Converged after", epoch, "epochs")
        break

# Plot the epochs against the error values
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('sigmoid_activation Learning Curve')
plt.show()
```

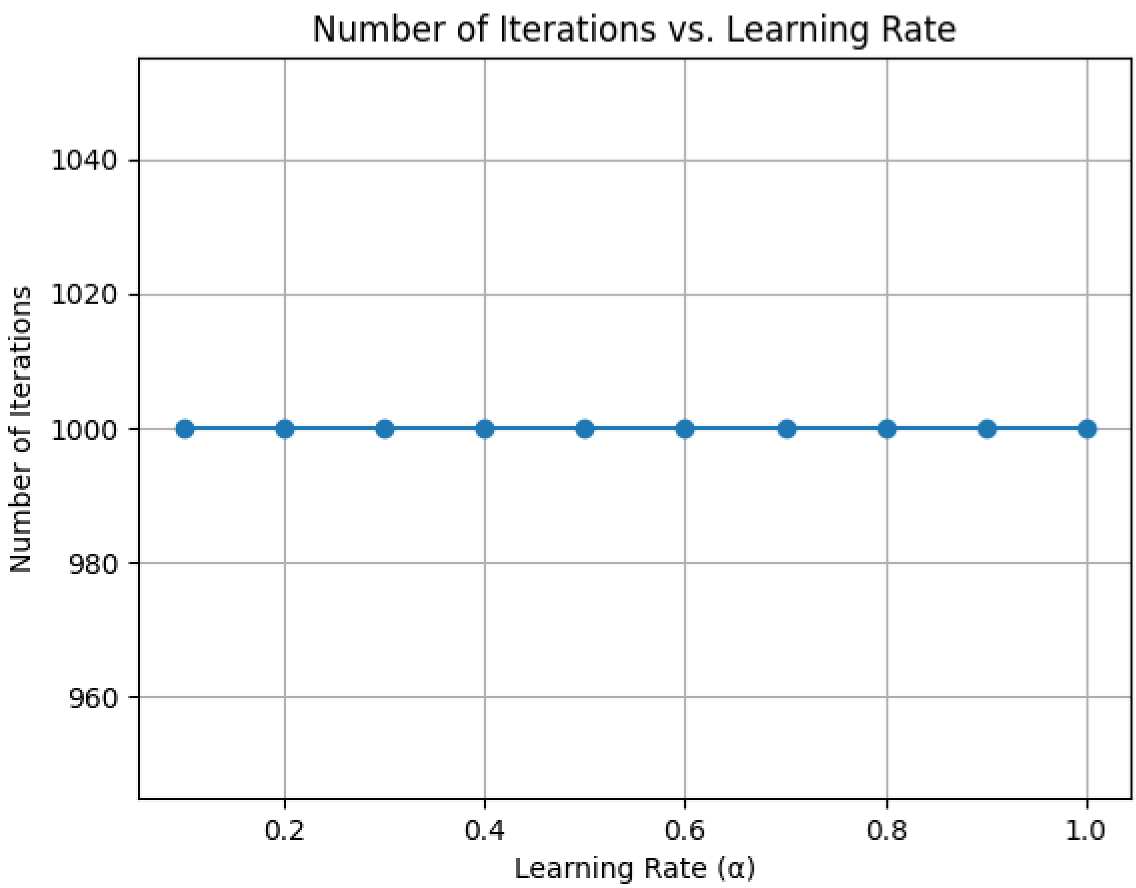

sigmoid_activation Learning Curve
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
Error
0
200
400
600
800
1000
Epoch

In [ ]:
```
for epoch in range(max_epochs):
    error = 0
    for i in range(len(X)):
        # Calculate the predicted output
        weighted_sum = W0 + W1 * X[i][0] + W2 * X[i][1]
        y_pred = relu_activation(weighted_sum)

        # Calculate the error
        delta = alpha * (y[i] - y_pred)

        # Update the weights
        W0 += delta
        W1 += delta * X[i][0]
        W2 += delta * X[i][1]

        # Calculate the sum-square-error
        error += delta**2

    # Append the error and epoch to the lists
    errors.append(error)
    epoch_list.append(epoch)

    # Check for convergence
    if error <= convergence_error:
        print("relu_activation: Converged after", epoch, "epochs")
        break

# Plot the epochs against the error values
plt.plot(epoch_list, errors)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('relu_activation Learning Curve')
plt.show()
```


relu_activation Learning Curve
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Error
0
200
400
600
800
1000
Epoch

In [36]:
```
# Initialize the learning rates and corresponding iteration counts
learning_rates = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]
iteration_counts = []

# Perform the experiment for each learning rate
for alpha in learning_rates:
    # Initialize weights for each learning rate
    W0_temp = W0
    W1_temp = W1
    W2_temp = W2

    # Initialize the error and epoch lists
    errors = []

    # Train the perceptron model
    for epoch in range(max_epochs):
        error = 0
        for i in range(len(X)):
            # Calculate the predicted output
            weighted_sum = W0_temp + W1_temp * X[i][0] + W2_temp * X[i][1]
            y_pred = step_activation(weighted_sum)

            # Calculate the error
            delta = alpha * (y[i] - y_pred)

            # Update the weights
            W0_temp += delta
            W1_temp += delta * X[i][0]
            W2_temp += delta * X[i][1]

            # Calculate the sum-square-error
            error += delta**2

        # Check for convergence
        if error <= convergence_error:
            break

    # Append the number of iterations to the iteration_counts list
    iteration_counts.append(epoch + 1)  # Add 1 to account for 0-based indexin
```

In [ ]:
```
# Plot the learning rate vs. number of iterations
plt.plot(learning_rates, iteration_counts, marker='o')
plt.xlabel('Learning Rate (α)')
plt.ylabel('Number of Iterations')
plt.title('Number of Iterations vs. Learning Rate')
plt.grid()
plt.show()
```



In [37]:
```
import numpy as np

# Customer data
data = np.array([
    [20, 6, 2, 386, 1],  # High Value
    [16, 3, 6, 289, 1],  # High Value
    [27, 6, 2, 393, 1],  # High Value
    [19, 1, 2, 110, 0],  # Low Value
    [24, 4, 2, 280, 1],  # High Value
    [22, 1, 5, 167, 0],  # Low Value
    [15, 4, 2, 271, 1],  # High Value
    [18, 4, 2, 274, 1],  # High Value
    [21, 1, 4, 148, 0],  # Low Value
    [16, 2, 4, 198, 0]   # Low Value
])

# Initialize weights and bias
np.random.seed(0)
weights = np.random.rand(5)  # Weights for Candies, Mangoes, Milk Packets, Pay
bias = np.random.rand()

# Learning rate
learning_rate = 0.01

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Function to train the perceptron
def train_perceptron(data, weights, bias, learning_rate, epochs):
    for epoch in range(epochs):
        total_error = 0
        for row in data:
            features = row[:-1]  # Input features (Candies, Mangoes, Milk Pack
            target = row[-1]     # Target (High Value or Low Value)

            # Calculate the predicted value using the sigmoid activation
            net_input = np.dot(features, weights[1:]) + weights[0] * 1  # Incl
            predicted = sigmoid(net_input)

            # Calculate the error
            error = target - predicted
            total_error += error ** 2

            # Update weights and bias
            weights[1:] += learning_rate * error * predicted * (1 - predicted)
            weights[0] += learning_rate * error * predicted * (1 - predicted)

        if total_error == 0:
            break

# Train the perceptron
train_perceptron(data, weights, bias, learning_rate, epochs=10000)

# Test the perceptron on the same data
for row in data:
    features = row[:-1]
```

```
    target = row[-1]

    net_input = np.dot(features, weights[1:]) + weights[0] * 1  # Include bias
    predicted = sigmoid(net_input)

    if predicted >= 0.5:
        prediction = "High Value"
    else:
        prediction = "Low Value"

    print(f"Actual: {target} | Predicted: {prediction} | Probability: {predict
```

```
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 0 | Predicted: High Value | Probability: 1.0000
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 0 | Predicted: High Value | Probability: 1.0000
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 1 | Predicted: High Value | Probability: 1.0000
Actual: 0 | Predicted: High Value | Probability: 1.0000
Actual: 0 | Predicted: High Value | Probability: 1.0000
```

In [ ]:
```
import numpy as np

