EE/CNS/CS 148 HW 3: MNIST Classification Using Convolutional Neural Networks

Johanna Karras

Assignment Description: In this assignment, I design, build, train, and evaluate a convolutional neural network using Pytorch for classification of MNIST handwritten digits.

GitHub Repo: `https://github.com/JSKarras/MNIST-Classification-CNN`

1 Data Augmentation

In this section, I compare the performance of the default CNN and that of the CNN once data augmentation has been added to the data loader.

For data augmentation, I used the following set of transformations

```
train_transform = Compose([
    transforms.GaussianBlur(kernel_size=15, sigma=(0.01,0.2)),
    transforms.ColorJitter(brightness=0.1, saturation=0.1, hue=0.1),
    transforms.RandomAffine(degrees=15, scale=(0.9,1.1)),
    transforms.ToTensor(),
    transforms.Normalize(mean=0.131, std=0.3085)
])
```

where mean and std refer to the mean and standard deviation that are calculated from the MNIST train dataset.

GaussianBlur: Apply a gaussian blur of size 15pixels to the digit where the standard deviation is chosen between 0.1 and 0.5.

ColorJitter: Jitter the color of the image by randomly multiplying the pixels by a brightness factor randomly chosen between 0 and 1, a randomly chosen saturation factor between 0 and 2, and a randomly chosen hue factor between -0.2 and 0.2.

RandomAffine: Apply a random rotation between -30 and 30 degrees to the image and a random shear between -45 and 45 degrees.

Note: I originally set the random rotation to 45 degrees and added a random horizontal flip, but this lowered the accuracy. That was likely because it is too difficult to learn the digits that are rotated too much, since rotation is an important attribute to the recognizability of a digit.

Normalize: Images are normalized to the same mean and pixel values are scaled from 0-255 to 0-1.

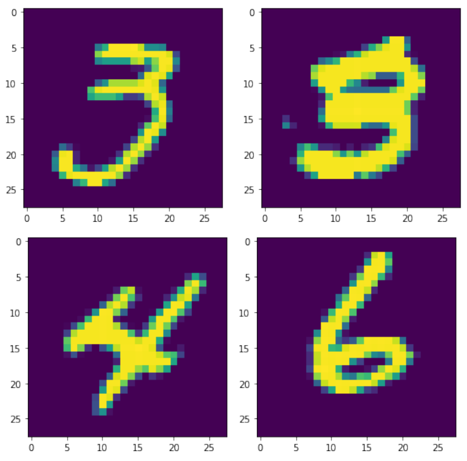


Figure 1: Examples of augmented MNIST images.

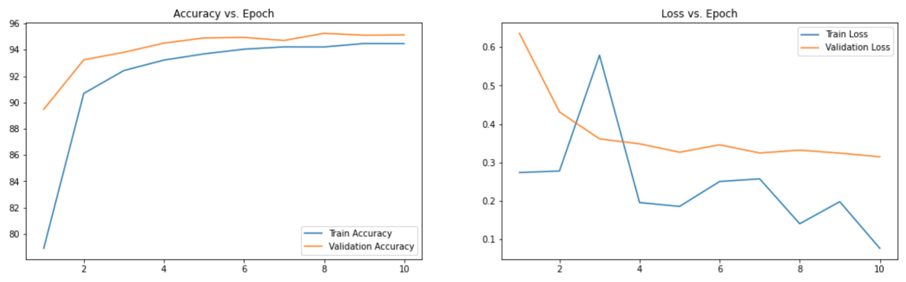
	Train Accuracy	Validation Accuracy
Default ConvNet	94%	95%
Default ConvNet w/ Data Augmentation	93%	96%

Table 1: Comparison of final accuracy of the default ConvNet, with and without data augmentation, after ten epochs.

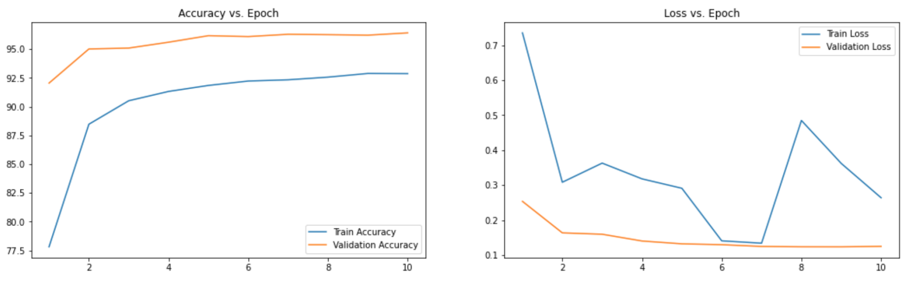
Adding in data augmentation increases the diversity of the training data, without requiring the acquisition of more training data. As a result, the network is more robust and less likely to overfit. This is evidenced by the numerical results in Table 1 and plots in Figure 2. In Table 1, we see that although the network trained without data augmentation achieves a higher training accuracy of 94%, compared to the 93% of the network trained with data augmentation, its validation accuracy of 95% is lower than the 95% training accuracy of the network trained with data augmentation.

