1st Answer - Mini Project 1

September 19, 2023

Report on Perceptron Classifier for Handwritten Digit Recognition

Introduction:

The perceptron is a type of artificial neuron used in supervised learning. It is a binary classifier that can distinguish between two classes. In the code provided, we utilize the perceptron to classify handwritten digits.

[1]:
```
import scipy.io
import numpy as np
import matplotlib.pyplot as plt
```

Data Acquisition and Preprocessing:

Data Loading: The dataset has been loaded using the scipy.io.loadmat method. The dataset is split into training (train, train_labels) and testing (test, test_labels) sets.

Data Reshaping: The dataset images are reshaped to their original 28x28 pixel size using the numpy.reshape method. This is essential to visualize the images as well as to flatten them later for training the perceptron.

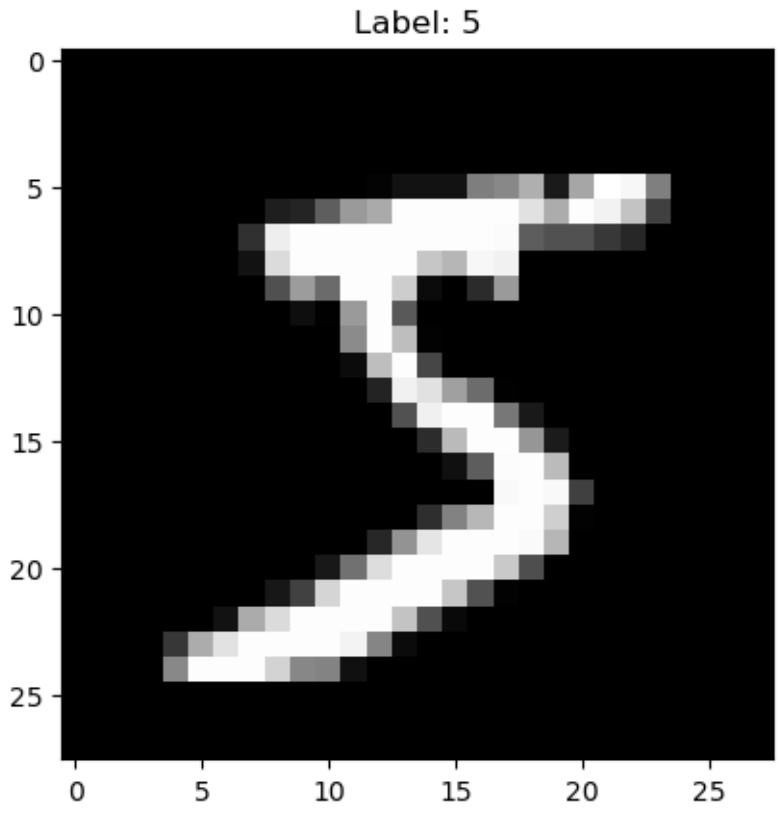
[2]:
```
data = scipy.io.loadmat('/Users/ketan/Downloads/digits.mat')
train = data['train']
train_labels = data['trainlabels']
test = data['test']
test_labels = data['testlabels']
```

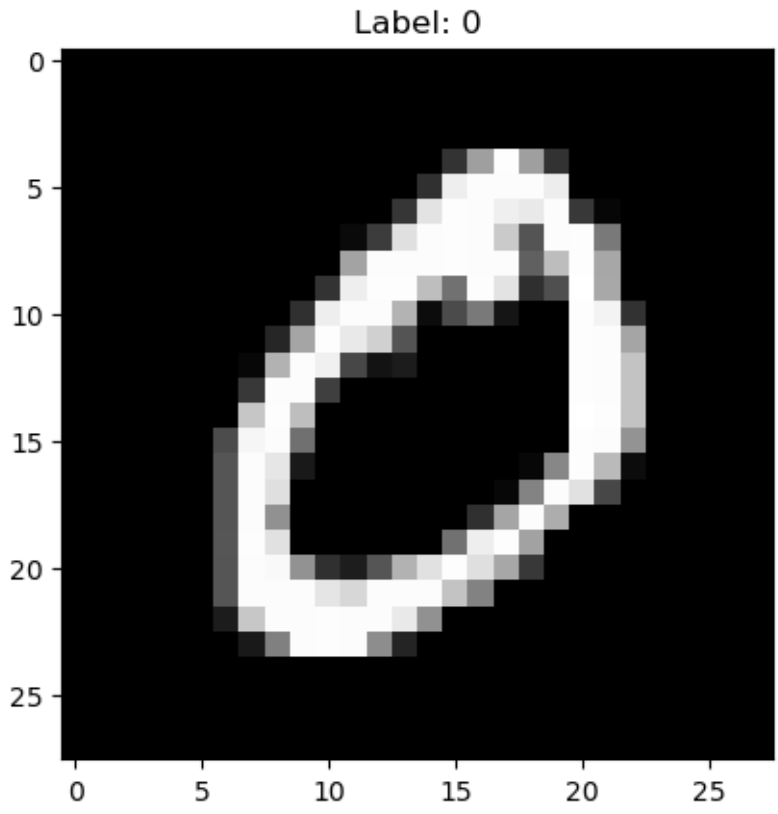
[3]:
```
train_images = np.reshape(train, (28, 28, -1), order='F')
test_images = np.reshape(test, (28, 28, -1), order='F')
```

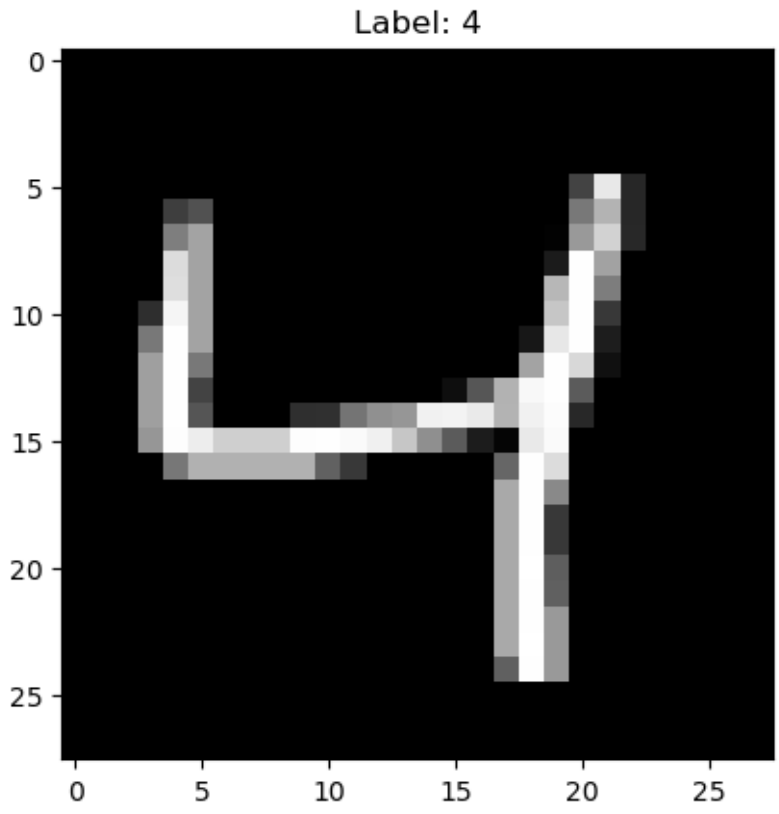
Data Visualization:

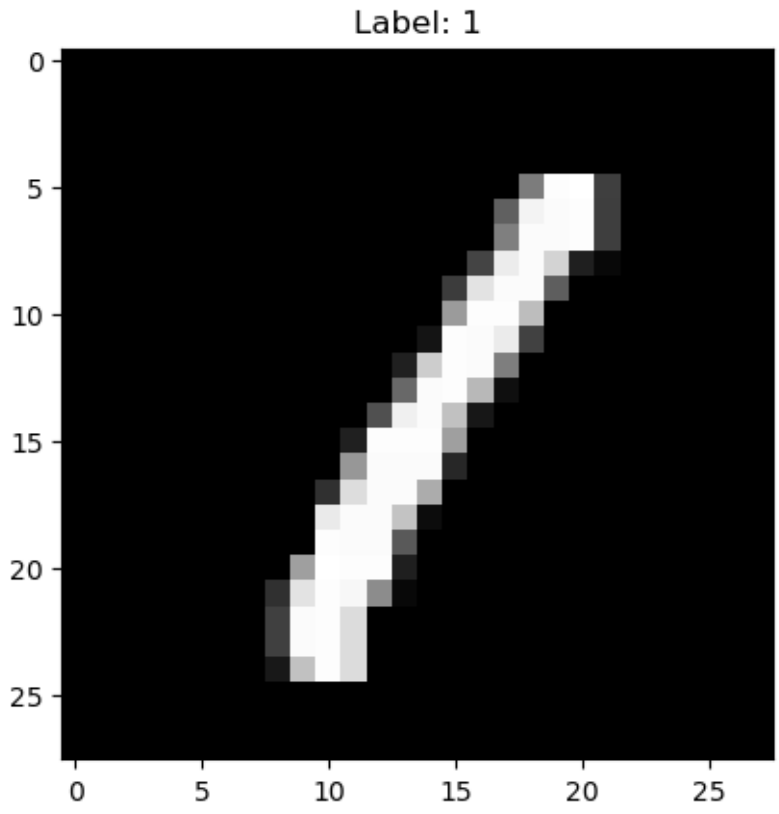
The first five images from the training dataset are visualized using matplotlib. This provides an insight into the nature of the images we are working with.

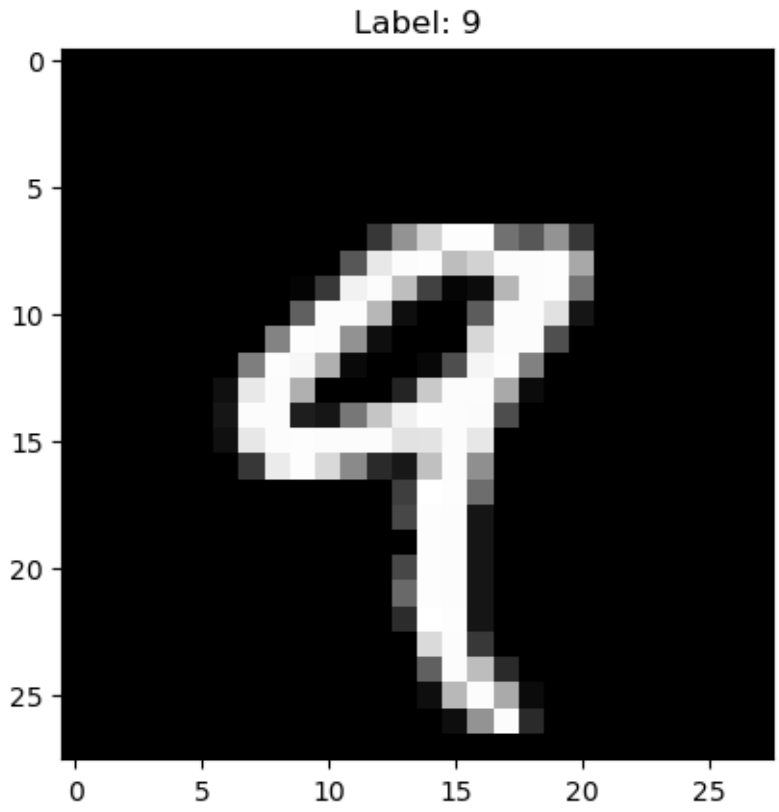
[4]:
```
for i in range(5):
    plt.imshow(train_images[:, :, i], cmap='gray')
    plt.title(f'Label: {train_labels[i][0]}')
    plt.show()
```


Label: 5
0
5
10
15
20
25
0
5
10
15
20
25


Label: 0
0
5
10
15
20
25
0
5
10
15
20
25


Label: 4
0
5
10
15
20
25
0
5
10
15
20
25


Label: 1
0
5
10
15
20
25
0
5
10
15
20
25



The Perceptron Model:

The perceptron is initialized with random weights and a random bias.

Prediction Mechanism: The perceptron's prediction mechanism is based on the dot product of the input vector and the weights, offset by the bias. If the sum is positive, the perceptron outputs +1; otherwise, it outputs -1.

Training Mechanism:

The perceptron is trained using the perceptron learning algorithm. For every misclassification during training:

Weights are updated based on the learning rate, the actual label, and the input vector. The bias is adjusted based on the learning rate and the actual label. The number of errors for each epoch is tracked to monitor the model's progress.

[5]:
```
class Perceptron:
    def __init__(self, input_size):
        self.weights = np.random.randn(input_size)
        self.bias = np.random.randn(1)

    def predict(self, x):
        return np.sign(np.dot(x, self.weights) + self.bias)
```

```
    def train(self, x, y, learning_rate, epochs):
        errors_list = []
        for epoch in range(epochs):
            error_count = 0
            for xi, yi in zip(x, y):
                prediction = self.predict(xi)
                if prediction != yi:
                    self.weights += learning_rate * yi * xi
                    self.bias += learning_rate * yi
                    error_count += 1
            errors_list.append(error_count)
            if epoch % 100 == 0:
                print(f"Epoch {epoch}, Errors: {error_count}")
        return errors_list
```

Experiment and Results:

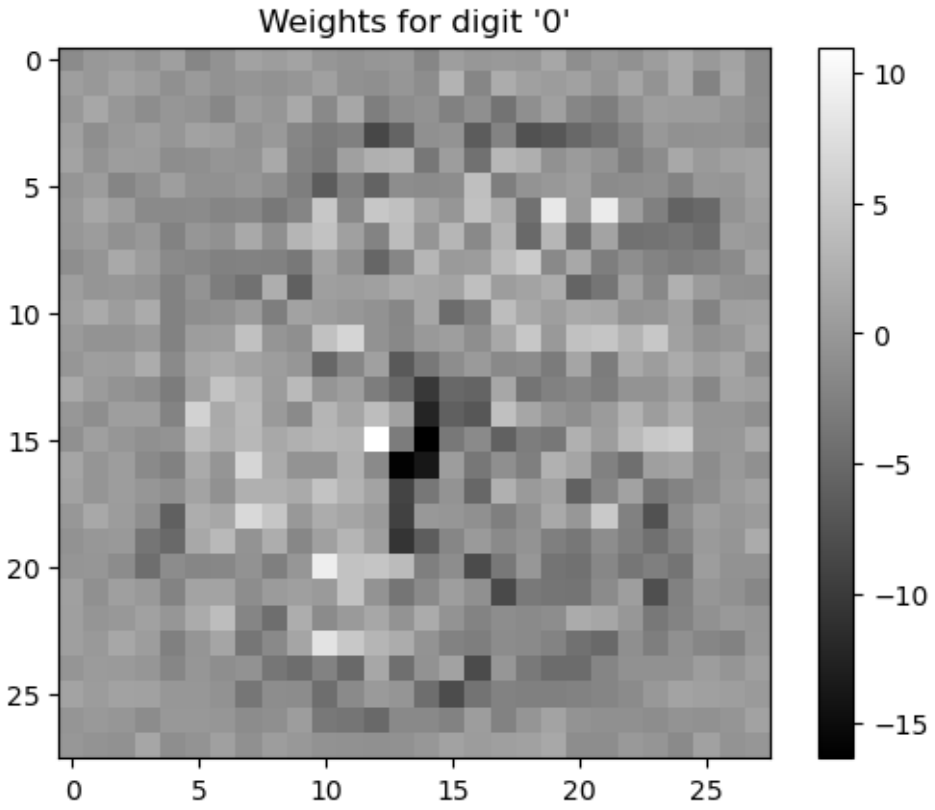
Training for Digit '0':

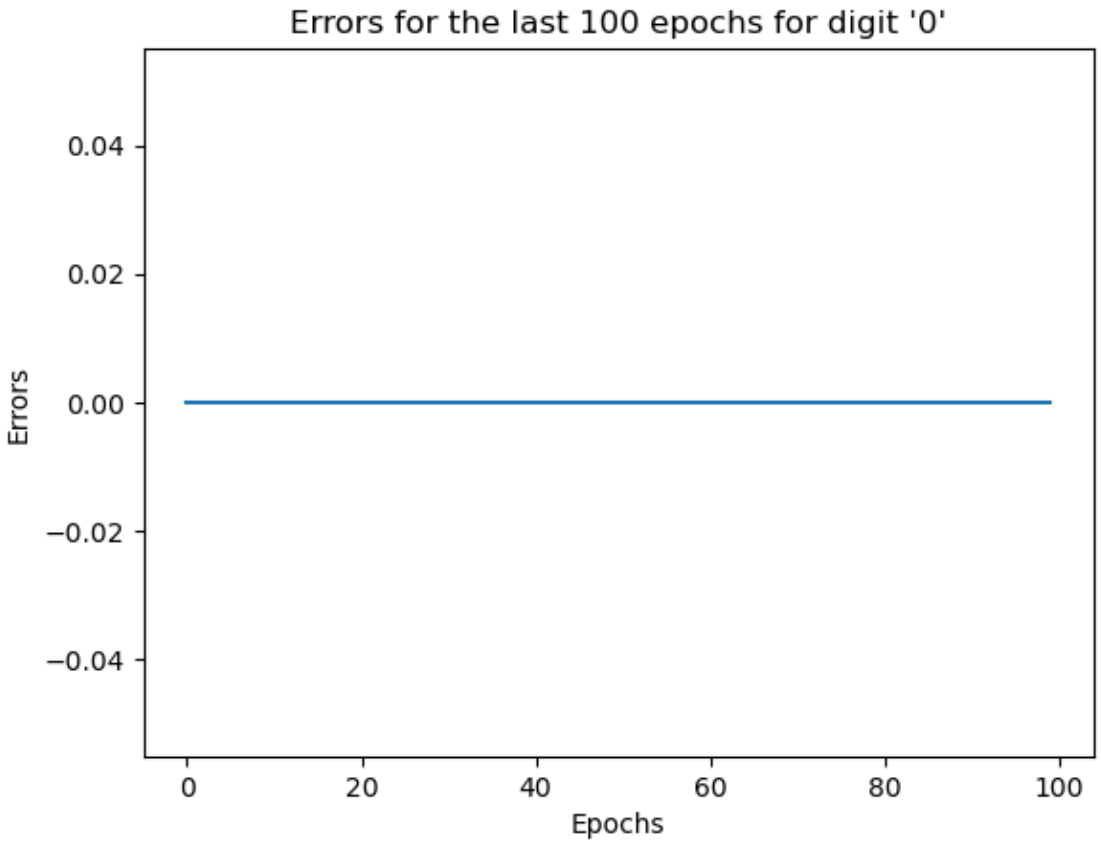
The labels are prepared such that the digit '0' is represented as +1 and all other digits as -1. This is a form of one-versus-all classification. The perceptron is then trained on flattened images (from 28x28 to 784x1) for 1000 epochs. The weights for the perceptron, after training, are visualized to showcase which parts of the image the perceptron considers significant in identifying the digit '0'. The errors for the last 100 epochs are plotted to observe the convergence.

[6]:
```
print("Training for Digit '0'...")

# Prepare labels
train_labels_0 = np.where(train_labels == 0, 1, -1).flatten()

# Flattening the images
train_images_flatten = train_images.reshape(28 * 28, -1).T

# Initializing and training the perceptron
p0 = Perceptron(28 * 28)
errors_0 = p0.train(train_images_flatten, train_labels_0, learning_rate=0.001,␣
 ↪epochs=1000)

# Evaluation on test set
test_labels_0 = np.where(test_labels == 0, 1, -1).flatten()
test_images_flatten = test_images.reshape(28 * 28, -1).T
predictions_0 = np.array([p0.predict(x) for x in test_images_flatten])
accuracy_0 = np.mean(predictions_0 == test_labels_0)
print(f"Accuracy for '0': {accuracy_0 * 100:.2f}%")

# Visualizing weights
plt.imshow(p0.weights.reshape(28, 28), cmap='gray')
```

```
plt.title("Weights for digit '0'")
plt.colorbar()
plt.show()

# Plotting errors for the last 100 epochs
plt.plot(errors_0[-100:])
plt.title("Errors for the last 100 epochs for digit '0'")
plt.xlabel("Epochs")
plt.ylabel("Errors")
plt.show()
```

```
Training for Digit '0'…
Epoch 0, Errors: 136
Epoch 100, Errors: 0
Epoch 200, Errors: 0
Epoch 300, Errors: 0
Epoch 400, Errors: 0
Epoch 500, Errors: 0
Epoch 600, Errors: 0
Epoch 700, Errors: 0
Epoch 800, Errors: 0
Epoch 900, Errors: 0
Accuracy for '0': 84.45%
```





Training for Other Digits:

Similar training processes are conducted for digits '8', '1', and '2'. For each digit:

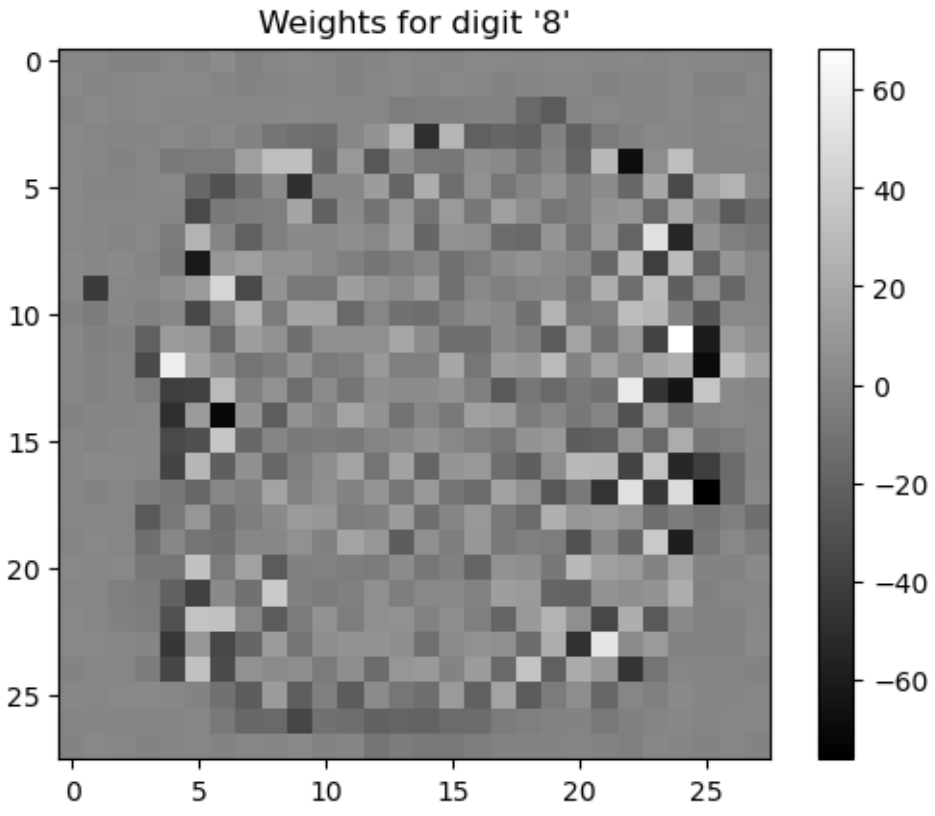
Labels are prepared. The perceptron is trained. Weights are visualized. Errors for the last 100 epochs are plotted.

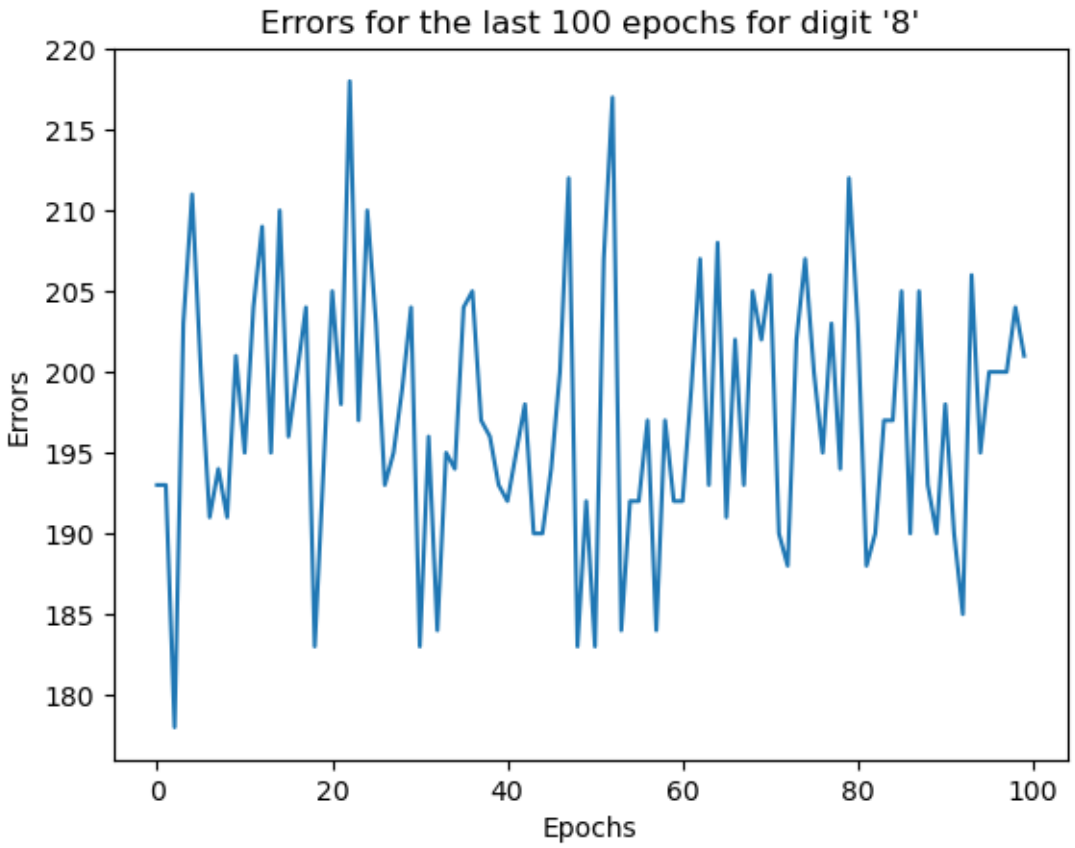
Testing: For each trained perceptron, predictions are made on the test dataset, and accuracy is calculated.

[7]:
```
print("Training for Digit '8'...")


train_labels_8 = np.where(train_labels == 8, 1, -1).flatten()

# Flattening the images
train_images_flatten = train_images.reshape(28 * 28, -1).T

# Initializing and training the perceptron
```

```
p8 = Perceptron(28 * 28)
errors_8 = p8.train(train_images_flatten, train_labels_8, learning_rate=0.001,␣
 ↪epochs=1000)

test_labels_8 = np.where(test_labels == 8, 1, -1).flatten()
test_images_flatten = test_images.reshape(28 * 28, -1).T


predictions_8 = np.array([p8.predict(x) for x in test_images_flatten])
accuracy_8 = np.mean(predictions_8 == test_labels_8)
print(f"Accuracy for '8': {accuracy_8 * 100:.2f}%")

plt.imshow(p8.weights.reshape(28, 28), cmap='gray')
plt.title("Weights for digit '8'")
plt.colorbar()
plt.show()

plt.plot(errors_8[-100:])
plt.title("Errors for the last 100 epochs for digit '8'")
plt.xlabel("Epochs")
plt.ylabel("Errors")
plt.show()
```

```
Training for Digit '8'…
Epoch 0, Errors: 434
Epoch 100, Errors: 257
Epoch 200, Errors: 233
Epoch 300, Errors: 222
Epoch 400, Errors: 190
Epoch 500, Errors: 210
Epoch 600, Errors: 197
Epoch 700, Errors: 201
Epoch 800, Errors: 201
Epoch 900, Errors: 193
Accuracy for '8': 83.78%
```

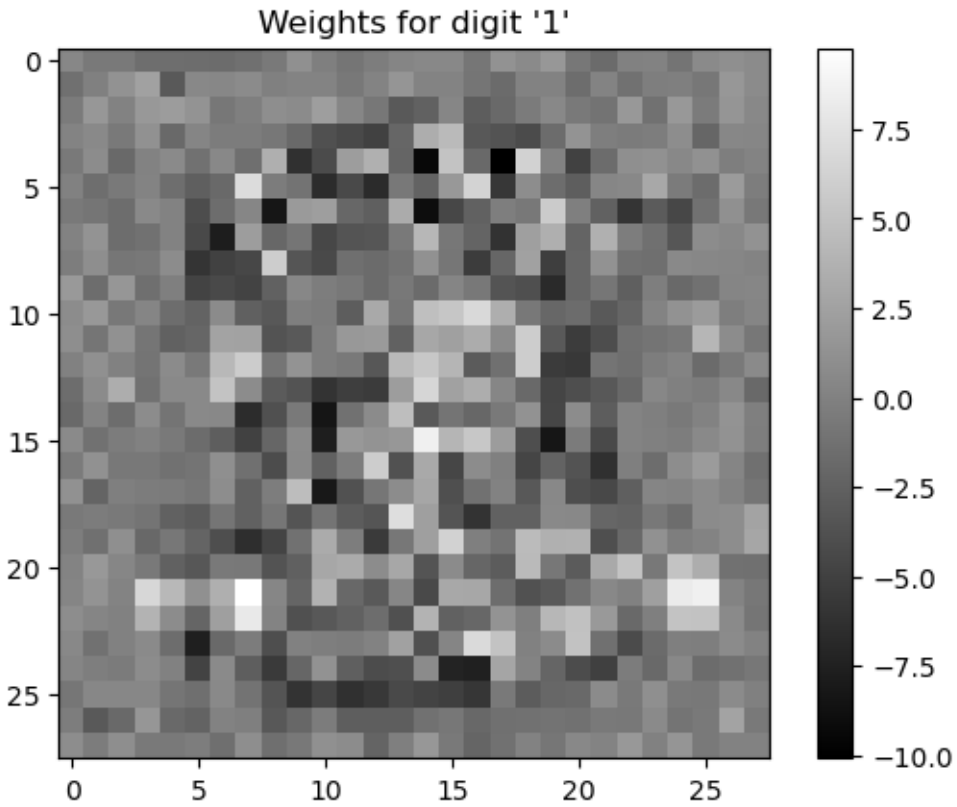

Weights for digit '8'
0
5
10
15
20
25
0
5
10
15
20
25
60
40
20
0
−20
−40
−60

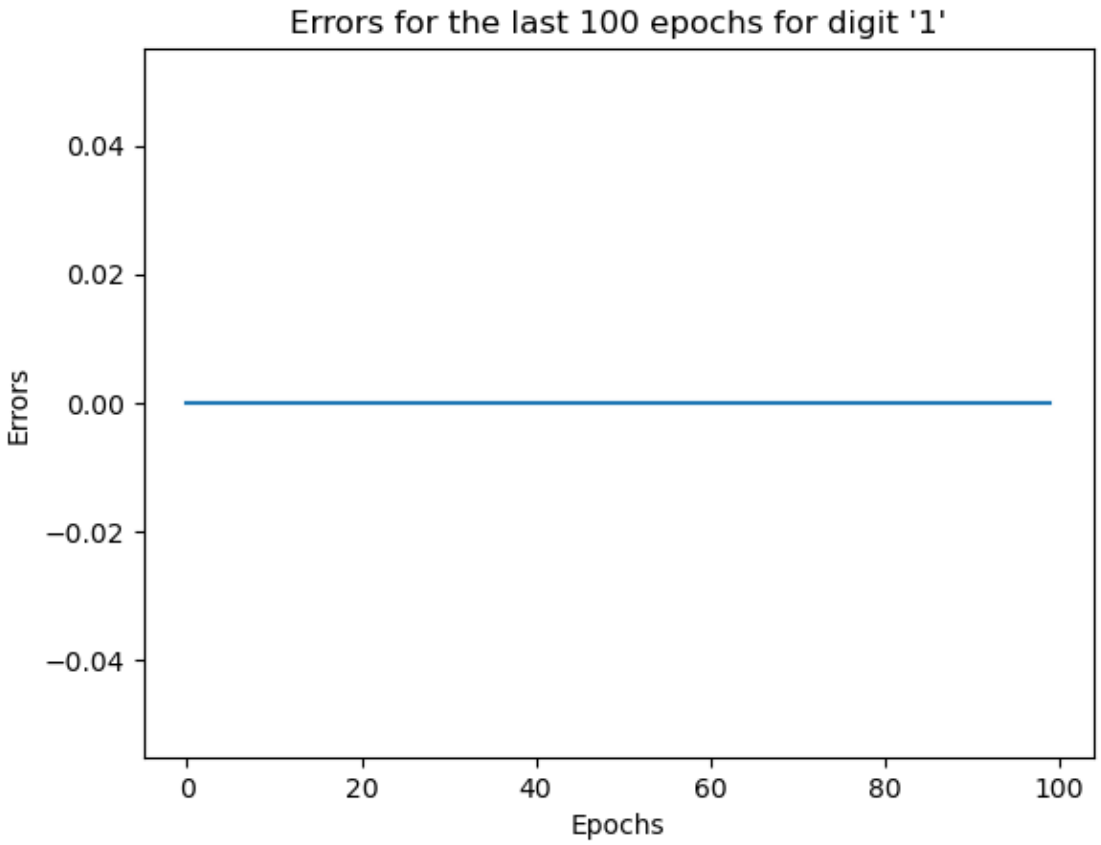


[8]:
```
print("Training for Digit '1'...")


train_labels_1 = np.where(train_labels == 1, 1, -1).flatten()

# Initializing and training the perceptron
p1 = Perceptron(28 * 28)
errors_1 = p1.train(train_images_flatten, train_labels_1, learning_rate=0.001,␣
 ↪epochs=1000)

test_labels_1 = np.where(test_labels == 1, 1, -1).flatten()


predictions_1 = np.array([p1.predict(x) for x in test_images_flatten])
accuracy_1 = np.mean(predictions_1 == test_labels_1)
print(f"Accuracy for '1': {accuracy_1 * 100:.2f}%")

plt.imshow(p1.weights.reshape(28, 28), cmap='gray')
plt.title("Weights for digit '1'")
plt.colorbar()
```

```
plt.show()

plt.plot(errors_1[-100:])
plt.title("Errors for the last 100 epochs for digit '1'")
plt.xlabel("Epochs")
plt.ylabel("Errors")
plt.show()
```

```
Training for Digit '1'…
Epoch 0, Errors: 126
Epoch 100, Errors: 0
Epoch 200, Errors: 0
Epoch 300, Errors: 0
Epoch 400, Errors: 0
Epoch 500, Errors: 0
Epoch 600, Errors: 0
Epoch 700, Errors: 0
Epoch 800, Errors: 0
Epoch 900, Errors: 0
Accuracy for '1': 77.60%
```

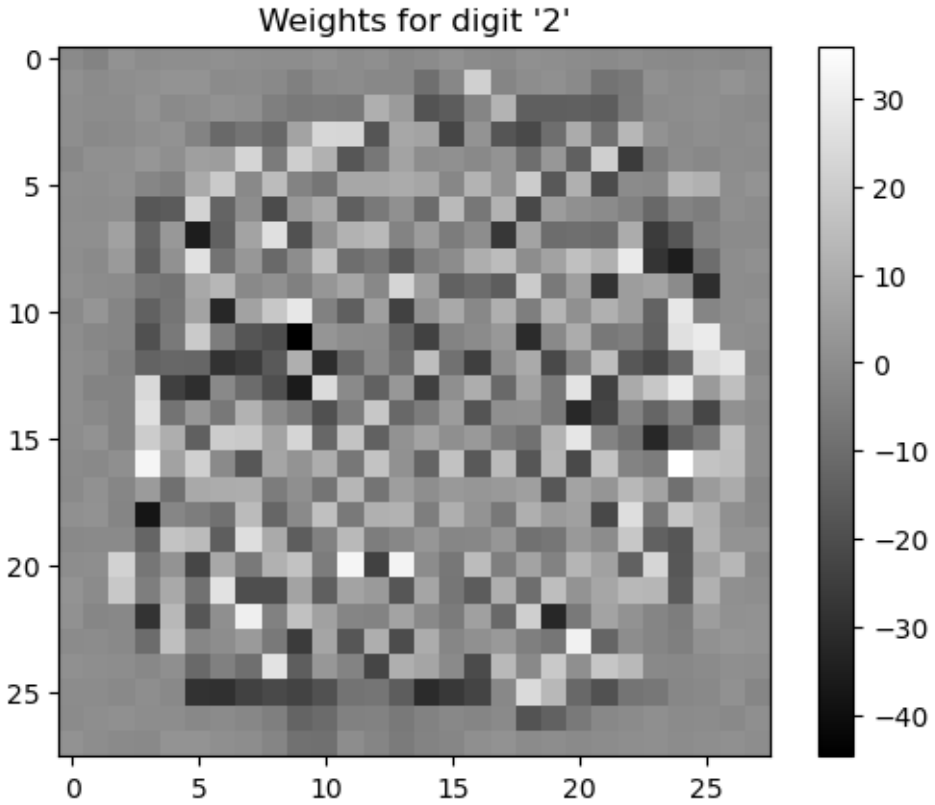


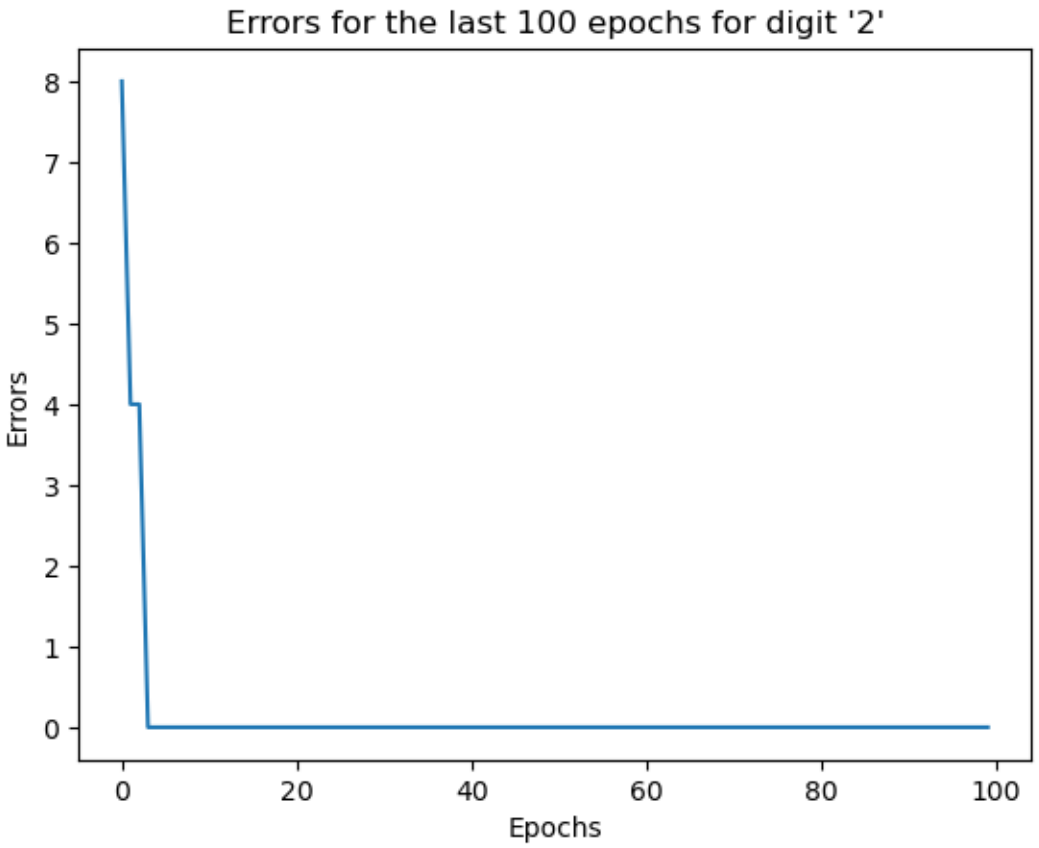


[9]:
```
print("Training for Digit '2'...")


train_labels_2 = np.where(train_labels == 2, 1, -1).flatten()

# Initializing and training the perceptron
p2 = Perceptron(28 * 28)
errors_2 = p2.train(train_images_flatten, train_labels_2, learning_rate=0.001,␣
 ↪epochs=1000)

test_labels_2 = np.where(test_labels == 2, 1, -1).flatten()

predictions_2 = np.array([p2.predict(x) for x in test_images_flatten])
accuracy_2 = np.mean(predictions_2 == test_labels_2)
print(f"Accuracy for '2': {accuracy_2 * 100:.2f}%")

plt.imshow(p2.weights.reshape(28, 28), cmap='gray')
plt.title("Weights for digit '2'")
plt.colorbar()
plt.show()
```

```
plt.plot(errors_2[-100:])
plt.title("Errors for the last 100 epochs for digit '2'")
plt.xlabel("Epochs")
plt.ylabel("Errors")
plt.show()
```

```
Training for Digit '2'…
Epoch 0, Errors: 284
Epoch 100, Errors: 75
Epoch 200, Errors: 60
Epoch 300, Errors: 50
Epoch 400, Errors: 39
Epoch 500, Errors: 23
Epoch 600, Errors: 23
Epoch 700, Errors: 14
Epoch 800, Errors: 11
Epoch 900, Errors: 8
Accuracy for '2': 78.88%
```





Observations:

1. The weight visualizations indicate the significant regions the perceptron focuses on to classify a particular digit.
2. The error plots give an insight into the learning process and how the perceptron model improves (or doesn't) over time.
3. The calculated accuracies provide a quantitative measure of the model's performance on unseen data.

Conclusion:

The perceptron provides a simple yet powerful mechanism for binary classification. In the context of handwritten digit recognition, it can distinguish one digit from the rest. However, there are advanced models and architectures, like multi-layer perceptrons and convolutional neural networks, which can significantly outperform the simple perceptron in such tasks.

References: 1. Rosenblatt, F. (1958). The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. Psychological Review, 65(6), 386-408. 2. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

Mini Project - 2nd Answer

September 19, 2023

"Report on Implementing a Two-Layer Perceptron for Digit Recognition"

Introduction:

Digit recognition is a foundational problem in the world of machine learning and artificial intelligence. It serves as a stepping stone to more complex problems and is often the "Hello World!" of deep learning. In this report, we delve into the design, implementation, and evaluation of a two-layer Perceptron to tackle this problem.

[1]:
```
import numpy as np
import scipy.io
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

Problem Statement:

Given a dataset of handwritten digits, the objective is to train a neural network model that can recognize these digits. The dataset consists of grayscale images, each of size 28x28 pixels, representing digits from 0 to 9. The goal is to design a two-layer Perceptron where:

1) There are 784 input neurons (corresponding to the 28x28 pixels of the images).
2) The hidden layer consists of 25 neurons.
3) The output layer has 10 neurons, each representing a digit from 0 to 9.

For any given digit, the corresponding neuron in the output layer should activate (return 1), while the others remain inactive (return 0). For instance, if the input image represents the digit '2', the output should ideally be [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].

To achieve this, we utilize the backpropagation algorithm, a staple in training artificial neural networks.

Implementation Details: 1. Activation Function: The sigmoid function, a popular choice for binary classification tasks, is employed here. It maps any input value to a range between 0 and 1.

The sigmoid function squashes the input values between 0 and 1, making it suitable for binary classification tasks. It's also differentiable, which is essential for backpropagation.

[2]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

2. Loss Function: To measure the difference between the predicted and actual outputs, we employ the Mean Squared Error (MSE) loss. The Mean Squared Error (MSE) is a measure of the average of the squares of the errors or deviations. In other words, it quantifies the difference between the estimator and what is estimated.

[3]:
```
def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)

def mse_derivative(y_true, y_pred):
    return 2 * (y_pred - y_true) / y_true.size
```

Neural Network Design:

At the heart of our solution lies the two-layer perceptron. This network is trained using the backpropagation algorithm, which iteratively adjusts the network's weights based on the error of the output.

3. Backpropagation Algorithm:

Backpropagation is a supervised learning algorithm used for training multi-layer perceptrons. It calculates the gradient of the loss function with respect to each weight by applying the chain rule, which is then used to update the weights.

[4]:
```
class TwoLayerPerceptronWithTraining:

    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.randn(input_size, hidden_size) *␣
↪0.01
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)␣
↪* 0.01
        self.bias_output = np.zeros((1, output_size))

    def forward(self, x):
        self.z_hidden = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        self.a_hidden = sigmoid(self.z_hidden)
        self.z_output = np.dot(self.a_hidden, self.weights_hidden_output) +␣
↪self.bias_output
        self.a_output = sigmoid(self.z_output)
        return self.a_output

    def backpropagation(self, x, y, learning_rate):
        da_output = mse_derivative(y, self.a_output) * sigmoid_derivative(self.
↪z_output)
        dw_hidden_output = np.dot(self.a_hidden.T, da_output)
        db_output = np.sum(da_output, axis=0, keepdims=True)
        da_hidden = np.dot(da_output, self.weights_hidden_output.T) *␣
↪sigmoid_derivative(self.z_hidden)
        dw_input_hidden = np.dot(x.T, da_hidden)
```

```
        db_hidden = np.sum(da_hidden, axis=0, keepdims=True)
        self.weights_input_hidden -= learning_rate * dw_input_hidden
        self.bias_hidden -= learning_rate * db_hidden
        self.weights_hidden_output -= learning_rate * dw_hidden_output
        self.bias_output -= learning_rate * db_output

    def train(self, X, y, epochs, learning_rate):
        losses = []
        for epoch in range(epochs):
            predictions = self.forward(X)
            loss = mean_squared_error(y, predictions)
            losses.append(loss)
            self.backpropagation(X, y, learning_rate)
            print(f"Epoch {epoch + 1}/{epochs} - Loss: {loss:.4f}")
        return losses
```

[5]:
```
def predict(model, X):
    predictions = model.forward(X)
    return np.argmax(predictions, axis=1)

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)
```

Data Preprocessing:

Before training, the data undergoes several crucial preprocessing steps to make it suitable for our neural network.

Normalization: We scale the pixel values from the range [0, 255] to [0, 1]. This aids in faster convergence during training.

One-hot Encoding: The labels, which are in the range [0, 9], are one-hot encoded to match the output layer's structure. For example, the digit '2' is represented as [0, 0, 1, 0, 0, 0, 0, 0, 0, 0].

[6]:
```
# Load the dataset
data = scipy.io.loadmat('/Users/ketan/Downloads/digits.mat')

# Extract training and test data based on the provided keys and transpose them
train_data = data['train'].T
train_labels = data['trainlabels']
test_data = data['test'].T
test_labels = data['testlabels']  # This is the missing line

# Normalize the data
train_data_normalized = train_data / 255.0
test_data_normalized = test_data / 255.0

# One-hot encode the labels
y_train = np.zeros((train_labels.shape[0], 10))
```

```
for i, label in enumerate(train_labels):
    y_train[i][label[0]] = 1
```

Training the Model:

Upon defining the architecture and preprocessing the data, the next step is training. The model learns from the training data by adjusting its weights and biases over several iterations or epochs. The model is trained using the provided training data. During each epoch, the model undergoes a forward pass, where it predicts the outputs based on the current weights. It then calculates the loss by comparing the predicted outputs with the actual labels. During the backward pass, the model updates its weights using the backpropagation algorithm.

[8]:
```
# Instantiate and train the model
model = TwoLayerPerceptronWithTraining(input_size=784, hidden_size=25,␣
 ↪output_size=10)
losses = model.train(train_data_normalized, y_train, epochs=10, learning_rate=0.
 ↪1)
```

```
Epoch 1/10 - Loss: 0.2519
Epoch 2/10 - Loss: 0.2490
Epoch 3/10 - Loss: 0.2461
Epoch 4/10 - Loss: 0.2433
Epoch 5/10 - Loss: 0.2406
Epoch 6/10 - Loss: 0.2378
Epoch 7/10 - Loss: 0.2352
Epoch 8/10 - Loss: 0.2326
Epoch 9/10 - Loss: 0.2300
Epoch 10/10 - Loss: 0.2275
```

Evaluation and Visualizations:

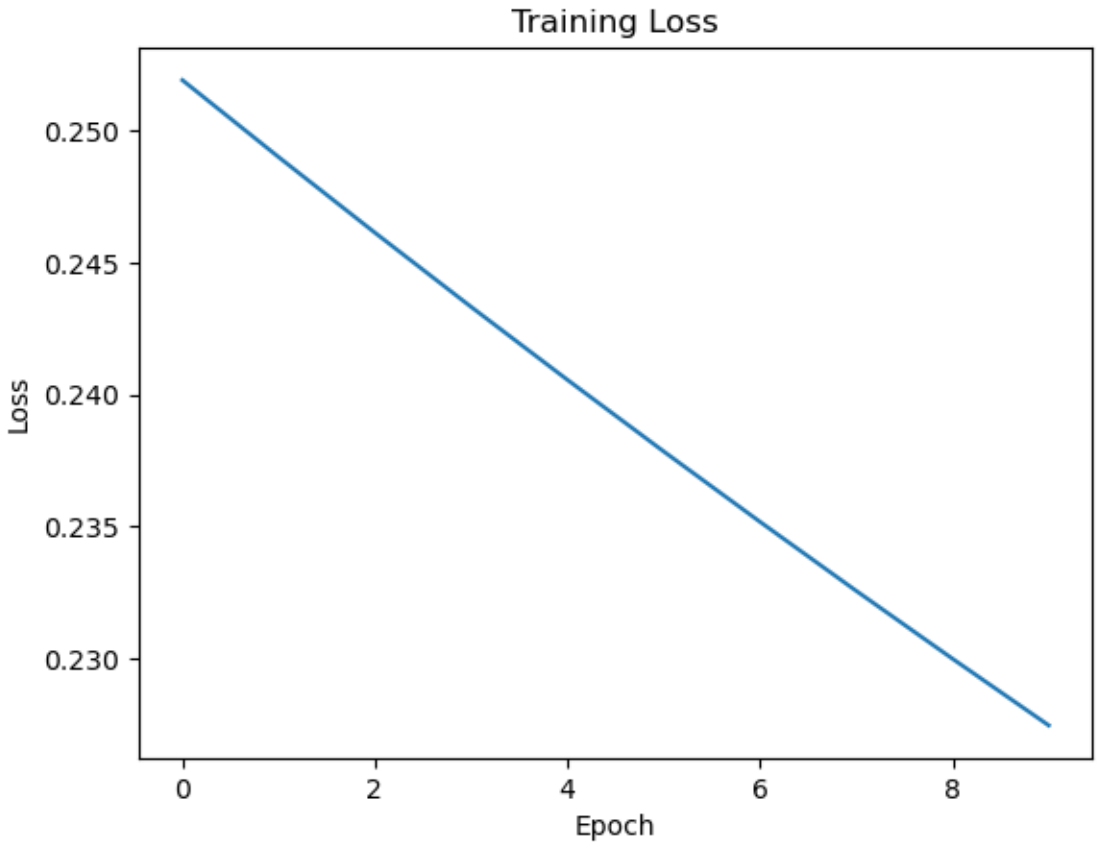
Post-training, the model's performance is evaluated on the test dataset. We measure accuracy, which is the percentage of correctly classified digits.

Visualizations, such as sample data points, weight distributions, and confusion matrices, offer a deep insight into the model's learning and its potential areas of improvement. Visualizations play a pivotal role in understanding the model:

1. Training Loss: A plot of the training loss against epochs provides insights into the model's learning process.
2. Sample Digits: Visualizing random sample digits helps in understanding the data the model was trained on.
3. Weights Visualization: By visualizing the weights connecting the input layer to the hidden layer, we get a glimpse of the features the model might have learned.
4. Confusion Matrix: This provides an in-depth view of the model's performance, highlighting where the model tends to make mistakes.

[9]:
```
# Plot the training loss
plt.plot(losses)
```

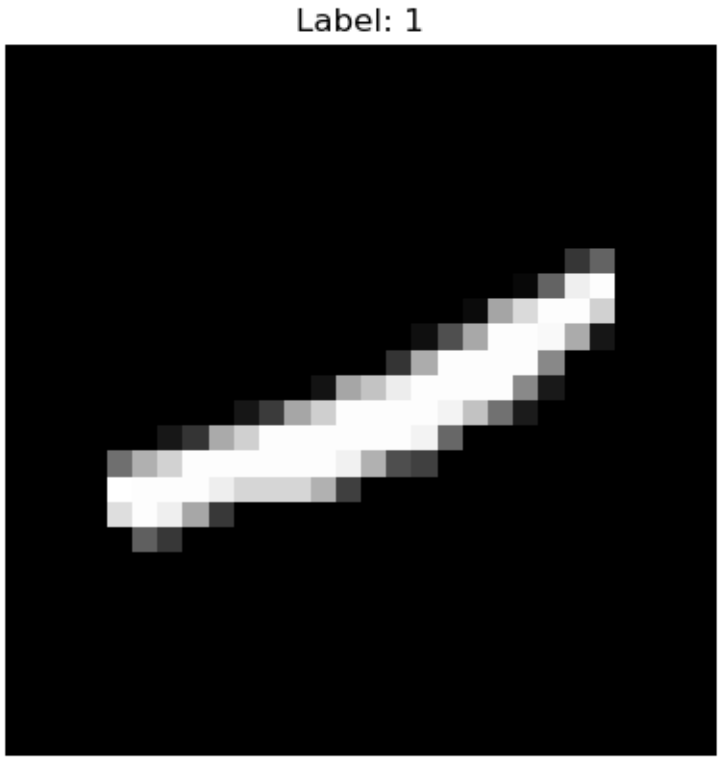
```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss')
plt.show()

# Predict and evaluate on the test set
test_predictions = predict(model, test_data_normalized)
print(f"Test Set Accuracy: {accuracy(test_labels.ravel(), test_predictions) *␣
↪100:.2f}%")
```



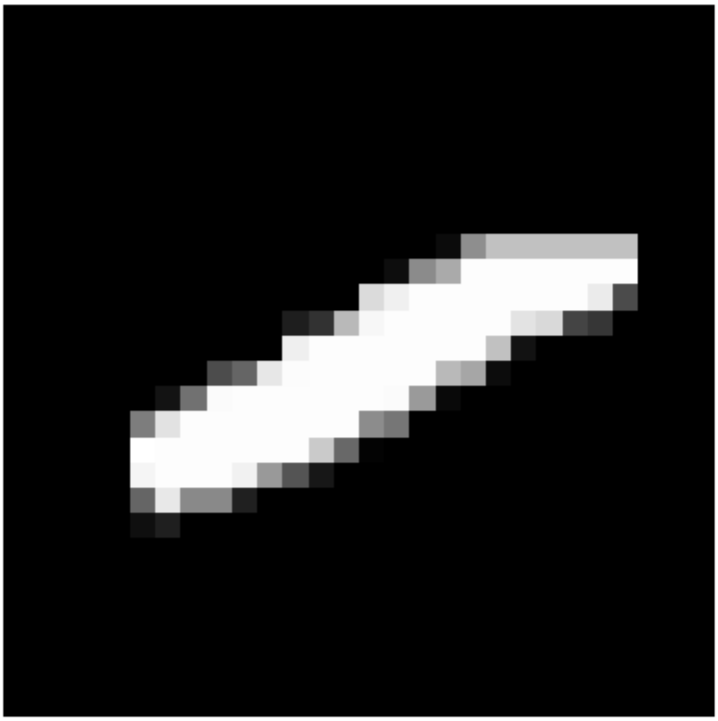

```
Test Set Accuracy: 9.90%
```

[10]:
```
def visualize_samples(data, labels, num_samples=5):
    indices = np.random.choice(data.shape[0], num_samples, replace=False)
    for index in indices:
        plt.imshow(data[index].reshape(28, 28), cmap='gray')
        plt.title(f"Label: {labels[index][0]}")
        plt.axis('off')
        plt.show()
```

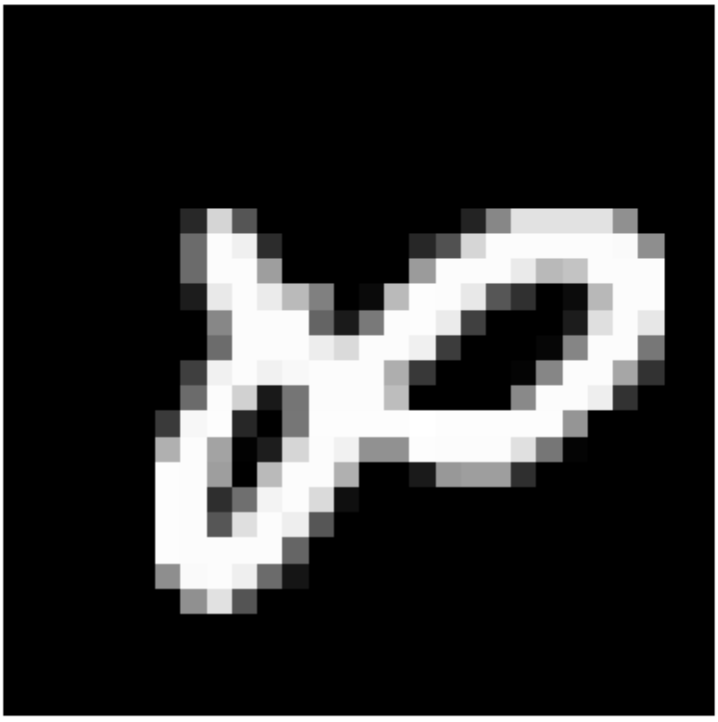
```
visualize_samples(train_data, train_labels)
```

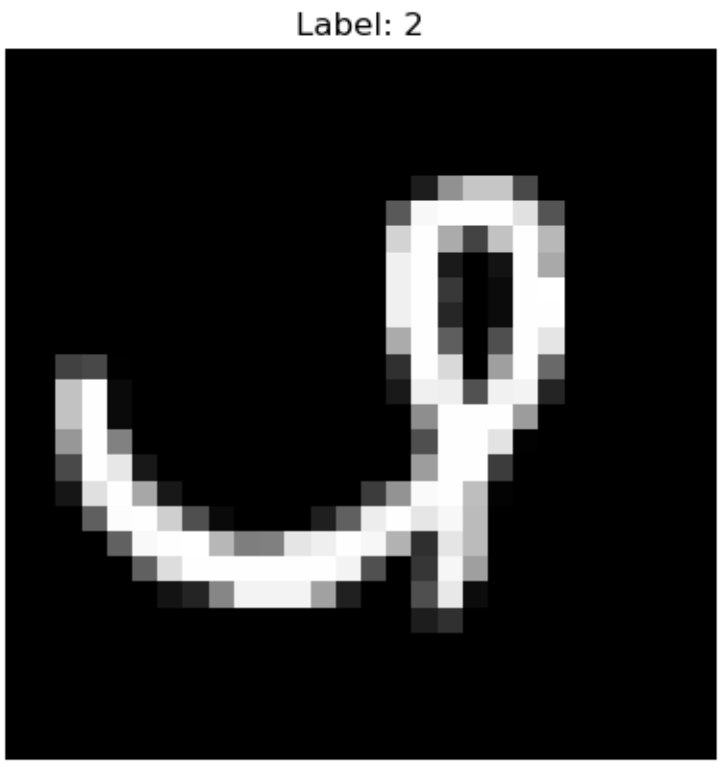


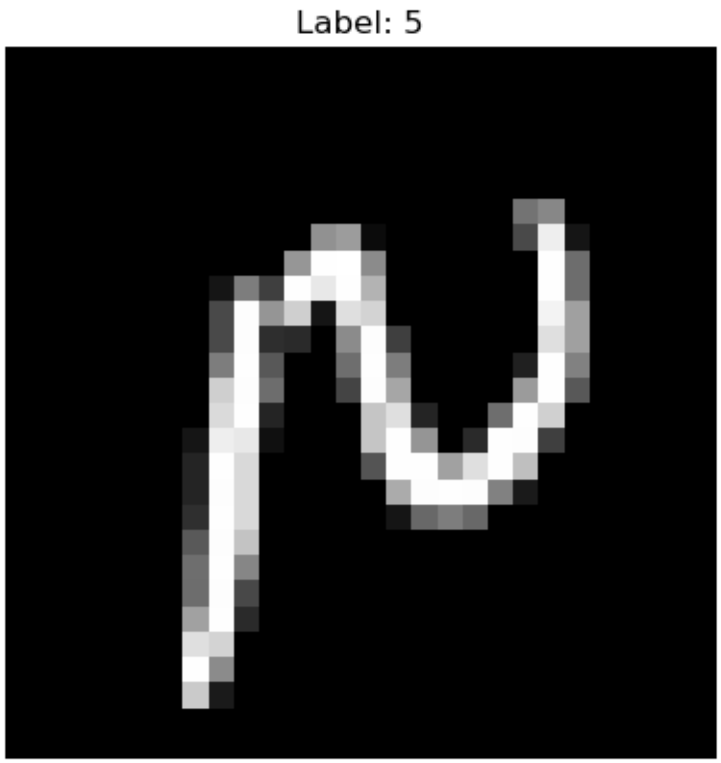
Label: 1



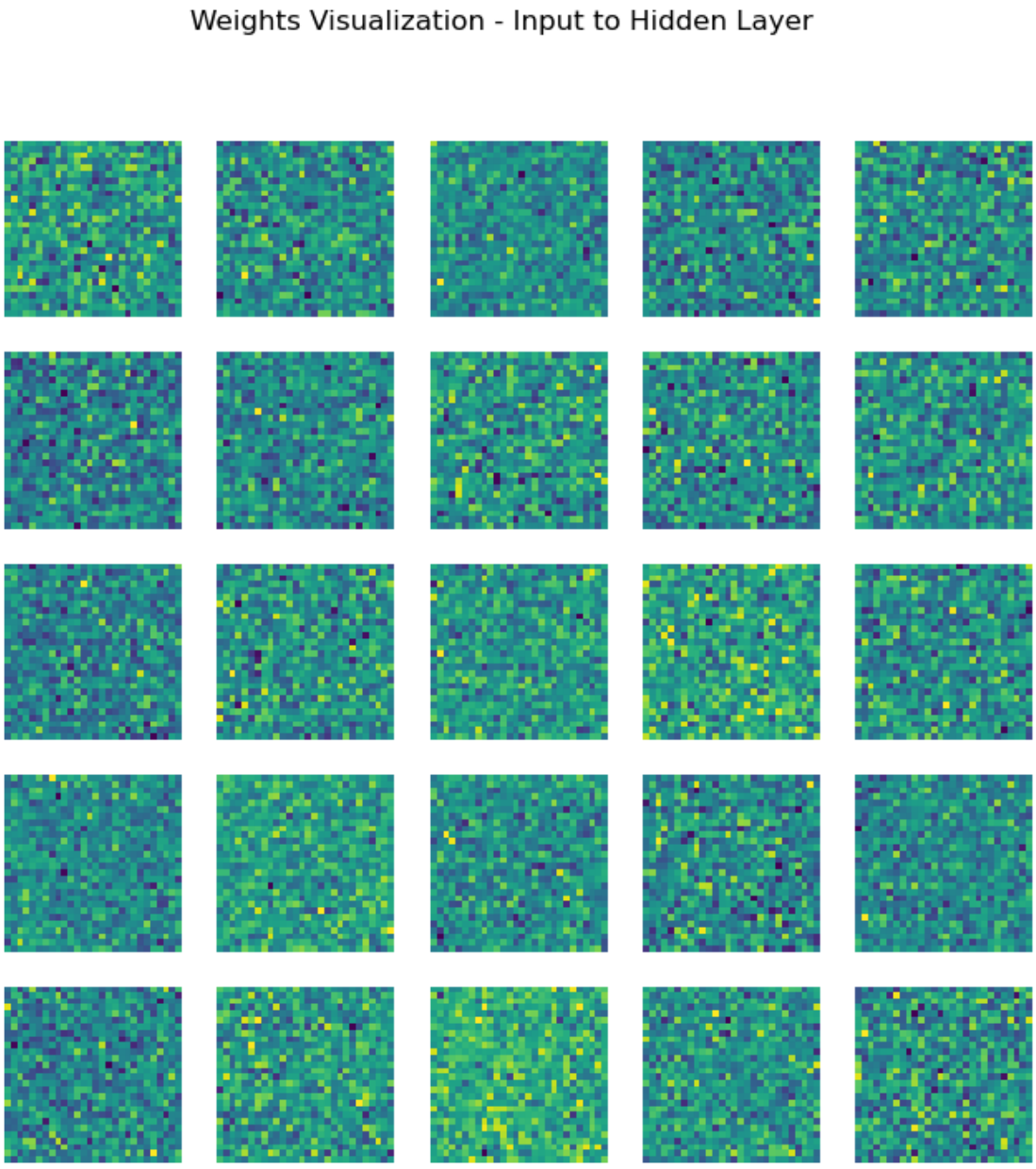
Label: 8



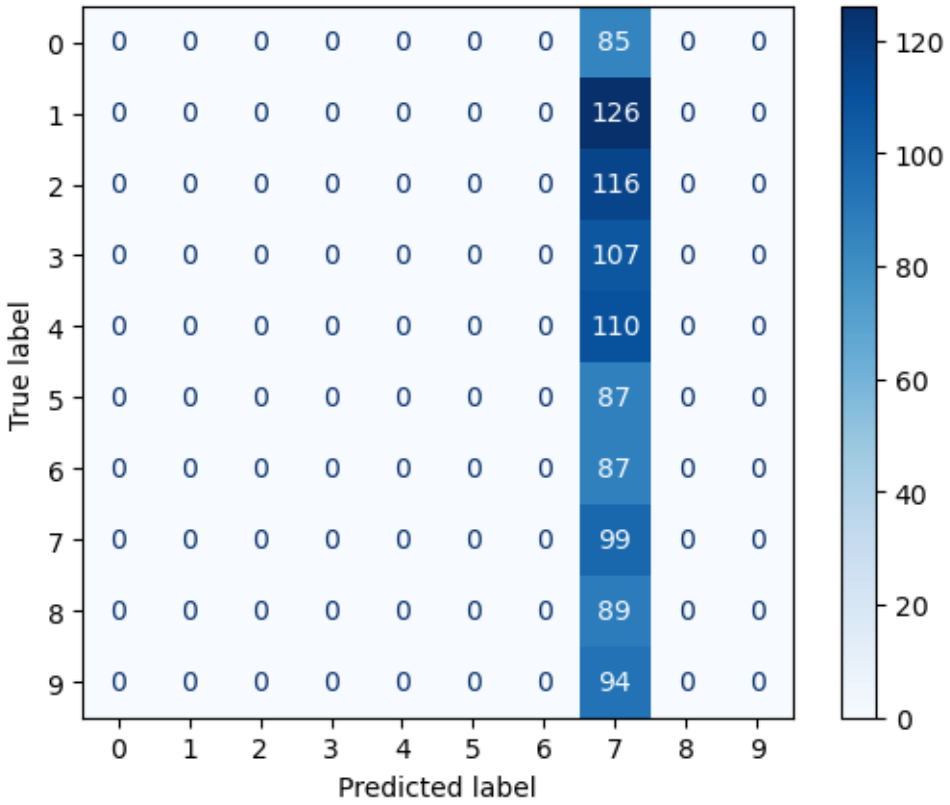

Label: 2



[11]:
```
def visualize_weights(weights, layer_name):
    fig, axes = plt.subplots(5, 5, figsize=(8, 8))
    for i, ax in enumerate(axes.ravel()):
        ax.imshow(weights[:, i].reshape(28, 28), cmap='viridis')
        ax.axis('off')
    plt.suptitle(f"Weights Visualization - {layer_name}")
    plt.show()

visualize_weights(model.weights_input_hidden, "Input to Hidden Layer")
```



[12]:
```
def plot_confusion_matrix(true_labels, predicted_labels):
    cm = confusion_matrix(true_labels, predicted_labels)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.
 ↪arange(10))
    disp.plot(cmap=plt.cm.Blues)

plot_confusion_matrix(test_labels.ravel(), test_predictions)
```



Conclusion:

Implementing a two-layer perceptron for digit recognition provided invaluable insights into neural network design, training, and evaluation. While this model offers a solid introduction, more advanced architectures and techniques can further enhance performance in real-world applications.

References:

1. Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." Nature 323.6088 (1986): 533-536.
2. LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.