# Customer data
data = np.array([
    [20, 6, 2, 386, 1],  # High Value
    [16, 3, 6, 289, 1],  # High Value
    [27, 6, 2, 393, 1],  # High Value
    [19, 1, 2, 110, 0],  # Low Value
    [24, 4, 2, 280, 1],  # High Value
    [22, 1, 5, 167, 0],  # Low Value
    [15, 4, 2, 271, 1],  # High Value
    [18, 4, 2, 274, 1],  # High Value
    [21, 1, 4, 148, 0],  # Low Value
    [16, 2, 4, 198, 0]   # Low Value
])

# Extract features and target
X = data[:, :-1]
y = data[:, -1]

# Add bias term (intercept)
X = np.hstack((np.ones((X.shape[0], 1)), X))

# Calculate the pseudo-inverse of X
X_pseudo_inv = np.linalg.pinv(X)

# Calculate the weights using the pseudo-inverse
weights = np.dot(X_pseudo_inv, y)

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Predict using the obtained weights
predicted = sigmoid(np.dot(X, weights))

# Threshold predictions (>= 0.5 as High Value, < 0.5 as Low Value)
threshold = 0.5
predicted_binary = (predicted >= threshold).astype(int)

# Compare predicted vs. actual labels
accuracy = np.mean(predicted_binary == y)
print(f"Accuracy using Matrix Pseudo-Inverse: {accuracy * 100:.2f}%")
```

```
Accuracy using Matrix Pseudo-Inverse: 60.00%
```

In [38]:
```
import numpy as np
import matplotlib.pyplot as plt

# Customer data
data = np.array([
    [20, 6, 2, 386, 1],  # High Value
    [16, 3, 6, 289, 1],  # High Value
    [27, 6, 2, 393, 1],  # High Value
    [19, 1, 2, 110, 0],  # Low Value
    [24, 4, 2, 280, 1],  # High Value
    [22, 1, 5, 167, 0],  # Low Value
    [15, 4, 2, 271, 1],  # High Value
    [18, 4, 2, 274, 1],  # High Value
    [21, 1, 4, 148, 0],  # Low Value
    [16, 2, 4, 198, 0]   # Low Value
])

# Initialize weights and bias with small random values
np.random.seed(0)
weights = np.random.uniform(-1, 1, 5)  # Initialize weights between -1 and 1
bias = 0.0

# Learning rate
learning_rate = 0.01

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Function to train the perceptron
def train_perceptron(data, weights, bias, learning_rate, epochs):
    error_history = []

    for epoch in range(epochs):
        total_error = 0
        for row in data:
            features = row[:-1]  # Input features (Candies, Mangoes, Milk Pack
            target = row[-1]     # Target (High Value or Low Value)

            # Calculate the predicted value using the sigmoid activation
            net_input = np.dot(features, weights[1:]) + weights[0]  # Include
            predicted = sigmoid(net_input)

            # Calculate the error
            error = target - predicted
            total_error += error ** 2

            # Update weights and bias
            weights[1:] += learning_rate * error * predicted * (1 - predicted)
            weights[0] += learning_rate * error * predicted * (1 - predicted)

        error_history.append(total_error)

        # Check for convergence
        if total_error <= 0.002:
            print(f"Converged after {epoch + 1} epochs.")
            break
```

```
    return error_history, weights

# Train the perceptron
error_history, final_weights = train_perceptron(data, weights, bias, learning_

# Test the perceptron on the same data
def predict_with_perceptron(features, weights):
    net_input = np.dot(features, weights[1:]) + weights[0]  # Include bias
    predicted = sigmoid(net_input)
    return predicted