Similarly, these results are corroborated by the loss and accuracy plots of training and testing datasets during training of both networks, shown in Figure 2. Notice that the validation loss plateaus even as the training curve continues to decrease, a sign of overfitting, for the network trained without data augmentation. It is, however, odd that the validation accuracies are better than the training accuracies for both networks, and even the validation loss is lower than the training loss for the network trained with data augmentation. This may just be a coincidence of the dataset or random luck of the random initialization of the weights.

For VGG16 and ResNet50, the discrepancy is much more profound with and without data augmentation, likely because they are larger networks with more parameters and thus more likely to overfit. For ResNet50 for example, with data augmentation, the final validation and training accuracies are 99%. Without data augmentation, although training accuracy is still 99%, the final validation accuracy is only 87%!



(a) Training without data augmentation.

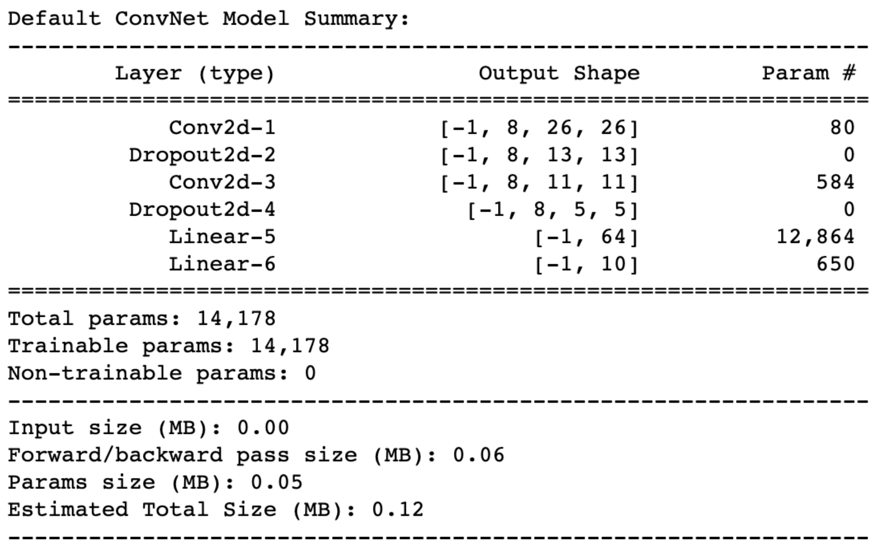


(b) Training with data augmentation.

Figure 2: Accuracy and loss plots during training of training and validation sets.

2 Custom Convolutional Neural Network

In designing my custom convolutional neural network, I began with the default convolutional neural network, summarized in Fig. 3. Then, I experimented by changing one parameter at a time, as summarized in section 2.1. Afterwards, I implemented a couple of object detection networks discussed in class using PyTorch, namely VGG16 and ResNet.



```
Default ConvNet Model Summary:
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1            [-1, 8, 26, 26]              80
         Dropout2d-2            [-1, 8, 13, 13]               0
            Conv2d-3            [-1, 8, 11, 11]             584
         Dropout2d-4              [-1, 8, 5, 5]               0
            Linear-5                   [-1, 64]          12,864
            Linear-6                   [-1, 10]             650
================================================================
Total params: 14,178
Trainable params: 14,178
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.05
Estimated Total Size (MB): 0.12
----------------------------------------------------------------
```

Figure 3: Model summary of the default CNN.

2.1 Tuning Parameters in Default ConvNet:

2.1.1 Batch Normalization

Batch normalization normalizes inputs to layers of the network. According to numerous sources I referenced, batch normalization speeds up training and prevents over-fitting to the noise in the training data. However, after adding batch normalization after the activation (ReLU) function, the results were disastrous (see Fig. 4). This makes sense, because ReLU is a linear function, so if the layers are always normalized after applying ReLU, the inputs get smaller and smaller with each batch normalization. Then, I changed the ordering to place the batch normalization after the ReLU, and achieved much less overfitting and also the values of both training and validation accuracy and loss stabilized quickly after only a few epochs of training.

