Zain Chalisa, Akash Shah, Rishi Ajjarapu
zc285, ass209, ra864
Intro to AI - Final Project

Perceptron:

Perceptron Algorithm Explanation

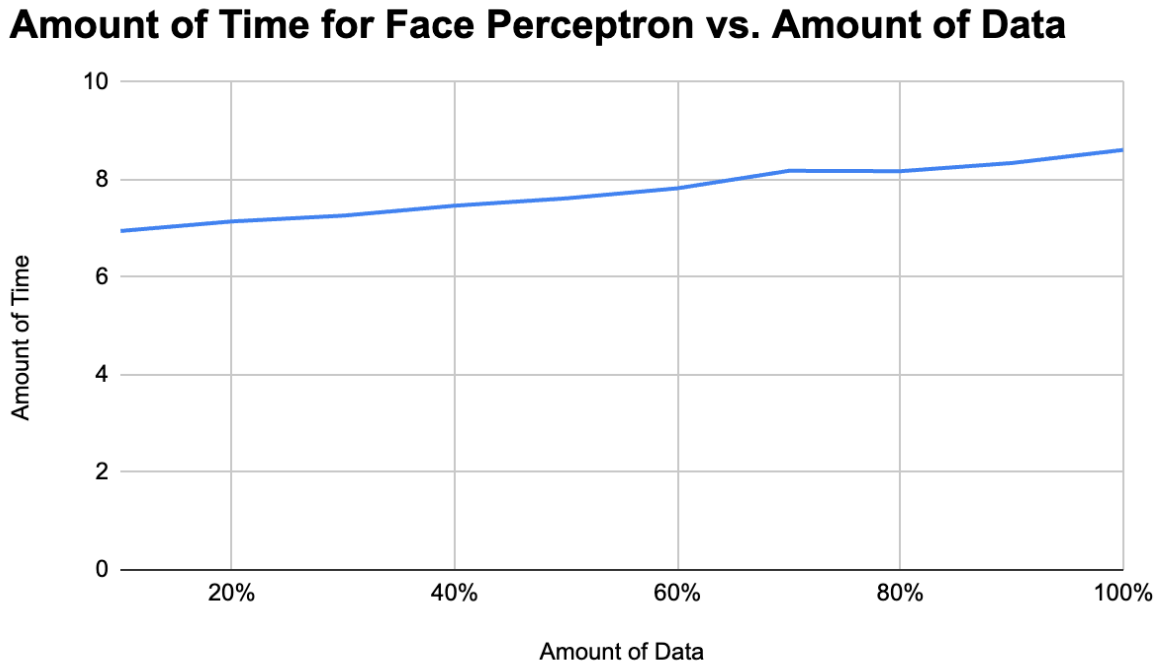
For Perceptron, we treated each pixel as a feature, and constructed 2D arrays out of them as our input having 60 rows by 70 columns. We then fed the input through our Perceptron, multiplying the features by their respective weights which were built to also be a 60 by 70 2D array, keeping track of a running sum. We took this sum and passed it through a conditional. If the sum is larger than 0 and the image's label is a 1 or the sum is smaller than 0 and the label is a 0, we consider the perceptron's prediction to be accurate. If it did not meet these conditions when comparing the prediction to the actual result we updated the weights by subtracting them from their corresponding pixels.

The algorithm changed a bit for digits, however, since we were working with more outputs from our perceptron. Instead of having an output of one node, we had an output of 10 of which we took the largest value. This argmax value was then compared to the label of the image to see if the prediction was correct and if not the weights were updated accordingly similar to how the weights were updated in the face perceptron algorithm.

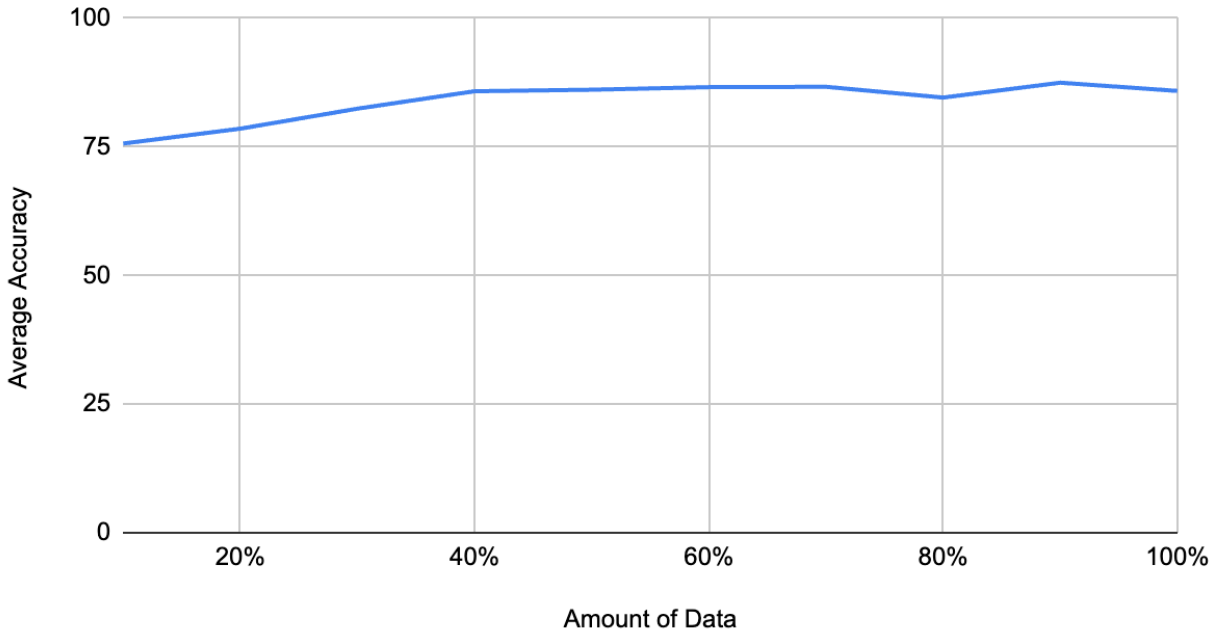
When analyzing our results for Perceptron, we saw that the time for the algorithms increased as we trained on more data for both digit and face, which makes sense, as training on more data will increase training time. For the face, we saw the accuracy went up and peaked at around 87.47% when training on 90% of the data, and then after, accuracy went down a little bit. This could be due to overfitting after a certain point. When examining the digit we noticed that the accuracy also went up as we increased the amount of data, with the accuracy peaking at 78.12% when training on 100% of the data. Both digit and face took a similar amount of time to run.

Results for Face (Perceptron)

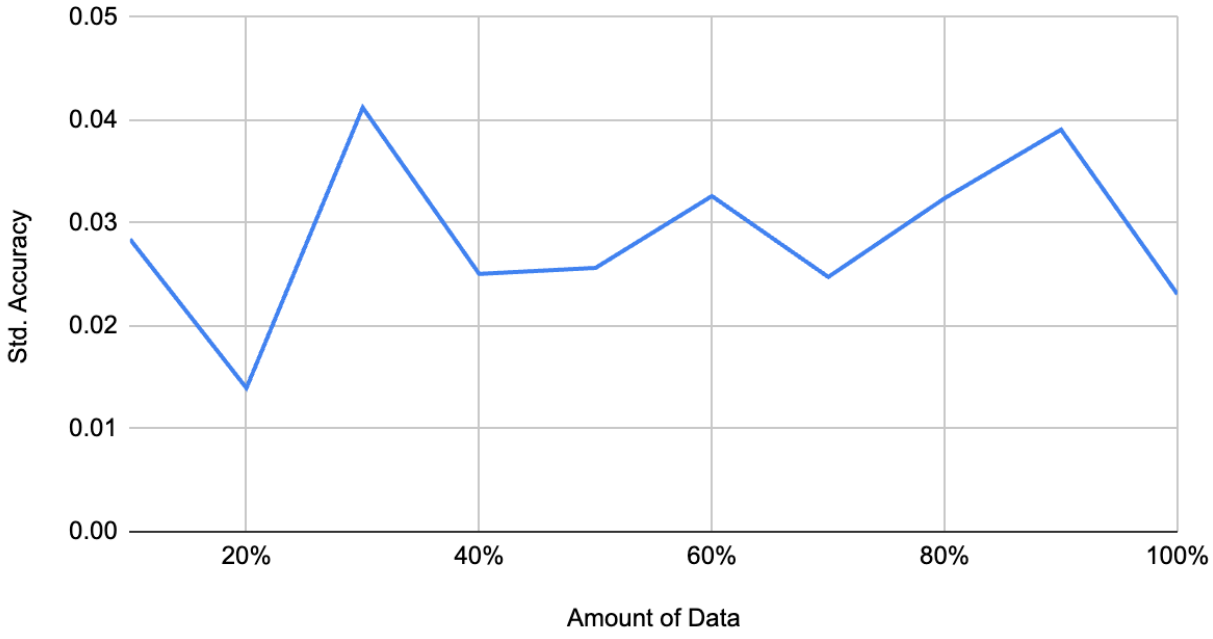
% of Training Data Used	Accuracy %	Standard Deviations	Time in seconds
10%	75.64784053156146	0.028428282691046768	6.948185920715332
20%	78.53820598006644	0.013960770519066064	7.143908977508545
30%	82.42524916943521	0.04121843407709685	7.262869119644165
40%	85.81395348837209	0.025055946249338365	7.466693162918091
50%	86.11295681063122	0.025640548212452367	7.61726713180542
60%	86.61129568106313	0.03262028683684351	7.82392692565918
70%	86.67774086378737	0.024756932491818883	8.182499170303345
80%	84.58471760797341	0.032410946179886255	8.17408275604248
90%	87.47508305647841	0.039076464114126006	8.341362953186035
100%	85.88039867109634	0.023085564470430506	8.61025595664978



