PROJECT – 2 REPORT

TEAM MEMBERS: -
SHIVANGI TRIPATHI
PRATIK SANGHVI

Course Number: -
CSE574

In this assignment, we implement a Multilayer Perceptron Neural Network on MNIST (Modified National Institute of Standards and Technology) handwritten digit dataset and evaluate its performance in classifying the handwritten digits. We then use the same Neural Network on a Face dataset and evaluate its performance against a deep neural network and a convolutional neural network.

Building a Neural Network in nnScript.py:

Pre-Processing:

We load the mnist_all.mat file as a dictionary. We then separate the train and test keys into 2 arrays. The indices of training data is then shuffled using np.random.permutation and we then split the data with the first 50000 as train data and the next 10000 as validation data. The shuffled indices are also used to split and create the train and validation labels.

Feature Selection:

There are 784 features. We want to check if there are any features that have the same value for all the examples, if yes, then they don't add any value and should be removed. We use np.all and remove 67 such features. After pre-processing, we have 717 features.

Finally, in order to normalize the data, we divide by 255.

Initializing Weights:

Based on the input and output layer nodes, InitializeWeights function returns random weights. The output is a vector of weights. The outputs for w1 and w2 are then flattened and concatenated into a single vector to get our initial weights for the input layer and the hidden layer.

NN Objective Function:

For our neural network, we need to compute the objective function (negative log likelihood of error function with regularization). We learn the model parameters in a 2 step process. We initialize

weights to random numbers, compute output in a Feedforward step, compute error in our prediction, transmit the error backwards in backpropagation step and update weights.

Feedforward Propagation:

In Feedforward, we take a linear combination of input feature vector and the weight vector w1. This then goes into an activation function, in our case sigmoid to get a probability value for that node. This is done for all the nodes of the hidden layer. The output from nodes of hidden layer then becomes the input for the output layer. We then take a linear combination of input to output layer and weight vector w2. This then goes into another activation function (sigmoid). This output is then used in the back propagation to compute error.

Error Function & Backpropagation:

Error function is the negative log likelihood error function given by:

$$J(W^{(1)}, W^{(2)}) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{l=1}^{k}(y_{il}\ln o_{il} + (1-y_{il})\ln(1-o_{il}))$$

To avoid overfitting, we add a regularization term, so the error function now becomes:

$$\tilde{J}(W^{(1)}, W^{(2)}) = J(W^{(1)}, W^{(2)}) + \frac{\lambda}{2n}\left(\sum_{j=1}^{m}\sum_{p=1}^{d+1}(w_{jp}^{(1)})^2 + \sum_{l=1}^{k}\sum_{j=1}^{m+1}(w_{lj}^{(2)})^2\right)$$

Next step is Backpropagation. For this, we compute the derivative of error function with respect to weights. This value is then used to update weights. The new weight vector is again fed to the Feedforward propagation step.

$$\begin{aligned}\frac{\partial J_i}{\partial w_{lj}^{(2)}} &= \frac{\partial J_i}{\partial o_l}\frac{\partial o_l}{\partial b_l}\frac{\partial b_l}{\partial w_{lj}^{(2)}} \\ &= \delta_l z_j\end{aligned}$$

Above image represents the derivative of error function with respect to weight from hidden unit j to output unit l.

$$\frac{\partial J_i}{\partial w_{jp}^{(1)}} = \sum_{l=1}^{k}\frac{\partial J_i}{\partial o_l}\frac{\partial o_l}{\partial b_l}\frac{\partial b_l}{\partial z_j}\frac{\partial z_j}{\partial a_j}\frac{\partial a_j}{\partial w_{jp}^{(1)}}$$

Above image represents the derivative of error function with respect to weight from input unit p to hidden unit j.

Now, in order to get optimized values for weights, we use scipy.optimize.minimize. This function takes in the initial weight vector, the objection function (nnObjFunction in our case), input arguments (number of input units, number of hidden units, training data, training labels and regularization parameter).

We use conjugate gradient descent which uses a more sophisticated learning rate in each iteration, so that convergence is faster.

