Digit Classification and Face Detection Project

Roshan Patel 200000858 Abhay Dhiman 192009888

May 4th 2024

1 Introduction

In this project we investigate Neural Networks. We implement a Perceptron model and a Two Layer Neural Network model to perform Face Detection and Digit Classification. We used data provided by the Berkeley CS118 project.

2 Features

For the features of this model we found that using the raw binary information works better for Perceptron while grid-and-counting-the-1s worked well enough for the Neural Network. Raw data works better with the Neural Network as well, but because the Neural Network is slower it takes more time as opposed to the Perceptron. Feature engineering reduces the accuracy, however we found it doesn't reduce by much but performs faster.

3 Perceptron

The most fundamental Neural Network model is the Perceptron. It is essentially the building block of Neural Networks. In our Perceptron model, the user initializes the model by giving size of the input (number of features) and the size of the output. The output size varies based on which type of classification is being performed (2 for Face detection and 10 for Digit Classification). The user can then train a Perceptron model by providing the data of the features for training, validation data, and the labels for each image in the training and validation data, and the number of epochs.

3.1 Perceptron Algorithm

In the inner working of the Perceptron, we start with a weight matrix initialized to 0 that is of the dimensions input size x output size. We take the matrix multiplication of Weights x Input(features). This results in a weighted sum for each class in the output(10 classes for the 10 digits in digit classification and 2 classes for Yes or No in face detection). The prediction for each type of

classification is slightly different. For face detection we see if the "Yes" class has summed to above 0.5. If this is the case then predict 1(yes) or 0(no). For digit classification simply find the index of the max of the weighted sums and make that the prediction. With a prediction made, check it with the label for that image. If the prediction is correct move on. If not, then you essentially punish the weights of the class that you made a prediction of and reward the weights of the class of the actual label. This punishing and rewarding of weights is essentially what allows the Perceptron to learn.

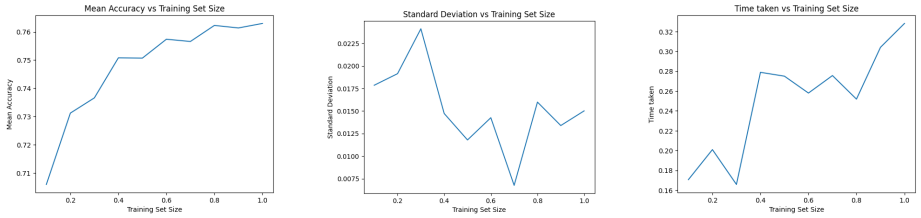
3.2 Perceptron Implementation

In our implementation and experimentation of the Perceptron model we had essentially 10 training sets which are essentially random picks from the training set that total to multiples of 10 [random 10% of the training data is one set, random 20% of the training data is another set...]. In each set, we trained on the data for 10 iterations. In each iteration, the data is trained for max 10 epochs with early abandoning implemented. Early abandoning tracks if the validation accuracy is falling and if it is for a certain window of epochs, then the training terminates. In our implementation, we found an early abandonment window of 3 works well. This means that if the validation accuracy of one of the next three epochs does not beat the validation accuracy of the current epoch then the model is trained enough and that further training can ruin the performance. From here, the model that results from the epoch with best validation accuracy is chosen to predict on the testing data set. The test accuracy and the time it took to train is saved in an array to represent the accuracy and time of that iteration. The same thing is done with the other 9 iterations. Finally, the mean and the standard deviation of the 10 accuracies is calculated as well as the mean of the times. These three calculations are saved in an array for that training set. The same protocols are performed for the other 9 training sets.

As mentioned earlier, the Perceptron essentially takes in four parameters: data, output vector size, and number of epochs, and size of early abandoning. Considering the Perceptron works relatively fast, we decided to use the raw binary data to get as accurate as possible, therefore we didn't have to feature engineer the training, validation, or testing data. The output vector size is decided by the user based on if they are doing Digit Classification or Face Detection, 10 would be set for the former while 2 for the latter. The user can set an early stopping window size. We found that 3 works well for both face and digit implementations. Lastly, the user can also set the number of epochs to train on. We found 20 epochs works well enough considering early abandoning terminates before 20 epochs.

