Mini Project # 2 Perceptron Neural Network Handwriting Recognition

Due: Sep. 24/2025

James Levi

1. (a–c) Data Preparation

a. Download the data file 'digits.mat' from Canvas.

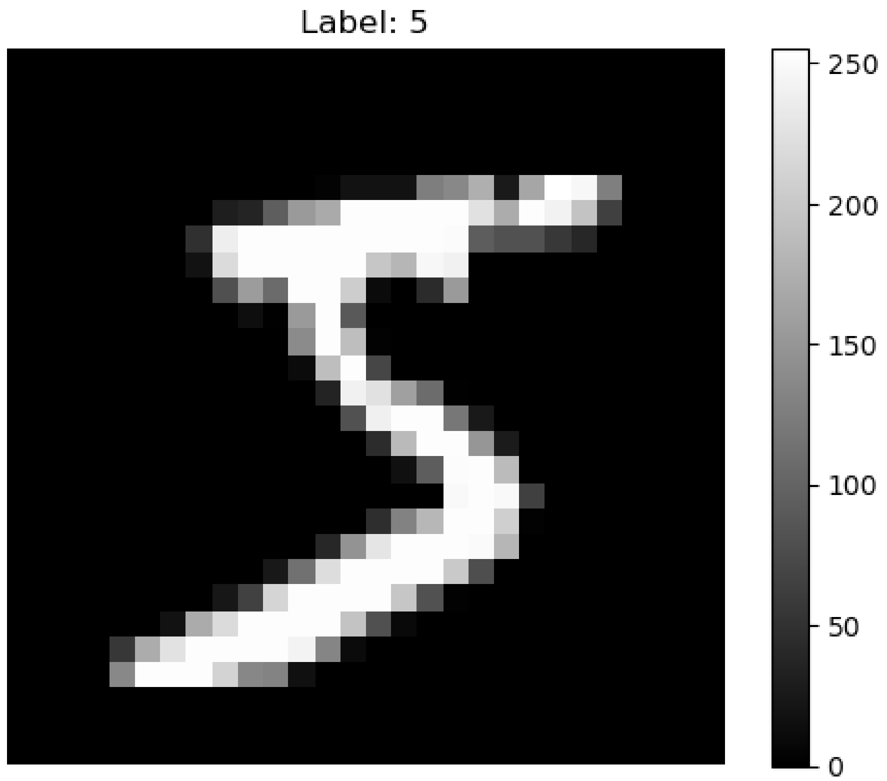
b. Load this data file to your program (in Matlab type: load digits.mat)

c. You will have four variables:

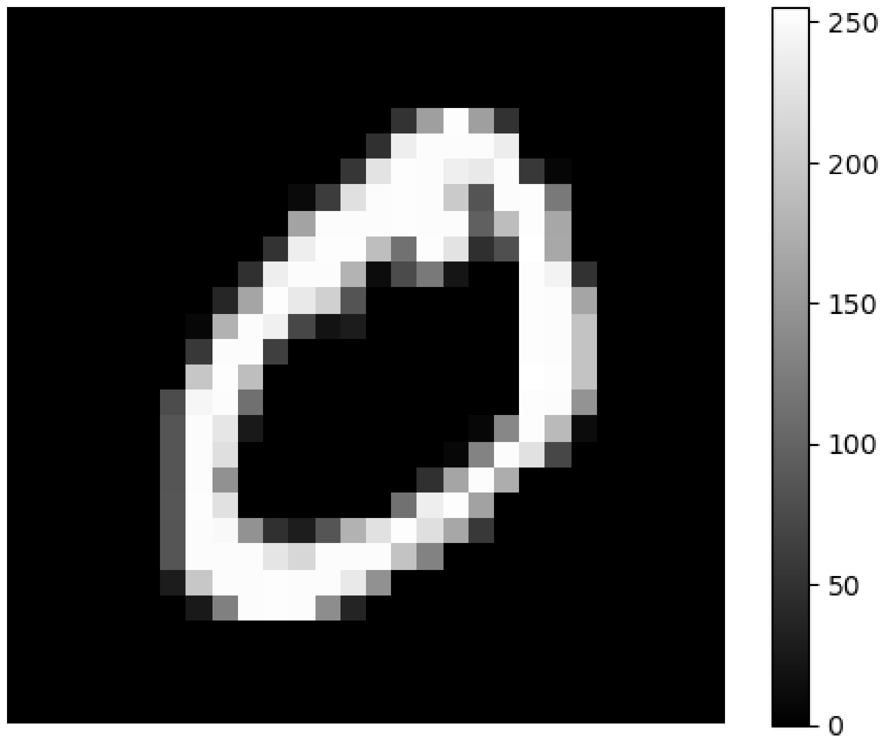
Name	Size	Bytes	Class
test	784x1000	784000	uint8
testlabels	1000x1	1000	uint8
train	784x5000	3920000	uint8
trainlabels	5000x1	5000	uint8

The variable 'train' includes images of 5000 hand-written digits. However, to see images of these digits you should change 'train' from a 784 x 5000 to a variable such as 'x' which has the dimension of 28 x 28 x 5000. Then you will have five thousand 28 x 28 images of these digits. A simple way of doing this is:

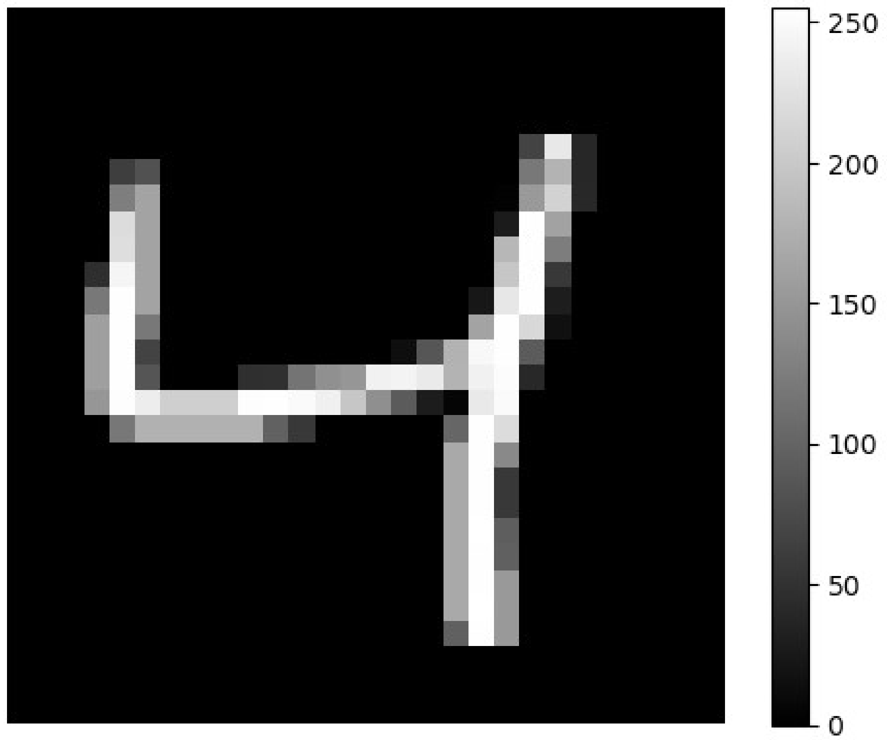
```
from scipy.io import loadmat
data= loadmat("digits.mat")


from scipy.io import loadmat
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

# Load the .mat file
data = loadmat("digits.mat")

train = data['train']
trainlabels = data['trainlabels']

x = train.reshape(28, 28, 5000, order='F')

# Loop through and display images
for k in range(10):  # change to 5000 if you want all
    plt.imshow(x[:, :, k], cmap='gray')
    plt.title(f"Label: {trainlabels[k][0]}")
    plt.axis('off')
    plt.colorbar()
    plt.show()
    plt.pause(0.2)
```



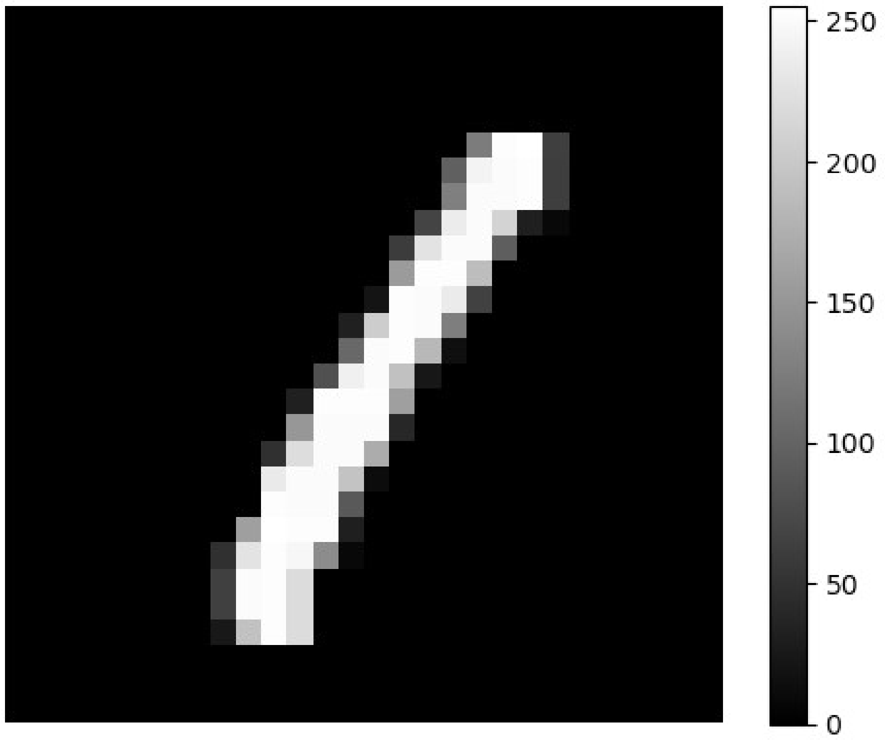
Label: 0



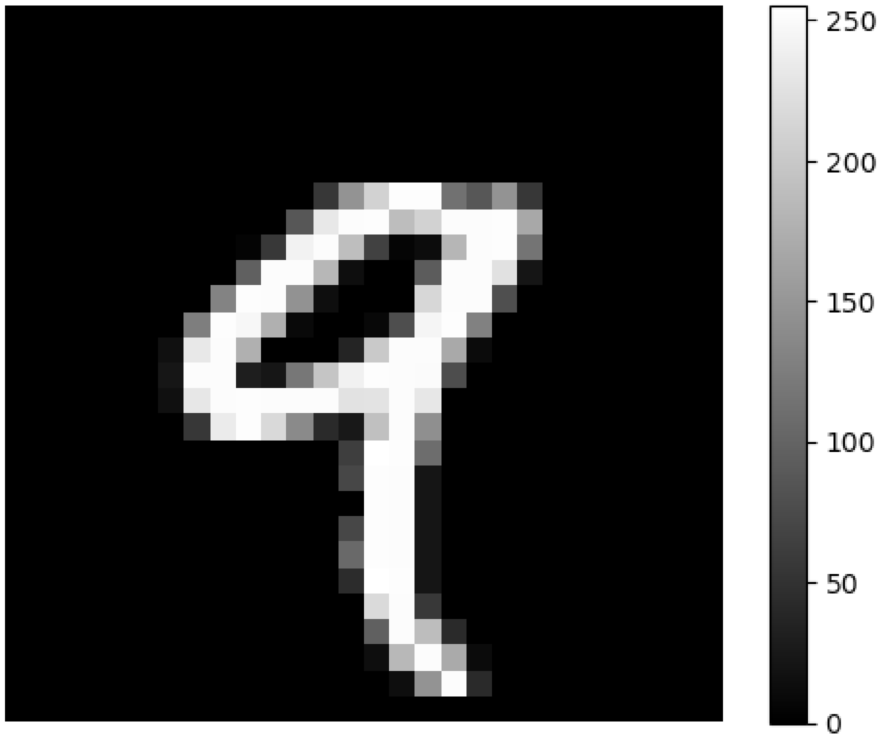
Label: 4



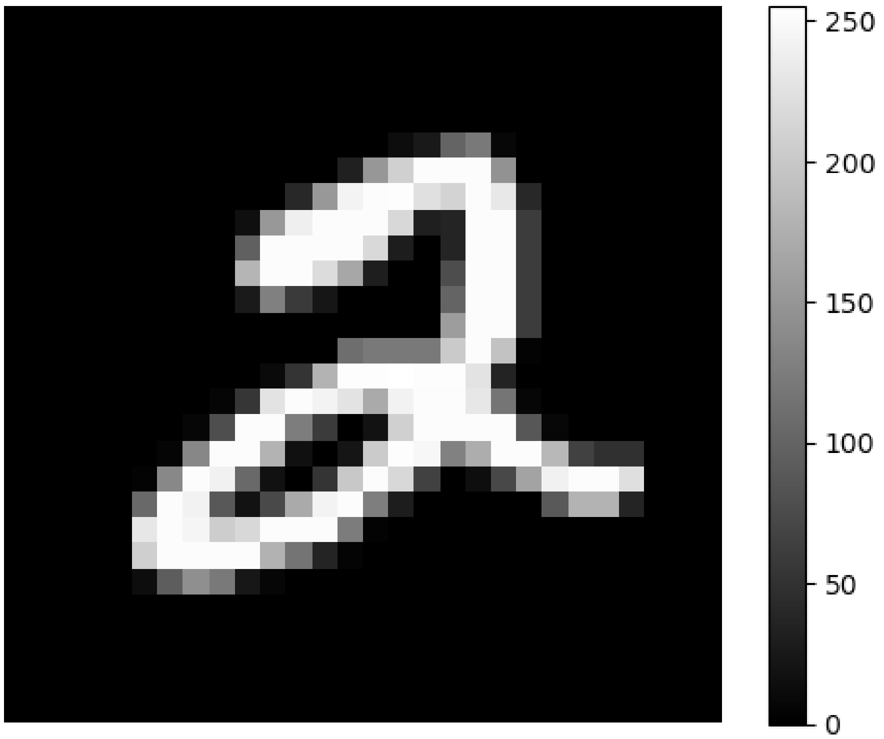
Label: 1



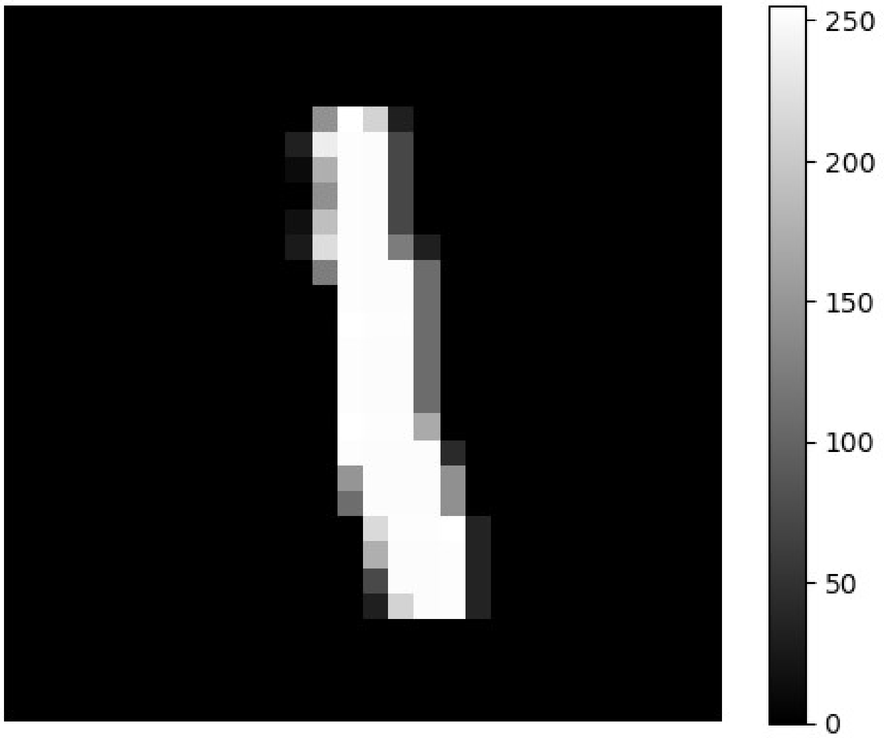
Label: 9



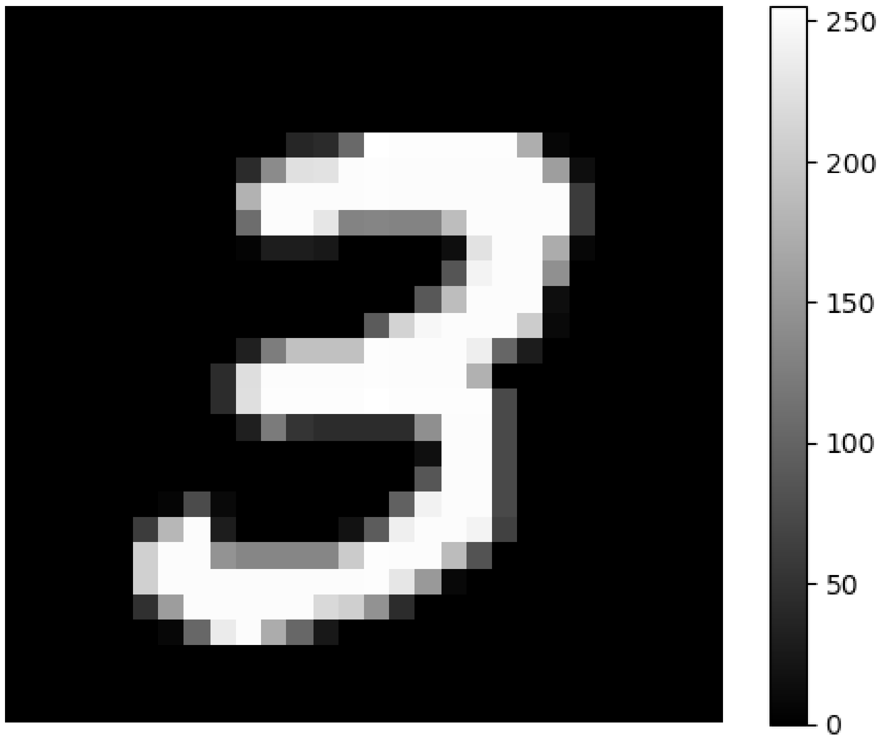
Label: 2



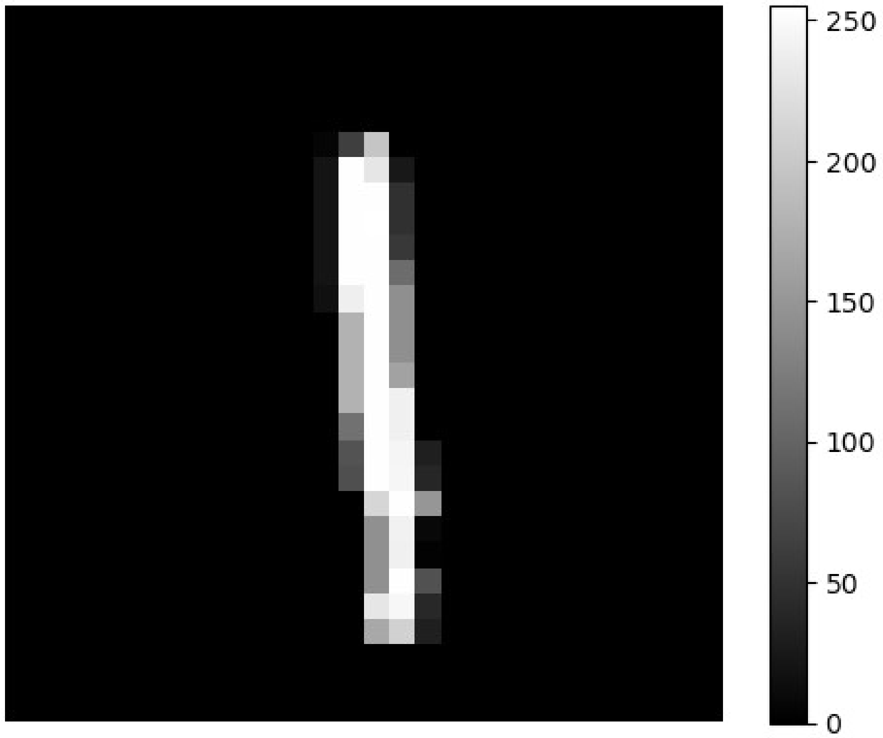
Label: 1

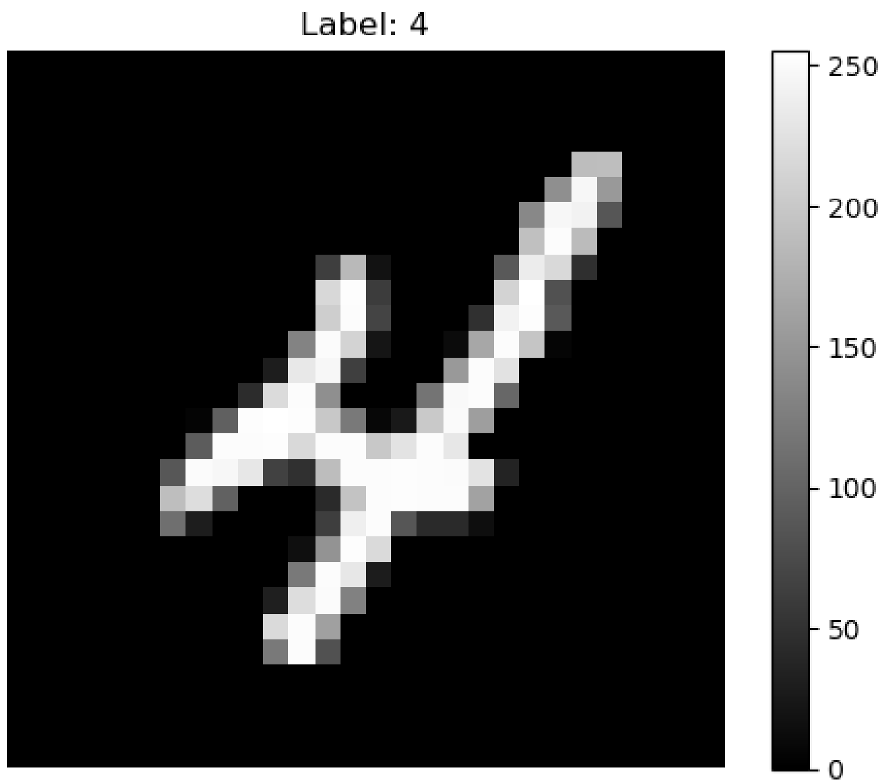


Label: 3



Label: 1





1(d) Detect Digit '0'

The number associated with each digit in 'train' is given in 'trainlabels'.
In this project, you should design and train a single-layer 2-dimensional Perceptron to recognize these digits.

Design and train a single-layer perceptron that can detect the digit '0'. In other words, your perceptron should return +1 when the digit in the image is '0' and -1 otherwise. Train your perceptron with 5000 images available in 'train' and test the performance of your system by using the other variable, 'test', in your data file. When you are training your perceptron, simultaneously, monitor the number of errors in the last 100 iterations. (Try to choose a reasonable value for your training rate and reduce it as a function of iteration number). How is the performance of the system? Also look at the image of your weights ‘w’ which is a 28 x 28 variable vector. How does it look? Why?

```
from math import sqrt

train = data['train'].astype(np.float32) / 255.0   # cast to 32 bit floats so that it is not stuck as integers, and we normalize since the values could have been from 0-255. Now it is from 0-1
train_labels = data['trainlabels'].ravel()          # flatten labels. now shape is (5000,)
train_bias = np.vstack([np.ones((1, train.shape[1])), train]) #adding a row of 1's for biases for w0.
```

```
test = data['test'].astype(np.float32) / 255.0   # cast to 32 bit
floats so that it is not stuck as integers, and we normalize since the
values could have been from 0-255. Now it is from 0-1
test_labels = data['testlabels'].ravel()          # flatten labels.
now shape is (5000,)
test_bias = np.vstack([np.ones((1, test.shape[1])), test]) #adding a
row of 1's for biases for w0.


class digit_detector:

    def __init__(self,digit):
        self.digit= digit
        self.w  = None # the size of w depends on the size of the data
which is passed in the train function

    #the perceptrons function f(x.T @ W)
    def f(self,x):
        return np.dot(x,self.w)

    #desired function
    def d(self,label):
        if (label == self.digit):
            return 1
        else:
            return -1

    def sgn(self,scalar):
        if (scalar > 0):
            return 1
        else:
            return -1

    #it will take the x with the bias in the train function. So x
inclulides the bias row
    def train(self,x,labels, eta, epochs):

        assert np.all(x[0] == 1), "Expected bias row of 1s in x"
        if self.w is None or self.w.shape[0] != x.shape[0]:
            self.w = np.zeros(x.shape[0], dtype=x.dtype)    #we do not
need to add an extra w for the bias one since the parameter x includes
the bias row. (remmeber the size of w is the the amount of rows of the
training data)


        for j in range (epochs):

            TP = 0.00 #true positive
            FN = 0.00 #flase negative
            TN = 0.00 #true negative
            FP = 0.00 #false positive

            count = 0

            N = x.shape[1]
            idx = np.arange(N)
            np.random.shuffle(idx)    #  shuffle

            for i in idx: #loop through the shuffled images, each of
the 5000 columns is one image
                x_column = x[:, i]       # i'th column
                f = self.f(x_column)
                y= self.sgn(f)
                d = self.d(labels[i])
                self.w = self.w + (0.5)*eta*(d - y) * x_column
                eta *= 0.999


                if (i >= N-100):
                    if (y==1 and d== 1):
                        TP += 1
                    elif (y==-1 and d==-1):
                        TN += 1
                    elif (y==-1 and d==1):
                        FN += 1
                    elif (y==1 and d == -1):
                        FP += 1

            accuracy = (TP + TN) / (TP + TN + FP + FN)
            precision = TP / (TP + FP)
            recall = TP / (TP + FN)

            print(
                "Metrics for final 100 of training data of Epoch",
j+1,
                "\nAccuracy:", accuracy,
                "\nPrecision:", precision,
                "\nRecall:", recall
            )



    def test(self,x,labels):

        N = x.shape[1]
        TP = 0.00 #true positive
        FN = 0.00 #flase negative
        TN = 0.00 #true negative
        FP = 0.00 #false positive
        for i in range (N):
            x_column = x[:, i]       # i'th column
            f = self.f(x_column)
            y= self.sgn(f)
            d = self.d(labels[i])

            if (i >= N-100):
                if (y==1 and d== 1):
                    TP += 1
                elif (y==-1 and d==-1):
                    TN += 1
                elif (y==-1 and d==1):
                    FN += 1
                elif (y==1 and d == -1):
                    FP += 1

        accuracy = (TP + TN) / (TP + TN + FP + FN)
        precision = TP / (TP + FP)
        recall = TP / (TP + FN)

        print(
            "\nMetrics for TEST data",
            "\nAccuracy:", accuracy,
            "\nPrecision:", precision,
            "\nRecall:", recall
            )

zero_detector = digit_detector(0)
zero_detector.train(train_bias,train_labels, 0.1, 1)
zero_detector.test(test_bias,test_labels)
```

```
Metrics for final 100 of training data of Epoch 1
Accuracy: 0.97
Precision: 0.8181818181818182
Recall: 0.9

Metrics for TEST data
Accuracy: 0.98
Precision: 0.75
Recall: 1.0
```

Comments

The performance of the system is quite impressive. We see how there is a 98% accuracy when detecting the digit 0. Since only approximately 10% of the data are labeled as 0's, accuracy may not be the best metric to consider, since if the perceptron outputs -1 for every image it sees, it will get approximately a 90% regardless. This is why additional metrics such as precision and recall are usefull. Precision showscases of all the digits that were predicted to be 0, how many of them were actually 0? This is usefull information to show the false alarms. Additionally, the well performed recal shows of all the real 0's in the dataset, how many did the model correctly find? This proves how the model did not simply mark them all at -1 in a biased approach, it infact caught all of the 0's.

1(e) Detect Digits '8', '1', and '2'

Repeat this experiment, but this time try to detect digit '8'. Does your network perform better or worse?
Why? Now try '1' and '2'.

```
eight_detector = digit_detector(8)
eight_detector.train(train_bias,train_labels, 0.1, 1)
eight_detector.test(test_bias,test_labels)
```

```
Metrics for final 100 of training data of Epoch 1
Accuracy: 0.93
Precision: 0.4444444444444444
Recall: 0.6666666666666666

Metrics for TEST data
Accuracy: 0.9
Precision: 0.3333333333333333
Recall: 0.42857142857142855
```

Comments on detecting 8

The model performs less well for the digit 8. We see how, compared to 0, there is a lower accuracy, precision, and recall. When looking at precision, of all the digits that the model predicted to be 8's, only a third of them were actually 8's. This is mostly because digits like 0,3,6,9 do have a fairly similar visual resemblance to 8's.

```
one_detector = digit_detector(1)
one_detector.train(train_bias,train_labels, 0.1, 1)
one_detector.test(test_bias,test_labels)
```

```
Metrics for final 100 of training data of Epoch 1
Accuracy: 0.98
Precision: 1.0
Recall: 0.9

Metrics for TEST data
Accuracy: 0.99
Precision: 1.0
Recall: 0.9375
```

```
two_detector = digit_detector(2)
two_detector.train(train_bias,train_labels, 0.1, 1)
two_detector.test(test_bias,test_labels)
```

```
Metrics for final 100 of training data of Epoch 1
Accuracy: 0.96
Precision: 0.8333333333333334
Recall: 0.625

Metrics for TEST data
Accuracy: 0.94
Precision: 0.9090909090909091
Recall: 0.6666666666666666
```

Comments on detecting 1's and 2's

The model performs very well for detecting 1's, and slightly less well for detecting 2's. We notice how due to the digit 1 having a simple visual apearance, it is detected fairly accurately. When looking at the precision, we see how in this run how it detected all of the 1's. For 2, the performance was slightly worse for the precison and recall, indicatling that there were more false alarms and quite many missed positives.

2. Two-Layer Perceptron with Backpropagation

Consider the following two-layer Perceptron. We have 784 input neurons, 25 hidden neurons, and 10 output neurons in this network. We want to use this network for digit recognition like what you did in problem 1. However, this time we have 10 output neurons. Neuron number i (i=0,1,2,...,9) should return 1 when the digit in the input pattern is 'i' and 0 otherwise. For example, if the input pattern represents number '2' the output vector should look like [0,0,1,0,0,0,0,0,0,0]. Therefore, in your project you prepare a computer code to train the network by adjusting weights for this task. The algorithm that we use in this project is BACK-PROPAGATION algorithm.

two layer

```
import numpy as np

def activation_function(z):
    return 1.0 / (1.0 + np.exp(-z)) #sigmoid

def one_hot_vector(label, K=10): #one hot for the output of the neural network
    d = np.zeros((K,1), dtype=np.float32)
    d[label, 0] = 1.0
    return d

class twolayer: #class for two layer perceptron neural network
```

```
    def __init__(self):
        self.W1 = None #W1 depends on how many hidden nodes there are,
and how many rows the data is (#rows=#pixels)
        self.W2 = None #W2 depends on how many output nodes, and how
many rows the data is.
        self.d  = None #the size of d depends on the amount of output
from the neural netork (10 in this example)

    def forward_pass_step(self, nodes, x_vec, step): #we generalize
the forward pass as it can be used from input to hidden layer, but
also hidden layer to output layer. This is what parameter 'step' is
used for
        if step == 1:
            if self.W1 is None:
                self.W1 = 0.01 * np.random.randn(nodes,
x_vec.shape[0])
            v = self.W1 @ x_vec
            y = activation_function(v)
            y_bias = np.vstack([[[1.0]], y]) #add bias for the second
layer to use
            return y, y_bias

        elif step == 2:
            #same as above, except W2 is being used, and bias does not
need to be added, since the next layer is the output
            if self.W2 is None:
                self.W2 = 0.01 * np.random.randn(nodes,
x_vec.shape[0])
            v = self.W2 @ x_vec
            y = activation_function(v)
            return y
        else:
            raise ValueError("step must be 1 or 2") #makes sure the
parameter step is 1 or 2

    def error_function(self, d, y):
        return (d - y) #the error function is the squared error, but
after proofs it ends up being d-y.

    def delta2(self, d, y):
        return (d - y) * (y * (1 - y))

    def delta1(self, d2, y_hidden_no_bias):
        W2_no_bias = self.W2[:, 1:]
        back = W2_no_bias.T @ d2
        return back * (y_hidden_no_bias * (1 - y_hidden_no_bias))

    def updateWeights(self, which_weight, eta, delta, old_input):
        if which_weight == 1:
            self.W1 = self.W1 + eta * (delta @ old_input.T)
        elif which_weight == 2:
            self.W2 = self.W2 + eta * (delta @ old_input.T)
        else:
            raise ValueError("which_weight must be 1 or 2")

    def train(self, hidden_nodes, output_nodes, x, labels, epochs,
eta):
        N = x.shape[1]
        for j in range(epochs):
            idx = np.arange(N)
            np.random.shuffle(idx)

            preds, true = [], []

            for i in idx:
                x_col = x[:, i:i+1]
                y1_nb, y1 = self.forward_pass_step(hidden_nodes,
x_col, step=1)
                y2 = self.forward_pass_step(output_nodes, y1, step=2)

                d = one_hot_vector(int(labels[i]), K=output_nodes)
                d2 = self.delta2(d, y2)
                d1 = self.delta1(d2, y1_nb)

                self.updateWeights(2, eta, d2, y1)
                self.updateWeights(1, eta, d1, x_col)

                eta *= 0.999999

                preds.append(int(np.argmax(y2)))
                true.append(int(labels[i]))

            # full epoch accuracy
            preds = np.array(preds)
            true = np.array(true)
            acc_epoch = np.mean(preds == true)

            print(f"Epoch {j+1}: accuracy = {acc_epoch:.4f}")
        print("Training Complete\n\n\n")


    def test(self, x, labels):
        N = x.shape[1]
        correct = 0
        for i in range(N):
            x_col = x[:, i:i+1]
            y1_nb, y1 = self.forward_pass_step(self.W1.shape[0],
x_col, step=1)
            y2 = self.forward_pass_step(self.W2.shape[0], y1, step=2)
            pred = int(np.argmax(y2))
            if pred == int(labels[i]):
                correct += 1
        acc = correct / N
        print(f"TEST accuracy for testing data: {acc:.4f}")
        return acc

net = twolayer()
net.train(hidden_nodes=25, output_nodes=10,
          x=train_bias, labels=train_labels, epochs=10, eta=0.2)

net.test(test_bias, test_labels)
```

```
Epoch 1: accuracy = 0.5362
Epoch 2: accuracy = 0.8930
Epoch 3: accuracy = 0.9206
Epoch 4: accuracy = 0.9322
Epoch 5: accuracy = 0.9446
Epoch 6: accuracy = 0.9508
Epoch 7: accuracy = 0.9608
Epoch 8: accuracy = 0.9628
Epoch 9: accuracy = 0.9680
Epoch 10: accuracy = 0.9722
Training Complete



TEST accuracy for testing data: 0.9040

0.904
```