Lab-4 Report

In this report, I talk about implementing the Multilayer Perceptron in C and a test to check its correctness and feasibility on two data sets. One for classification and the other for regression purposes.

In this lab, I learnt the following

- Neural Network
- Activation Functions
- Feed Forwarding the network
- Backpropagation Algorithm
- Gradient Descent and it's variants.
- Learning Parameter
- Training a neural network
- Testing the trained model
- Need for normalising data

I first discuss the internal logic and design of the library, then I discuss the tests and their results, followed by graphs and data values section. I then also discuss the possible use improvements in the library to make it a more robust and better implementation of the Multilayer Perceptron Classifier.

The library (mlpClassifier.so) consists of below general functions, and provides the struct used for the MLPClassifier.

1. Struct MLPClassifier
2. GetMLPClassifier(..)
3. TrainModel(...)
4. ClassifyData(...)
5. PrintModelWeights(..)
6. ReadInputData(..)

Rest of the functions remain abstracted to the user and are present in the implementation file, "mlpClassifier.c", present in the "src" directory.

There is a "test.c" file, which captures the implementation of the library usage, with the help of command line arguments to the binary constructed from this source file.
The command line arguments are described in the usage of the binary.

Forward Propagation

This consists of calculating the output values of each node present in our neural network from an instance of the inputs.
In my implementation I have done it in the following manner :

- Allocated Input and Output cache for each node.
- InputCache is calculated from InputCache[i][0] to InputCache[i][lsize-1] , for each layer “i”, and number of nodes in the layer being represented as “lsize”.
- OutputCache is calculated from OutputCache[i][0] to OutputCache[i][lsize]. Therefore OutputCache for each layer consists of “lsize+1” data points. This is because I have used OutputCache[i][0] as the Bias node, for the next layer.
- Bias node for each layer remains 1, however the bias weights will be different.

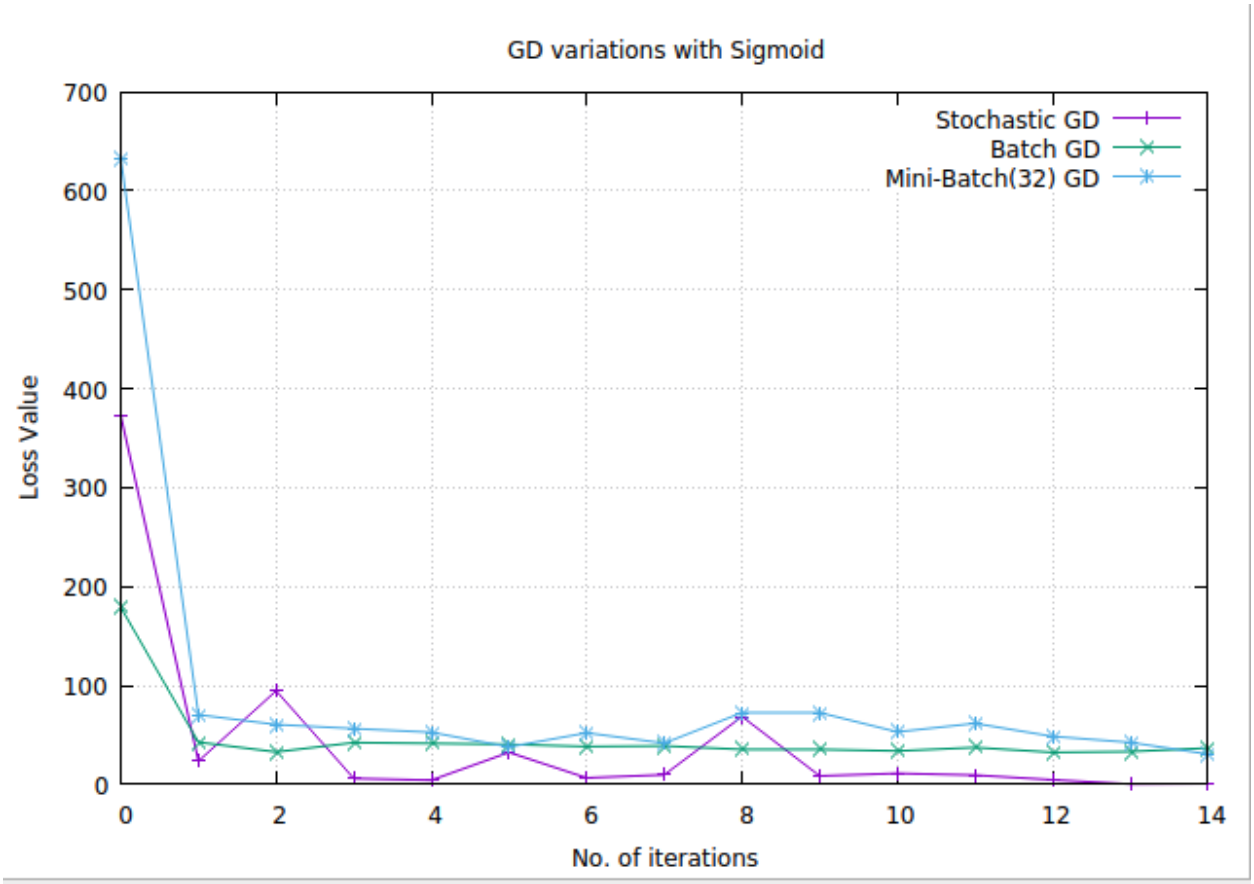
Backward Propagation & Weight Correction