3.3 Results

With that being said, Figure 1 shows the results for Digit Classification, and Figure 2 shows the results for Face Detection. For Digit Classification, using

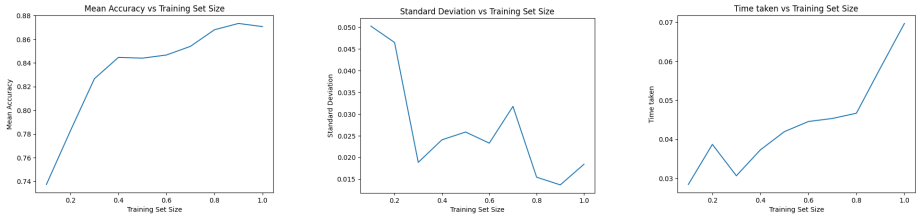


(a) Mean Accuracies of Digits

(b) Standard Deviation of Accuracies of Digits

(c) Mean Training Times of Digits

Figure 1: Perceptron digit classification results over training sets



(a) Mean Accuracies of Faces

(b) Standard Deviation of Accuracies of Faces

(c) Mean Training Times of Faces

Figure 2: Perceptron face detection results over training sets

100% of the data, we achieved a mean accuracy of 76.30%, while for Face Detection, using 100% of the data, we achieved a mean accuracy of 87.07%. As both mean accuracy graphs depict, the model has a steep learning curve in the early training sets and it starts to level off as it reaches 100% of the training sets. This suggests that more data available the better it will learn however there is somewhat of a ceiling as it eventually levels off at a certain accuracy and more data could potentially make it worse due to over-fitting. This can also be explained by the standard deviation graphs as they both depict a gradual decline in the standard deviation as the model starts to level off at larger training sets. The time also depicts an upward trajectory as expected however there are dips and peaks due to early abandoning. Nonetheless, the Perceptron model works pretty well and relatively quickly.

4 Neural Network

The 2 layer Neural Network is very similar to the Perceptron by how it is structured where the main difference is that it has a hidden layer between the input and output, whereas the Perceptron does not. The Perceptron has one set of weights connecting from input to output, whereas the 2 layer neural network has 2 sets of weights and biases that connect from input to hidden, then from hidden to output.

However how the neural network calculates values for the hidden layer and the output layer and how it adjusts weights and biases to learn is very different.

4.1 Neural Network Algorithm

In every fresh iteration of the model, we start with two randomized sets of weights, W1 and W2. W1 is the weight matrix that connects input layer, A1, to the hidden layer, A2, therefore it is of dimensions hidden layer size x input layer size. W2 is the weight matrix that connects A2 to the output layer, A3, therefore it is of dimensions output layer size x hidden layer size. In forward propagation, we add a bias of 1 to A1 and adjust A1 and W1 accordingly. We matrix multiply W1 x A1. We get Z1 which we run through an activation function, which in our implementation is sigmoid, and get A2. We do the same thing for the second layer. We matrix multiply W2 x A2 to get Z2 and apply a sigmoid function as well to get A3. A3 is our output layer which is essentially our predictions. If our prediction is correct, we don't need to back-propagate and can move on the next image. When the prediction is incorrect than we have to back-propagate.

In back-propagation, what we are fundamentally doing is minimizing the cost function. We do this by taking the gradient of the cost function with respect to each weight. While this seems overwhelming, it can be broken down into simpler matrix calculations. What we want to do first is find the cost of each node in the hidden and output layers. For the output layer, we simply do this by subtracting the one hot encoding of the label from the output vector. We call this result dZ3. Next we matrix multiply W2.T x dZ3 and multiply the result with (A2 * A2-1). This calculates dZ2 which is the cost of the nodes in the hidden layer. With these values we can calculate the gradients for each weight matrix. For L1, which is the gradient matrix for W1, we calculate it by multiplying each value of A1 with the values of dZ2. L1 should be of the same size as W1 as each element in L1 corresponds to a weight in W1. We do the same thing for L2, which is the gradient matrix for W2. We calculate every element of A2 being multiplied with each element in dZ3. Again L2 should be of the same dimensions as W2 as each element in L2 corresponds to a weight in W2.

Now with gradient matrices calculated all that is left is to update the weights and biases using them. Before we apply the gradients matrices, we first regu-

larize them. We first divide each value by n, which is the length of the training set. Then for all weights that are not for the bias unit, we multiply with W1 and W2 respectively, along with a hyper parameter lambda, we chose as 0.001. Finally we multiply the regularized gradient matrices by a learning rate and subtract them from the weight matrices. This process is essentially how the weight matrices are updated and the model learns.

4.2 Neural Network Implementation

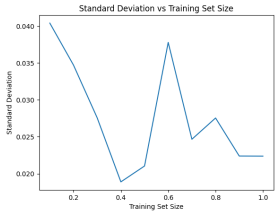
In our implementation, the user inputs the training, validation, and testing data and their respective labels, along with size of hidden layer, number of epochs, learning rate, early abandoning window size, and size of the output vector. We executed the same training and experimentation process as for Perceptron meaning taking 10 iterations on each training set, choosing the best model to predict in each iteration, and then taking the mean of the predictions for in each training set.

For Digit Classification we found that feature engineering the data worked well enough. Otherwise it would take a relatively long time to produce similar results. The feature engineering was a simple counting the number of 1s or white pixels in each 2x2 grid. This reduces the size of the input size of each image in training, validation, and testing from 784 to 196. We found that a learning rate of 0.01 and a hidden layer size of 128 worked well. We set the epochs the 50 but with early abandoning of 3 epochs of no growth. This usually terminates the training much earlier than 50 epochs. The output vector size remains a 10 for the 10 different digits.

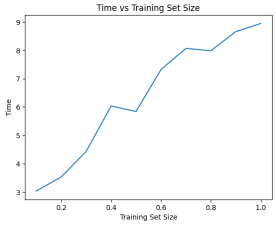
4.3 Results

With that being said Figure 3 shows the Digit Classification results through the Two Layer Network model. As shown by Figure 3, Digit Classification achieved a little more than 70% accuracy, with an exact accuracy of 72.68% at 100% use of the training data. Similar to Perceptron, it learned fast in the early training sets suggesting that more data leads to better accuracy. However it starts to level off at around 50% of the training data. This suggests that there is a ceiling for accuracy and that more data does not exactly mean more accuracy. This can also be explained by the standard deviation showing a steep drop of deviation of the accuracies as the training set gets larger. This suggests that model learns quickly in each epoch as the training set gets larger but hits the ceiling quickly as well calling for early abandoning and therefore a relatively small standard deviation. The time is plotted as expected showing more time elasped as the training set gets larger. The dips and peaks can be explained by early abandoning causing some training to last longer than others, but nonetheless depict an upward trend.

(a) Mean Accuracies of Digits



(b) Standard Deviation of Accuracies of Digits

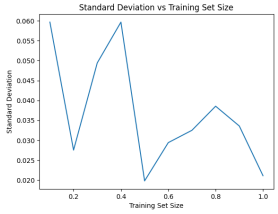


(c) Mean Training Times of Digits

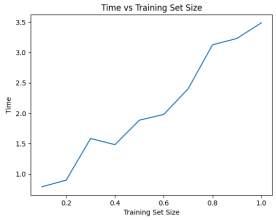
Figure 3: Two Layer Neural Network Digit Classification results over training sets

For Face Detection, we used the same approach as for Digit Classification, and were able to get a higher accuracy, achieving 84.67% accuracy at 100% of the training set. We feature engineered to reduce the size of the input vector from 4200 to 1050 while still maintaining high accuracy. We found that a learning rate of 0.1 and a hidden layer size of 256 worked well. Just like Digit Classification we set the epochs to 50 but utilized early stopping. This time we set the early stopping window to 5 which we thought worked better for Face Detection as opposed to 3 like in Digit Classification. The output layer size is set to 2 accordingly. With that being said, Figure 4 shows the results we got from our model and implementation for Face Detection.

(a) Mean Accuracies of Face Detection



(b) Std Deviation of Accuracies of Face Detection



(c) Mean Training Times of Face Detection

Figure 4: Two Layer Neural Network Face Detection results over training sets

As shown in Figure 4, the trends that can be seen in the Mean Accuracies and Mean Training Times graphs are nearly identical to those for Digit Classification. In the Mean Accuracy line graph, as the training data size increases so does the average accuracy of the model. This graph, like it's twin in Figure 3, also falters after it reaches a peak accuracy with partial training data, indicating the same insight from before that the quantity of training data is not everything. In the Mean Training Times graph the trend we see is the same as its Figure 3 counterpart as well, where the amount of time required to train the

model increases linearly with the size of the training dataset. The peaks and valleys here are explained by early abandoning as well.

The standard deviation graphs are a bit more complex. The main trend is that the standard deviation starts very high but ends very low. This indicates that, given more training data, the model is able to become less sensitive to variations, or "noise" in the data. The downward slopes throughout the graph are all indications of this positive development. The upward slopes, or "spikes" in the standard deviation, provide additional evidence that more training data does not necessarily mean more accuracy. They also give us insight into the nature of data and machine learning. The cause of spikes is the unpredictable and unknown nature of collected data. The spikes are most likely due to unpredictable variation in the data that does not directly relate to either trend shown in the other two graphs. Since the algorithm is attempting to generate rules out of just data, it starts off by looking at all data equally. The algorithm does not know which data or trends are relevant and which ones are not, so it essentially follows every path/trend/pattern that appears. Randomizing the initial weights is simply a way of making this blind approach more computationally efficient. An increase in standard deviation happens when the subset of the training data corresponding to the current iteration appears to not have strong trends. The variance increases because there is a lack of understanding of what binds the data together(or the "rules" governing the data), which makes it hard to predict. As a disclaimer, we are personifying the algorithm as a way to make the explanation simpler and clearer. Essentially, when the algorithm struggles to identify trends, the standard deviation increases, and when it is able to find rules and verify that they hold, the standard deviation decreases.