Average Accuracy for Face Perceptron vs. Amount of Data



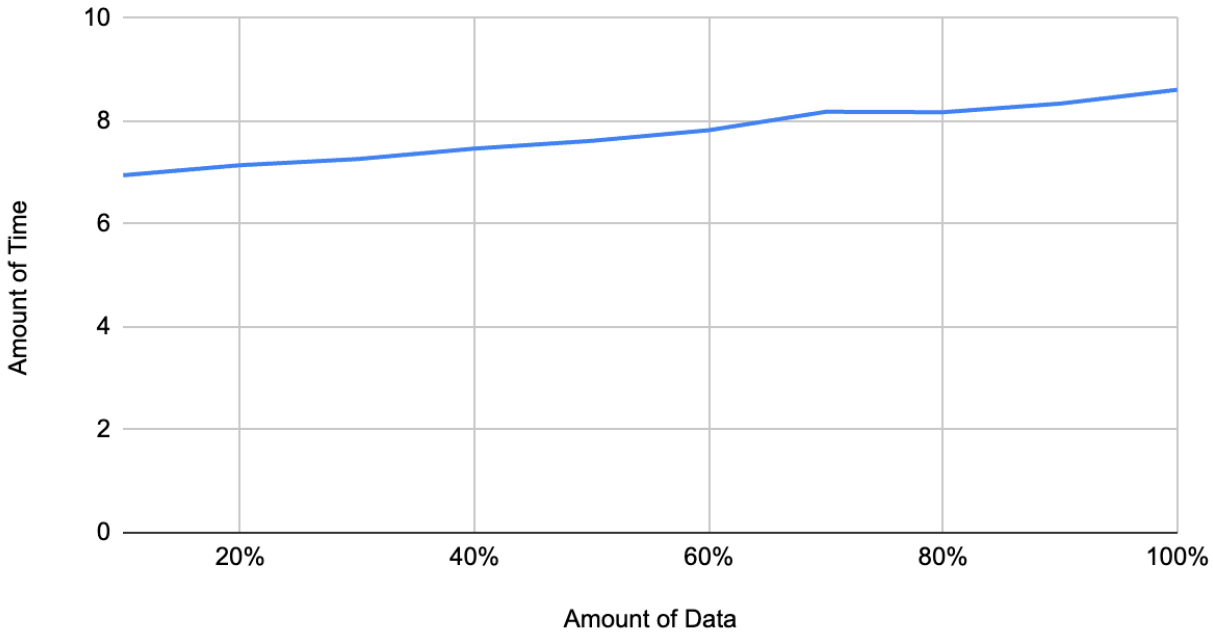
Std. Accuracy vs. Amount of Data



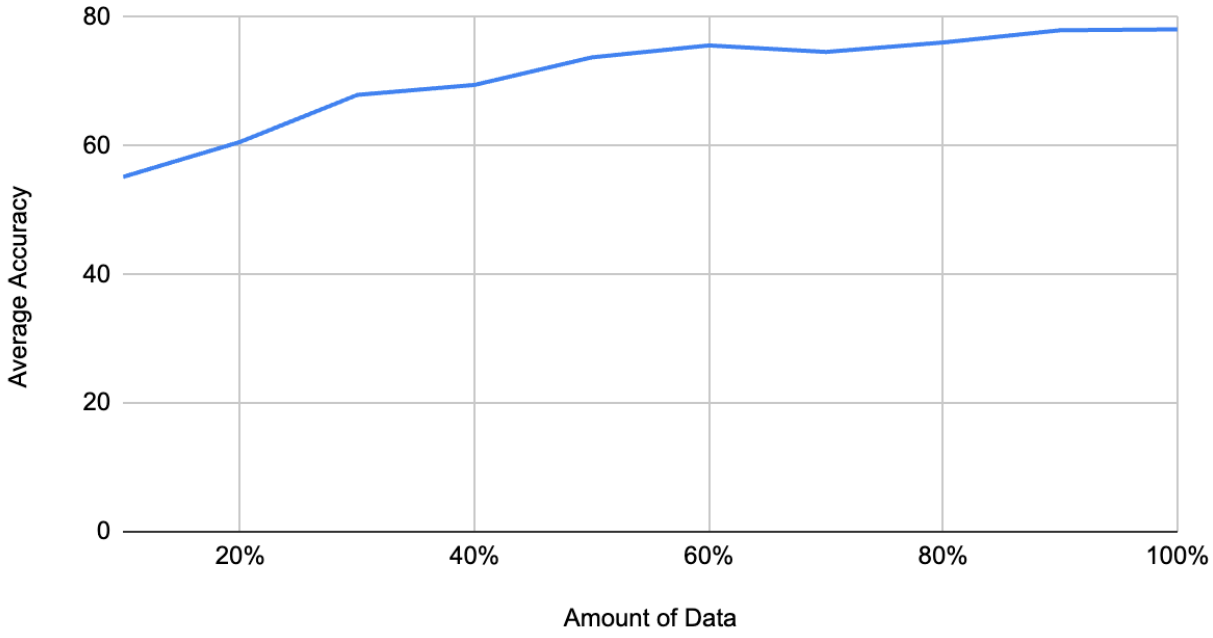
Results for Digit (Perceptron)

