Problem of the Week 7

Name: Ryan Rajaie

Student ID: 016105024

Date: 10/5/2025

Motivation

This week's Problem of the Week is focused on comparing and contrasting the single- and multi-layer perceptrons.

Completing this assignment

Fill in the coding and markdown cells as indicated below. When you are happy with you responses, restart the kernel and run all cells to produce a clean rendition of your notebook. Then, create a PDF of your notebook and upload both documents (PDF and ipynb file) to Canvas.

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

Problem 7.1

In class, we created two classes - one for the single layer perceptron and one for the multi layer perceptron. Copy those classes over here.

```
# copy the single layer perceptron code here
class SingleLayerPerceptron:

    def __init__(self, num_features, num_classes, random_seed=123):

        self.num_classes = num_classes
        generator = np.random.RandomState(random_seed)
        self.weight_out = generator.normal(loc=0.0, scale=0.1,
size=(num_classes, num_features))
        self.bias_out = np.zeros(num_classes)

    def int_to_onehot(self, y, num_labels):
        ary = np.zeros((y.shape[0], num_labels))
        for i, val in enumerate(y):
            ary[i, val] = 1
        return(ary)

    def sigmoid(self,z):
```

```
        return (1/(1+np.exp(-z)))

    # EDIT: define the forward function
    def forward(self, x):
        z = np.dot(x, self.weight_out.T)+self.bias_out
        p = self.sigmoid(z)
        return(p)


    def predict(self, X):
        probs = self.forward(X)
        return np.argmax(probs, axis=1)

    # EDIT: define the backward function
    def backward(self, x, p, y):
        y_onehot = self.int_to_onehot(y, self.num_classes)

        d_L__d_p = -2*(y_onehot - p)/y.shape[0]

        d_p__d_z = p*(1-p)

        D = d_L__d_p * d_p__d_z

        d_L__d_w = np.dot(D.T, x)

        d_L__d_b = np.dot(D.T, np.ones((x.shape[0],)))

        return(d_L__d_w, d_L__d_b)


    def mse_loss(targets, probs, num_labels=10):
        onehot_targets = int_to_onehot(targets, num_labels)
        err = np.mean((onehot_targets - probs)**2)
        return(err)

    def compute_mse_and_acc(self, X, y, num_labels=10):
        probs = self.forward(X)
        predicted_labels = np.argmax(probs,axis=1)
        onehot_targets =self.int_to_onehot(y, num_labels)
        loss = np.mean((onehot_targets - probs)**2)
        acc = np.sum(predicted_labels == y)/len(y)
        return(loss, acc)

    def train(self,X_train, Y_train, X_test, Y_test, num_epochs,
learning_rate=0.1):
        train_losses = []
        test_losses = []
        train_accs = []
        test_accs = []
        for e in range(num_epochs):
            a_out = self.forward(X_train)
```

```
            # EDIT: comput the gradients and update the weights and
biases
            d_L__d_w, d_L__d_b = self.backward(X_train, a_out,
Y_train)
            self.weight_out -= d_L__d_w * learning_rate
            self.bias_out -= d_L__d_b * learning_rate

            train_loss, train_acc = self.compute_mse_and_acc(X_train,
Y_train)
            train_losses.append(train_loss)
            train_accs.append(train_acc)

            test_loss, test_acc = self.compute_mse_and_acc(X_test,
Y_test)
            test_losses.append(test_loss)
            test_accs.append(test_acc)
        return(train_losses, train_accs, test_losses, test_accs)

