Ryan Frohman

Project 2 Report

Part 1: Linear SVM

For this part, I used the python sklearn module to implement the SVM classifier as well as tensorflow to load in the MNIST dataset. To make the SVM linear, I called the svm() function with the argument kernel = 'linear.' Another adjustable value was the C, regularization parameter. I found that using a value of C equal to 1 ended up in the best results. Shown below is a table of different values of C and the corresponding accuracies of the SVM classifier.

C = 1	C = 5	C = 20
Accuracy when classifying 0: 97.65	Accuracy when classifying 0: 97.244	Accuracy when classifying 0: 97.142
Accuracy when classifying 1: 98.85	Accuracy when classifying 1: 98.502	Accuracy when classifying 1: 98.590
Accuracy when classifying 2: 93.70	Accuracy when classifying 2: 92.344	Accuracy when classifying 2: 91.763
Accuracy when classifying 3: 93.76	Accuracy when classifying 3: 94.059	Accuracy when classifying 3: 92.970
Accuracy when classifying 4: 95.92	Accuracy when classifying 4: 94.908	Accuracy when classifying 4: 94.806
Accuracy when classifying 5: 90.02	Accuracy when classifying 5: 87.780	Accuracy when classifying 5: 87.668
Accuracy when classifying 6: 94.98	Accuracy when classifying 6: 93.841	Accuracy when classifying 6: 92.797
Accuracy when classifying 7: 93.09	Accuracy when classifying 7: 91.828	Accuracy when classifying 7: 91.245
Accuracy when classifying 8: 90.04	Accuracy when classifying 8: 88.603	Accuracy when classifying 8: 88.193
Accuracy when classifying 9: 91.37	Accuracy when classifying 9: 90.683	Accuracy when classifying 9: 89.098

It seems that as the regularization parameter gets higher, the SVM model prioritizes correctly classifying the training data, possibly resulting in overfitting. The values of C shown above all have high success, but the C = 1 value seems like there is more "wiggle room" when the test data is used on the SVM classifier.

Overall, part 1 resulted in relatively minor issues and the ones that occurred resolved themselves rather quickly. The SVM classifier ended up having quite large success, with the average classification for the 10 different digits being at least 88-90%, with numerous being 95%.

Part 2: Deep Learning

For this part, I used the starter code given to us by Professor Babadi. This part of the project is where the major experimenting began. The parts I tested out were different layer configurations, different activation and loss functions, and different optimization methods. The first to be discussed is the type of layering.

In choosing the number of epochs to use, I found that 2 epochs was the optimal amount. Any less and the CNN seemed like it would not have enough training to classify as accurately. It was evident during the second epoch that accuracy jumped up from roughly 85% to 96+%. Any more epochs and the accuracy slowly started to dwindle, resulting in possibly overfitting and lessened accuracy on the test data.

In designing the CNN, I first ran the original code on the MNIST dataset without changing anything. The resulting accuracy for both training and test data was quite high, ranging from 96 to high 97 percent accuracy. I then tried removing a convolutional/pooling layer and running the CNN again. To my surprise, I still ranged around 96 to 97 percent accuracy. Lastly, I tried adding in an extra convolutional layer instead of removing one. This is where I first observed a 98 percent accuracy, shown below. It appears to me that adding a third convolutional layer actually decreased the accuracy a bit, perhaps distorting key features.

Removing a layer	No changes to layers	Adding a layer
Train accuracy: 0.9745833277702332 Test accuracy: 0.9731000065803528	Train accuracy: 0.9868166446685791 Test accuracy: 0.9824000000953674	Train accuracy: 0.9585999846458435 Test accuracy: 0.957099974155426

Next, I changed the optimization method. Out of SGD, Adam, and Adagrad, I found that Adam consistently gave me the highest accuracy. Shown below is a table of the various optimizers and their accuracies:

SGD	Adam	Adagrad
Train accuracy: 0.8780500292778015 Test accuracy: 0.885200023651123	Train accuracy: 0.9742666482925415 Test accuracy: 0.9778000116348267	Train accuracy: 0.5800999999046326 Test accuracy: 0.5841000080108643

Next, I experimented with the loss function. Of cross entropy, mean squared error, and hinge, I found that cross entropy by far gave me the best results, as shown below in the table below. It was quite apparent that both Mean Squared Error and Hinge gave quite inaccurate results in tandem with the other.

Mean Squared Error	Cross Entropy	Hinge
Train accuracy: 0.10010000318288803 Test accuracy: 0.10130000114440918	Train accuracy: 0.9723333120346069 Test accuracy: 0.9724000096321106	Train accuracy: 0.097933329641819 Test accuracy: 0.10159999877214432

The last parameter that I experimented with was the activation function of the various layers. The ones I considered were a combination of softmax with hyperbolic tangent, ReLu, and sigmoid. The reason I combined everything with softmax was to have a softmax output for the classical neural network part CNN. Shown below in the table are the classification accuracies for the various values. Based on the table, it appeared that the combination of tanh and softmax was the best activation function to use with my setup. ReLu and softmax was also another fantastic combination and the two could arguably be used interchangeably, but for my runs it appeared that tanh was just a tiny bit higher than ReLu.

tanh	ReLu	Sigmoid
Train accuracy: 0.9795666933059692 Test accuracy: 0.9797000288963318	Train accuracy: 0.9723333120346069 Test accuracy: 0.9724000096321106	Train accuracy: 0.8744833469390869 Test accuracy: 0.8804000020027161

Overall, the convolution neural network was a success. With an average accuracy of high 96 to 98 percent, the parameters I experimented with and optimized allowed for me to have the best possible success.