# Apply perceptron predictions to the data
predictions = [1 if predict_with_perceptron(row[:-1], final_weights) >= 0.5 el

# Compare predicted vs. actual labels
accuracy = np.mean(predictions == data[:, -1])
print(f"Accuracy using Perceptron: {accuracy * 100:.2f}%")
```

```
Accuracy using Perceptron: 40.00%
```

In [ ]:
```
import numpy as np

# Define the AND gate truth table
truth_table = np.array([[0, 0, 0],
                        [0, 1, 0],
                        [1, 0, 0],
                        [1, 1, 1]])

# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Initialize the neural network parameters
input_size = 2
hidden_size = 2
output_size = 1
learning_rate = 0.05
epochs = 1000

# Initialize the weights and biases
np.random.seed(0)
weights_input_hidden = np.random.uniform(-1, 1, (input_size, hidden_size))
bias_hidden = np.zeros((1, hidden_size))
weights_hidden_output = np.random.uniform(-1, 1, (hidden_size, output_size))
bias_output = np.zeros((1, output_size))

# Training Loop
for epoch in range(epochs):
    total_error = 0

    for sample in truth_table:
        # Forward pass
        input_layer = np.array([sample[:2]])
        target_output = np.array([sample[2]])

        # Calculate hidden layer output
        hidden_layer_input = np.dot(input_layer, weights_input_hidden) + bias_
        hidden_layer_output = sigmoid(hidden_layer_input)

        # Calculate output layer output
        output_layer_input = np.dot(hidden_layer_output, weights_hidden_output
        output_layer_output = sigmoid(output_layer_input)

        # Calculate the error
        error = target_output - output_layer_output
        total_error += np.mean(np.abs(error))

        # Backpropagation
        delta_output = error * sigmoid_derivative(output_layer_output)
        delta_hidden = delta_output.dot(weights_hidden_output.T) * sigmoid_der

        # Update weights and biases
        weights_hidden_output += hidden_layer_output.T.dot(delta_output) * lea
        bias_output += np.sum(delta_output, axis=0, keepdims=True) * learning_
```

```
        weights_input_hidden += input_layer.T.dot(delta_hidden) * learning_rat
        bias_hidden += np.sum(delta_hidden, axis=0, keepdims=True) * learning_

    # Check for convergence
    if total_error <= 0.002:
        print(f"Converged after {epoch + 1} epochs.")
        break

# Test the trained neural network
for sample in truth_table:
    input_layer = np.array([sample[:2]])
    hidden_layer_input = np.dot(input_layer, weights_input_hidden) + bias_hidc
    hidden_layer_output = sigmoid(hidden_layer_input)
    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) +
    predicted_output = sigmoid(output_layer_input)

    print(f"Input: {sample[:2]}, Target Output: {sample[2]}, Predicted Output:
```

```
Input: [0 0], Target Output: 0, Predicted Output: [0.23842572]
Input: [0 1], Target Output: 0, Predicted Output: [0.26273222]
Input: [1 0], Target Output: 0, Predicted Output: [0.27488417]
Input: [1 1], Target Output: 1, Predicted Output: [0.29894486]
```

In [39]:
```
import numpy as np

# Define the inputs and labels for the AND gate logic
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
labels = np.array([[0], [1], [1], [0]])

# Initialize the weights randomly with small values
np.random.seed(1)
weights1 = np.random.rand(2, 4)
weights2 = np.random.rand(4, 1)

# Define the Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

# Implement the back-propagation algorithm
for i in range(1000):
    # Forward pass
    layer1 = sigmoid(np.dot(inputs, weights1))
    layer2 = sigmoid(np.dot(layer1, weights2))

    # Backward pass
    layer2_error = labels - layer2
    layer2_delta = layer2_error * sigmoid_derivative(layer2)
    layer1_error = layer2_delta.dot(weights2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1)

