Assignment 4

COP-701

Toy Neural Network

Name : Sourav Sharma

Entry No. 2021MCS2154

Department of Computer Science & Engineering

Indian Institute of Technology Delhi

1 Software Requirements

1. GNU Compiler Collections (GCC).
2. Git Version Control System.

2 Directory Structure

1. include
2. lib
3. obj
4. src
5. Makefile

3 Introduction to Neural Network

A neural network has it's basic building block that is called as neurons. Several neurons interacts with next and previous layer neurons for their state computations. These neurons are highly interconnected with a weight associated with each connection. The architecture of a multi-layer perceptron consists of input layers, hidden layers and output layer(s).

3.1 Types of Layers in Neural Network

1. **Input Layer** This layer contains the input data that is feed to it. There is basically no computation done on the input layer. It contains number of nodes equal to the number of features.
2. **Hidden Layers** These are the layers where actual computation is done via change in the weights. **In MLP there must be at least one hidden layer.**
3. **Output Layers** This layers give the output which we generate through the forward propagation. This is matched with the actual output and we then we computed how much changes we need to make in our neural network to make it predict things with less loss.

3.2 Connections and Weights in Neural Networks

There are connections in the neural network from the current layer to the next layer. These connections contains the weights. These is also a bias node. Bias nodes are added to increase the flexibility of the model to fit the data. Specifically, it allows the network to fit the data when all input features are equal to 0, and very likely decreases the bias of the fitted values elsewhere in the data space. The bias node in a neural network is a node that is set to 1 always without regard for the data in a given pattern.

4 Activation Functions

An activation functions takes a value as an input and produced another value as it's output. There are various activation functions that one can use. The activation function which are used in this projects are Sigmod, Relu and Tanh.

1. **Sigmoid function or Logistic function** This activation function maps the inputs from any range to the value in the range [0, 1].

$$f(x) = \frac{1}{1 + e^{-x}}$$

2. **Hyperbolic Tangent (Tanh)** This activation function maps the inputs from any range to the value in the range [-1, 1].

$$f(x) = \tanh x = \frac{2}{1 + e^{-2x}} - 1$$

3. **Rectified Linear Unit(ReLU)** This activation function if input is less than 0 then it map it to 0 always and if it is greater then zero then it act as function

$$ReLU(x) = x, if x >= 0$$

$$ReLU(x) = 0, if x < 0$$

4. **Softmax** This is the activation function which is used at the last layer in case of the MSE cost function calculation.

$$softmax = \frac{e^{z_i}}{\sum_{j=1}^{n} e_j^z}$$

5 Hyperparameters : Loss Functions and Learning Rates

1. **Learning Rates (α)** Learning rates is basically the tuning parameter. It determines the step size at every iterations while we are converging at the loss function.

2. **Loss Functions** This is also referred as the objective function and our goal in general is to minimize this. We generally use two different type of Loss functions and they are cross entropy and mean squared error.

   I **Cross Entropy** It is also referred as the log loss. This is used as a loss function when optimizing classification model are used. Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. Y represent actual output and a is the output produced by the classifier.

$$h(x) = -\frac{1}{n} \sum_{i=1}^{n} y_i.log(a_i)$$

II **Mean Squared Error** To compute MSE we take the difference between the predicted output and actual output and square it. Take the average of this across the whole data set.Here n is the size of data set and a and y represent the predicted and actual output respectively.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(a_i - y_i)^2$$

Here,

a$_i$ is the predicted output
y$_i$ is the actual output
n represent the train set size

6 Forward Propagation

Forward propagation is a method to calculate the intermediate variables i.e z values(linear transformation of given input and activation values for a neural network in order from the input layer to the output layer.
For the input layer the activation value are same as the input feed to it.

Some Notations:

1. b^l : Bais vector for the lth layer
2. W^l : Weight matrix for the lth layer
3. Z^l : Linear transformation of given input value to the lth layer
4. σ^l : Activation function applied on lth layer
5. A^l : Activation value for lth layer

To compute the value of the A^l We first need to compute the value of Z^l.

$$z^L = W^{L-1} * a^{L-1} + b^{L-1}$$
$$a^L = \sigma(z^L)$$

where W$_{ij}^{L-1}$ represent the weight matrix of the layer L - 1, which is from the neurons i of layer L - 1 to the neurons j of the layer L. Same is the case for the bias. z$_i^L$ represent the input value of the neuron i in the layer L. And σ^L represents the activation function for the layer L. a$_i^{L-1}$ is the activation value of the layer L - 1 for the neuron i.

For the last layer or the output layer we find the cost function. If it is the MSE which is used in case of classifier models we use softmax function and in the case of CE which in general used with regression models we use sigmoid.

7 Back Propagation Algorithms

It is used to change the value of the weight on the connections and adjust them accordingly. It calculates the effect of change in cost function by change in the a value and z values of the output layer. It allows information to go back from the cost backward through the network in order to compute the gradient. It is done to fine-tuning of the weights based on the loss value. A good tuning of these weight ensures the lower values of the loss value for the testing as well as the training data.

Notation used below

Derivative of cost function w.r.t Z^l : $\frac{\delta C}{\delta Z^{(l)}}$

Derivative of cost function w.r.t W^l : $\frac{\delta C}{\delta W^{(l)}}$

Derivative of cost function w.r.t b^l : $\frac{\delta C}{\delta b^{(l)}}$

Derivative of cost function w.r.t A^l : $\frac{\delta C}{\delta A^{(l)}}$

Using Chain Rule:

$$\frac{\delta C}{\delta W_{jk}^{L}} = \frac{\delta z_j^L}{\delta W_{jk}^{L}} * \frac{\delta a_j^L}{\delta z_j^L} * \frac{\delta C}{\delta a_j^L}$$

$$\frac{\delta C}{\delta b_j^{L}} = \frac{\delta z_j^L}{\delta b_j^{L}} * \frac{\delta a_j^L}{\delta z_j^L} * \frac{\delta C}{\delta a_j^L}$$

$$\frac{\delta C}{\delta a_j^{L-1}} = \frac{\delta z_j^L}{\delta a_j^{L-1}} * \frac{\delta a_j^L}{\delta z_j^L} * \frac{\delta C}{\delta a_j^L}$$

7.1 Stochastic Gradient Descent

It's a method which is used to for optimizing an objective function with the use of derivatives. The **algorithm** proceeds as follows. This what we perform in one **epoch**.

Algorithm for SGD

1. Partition the data set into training and testing data set.
2. Input the training data row by row to the neural network. If the training data is huge then only pick a set of random training rows from whole set of training data.
3. Compute the gradient of loss function w.r.t w$_{ij}^L$, b$_j^L$ and a$_j^L$. This should done after the each example is done with it's forward propagation step.
4. Once gradient are done, update the weight matrix and bias matrix.

$$W_{jk}^{(L)} = W_{jk}^{(L)} - \alpha * \frac{\delta C}{\delta W_{jk}^{(L)}}$$

$$b_j^{(L)} = b_j^{(L)} - \alpha * \frac{\delta C}{\delta b_j^{(L)}}$$

5. Repeat these steps for some random set of data point from the training set.

SGD generally converges faster than any other method and the reason is we have huge number of back propagation step in each epochs. For each iteration of forward propagation we have one backward propagation step also. If the data set is huge then computation might take more time and thus we pick points from the training set **stochastically** or randomly. If data set is small then this can be even proceed with running on every training data set.

7.2 Batch Gradient Descent (BGD)

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that's just one step of gradient descent in one epoch.

Algorithm for BGD

1. Take the complete training data set.
2. Run forward propagation for each training example by feeding it to the neural network.
3. Calculate the mean gradient of the whole batch.
4. Use this mean gradient to update the weights.

7.3 Mini Batch Gradient Descent

Mini Batch Gradient Descent is a hybrid approach to the SGD and BGD. In this method we break our training data set into **mini batches**. Each batch consists of some training example. In one epoch we take these all mini-batches and run forward propagation for each training example in the current mini batch and then afterward we run a single time backward propagation over the mean gradients. This methods take the advantages of both Stochastic Gradient Descent and Batch Gradient Descent.

Algorithm for MBGD

1. Decide the number of batches.Logically partition the training data into mini batches.
2. Feed the mini batches to the neural network.
3. Calculate the mean gradient of the mini-batch.
4. Use this mean gradient to update the weights in the W matrix and B matrix.
5. Repeat the above step for each mini-batch that is generated.

8 Observation and Plot of Loss Functions

8.1 Testing accuracy comparison between Python and C Library.

Table 1: Table for Testing accuracy in Stochastic Gradient Desent

	Sigmoid	**ReLU**	**Tanh**
Python	80.216	82.452	84.211
C Library	90.354	87.719	86.710

Table 2: Table for Testing accuracy in Batch Gradient Desent

	Sigmoid	**Tanh**	**ReLU**
Python	86.842	89.476	87.719
C Library	85.913	85.961	85.087

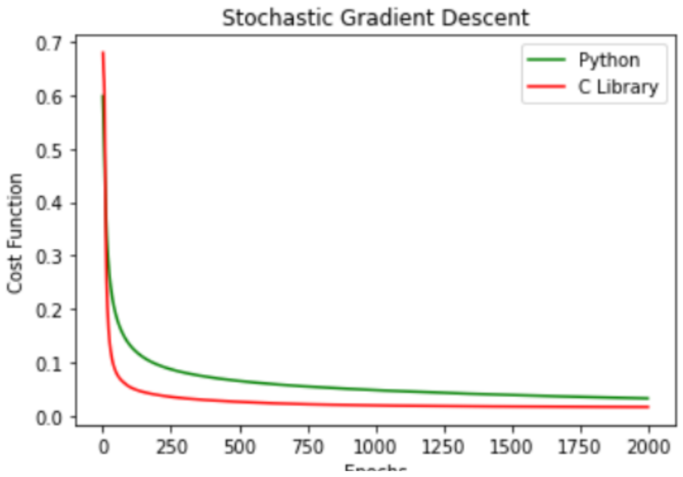
Table 3: Table for Testing accuracy in Mini-Batch Gradient Desent

	Sigmoid	**ReLU**	**Tanh**
Python	86.399	87.719	88.596
C Library	93.854	88.5964	88.519

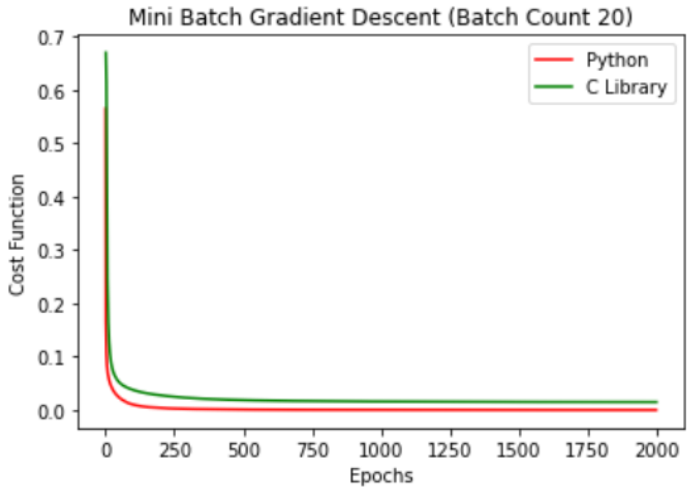
8.2 Plot of Loss Function for Classifier & Regression

Parameters for different activation functions (ReLU, Tanh, Sigmoid)

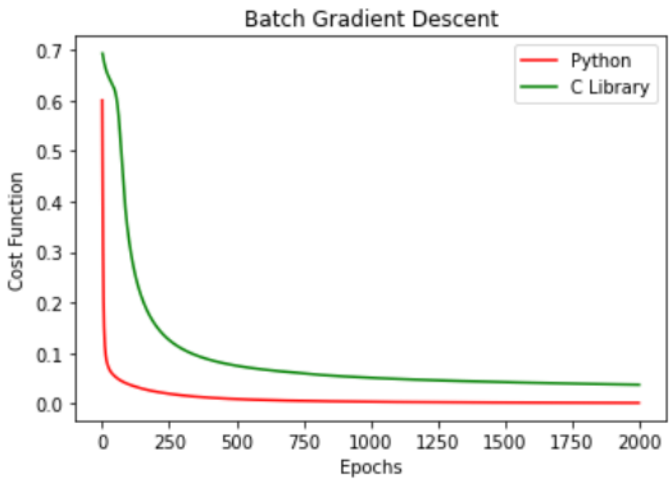
- Number of Iterations : 2000
- Learning rate (α) for Classifier : 0.001 for stochastic and mini-batch, 0.05 for batch gradient.
- Learning rate (α) for Regression : 0.000001 for Stochastic and Mini-batch and 0.001 for batch gradient.
- Number of Hidden Layers : 1
- Number of neurons in hidden layer : 50
- Partition : 80% for Train and 20% for Test
- Number of Batch in Classifier : 20
- Number of Batch in Regression : 25



(a) SGD for Classifier

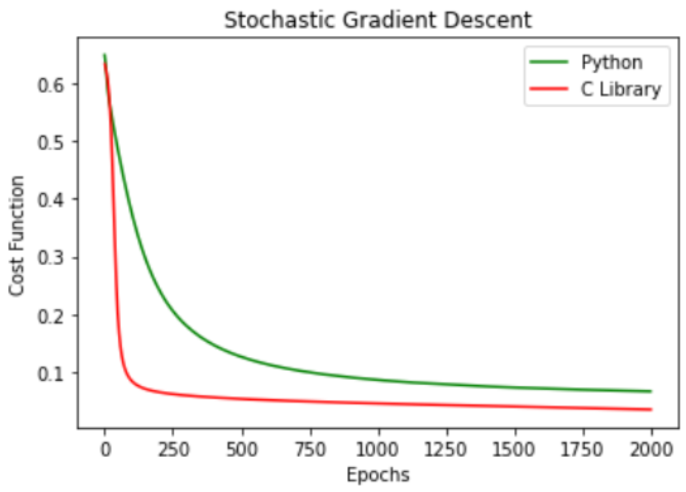


(b) MBGD for Classifier

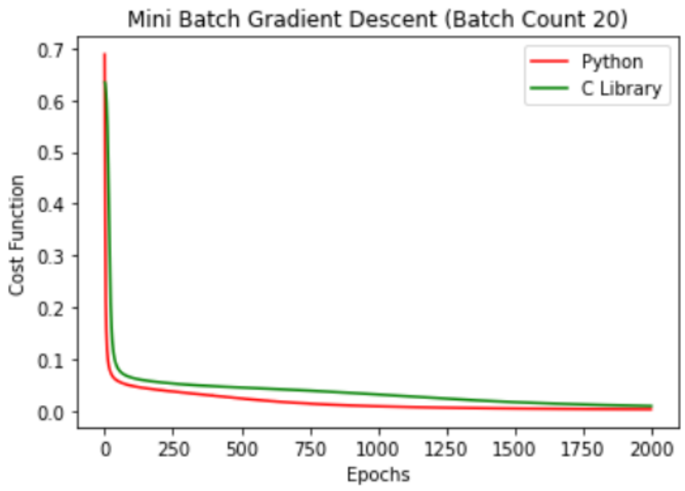


(c) BGD for Classifier

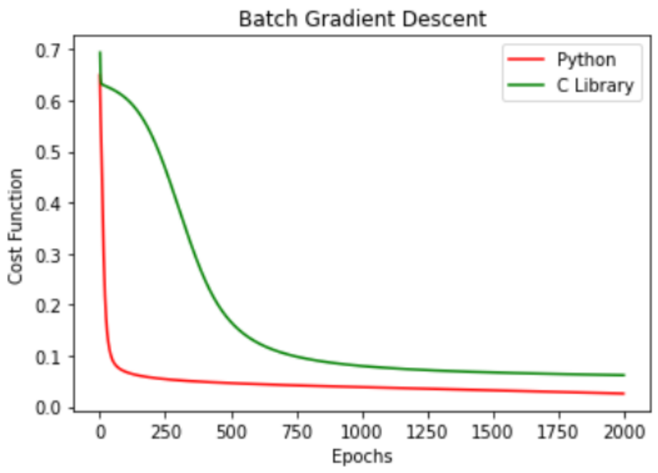
Figure 1: Relu Function



(a) SGD for Classifier

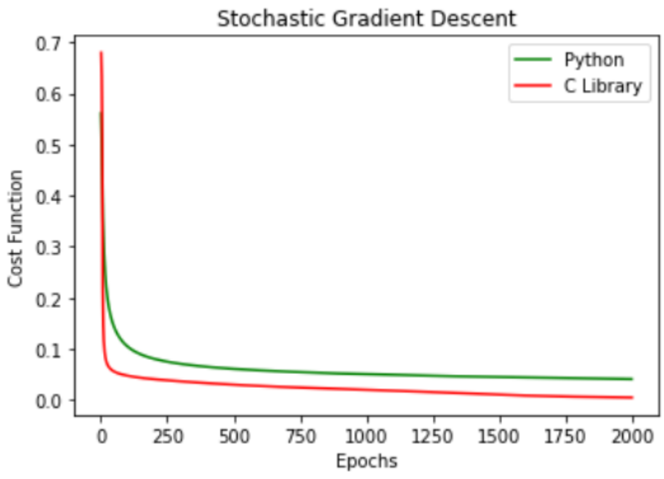


(b) MBGD for Classifier

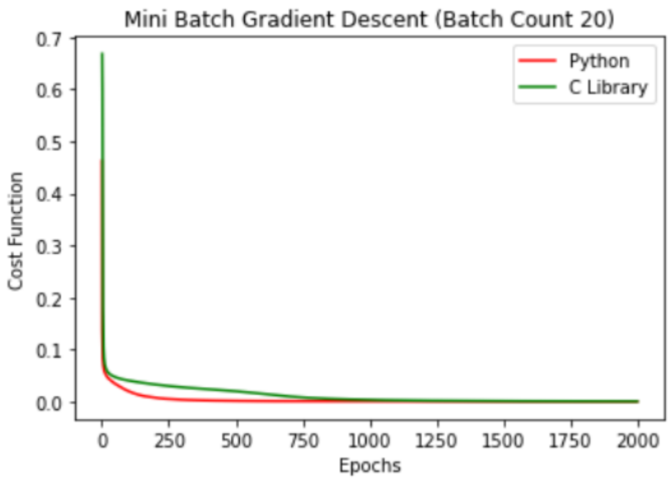


(c) BGD for Classifier

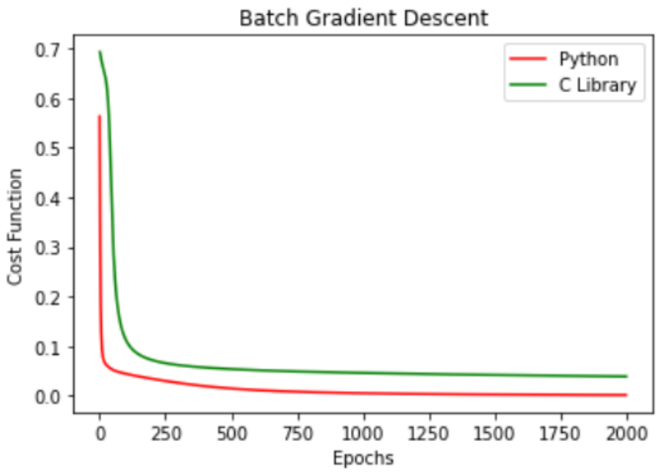
Figure 2: Sigmoid Function



(a) SGD for Classifier

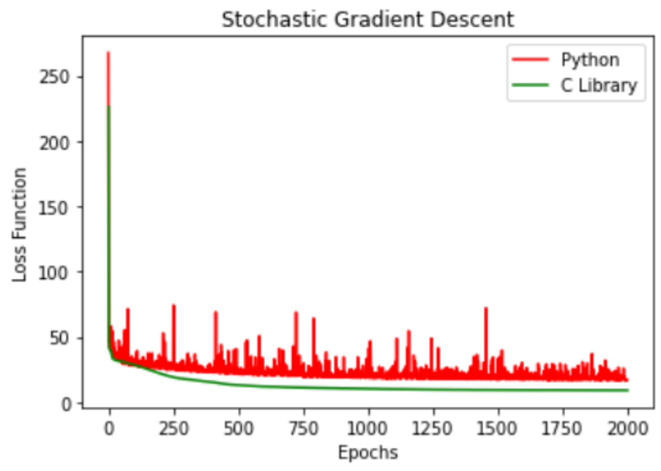


(b) MBGD for Classifier

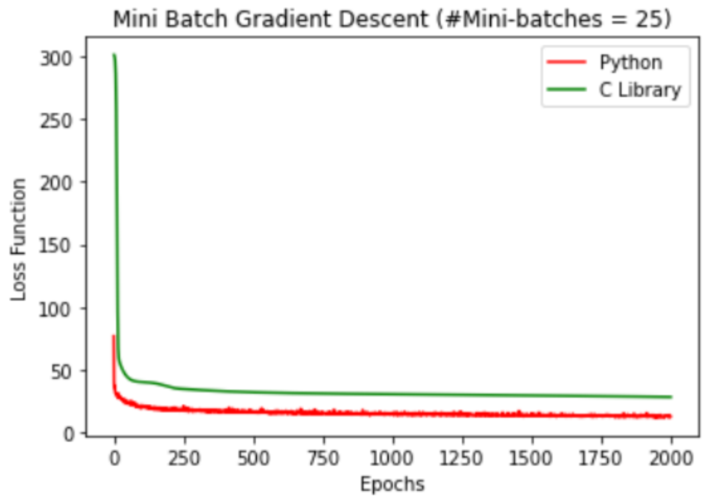


(c) BGD for Classifier

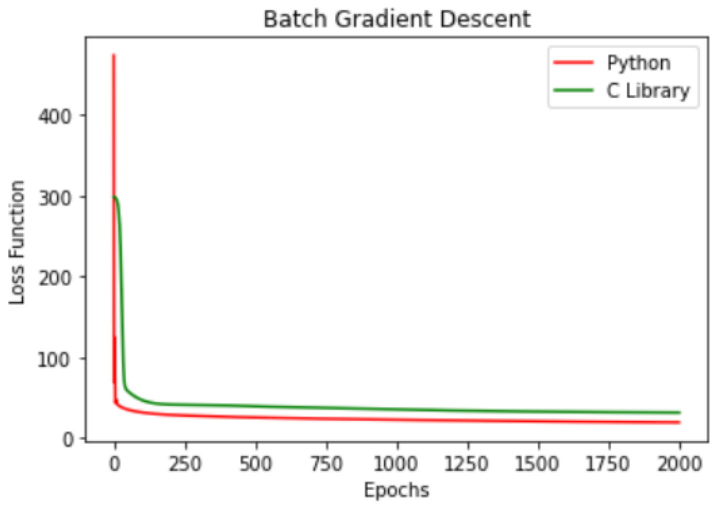
Figure 3: Tanh Function



(a) SGD for Regression

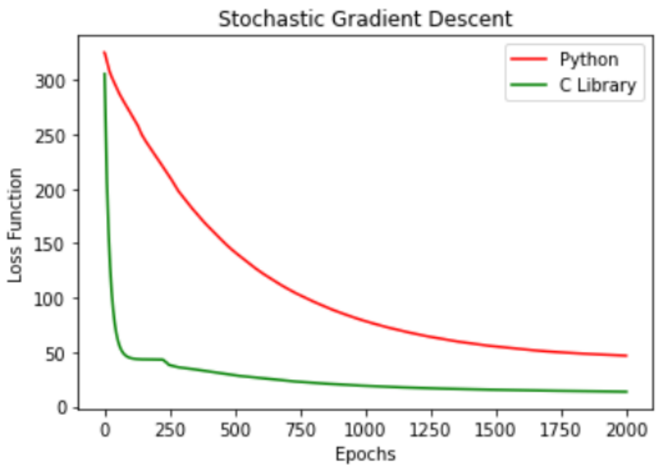


(b) MBGD for Regression

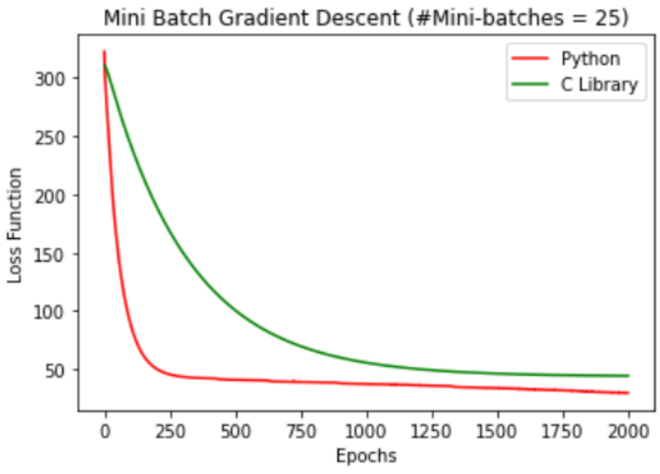


(c) BGD for Regression

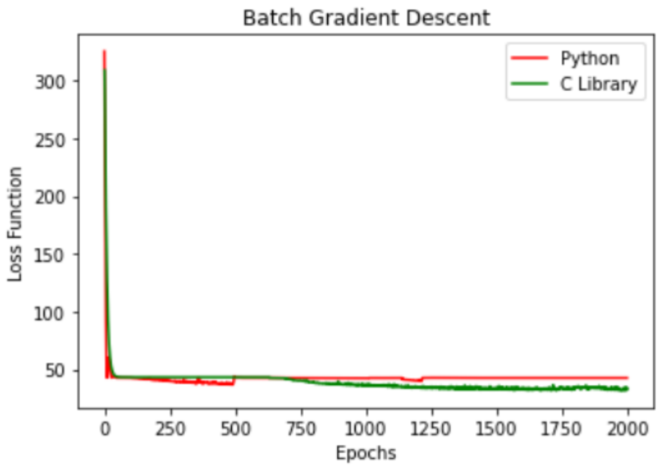
Figure 4: Relu Function



(a) SGD for Regression

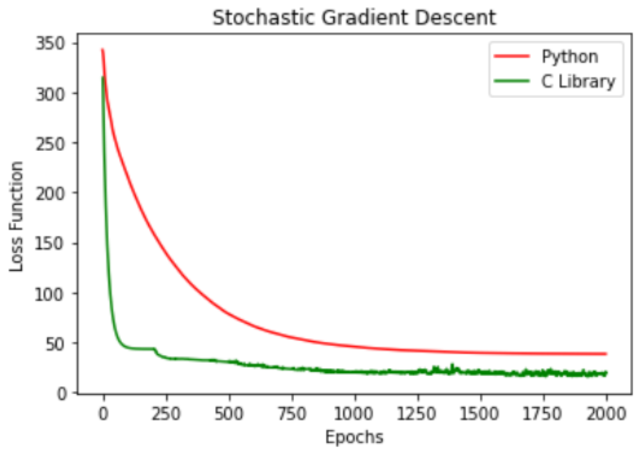


(b) MBGD for Regression

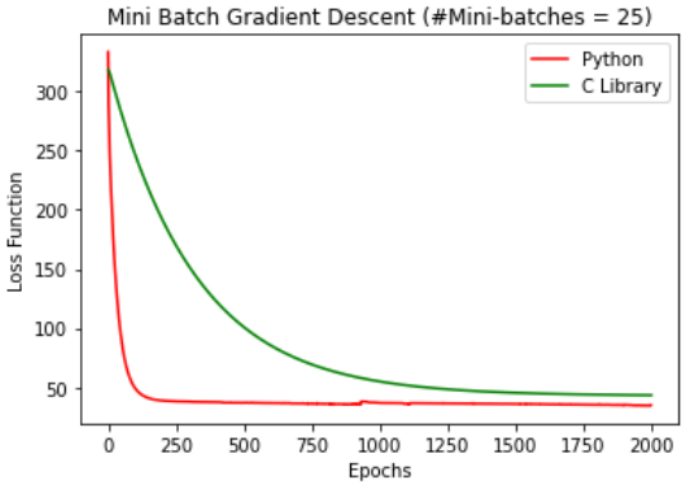


(c) BGD for Regression

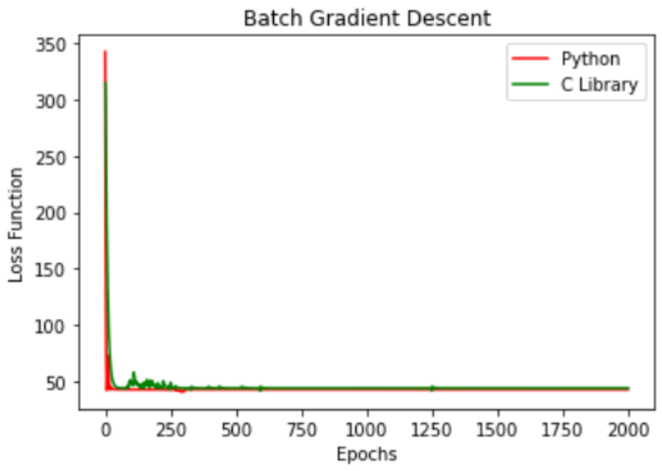
Figure 5: Sigmoid Function



(a) SGD for Regression



(b) MBGD for Regression



(c) BGD for Regression

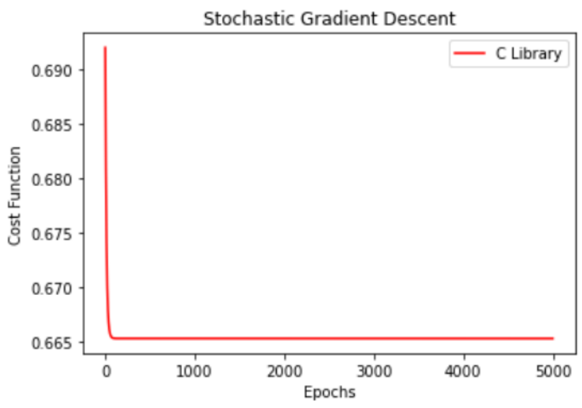
Figure 6: Tanh Function

8.3 Gradient Vanishing Problem

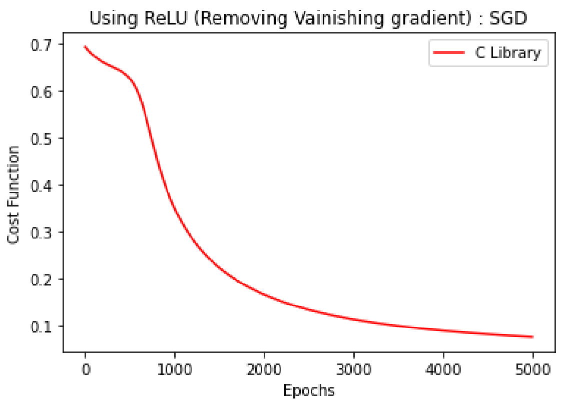
Certain activation functions, like the **sigmoid function**, it takes any large input and convert it to a small input space of [0, 1]. Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence the gradient become very small. Because we have a chain rule. A small value of gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network. One solution to this vanishing gradient descent problem is to use activation function like ReLU which doesn't converge and thus not cause a small gradient.

An example of showing this vanishing gradient descent problem.

Parameters used: activation function : **sigmoid**, learning rate: **1e-4**, Number of neurons in hidden layer = **65**, Number of Hidden layer = **1**, Loss function : **Cross Entropy**



Yes, vanishing gradient problem can be solve. The simplest solution to the vanishing gradient problem is to use some other activation function like **ReLU**, which doesn't cause a small derivative.



9 References

1. http://deeplearning.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

2. https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7

3. https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9

4. https://builtin.com/data-science/gradient-descent

5. https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31

6. https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484