Project 3 Report

**2.5 Overfitting + 2.7**

Using a random sample of 2000 images per epoch, no overfitting occurs. This can be seen by how the validation loss and accuracy very nearly matches that of the training set, in Figure 1. It is important to note that while usually validation/testing accuracy should not be higher than training accuracy, I run the validation set after the training set updates the model’s weights in each epoch. Especially in the beginning, this is why the validation loss and accuracy performs better than the training.

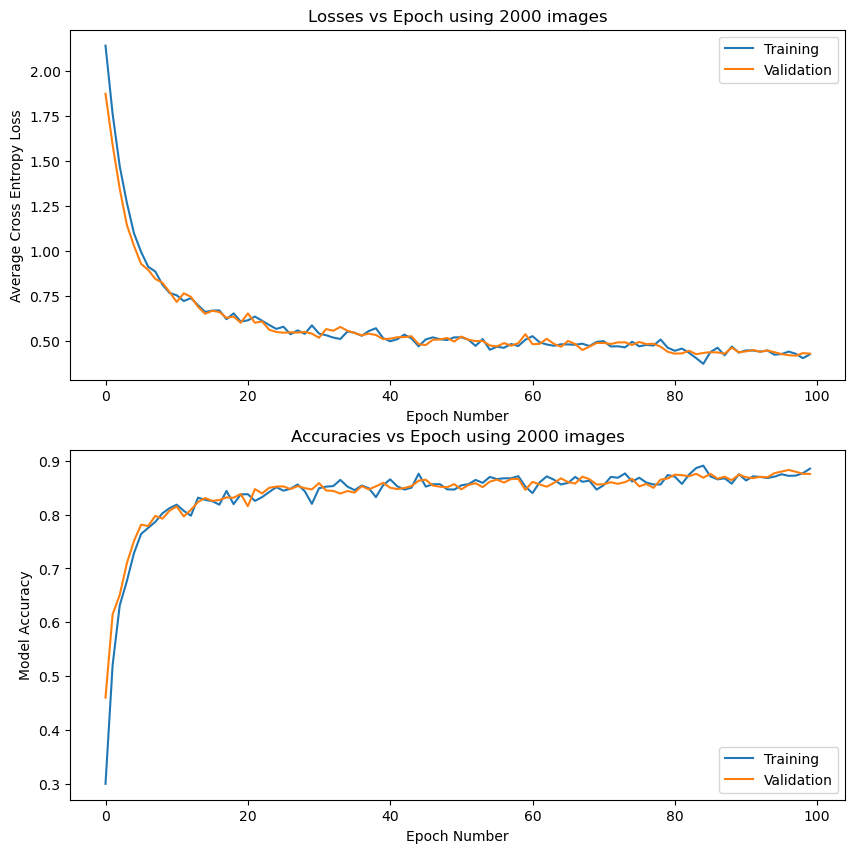


Figure 1: MLP Performance using 2000 images per epoch

Using all 50,000 images for training, I also see no overfitting in Figure 2. This modification also allows the model to keep steadily improving faster, while the 2000 images one seemed to stagnate/improve much slower. While the 2000 images model reached a validation accuracy less than 90%, this one using the full dataset reached a validation accuracy of 92.7%, and looks like it would keep improving steadily past 100 epochs. Although I was expecting to have to implement a modification to get the model to get above 90% validation accuracy, I decided this model should be good enough to run on the testing set now without modifications. The MLP parameters (W1, b1, W2, b2) are also saved as “MLP\_weights.npz” at this time, and frozen before testing.

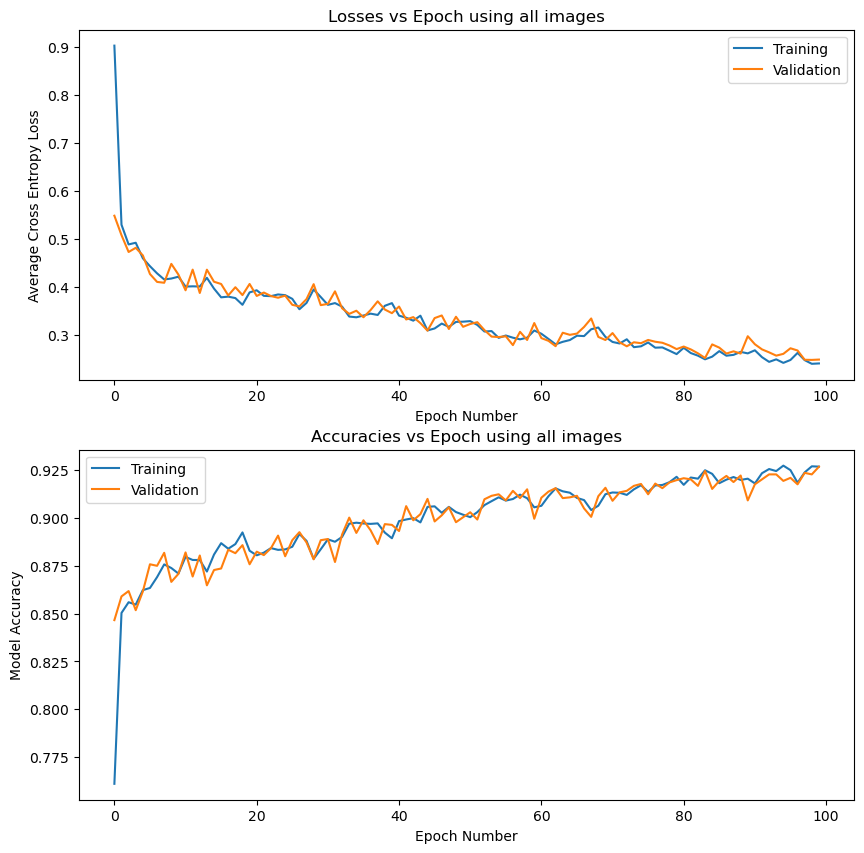
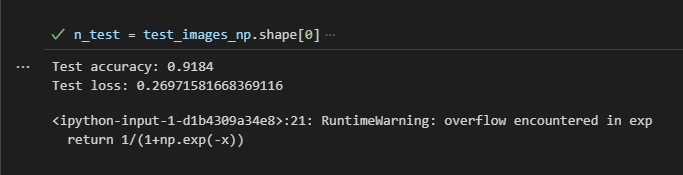


Figure 2: MLP Performance using mini-batches over the entire training set

**2.6 How well does it work**

Since we’re testing, we cannot improve our model and have nothing to graph, so here are the printed accuracy and loss of the mode on the test set:



**2.7 Saving and loading weights**

You should be able to use the def load(self, filename) function I defined in the MLP class to load the weights from 2.5 and get the same answer as 2.6 - I did not make a predict function, but if you run:

>> import numpy as np

>>

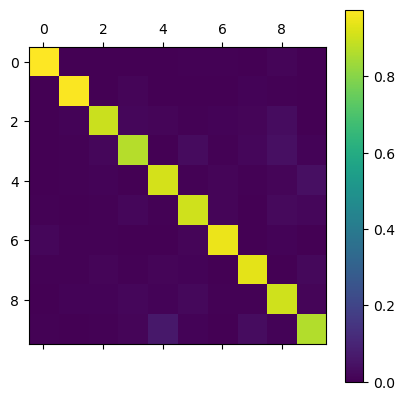
test\_images\_np=np.load('./Project3\_Data/MNIST\_test\_images.npy')

test\_labels\_np=np.load('./Project3\_Data/MNIST\_test\_labels.npy')

and the code block starting with #%% 2.6, it should work

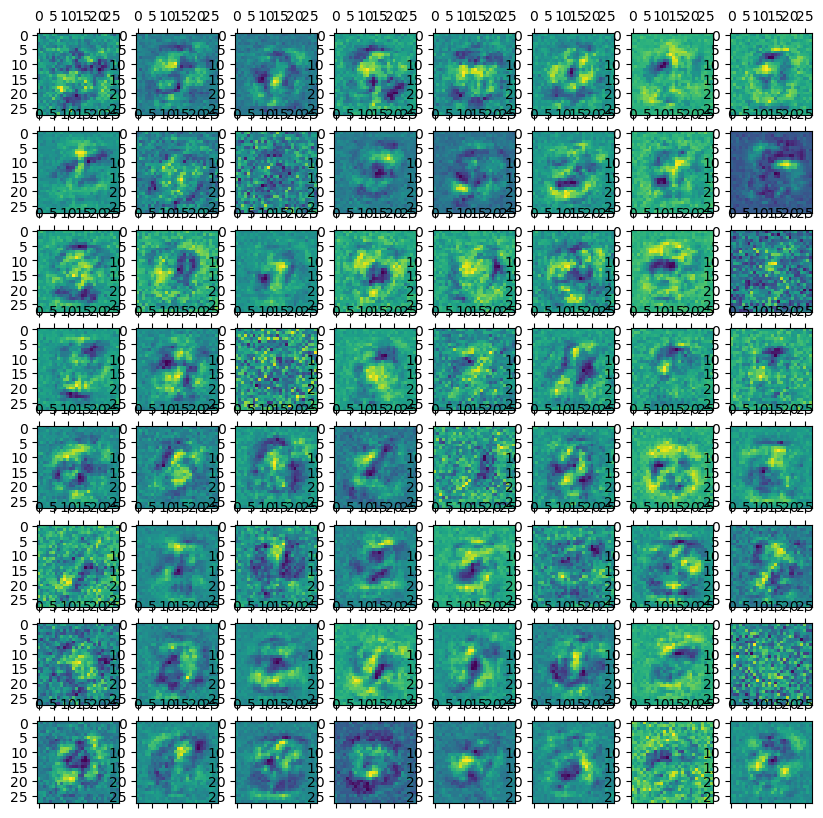
**2.8 Confusion Matrix**

The real labels of the test set are the rows and the predictions are columns. 3 and 9 are hardest to identify: 3 has the most mistakes with 5 and 8, and 9 has the most mistakes with 4 and 7

****

**2.9 Visualize the Weights**

Below are visualizations of the different “templates” of W1. W1 is 64x784, where 784 comes from the vectorized xs (inputs). Each image is 28x28, so we reshape each W1i back from 1x784 to 28x28. Most of these templates look fairly smooth, which makes sense for learning different configurations of smooth lines. For example, the second row rightmost column could be a template closely matching the number “5” because of the s-shape. However, some templates still look nearly like complete noise, such as row 2 column 3, row 4 column 3, and row 7 column 8. These templates probably do not contribute much to the model learning, or overfits the data.

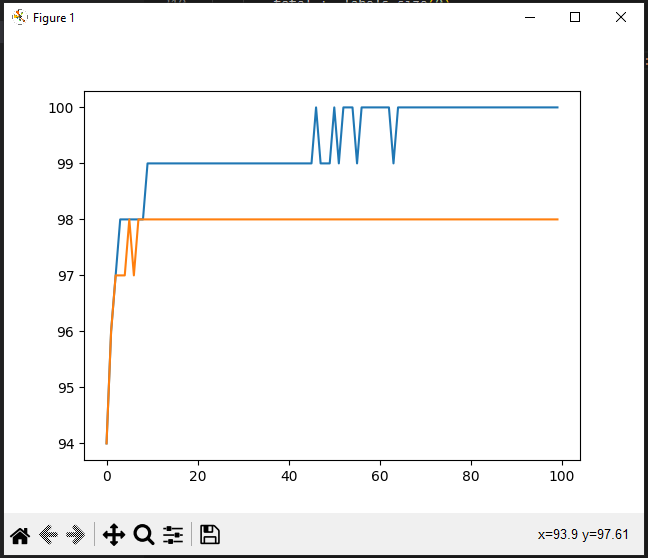


**3.1 Describing CNN**

This CNN first takes in a 1-channel input image (e.g. grayscale) in a convolutional layer, and outputs 6 channels, meaning it applies 6 filters onto the image. The kernel size of the filters is 5 (5x5), and since there are no other parameters, stride=1, padding=0, and padding\_mode='zeros' for this first convolution layer. Then, these values go into ReLU (rectified linear unit) activation functions. Next, the image gets reduced to half the size (in both directions) - the maximum pixel value from each set of 2x2 pixels is chosen to be the one remaining in the smaller version. Then, a second layer of convolution is applied: 16 filters, also of a kernel size 5 and no padding, so 4 pixels will be removed from the sides of the image. These also get fed into ReLU functions, and then get pooled again, reducing the size by half. After this, the values are fed into three fully connected layers: The first layer takes this output and expands it to a large row image (1x120), then decreased to 84, and then 10 for a final choice between classification .

**3.3 Overfitting**

The blue represents the training dataset, and the orange represents the testing dataset. As we can see, the training dataset seems to always be slightly ahead of the testing dataset. Due to us using a large dataset, there is a lower risk of overfitting, and it seems that 98% is similar to around 100% accuracy.



**3.5 Saving and loading the weights**

The CNN model is saved within cnnweights.pt and can be loaded as the last code segment. It is required to have the load function and also the datasets. The final accuracy should be 98%.