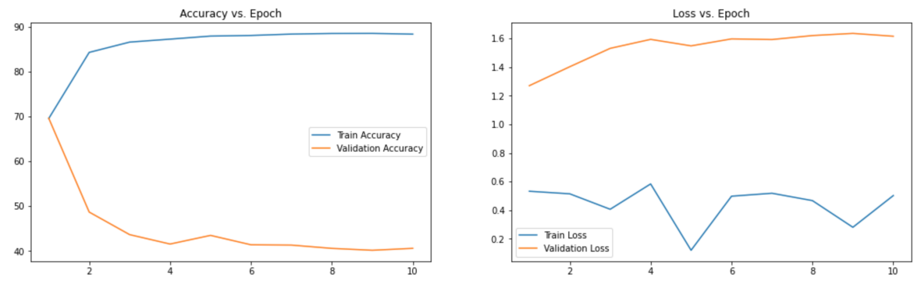


Figure 4: Extreme overfitting when batch normalization is added before the ReLU activation.

2.1.2 Output Channels (Breadth):

Then, I increased the output channels from 8 to 32 (as well as recomputed the shapes of the proceeding layers). This helped both training and validation accuracy increase a great deal in the beginning of training and stabilize after only a few epochs. However, there was more overfitting, since the final train accuracy was 97% and the final validation accuracy was 92%. Moreover, the validation curve decreases noticeable after only the second epoch.

Later on, I increase the output channels to 64 in the second convolutional layer and back to 32 after the third convolutional layer.

2.1.3 Adding More Convolutional Layers (Depth):

I increased the number of convolutional layers (Conv2d) from two to three.

2.1.4 Early Stopping:

As such, I thought about decreasing the learning rate, but decided to implement an early stopping callback, instead. The reason is that a smaller learning rate would cause training to take more time, instead of just stopping at an earlier epoch as with a callback that monitors the validation loss. I simply stop training once there is a decrease in validation accuracy after one epoch. As such, training stops after 5 epochs with a final train accuracy was 97% and a final validation accuracy was 98%.

2.1.5 Increasing Stride:

If I increase stride in a very early layer, the number of parameters decreases quickly in the network, causing worse performance.

2.1.6 Padding:

I noticed that Conv2d reduces the image size quite a bit, which intuitively seems like loss of information that could be useful for learning. So I decided to try adding padding to the Conv2d layers in effort to increase the image size. Initially, I added 2 pixels of padding into each Conv2d layer and this gratly increased the number of learnable parameters from 14,000 to 141,000. More importantly, it also raised test performance to

2.1.7 Optimizers:

I tested Adadelta, Adam, Adamax, SGD, and RMSprop optimizers. Most performed poorly and the accuracy stagnated around 10%. The best was the default Adadelta optimizer.

2.1.8 Increasing Batch Size:

I tried increasing the batch size from 32 to 128 for train and validation datasets. This increased the speed of training down to 2 minutes! The final accuracies were about the same as before.

2.2 Exploring VGG16 and ResNet50 Implementations

After playing around with the default network, I looked into the implementations of two state-of-the-art image clasifaction networks discussed in lecture, namely VGG16 and ResNet50.

2.2.1 VGG Implementation:

I had to switch the optimizer to Adam and the learning rate to 0.001 in order to prevent the train accuracy from plateauing at 10%. This worked out very well and the final train accuracy was 98% and validation accuracy was 99%.

2.2.2 ResNet50 Implementation:

In the ResNet50, there is the introduction of skip connections that (1) allow for alternative pathways through the network during back-propagation, which is beneficial to avoid vanishing gradients, and (2) allow the network to learn a simpler function, $x + F(x)$, through each block. I tried training the ResNet50 for 10 epochs using the SGD optimizer, which I read online is good for ResNet's, with various learning rates from 1 to 1e-6, but the loss would eventually become too negative and become NaN. What is strange is that no matter what optimizer I pick, I get a negative loss function, though it doesn't seem to affect the accuracy from increasing. I actually found Adamax with a learning of 0.001 worked fine, getting to a final train accuracy was 98% and final validation accuracy of 99%.