This consists of learning of the neural network from the errors caused in prediction and deviation from the expected values. For the weights, I have constructed a 3-Dimensional array, Weights[n-1][lsize-1][lsize] , where n denotes the number of total layers in our neural network. Users of the library can use different activation functions mentioned in the next section and can use 3 variants of Gradient Descent as Backpropagation algorithms.

Back Propagation Algorithms

1. Stochastic Gradient Descent
1. Batch Gradient Descent
2. Mini-Batch Gradient Descent

I observed that the graphs for Batch Gradient Descent were smoother in comparison to Stochastic Gradient Descent. This can be due to the fact that in Batch Gradient Descent, we iterate over all the training data and then update the weights with the mean weight correction calculated. Thus, the loss value gradually goes down without much noise. But for stochastic, since weights are updated after every instance of training row, there can be ups/down in the graph based on the correction made. Hence the loss value curve is not very smooth.



Activation Functions

These are basically functions which take in an input value and produce an output value which helps the neural network learn on the basis of the output produced. Desirable properties of activation functions are its non-linearity, continuously differentiable, range, monotonic and approximate identity near the origin.

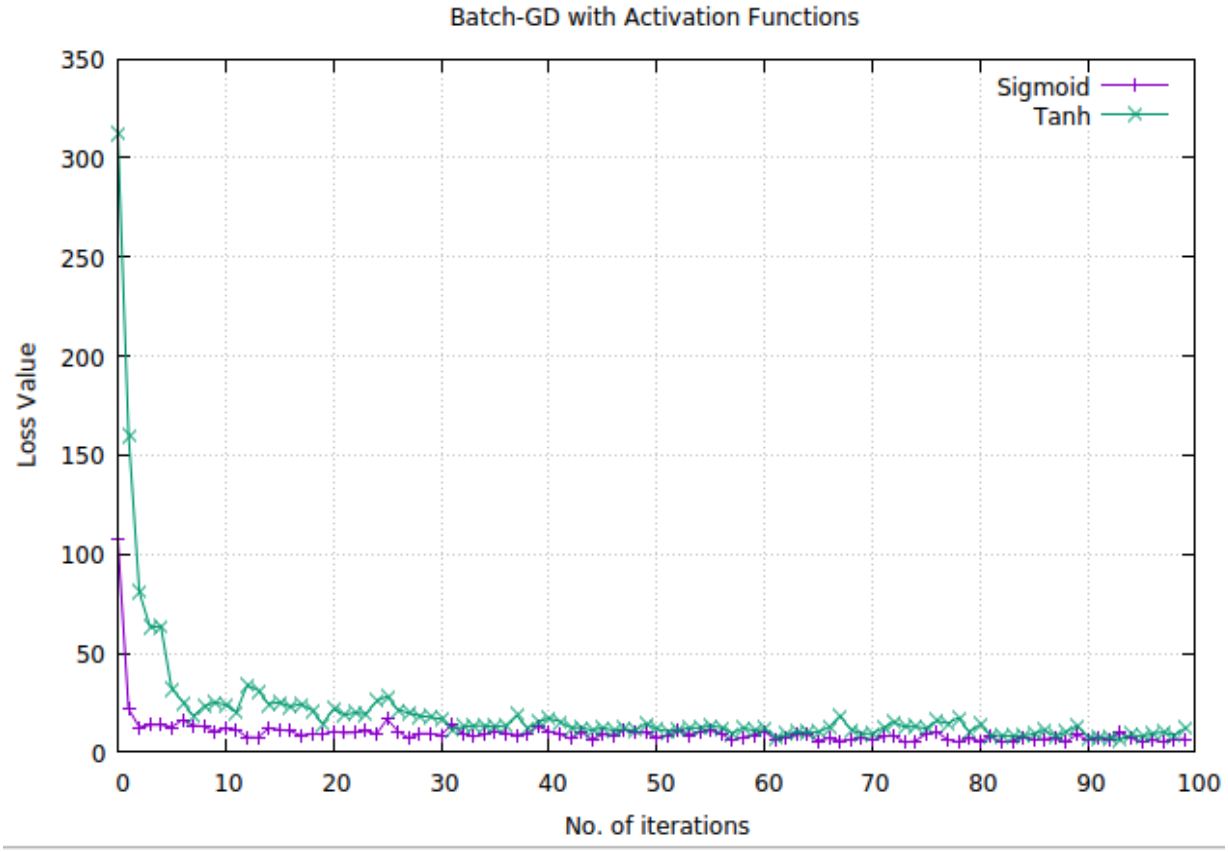
I have implemented 4 activation functions, as follows -