    # Update weights
    weights2 += layer1.T.dot(layer2_delta) * 0.05
    weights1 += inputs.T.dot(layer1_delta) * 0.05

    # Check convergence error condition
    if np.mean(np.abs(layer2_error)) <= 0.002:
        break

# Test the Neural Network on the AND gate logic and calculate the accuracy
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
test_labels = np.array([[0], [0], [0], [1]])
test_layer1 = sigmoid(np.dot(test_inputs, weights1))
test_layer2 = sigmoid(np.dot(test_layer1, weights2))
test_predictions = np.where(test_layer2 >= 0.5, 1, 0)
accuracy = np.mean(test_predictions == test_labels)
print('Accuracy:', accuracy)
```

```
Accuracy: 0.75
```

In [ ]:
```

```

```
[[0.10175656 0.10175656]
 [0.39097334 0.39097334]
 [0.39097334 0.39097334]
 [0.48620607 0.48620607]]
```

In [ ]:
```
#A9)
v10,v20 = 0.01,0.4
v11,v12,v21,v22 = 10,0.2,-0.75,0.09 # input layer weights
w10,w20 = 0.11,0.41
w11,w12,w21,w22 = -20,0.1,-1.2,0.7 # output layer weights
learning_rate = 0.05
# input vectors
bias = [1,1,1,1]
x1 = [0,0,1,1]
x2 = [0,1,0,1]
# output vectors [x1 AND x2]
output_actual = [1,0,0,1]
output_actual1 = [0,1,1,0]
output_actual2 = [1,0,0,1]
output_predicted = 0
# hidden layer units
h1,h2 = 0,0
output_predicted1, output_predicted2 = 0, 0
iterations=0
while (iterations < 2500):
    print("Epoch",iterations+1)
    for i in range(0,len(bias)):
        h1 = bias[i] * v10 + x1[i] * v11 + x2[i] * v21
        h2 = bias[i] * v20 + x1[i] * v12 + x2[i] * v22
        output_predicted1 = 1/(1+ np.exp(-h1))
        output_predicted2 = 1/(1+ np.exp(-h2))
        output_predicted01 = 1/(1+ np.exp(-(w10 + output_predicted1 * w11 + ou
        output_predicted02 = 1/(1+ np.exp(-(w20 + output_predicted1 * w12 + ou
        if (output_predicted01 == output_actual[i]):
            print("The Output 1: ")
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\
            continue
        else:
            derivative = output_predicted01*(1-output_predicted01)
            deltak = derivative*(-output_predicted01 + output_actual1[i])
            deltah1 = output_predicted1*(1-output_predicted1)*(w11)*deltak
            deltah2 = output_predicted2*(1-output_predicted2)*(w21)*deltak
            w10 = w10 + learning_rate * deltak * 1
            w11 = w11 + learning_rate * deltak * output_predicted1
            w12 = w21 + learning_rate * deltak * output_predicted2
            v10,v20 = (v10 + learning_rate*deltah1*bias[i]),(v20 + learning_ra
            v11,v12,v21,v22 = (v11 + learning_rate*deltah1*x1[i]),(v12 + learn
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\
        if (output_predicted02 == output_actual[i]):
            print("The Output 2: ")
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\
            continue
        else:
            derivative = output_predicted02*(1-output_predicted02)
            deltak = derivative*(-output_predicted02 + output_actual2[i])
            deltah1 = output_predicted1*(1-output_predicted1)*(w12)*deltak
            deltah2 = output_predicted2*(1-output_predicted2)*(w22)*deltak
            w20 = w20 + learning_rate * deltak * 1
            w21 = w21 + learning_rate * deltak * output_predicted1
            w22 = w22 + learning_rate * deltak * output_predicted2
            v10,v20 = (v10 + learning_rate*deltah1*bias[i]),(v20 + learning_ra
            v11,v12,v21,v22 = (v11 + learning_rate*deltah1*x1[i]),(v12 + learn
```

```
            print("\n""bias = ",bias[i],"\n""x1 = ",x1[i],"\n""x2 = ",x2[i],"\

    iterations=iterations+1
    if abs(output_predicted01 - output_actual1[i]) < 0.002 and abs(output_prec
        print("The error is ",abs( output_predicted02 - output_actual2[i]),abs
        break
    else:
        continue
```

