Feature Selection in ProcessData.py

For both the faces and digits, we scanned each line of the .txt file, and matched each character present with a numerical value:

1. ‘ ‘ = 0
2. ‘+’ = .5
3. ‘#’ = 1

The numbers from 0 to 1 represent the darkness of the image at that point. 0 means it is completely white, 1 means it is completely black.

For both the faces and digits, we simply took the numerical value of each character in the file and added it to a numpy vector. For the digits, each digit was represented in the .txt file using 28x28 characters, therefore our feature was a 784x1 vector filled with 0,.5, and 1 at the appropriate location. For the faces, which were represented using 70x60 characters, we had a 4200x1 vector. We did not average the numbers or sum them here, as simple extraction led to sufficient accuracy.

How the computer learns in Perceptron.py

Description

Perceptron uses linear regression techniques to properly classify images. We train the computer to make a line that takes an x value, and passes it through a linear function to output an y value. This y value may be passed through an activation function to map all predictions between 0 to 1, as only 0 or 1, etc. Classic perceptron takes any y >= 0, makes it a 1, and otherwise a 0. We used a sigmoid activation function that passed y through the sigmoid function:

$$f = \frac{1}{1+e^{-y}}$$

The x value is the extracted feature, and the f value is the computer’s prediction of the label. The learning algorithm continuously changes this line’s placement and its slope in a multidimensional coordinate system until the error between the prediction and label is minimized.

For digits, we have ten equations for classifying each digit.

$$y_0 = b_0 + w_{0,0} \times x_0 + w_{0,1} \times x_1 + ... + w_{0,783} \times x_{783}$$
$$y_1 = b_1 + w_{1,0} \times x_0 + w_{1,1} \times x_1 + ... + w_{1,783} \times x_{783}$$
$$\dots$$
$$y_9 = b_9 + w_{9,0} \times x_0 + w_{9,1} \times x_1 + ... + w_{9,783} \times x_{783}$$

$w_{i,j}$ = *weight for digit i associated with the value at j in the feature vector x*

This method lends itself very obviously to linear algebra operations.

$$\vec{y} = w \times \vec{x} + \vec{b}$$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_9 \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,783} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,783} \\ \vdots & \vdots & \ddots & \vdots \\ w_{9,0} & w_{9,1} & \cdots & w_{9,783} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{783} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_9 \end{bmatrix}$$

We only need one equation to classify faces.

$$\begin{bmatrix} y_0 \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,4199} \end{bmatrix} \times \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{4199} \end{bmatrix} + \begin{bmatrix} b_0 \end{bmatrix}$$

Because perceptron is a type of linear regression, the use of matrices and linear algebra facilitates calculations . This also increased efficiency as Python supports a linear algebra library known as Numpy.

Implementation

Setup/Overview

Initialize the weights matrix and the bias vector randomly, which will be used as the coefficients for the linear regression.

Set up the labels to be vectors:

For digits: 10x1 vector, where the row index of the classified number is equal to 1, and the rest are 0.

For example, the label of 3 would be:

$$l = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

For faces: 1x1 vector which is 1 if the image is a face, and 0 otherwise.
Then, I made a list of tuples where each tuple was the feature vector and label vector. I then shuffled the training data, and extracted the first 10%, 20%, 30%, etc, depending on which test I wanted. Next I made a for loop to go through epochs inside which one learning process would occur. Once the learning process ended, I tested the computer's corrected weights and biases with the test data features to calculate an accuracy.

Learning Process

Using the feature vector $\vec{x}$, calculate $\vec{y}$:

$$\vec{y} = w \times \vec{x} + \vec{b}$$

Then pass $\vec{y}$ through the sigmoid function to get f, our prediction. We calculate new weights and biases to minimize the error between f (prediction) and l (label). To do so you must take the partial derivative of the cost function to find which direction to move **w** and $\vec{b}$ so the derivative of the error is 0 (the minimum). This is pretty much calculating the error between the prediction and label. The cost function and its derivative is:

$$C = (l - f)^2, \quad f = \sigma(\vec{y})$$
$$C = (l - \sigma(\vec{y}))^2$$

$$\vec{y} = w \times \vec{x} + \vec{b}$$
$$\frac{\partial}{\partial w}\vec{y} = \vec{x}$$

$$\frac{\partial}{\partial w}C = 2 \times (l - \sigma(\vec{y})) \times \sigma'(\vec{y}) \times \vec{x}$$
$$\frac{\partial}{\partial w}C = 2 \times (l - \sigma(\vec{y})) \times \sigma'(\vec{y}) \times \vec{x}$$
$$\frac{\partial}{\partial w}C = 2 \times (l - \sigma(\vec{y})) \times \frac{-e^{-\vec{y}}}{(1 + e^{-\vec{y}})^2} \times \vec{x}$$

$$\frac{e^{-\vec{y}}}{(1 + e^{-\vec{y}})^2} = (1 - f)(f)$$

$$\frac{\partial}{\partial w}C = 2 \times (f - l) \times f \times (1 - f) \times \vec{x}$$
$$error = (f - l) \times f \times (1 - f) \times \vec{x}$$

Therefore, to update the weights and biases, we take error of each training image, and add it to the value errorW and errorB. This then gives us the total error for this training set, then we add this error to the weights and biases.

$$errorW = \sum error \times \vec{x}$$
$$errorB = \sum error$$
$$w = w - \alpha \times errorW$$
$$b = b - \alpha \times errorB$$

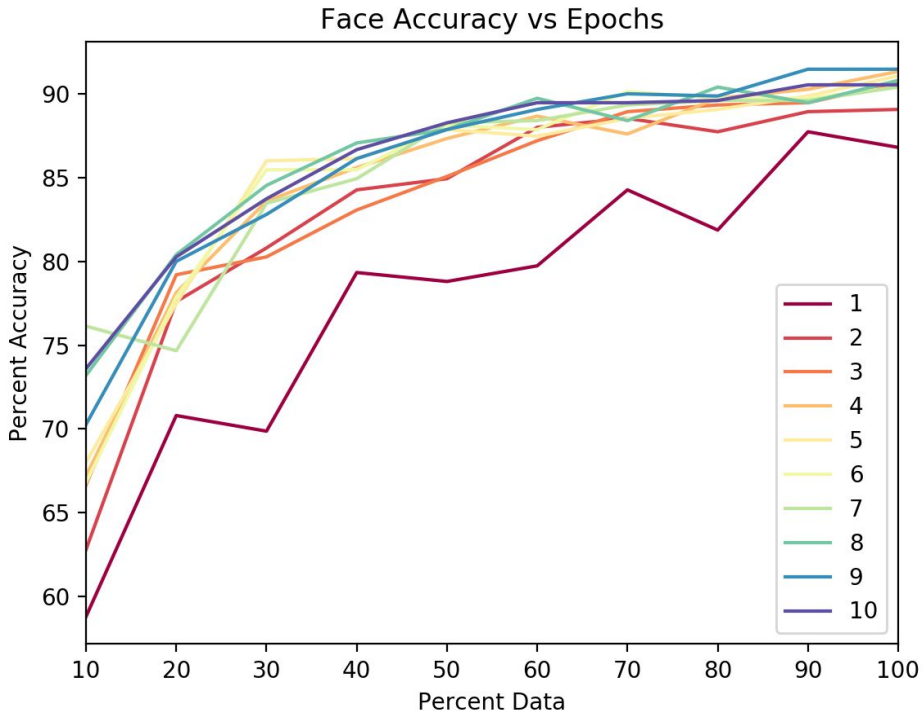
α = learning rate, which is how quickly you want your algorithm to reach the minimum, without stepping over it. I find the error for every image, then update the weights and biases for every image in the training data that we are looking at. Once this is complete, if the epoch is more than 1, we repeat this process again after shuffling the part of the training data we are looking at. Each

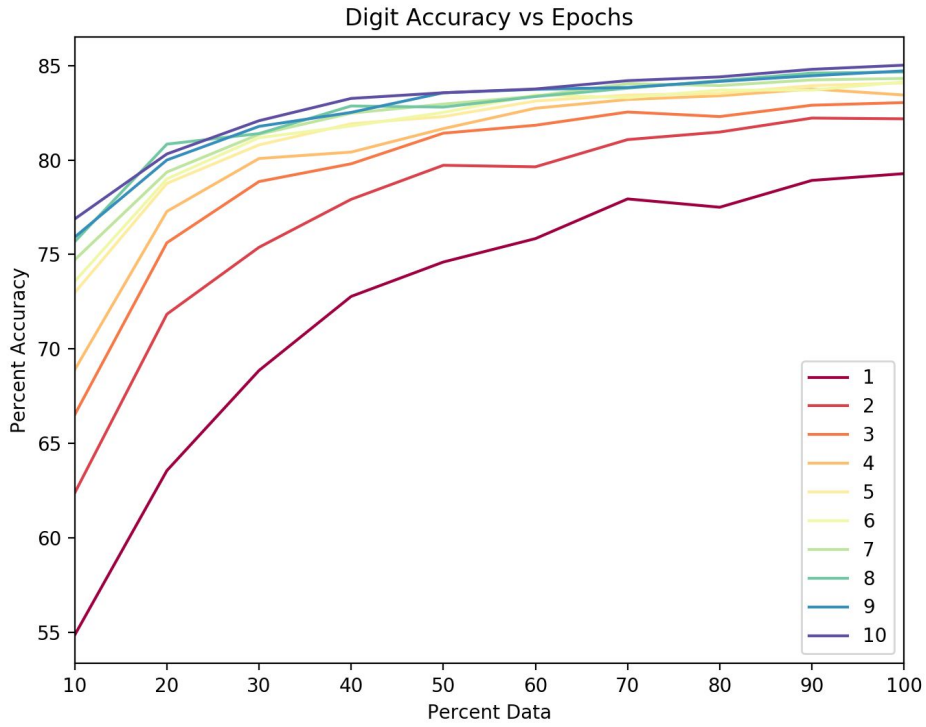
epoch builds on top of the previous learning using the same images each time, but in a different order.

Testing

Once the weights and biases have been calculated, we use them to get a prediction for the test images. The sigmoid function returns a 10x1 vector for the digits, the index of the maximum row is the prediction of our algorithm. This is then matched with the index of the maximum row of the label (where the value=1). For the faces, it returns a 1x1 vector, which if $f >= .5$, the prediction is set to 1, else 0 which is then matched to the label.

Performance





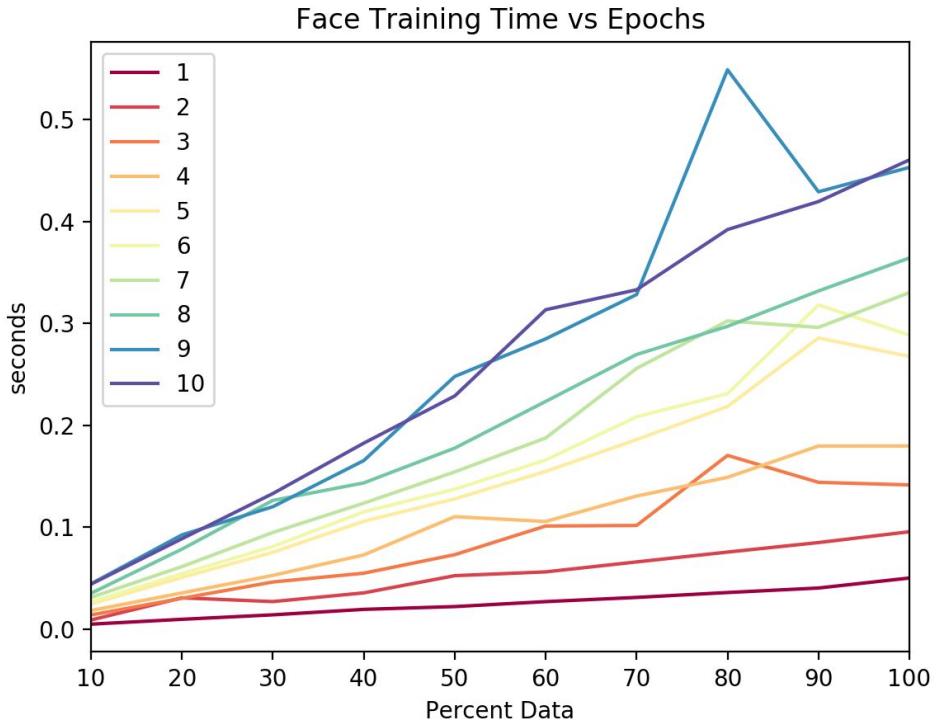
The different colored lines here represent the number of epochs for each perceptron run (seen in the legend) Overall, the accuracy of the faces tends to jump around more than that of the digits, which is reflected in the standard deviation for each percentage (at 2 epochs) as well as the graphs above (at varying epochs):

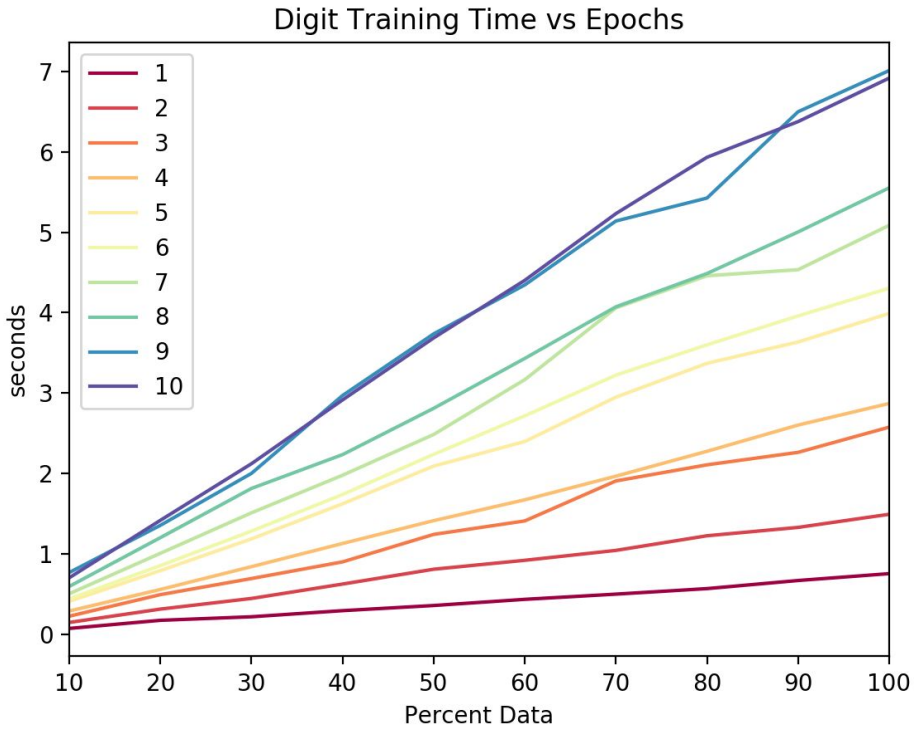
%	SD of Accuracy as % for Faces	SD of Accuracy as % for Digits
10	8.291896982	3.243763247
20	3.608631627	1.852835665
30	4.147288271	2.418057071
40	3.252349578	1.495660389
50	2.812669748	0.722495675
60	2.108185107	1.083051245
70	0.869226987	0.813633824
80	1.801234145	0.580517011
90	1.115546702	0.414728827
100	1.534781924	0.936482781

However, though the accuracy for the faces was not as stable, it consistently gave higher accuracies than that of digits. This might be due to the fact that there was only one row in the vector so there is already a 50% chance of a correct prediction even if the guesses are random. Face accuracy reached as high as 92% during some trials, whereas the digit accuracies fluctuated around 85% as maximum.

With increased number of epochs, the prediction accuracy also increased. This leads me to believe that not only does the amount of training data we select matter, but so does the order in which we pass these images into the algorithm. If order did not matter, additional epochs would not give higher accuracies as the specific images would be the same with each epoch. However, with each epoch we shuffle our training data, and build on top of the weights and biases provided by the preceding epoch. This means that the order we pass these images into our learning process does matter. Each epoch is a way of providing more information, not more data, that can additionally help correct wrong predictions made by the weights and biases from the previous epoch. However, with too many epochs, your algorithm starts to almost memorize the training data, and not necessarily learn how to classify images from a different data set. There is a happy medium you must find. Because I only ran my project on 10 epochs, I did not reach this case, known as “overfitting”. Eventually, increasing the number of epochs will not increase accuracy. The accuracy will plateau.

Though a larger number of epochs gives more accurate results (to a certain extent), more epochs mean a longer run time. As the number of epochs increase, the run time also increases. Adding on, as the percentage of training data used increases, so does the runtime as well. As seen from the line graphs on the next page. The face training time has an outlier at 80% for 9 epochs as there was a trial in that group that took more than 1 second which significantly affected the average. However, some other epochs at 80% also have a higher runtime. Overall, face training time was under 1 second. Digit training time was much longer as there was more training data, starting around under 1 second for 10% data, and then steadily increasing. The runtime for faces again less stable than the runtime for digits.


Face Training Time vs Epochs
seconds
Percent Data
1
2
3
4
5
6
7
8
9
10
0.0
0.1
0.2
0.3
0.4
0.5
10
20
30
40
50
60
70
80
90
100


Digit Training Time vs Epochs
seconds
Percent Data
1
2
3
4
5
6
7
8
9
10
0
1
2
3
4
5
6
7
10
20
30
40
50
60
70
80
90
100