EE/CNS/CS 148 Homework 3 Report

Albert Zhai

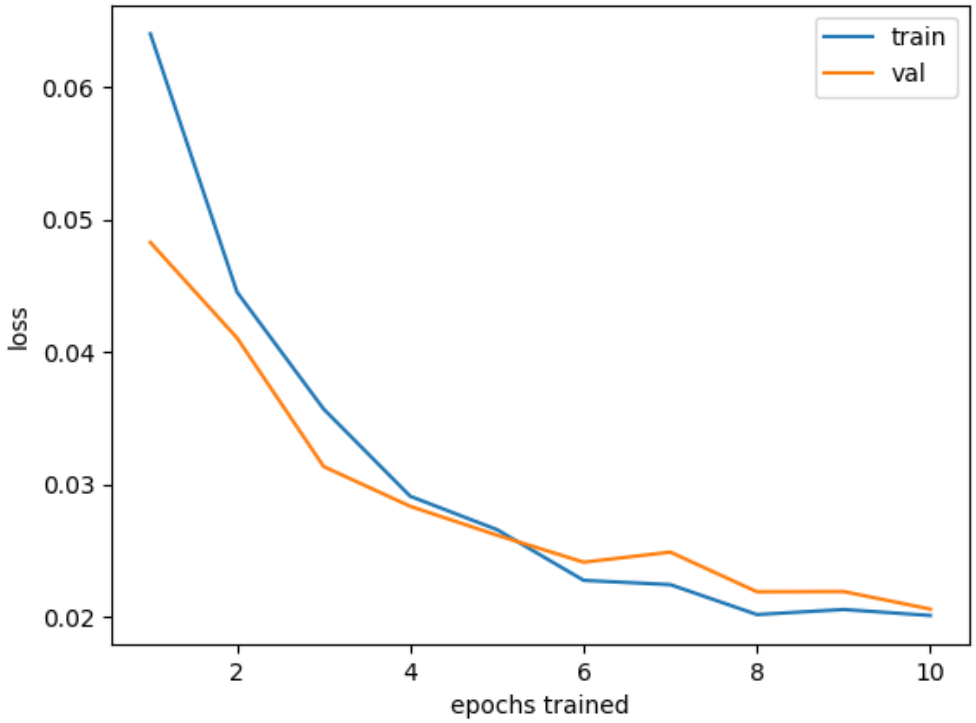
4 May 2020

The default ConvNet was trained for 10 epochs using an Adadelta optimizer with learning rate 1.0, decaying by 30 percent every epoch. The training set accuracy was 97.75 percent, and the validation set accuracy was 97.26. Random rotations from -10 to 10 degrees were then applied to the training images as data augmentation, and the ConvNet was trained for 12 epochs. The resulting training set accuracy was 97.30, and the validation set accuracy was 97.26. The training accuracy decreased because the added variance made it more difficult to fit to the effective training distribution. The validation accuracy could have increased if the augmentation helped the network learn a significantly more general pattern, but the effect ended up cancelling out with the slight distributional shift and difficulty in fitting.

To improve the classifier, I generally tuned one parameter at a time, perturbing it by a small amount and then changing in the direction of improvement. I stopped when the improvement in validation set performance seemed to be less than the variability induced by randomness in the training procedure. In a sense, I was roughly descending along partial derivatives of the validation accuracy with respect to each parameter. I found all of the following effective: increasing layer widths, adding a third convolutional layer, and adding batch normalization. Data augmentation was helpful once the network became fairly large. Changing the optimizer or number of epochs was not. My final network architecture, achieving 99.5 percent test accuracy, was the following:

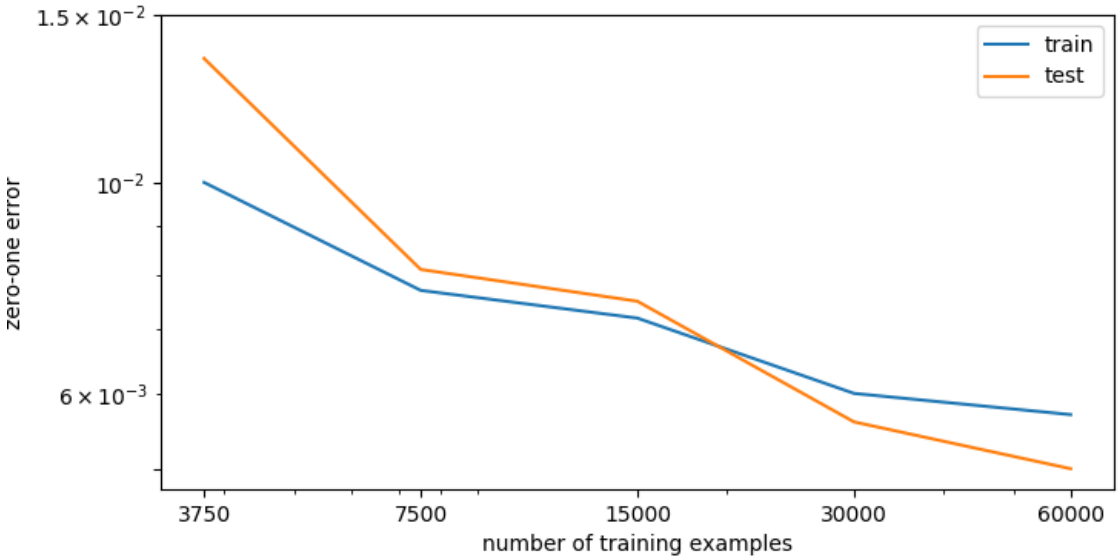
```
self.conv1 = nn.Conv2d(in_channels=1, out_channels=64, kernel_size=(5, 5), stride=1)
self.conv2 = nn.Conv2d(64, 64, 3, 1)
self.conv3 = nn.Conv2d(64, 128, 3, 1)
self.bn1 = nn.BatchNorm2d(64)
self.bn2 = nn.BatchNorm2d(64)
self.bn3 = nn.BatchNorm2d(128)
self.dropout1 = nn.Dropout2d(0.4)
self.dropout2 = nn.Dropout2d(0.3)
self.dropout3 = nn.Dropout2d(0.2)
self.fc1 = nn.Linear(128 * 9, 256)
self.fc2 = nn.Linear(256, 10)
```

I used the given Adadelta optimizer and scheduler, training the network 10 epochs. Data augmentation consisted of random rotations of up to 10 degrees, random translations of up to 5 percent, and random brightness jittering of up to 10 percent. The history of the training and validation loss is shown below:



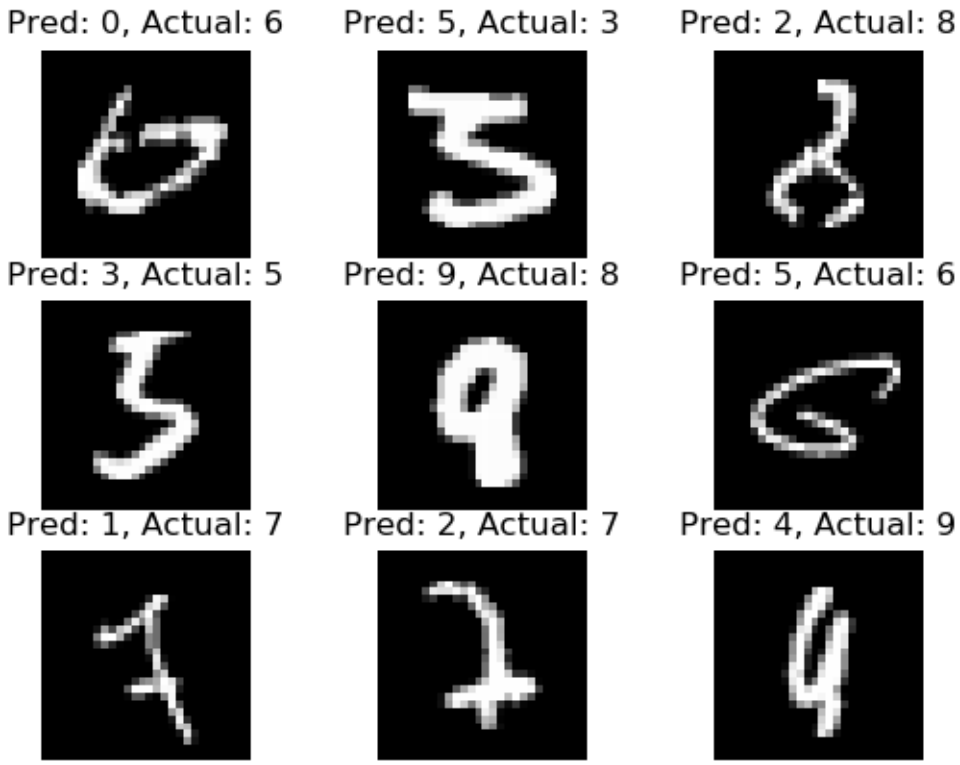
The two curves remain very close, indicating that there is little overfitting.

The following plot of training and test error versus number of training examples was produced:



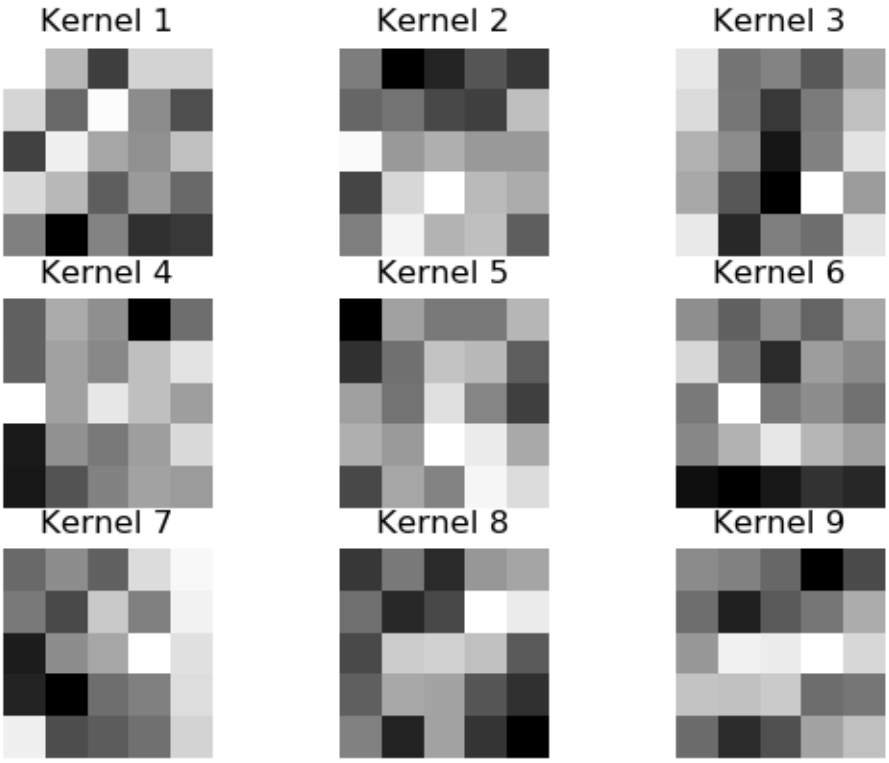
The test error decreases because there is more data to learn a general pattern from. However, the training error also decreases, perhaps due to the fact that there are more gradient steps overall in the 10 epochs, causing tighter convergence.

The following plot shows examples where the network made a mistake:



In almost every case, the appearance of the image indeed looks like the predicted digit through the lens of human perception, so the mistakes are "acceptable" in the sense that a slightly careless human could have made them. The only counterexample is the image in the center right, which is clearly a very thin and flat 6. The network most likely failed to recognize the pattern due to the orientation and sharpness of the curves. This could probably be fixed with further data augmentation.

The following plot shows examples of kernels learned in the first layer:



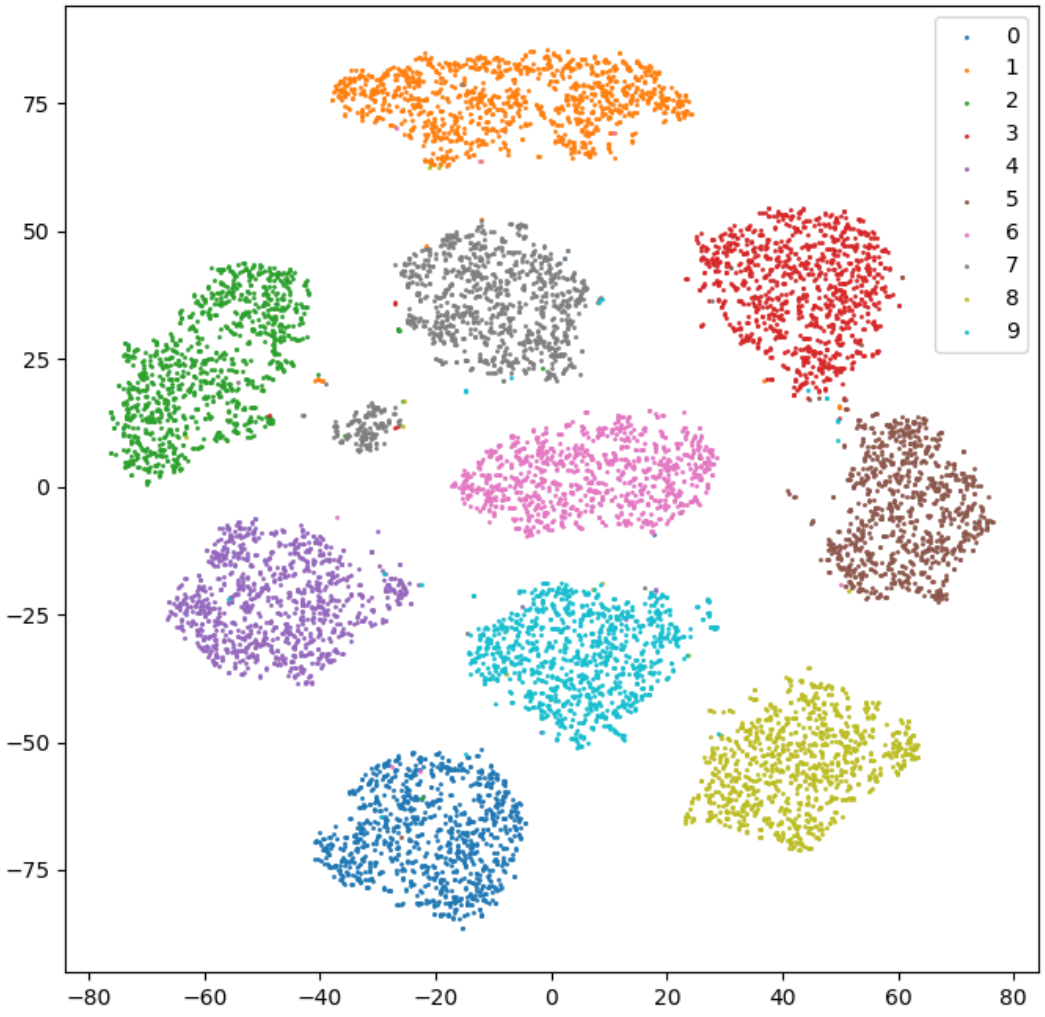
It is difficult to interpret using natural language what patterns are being matched by these filters. Kernel 3 looks like it may function somewhat as a diagonal edge detector, but generally we cannot really understand how the later layers of the network are using its output.

The following confusion matrix was produced:

True \ Predicted	0	1	2	3	4	5	6	7	8	9
0	979	0	0	0	0	0	0	1	0	0
1	0	1133	0	2	0	0	0	0	0	0
2	1	0	1028	1	0	0	0	2	0	0
3	0	1	1	1006	0	1	0	0	1	0
4	0	0	0	0	978	0	0	1	0	3
5	1	0	0	6	0	883	1	0	0	1
6	3	4	1	0	1	1	947	0	1	0
7	0	2	2	0	0	0	0	1023	0	1
8	0	0	2	0	0	1	0	1	969	1
9	0	0	0	0	3	1	0	1	0	1004

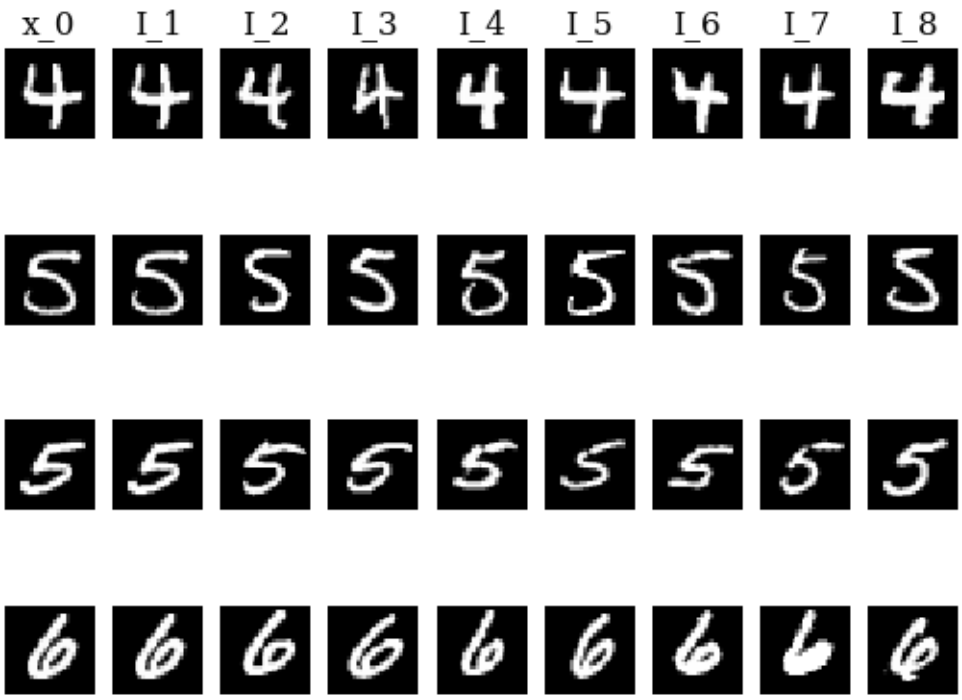
There is no noticeable trend here since the mistake counts are all so low. The most common failure cases appear to be confusing 5's for 3's and 6's for 1's.

The following tSNE scatter-plot was produced:



Each digit class occupies a very distinct, separate region of the embedding space. Even when the network is wrong, it is highly confident about its prediction. We can also see here that there seems to be two modes of 7's, probably coming from whether it was written with a horizontal dash in the middle or not. The 3's are close to the 5's, and the 6's are far away from the 0's and 8's. The 4's happen to lie in the center of all of the digits.

The following plot shows examples of images and their 8 nearest neighbors in the embedding space:



All of the neighbors perceptually look extremely similar to the original image. Not only is the digit class the same, the handwriting style and orientation are also almost identical. This suggests that the convolutional network has learned a feature representation which is at least somewhat comparable to our high-level neuronal representations.

The GitHub repository is located at https://github.com/ajzhai/caltech-ee148-spring2020-hw03. The code for training the network is in the "main.py" file, and the code for generating visualizations is in the "visualize.py" file.