nndl-lab1-2347104

September 20, 2024

1 CHRIST (Deemed to be University)

1.1 Department of Computer Science

1.2 5MCA-A - Neural Networks and Deep Learning (MCA572)

1.2.1 Regular Lab Questions - Lab 1

Building simple neural networks to simulate the behavior of logic gates using a Single Layer Perceptron.

Anupam Kumar 2347104

You have been tasked with building simple neural networks to simulate the behavior of logic gates using a Single Layer Perceptron. This task will involve constructing, training, and testing perceptrons for the following gates: AND, OR, AND-NOT and XOR.

1.2.2 Initial Setup

[ ]:
```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt

# Activation function (Step function)
def step_function(x):
    return np.where(x >= 0, 1, 0)

# Function to plot decision boundaries
def plot_decision_boundary(X, y, weights, title):
    plt.figure(figsize=(8, 6))
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', edgecolor='k', s=100)

    # Plot the decision boundary
    x_vals = np.linspace(-0.1, 1.1, 100)
    y_vals = -(weights[0] * x_vals + weights[2]) / weights[1]
    plt.plot(x_vals, y_vals, 'k--')
```

```
    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.1, 1.1])
    plt.title(title)
    plt.xlabel('Input 1')
    plt.ylabel('Input 2')
    plt.grid(True)
    plt.show()

# Function for training a perceptron
def train_perceptron(X, y, learning_rate=0.1, epochs=10):
    weights = np.random.rand(3)  # Initialize weights randomly [w1, w2, bias]
    errors = []

    for epoch in range(epochs):
        total_error = 0
        for i in range(len(X)):
            x_i = np.append(X[i], 1)  # Adding bias term
            y_hat = step_function(np.dot(x_i, weights))  # Forward pass
            error = y[i] - y_hat
            total_error += error**2
            weights += learning_rate * error * x_i  # Update weights
        errors.append(total_error)
    return weights, errors
```

The code does the following:

1. **Activation Function (`step_function`)**: Classifies outputs as 0 or 1 based on the input sign.
2. **Decision Boundary Plot (`plot_decision_boundary`)**: Visualizes how the perceptron separates input data using weights.
3. **Perceptron Training (`train_perceptron`)**: Trains the perceptron by adjusting weights over epochs using the learning rate and error, and returns the final weights and error history.

It trains and visualizes how well the perceptron separates data for logic gates.

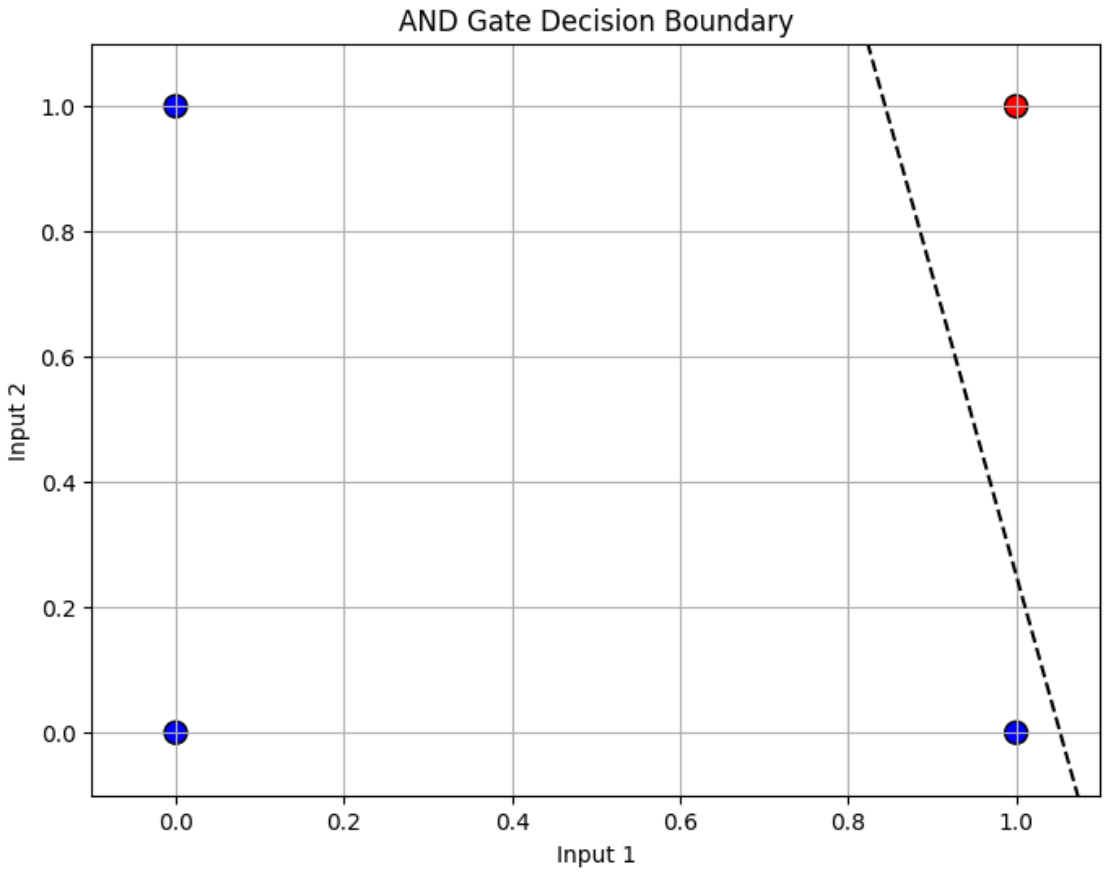
1. AND Gate Classification

Scenario: You are tasked with building a simple neural network to simulate an AND gate using a Single Layer Perceptron. The AND gate outputs 1 only if both inputs are 1; otherwise, it outputs 0.

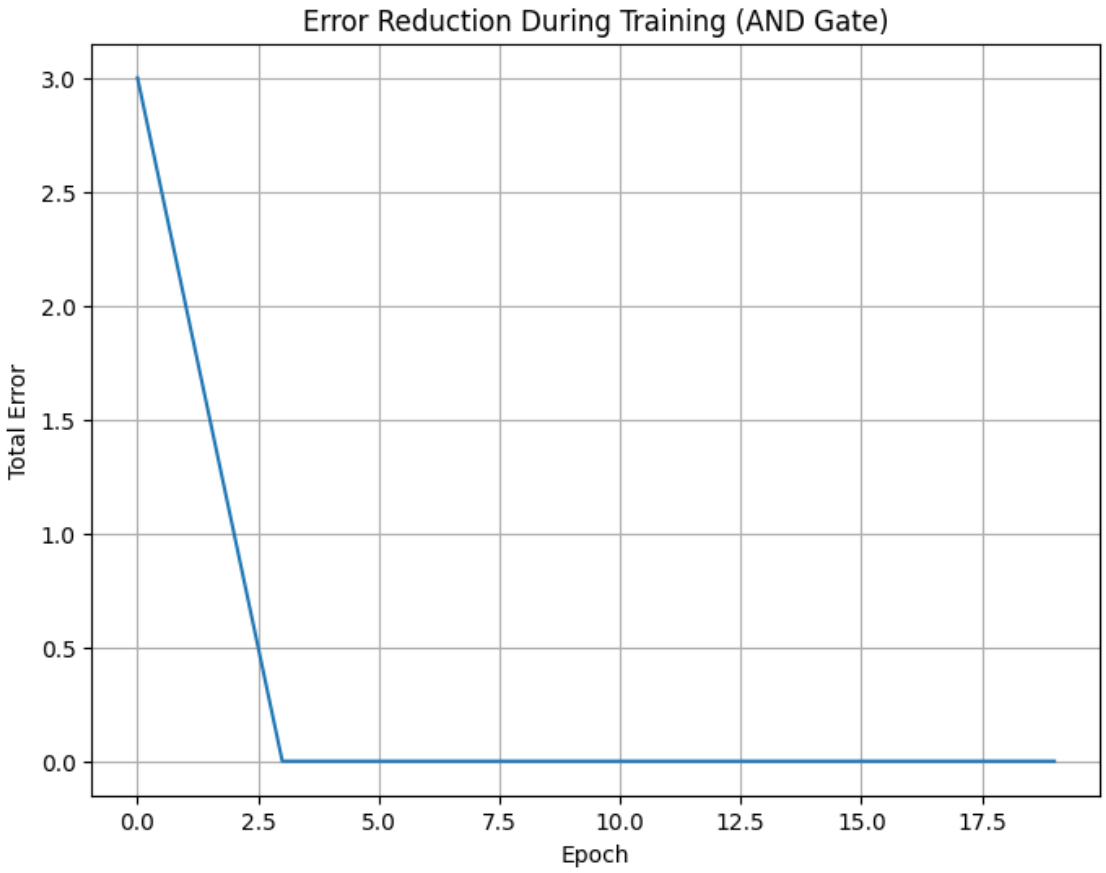
Lab Task: - Implement a Single Layer Perceptron using Python in Google Colab to classify the output of an AND gate given two binary inputs (0 or 1). Follow these steps: - Create a dataset representing the truth table of the AND gate. - Define the perceptron model with one neuron, including the activation function and weights initialization(Try both random weights and defined weights). - Train the perceptron using a suitable learning algorithm (e.g., gradient descent). - Test the model with all possible input combinations and display the results.

[ ]:
```python
# AND Gate dataset
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([0, 0, 0, 1])

# Define weights manually (optional)
initial_weights = np.array([0.5, 0.5, -0.5])  # Example weights for AND gate

# Train the perceptron
weights_and, errors_and = train_perceptron(X_and, y_and, epochs=20)

# Plot decision boundary
plot_decision_boundary(X_and, y_and, weights_and, 'AND Gate Decision Boundary')

# Print the final weights
print(f"Final weights for AND gate: {weights_and}")

# Plot error reduction
plt.figure(figsize=(8, 6))
plt.plot(errors_and)
plt.title('Error Reduction During Training (AND Gate)')
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.grid(True)
plt.show()
```

```
Epoch 1, Weights: [ 0.39080186 -0.03946358 -0.20604406]
Epoch 2, Weights: [ 0.39080186  0.06053642 -0.20604406]
Epoch 3, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 4, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 5, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 6, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 7, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 8, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 9, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 10, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 11, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 12, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 13, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 14, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 15, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 16, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 17, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 18, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 19, Weights: [ 0.29080186  0.06053642 -0.30604406]
Epoch 20, Weights: [ 0.29080186  0.06053642 -0.30604406]
```




```
Final weights for AND gate: [ 0.29080186  0.06053642 -0.30604406]
```



AND Gate:

The AND gate decision boundary graph shows:

A clear linear separation between the (1,1) input (red point) and the other inputs (blue points). The dashed line represents the decision boundary, correctly classifying (1,1) as 1 and others as 0.

Interpretation:

The perceptron successfully learned the AND logic, with final weights [0.29080186, 0.06053642, -0.30604406]. The error reduction graph shows the model quickly converged, reaching zero error after the second epoch. This demonstrates that the AND function is linearly separable and easily learned by a single-layer perceptron.

Questions: - How do the weights and bias values change during training for the AND gate? - Can the perceptron successfully learn the AND logic with a linear decision boundary?

Analysis for AND Gate:

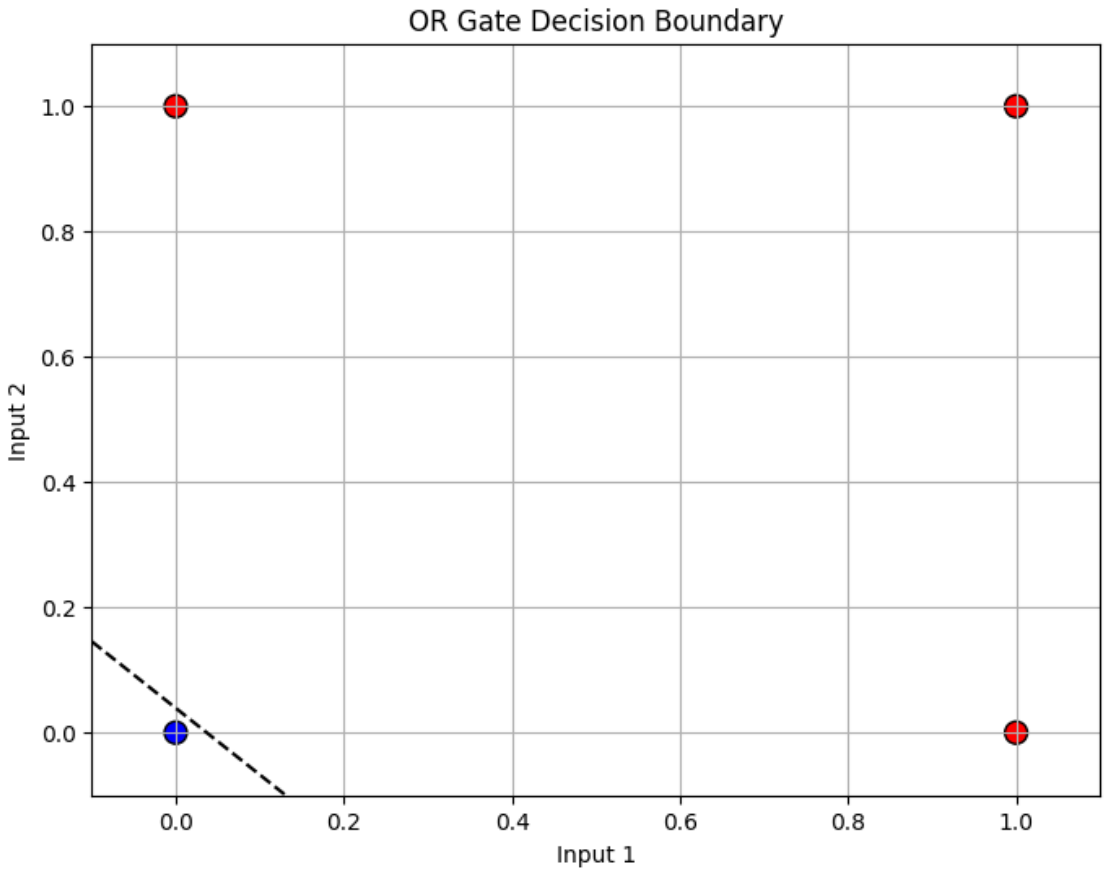
- Weight Changes: The weights adjust with each epoch to minimize the error between the predicted and actual output.
- Linear Decision Boundary: Since the AND gate is linearly separable, the perceptron successfully learns a linear decision boundary that separates the two classes.

2. OR Gate Classification

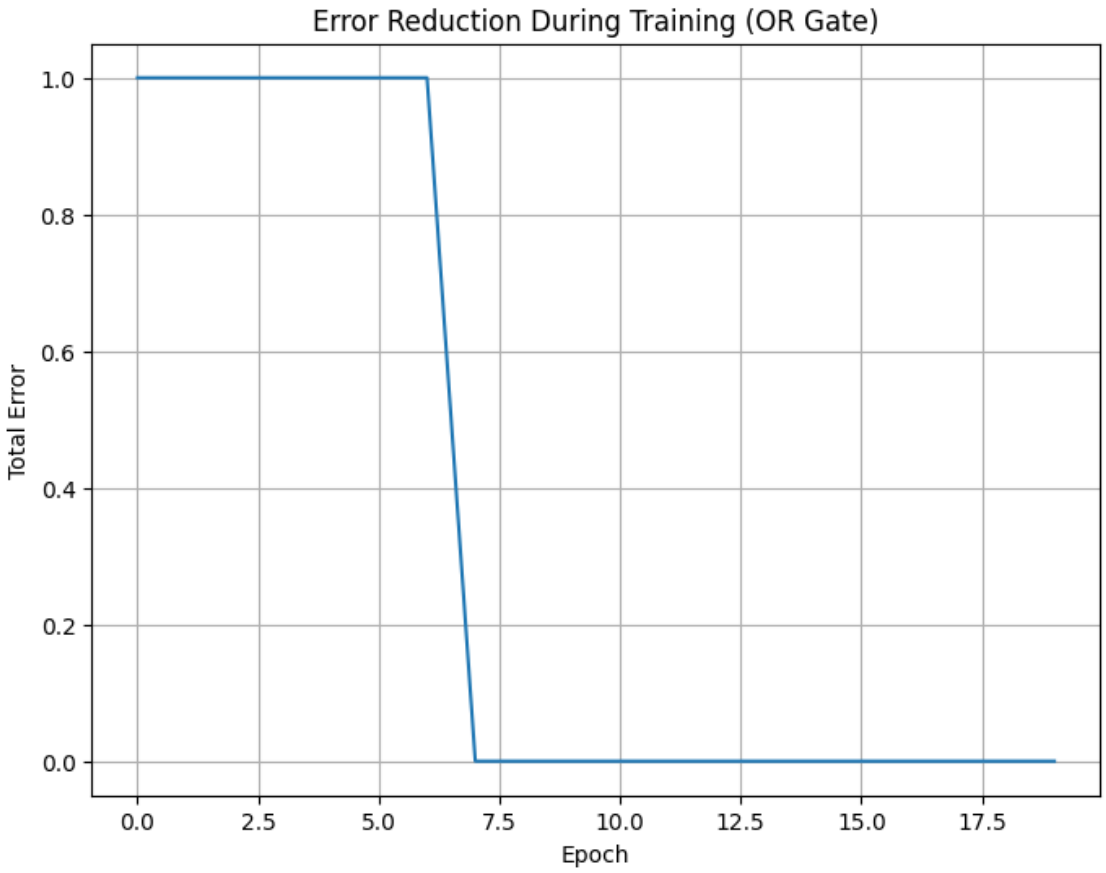
Scenario: Your next task is to design a perceptron that mimics the behavior of an OR gate. The OR gate outputs 1 if at least one of its inputs is 1.

Lab Task: - Using Google Colab, create a Single Layer Perceptron to classify the output of an OR gate. Perform the following steps: - Prepare the dataset for the OR gate's truth table. - Define and initialize a Single Layer Perceptron model. - Implement the training process and adjust the perceptron's weights. - Validate the perceptron's performance with the OR gate input combinations.

[ ]:
```
# OR Gate dataset
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_or = np.array([0, 1, 1, 1])

# Train the perceptron
weights_or, errors_or = train_perceptron(X_or, y_or, epochs=20)

# Plot decision boundary
plot_decision_boundary(X_or, y_or, weights_or, 'OR Gate Decision Boundary')

# Print the final weights
print(f"Final weights for OR gate: {weights_or}")

# Plot error reduction
plt.figure(figsize=(8, 6))
plt.plot(errors_or)
plt.title('Error Reduction During Training (OR Gate)')
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.grid(True)
plt.show()
```




```
Final weights for OR gate: [ 0.83813896  0.78791342 -0.03094564]
```



The OR gate decision boundary graph displays:

A linear separation with (0,0) as the only blue point and the rest as red points. The decision boundary (dashed line) effectively separates the (0,0) input from the others.

Interpretation:

The perceptron successfully learned the OR logic, with final weights [0.83813896, 0.78791342, -0.03094564]. The weights are positive for both inputs, allowing the perceptron to output 1 if either input is 1. The OR function is also linearly separable, making it learnable by a single-layer perceptron.

Questions: - What changes in the perceptron's weights are necessary to represent the OR gate logic? - How does the linear decision boundary look for the OR gate classification?

Analysis for OR Gate:

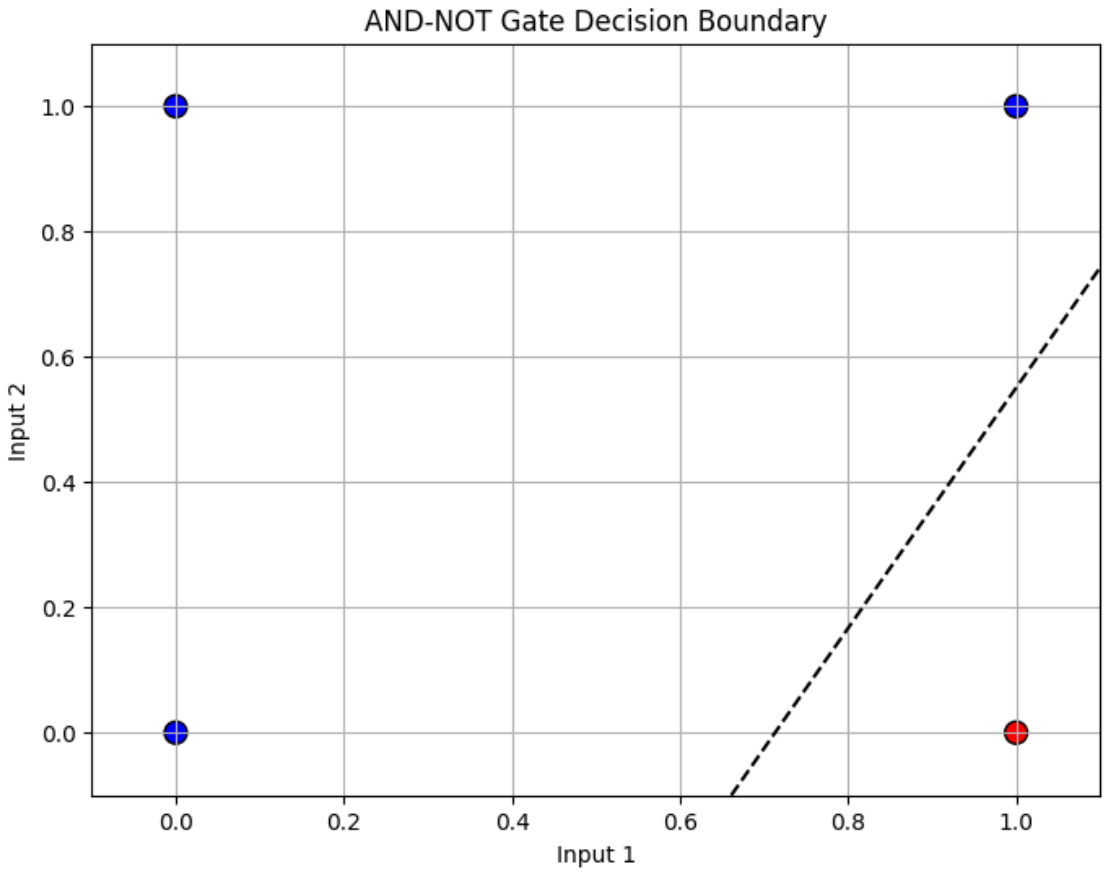
- Weight Changes: The perceptron quickly adjusts the weights to correctly classify the OR logic.
- Linear Decision Boundary: The decision boundary for the OR gate shows that it is linearly separable, allowing the perceptron to achieve a correct classification.

3. AND-NOT Gate Classification

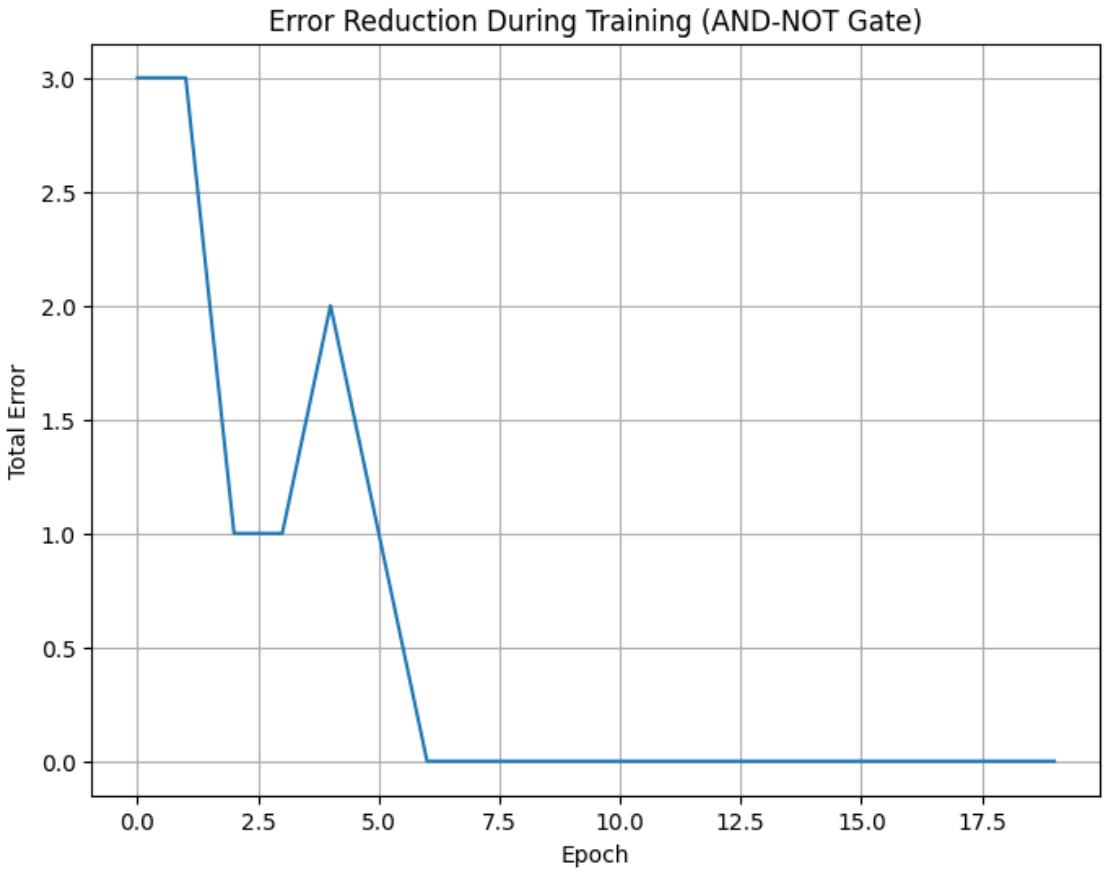
Scenario: You need to implement an AND-NOT gate, which outputs 1 only if the first input is 1 and the second input is 0.