- Sigmoid
- Tanh
- ReLu
- Softmax (Used with Cross-Entropy)

Some observations made on activation functions :

- In the case of ReLu, if a node receives a negative input, then that node stops playing a role in the neural network.
- Softmax is not used in regression. It is used in classification, where each class would have its own probability of outcome.

- The Sigmoid function's learning curve for our first dataset was faster, in comparison to the Tanh activation function.



Testing the Library & Results

Classification (Data Set 1)

This data set consists of Breast Cancer Wisconsin. They describe characteristics of the cell nuclei present in the digitized image of a fine needle aspirate (FNA) of a breast mass.
Number of Features = 31 (including 1 for Output Feature)
Number of Classes = 2 ('B' or 'M')
Number of Instances = 569

I have changed the 'B' and 'M' classes to 0 and 1 (as integers) for the input to my library. Also, I have made another data which consists of normalised values from this data set.

Source : https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

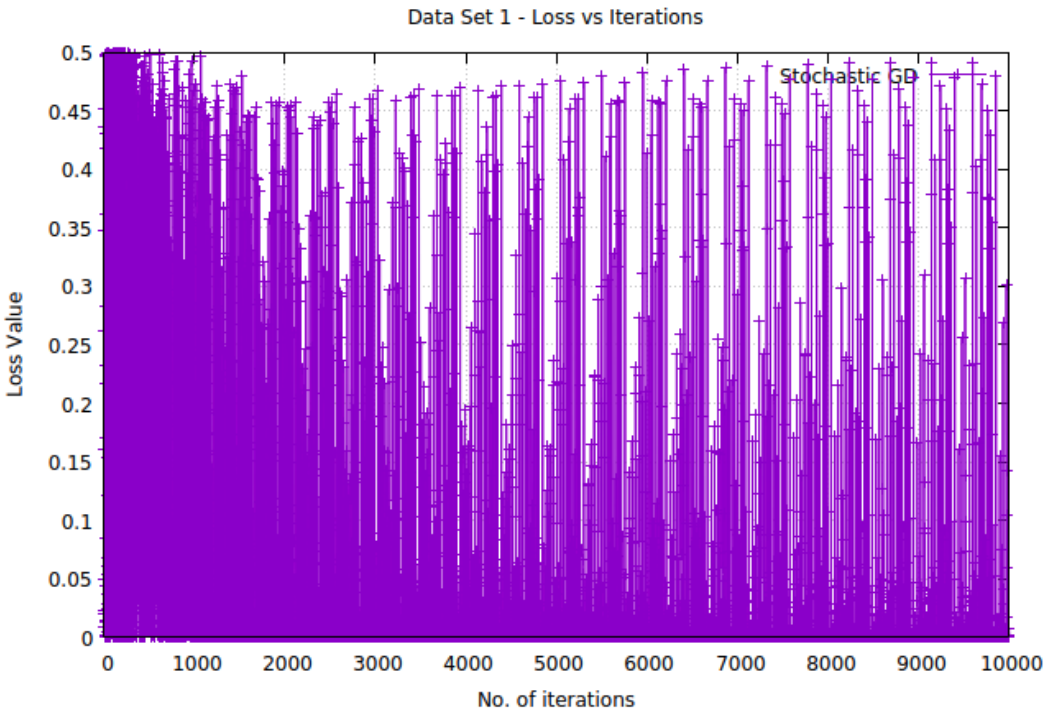
Observations & Results

It was found that in terms of speed,
Speed ----- SGD >> Mini-Batch GD > Batch GD
Average Accuracy ----- SGD fared best with Sigmoid as activation function.
[on multiple runs]

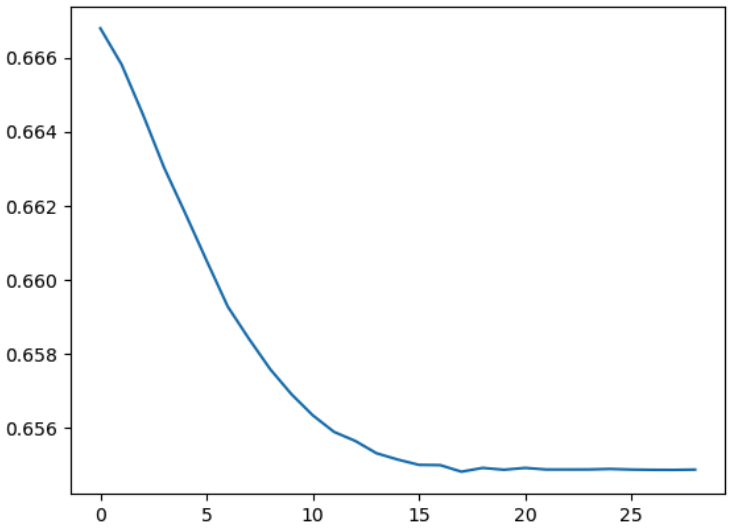
For this particular data set, it was observed that SGD was able to learn fast and its accuracy was much better than the others. In the case of Batch GD, it was learning very slow and lagged in the speed as well. This could be related to the large data we have to work on, before updating the weights.

It was found that Sigmoid trained faster, whereas Tanh took more iterations and time to reach the same estimate of accuracy.

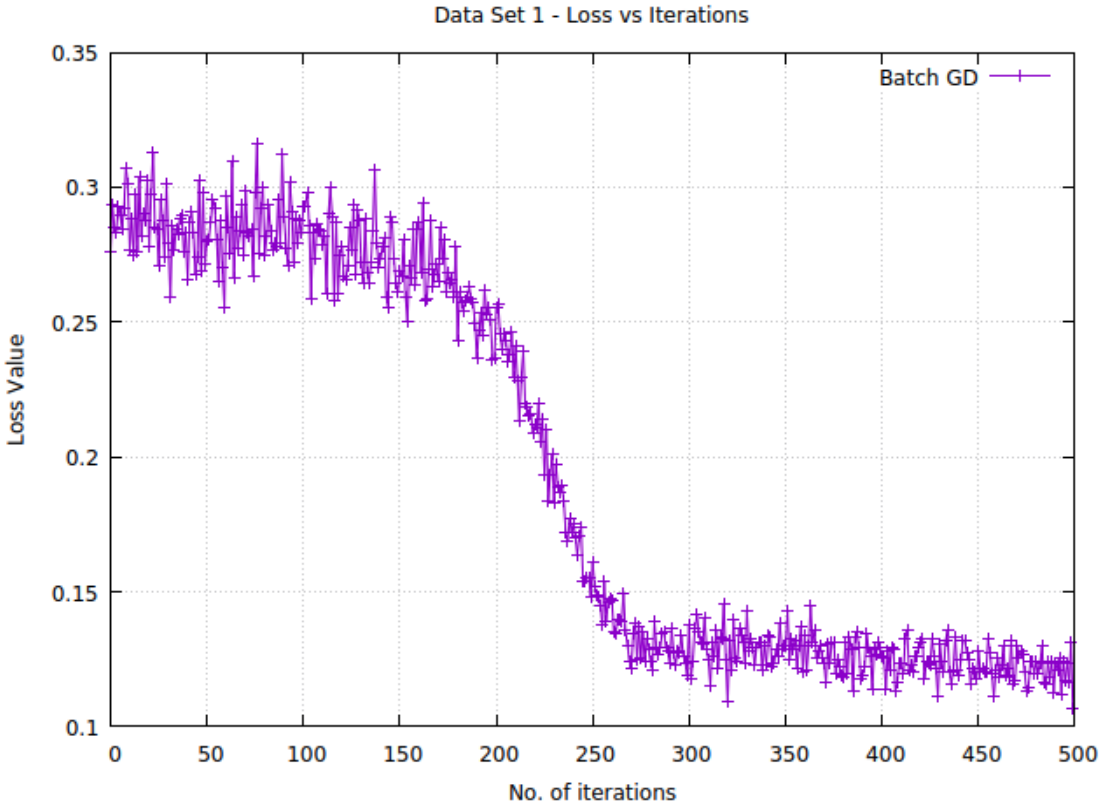
Backprop-Algo	*Activation Func*	*Iterations*	*Accuracy*	*Time*
SGD	Sigmoid	10000	97.97%	1.617s
SGD	Tanh	10000	46.4%	1.789s
SGD	Tanh	100000	74.7%	16.54s
Batch GD	Sigmoid	100	62.6%	4.5s
Batch GD	Sigmoid	500	77.7%	24.03s
Mini-Batch (32)	Sigmoid	1000	61.2%	3.8s
Mini-Batch (32)	Sigmoid	5000	90.1%	18.2s

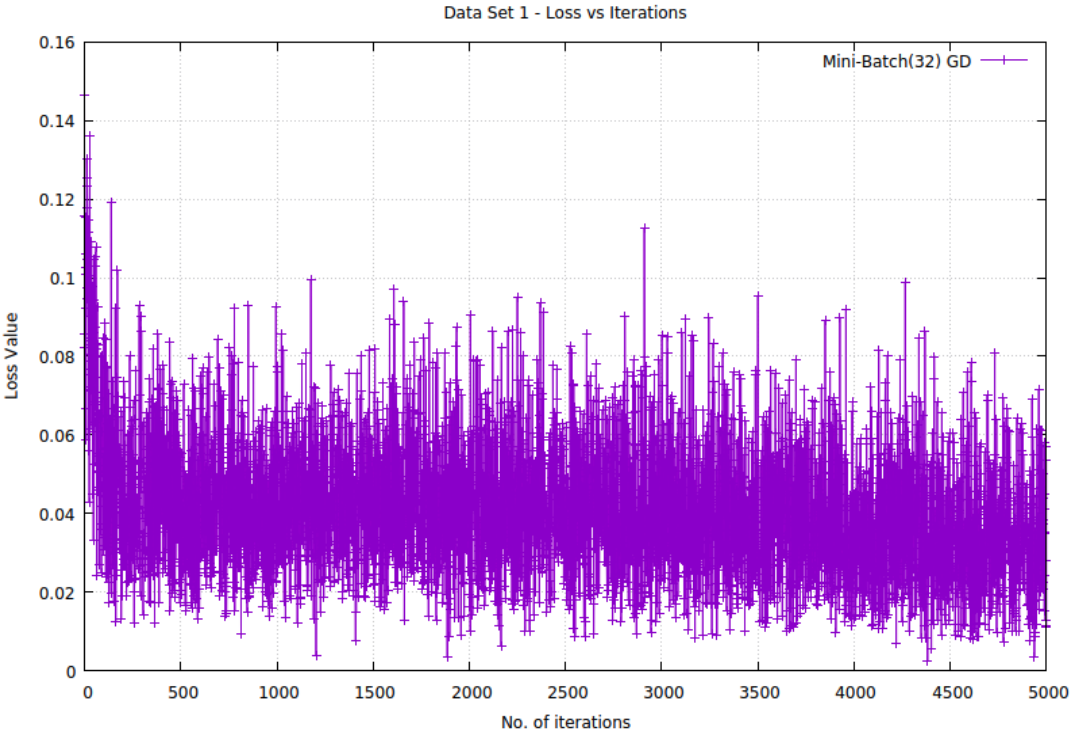


The images with purple lines, represent the loss function curve from the MlpClassifier built using C language. In the above image, we can see that overall the loss values are decreasing, but there are sudden jumps in the loss values, this is due to the behaviour of Stochastic Gradient Descent algorithm. In the next page, you will find smoother curves for Batch and Mini-batch gradient descent algorithms.



The image on the left, with the blue curve, represents the loss curve for MlpClassifier from Scikit-learn library in python.


Data Set 1 - Loss vs Iterations
Batch GD
Loss Value
No. of iterations
0.35
0.3
0.25
0.2
0.15
0.1
0
50
100
150
200
250
300
350
400
450
500


Data Set 1 - Loss vs Iterations
Mini-Batch(32) GD
Loss Value
No. of iterations
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
500
1000
1500
2000
2500
3000
3500
4000
4500
5000

Regression (Data Set 2)

This data set consists of information regarding houses in Boston.
Number of Features = 13 (including 1 for Output Feature)
Number of Instances = 506

Also, I have made another data which consists of normalised values from this data set.

Source : https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html

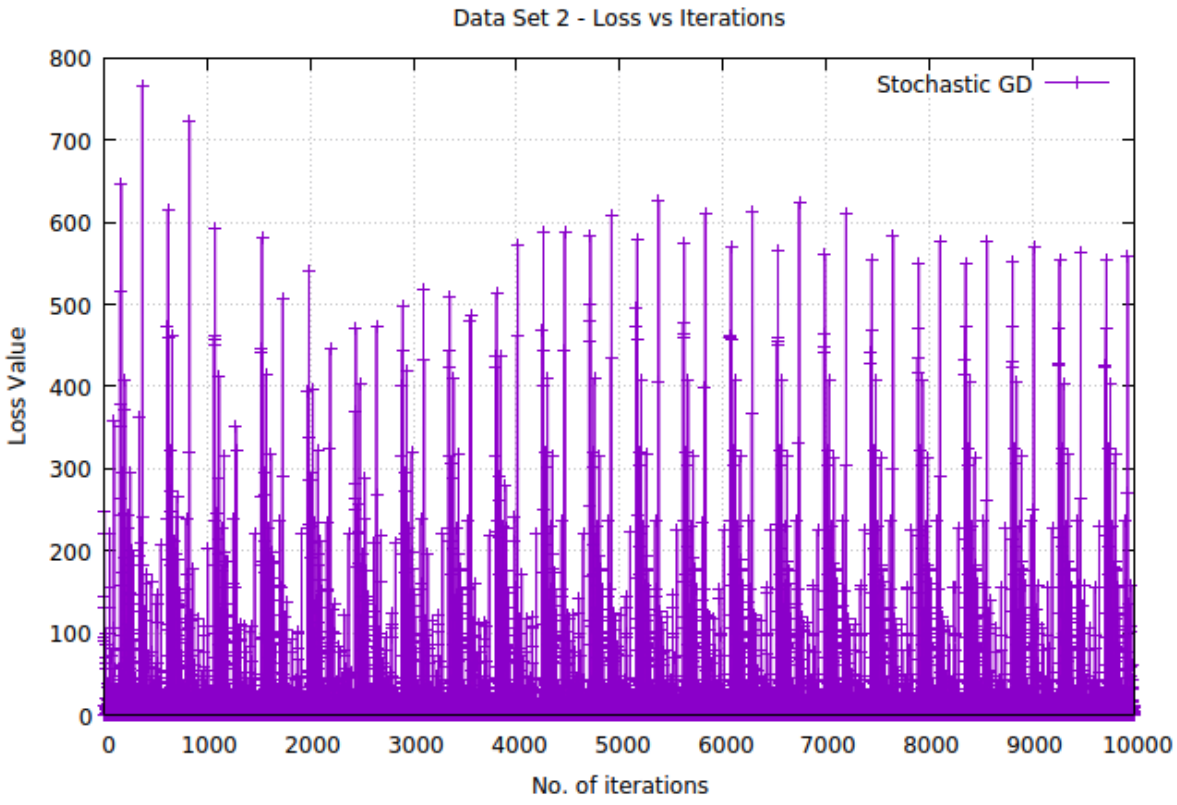
Observations & Results

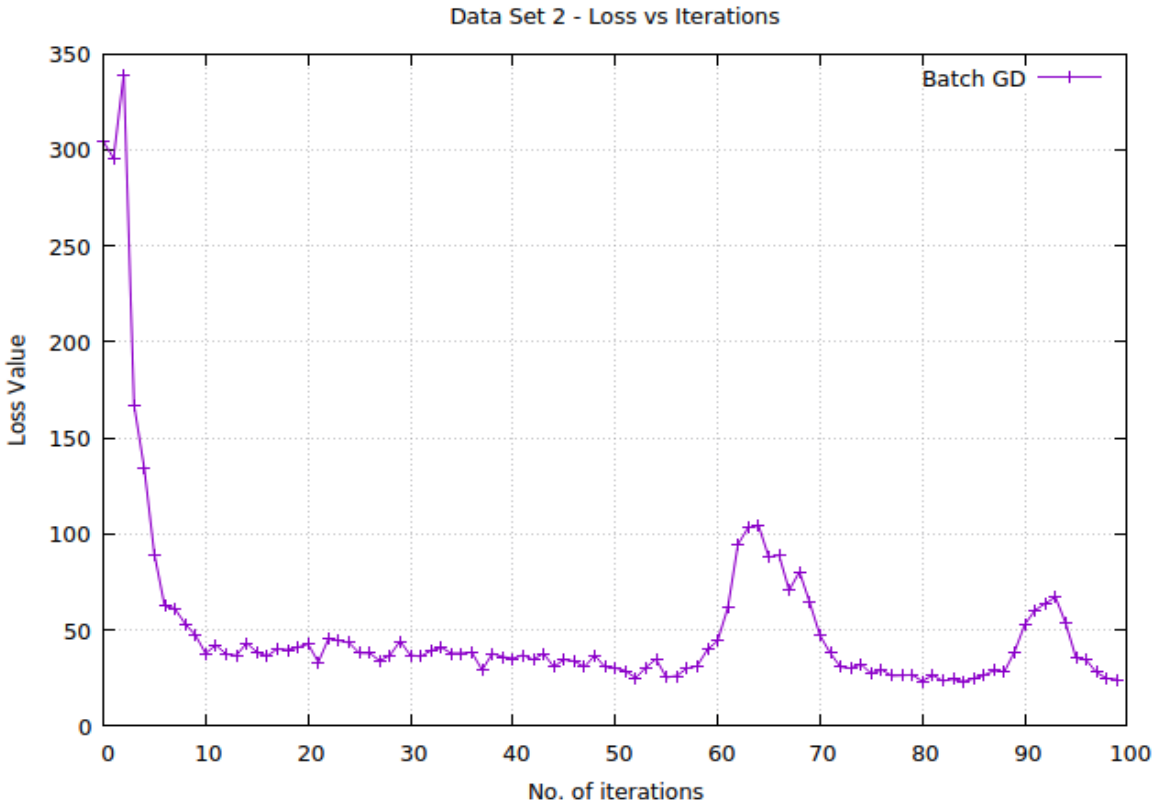
It was found that in terms of speed,

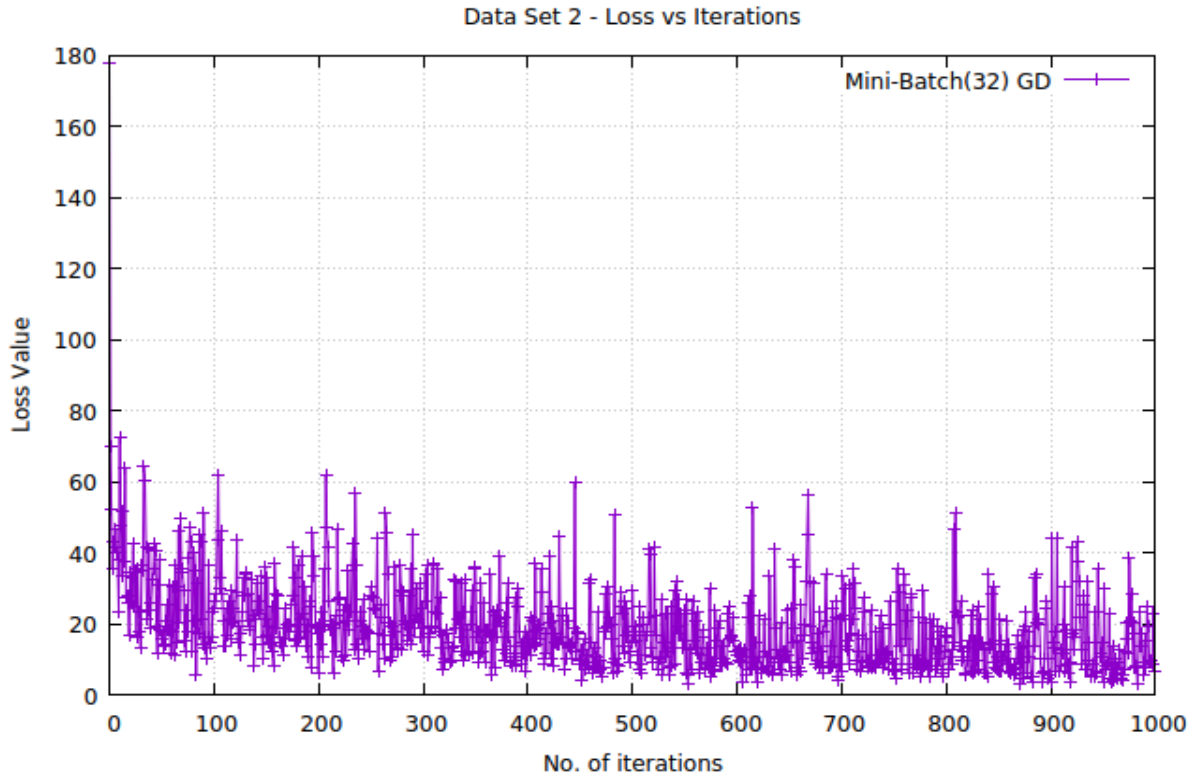
Speed ----- SGD > Mini-Batch GD > Batch GD

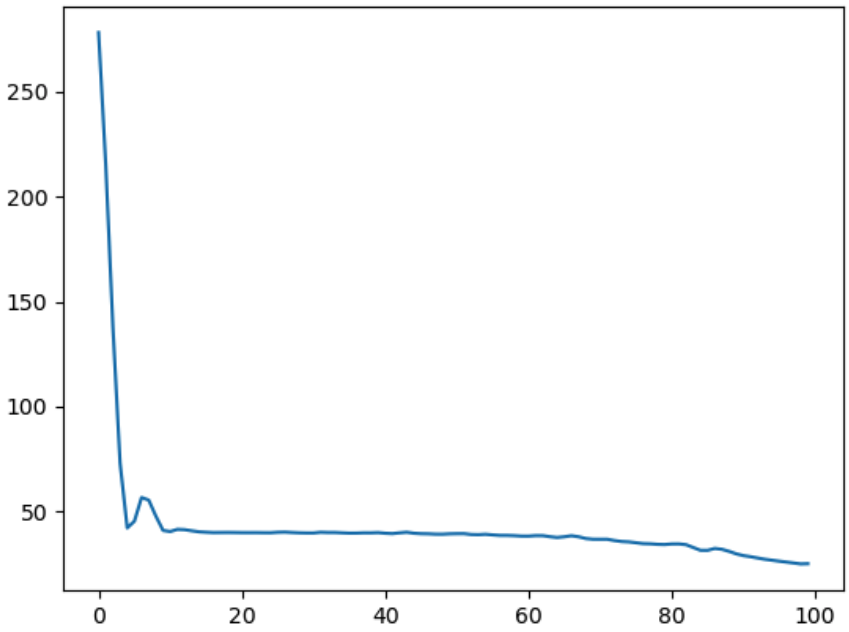
Average Accuracy ----- Batch GD fared best with this data set. (This could be due to missing data values in the data.)

[on multiple runs]




Data Set 2 - Loss vs Iterations
Batch GD
Loss Value
No. of iterations
0
50
100
150
200
250
300
350
0
10
20
30
40
50
60
70
80
90
100


Data Set 2 - Loss vs Iterations
Mini-Batch(32) GD
Loss Value
No. of iterations
0
20
40
60
80
100
120
140
160
180
0
100
200
300
400
500
600
700
800
900
1000



The image, with the blue curve, represents the loss curve for MlpRegressor from Scikit-learn library in python.
The images with purple lines, represent the loss function curve from the MlpClassifier built using C language.

Data

Data for the graphs and exact values are present inside the “data” directory. Since, the data points are large in numbers, I have skipped them and provided them as separate files.

(please turn over)

Important Observations :

1. When we train the model using the normalised values (data) then, the model is able to train faster. We were able to see that the model's accuracy reached above 95% when using normalised data.
2. When using data which was not normalised, the neural network seems to be not working properly. It was acting as if it was stagnant and its output was not depending on the input value. For each test, it predicted the same value. The same observation was made when the classifier was built from the python sklearn library as well.
3. When using the ReLu activation function, if the value at the node becomes negative, the learning stops for that node. It's like, the node gets deactivated.
4. To use cross entropy in classification problems, we need to make sure that there is an output node for each class. Then we use the softmax activation function on the output layer.

Further Improvements

One, can make the following improvements in the library to make it more robust:

1. Correction of Cross Entropy cost function by having two separate output nodes for binary classification.
2. Adding more robust error checks for user inputs and command line arguments.
3. Adding support for Custom Batch Size in Mini-Batch Gradient Descent Algorithm for backpropagation.

References :

1. Manohar Mukku's Implementation of MLP.
2. Sankhya Mondal's Implementation of MLP.
3. Towards Data Science blogs.
4. Scikit-Learn Documentation
5. Wikipedia.
6. GeeksForGeeks (for different articles related to lab)