Project -1

Handwriting Digit Recognition with a Multi Layered Perceptron

Submitted by **Achyut Ganti** (G01676643) and **Sizen Nuepane** (G01674480)

The main goal of this project is to code and train a Multilayered Percpetron to classify the digits from the MNIST dataset. The language of choice was Python and the code has been written using the Numpy and Pandas framework available in Python. The training dataset used to train the model has 40000 observations in it and the test data has 2000 observations in it.

Let us answer the following questions first.

1Q. Describe the code. What approaches or other techniques did you use to improve the efficieny of the code.

For both and the Sequential technique, The network consists of 784 input neurons, 500 hidden neurons and 10 output neurons.The code consists of several functions that works overall to build the multi-layer back propagation neural network. The functions used are:

```
a) Initialize_network(args):
```

This function assigns the random initial weights to the hidden layer and output layer. In both cases, the weight is also assigned to the bias.

```
b) Forward_propagate(args):
```

This function calculates the output from the hidden layer and output layer. The matrix multiplication is used for fast computation. The output from each layer is sent to the sigmoid function to find the sigmoid approximation.

```
c) Sigmoid(df):
```

Each output from hidden neurons and output neurons are sent to the function to calculate the sigmoid approximation.

```
d) Backward_propagate_error(args):
```

This function calculates the error associated with the outputs from forward propagation. The errors are calculated for both output from output neurons and hidden neurons.

```
e) Deri_output(args):
```

This function calculates the difference in the expected output and predicated output in output unit. This function is called from backward_propagate_error function.

```
f) Update_weights(args):
```

This function updates the weights of network proportionately according to the calculated error.

For the batch technique

The code consists of several functions that collectively helps in building the neural network. Some of the functions are

- `scaling(dF)`

Takes in a dataframe and normalizes it.

- `initialize_weights(args)`

Used to Initialize the initial weights that initialize the neural network.

- `sigmoid(args)`

A sigmoid function that takes in a ndarray and applies the sigmoid function and returns the sigmoid approximation of that ndarray.

- `randomize(df)`

This is used to shuffle the data after every iteration. It takes in a dataframe and returns the shuffed dataframe.

- `forward_prop(args)`

This is where the forward propogation for the neural network happens. It takes in the input data and the weights as inputs and gives us the final Value of the neural network as the output along with other values that are obtained as a part of the forward propagation.

- `error_function(args)`

Calculates the errors associated with the outputs that were obtained by the forward propagation function. The outputs obtained are total error of the neural network, the error of the output unit of the neural network, and the error of the hidden_layer of the neural network.

- `update_weights(args)`

This function is used to update the weights . It updates the input_weights, output_weights and as well as those associated with the bias units.

After training is done, it is finally time to test the model on the test data and predict the output labels. And then we calculate the accuracy associated with the model.

Some of the techniques that were used to improve the efficiency were

- **Scaling the data** - We had to scale the input data before using it to train the model. We have tried training the model without scaling the data and encountered an overflow warning which eventually converted all the output values that are closer to one to one. Further operations are impossible when tha happens. Could take care of that problem by scaling the data.
- **Numpy array and matrices** - A very convinient and efective methods when we have larger number of hidden neurons or layers is by storing the data in the form of matrices and carrying out vectorized operations on them. This is many times effctive that looped operations especially when dealing with problem like building a neural network and dealing with datasets of that size.
- **Vectorized multiplicaion** - As mentioned above, run and training time of the progrm was drastically reduced using these opeartions.
- **Reducing the feature size** - We tried to observe those pixels that had no information in them and try to eliminate them. But it didn't make much of a difference in our case, so we just left this option out. but reducing the input size is a good option when you have really big feature vector size because when we are acheiving better results with reducing the complexity of the data, then why not?.
- Also, initializing the weights using smaller values proved beneficial especially for when using Batch technique.

2Q. MLP Architecture

Our Multi-layered perceptron has three layers. The first one is the input layer which has 784 units, i.e., feature vectors as the input for the neural network initialization. the seond layer is hidden layer which had 500 hidden neurons in it. We were just curious as to see what happens as we consider different values for the hidden neurons. Also, viewed a few links where there were some thumb rules for considering the number of hidden neurons. The link will be in the reference section below. And the third layer that is the output layer has about 10 units in it. Because the problem we are dealing with is a 10-class classification problem.

The learning rate that we considered was 0.5. Weight decay - We didn't consider any weight decay as we didn't find any problems in the weight update without considering it too.

The upper limit on epochs we considered was 15 for sequential and 2000 for batch. For sequential, 10 epochs, 13 epochs and as well as 15 epochs showed not much difference in the results. So we didn't train the algorithm after 15 epochs.

The training size was 31000 observations, the validation was 9000 observations and the test size awas 2000 observations.

Randomly initialized runs - We initialized only once.

Some of the problems that we faced when building the neural network were:

```
-We faced overflow problem for both sequentaial and the batch technique when we train
ed the model using the unscaled data. This problem wsa solved for the sequential tech
nique when we scaled the data but for the batch technique it persisted.
- Later on we found out that weights we considered for the batch were considerably la
rge. Reducing the size of our initialized weights solved it.
- In our case, the sequential technique took longer time to train and update when com
pared to the batch technique but the accuracy of the batch was really far better when
compared to the batch technique.
```

3Q. What's the effectiveness of the classifier.
Our classifier after training on our training data gave 97.3% accuracy and when tested on the testing data gave 96.2% accuracy.

For 10 epochs the training accuracy was 97.3% and for 15 epochs the accuracy was about 97.32%. Batch gave about 50% accuracy when it was tested. It did worse when compared to the sequential technique.

4Q. How long does it take to train the model.

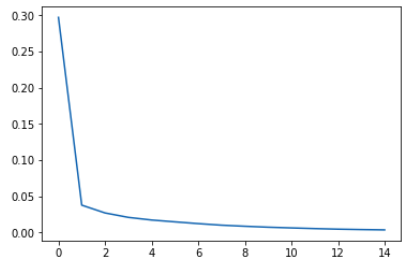
Depending te size of the training data that we considred and the number of iterations we made it run, our model takes anywhere in between 1 to 1.5 hours to learn the input data. But the batch operation was a bit faster than the sequential one but the accuracy is better for the sequential technique.

5Q. Did you choose batch, sequential, or mini-batch training? Why?
The methods we chose to train the data were batch and sequential. No particular reason why we chose these. We felt like these two techniques are the extremes in model training. One method considers the whole data as a batch in one iteration whereas the other takes one datapoint at a time. We also wanted to try mini-batch but couldn't because of the time constraint. So we decided mini-batch would be for futher investigation.

Some analysis and discussion of the results obtained from the MLP
The main goal of this project was to obtain those perfect weights that will can be used to predict our digits accurately. So, we randomly initialized our weights with smaller values, somewhere in between 0 and 1. As we keep iterating the network, we forward propagate the network using these weights and the input values to caluate the output value of the neural network. But as those values are not the optimal ones, we backpropagate it by calculating the errors for the total network, the output units and the hidden units. And by using these values as reference we keep updating the weights. All this is done several times in a loop till we get obtain those near perfect weights that can be used to predict the labels on our test dataset. And then we calculate the errors and the accuracy of our model.

In [8]:
```
PATH = "/home/achyutganti/Downloads/"
Image(filename = PATH + "error.PNG", width=4000, height=4000)
```

Out[8]:




```
[0.29709651011321803, 0.037558534211762175, 0.026768876076311642, 0.020647834506463701, 0.016988876151457971, 0.014459167701449
598, 0.012019362149629178, 0.009856228178793944, 0.0083651560339857555, 0.007063527763603942 2, 0.006069084671264133 5, 0.005140
37304810865, 0.0043482524813573493, 0.0037913947663228953, 0.0033569914884531188]
```

Refernce Links

- https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw (https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw)