Implementation of MCP Neuron for AND and OR Function

```
# Connect Google Drive to Google Colab
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly
remount, call drive.mount("/content/drive", force_remount=True).
```

```
def MCP_Neurons_AND(X1, X2, T):
    """
    This function implements basic AND operations with MCP Neuron for
two inputs.

    Arguments:
    Inputs:
    X1 (1D array): An array of binary values.
    X2 (1D array): An array of binary values.
    T (int): Threshold value.

    Output:
    state_neuron (1D list): The state of the neuron (1 or 0) for the
particular inputs.
    """
    assert len(X1) == len(X2), "Input arrays must have the same
length."

    # Initialize the output list
    state_neuron = []

    # Iterate through the inputs and compute the output
    for x1, x2 in zip(X1, X2):
        # Aggregate the inputs
        aggregated_sum = x1 + x2

        # Apply the threshold condition
        if aggregated_sum >= T:
            state_neuron.append(1)
        else:
            state_neuron.append(0)

    return state_neuron

# Example usage for MCP_Neurons_AND function
X1 = [0, 0, 1, 1]
X2 = [0, 1, 0, 1]
T = 2  # Threshold value

# Call the MCP_Neurons_AND function
result = MCP_Neurons_AND(X1, X2, T)
```

```
# Print the result
print(f"Output of AND gate for inputs {X1} and {X2} with threshold
{T}: {result}")
```

```
Output of AND gate for inputs [0, 0, 1, 1] and [0, 1, 0, 1] with
threshold 2: [0, 0, 0, 1]
```

```
def MCP_Neurons_OR(X1, X2, T):
    """
    This function implements basic OR operations with MCP Neuron for
two inputs.

    Arguments:
    Inputs:
    X1 (1D array): An array of binary values.
    X2 (1D array): An array of binary values.
    T (int): Threshold value.

    Output:
    state_neuron (1D list): The state of the neuron (1 or 0) for the
particular inputs.
    """
    assert len(X1) == len(X2), "Input arrays must have the same
length."

    # Initialize the output list
    state_neuron = []

    # Iterate through the inputs and compute the output
    for x1, x2 in zip(X1, X2):
        # Aggregate the inputs
        aggregated_sum = x1 + x2

        # Apply the threshold condition
        if aggregated_sum >= T:
            state_neuron.append(1)
        else:
            state_neuron.append(0)

    return state_neuron

# Example usage for MCP_Neurons_OR function
X1 = [0, 0, 1, 1]
X2 = [0, 1, 0, 1]
T = 1  # Threshold value for OR gate

# Call the MCP_Neurons_OR function
result_or = MCP_Neurons_OR(X1, X2, T)

# Print the result
```

```
print(f"Output of OR gate for inputs {X1} and {X2} with threshold {T}:
{result_or}")
```

```
Output of OR gate for inputs [0, 0, 1, 1] and [0, 1, 0, 1] with
threshold 1: [0, 1, 1, 1]
```

Question - 1: List out all the limitations of MCP - Neurons.

1. Linear Separability – MCP neurons can only solve linearly separable problems (e.g., AND, OR) but fail for non-linearly separable problems like XOR.
2. No Learning Mechanism – The model has fixed weights and thresholds, meaning it does not adapt or learn from data.
3. Binary Activation – The output is limited to 0 or 1, which makes it unsuitable for problems requiring continuous outputs.
4. Lack of Weight Optimization – Weights are predefined rather than learned through training, making it rigid.

Question - 2: Think if you can develop a logic to solve for XOR function using MCP Neuron.

XOR Function Using MCP Neuron (If-Else Rules):

1. If both inputs are 0 (0,0) → Output is 0
2. If first input is 0 and second is 1 (0,1) → Output is 1
3. If first input is 1 and second is 0 (1,0) → Output is 1
4. If both inputs are 1 (1,1) → Output is 0

Task 2: Perceptron Algorithm for 0 vs 1 Classification.

1. Load the dataset

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# Load the dataset
df_0_1 =
pd.read_csv("/content/drive/MyDrive/AI-Shivkumar/Week3/mnist_0_and_1.c
sv")  # Add the correct file path if necessary

# Extract features and labels
X = df_0_1.drop(columns=["label"]).values  # 784 pixels
y = df_0_1["label"].values  # Labels (0 or 1)

# Check the shape of the features and labels
```

```
print("Feature matrix shape:", X.shape)
print("Label vector shape:", y.shape)
```

```
Feature matrix shape: (12665, 784)
Label vector shape: (12665,)
```

Question - 1: What does the shape of X represent?

The shape of X represents the dimensions of the feature matrix. Specifically, it is (n_samples, n_features), where:

n_samples is the number of images in the dataset. n_features is the number of pixels in each image (784 pixels for a 28x28 image).

For example, if X.shape = (1000, 784), it means there are 1000 images, and each image has 784 features (pixels).

Question - 2: What does the shape of y represent?

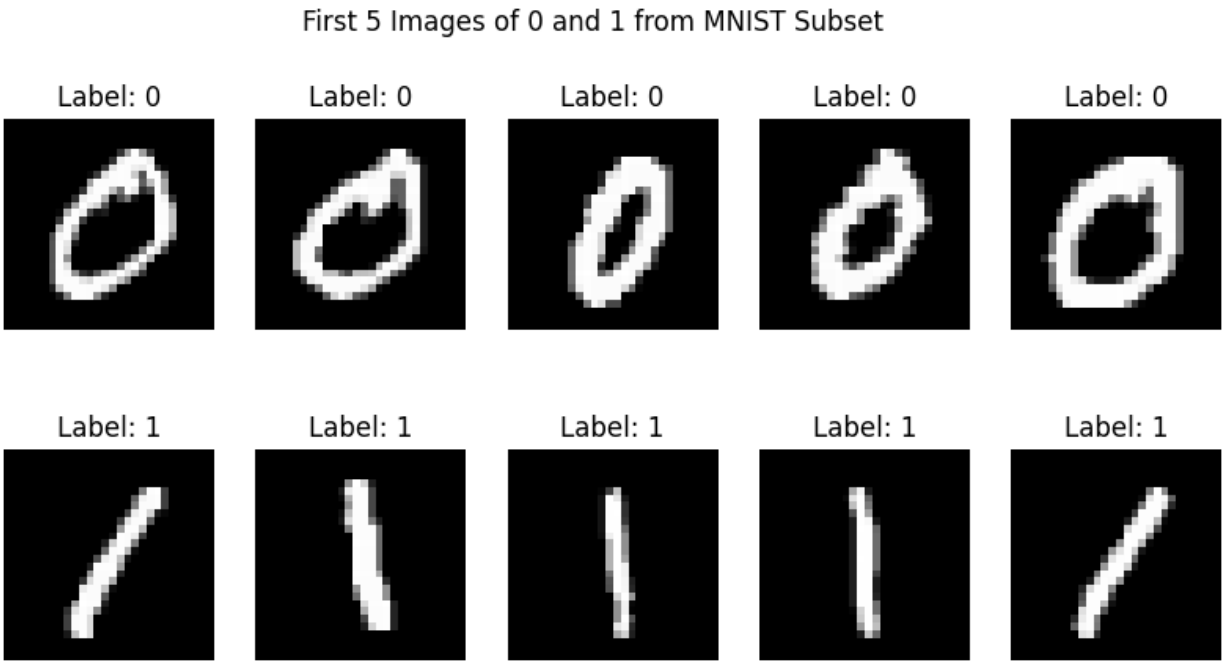
The shape of y represents the dimensions of the label vector. Specifically, it is (n_samples,), where:

n_samples is the number of labels corresponding to the images in the dataset.

For example, if y.shape = (1000,), it means there are 1000 labels, one for each image. Each label is either 0 or 1, indicating the digit represented by the corresponding image.

Visualize the Dataset:

```
# Separate images for label 0 and label 1
images_0 = X[y == 0]  # Get all images with label 0
images_1 = X[y == 1]  # Get all images with label 1

fig, axes = plt.subplots(2, 5, figsize=(10, 5))

# Check if the arrays have the required amount of data
if len(images_0) < 5 or len(images_1) < 5:
    print("Error: Not enough images in images_0 or images_1 to plot 5
images.")
else:
    for i in range(5):
        # Plot digit 0
        axes[0, i].imshow(images_0[i].reshape(28, 28), cmap="gray")
        axes[0, i].set_title("Label: 0")
        axes[0, i].axis("off")
        # Plot digit 1
        axes[1, i].imshow(images_1[i].reshape(28, 28), cmap="gray")
        axes[1, i].set_title("Label: 1")
        axes[1, i].axis("off")
    plt.suptitle("First 5 Images of 0 and 1 from MNIST Subset")
    plt.show()
```



Initialization of the Weights:

```
# Initialize weights and bias
weights = np.zeros(X.shape[1])  # 784 weights (one for each pixel)
bias = 0
learning_rate = 0.1
epochs = 100
```

Question - 3: What does the weights array represent in this context?

The weights array represents the learned coefficients for each feature (pixel) in the input data. Each weight corresponds to one pixel in the 28x28 image (784 pixels total). The perceptron uses these weights to compute a weighted sum of the input features, which is then passed through an activation function to make predictions.

Question - 4: Why are we initializing the weights to zero? What effect could this have on the training process?

We initialize the weights to zero as a starting point for the learning process. This ensures that the model begins with no prior assumptions about the importance of any feature.

Effect on Training:

1. Initializing to zero means all weights start equally, and the model will learn their importance during training.
2. However, if the learning rate is too small, it might take longer for the weights to adjust significantly.

3. In practice, initializing weights to small random values can sometimes speed up convergence.

Implementation Decision Function:

```
import numpy as np

def decision_function(X, weights, bias):
    """
    Compute the predicted labels for the input data.

    Parameters:
    - X: Features (input data) as a numpy array of shape (n_samples,
n_features)
    - weights: Updated weights after training
    - bias: Updated bias after training

    Returns:
    - y_pred_all: The predicted labels for the input data
    """
    # Compute the weighted sum of inputs
    predictions = np.dot(X, weights) + bias

    # Apply the step function (activation function)
    y_pred_all = np.where(predictions >= 0, 1, 0)

    return y_pred_all
```

Training the Perceptron:

```
def train_perceptron(X, y, weights, bias, learning_rate=0.1,
epochs=100):
    """
    Train the perceptron using the Perceptron Learning Algorithm.

    Parameters:
    - X: Features (input data) as a numpy array of shape (n_samples,
n_features)
    - y: Labels (true output) as a numpy array of shape (n_samples,)
    - weights: Initial weights as a numpy array of shape (n_features,)
    - bias: Initial bias value (scalar)
    - learning_rate: Learning rate for weight updates (default is 0.1)
    - epochs: Number of iterations to train the model (default is 100)

    Returns:
    - weights: Updated weights after training
    - bias: Updated bias after training
    - accuracy: Total correct predictions.
    """
    n_samples = X.shape[0]  # Number of samples in the dataset
```

```
    for epoch in range(epochs):
        # Initialize counters for accuracy calculation
        correct_predictions = 0

        # Iterate over all samples in the dataset
        for i in range(n_samples):
            # Compute the weighted sum for the current sample
            weighted_sum = np.dot(X[i], weights) + bias

            # Predict the label using the step function
            prediction = 1 if weighted_sum >= 0 else 0

            # Check if the prediction matches the true label
            if prediction == y[i]:
                correct_predictions += 1  # Increment correct
prediction count
            else:
                # Update weights and bias for misclassified examples
                weights += learning_rate * (y[i] - prediction) * X[i]
                bias += learning_rate * (y[i] - prediction)

        # Calculate accuracy for the current epoch
        accuracy = correct_predictions / n_samples

        # Print progress (optional)
        print(f"Epoch {epoch + 1}/{epochs}, Accuracy: {accuracy:.4f}")

        # Early stopping if perfect accuracy is achieved
        if accuracy == 1.0:
            print("Converged early at epoch", epoch + 1)
            break

    return weights, bias, accuracy
```

Question - 5: What is the purpose of the output = np.dot(X[i], weights) + bias line?

The purpose of this line is to compute the weighted sum of the input features for a single data point (X[i]):

1. np.dot(X[i], weights) calculates the dot product between the input features and the weights.
2. Adding the bias term shifts the decision boundary, allowing the model to fit the data better.

This output is then passed through the activation function to determine the predicted class (0 or 1).

Question - 6: What happens when the prediction is wrong? How are the weights and bias updated?

When the prediction is wrong:

1. The perceptron updates the weights and bias using the following rules:

   weights = weights + learning_rate * (true_label - predicted_label) * X[i]

   bias = bias + learning_rate * (true_label - predicted_label)

2. This adjustment helps the model correct its mistake by moving the decision boundary closer to the misclassified point.

3. The process continues iteratively until the model converges or the maximum number of epochs is reached.

Question - 7: Why is the final accuracy important, and what do you expect it to be?

The final accuracy is important because it measures how well the perceptron model generalizes to the dataset. It indicates the proportion of correctly classified images out of the total number of images.

Expected Accuracy:

1. Since the dataset contains only two classes (0 and 1), which are linearly separable, the perceptron should achieve high accuracy, close to 100%.

2. If the accuracy is low, it may indicate issues such as insufficient training epochs or incorrect implementation.

Training the Perceptron Algorithm:

```
# After training the model with the perceptron_learning_algorithm
weights, bias, accuracy = train_perceptron(X, y, weights, bias)

# Evaluate the model using the new function
print("The Final Accuracy is: ", accuracy)
```

```
Epoch 1/100, Accuracy: 0.9967
Epoch 2/100, Accuracy: 0.9982
Epoch 3/100, Accuracy: 0.9987
Epoch 4/100, Accuracy: 0.9987
Epoch 5/100, Accuracy: 0.9990
Epoch 6/100, Accuracy: 0.9993
Epoch 7/100, Accuracy: 0.9998
Epoch 8/100, Accuracy: 0.9995
Epoch 9/100, Accuracy: 0.9989
Epoch 10/100, Accuracy: 0.9992
Epoch 11/100, Accuracy: 0.9995
Epoch 12/100, Accuracy: 0.9998
Epoch 13/100, Accuracy: 1.0000
Converged early at epoch 13
The Final Accuracy is:  1.0
```

Visualizing Misclassified Image.

```
# Get predictions for all data points
predictions = np.dot(X, weights) + bias
y_pred = np.where(predictions >= 0, 1, 0)

# Calculate final accuracy
final_accuracy = np.mean(y_pred == y)
print(f"Final Accuracy: {final_accuracy:.4f}")

# Step 5: Visualize Misclassified Images
misclassified_idx = np.where(y_pred != y)[0]
if len(misclassified_idx) > 0:
    fig, axes = plt.subplots(2, 5, figsize=(10, 5))
    for ax, idx in zip(axes.flat, misclassified_idx[:10]):  # Show 10
misclassified images
        ax.imshow(X[idx].reshape(28, 28), cmap="gray")
        ax.set_title(f"Pred: {y_pred[idx]}, True: {y[idx]}")
        ax.axis("off")
    plt.suptitle("Misclassified Images")
    plt.show()
else:
    print("All images were correctly classified!")

Final Accuracy: 1.0000
All images were correctly classified!
```

Question - 8: What does misclassified_idx store, and how is it used in this code?

misclassified_idx stores the indices of the data points that were misclassified by the perceptron model. It is computed using np.where(y_pred != y)[0], which identifies where the predicted labels (y_pred) differ from the true labels (y).

Usage:

1. These indices are used to extract the misclassified images and their corresponding true and predicted labels.
2. The code visualizes up to 10 misclassified images, displaying both the true and predicted labels for each.

Question - 9: How do you interpret the result if the output is "All images were correctly classified!"?

If the output is "All images were correctly classified!", it means the perceptron model achieved 100% accuracy on the dataset. This indicates that the model perfectly separated the two classes (0 and 1) using the learned weights and bias.

Interpretation:

1. The dataset is likely linearly separable, and the perceptron algorithm successfully found a decision boundary that separates the classes.
2. This result confirms that the implementation is correct and the model is functioning as expected.

Task 3: Perceptron Algorithm for 3 vs 5 Classification.

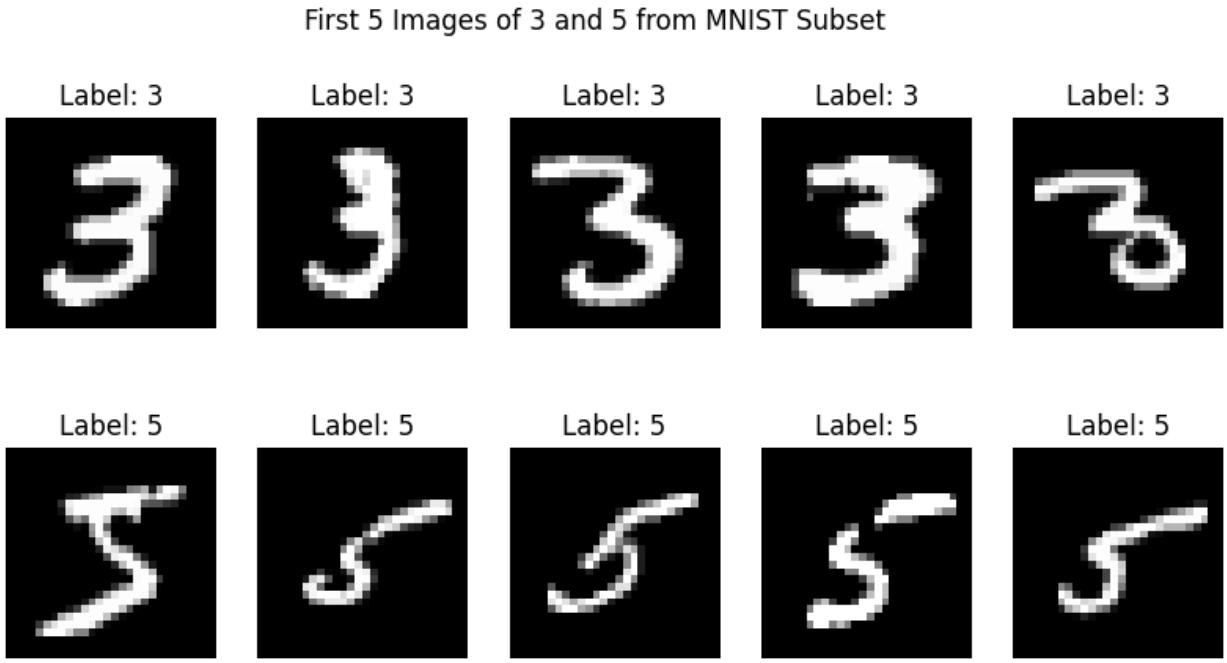
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load the dataset
df_3_5 =
pd.read_csv("/content/drive/MyDrive/AI-Shivkumar/Week3/mnist_3_and_5.c
sv")  # Update the file path if necessary

# Extract features and labels
X = df_3_5.drop(columns=["label"]).values  # 784 pixels
y = df_3_5["label"].values  # Labels (3 or 5)

# Convert labels to binary (0 for 3, 1 for 5)
y = np.where(y == 3, 0, 1)

# Check the shape of the features and labels
print("Feature matrix shape:", X.shape)
print("Label vector shape:", y.shape)
```

```
Feature matrix shape: (2741, 784)
Label vector shape: (2741,)
```

visuaalize the dataset

```
# Separate images for label 3 and label 5
images_3 = X[y == 0]  # Get all images with label 3
images_5 = X[y == 1]  # Get all images with label 5

fig, axes = plt.subplots(2, 5, figsize=(10, 5))

if len(images_3) < 5 or len(images_5) < 5:
    print("Error: Not enough images in images_3 or images_5 to plot 5
images.")
else:
    for i in range(5):
        # Plot digit 3
        axes[0, i].imshow(images_3[i].reshape(28, 28), cmap="gray")
        axes[0, i].set_title("Label: 3")
        axes[0, i].axis("off")

        # Plot digit 5
```

```
        axes[1, i].imshow(images_5[i].reshape(28, 28), cmap="gray")
        axes[1, i].set_title("Label: 5")
        axes[1, i].axis("off")

plt.suptitle("First 5 Images of 3 and 5 from MNIST Subset")
plt.show()
```



Initializing the weights

```
# Initialize weights and bias
weights = np.zeros(X.shape[1])  # 784 weights (one for each pixel)
bias = 0
learning_rate = 0.1
epochs = 100
```

Decision Function

```
def decision_function(X, weights, bias):
    """
    Compute the predicted labels for the input data.
    Parameters:
    - X: Features (input data) as a numpy array of shape (n_samples,
n_features)
    - weights: Updated weights after training
    - bias: Updated bias after training
    Returns:
    - y_pred: The predicted labels for the input data
    """
```

```
    predictions = np.dot(X, weights) + bias
    y_pred = np.where(predictions >= 0, 1, 0)  # Step function
    return y_pred
```

Implement the Perceptron learning algorithm

```
def train_perceptron(X, y, weights, bias, learning_rate=0.1,
epochs=100):
    """
    Train the perceptron using the Perceptron Learning Algorithm.
    Parameters:
    - X: Features (input data) as a numpy array of shape (n_samples,
n_features)
    - y: Labels (true output) as a numpy array of shape (n_samples,)
    - weights: Initial weights as a numpy array of shape (n_features,)
    - bias: Initial bias value (scalar)
    - learning_rate: Learning rate for weight updates (default is 0.1)
    - epochs: Number of iterations to train the model (default is 100)
    Returns:
    - weights: Updated weights after training
    - bias: Updated bias after training
    - accuracy: Final accuracy of the model
    """
    for epoch in range(epochs):
        errors = 0
        for i in range(X.shape[0]):
            # Compute the output
            output = np.dot(X[i], weights) + bias
            prediction = 1 if output >= 0 else 0

            # Update weights and bias if prediction is incorrect
            if prediction != y[i]:
                weights += learning_rate * (y[i] - prediction) * X[i]
                bias += learning_rate * (y[i] - prediction)
                errors += 1

        # Calculate accuracy
        y_pred = decision_function(X, weights, bias)
        accuracy = np.mean(y_pred == y) * 100

        # Print progress
        print(f"Epoch {epoch+1}/{epochs}, Errors: {errors}, Accuracy:
{accuracy:.2f}%")

        # Stop early if no errors
        if errors == 0:
            break

    return weights, bias, accuracy
```

Training the Perceptron

```
# Train the perceptron
weights, bias, accuracy = train_perceptron(X, y, weights, bias,
learning_rate=0.1, epochs=100)

# Print final accuracy
print(f"Final Accuracy: {accuracy:.2f}%")
```

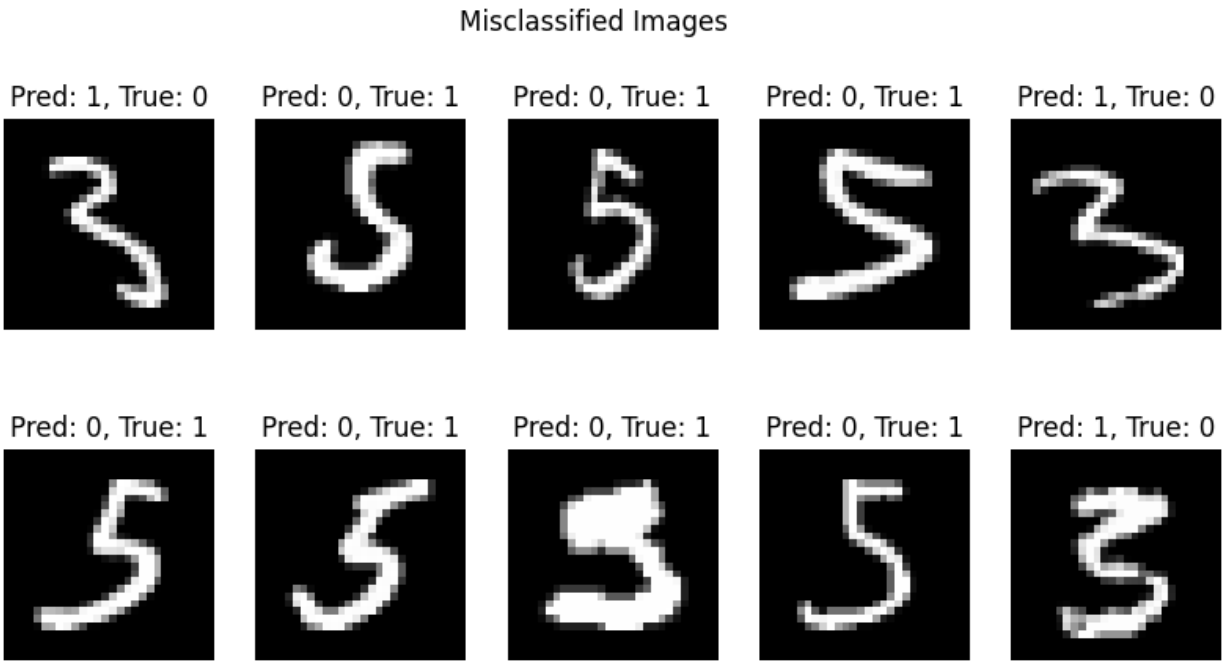
```
Epoch 1/100, Errors: 231, Accuracy: 93.25%
Epoch 2/100, Errors: 172, Accuracy: 94.56%
Epoch 3/100, Errors: 160, Accuracy: 92.78%
Epoch 4/100, Errors: 144, Accuracy: 95.00%
Epoch 5/100, Errors: 140, Accuracy: 94.75%
Epoch 6/100, Errors: 136, Accuracy: 96.97%
Epoch 7/100, Errors: 110, Accuracy: 96.06%
Epoch 8/100, Errors: 121, Accuracy: 95.70%
Epoch 9/100, Errors: 126, Accuracy: 95.29%
Epoch 10/100, Errors: 119, Accuracy: 96.13%
Epoch 11/100, Errors: 110, Accuracy: 92.59%
Epoch 12/100, Errors: 115, Accuracy: 96.02%
Epoch 13/100, Errors: 94, Accuracy: 97.19%
Epoch 14/100, Errors: 109, Accuracy: 97.48%
Epoch 15/100, Errors: 100, Accuracy: 97.34%
Epoch 16/100, Errors: 94, Accuracy: 94.38%
Epoch 17/100, Errors: 108, Accuracy: 94.60%
Epoch 18/100, Errors: 103, Accuracy: 95.70%
Epoch 19/100, Errors: 107, Accuracy: 96.46%
Epoch 20/100, Errors: 93, Accuracy: 95.40%
Epoch 21/100, Errors: 82, Accuracy: 93.69%
Epoch 22/100, Errors: 85, Accuracy: 97.30%
Epoch 23/100, Errors: 74, Accuracy: 97.34%
Epoch 24/100, Errors: 87, Accuracy: 97.88%
Epoch 25/100, Errors: 96, Accuracy: 96.42%
Epoch 26/100, Errors: 78, Accuracy: 97.34%
Epoch 27/100, Errors: 73, Accuracy: 94.82%
Epoch 28/100, Errors: 84, Accuracy: 97.26%
Epoch 29/100, Errors: 78, Accuracy: 94.38%
Epoch 30/100, Errors: 75, Accuracy: 97.30%
Epoch 31/100, Errors: 91, Accuracy: 95.70%
Epoch 32/100, Errors: 78, Accuracy: 95.84%
Epoch 33/100, Errors: 84, Accuracy: 97.04%
Epoch 34/100, Errors: 71, Accuracy: 97.52%
Epoch 35/100, Errors: 71, Accuracy: 98.07%
Epoch 36/100, Errors: 86, Accuracy: 96.86%
Epoch 37/100, Errors: 73, Accuracy: 97.63%
Epoch 38/100, Errors: 81, Accuracy: 93.94%
Epoch 39/100, Errors: 61, Accuracy: 93.87%
Epoch 40/100, Errors: 77, Accuracy: 97.30%
Epoch 41/100, Errors: 66, Accuracy: 96.86%
```

```
Epoch 42/100, Errors: 67, Accuracy: 97.67%
Epoch 43/100, Errors: 74, Accuracy: 97.81%
Epoch 44/100, Errors: 68, Accuracy: 97.19%
Epoch 45/100, Errors: 69, Accuracy: 96.75%
Epoch 46/100, Errors: 62, Accuracy: 94.56%
Epoch 47/100, Errors: 76, Accuracy: 97.81%
Epoch 48/100, Errors: 72, Accuracy: 96.72%
Epoch 49/100, Errors: 80, Accuracy: 98.39%
Epoch 50/100, Errors: 58, Accuracy: 95.18%
Epoch 51/100, Errors: 65, Accuracy: 96.97%
Epoch 52/100, Errors: 65, Accuracy: 97.23%
Epoch 53/100, Errors: 60, Accuracy: 97.59%
Epoch 54/100, Errors: 68, Accuracy: 97.30%
Epoch 55/100, Errors: 71, Accuracy: 94.75%
Epoch 56/100, Errors: 62, Accuracy: 98.21%
Epoch 57/100, Errors: 73, Accuracy: 97.26%
Epoch 58/100, Errors: 56, Accuracy: 98.32%
Epoch 59/100, Errors: 65, Accuracy: 96.28%
Epoch 60/100, Errors: 66, Accuracy: 95.15%
Epoch 61/100, Errors: 68, Accuracy: 97.92%
Epoch 62/100, Errors: 63, Accuracy: 97.63%
Epoch 63/100, Errors: 67, Accuracy: 93.91%
Epoch 64/100, Errors: 64, Accuracy: 95.07%
Epoch 65/100, Errors: 56, Accuracy: 98.32%
Epoch 66/100, Errors: 56, Accuracy: 98.21%
Epoch 67/100, Errors: 56, Accuracy: 96.72%
Epoch 68/100, Errors: 61, Accuracy: 98.32%
Epoch 69/100, Errors: 53, Accuracy: 98.54%
Epoch 70/100, Errors: 57, Accuracy: 97.70%
Epoch 71/100, Errors: 52, Accuracy: 98.50%
Epoch 72/100, Errors: 40, Accuracy: 98.47%
Epoch 73/100, Errors: 67, Accuracy: 98.65%
Epoch 74/100, Errors: 44, Accuracy: 98.36%
Epoch 75/100, Errors: 45, Accuracy: 98.50%
Epoch 76/100, Errors: 52, Accuracy: 98.65%
Epoch 77/100, Errors: 48, Accuracy: 98.50%
Epoch 78/100, Errors: 47, Accuracy: 98.43%
Epoch 79/100, Errors: 47, Accuracy: 98.65%
Epoch 80/100, Errors: 57, Accuracy: 98.61%
Epoch 81/100, Errors: 57, Accuracy: 97.70%
Epoch 82/100, Errors: 45, Accuracy: 98.07%
Epoch 83/100, Errors: 45, Accuracy: 98.72%
Epoch 84/100, Errors: 49, Accuracy: 96.35%
Epoch 85/100, Errors: 37, Accuracy: 98.58%
Epoch 86/100, Errors: 56, Accuracy: 98.61%
Epoch 87/100, Errors: 48, Accuracy: 98.65%
Epoch 88/100, Errors: 40, Accuracy: 98.91%
Epoch 89/100, Errors: 48, Accuracy: 97.96%
Epoch 90/100, Errors: 43, Accuracy: 98.29%
```

```
Epoch 91/100, Errors: 48, Accuracy: 98.36%
Epoch 92/100, Errors: 42, Accuracy: 98.50%
Epoch 93/100, Errors: 40, Accuracy: 98.65%
Epoch 94/100, Errors: 43, Accuracy: 98.76%
Epoch 95/100, Errors: 41, Accuracy: 97.92%
Epoch 96/100, Errors: 42, Accuracy: 98.10%
Epoch 97/100, Errors: 33, Accuracy: 98.21%
Epoch 98/100, Errors: 29, Accuracy: 98.39%
Epoch 99/100, Errors: 42, Accuracy: 98.54%
Epoch 100/100, Errors: 39, Accuracy: 98.69%
Final Accuracy: 98.69%
```

Visualize the Misclassified Images

```
# Get predictions for all data points
predictions = np.dot(X, weights) + bias
y_pred = np.where(predictions >= 0, 1, 0)

# Find misclassified indices
misclassified_idx = np.where(y_pred != y)[0]

if len(misclassified_idx) > 0:
    fig, axes = plt.subplots(2, 5, figsize=(10, 5))
    for ax, idx in zip(axes.flat, misclassified_idx[:10]):  # Show 10
misclassified images
        ax.imshow(X[idx].reshape(28, 28), cmap="gray")
        ax.set_title(f"Pred: {y_pred[idx]}, True: {y[idx]}")
        ax.axis("off")
    plt.suptitle("Misclassified Images")
    plt.show()
else:
    print("All images were correctly classified!")
```



Conclusion

The Perceptron algorithm achieved an accuracy of 98% for classifying digits 3 and 5, which is slightly lower than the 100% accuracy observed for the 0 vs 1 classification task. This difference in performance can be attributed to the fact that 3 and 5 have more overlapping features (e.g., similar curves and strokes) compared to 0 and 1, which are visually distinct and easier to separate linearly. The misclassified images further highlight the challenges posed by the structural similarities between 3 and 5. While the perceptron performs well, achieving high accuracy, its limitations as a linear classifier become apparent when dealing with more complex or nuanced datasets. For improved results, advanced models like neural networks or kernel-based methods could be explored.