John Bailon (201003882)
Intro to AI
Final Project: Face and Digit Classification

Introduction:
The final project asked us to implement the preceptron algorithm and a two-layer neural network to classify faces and digits. To this end, I implemented both algorithms using basic python libraries, such as numpy and rando. This project taught me how neural networks work on a fundamental level, and helped me improve my coding skills through planning the overall structure of the program, and debugging complex algorithms.

Perceptron
Perceptron is an algorithm that takes an image, subdivides them into features, weighs them, and categorizes them using the resulting output. We'll discuss single-class perceptrons for faces and multi-class perceptrons for digits.

Single-Class Perceptron: Faces
First, I decided to extract features based on the number of pixels in a segment of the image. The input image is (70 x 60) so I split them into 20 features of size (14 x 15). I chose this since the implementation was simple and resulted in sufficient accuracy.

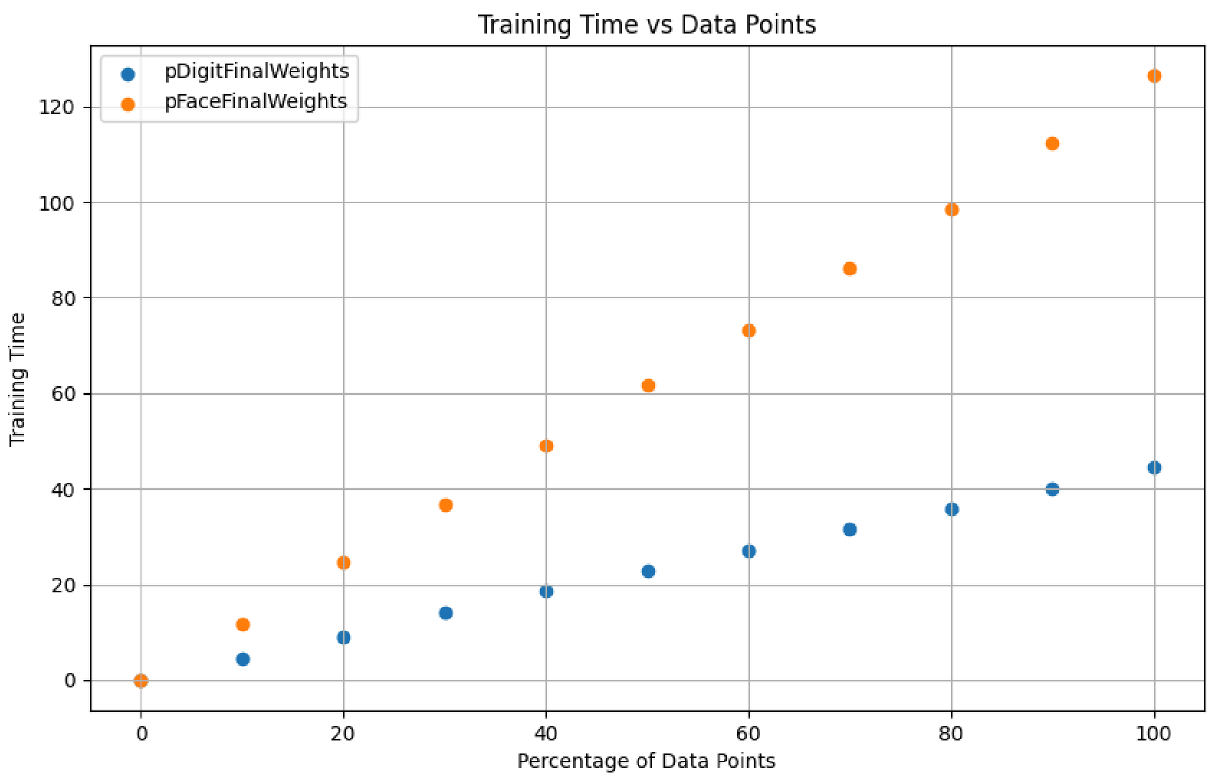
Next, I found weights by the following pseudocode:

- Cycle through all the images in the set
- Predict using the current model, find the error (actual - guess)
- If correct move on
- If incorrect, add the prediction from features to the weights

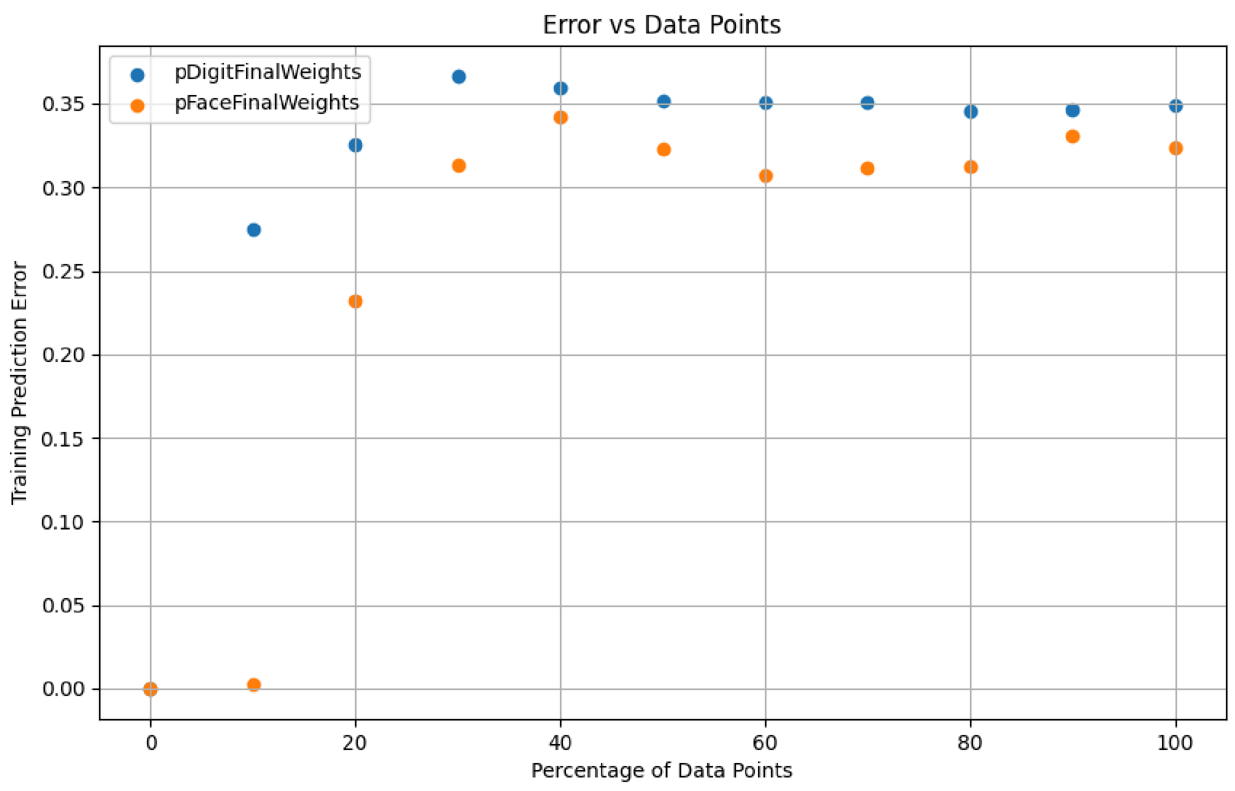
I used a binary activation function to ensure that the outputs were either 0 (not a face) or 1 (a face). From here, I trained the model using 10, 20, … 100% of the testing data.

Multi-Class Perceptron: Digits
I took a similar approach to feature extraction as faces. Since digits were (28 x 28), I split the images into 16 features of (7 x7). Similar to faces, this implementation was relatively simple. Weights were found similar to faces, with one change: For incorrect guesses, penalize the incorrect guess, and reward the real label. Here, I used argmax as the activation function to classify the output as a digit from 0 to 9.

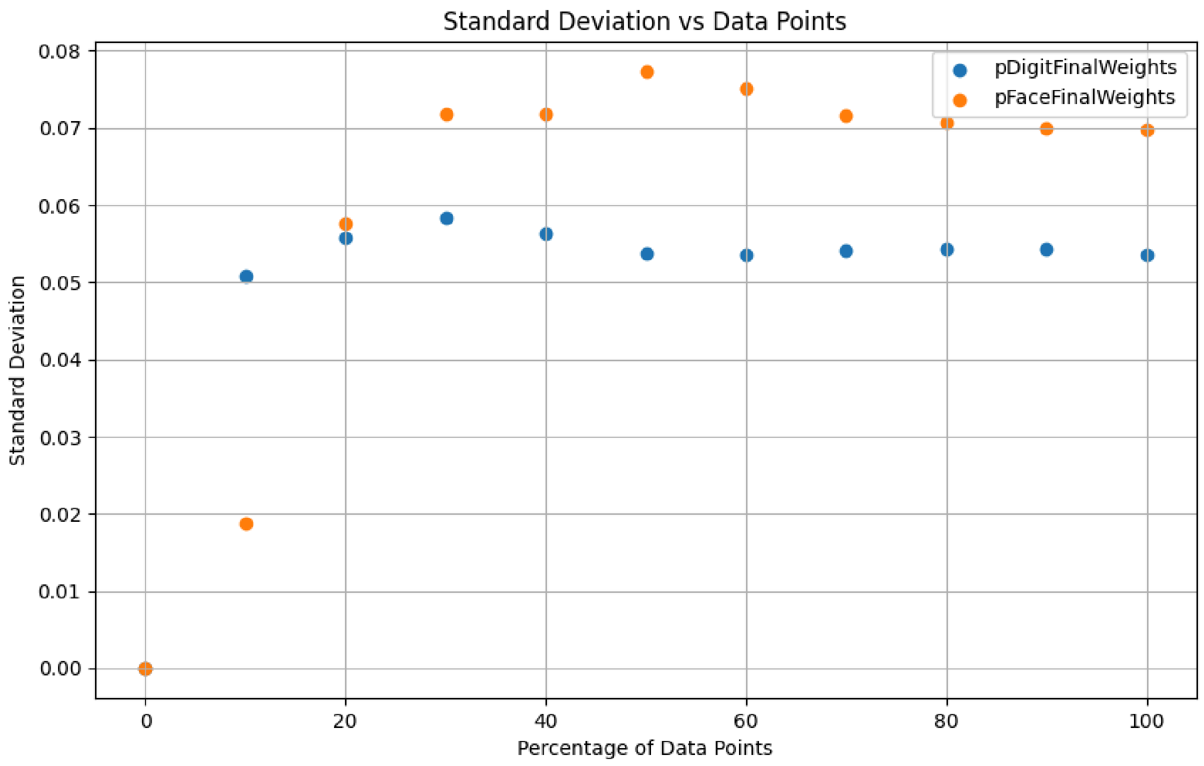
From here, I trained the model using 10, 20, … 100% of the testing data. Training time for the two perceptron algorithms are shown below.



As expected, as data points increased, the longer training the model took. I first thought that faces would perform faster than digits. But after implementing, I think that the number of epochs needed to train the face model contributed to a greater runtime.



Above is the error of both models as a function of data points. Strangely, prediction starts low and spikes after about 20% of data points. Here it fluctuates as it approaches 100%. I think the fluctuation is due to how the model adapted to the shuffling of batches during training. It might prioritize certain patterns in the beginning of training or found in a given batch, and change them as more data is added.



Lastly, is the standard deviation of both functions. For both functions it stabilizes as more data points are added. It shows that the model is relatively consistent as training continues.

Neural Networks

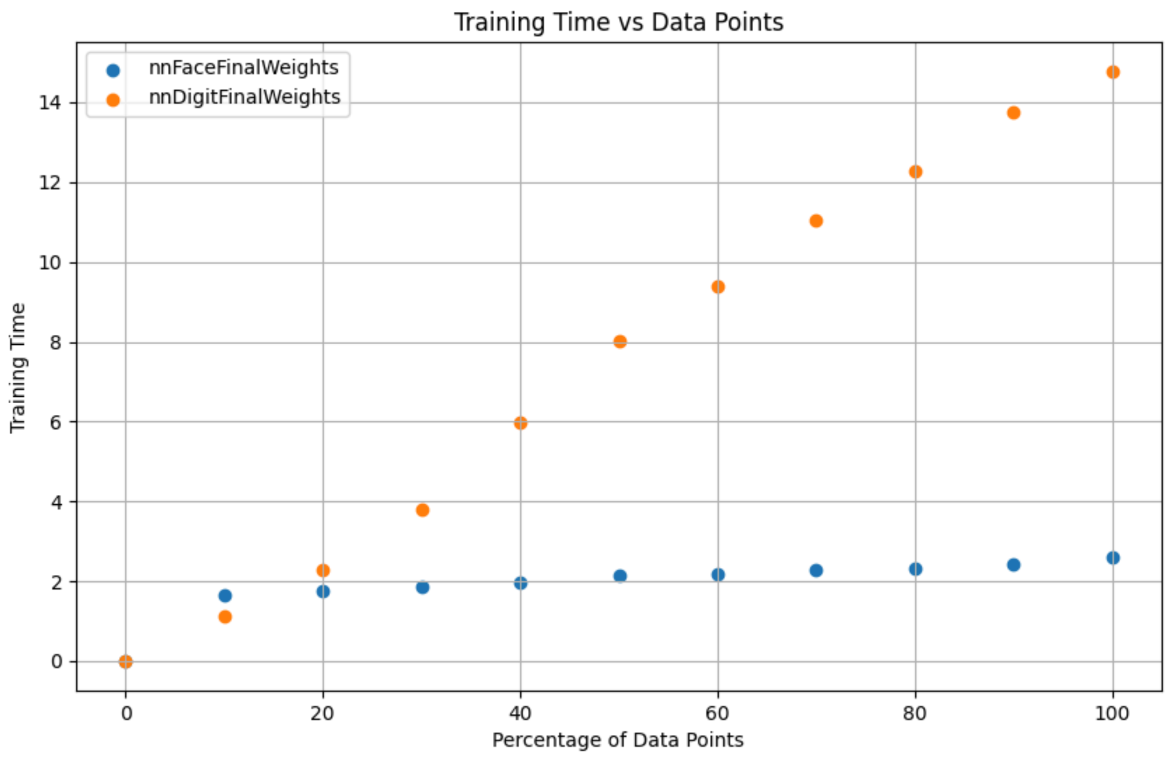
A neural network is another classifier that uses nodes and layers to classify images. For this neural network, there are three layers. An input layer, a hidden layer, and an output layer. Each node has a given value and weight. At each hidden layer, an activation function is applied to create a non-linear output that can improve the model's accuracy.

There are many activation functions, and for the hidden layer I tried sigmoid and ReLU mainly. I ended up choosing ReLu given its simple implementation, runtime performance, and how it performed similarly to sigmoid.

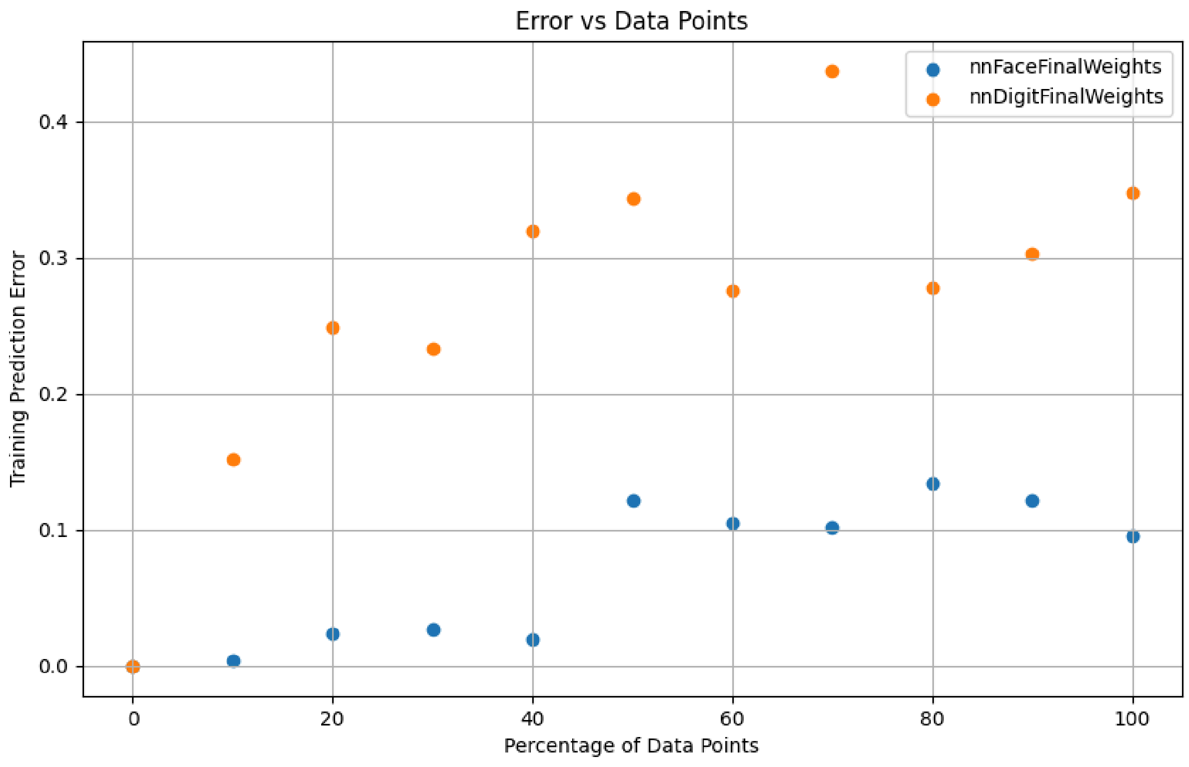
To train a neural network, we predict the output using forward propagation. Each feature of the training image is mapped to an input node. From here, it is multiplied by its relative weight and a bias is added. We apply an activation function, as mentioned prior ReLU. This process repeats to get the output layer. For the output layer, I used the softmax activation function to convert the output to each respective digit or face label.

After forward propagation comes backwards propagation. Gradient descent is used to update weights of each neuron by computing the loss, or how incorrect the model's guess is. From here, we take partial derivatives starting from the hidden layer back to the output layer. These values are used to update the weights at each layer.

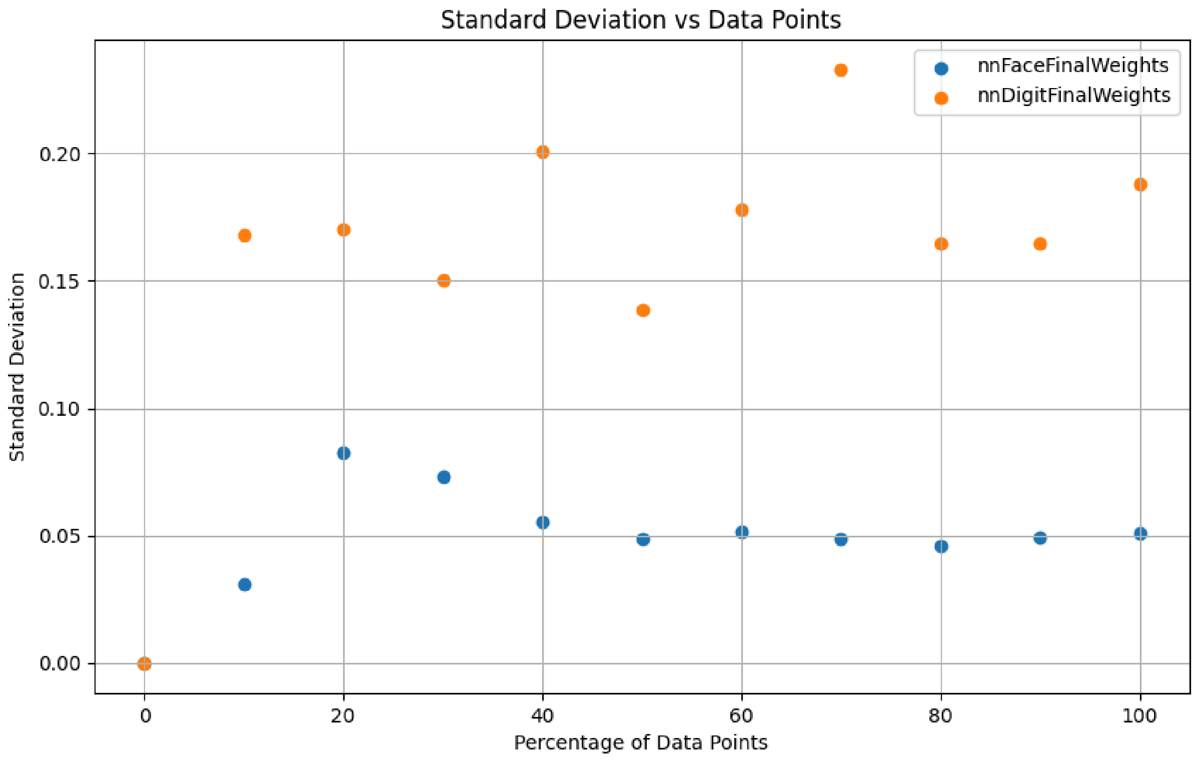
The implementation of the face and digit neural networks are very similar. Notable differences include different hidden layer sizes, (12 for faces, 10 for digits) and input layer sizes. For faces, I took 20 slices (14x15) and counted the number of pixels present. For digits, I processed each pixel to the input layer.



Above is the training time for each function. Strangely, the Face training time stays relatively constant. I think this is due to the small training size, and the compression of the images due to the features extracted.



Above is the error of both models as a function of data points. Like, perceptron, values fluctuate as we use more data points. It is possible that overfitting contributes to this, but I am unsure why digit prediction errors fluctuate significantly more than digits.



Lastly, is the standard deviation of both functions. Face values stabilize indicating a relatively consistent model, Digits fluctuates wildly. I think overfitting might have contributed to a large standard deviation.

Conclusion

Implementing a neural network was difficult. It took me trial and error to figure out appropriate learning rates, number of epochs, features, and other parameters. It helped me understand neural networks better and hopefully I can apply this to other use cases or improve the accuracy of these models.