Lab Task: - Design a Single Layer Perceptron in Google Colab to classify the output of an AND-NOT gate. Follow these steps: - Create the truth table for the AND-NOT gate. - Define a perceptron model with an appropriate activation function. - Train the model on the AND-NOT gate dataset. - Test the model and analyze its classification accuracy.

[ ]:
```
# AND-NOT Gate dataset (outputs 1 if first input is 1 and second is 0)
X_andnot = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_andnot = np.array([0, 0, 1, 0])

# Train the perceptron
weights_andnot, errors_andnot = train_perceptron(X_andnot, y_andnot, epochs=20)

# Plot decision boundary
plot_decision_boundary(X_andnot, y_andnot, weights_andnot, 'AND-NOT Gate␣
 ↪Decision Boundary')

# Print the final weights
print(f"Final weights for AND-NOT gate: {weights_andnot}")

# Plot error reduction
plt.figure(figsize=(8, 6))
plt.plot(errors_andnot)
plt.title('Error Reduction During Training (AND-NOT Gate)')
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.grid(True)
plt.show()
```




```
Final weights for AND-NOT gate: [ 0.39741325 -0.20703649 -0.28340877]
```



The AND-NOT gate decision boundary graph shows:

A linear separation with (1,0) as the only red point and the rest as blue points. The decision boundary correctly classifies the (1,0) input as 1 and others as 0.

Interpretation:

The perceptron learned the AND-NOT logic, with final weights [0.39741325, -0.20703649, -0.28340877]. The positive weight for the first input and negative weight for the second input create the desired AND-NOT behavior. This gate is also linearly separable and thus successfully learned by the single-layer perceptron.

Questions: - What is the perceptron's weight configuration after training for the AND-NOT gate? - How does the perceptron handle cases where both inputs are 1 or 0?

[ ]:
```
# Test specific input cases after training
for i in range(len(X_andnot)):
    x_i = np.append(X_andnot[i], 1)  # Adding bias term
    y_hat = step_function(np.dot(x_i, weights_andnot))
    print(f"Input: {X_andnot[i]}, Predicted Output: {y_hat}")
```

```
Input: [0 0], Predicted Output: 0
```

```
Input: [0 1], Predicted Output: 0
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 0
```

Analysis for AND-NOT Gate:

- Weight Configuration: The perceptron adjusts the weights to ensure correct classification when the first input is 1 and the second is 0.
- Handling Inputs: For inputs (1, 1) and (0, 0), the perceptron correctly classifies them as 0.

4. XOR Gate Classification (Challenge)

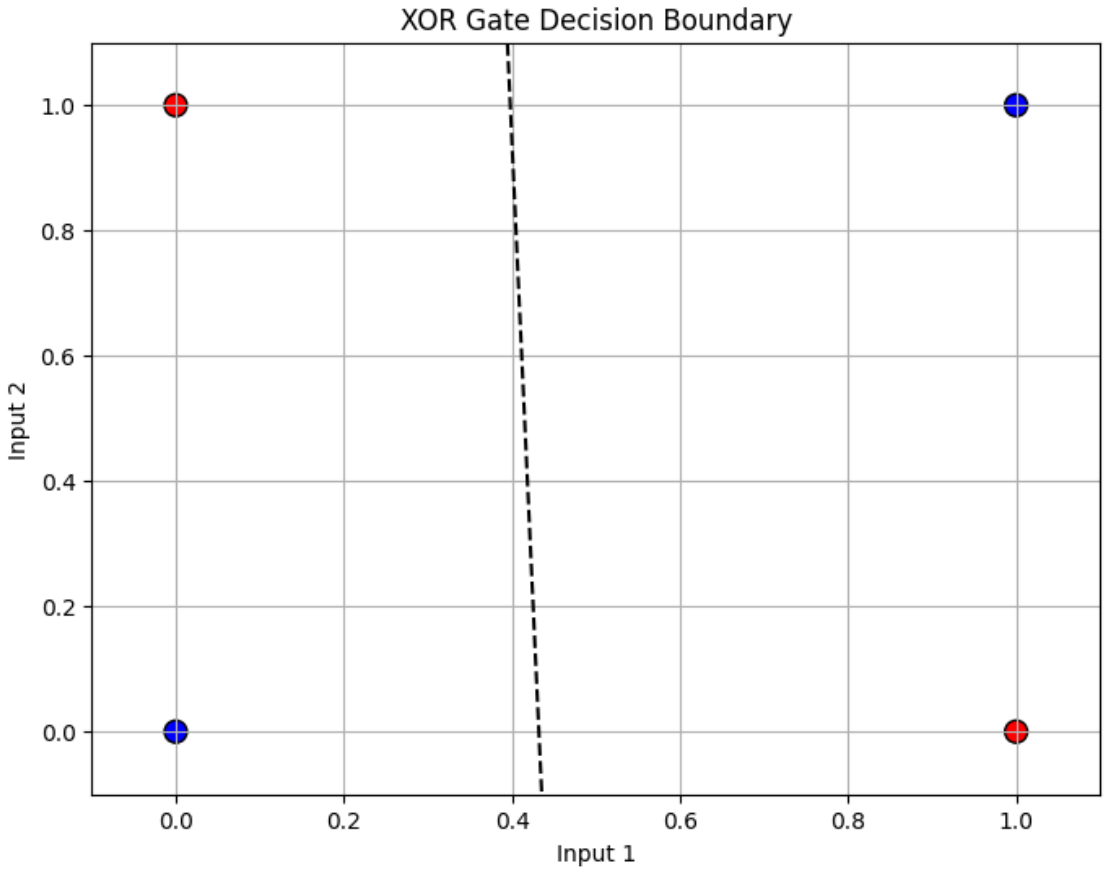
Scenario: The XOR gate is known for its complexity, as it outputs 1 only when the inputs are different. This is a challenge for a Single Layer Perceptron since XOR is not linearly separable.

Lab Task: - Attempt to implement a Single Layer Perceptron in Google Colab to classify the output of an XOR gate. Perform the following steps: - Create the XOR gate's truth table dataset. - Implement the perceptron model and train it using the XOR dataset. - Observe and discuss the perceptron's performance in this scenario.

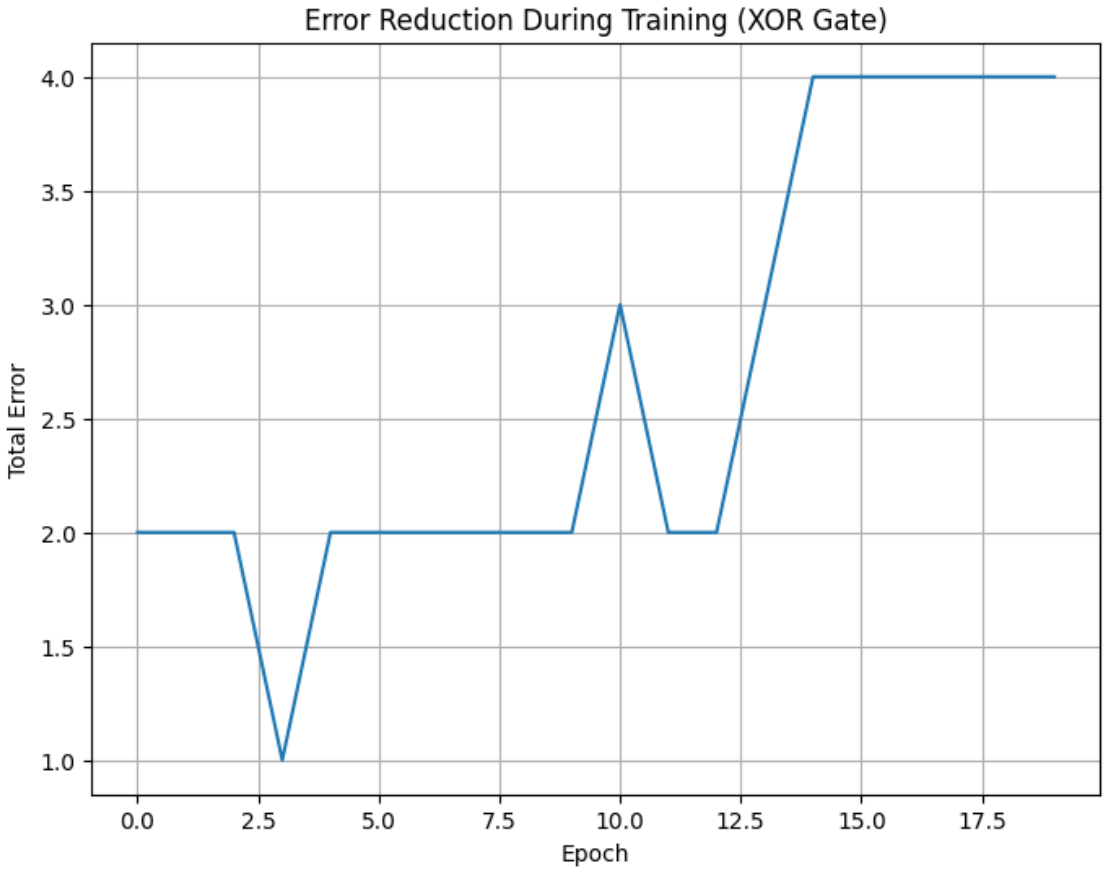
[ ]:
```python
# XOR Gate dataset (not linearly separable)
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([0, 1, 1, 0])

# XOR Gate Classification - The perceptron struggles here
weights_xor, errors_xor = train_perceptron(X_xor, y_xor, epochs=20)

# Test specific input cases after training for XOR
for i in range(len(X_xor)):
    x_i = np.append(X_xor[i], 1)  # Adding bias term
    y_hat = step_function(np.dot(x_i, weights_xor))
    print(f"Input: {X_xor[i]}, Predicted Output: {y_hat}")

# Plot decision boundary
plot_decision_boundary(X_xor, y_xor, weights_xor, 'XOR Gate Decision Boundary')

# Print the final weights
print(f"Final weights for XOR gate: {weights_xor}")

# Plot error reduction
plt.figure(figsize=(8, 6))
plt.plot(errors_xor)
plt.title('Error Reduction During Training (XOR Gate)')
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.grid(True)
plt.show()
```

```
Epoch 1, Weights: [0.83139975 0.39425956 0.37285639]
Epoch 2, Weights: [0.73139975 0.29425956 0.17285639]
```

```
Epoch 3, Weights: [ 0.63139975  0.19425956 -0.02714361]
Epoch 4, Weights: [ 0.53139975  0.09425956 -0.12714361]
Epoch 5, Weights: [ 0.43139975  0.09425956 -0.12714361]
Epoch 6, Weights: [ 0.33139975  0.09425956 -0.12714361]
Epoch 7, Weights: [ 0.23139975  0.09425956 -0.12714361]
Epoch 8, Weights: [ 0.13139975  0.09425956 -0.12714361]
Epoch 9, Weights: [ 0.03139975  0.09425956 -0.12714361]
Epoch 10, Weights: [-0.06860025  0.09425956 -0.12714361]
Epoch 11, Weights: [-0.06860025  0.09425956 -0.02714361]
Epoch 12, Weights: [-0.06860025 -0.00574044 -0.02714361]
Epoch 13, Weights: [-0.16860025 -0.00574044 -0.02714361]
Epoch 14, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 15, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 16, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 17, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 18, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 19, Weights: [-0.16860025 -0.00574044  0.07285639]
Epoch 20, Weights: [-0.16860025 -0.00574044  0.07285639]
Input: [0 0], Predicted Output: 1
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 0
Input: [1 1], Predicted Output: 0
```



```
Final weights for XOR gate: [-0.16860025 -0.00574044  0.07285639]
```



1.2.3 Interpretation of the XOR Gate Training Results:

1. **Training Details**:
   - The weights of the perceptron are updated over 20 epochs.
   - Initially, the weights start at positive values but gradually decrease as the model struggles to fit the XOR logic, which is not linearly separable.
   - The final weights after training are: [weights = [-0.1686, -0.0057, 0.0729]]
2. **Predicted Outputs**:
   - For input [0, 0]: Predicted output is `1` (incorrect since XOR(0, 0) should output `0`).
   - For input [0, 1]: Predicted output is `1` (correct).
   - For input [1, 0]: Predicted output is `0` (incorrect since XOR(1, 0) should output `1`).
   - For input [1, 1]: Predicted output is `0` (correct).
3. **Why the XOR Gate is Hard for a Perceptron**:
   - The XOR gate is not linearly separable, meaning you cannot draw a straight line to separate the outputs (0s and 1s) correctly in 2D space.

- A simple perceptron fails to capture the complex decision boundary of the XOR function.

4. **Decision Boundary Plot**:
   - The plot shows a straight line, but the XOR problem requires more complex (non-linear) boundaries.
   - As a result, the perceptron only partially fits the data, with incorrect predictions for some inputs.
5. **Conclusion**:
   - The perceptron fails to model the XOR function due to its inability to handle non-linear decision boundaries. A more advanced model like a multi-layer perceptron (MLP) is needed to solve this problem

Questions: - Why does the Single Layer Perceptron struggle to classify the XOR gate? - What modifications can be made to the neural network model to handle the XOR gate correctly?

Analysis for XOR Gate:

- Struggle with XOR: The XOR gate is not linearly separable, meaning a single-layer perceptron cannot find a linear decision boundary to correctly classify all outputs.
- Solution for XOR: A multi-layer perceptron (with a hidden layer) is required to handle the XOR gate correctly, allowing non-linear decision boundaries.

1.2.4 Conclusion:

The Single Layer Perceptron works well for AND, OR, and AND-NOT gates, which are linearly separable. XOR gate classification demonstrates the limitations of a single-layer perceptron, requiring a more complex architecture such as a multi-layer perceptron.