HW #1: Perceptrons

This experiment uses a neural network of ten perceptrons encapsulated in a Perceptron class object. The perceptrons themselves exist in a weight matrix property of the Perceptron class that is returned as a Numpy array from the train() method. There are two layers in this network. An input layer represented as x^K of 784 inputs, plus one for the bias. And an output layer represented as y^K. There are no hidden layers in this experiment. We ran the experiment with learning rates, represented as η, of 0.00001, 0.001, and 0.1.

x^K	The input layer vector of 784 (+1 bias) pixels of a single image
y^K	The output (activation) layer vector of 10 nodes representing activation level of a digit
t^K	The output layer vector of 10 nodes from the labeled training data
w_i	The weight element at index i of 785 weights
Δw_i	The gradient or vector derivative in order to minimize the cost function

The Perceptron Learning Algorithm was implemented by forwarding an activation vector of 10 elements to the output layer in the forward() method.

$$y^K = a(w \cdot x)$$

After forwarding the activation layer, we updated weights through back propagation using the following formula in the back() method:

$$\Delta w_i = \eta(t^K - y^K)x_i{}^K$$
$$w_i \longleftarrow w_i + \Delta w_i$$

Results

In [1]:
```
import perceptron as pt
```

First we instantiate our Perceptron object (or rather network of 10 perceptrons). We set the bias, the number of epochs and the sizes of our input and output layers.

The Perceptron constructor optionally takes in the training and test file names and loads them into Numpy arrays.

In [2]:
```
train_file = 'mnist_train.csv'
test_file = 'mnist_validation.csv'

bias = 1
epochs = 50

p = pt.Perceptron(sizes=[785, 10], train_filename=train_file, test_filename=test_file, bias=bias)
```

```
Loading Data: mnist_train.csv
Loading Data: mnist_validation.csv
```

Train with a rate of 0.00001

In [ ]:
```
rate = 0.00001
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```

Train with a rate of 0.001

In [ ]:
```
rate = 0.001
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```

Train with a rate of 0.001

In [ ]:
```
rate = 0.1
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```

Summary

The digits having the most issues being classified were 8, 5, and 2, all being mistaken for a 3. The rest of the digits did quite well, especially 0, 1, 5, and 7. That plots and confusions matrices do not show any significant differrences between our three learning rates.

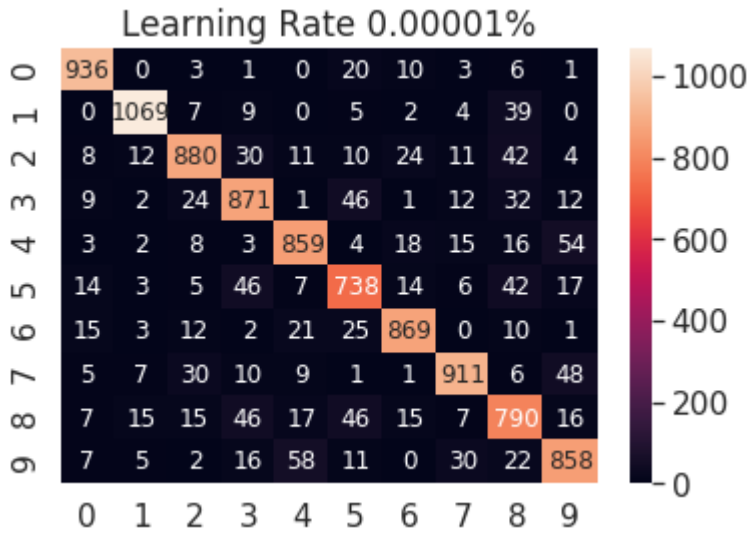
I noticed with all them that starting with the Epoch 0 the accuracy was very low (as expected). It was less than 10%. At Epoch 1, the accuracy jumps to ~85% and then over the next 49 epochs we only get an improvement of 3%. I am confused by this. Also, I don't see any oscillations here, so it appears that I am overfitting.

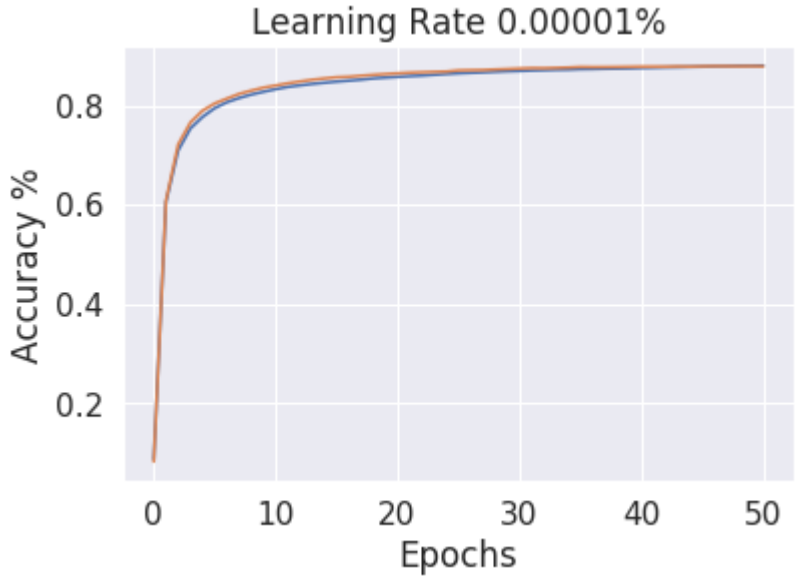
Summary

The digits having the most issues being classified were 8, 5, and 2, all being mistaken for a 3. The rest of the digits did quite well, especially 0, 1, 5, and 7. That plots and confusions matrices do not show any significant differrences between our three learning rates.

I noticed with all them that starting with the Epoch 0 the accuracy was very low (as expected). It was less than 10%. At Epoch 1, the accuracy jumps to ~85% and then over the next 49 epochs we only get an improvement of 3%. I am confused by this. Also, I don't see any oscillations here, so it appears that I am overfitting.

In [3]:
```
rate = 0.00001
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```