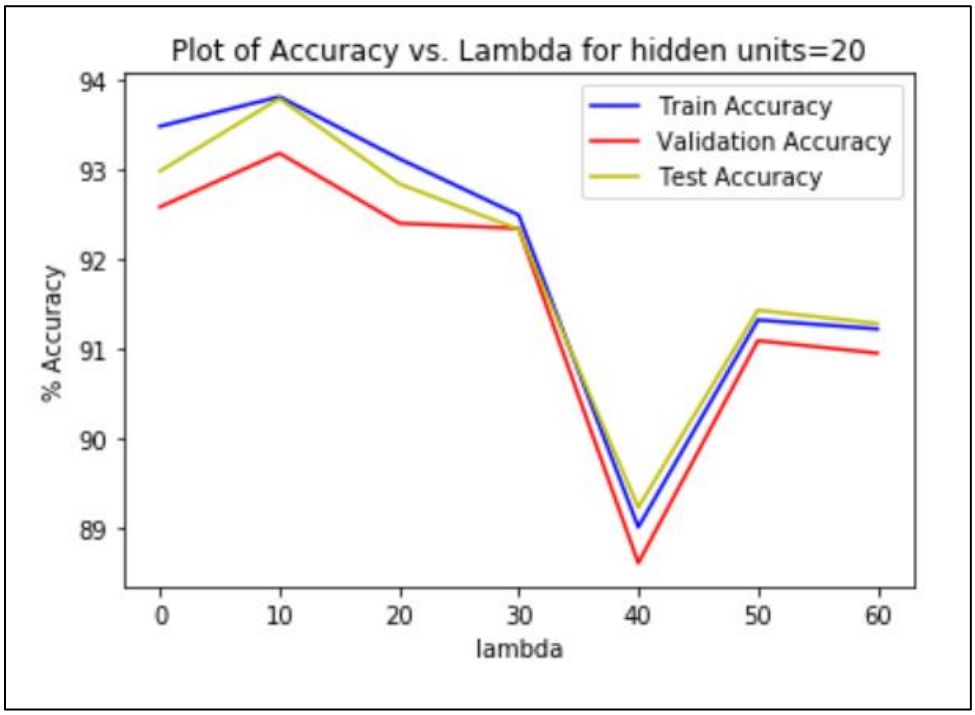
The output is optimized values of weights that are used as inputs for nnPredict function.

nnPredict function:

The optimized weight vector is then used as input to this function. Here using the Feedforward step, the output of output layer is computed. We use argmax function to find the unit that has the maximum probability value, this is our prediction.

Performance of Neural Network:

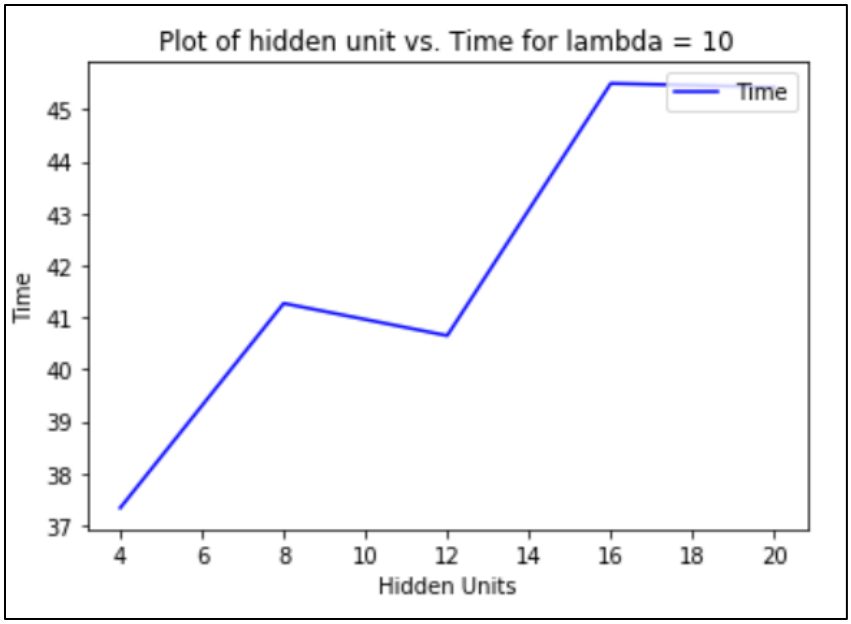
1. Selecting Optimal Hyper Parameters:



Regularization parameter lambda is incremented from 0 to 60 with step size of 10. From the graph, we see that the accuracy increases initially with increase in lambda value, however after 10, the accuracy for all datasets reduces.