```
Streaming output truncated to the last 5000 lines.
h1 unit =  0.9996495800605508
h2 unit = 0.6428848815820831
Epoch 2399

bias =  1
x1 =  0
x2 =  0
h1 unit =  0.21517006498864732
h2 unit= 0.5850810122276242

bias =  1
x1 =  0
x2 =  0
h1 unit =  0.21517006498864732
h2 unit = 0.5850810122276242

bias =  1
x1 =  0
```

In [ ]:

```
#A10)
# AND Gate
x = [[0, 0], [0, 1], [1, 0], [1, 1]]
y = [0, 0, 0, 1]
clf0 = MLPClassifier(solver='lbfgs', activation='logistic', hidden_layer_sizes
clf0.fit(x, y)
print(clf0.score(x, y))
clf0.predict([[0,0],[1,1]])

# XOR Gate
x1 = [[0, 0], [0, 1], [1, 0], [1, 1]]
y1 = [0, 1, 1, 0]
clf1 = MLPClassifier(solver='lbfgs', activation='logistic', hidden_layer_sizes
clf1.fit(x1, y1)
print(clf1.score(x, y))
clf1.predict([[0,0],[1,1]])
```

```
1.0
0.75
```

Out[24]:

```
array([0, 1])
```

In [ ]:
```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.impute import SimpleImputer

# Read the CSV file
df = pd.read_csv('train_agriculture.csv')

# Split the dataset into features and target
X = df.iloc[:, 1:-1].values
y = df.iloc[:, -1].values

# Handle missing values
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X = imputer.fit_transform(X)

# Create an instance of the MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, alpha=0.0001,

# Fit the model to the data
clf.fit(X, y)
```

```
Iteration 244, loss = 0.43033988
Iteration 245, loss = 0.42947988
Iteration 246, loss = 0.42991084
Iteration 247, loss = 0.43017363
Iteration 248, loss = 0.43148457
Iteration 249, loss = 0.43148884
Iteration 250, loss = 0.43089437
Iteration 251, loss = 0.42952832
Iteration 252, loss = 0.42964530
Iteration 253, loss = 0.43009131
Iteration 254, loss = 0.43092869
Iteration 255, loss = 0.42961516
Iteration 256, loss = 0.43022513
Training loss did not improve more than tol=0.000000 for 10 consecutive ep
ochs. Stopping.
```

Out[54]:
```
MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=500, random_state=21,
              tol=1e-09, verbose=10)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or

In [ ]:
```
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.impute import SimpleImputer

# Read the CSV file
df = pd.read_csv('train_agriculture.csv')

# Split the dataset into features and target
X = df.iloc[:, 1:-1].values
y = df.iloc[:, -1].values

# Handle missing values
imputer = SimpleImputer(missing_values=np.nan, strategy='mean')
X = imputer.fit_transform(X)

# Create an instance of the MLPClassifier
clf = MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1000, alpha=0.0001,

# Fit the model to the data
clf.fit(X, y)

# Predict the target variable for the test data
y_pred = clf.predict(X)

# Print the accuracy score of the model
accuracy = clf.score(X, y)
print("Accuracy:", accuracy)
```

```
Iteration 186, loss = 0.43219875
Iteration 187, loss = 0.43261586
Iteration 188, loss = 0.43164889
Iteration 189, loss = 0.43227856
Iteration 190, loss = 0.43209745
Iteration 191, loss = 0.43173495
Iteration 192, loss = 0.43142838
Iteration 193, loss = 0.43215103
Iteration 194, loss = 0.43213891
Iteration 195, loss = 0.43236566
Iteration 196, loss = 0.43162367
Iteration 197, loss = 0.43157090
Iteration 198, loss = 0.43149272
Iteration 199, loss = 0.43177309
Iteration 200, loss = 0.43171640
Iteration 201, loss = 0.43176681
Iteration 202, loss = 0.43133497
Iteration 203, loss = 0.43167277
Training loss did not improve more than tol=0.000100 for 10 consecutive ep
ochs. Stopping.
```