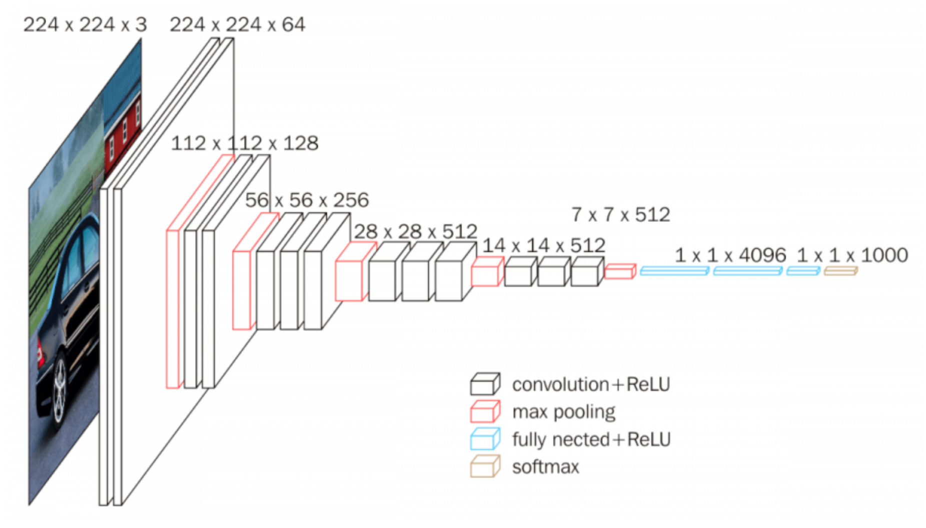
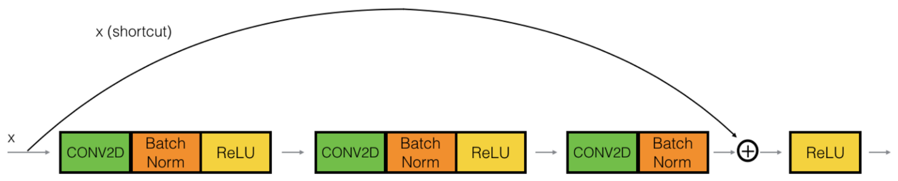
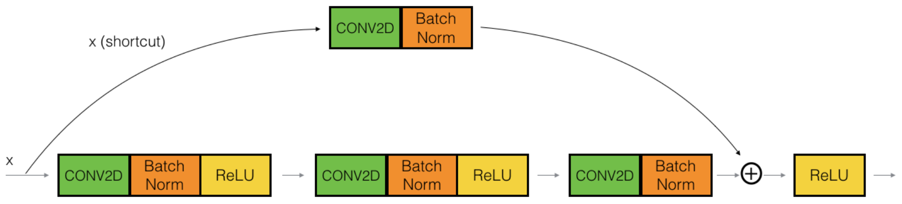


Figure 5: VGG 16 Network Architecture



(a) ID block with skip connection.



(b) Convolution block with skip connection.

Figure 6: Two main blocks of ResNet 50.

I used this GitHub repository to develop my implementation of ResNet50: `https://github.com/JayPatwardhan/ResNet-PyTorch/blob/master/ResNet/ResNet.py`.

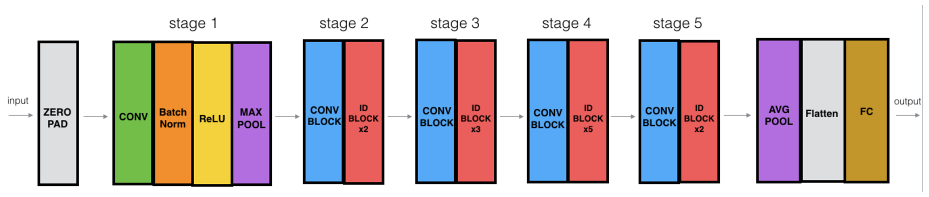


Figure 7: ResNet 50 Architecture

2.3 Final Custom Convolutional Network

Drawing inspiration mostly from VGG, I implemented my custom CNN to be both wider and deeper than the original ConvNet() using increased output channels and convolutional layers. I train using the data-augmented dataset and the Adam optimizer with a learning rate of 0.001. Here is a summary of the model: Below are the results compared to the default ConvNet() provided.

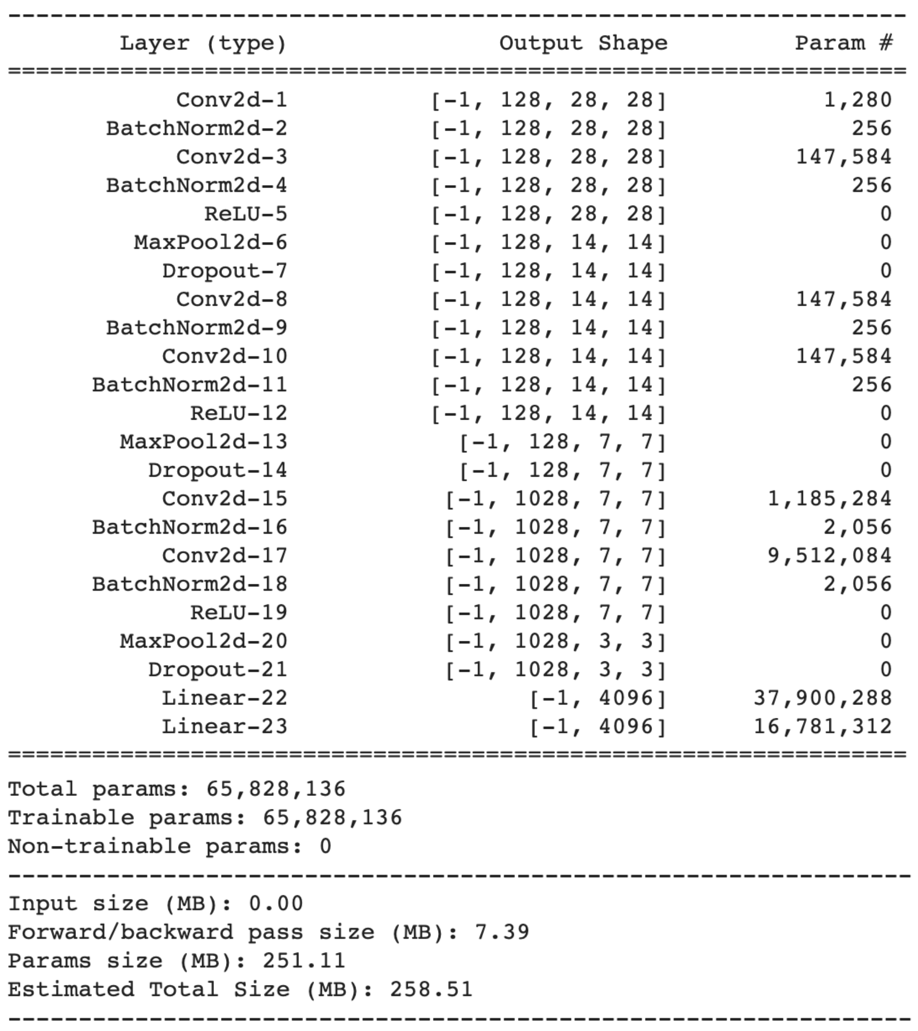
	Train Accuracy	Validation Accuracy
Default ConvNet	94%	95%
Default ConvNet w/ Data Augmentation	93%	96%
Custom Net w/ Data Augmentation	99%	99%

Table 2: Comparison of final accuracy of the default ConvNet, with and without data augmentation, after ten epochs.

3 Evaluaton:

In this section, I evaluate the model on the unseen test set when training on different-sized random subsets (1/2, 1/4, 1/8, 1/6) of the training data. The numerical results of training and testing error versus number of training samples are in Table 3 and the plot of these values is in Figure 9.

As the number of training samples increases, both the training and testing errors decrease, as expected. However, I notice that the errors remain relatively small, less than 0.1, up to when 1/8 of the training data is used.
These models can be found inside the folder called `Models`.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 128, 28, 28]           1,280
       BatchNorm2d-2          [-1, 128, 28, 28]             256
            Conv2d-3          [-1, 128, 28, 28]         147,584
       BatchNorm2d-4          [-1, 128, 28, 28]             256
              ReLU-5          [-1, 128, 28, 28]               0
         MaxPool2d-6          [-1, 128, 14, 14]               0
           Dropout-7          [-1, 128, 14, 14]               0
            Conv2d-8          [-1, 128, 14, 14]         147,584
       BatchNorm2d-9          [-1, 128, 14, 14]             256
           Conv2d-10          [-1, 128, 14, 14]         147,584
      BatchNorm2d-11          [-1, 128, 14, 14]             256
             ReLU-12          [-1, 128, 14, 14]               0
        MaxPool2d-13            [-1, 128, 7, 7]               0
          Dropout-14            [-1, 128, 7, 7]               0
           Conv2d-15           [-1, 1028, 7, 7]       1,185,284
      BatchNorm2d-16           [-1, 1028, 7, 7]           2,056
           Conv2d-17           [-1, 1028, 7, 7]       9,512,084
      BatchNorm2d-18           [-1, 1028, 7, 7]           2,056
             ReLU-19           [-1, 1028, 7, 7]               0
        MaxPool2d-20           [-1, 1028, 3, 3]               0
          Dropout-21           [-1, 1028, 3, 3]               0
           Linear-22                 [-1, 4096]      37,900,288
           Linear-23                 [-1, 4096]      16,781,312
================================================================
Total params: 65,828,136
Trainable params: 65,828,136
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 7.39
Params size (MB): 251.11
Estimated Total Size (MB): 258.51
----------------------------------------------------------------
```

Figure 8: My custom model summary.

	Train Error	Test Error
Full Training Set	0.0228	0.0157
1/2 Training Set	0.0468	0.0254
1/4 Training Set	0.0635	0.0406
1/8 Training Set	0.1151	0.0767
1/16 Training Set	0.2788	0.2105

Table 3: Comparison of training and testing accuracies when training my custom model on different-sized random subsets of the training data.

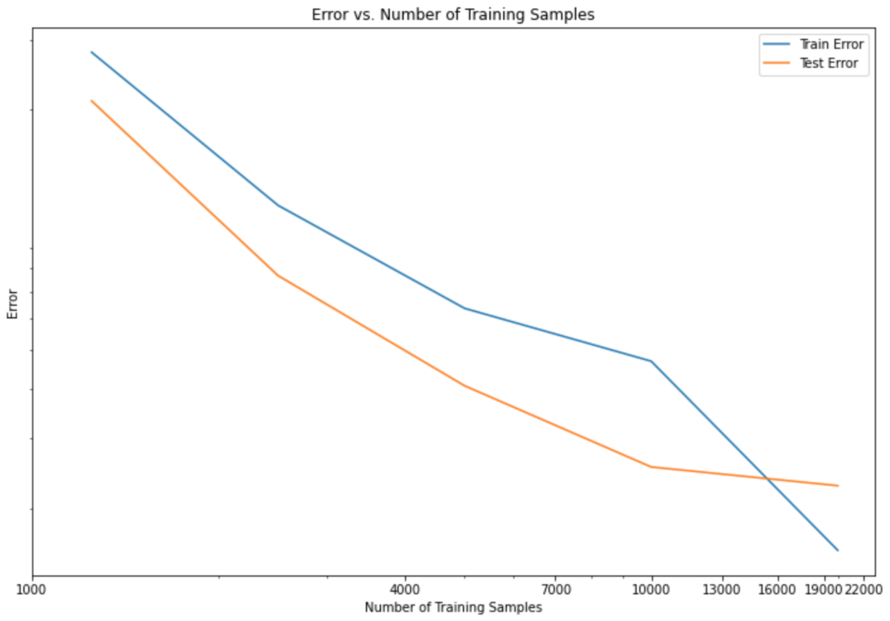


Figure 9: Training and testing errors versus number of training samples.

4 Incorrect Predictions

In Figure 10, I show 9 examples of incorrect predictions on the test set resulting from my custom CNN. Most of the incorrect predictions are actually pretty reasonable, meaning that the numbers are written ambiguously and could be mistaken for other digits, even to a human observer. For instance, in the first row and second column of Figure 10, the “5” could easily be a sloppy “3” and in the first row, third column, the “5” actually does look more like a “6”, as predicted by the neural network.

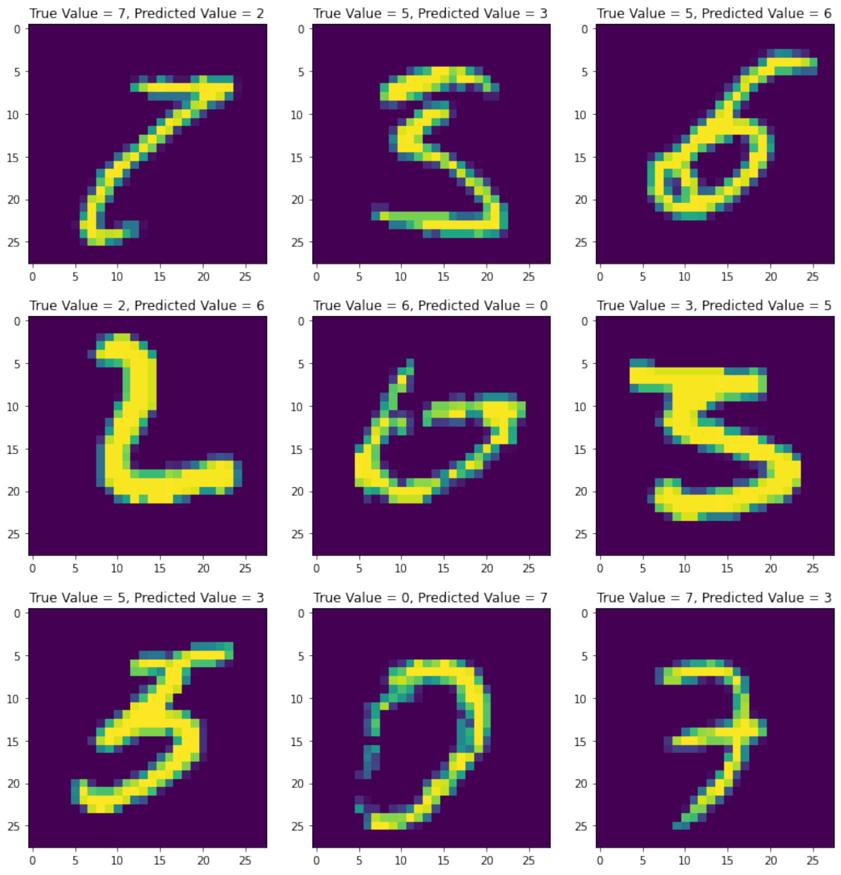
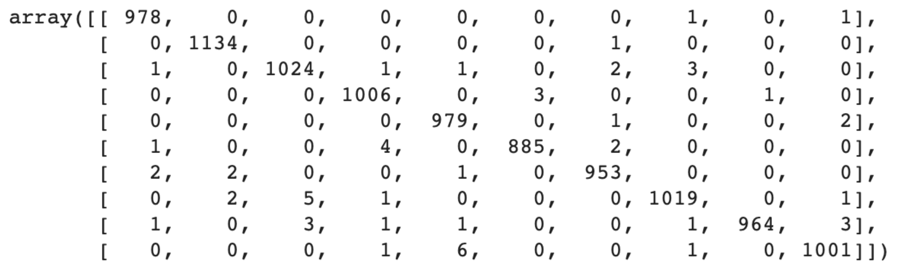

True Value = 7, Predicted Value = 2
True Value = 5, Predicted Value = 3
True Value = 5, Predicted Value = 6
True Value = 2, Predicted Value = 6
True Value = 6, Predicted Value = 0
True Value = 3, Predicted Value = 5
True Value = 5, Predicted Value = 3
True Value = 0, Predicted Value = 7
True Value = 7, Predicted Value = 3

Figure 10: Incorrect image predictions.

5 Confusion Matrix

In the confusion matrix below, we see that for the most part, the predictions are correct, since the diagonal (where prediction equals ground truth) has the most test samples (highest values). Of the incorrect predictions, the most common are confusing “4” for “9” (tenth row, fifth column), since this is where the high value is outside of the diagonal.



```
array([[ 978,    0,    0,    0,    0,    0,    0,    1,    0,    1],
       [   0, 1134,    0,    0,    0,    0,    1,    0,    0,    0],
       [   1,    0, 1024,    1,    1,    0,    2,    3,    0,    0],
       [   0,    0,    0, 1006,    0,    3,    0,    0,    1,    0],
       [   0,    0,    0,    0,  979,    0,    1,    0,    0,    2],
       [   1,    0,    0,    4,    0,  885,    2,    0,    0,    0],
       [   2,    2,    0,    0,    1,    0,  953,    0,    0,    0],
       [   0,    2,    5,    1,    0,    0,    0, 1019,    0,    1],
       [   1,    0,    3,    1,    1,    0,    0,    1,  964,    3],
       [   0,    0,    0,    1,    6,    0,    0,    1,    0, 1001]])
```

Figure 11: Confusion Matrix

6 tSNE

In the 2-dimensional tSNE plot, we can see that digit clusters that are similar to each other visually are close together, while digits that are visually far from each other also have clusters that are far from each other. For example, “9” and “4” are clustered close together, but “0” and “1” are far away.

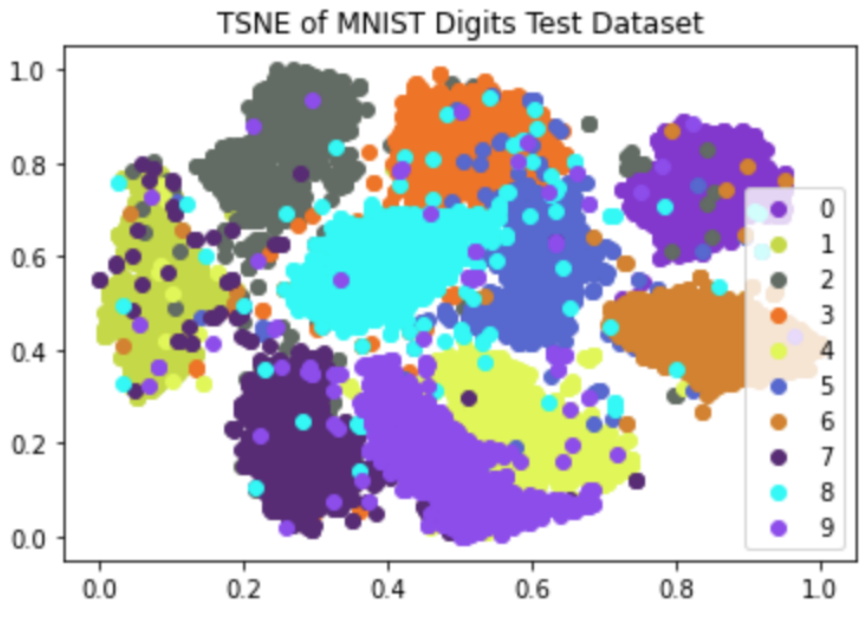
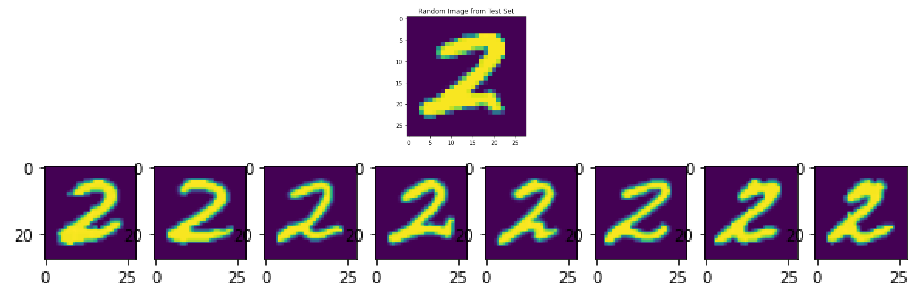
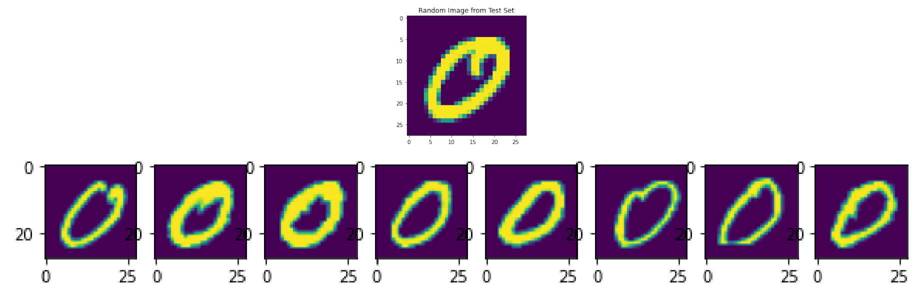


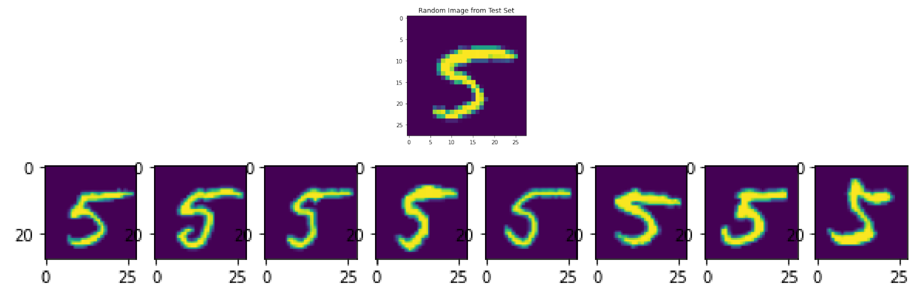
Figure 12: tSNE Plot

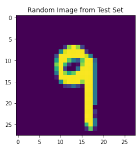
7 Closest Images:

Below I select four random images form the test set with feature vector x_0, then I find the eight images in the test set with closest feature vectors (using euclidean distance) to x_0. The results show that similar digits have similar features, since nearly all of the images with the top-eight closest feature vectors are the same digit. Only one exception is for the target digit “9”, where one of the top eight images is actually a “4”. This is understandable though, since “9” and “4” are quite similar and even their clusters in the 2D TSNE plot are overlapping.








Random Image from Test Set