Notice, at lambda=10 the accuracy for train, validation and test data is highest. Also, difference between training and test accuracy is minimum. This indicates that our model would neither under-fit (perform well on test data but not on train data) nor over-fit (perform well on train data but not on test data).

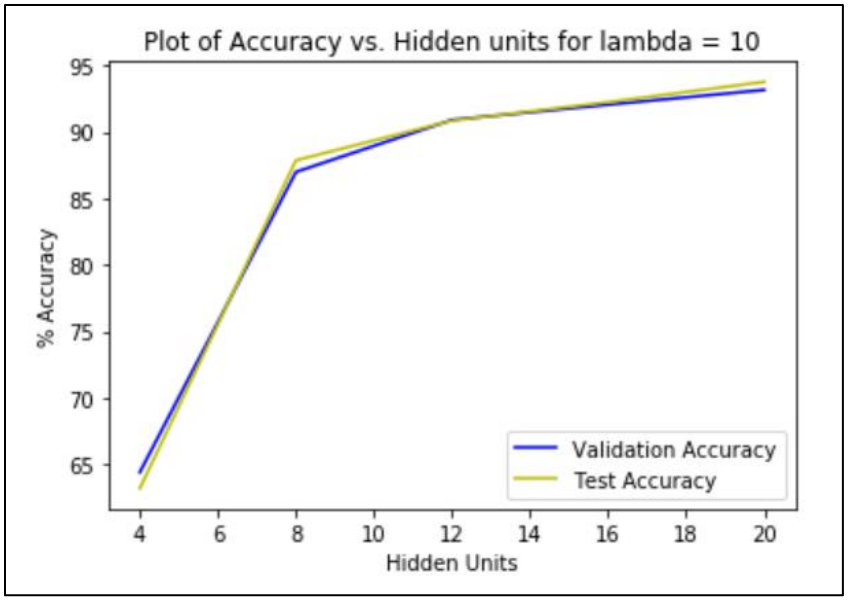
We now plot the time taken to train a neural network versus number of hidden units for lambda = 10.



To get a general sense of the relation between time to train versus hidden units, we plot the average time taken for different lambda values (0, 10, 20, 30, 40, 50, 60) versus hidden units.

From the graph above, we see that as the number of hidden nodes increases the time to train neural network increases. We notice a drop, but the general trend is that as the number of hidden nodes increases, the time taken to train a neural network also increases.

In order to find the optimal number of hidden units, we plot accuracy for validation and test data versus number of hidden units.



From the graph above, we can see that there is a steep jump in the accuracy initially and it then plateaus with increase in hidden units. After hidden units = 12, the increase is very less with increase in hidden units.

Looking at the above graphs and findings, we select the optimal hyper parameters as:

Lambda = 10

Number of Hidden Units = 12

The following hyper parameters have been saved in params.pickle file:- selected_features, number of hidden units, w1, w2 and optimal lambda value.

2. Accuracy of classification method on handwritten digits:

For optimal lambda value of 10 and number of hidden units 12:

Training set Accuracy: 91.426%

Validation set Accuracy: 90.92%

Test set Accuracy:90.86%

3. Accuracy of classification method on CelebA dataset:

For optimal lambda value of 10 and number of hidden units 256:

Training set Accuracy: 85.92%

Validation set Accuracy: 84.50%

Test set Accuracy: 86.15%

4. Comparison of Neural Network with Deep Neural Network:

	Neural Network with single Hidden Layer	Deep Neural Network
Accuracy in %	86.15	81.87
Time in seconds	118.07	611.52

Comparing the performance of neural network with deep neural network, we see accuracy of neural network with a single hidden layer is more than deep neural network. Also, a significant take away is the time taken. While the first one takes just 118 seconds to train the neural network, it takes 611 seconds to train deep neural network. This is because be adding just 1 more layer, the weights to be learned increases exponentially hence the huge increase in time.