# copy the multilayer perceptron code here
class MultiLayerPerceptron:

    def __init__(self, num_features, num_hidden, num_classes,
random_seed=123):

        self.num_classes = num_classes

        rng = np.random.RandomState(random_seed)

        # first layer weights and biases
        self.weight_h = rng.normal(loc=0.0, scale=0.1,
size=(num_hidden, num_features))
        self.bias_h = np.zeros(num_hidden)

        # second layer weights and biases
        self.weight_out = rng.normal(loc=0.0, scale=0.1,
size=(num_classes, num_hidden))
        self.bias_out = np.zeros(num_classes)

    def int_to_onehot(self, y, num_labels):
        ary = np.zeros((y.shape[0], num_labels))
        for i, val in enumerate(y):
            ary[i, val] = 1
        return(ary)

    def sigmoid(self,z):
        return (1/(1+np.exp(-z)))

    def forward(self, x):
        # enter code here to implement the forward method
        zeta = np.dot(x, self.weight_h.T) + self.bias_h
```

```
        h = self.sigmoid(zeta)

        z = np.dot(h, self.weight_out.T) + self.bias_out
        p = self.sigmoid(z)

        return(h, p)

    def predict(self, X):
        _, probs = self.forward(X)
        return np.argmax(probs, axis=1)

    def backward(self, x, h, p, y):

        # encode the labels with one-hot encoding
        y_onehot = self.int_to_onehot(y, self.num_classes)

        # enter code here to implement the backward method
        D2 = (-2 / y.shape[0] * (y_onehot - p) * p * (1 - p))
        d_L__d_w2 = np.dot(D2.T, h)
        d_L__d_b2 = np.dot(D2.T, np.ones_like(y))

        D1 = np.dot(D2, self.weight_out) * h * (1 - h)
        d_L__d_w1 = np.dot(D1.T, x)
        d_L__d_b1 = np.dot(D1.T, np.ones_like(y))

        return(d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1)

    def mse_loss(targets, probs, num_labels=10):
        onehot_targets = int_to_onehot(targets, num_labels)
        err = np.mean((onehot_targets - probs)**2)
        return(err)

    def compute_mse_and_acc(self, X, y, num_labels=10):
        _, probs = self.forward(X)
        predicted_labels = np.argmax(probs,axis=1)
        onehot_targets =self.int_to_onehot(y, num_labels)
        loss = np.mean((onehot_targets - probs)**2)
        acc = np.sum(predicted_labels == y)/len(y)
        return(loss, acc)

    def train(self,X_train, Y_train, X_test, Y_test, num_epochs,
learning_rate=0.1):
        train_losses = []
        test_losses = []
        train_accs = []
        test_accs = []
        for e in range(num_epochs):

            # compute the forward method
            h, p = self.forward(X_train)
```

```
            # compute the backward method
            d_L__d_w2, d_L__d_b2, d_L__d_w1, d_L__d_b1 =
self.backward(X_train, h, p, Y_train)

            # update the weights and the biases
            self.weight_out -= learning_rate * d_L__d_w2
            self.bias_out   -= learning_rate * d_L__d_b2
            self.weight_h   -= learning_rate * d_L__d_w1
            self.bias_h     -= learning_rate * d_L__d_b1

            train_loss, train_acc = self.compute_mse_and_acc(X_train,
Y_train)
            train_losses.append(train_loss)
            train_accs.append(train_acc)

            test_loss, test_acc = self.compute_mse_and_acc(X_test,
Y_test)
            test_losses.append(test_loss)
            test_accs.append(test_acc)
        return(train_losses, train_accs, test_losses, test_accs)
```

Problem 7.2

In class, we introduced the MNIST hand-written image database. Read in the data here and prepare it for use in your Perceptron code.

```
def read_mnist_images(subset='train'):
    if subset=='train':
        prefix = 'train-'
    else:
        prefix = 't10k-'

    with open(os.path.join('MNIST',prefix+'images.idx3-ubyte'), 'rb')
as f:
        # unpack header
        _, num_images, num_rows, num_cols = struct.unpack('>IIII',
f.read(16))

        # read image data
        image_data = f.read(num_images * num_rows * num_cols)
        images = np.frombuffer(image_data, dtype=np.uint8)
        images = images.reshape(num_images, num_rows, num_cols)

    with open(os.path.join('MNIST',prefix+'labels.idx1-ubyte'), 'rb')
as f:
        # unpack header
        _, num_labels = struct.unpack('>II', f.read(8))
```

```
        # read label data
        labels = np.frombuffer(f.read(), dtype=np.uint8)


    return images, labels
```

Problem 7.3

Generate models for each of the perceptrons, and train each model with an equivalent learning rate and number of iterations.

```
# port in data
train_images, train_labels = read_mnist_images(subset='train')
test_images, test_labels = read_mnist_images(subset='test')

X_train = train_images.reshape(-1, 784) / 255.0
X_test = test_images.reshape(-1, 784) / 255.0
Y_train = train_labels
Y_test = test_labels

# create an slp object and train it
slp = SingleLayerPerceptron(784, 10)
train_losses_slp, train_accs_slp, test_losses_slp, test_accs_slp =
slp.train(X_train, Y_train, X_test, Y_test, 1000, learning_rate = 0.5)

# create an mlp object and train it
mlp = MultiLayerPerceptron(784, 50, 10)
train_losses_mlp, train_accs_mlp, test_losses_mlp, test_accs_mlp =
mlp.train(X_train, Y_train, X_test, Y_test, 1000, learning_rate = 0.5)
```

Problem 7.4

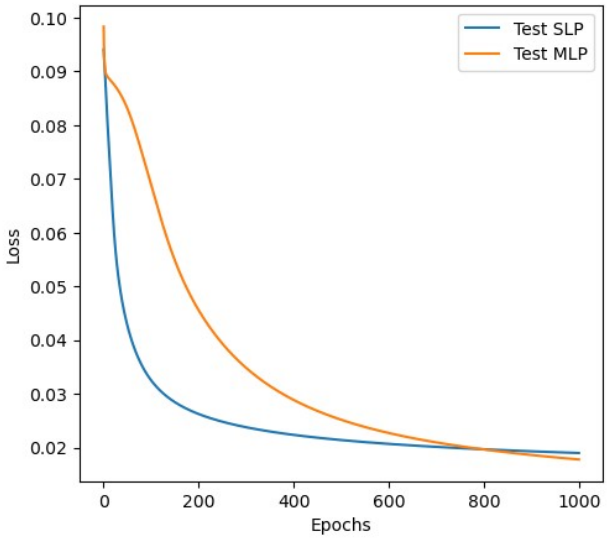
Generate two plots to compare and contrast the losses and accuracies of the perceptrons. The left plot should show two lines - one for the test losses of the single layer perceptron (blue) and one for the multi layer perceptron (orange). The right plot should be identical, but it should show the accuracies. Be sure to add a legend to your plot and label your axes.

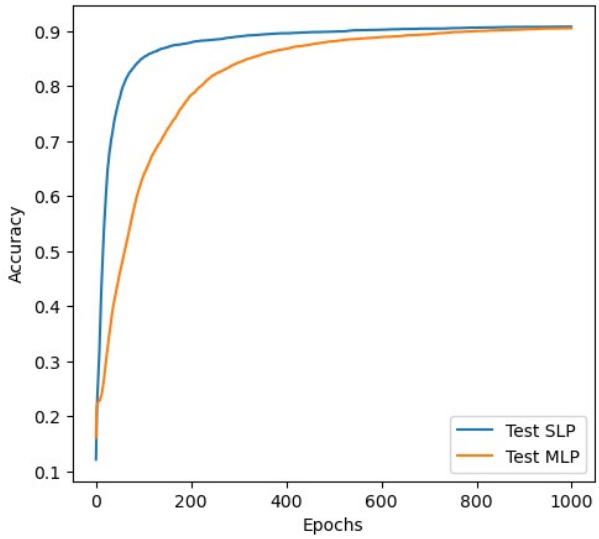
```
# create your plots here
plt.figure(figsize=(12,5))

plt.subplot(1,2,1)
plt.plot(test_losses_slp, label='Test SLP')
plt.plot(test_losses_mlp, label='Test MLP')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend()

plt.subplot(1,2,2)
plt.plot(test_accs_slp, label='Test SLP')
```

```
plt.plot(test_accs_mlp, label='Test MLP')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```





Problem 7.5

Based on the plot you've made above, answer the following questions:

1. What evidence do you see in yout plots to convince you that your model has converged?
2. Which model converged quicker?
3. Which model is more accurate? Explain why this model is more accurate (3 sentences max).

1. The plots have a pretty clean exponential drop-off in the amount of loss being lost. They are gaining less and less accuracy with each iteration.
2. The SLP converged quicker, it began to curb around 100 iterations, versus the MLP where it curbed at about 300 or so.
3. It depends, at around 1000 iterations the MLP begins to edge out to the SLP. Before that, however, the SLP is more accurate. MLP models are far more complex with far more variables and weights so it takes a longer time to train them, which is why. I expect, however, based on the current accuracy and loss trajectories that the MLP will eventually be noticeably more accurate than the SLP.