Github

Repo: https://github.com/KevinHuang8/caltech-ee148-spring2020-hw03

- `main.py` - Contains model architecture and training
- `visualize.ipynb` - Contains all the code for visualization, including plots

All models were trained with a batch size of 64 and for 14 epochs with early stopping. That is, we stop training as soon as the test loss increases from the test loss from the previous epoch.

Data Augmentation

Training the default `ConvNet` without data augmentation results in 98% accuracy on the validation set.

Our data augmentation scheme is to perform a random transformation to each image every epoch, consisting of the following:

- A random rotation of the image from -5 to 5 degrees
- A random translation of 1 pixel in the x and y directions
- A random shear of -10 to 10 degrees along the x and y directions

With data augmentation, training results in a 96% validation accuracy.

The reason the accuracy decreases is likely because the transformations change the images to be varied enough that it becomes more challenging for the network to learn. This method keeps the same total number of samples in the dataset, rather than adding transformed versions to the dataset. Thus, increasing the variance in the data while keeping the number of samples fixed likely contributed to the slight decrease in accuracy. However, the accuracy difference was small enough that it might simply be due to noise.

Network Details

The best architecture was a convolutional network similar to `ConvNet`, but with batch normalization instead of dropout and 5x5 kernels. No data augmentation was used. This network correctly classifies 8893 out of the 8895 samples (99%) in the validation set after training.

Most networks attempted performed well, achieving 98-99% accuracy, meaning that most of the modifications I tried had very minor impacts on performance. In fact, any difference might simply be due to random chance.

A simple neural network with 3 fully connected layers of 128 features each (no convolutional layers, no dropout or batch normalization) also preformed very well, with a 98% accuracy, barely worse than the convolutional networks. The reason for this is likely because the MNIST dataset is relatively large, while the images themselves are fairly simple. Thus, a simple fully connected network is able to learn efficiently, and convolutional layers may not be necessary. Adding an extra layer or removing an extra layer does not seem to have much impact on performance. Similarly, increasing the number of channels also does not seem to affect the performance much. However, batch normalization does perform better than dropout in nearly all cases, although the improvement is slight (about 1% in accuracy).

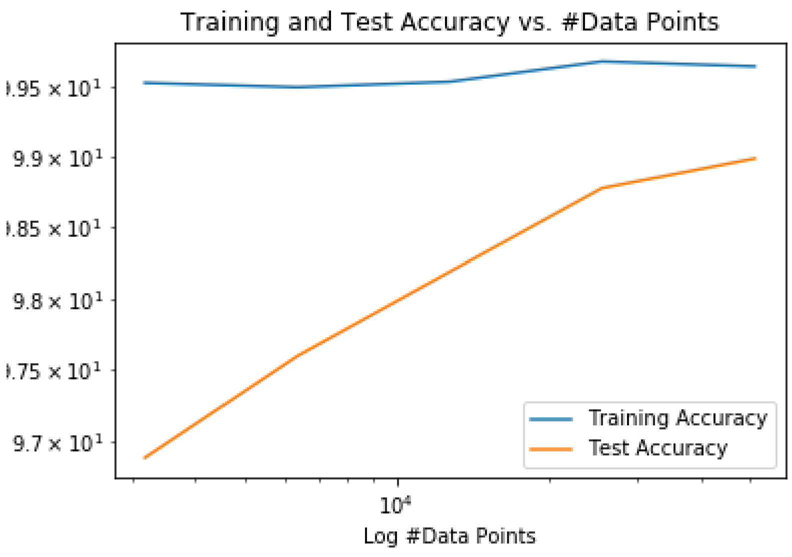
Overall, it seems like any network with enough parameters is able to perform well in classifying the MNIST dataset, and any performance differences are minor. This is most likely because MNIST is relatively "easy" dataset, with an abundant number of samples.

Results

With the best performing model described above, the results are

- Training accuracy: 99.57%
- Test accuracy: 98.99%

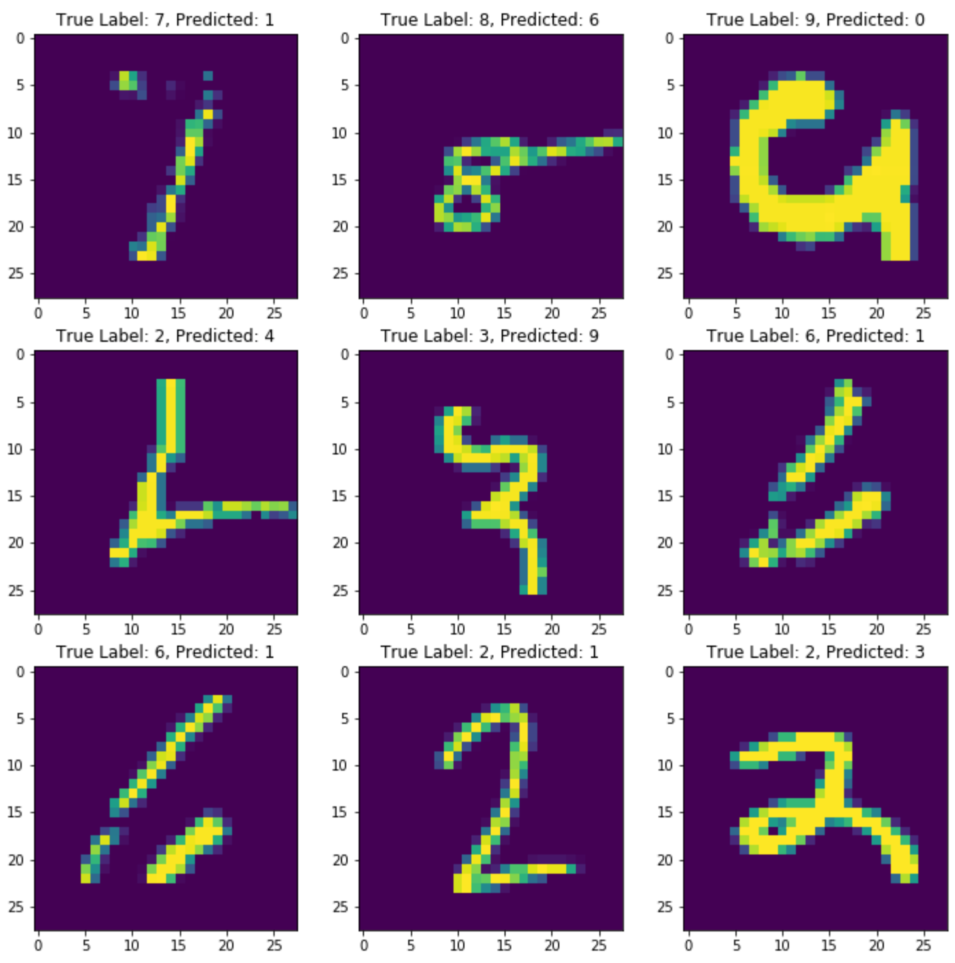
After varying the number of data points used for training, we obtain the following plot:



As we can see, the more data points we have the greater the test accuracy is. We have a log-log plot, and the relationship is roughly linear, meaning that as we decrease the number of data points, the test accuracy falls off roughly exponentially. However, the training accuracy does not change depending on the amount of data, and is always close to 100%. This means that our model is complex enough to capture the data fully.

Misclassified Examples

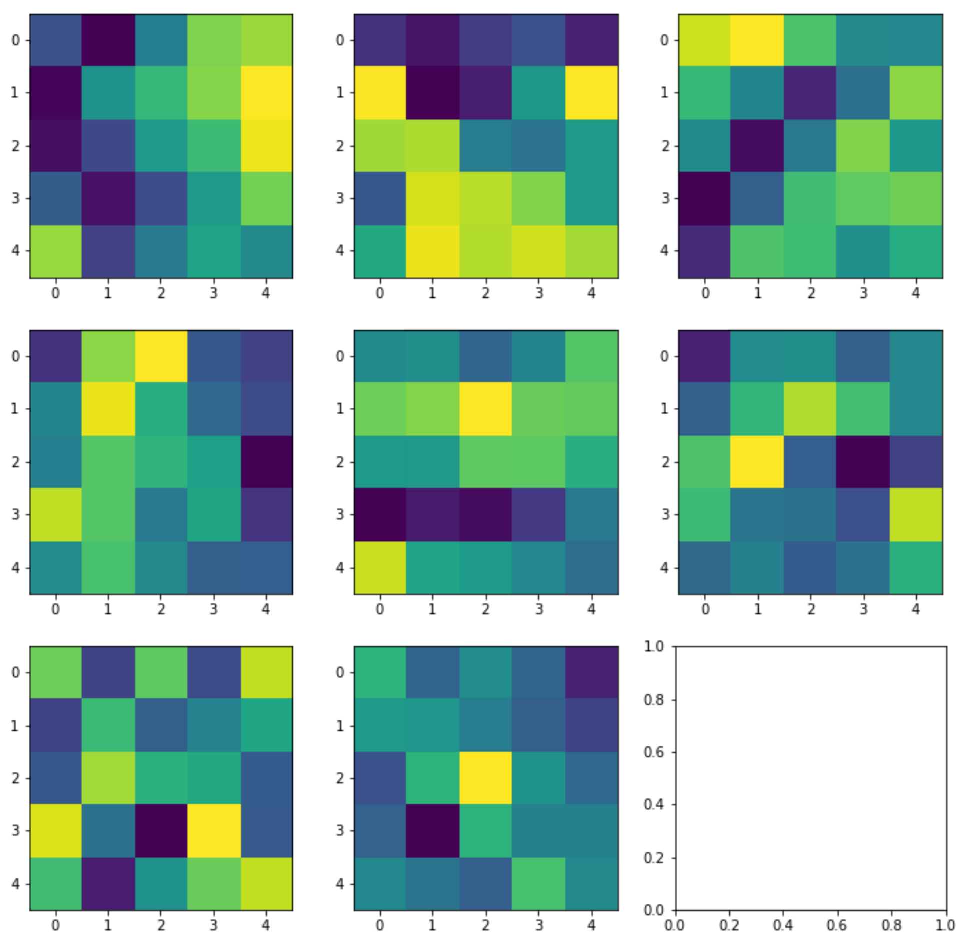
Here are 9 examples that were incorrectly classified by our network:



As we can see, some of these examples are challenging for even a human to classify, as many of them are poorly cropped, or poorly drawn. There are two 6's which are drawn such that the loop is collapsed into a single line, causing the network to identify them as 1's. The number 2 can also easily been drawn in such a way that it appears like the top of a 3 (when the final stroke is exaggerated, like the bottom right image), or it can appear similar to a 1. Overall, the network still gives reasonable guesses for most of the misclassified digits, and it is easy to imagine humans making the same errors.

Kernel Visualization

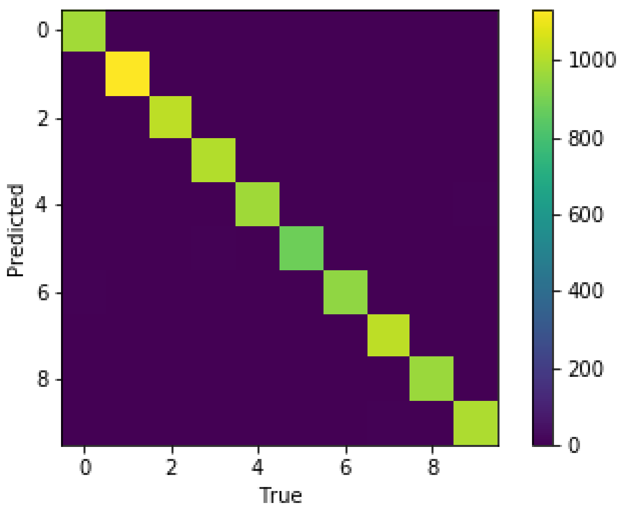
Here are the 8 kernels from the first convolutional layer:



Note that we only have 8 kernels in our first layer.

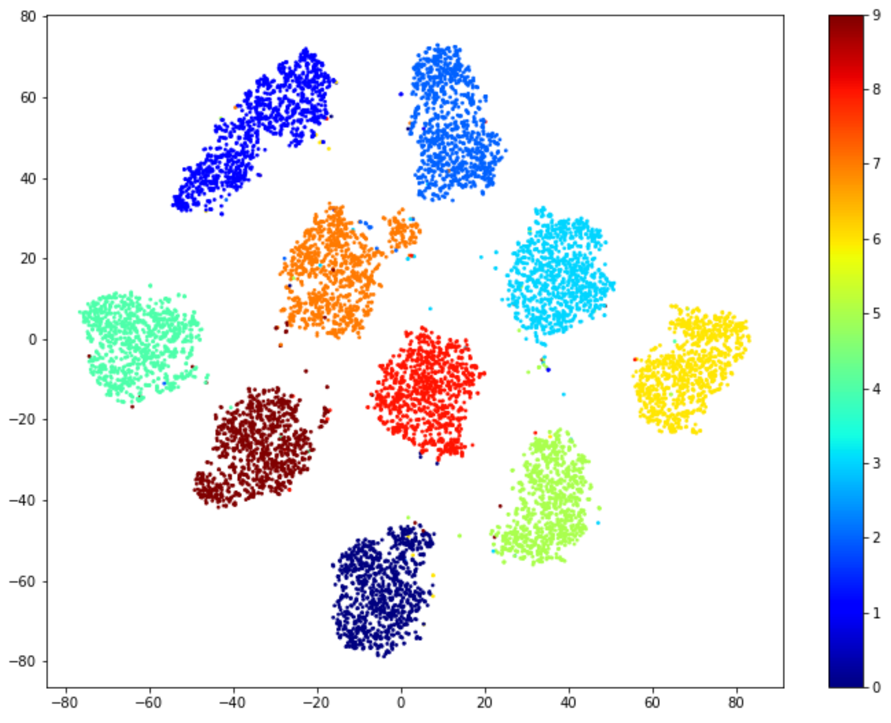
It is difficult to discern what each kernel represents by looking at them, but it seems like each kernel is recognizing a different aspect or feature of the images. For example, each kernel seems to have bright pixels at different locations than the other kernels, signifying that they are each learning something separate from each other. Vague, patterns like lines and edges can be made out, so that could be one interpretation of what the network is learning. It would probably be easier to interpret larger kernels.

Confusion Matrix

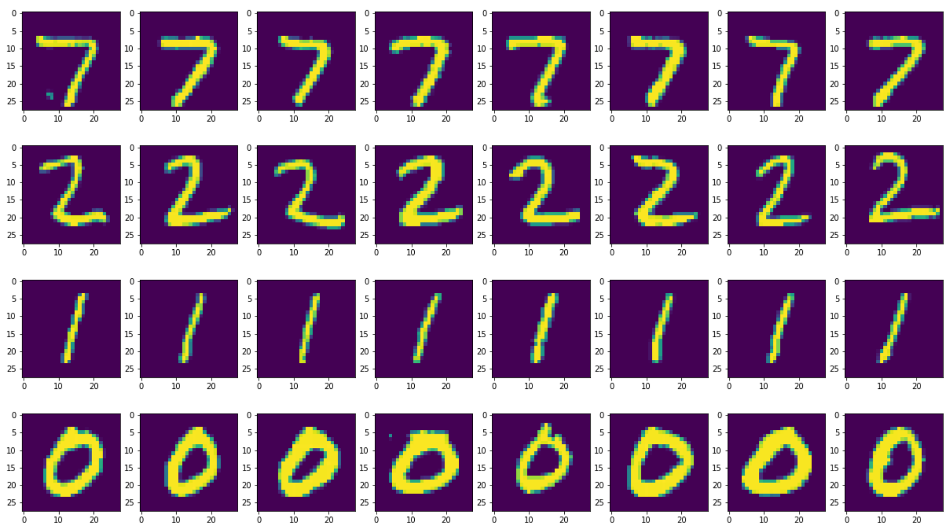


From this confusion matrix, we can see that all of the entries not along diagonal are close to 0, meaning that we misclassified very few images. Because we see no particular entries that stand out, there's not any one particular class that we are having trouble with, or any one particular class that tends to be classified as another class.

Feature Vector Visualization



We can clearly delineate the different classes. This means that the network was successful in classifying the digits, as each class has a high dimensional representation that is fairly unique from the other classes. From this image, we can also see numbers that can look similar to each other. For example, we see that '4' and '9' are close together, which matches our expectations since written 9's can easily look like 4's and vice versa. Indeed, we can also see that a few 9's are misclassified as 4's. From this image we can also see the points that are misclassified. They are either in between the clusters, meaning that these images do not match/fit into the network's understanding of the classes nicely, or are totally misinterpreted as part of a different class.



Each row of this image corresponds images that are close together in their feature representation. From this image, we can see that images with similar feature representations are very close together in appearance. This makes sense, as the network treats similar looking images in a similar manner. This also confirms that our model is working, as inputs with similar feature representations are generally all of the same class.