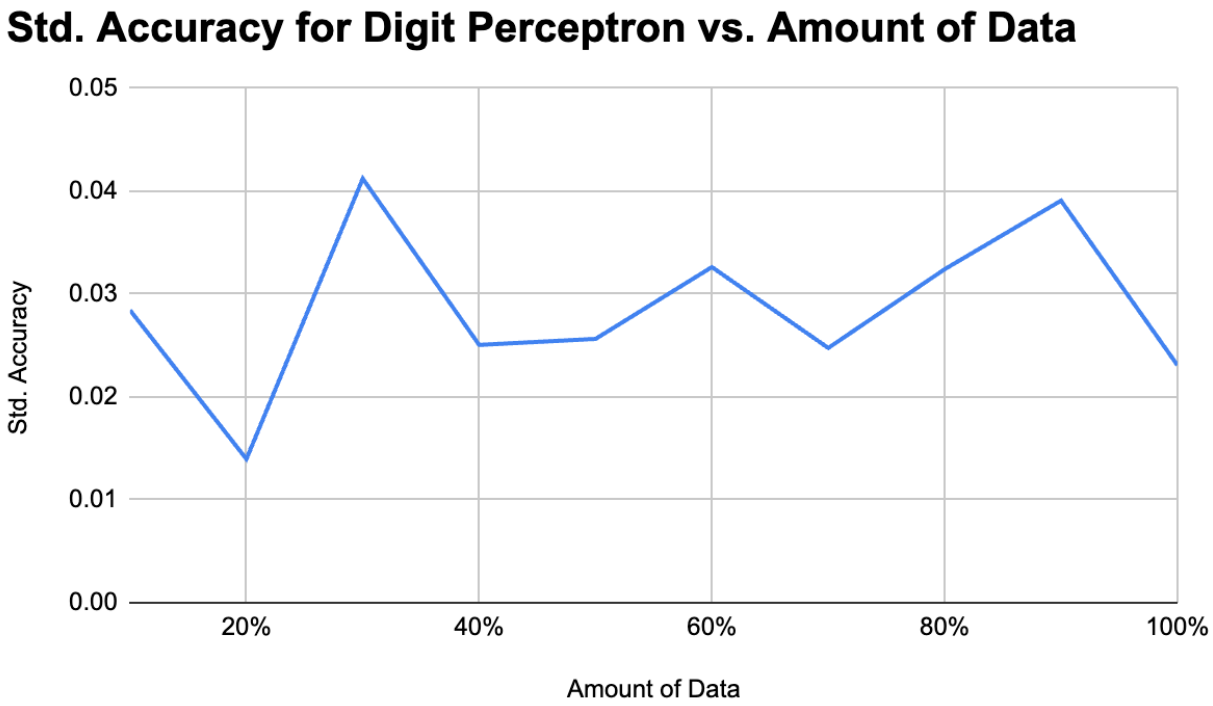
% of Training Data Used	Accuracy %	Standard Deviations	Time in seconds
10%	55.18	0.017237706741898673	6.965786933898926
20%	60.62	0.018020136775024263	8.213576078414917
30%	67.93	0.023158314983592197	8.400564908981323
40%	69.48	0.02838613046156235	8.909807205200195
50%	73.76	0.027336886281129066	9.56226396560669
60%	75.62	0.023786064215076043	11.195868015289307
70%	74.61	0.025307019904936032	12.999340057373047
80%	76.08	0.028440253997915513	12.848332166671753
90%	77.98	0.027329042397444544	13.732260704040527
100%	78.12	0.021689351260689122	14.93246579170227

Amount of Time for Digit Perceptron vs. Amount of Data



Average Accuracy of Digit Perceptron vs. Amount of Data




Std. Accuracy for Digit Perceptron vs. Amount of Data
0.05
0.04
0.03
0.02
0.01
0.00
Std. Accuracy
20%
40%
60%
80%
100%
Amount of Data

Neural Network Algorithm Explanation:

For Neural Networks, we treated each pixel as a feature in our algorithm, and we flattened the image into a 1D array of size 4200 by 1 since it was a 60 by 70 image before. We then fed these inputs into a hidden layer which consisted of 1000 hidden nodes. In this layer the computation was similar to that of face perceptron, however, we were computing the running run 1000 times for each of the nodes and each pixel had its own corresponding hidden layer weights. Once the total sum is computed it is then passed into a sigmoid function and bias is added to it. After all 1000 nodes in the hidden layer have been computed it moves forward to the next step and each node in the hidden layer is multiplied by the weights once the sum has been computed it is passed through an activation function and bias is added to the final value. Finally, this binary value is passed into a conditional. If the prediction is less than 0.5 and the label is a 0 or if the prediction is greater than 0.5 and the label is a 1 then we consider the model to be accurate. However, if the prediction does not meet either of those conditions we back-propagate through the model to adjust the weights to help with the accuracy.

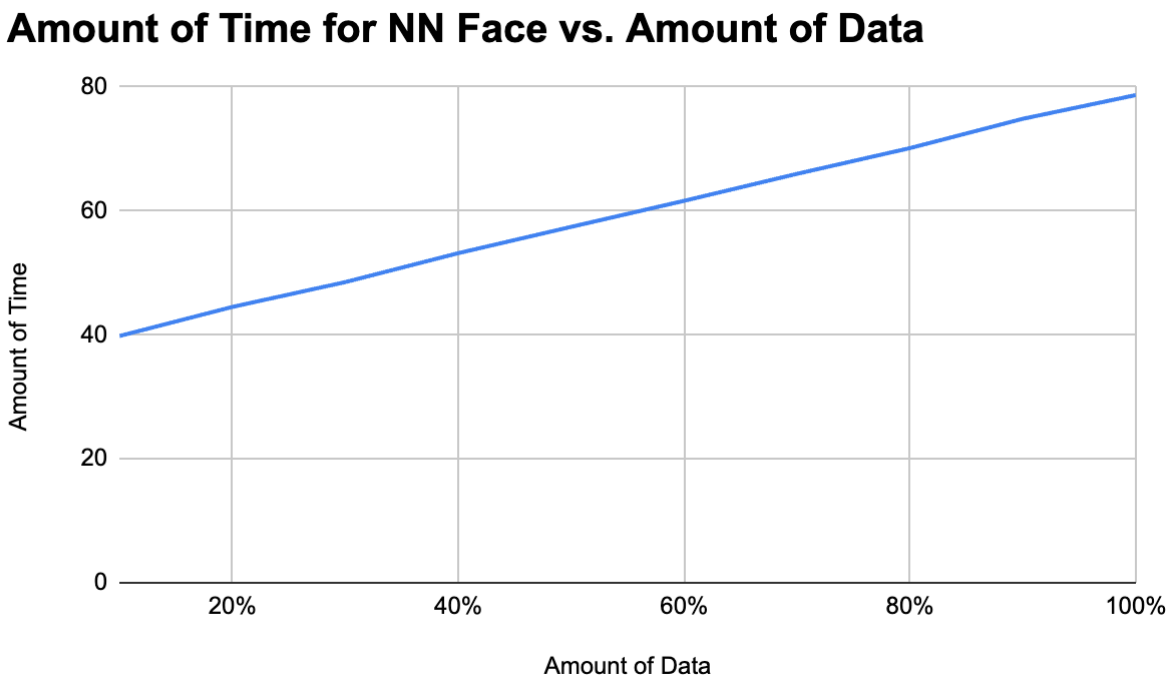
The backpropagation for the face model consists of 2 steps corresponding to the 2 sets of weights that we have. First, we compute the loss/error between the prediction and label. Once finding this value we use the backpropagation equations to compute the gradient for the output weights by multiplying the error by the hidden layer nodes. To adjust the weights of the hidden layer we take (Output Weights.T * Error) multiplied by the sigmoid derivative applied to each of the hidden layer values. Once that delta is computed it is then multiplied by every pixel in the sample image we're using our neural network on and that gives us the gradient for our hidden layer weights.

Similar to the Neural Network for Face the digit neural network forward and backpropagation worked the same since we had the same amount of hidden layers for both. However, in our digit neural network, our output consists of 10 nodes instead of the 1 node we had in the face neural network.

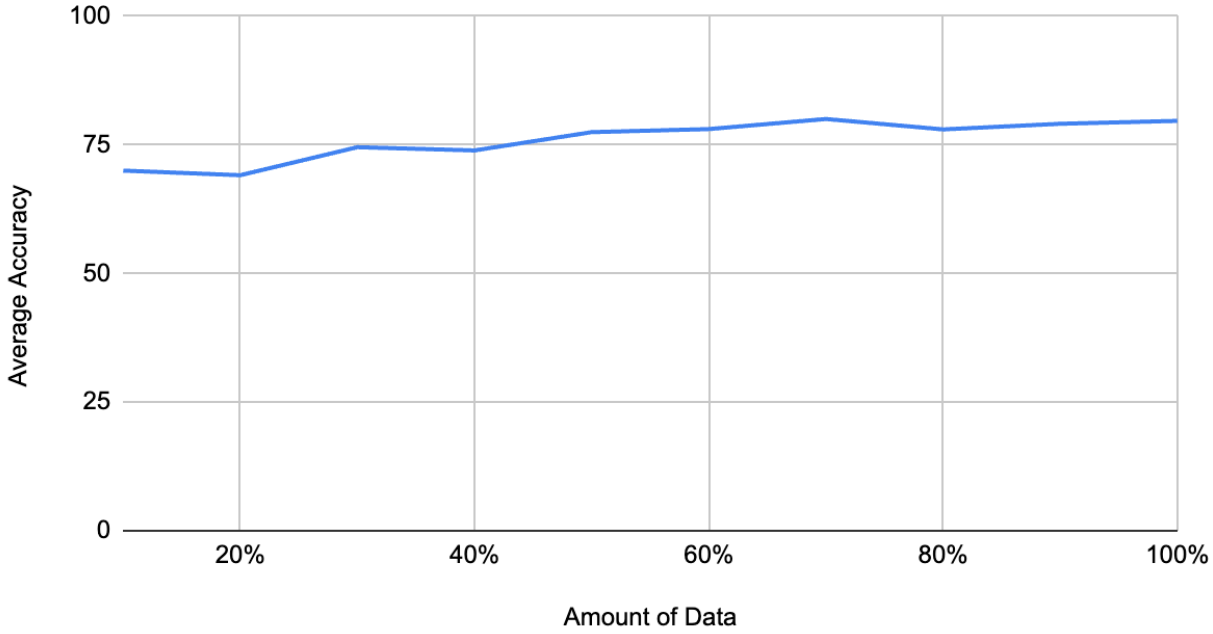
Looking at our results for the Face Neural Network we can see that as more training data was used the accuracy increased over time and the model also took more time to train. However, when looking at our perceptron performance in comparison to our neural network we noticed that the accuracy of perceptron was higher at 100% than our neural networks. We believe this to be because since the output is binary we can use a simpler model to do face classification like perceptron, however, neural networks with more epochs/iterations would eventually beat perceptron in the long run.

Results for Face (Neural Network):

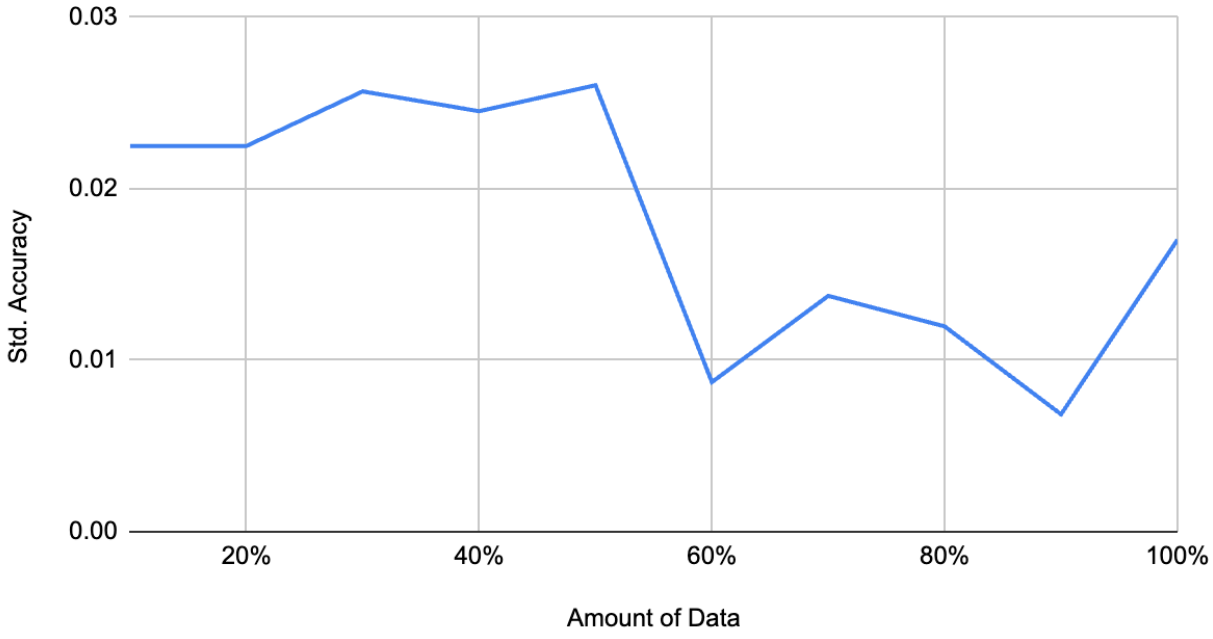
% of Training Data Used	Accuracy %	Standard Deviations	Time in seconds
10%	70	0.02249666711984611	39.91139769554138
20%	69.10299003322259	0.022494157632962964	44.23154902458191
30%	74.55149501661129	0.025676934712408162	48.70449995994568
40%	73.9202657807309	0.024520542505470325	52.7608060836792
50%	77.47508305647841	0.02603831507432826	57.234044313430786
60%	78.07308970099669	0.008723038932309446	61.952346086502075
70%	80.03322259136212	0.013753356328586612	66.02326202392578
80%	78.00664451827244	0.011972892565939823	69.99937200546265
90%	79.10299003322259	0.006846344412727003	74.22797894477844
100%	79.66777408637874	0.017019421706829067	79.13922595977783



Average Accuracy for NN Face vs. Amount of Data

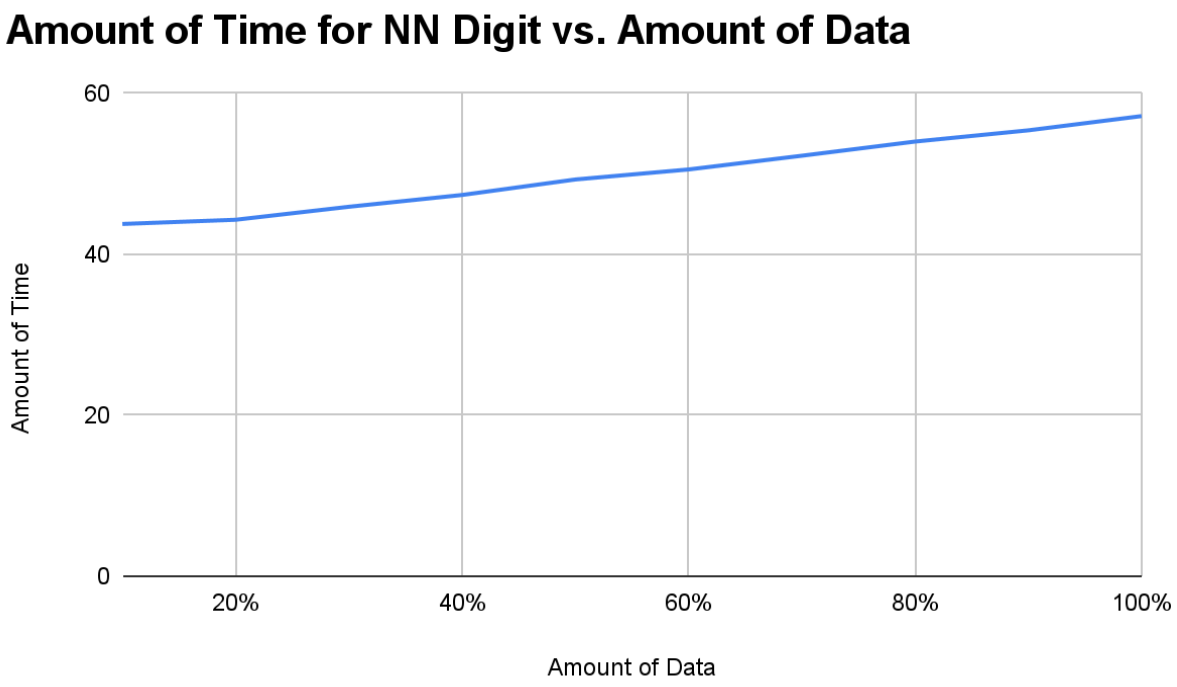


Std. Accuracy for NN Face vs. Amount of Data

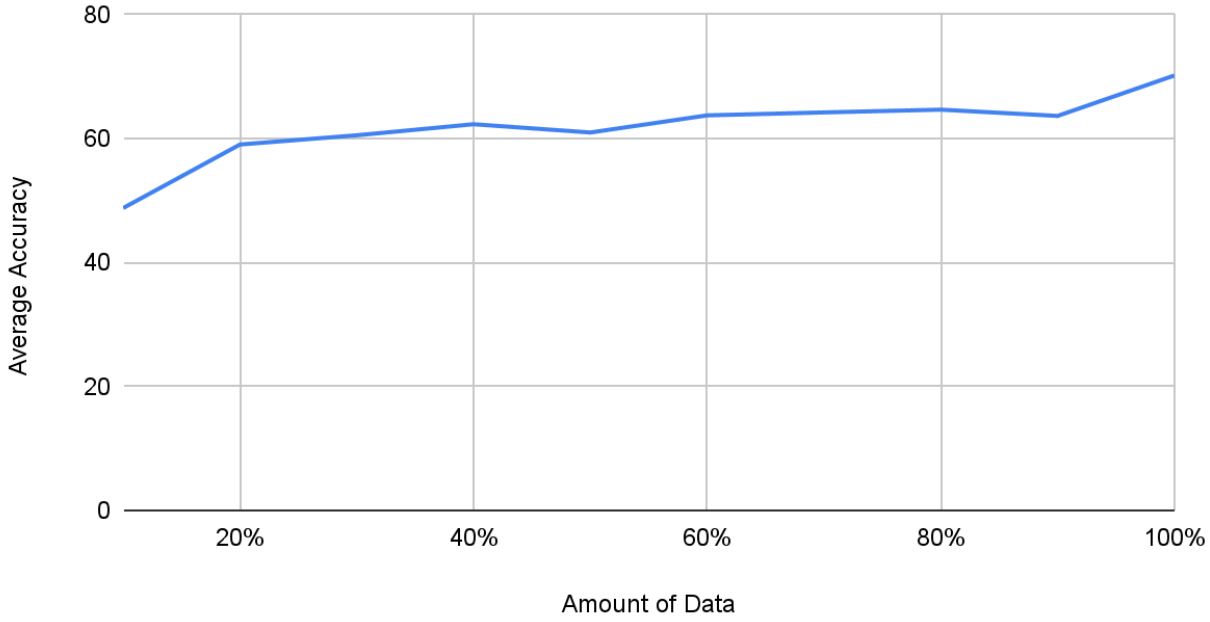


Results for Digit (Neural Network)

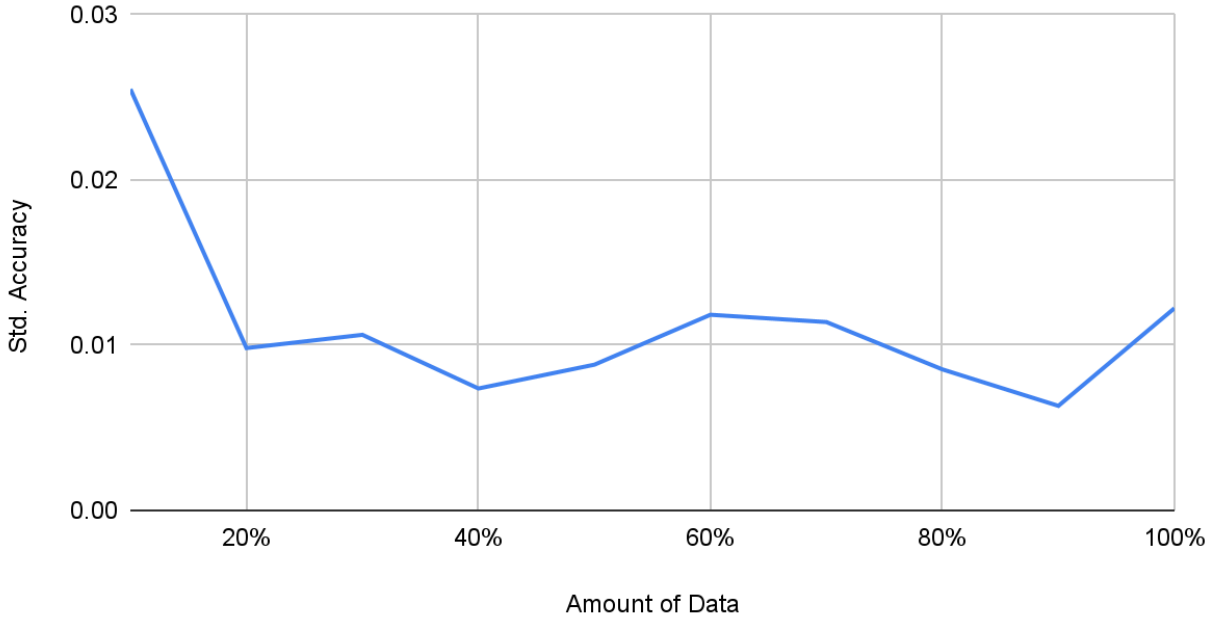
% of Training Data Used	Accuracy %	Standard Deviations	Time(Seconds)
10%	48.795	0.02549181872609036	43.76646590232849
20%	59.045	0.00982416901905768	44.289522886276245
30%	60.535	0.010624721577687984	45.903584003448486
40%	62.320	0.007377347772624984	47.37125492095947
50%	60.985	0.008817932555557672	49.29437780380249
60%	63.750	0.011839158134699204	50.54011297225952
70%	64.225	0.011397494433473185	52.247071981430054
80%	64.670	0.008531869538201764	54.00606918334961
90%	63.660	0.0063258277190521486	55.40198802947998
100%	70.185	0.012232221126571968	57.1713662147522



Average Accuracy for NN Digit vs. Amount of Data



Std. Accuracy for NN Digit vs. Amount of Data



Lessons Learned

The largest lesson learned from this assignment was the importance of optimizing your algorithm when you can! When initially coding our algorithms we used loops and normal lists, however, because of the amount of computation it would take forever to even train our models on 30-50% of the data. However, after implementing numpy operations and using np.arrays we were able to run faster and smarter computations which helped the overall efficiency of our code.

Another important lesson we took away was the quality of testing data to determine accurate predictions. It was interesting to notice that many times 100% of the data didn't lead to the most accurate performance, and how sometimes the accuracy would have small drops even with the percentage of the testing data increasing. Furthermore, it was interesting to note that since we utilized random samples in testing our data, there were often times when the same percentage of training data used yielded varying results in the accuracy of the prediction. It was also important to figure out the optimal values to initialize the weights at because when the weights were set to high or low it would lead to overfitting problems and sometimes even hurt our model from converging on the most optimal weights. The same logic applied to the learning rate, as we utilized a set of trial and error to determine the correct learning rate, if it was too high or too low the code would operate sub-optimally.