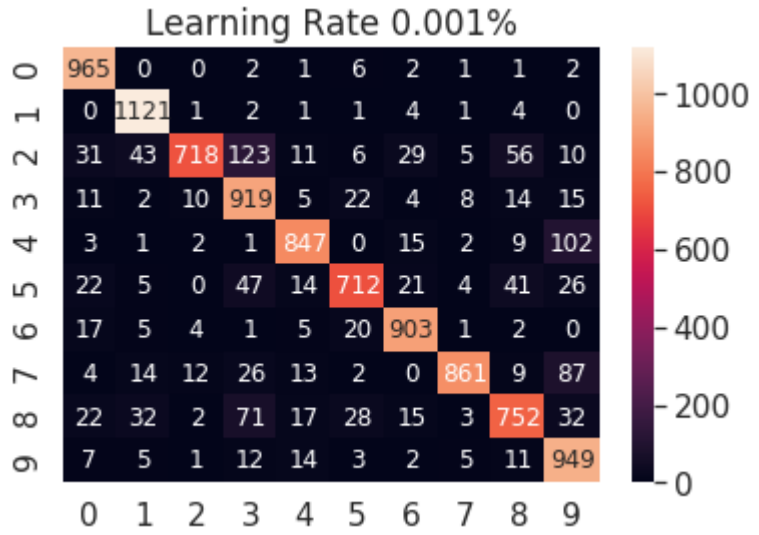


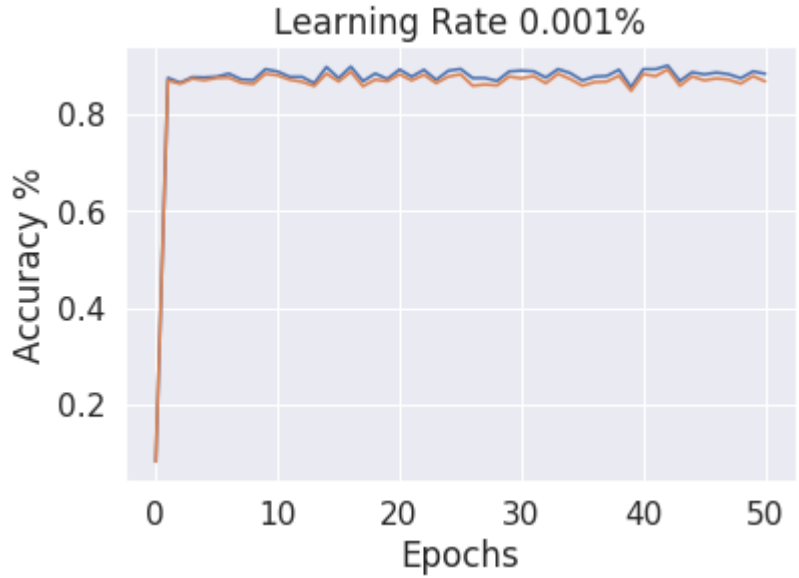



```
Test Accuracy: 87.8%
```

Train with a rate of 0.001

In [4]:
```
rate = 0.001
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```

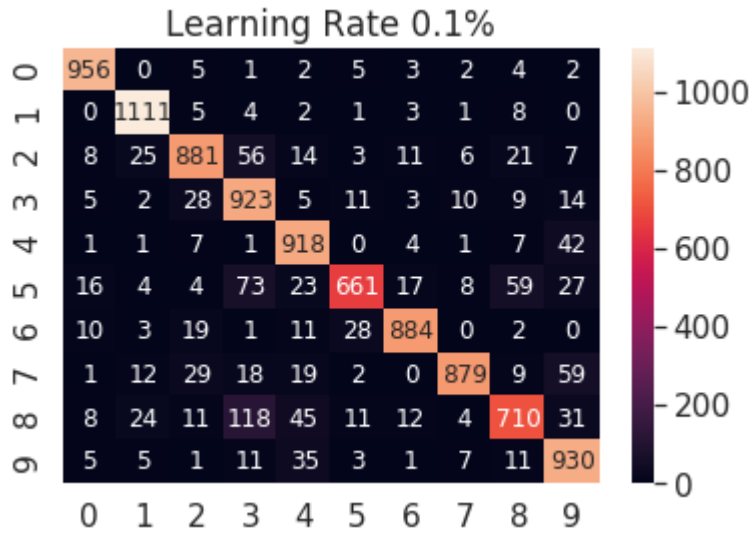





```
Test Accuracy: 87.5%
```

Train with a rate of 0.001

In [5]:
```
rate = 0.1
model, accuracy = p.train(rate=rate, epochs=epochs)
assert(model.shape == (785, 10))
assert(accuracy > .80)
```




```
Test Accuracy: 88.5%
```

Summary

The digits having the most issues being classified were 8, 5, and 2, all being mistaken for a 3. The rest of the digits did quite well, especially 0, 1, 5, and 7. That plots and confusions matrices do not show any significant differrences between our three learning rates.

I noticed with all them that starting with the Epoch 0 the accuracy was very low (as expected). It was less than 10%. At Epoch 1, the accuracy jumps to ~85% and then over the next 49 epochs we only get an improvement of 3%. I am confused by this. Also, I don't see any oscillations here, so it appears that I am overfitting.