Michelle Zhao

EE/CS 148 PS 3
MNIST Classification

Where the Code is Located

1. Code for my model (Net) is located in main.py
2. Code for loading MNIST (problem 3) is in Jupyter Notebook "Problem3_Setup_MNIST_data.ipynb"
3. Code for training ConvNet and doing data augmentation is in Jupyter Notebook "Problem5_Train_ConvNet_Model.ipynb"
4. Code for creating the best model and training is in Jupyter Notebook, "Problem6_Train_MyBest_Model.ipynb"
5. Code for retraining my best model using partial data is in Jupyter Notebook, "Problem7_Retrain_on_Partial_Data.ipynb"
6. Code for analyzing the network is in Jupyter Notebook, "Problem8_Analyze_Network.ipynb"
7. The Jupyter Notebook, "Working_Notebook.ipynb" is the one I used for running and testing all parts, and is where my "work" is. The other notebooks are cleaner, organized subsets of this notebook.

As usual, all code is pushed to https://github.com/mzhao98/caltech-ee148-spring2020-hw03.

3. Set up the MNIST Dataset

I randomly sample 15% of the training examples for each class to form a validation set, and leave the rest of the training data to form the training set. I set a seed of 0 so that the split doesn't change from run to run. Loading MNIST through Pytorch has pre-generated test and train sets, which I got by changing the flags. I set up a separate dataloader for the train, val, and test sets.

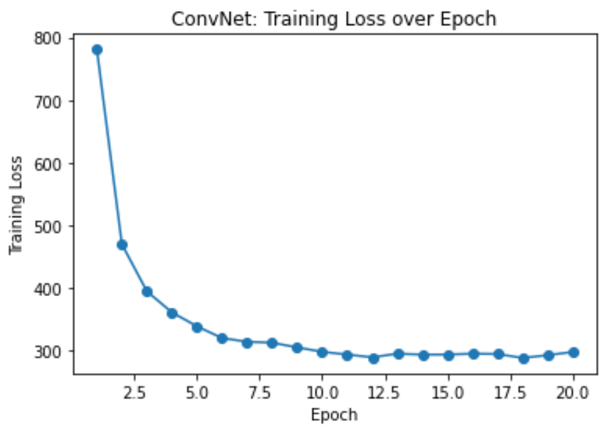
5. Train the default ConvNet and add data augmentation

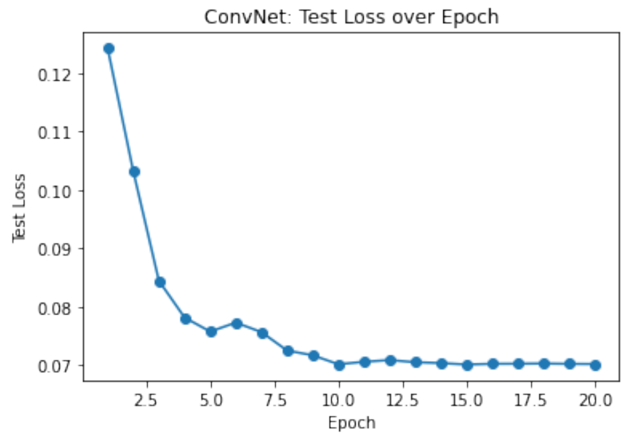
When training the default ConvNet, I got good performance, using negative log likelihood loss, which is a loss function useful for training a classification problem with C classes. The parameter set I used was the following:

batch_size = 32
epochs = 20
step = 1
test_batch_size = 1000
lr = 1.0
gamma=0.7
random seed = 1
log_interval = 10

I trained for 20 epochs, but the model converged around 10 epochs. Below is a graph of the training and test loss over epochs as the basic ConvNet trained. The model is saved as "mnist_model2.pt". The final performance on the entire training set and test set was as follows:

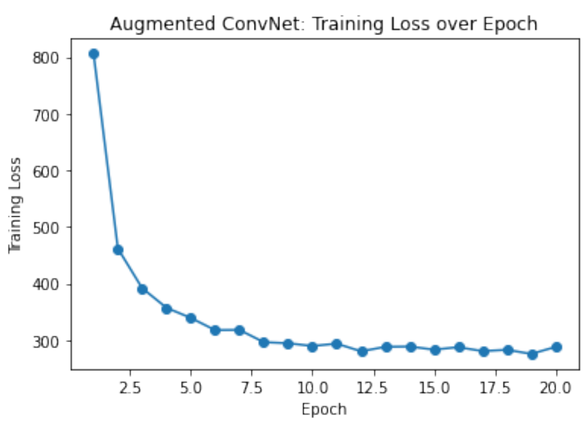
Training set: Average loss: 0.0607, Accuracy: 49970/60000 (83%)
Test set: Average loss: 0.0793, Accuracy: 9766/10000 (98%)

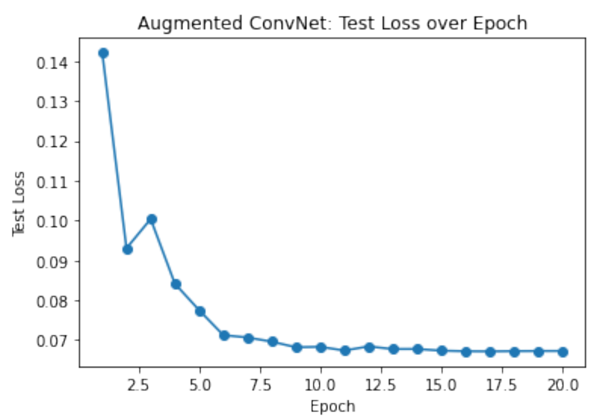




Then, I added data augmentation to the training data loader. For every training image, I blurred each input image using a Gaussian filter, and fed both the original image and the blurred image to the network. My reasoning behind this data augmentation was that the network should be able to classify an image even if it is blurry. To do the Gaussian blur, I zero-padded, and used a kernel of 5x5 size. The model is saved as "mnist_model2_aug2.pt". The final results were:

Training set: Average loss: 0.0608, Accuracy: 49930/60000 (83%)
Test set: Average loss: 0.0804, Accuracy: 9770/10000 (98%)

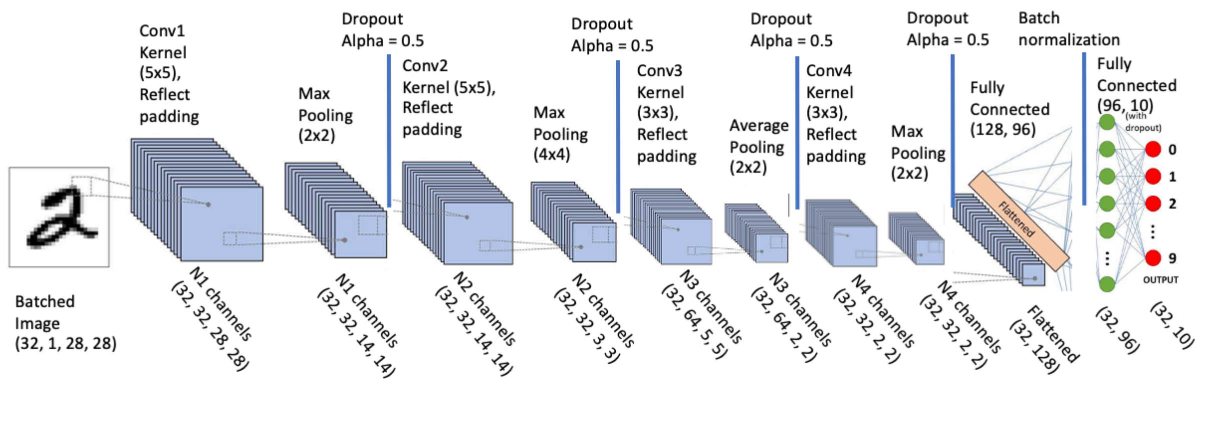




The data augmented network performed at about the same level of accuracy as the ConvNet without data augmentation. I hypothesize that this is because the Gaussian blur was not drastic enough of an alteration to the data set. Both models converged in training at around 12 epochs. One interesting difference is that there was a spike in the test error at epoch 3, likely due to the model not yet having mastered identifying the blurred data augmented images.

6. Build the best own MNIST Classifier you can

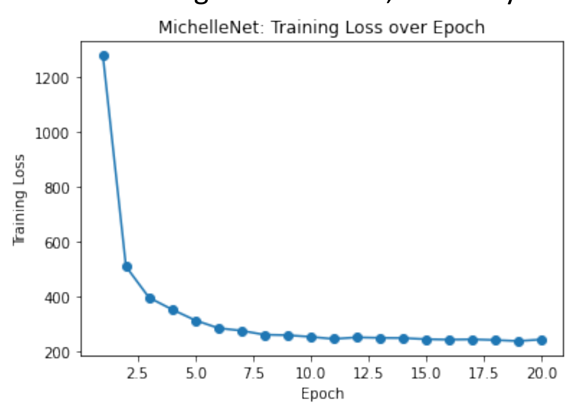
I built my own MNIST classifier to try to beat the basic and augmented ConvNets. Below is a diagram of the elements of my network.

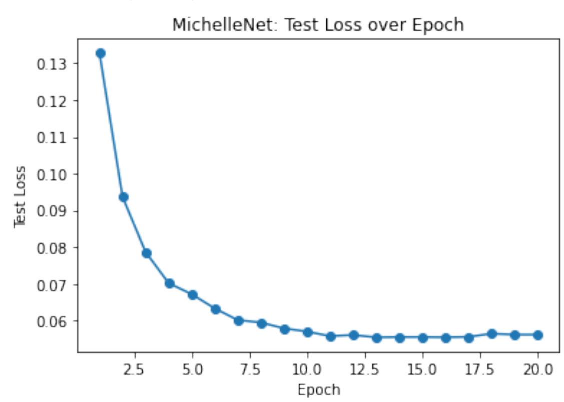


The reasoning behind my classifier network was that I wanted to perform more convolutional layers to learn a compressed, low-dimensional representation of the data, that would hopefully encode latent features, and then expand that low-dimensional encoding to a large fully connected network, and then finally make predictions. I trained the network using Cross Entropy Loss, instead of Negative Log Likelihood Loss, because cross entropy loss is also good for multiclass classification problems.

My MNIST classifier is saved as "mnist_model4_2.pt". The test and training results are as follows, as well as the graphs of the training and test losses during network training.

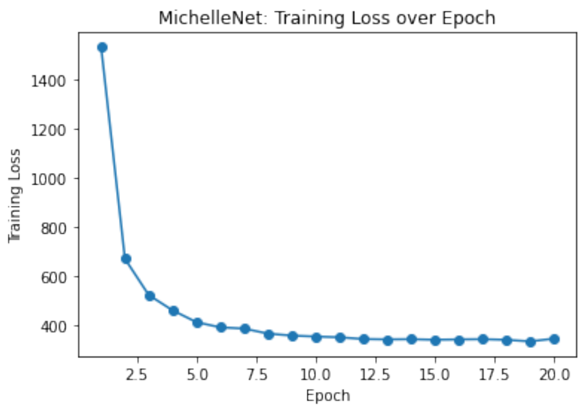
Training set: Average loss: 0.0454, Accuracy: 50180/60000 (84%)
Test set: Average loss: 0.0570, Accuracy: 9826/10000 (98%)

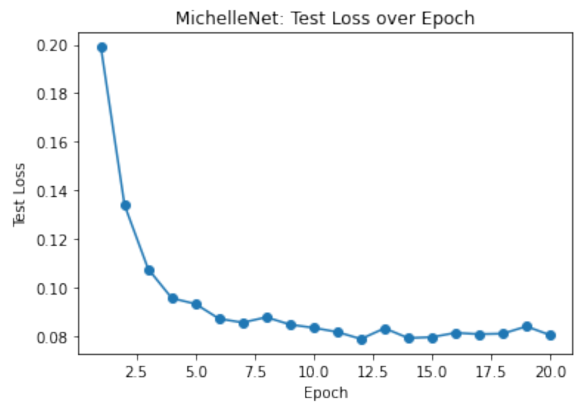




I performed data augmentation of my network by turning every image by 15 degrees using Pytorch's Random Affine transform. I also Gaussian blurred the original image. I fed both the Gaussian-blurred images, the turned images, and the original images to the network.

Training set: Average loss: 0.0322, Accuracy: 50410/60000 (84%)
Test set: Average loss: 0.0340, Accuracy: 9893/10000 (99%)





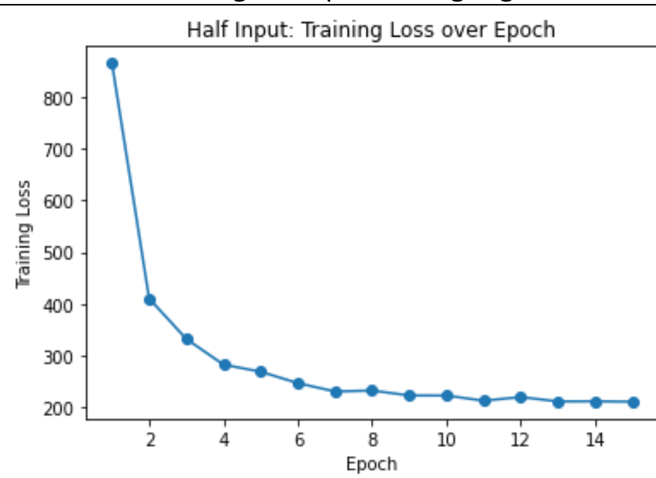
The model performed slightly better with the data augmentations, and got to 99% accuracy. We will refer to my augmented model at Net, which is the best model. This model is saved as "mnist_model4_aug.pt".

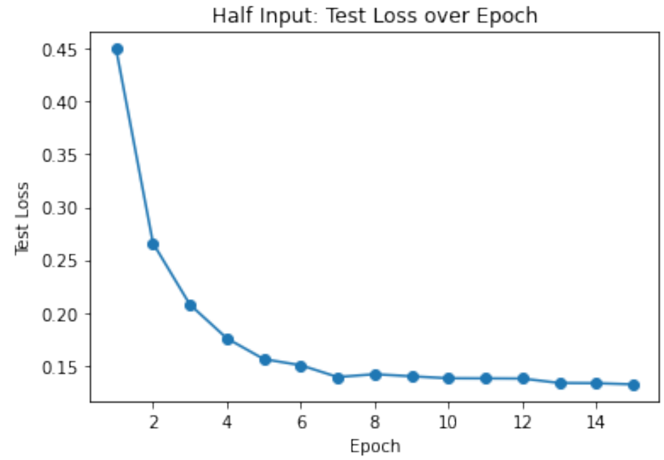
7. Once you have finished tweaking your network

a. FINAL RESULTS of Net:

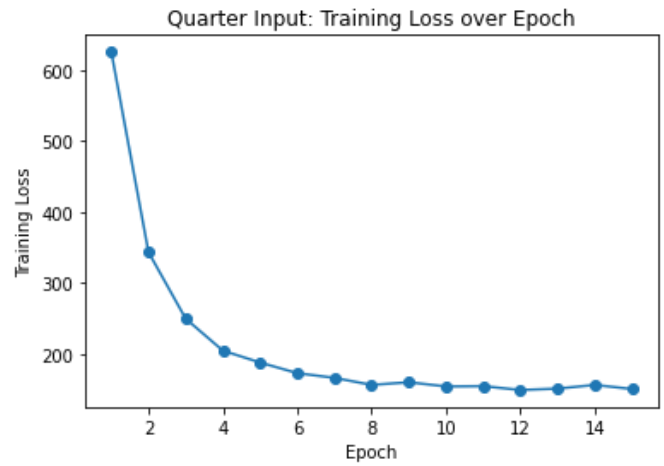
Training set: Average loss: 0.0322, Accuracy: 50410/60000 (84%)
Test set: Average loss: 0.0340, Accuracy: 9893/10000 (99%)

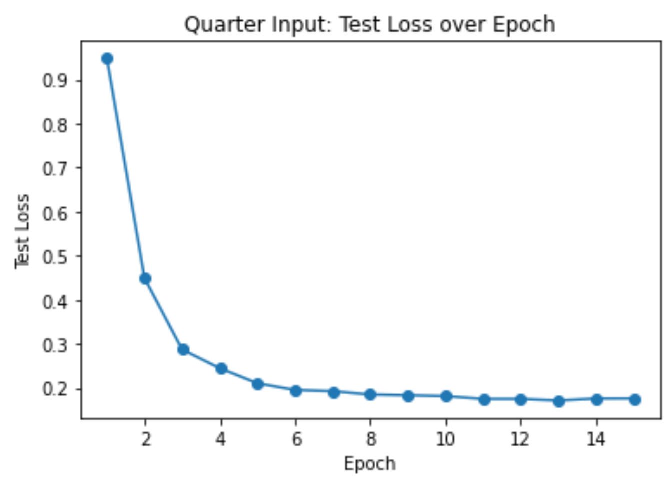
c. Re-train on a random subset of half, one quarter, one eighth, and one sixteenth of the training set and re-compute the accuracy values. Re-train on a random subset of 50% of the training set and re-compute the accuracy values. Report the results. Plot the training and test error as a function of the number of training examples on log-log scale.

Training Subset	Training/Test Results	Plots: training and test error as a function of the number of training examples on log-log scale
One half Model saved as: "mnist_model4_half.pt"	Training set: Average loss: 0.1006, Accuracy: 49148/60000 (82%) Test set: Average loss: 0.1050, Accuracy: 9674/10000 (97%)	

<table>
<tr><td></td><td></td><td>
</td></tr>
<tr><td>One quarter

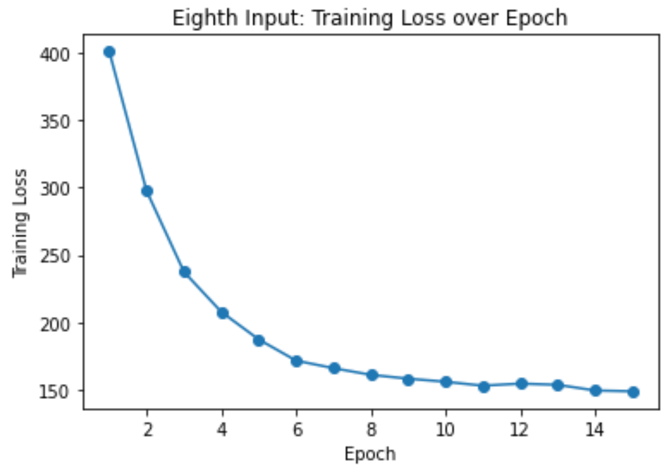
Model saved as: "mnist_model 4_quarter.pt"</td><td>Training set: Average loss: 0.1351, Accuracy: 48610/60000 (81%)

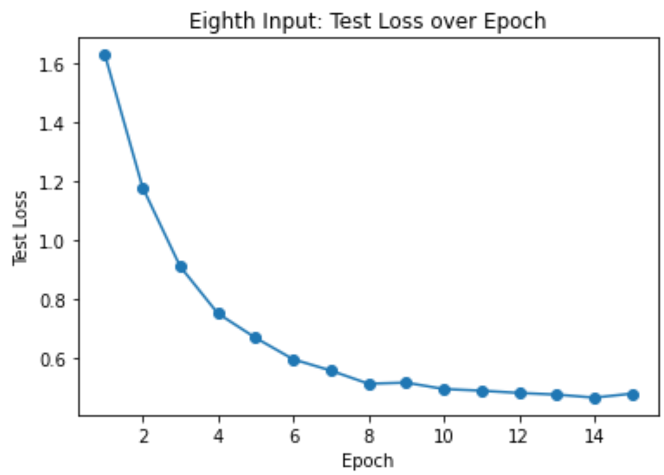
Test set: Average loss: 0.1426, Accuracy: 9560/10000 (96%)</td><td>


</td></tr>
</table>

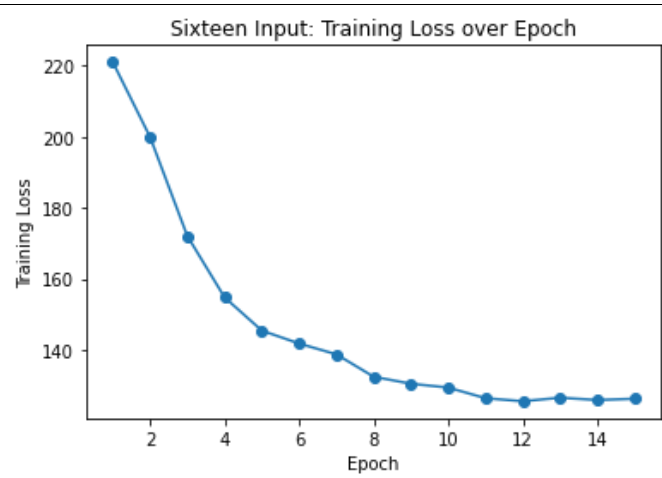
<table>
<tr>
<td>One eighth

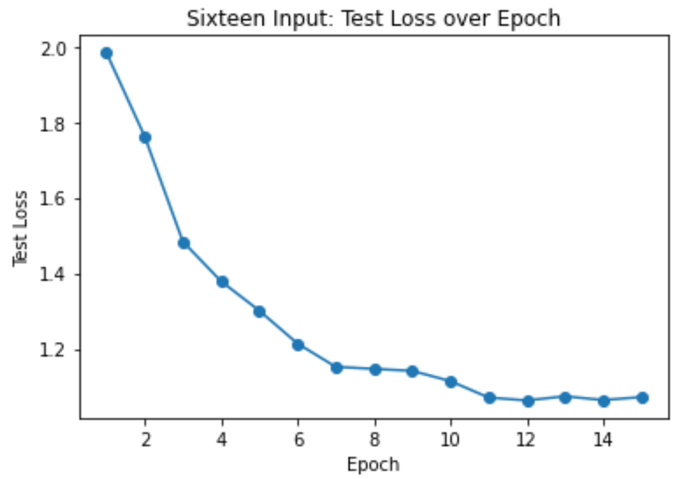
Model saved as: "mnist_model 4_eighth.pt"</td>
<td>Training set: Average loss: 0.4020, Accuracy: 45020/60000 (75%)

Test set: Average loss: 0.4559, Accuracy: 8902/10000 (89%)</td>
<td>


</td>
</tr>
<tr>
<td>One sixteenth

Model saved as: "mnist_model 4_sixteenth.pt"</td>
<td>Training set: Average loss: 0.9592, Accuracy: 33276/60000 (55%)

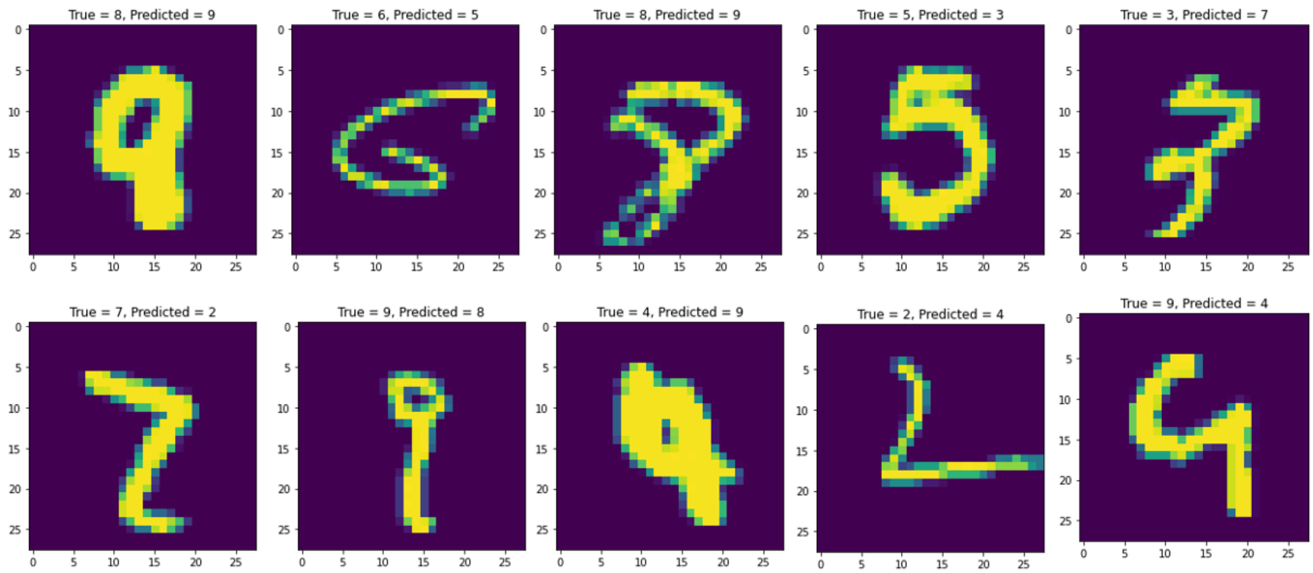
Test set: Average loss: 1.1196, Accuracy: 6586/10000 (66%)</td>
<td>
</td>
</tr>
</table>

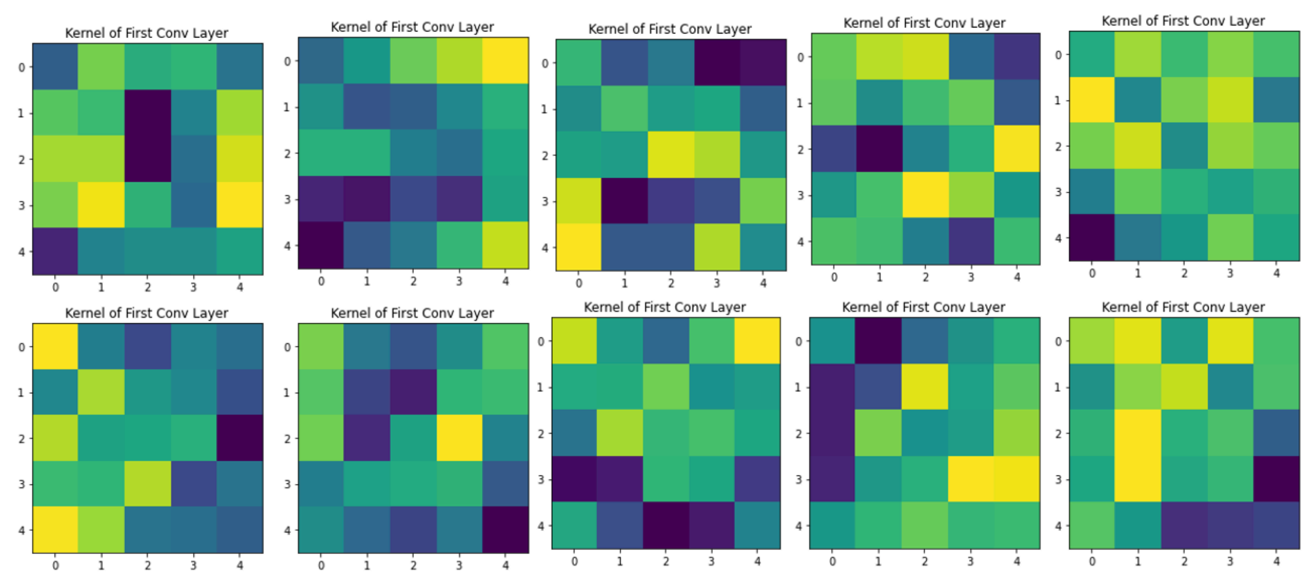
I notice that using diminishing fractions of the input leads to worsening performance of the model. Using half and one quarter of the input leads to decent results, with test accuracy near 90%. However, one eighth and one sixteenth of the input led to pretty bad performance. This tells us that we don’t need all of the data to train a decent model, but the more the better.

8. Analyze what the network has learned.

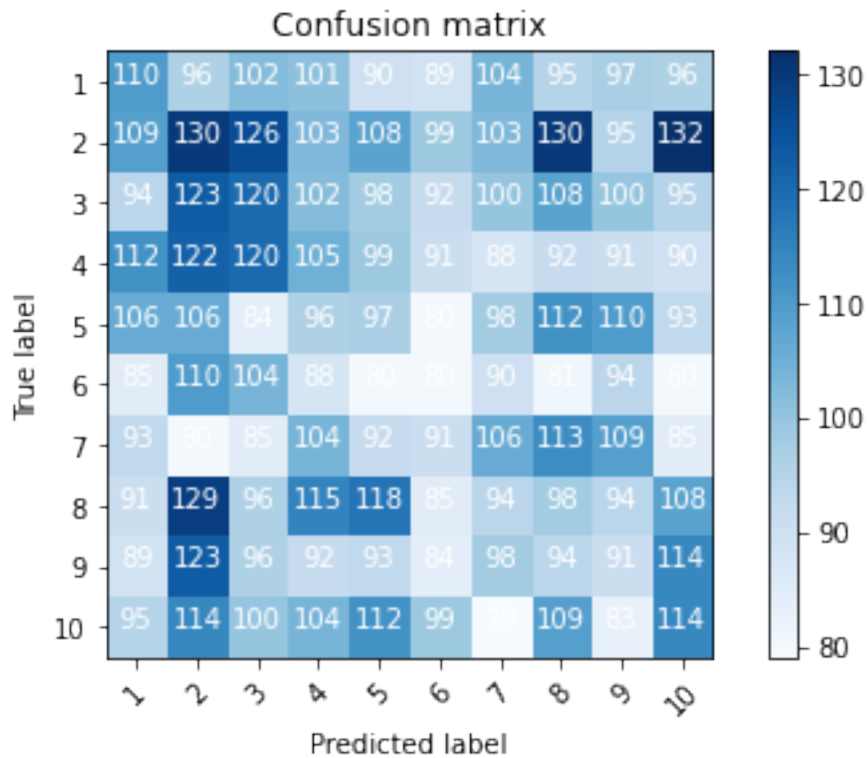
b. Below we present at least 9 examples from the test set where my best classifier, Net, made a mistake. Looking at the results, the mistakes make sense, because a lot of these numbers look very similar to the numbers that the network predicted, so a human might even be fooled.



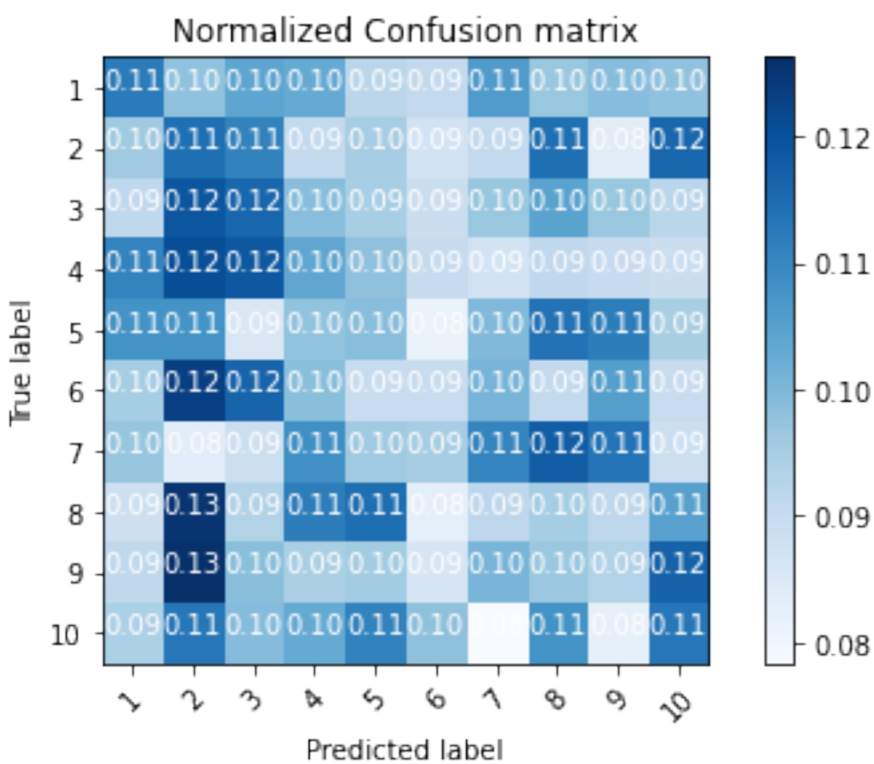
c. Visualize at least 9 of the learned kernels from the first layer of your network.



d. Generate a confusion matrix for the test set.

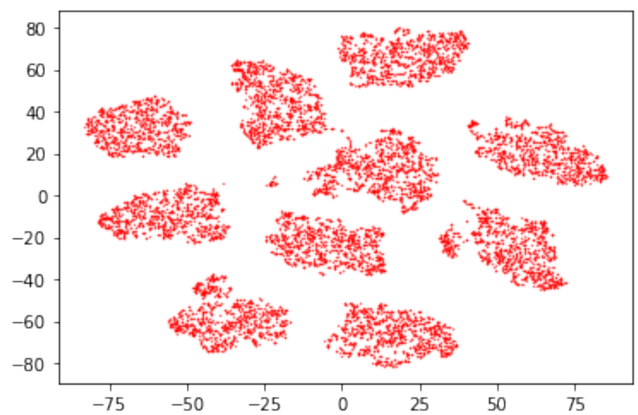


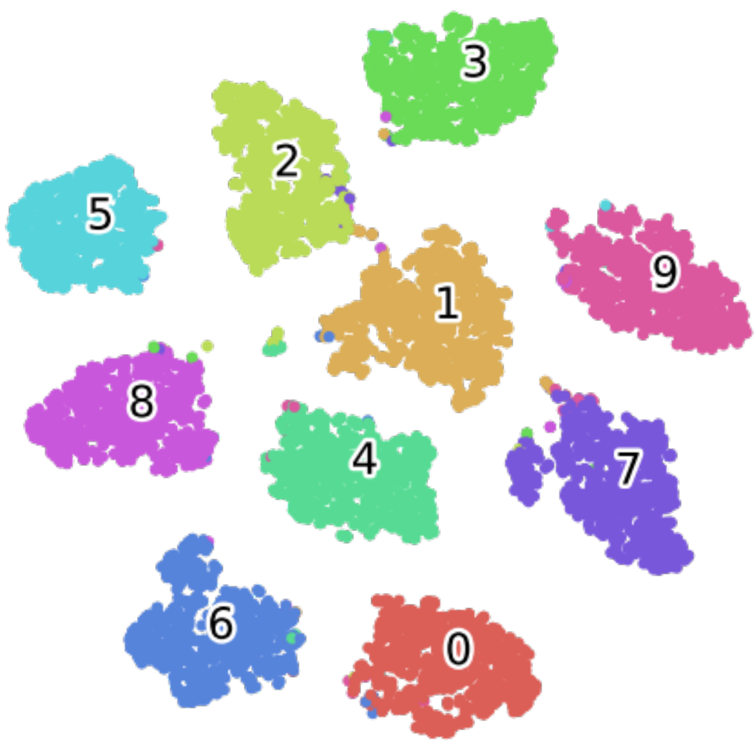
I also created a normalized confusion matrix, normalized over rows.



e. Use your network to convert each image in the test set into a feature vector (taken from just before the final linear layer).

i. Below we visualize this high-dimensional embedding in 2D using tSNE (each class has its own color). The red isn't differentiated by color, but gives us a shape of the what the space looks like.





I see that in this high dimensional embedding, each of the classes has been split up. There are a few outlier data points that look to be in the wrong group, but the high dimensional embedding has learned the latent features to split up the 10 classes.

ii. Choose one image I0 with feature vector x0 from the test set. Find the 8 images I1, I2, …, I8 in the test set whose feature vectors are closest in Euclidean distance to x0. Repeat this process for at least 3 more choices of I0. Present your results in an nx8 grid of images (where n is at least 4). Discuss what you see.

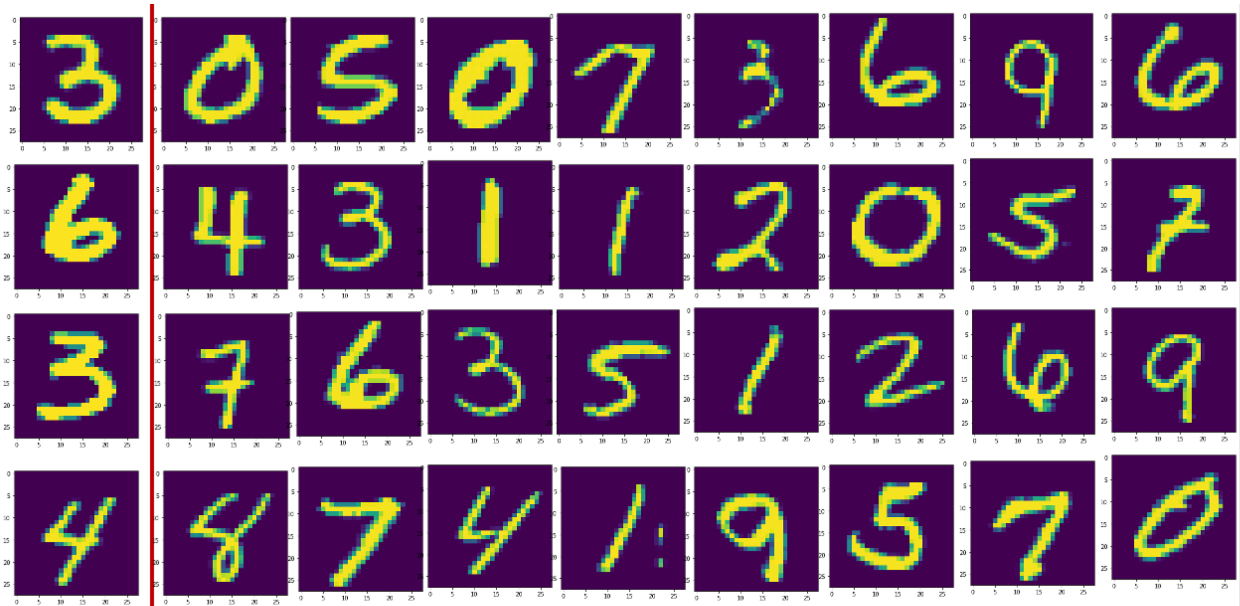


image I_0 with feature vector x_0 from the test set

8 images I1, I2, …, I8 in the test set whose feature vectors are closest in Euclidean distance to x0.

The images that are the closest in Euclidean distance to x0 are not as similar to the actual number (true label) of the original image. This shows that simple Euclidean distance, as can be done with clustering, is not sufficient for the multiclass classification of digit recognition. The Euclidean distance does not capture enough information about each of the digits to make accurate classifications, and thus, we need a high-dimensional embedding created by the CNN to make the correct classifications.