Evaluating the Accuracy of Deep neural network for different number of hidden layers on CelebA dataset:

Number of Hidden Layers	Accuracy in %	Time in seconds
2	81.87	611.52
3	78.54	383.88
5	77.90	454.31
7	76.68	535.43

From the above table, we can see that for deep neural network, as we increase the number of hidden layers the accuracy decreases and the time taken to train the deep neural network also increases. So the performance is reducing overall.

For a linearly separable data, we don't require a hidden layer. Hidden layers are required for non-linear separable data. In most cases, a single hidden layer is sufficient to understand complexity of data and adding more layers doesn't improve the performance. As can be seen from the table, accuracy is maximum for hidden layers = 2.

5. Result of Convolutional Neural Network:

In order to save figures and implement matplotlib on Springsteen server, following code has been adapted from the link given below:

Code:

```
import matplotlib
matplotlib.use('Agg')
```

Link:
https://kampmannlab.ucsf.edu/sites/kampmannlab.ucsf.edu/files/matplotlib_class_server.pdf

Following are the changes that we made in the basecode:

print_test_accuracy(show_example_errors=True, show_confusion_matrix=True)

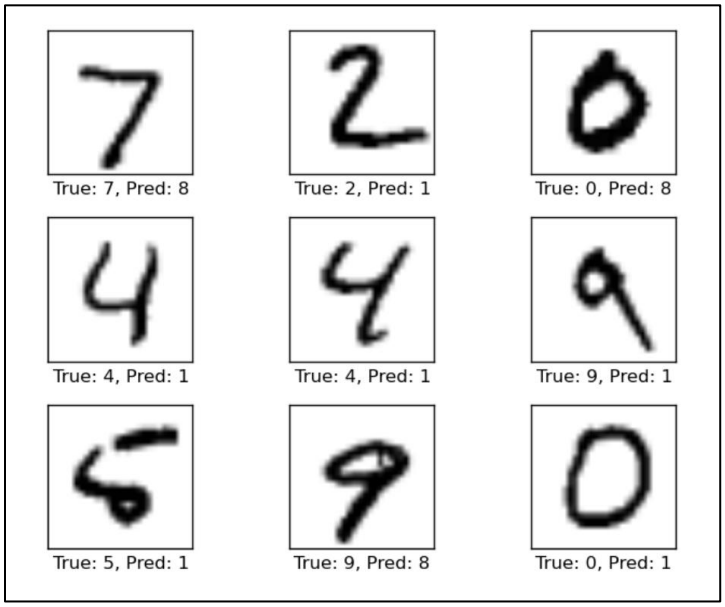
We changed **show_example_errors from False to True** and to display the confusion matrix added **show_confusion_matrix = True.**

Accuracy vs Training Time for each iteration using optimization:-

Number of Iterations	Accuracy	Training Time
1	10.9%	0:00:00
99	65.8%	0:00:07
999	92.3%	0:01:05
9999	98.7%	0:10:59

Iteration = 1 without optimization:

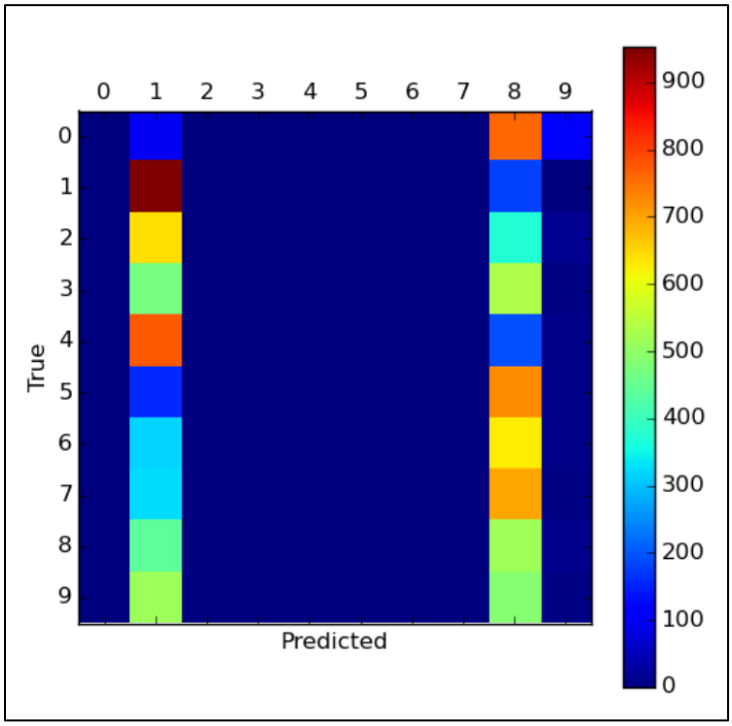
Error snapshot:



Confusion Matrix:

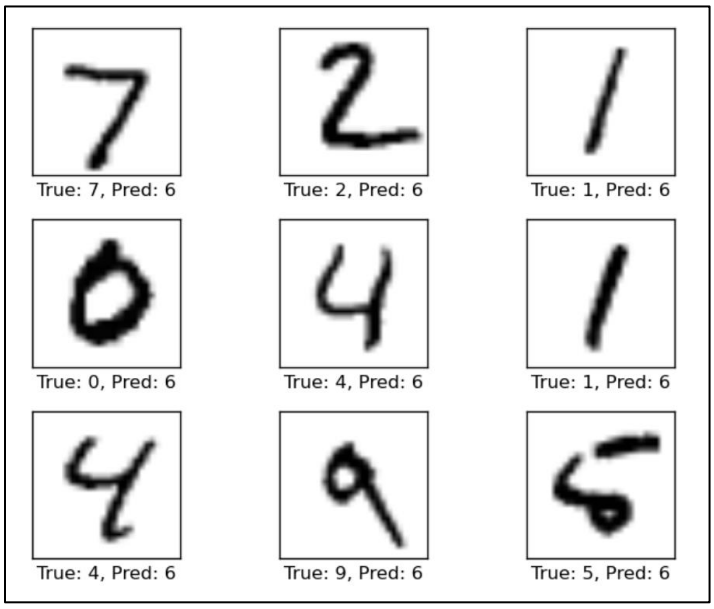
```
Size of:
- Training-set:         55000
- Test-set:             10000
- Validation-set:       5000
Accuracy on Test-Set: 14.8% (1478 / 10000)
Example errors:
Confusion Matrix:
[[  0  97   0   0   0   0   0   0 762 121]
 [  0 953   0   0   0   0   0   0 181   1]
 [  0 641   0   0   0   0   0   0 373  18]
 [  0 470   0   0   0   0   0   0 535   5]
 [  0 776   0   0   0   0   0   0 197   9]
 [  0 159   0   0   0   0   0   0 723  10]
 [  0 319   0   0   0   0   0   0 628  11]
 [  0 326   0   0   0   0   0   0 697   5]
 [  0 442   0   0   0   0   0   0 519  13]
 [  0 514   0   0   0   0   0   0 489   6]]
```

Confusion Matrix Plot:



With optimization Iteration = 1:

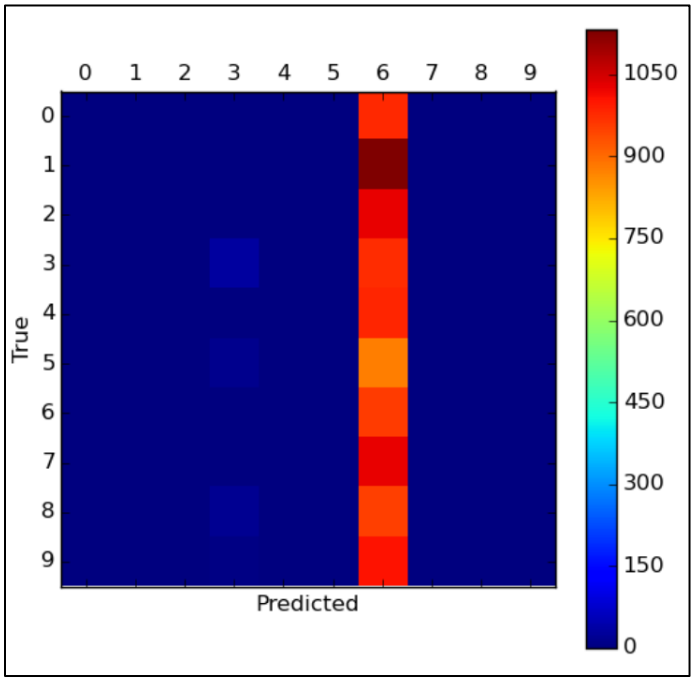
Error Snapshot:



Confusion Matrix:

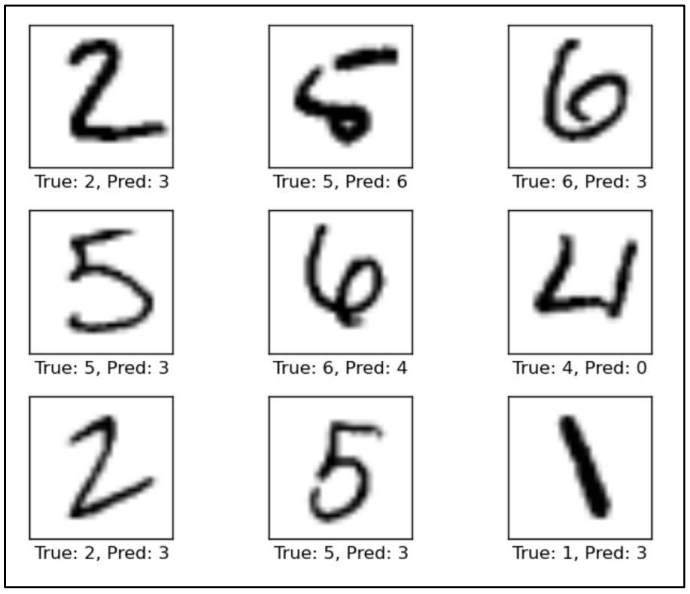
```
Accuracy on Test-Set: 9.9% (990 / 10000)
Example errors:
Confusion Matrix:
[[   0    0    0    0    0    0  980    0    0    0]
 [   0    0    0    3    0    0 1132    0    0    0]
 [   0    0    0    2    0    0 1030    0    0    0]
 [   0    0    0   33    0    0  977    0    0    0]
 [   0    0    0    0    0    0  982    0    0    0]
 [   0    0    0   16    0    0  876    0    0    0]
 [   0    0    0    0    0    0  957    0    0    1]
 [   0    0    0    0    0    0 1028    0    0    0]
 [   0    0    0   21    0    0  953    0    0    0]
 [   0    0    0    5    0    0 1004    0    0    0]]
```

Confusion Matrix Plot:



With optimization Iteration = 99:

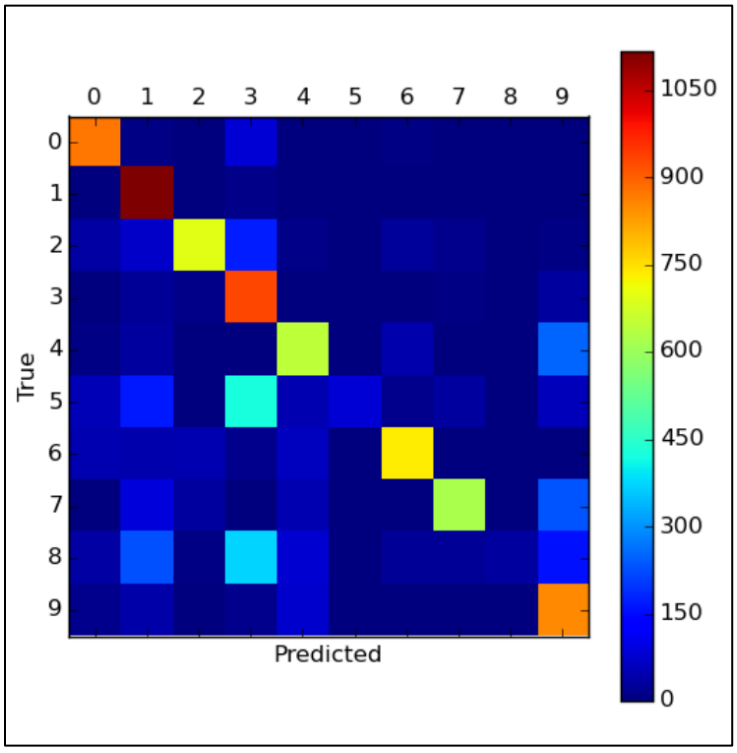
Error Snapshot:



Confusion Matrix:

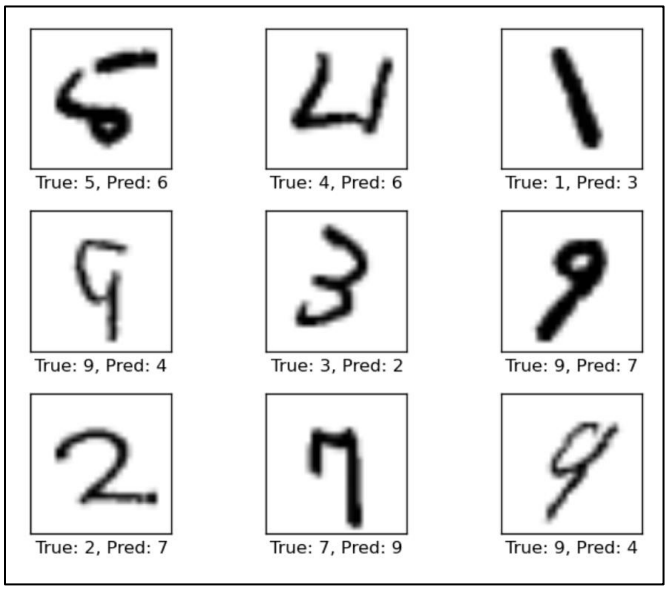
```
Time usage: 0:00:07
Accuracy on Test-Set: 65.8% (6582 / 10000)
Example errors:
Confusion Matrix:
[[ 875    7    0   85    2    0    8    0    0    3]
 [   0 1116    2    9    0    0    4    1    0    3]
 [  35   70  694  171    9    0   28   17    0    8]
 [   1   26    9  929    3    0    1    8    0   33]
 [   6   34    1    0  646    0   44    1    0  250]
 [  55  167    3  426   48   84   17   32    1   59]
 [  49   46   49   17   62    1  733    1    0    0]
 [   2   88   31    3   49    0    0  621    0  234]
 [  39  229    7  373   80    2   26   26   31  161]
 [  17   42    4   15   76    0    1    1    0  853]]
```

Confusion Matrix Plot:



With optimization Iteration = 999:

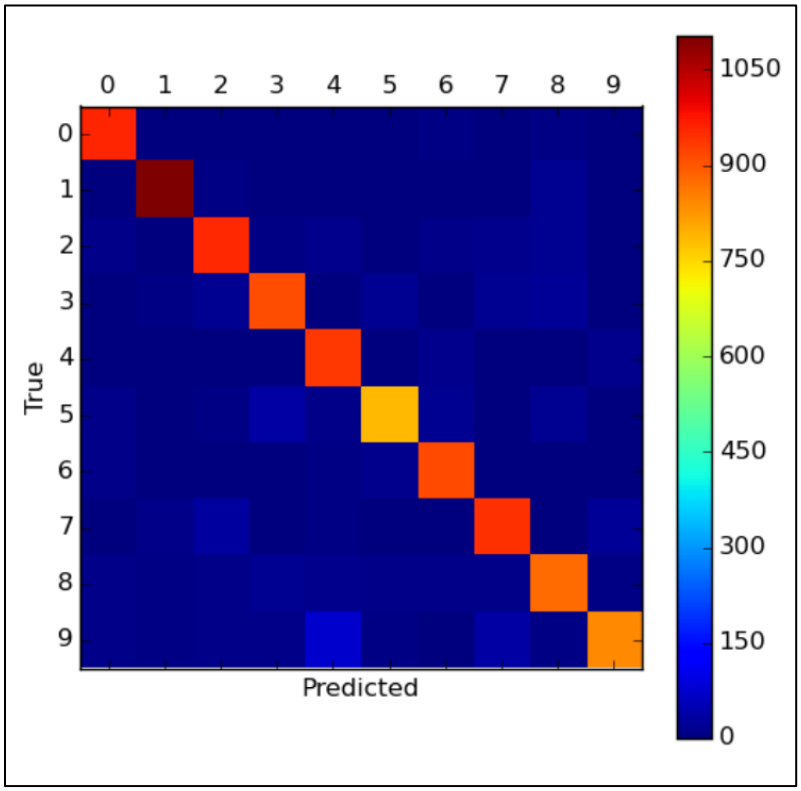
Error Snapshot:



Confusion Matrix:

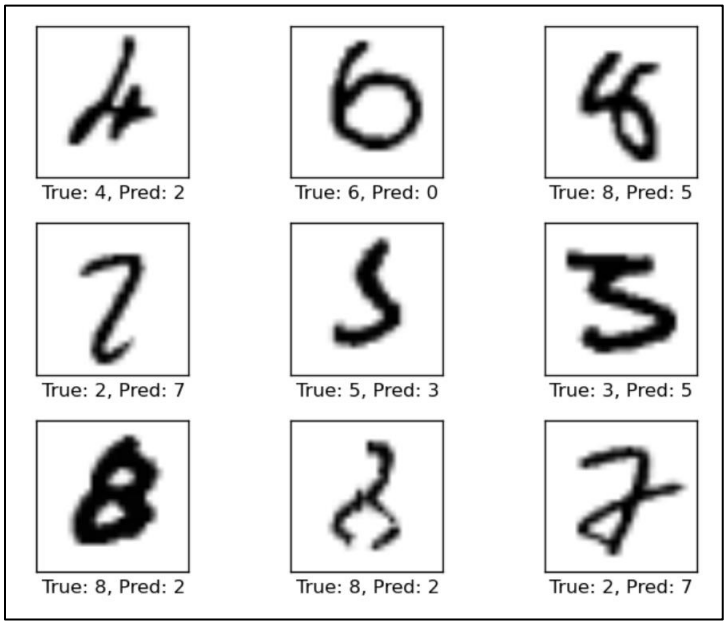
```
Time usage: 0:01:05
Accuracy on Test-Set: 92.3% (9230 / 10000)
Example errors:
Confusion Matrix:
[[ 959    0    1    2    0    3    8    1    6    0]
 [   0 1103    6    3    0    1    3    0   19    0]
 [  11    2  953    8   16    0    9   13   20    0]
 [   2    7   21  913    1   21    0   21   22    2]
 [   1    3    4    0  936    0   17    2    2   17]
 [   9    2    5   35   12  785   21    2   18    3]
 [   9    4    3    0    7   16  914    1    4    0]
 [   1   10   31    4    5    1    0  947    4   25]
 [   9    6    9   21   17    9   10   12  876    5]
 [  10    6   10   11   77    5    0   38    8  844]]
```

Confusion Matrix Plot:



With optimization Iteration = 9999:

Error Snapshot:



Confusion Matrix:

```
Time usage: 0:10:59
Accuracy on Test-Set: 98.7% (9869 / 10000)
Example errors:
Confusion Matrix:
[[ 972    0    1    0    0    0    2    1    3    1]
 [   0 1129    2    0    0    1    1    0    2    0]
 [   0    2 1019    2    1    0    0    4    3    1]
 [   1    0    0  998    0    6    0    2    2    1]
 [   0    0    2    0  975    0    0    0    0    5]
 [   2    0    0    3    0  886    1    0    0    0]
 [   3    3    0    0    3   11  938    0    0    0]
 [   1    1    3    1    0    0    0 1017    1    4]
 [   4    0    6    4    1    5    1    3  945    5]
 [   2    3    1    1    6    2    0    4    0  990]]
```

Confusion Matrix Plot: