CS440: Intro to Artificial Intelligence Final Project - Face and Digit Classification

Adityaraj Gangopadhyay (207001795)

May 1, 2024

Part 1 - Introduction

The project implements two classification algorithms for detecting faces and classifying digits:

(a) Perceptron

(b) Two-layer Neural Network (input layer, one hidden layer, output)

The project uses the code found found in the Berkeley page as the layout.

The project edits the Perceptron file and adds and implements a Two-layer Neural Network file. I also removed many of the extra classifiers, not part of the project.

Part 2 - Perceptron

The Perceptron is implemented in the PerceptronClassifier class in the perceptron.py file.

l**egalLabels** represent all the possible classes. (0-1) for faces and (0-9) for digits.

epochs is the number of times the training method iterates through a given training set during learning.

weights is a dictionary mapping each legalLabel to util.Counter object which depicts the weights vector. It initializes all features with a weight of 0.

```
def __init__(self, legalLabels, epochs):
  self.legalLabels = legalLabels
  self.type = "perceptron"
  self.epochs = epochs
  self.weights = {}
  for label in legalLabels:
    self.weights[label] = util.Counter() # this is the data-structure you should use
```

The train method:

- The train method loops through the trainingData.
- For each image in the trainingData it calculates the score for each legalLable (class) by taking the dot product of the weights and the feature vector (0s or 1s).
- It then compares the predicted class (the class with the highest score) to the actual correct label.
- If the prediction is correct it does nothing otherwise it updates the weights according to the minimizing loss function:
  – The feature vector of the misclassified image is added to the weights of the correct label.
  – The feature vector of the misclassified image is subtracted from the weights of the incorrectly predicted label.
- Then use the classify method to see the current accuracy and adjust the current weights to the best accuracy achieved in the iterations.

The **classify method**, goes through all the images in a dataset and calculates a score for each label and assigns the highest scored label for each image. It then returns a list of those labels. (The predicted labels for the dataset).

Part 3 - Two-layer Neural Network (input layer, one hidden layer, output)

The Two-layer Neural Network is implemented in the TwoLayerNeuralNetwork class in the twoLayerNeuralNetwork.py file.

```
def __init__(self, input_size, hidden_size, output_size):
    self.inputSize = input_size
    self.hiddenSize = hidden_size
    self.outputSize = output_size

    # Initialize weights and biases
    self.weightsInputHidden = numpy.random.randn(self.inputSize, self.hiddenSize)
    self.biasesHidden = numpy.zeros((1, self.hiddenSize))
    self.weightsHiddenOutput = numpy.random.randn(self.hiddenSize, self.outputSize)
    self.biasesOutput = numpy.zeros((1, self.outputSize))
```

Input Layer: The size of the input layer is determined by the size of the images in the data.

Hidden Layer: Layer of neurons that applies weights to the inputs. The size of the input layer can be determined manually.

Output Layer: The layer that produces the prediction. The size of the layer corresponds to the number of classes (labels).

The Train Method:
During the forward pass, the input data is passed through the network. The input to the hidden layer is computed by multiplying the input data with the weights of the input to the hidden layer and adding the bias. The same process is applied to compute the input to the output layer from the hidden layer output.

The classify method works similarly to the above forward pass.

Loss Function:
The loss function used is the mean squared error between the predicted output and the actual labels.

During the **backpropagation** the error is propagated back through the network to update the weights and biases. The derivative of the loss function with respect to the output is calculated, and then the derivative of the sigmoid function is applied to compute the gradients. These gradients are used to update the weights and biases of both layers.

The following source was used as a guidance to help implement and construct the two-layer neural network:

https://www.kaggle.com/code/ihalil95/building-two-layer-neural-networks-from-scratch

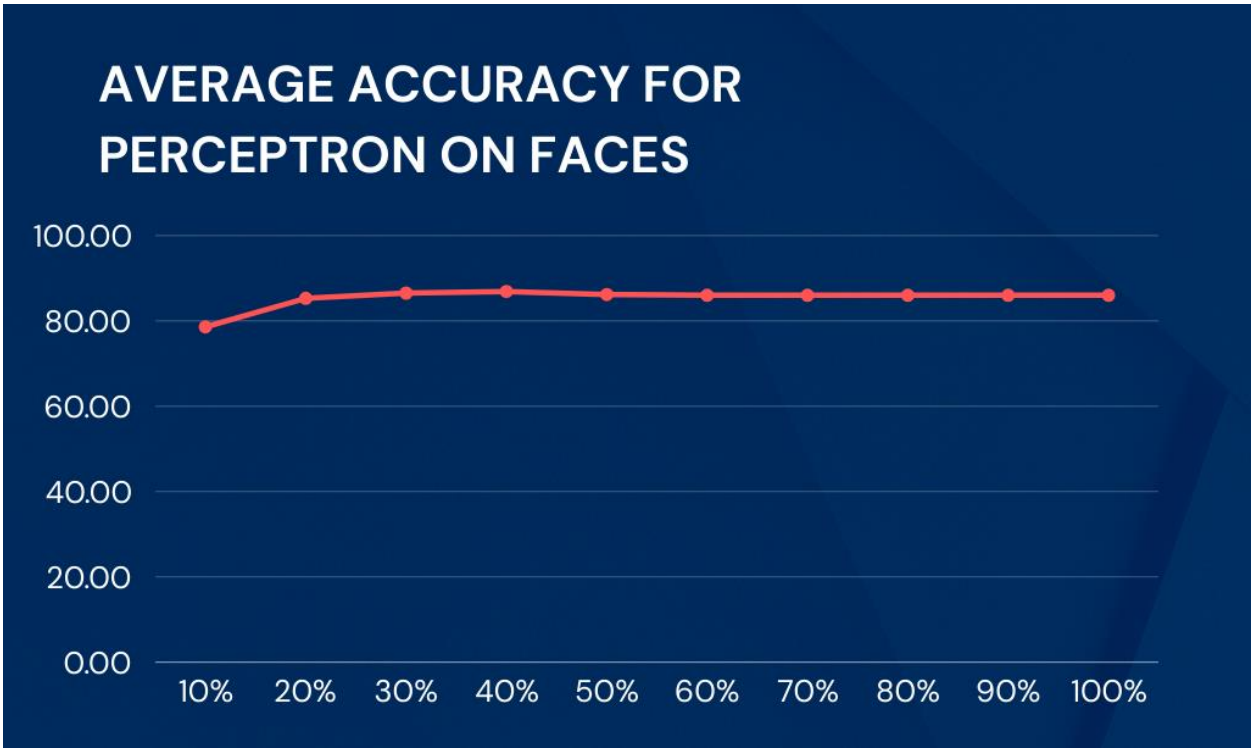
Part 4 - Test Results

For all testing for each percentage, I did 5 iterations.

Perceptron for Faces:

For perception for both Faces and Digits I initialized epochs = 3.

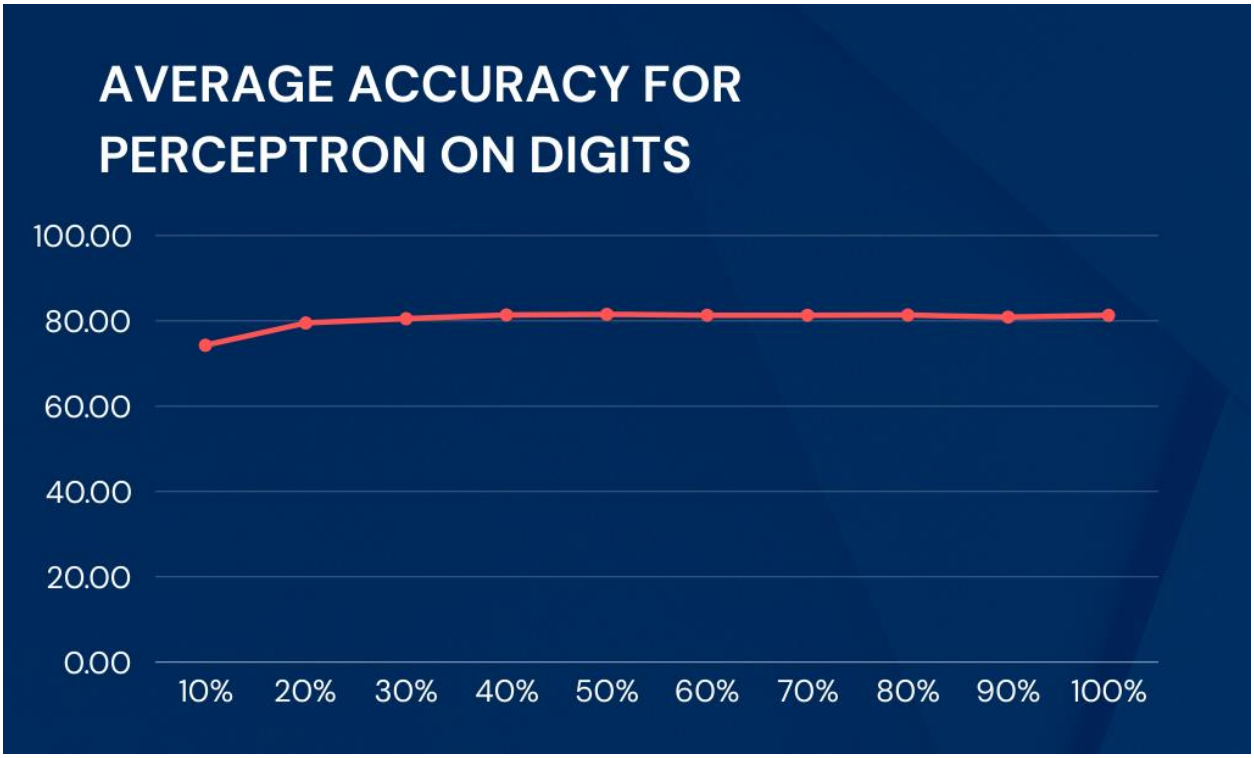
Percent of Data Tested	Avg Accuracy on Testing Data (%)	Avg Training Time (s)	Standard Deviation of Accuracy (%)
10	78.6	0.97	6.0
20	85.3	1.12	2.4
30	86.53	1.28	0.98
40	86.9	1.42	0.79
50	86.2	1.63	0.79
60	86.0	1.7	0
70	86.0	1.83	0
80	86.0	1.99	0
90	86.0	2.15	0
100	86.0	2.27	0



At 60% it seems we have hit out maximum accuracy possible. Beyond this point our neither our average accuracy nor our standard deviation (which is at 0) changes anymore. As expected the average training time increases at the percentage of data trained on increase.

Perceptron for Digits:

Percent of Data Tested	Avg Accuracy on Testing Data (%)	Avg Training Time (s)	Standard Deviation of Accuracy (%)
10	74.3	6.6	3.85
20	79.5	8.3	1.57
30	80.5	10.0	1.06
40	81.4	11.6	0.79
50	81.56	13.3	0.56
60	81.32	14.9	0.40
70	81.32	16.74	1.20
80	81.36	18.4	0.38
90	80.9	19.5	0.76
100	81.3	21.1	0.17



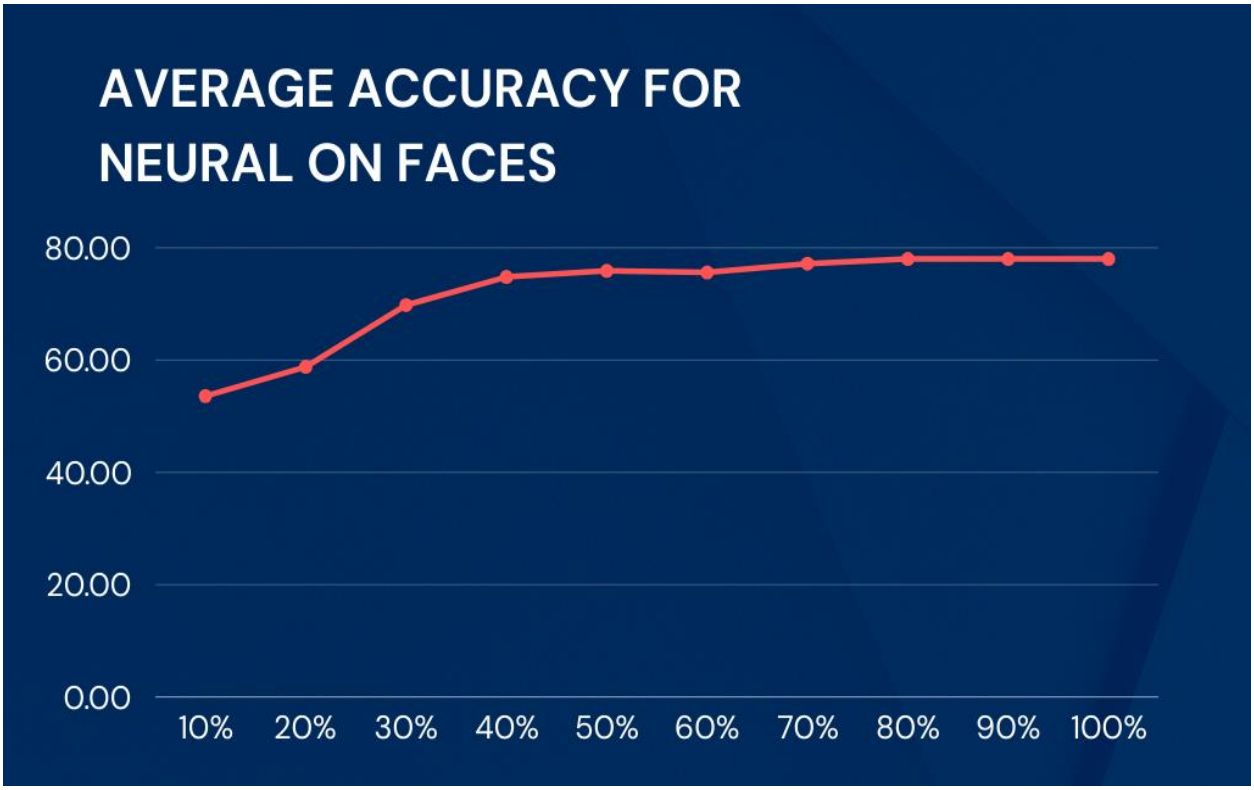
In the case of digits, it seems at roughly 40% we reach roughly our peak accuracy since after that point we see minimal movements in our accuracy numbers. Standard deviation seems to also be

barely changing beyond this point also. As expected the average training time increases as we increase the amount of training data. Training time being much larger than Faces is explained by the fact that there are just much more digit training data than faces training data.

Neural Network for Faces:

For faces we used a hidden size of 3 and a learning rate of 0.20 and 1000 epochs.

Percent of Data Tested	Avg Accuracy on Testing Data (%)	Avg Training Time (s)	Standard Deviation of Accuracy (%)
10	53.6	0.46	1.71
20	58.8	0.39	1.59
30	69.8	0.56	2.93
40	74.8	0.82	1.14
50	75.9	1.19	0.73
60	75.6	1.63	0.32
70	77.19	1.73	0.65
80	78.0	2.26	0
90	78.0	2.41	0
100	78.0	2.34	0

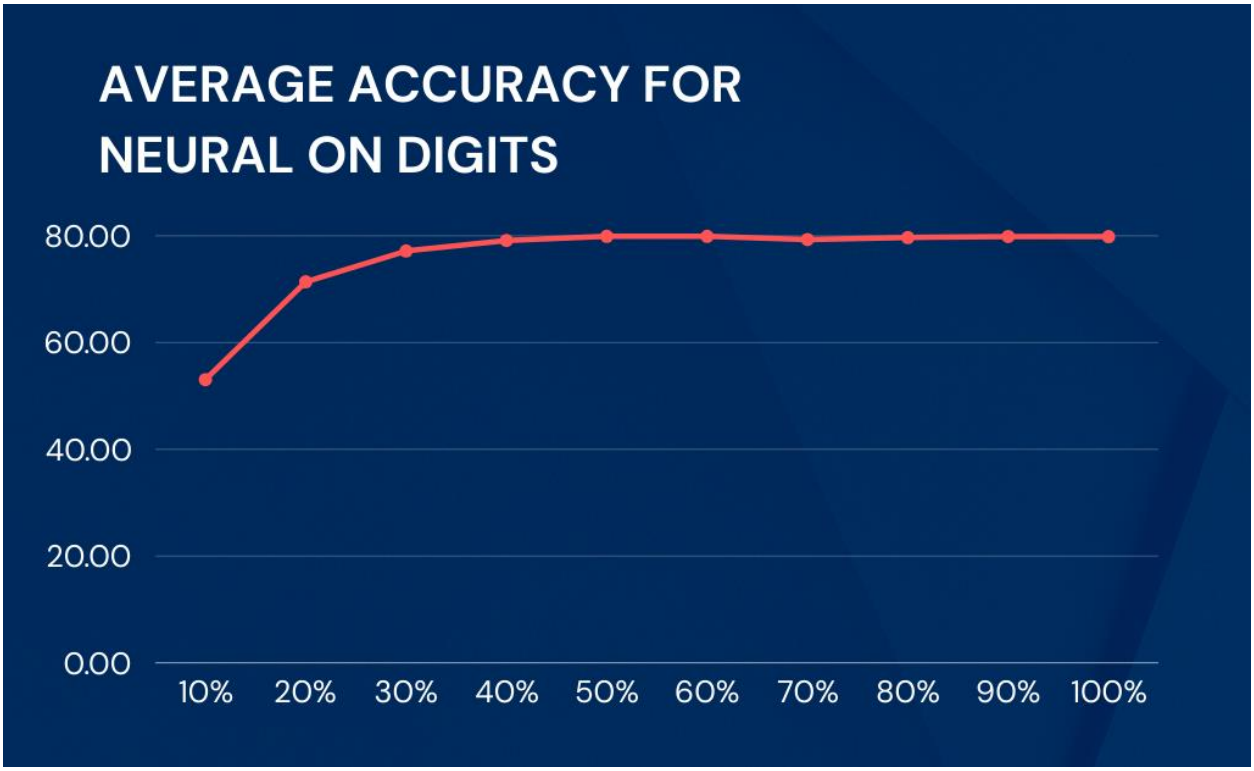


Once we have hit 80% of training data, we have reached peak average accuracy. There is also no more standard deviation beyond this point. As expected average time for training keeps increasing as the percentage of data used for training increase.

Neural Network for Digits:

For digits we used a hidden size of 10 and a learning rate of 0.001 and 1000 epochs.

Percent of Data Tested	Avg Accuracy on Testing Data (%)	Avg Training Time (s)	Standard Deviation of Accuracy (%)
10	53.06	1.36	6.79
20	71.38	1.92	2.33
30	77.16	3.13	1.70
40	79.12	5.18	0.33
50	79.9	8.30	0.30
60	79.9	14.4	0.34
70	79.3	14.8	0.41
80	79.68	13.2	0.14
90	79.88	15.1	0.23
100	79.84	12.6	0.16



Once we have hit 40% of training data used, we seem to have hit peak accuracy again. Beyond this point standard deviation also remains low and does not exhibit much movement. As for the training times, while it does generally increase as the percentage of training data increase, there are some outliers. These outliers are small and may be explained by the fact that a computer may have additional different loads on its CPU (not related to the project) affecting overall performance.

Last Notes:

The reason for why much lower percentage was needed for digits to reach peak accuracy is most likely due to there being much more digits training data available compared to faces training data.

Additionally the reason for Neural Network performing slightly faster than perceptron is most likely caused by the efficiencies of the numpy library and its methods.

Resources:

https://inst.eecs.berkeley.edu/~cs188/sp11/projects/classification/classification.html

https://www.kaggle.com/code/ihalil95/building-two-layer-neural-networks-from-scratch