International Journal of Pure and Applied Mathematical Sciences.
ISSN 0972-9828 Volume 18, Number 2 (2025), pp. 103-114
©Research India Publications
http://www.ripublication.com/ijpams.htm

Gradient Descent: *The Backbone of Machine Learning*

Tonny K B, Shikhi M

*Department of Mathematics, College of Engineering,
Trivandrum, Kerala, India*

Abstract

Machine learning has emerged as a powerful tool across various domains, such as computer vision, medical diagnosis, weather forecasting, and stock prediction; yet its success relies on a strong mathematical foundation. This review paper explores the key mathematical concepts that underpin basic machine learning algorithms with a broad audience in mind, including readers who have only a basic mathematical background. By providing a structured overview of the underlying mathematical principles, the paper aims to enhance the theoretical understanding of machine learning researchers and practitioners, facilitating the development of more efficient and reliable algorithms.
keywords: Machine learning, Deep Neural Network, Gradient Descent.
MSC: 68T07.

1. INTRODUCTION

Machine learning has revolutionized the field of artificial intelligence in recent years. The four pillars of machine learning require a solid mathematical foundation. Although mathematical theories were developed in the middle of the 20th century, their application was limited due to the lack of computational speed of machines.

Real-life problems like object identification and speech recognition, which were cumbersome with the traditional programming approaches, were cleared up with high accuracy and efficiency using machine learning. Machine learning is based on data-driven algorithms that enable the machine to learn itself from massive data and to build up some insights about the data, which helps the machine predict valuable information about new data. The present paper provides a comprehensive understanding of the underlying mathematical concepts of machine learning.

Machine learning primarily involves three key approaches viz, supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, the machine is provided with a well labeled data set of the form $(a_i, y_i), i \in \{1, 2, \ldots, m\}$ where y_i is the label of the data a_i, representing a required property. For example, considering the problem of identifying an animal as a cat or non-cat, a_i will be the picture of an animal and y_i will be 1 or 0 specifying whether the picture is that of a cat or non-cat, respectively. Supervised learning algorithms are used to solve classification problems or to predict continuous values of regression problems.

In contrast to supervised learning, the machine is not guided by the labels of the data in the case of unsupervised learning. Such machine learning algorithms are used for clustering problems and for dimensionality reduction of a data set to reduce the number of features of the data set while keeping the salient features. In unsupervised learning, large data are given as input to the machine and the machine categorizes the data according to some similarities or disparities, and the insights thus formed help the machine to make some valuable predictions about the similar properties of new data. Unsupervised learning can be used for customer segmentation, which is useful for personalized product recommendations. Another example of unsupervised learning is creating a generative model, which generates new data similar to a given dataset. This is useful for creating new images, sentences, or even an entire article.

In reinforcement learning, the machine interacts with the environment and learns through trial-and-error methods and feedback. Some examples are optimizing treatment strategies by learning the best actions for a patient over time and teaching autonomous vehicles to make driving decisions based on real-time feedback from the surroundings.

The present paper discusses the mathematical modeling of supervised learning of a machine. We mainly focus on deep learning network architecture.

2. DEEP NEURAL NETWORK

A deep Neural Network (DNN) is a special type of Artificial Neural Network inspired by how the human brain processes information. The human brain consists of approximately 86 billion neurons, each connected with many through synapses [2]. These neurons communicate through electrical impulses. Inspired by this, the structure of the first artificial neuron was proposed by Warren McCulloch and Walter Pitts in 1943. A deep neural network learns from data through two main processes: forward pass and backpropagation. Forward pass involves passing input data through the network to generate predictions. These predictions are then compared to the corresponding labels, and the resulting error is quantified using a loss function. During

backpropagation, the network adjusts its weights and biases to minimize this loss function [5]. The backbone of backpropagation is gradient descent, a technique for optimizing the network's parameters. This process involves computing the gradient of the loss function concerning the weights and biases, which indicates the direction of the steepest ascent. Then, the weights and biases are updated in the opposite direction of this gradient, gradually reducing the loss and improving the network's performance. We'll delve into further details of this process in the upcoming sections.

2.1. Basic structure

A typical deep neural network consists of the following basic structure as depicted in Figure 1.

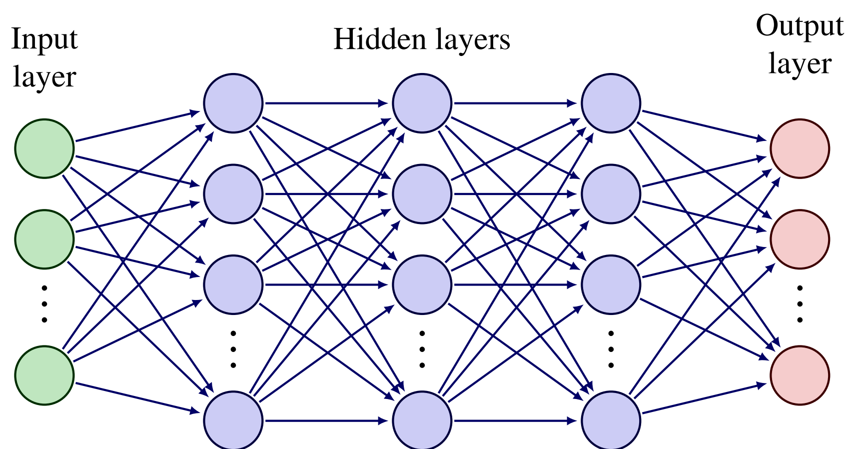

Figure 1: Deep neural network with 3 hidden layers.

The first layer of the network which receives the input data is named as input layer of the network. The neurons (nodes) in this layer receive the features of input data. The number of neurons in the input layer is determined by the shape of the input data.

The final layer of the network is the output layer. The number of neurons in the output layer depends on the type of task we assign to the machine. For example, one neuron is enough for binary classification (Yes/No type), and multiple neurons are required for multi-class classification (eg: digits identification).

The layers in between the input and output layers are called hidden layers. The neurons in the hidden layers extract information from the data for generalization. The neurons in hidden layers and the output layer process the corresponding input through a linear function followed by a non-linear activation.

The particular training problem assigned to the machine determines the number of hidden layers and the number of neurons in each hidden layer in the network. However, there are no general methods to determine these numbers to guarantee the best possible outcome. Therefore, during training, we attempt to optimize such quantities using systematic trial and error methods. These quantities are referred to as hyperparameters. Deep learning encompasses several hyperparameters, some of which, we will explore in the following sections.

2.2. Learnable parameters: Weights and Biases

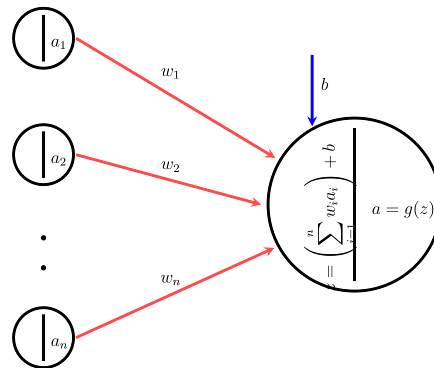

Figure 2: The pictorial representation of a neuron in the lth layer

Weights and biases constitute the parameters that a network learns during its training phase. Neurons in the input layer serve as placeholders for receiving the input data and passing it forward. They don't perform any computation themselves. All other neurons within the network undergo two computational phases during the forward pass. Initially, they compute the weighted sum of their input values, which are the outputs of neurons in the previous layer, and subsequently add a bias term. Secondly, they apply a nonlinear activation function to the result obtained from the first step. A pictorial representation of the process is depicted in Figure 2. These two phases, repeated across the layers of the network, enable it to transform input data into meaningful output predictions.

Mathematically, the first process can be represented as a linear function as follows:

$$z = \sum_{i=1}^{n} (w_i a_i) + b$$

where w_i's are the weights, a_i are the inputs and b is the bias associated to the corresponding neuron. The value of z thus obtained is the input to the non-linear

activation function $g(z)$. The bias term in a neural network enables each neuron's activation function to shift horizontally, independently of the input values. This flexibility allows the neural network to learn more complex relationships between the inputs and outputs.

Some Activation functions	
Name	Function
ReLU	$g(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$
Leacky ReLU	$g(z) = \begin{cases} kz & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$
Sigmoid	$g(z) = \dfrac{1}{1+e^{-z}}$
Tanh	$g(z) = \dfrac{e^{2x}-1}{e^{2x}+1}$

Table 1: Activation functions introduce non-linearity in neural networks

A deep neural network without nonlinear activations in its hidden layers is equivalent to a basic neural network without hidden layers because the composition of linear functions of the form $w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$ results in another linear function.

To demonstrate this concept, consider the neural network as depicted in Figure 3. Consider a neural network with input and output layers containing three and one neurons respectively and with one hidden layer containing two neurons.

Here we assume that there is no nonlinear activation in the hidden layer. Then using the principle of composition of functions,

$$z_3 = (u_{11}w_{11}+u_{12}w_{21})x_1+(u_{11}w_{12}+u_{12}w_{22})x_2+(u_{11}w_{13}+u_{12}w_{23})x_3+u_{11}b_1+u_{12}b_2+b_3$$

This means that the output z_3 is a linear function of the form $k_1x_1 + k_2x_2 + k_3x_3 + b$. Therefore, the neural network described is equivalent to a neural network without hidden layers, with three input neurons and one output neuron.

Introducing nonlinear activation functions in neural networks allows for the expansion of the network size, enhancing its capacity to capture complex relationships within data. Therefore, activation functions play a crucial role in deep neural networks, as they

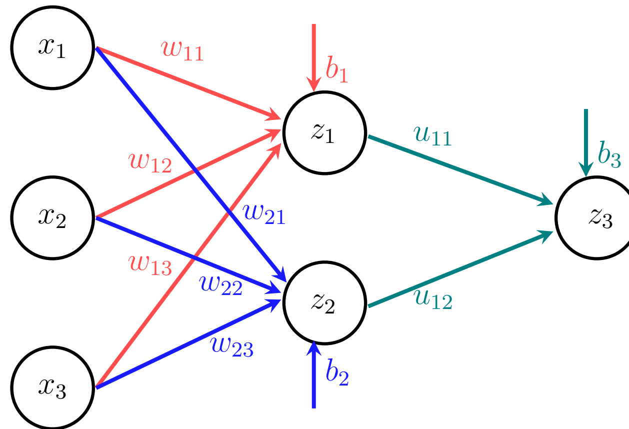

Figure 3: Neural network without activation function

introduce nonlinearity, which is essential for effectively modeling intricate patterns and structures present in the data.

2.3. Loss function and Cost function

The difference between the predicted output and the actual value (label) is the error in the prediction. The function that quantifies this error for a single training example is called the loss function or error function (denoted by E). That is, the loss function provides a quantitative

measure of how well the neural network is performing for a single data point. During the training process network's primary objective is to minimize this loss function. There are various types of loss functions, each suited to different types of machine learning tasks. Some of the loss functions are cross-entropy for classification, Mean Square Error(MSE) for regression, and Mean Absolute Error(MAE).

The cost function (denoted by C) measures the average error over the entire dataset. It aggregates the losses from all training examples into a single scalar value.

2.4. Optimizer

An optimizer is an algorithm used to minimize the loss function of a neural network by adjusting its parameters (weights and biases). Its goal is to find the optimal set of parameters that minimize the difference between the predicted value and the actual value. Some of the popular optimizers include Gradient Descent, Stochastic Gradient

Descent, Mini-Batch Gradient Descent, Gradient Descent with Momentum, Adagrad and Adam(Adaptive Moment Estimation).

In the backpropagation phase, it uses an optimizer to adjust the model's parameters using the gradients of the loss function with respect to the parameters. These gradients represent how the overall error changes with different parameters. The gradients are computed from the output layer back to the input layer, hence the term backpropagation. This phase allows the network to understand how to adjust its parameters to minimize the error. Once the gradients are computed, the present parameters are adjusted using a suitable updating rule. As previously mentioned, the negative gradient indicates the direction of the steepest descent. Therefore, a common updating rule is subtracting a constant (known as the learning rate) times the gradient of the loss function from the respective present values of the parameters.

Machine learning is applied across various branches of science. In the next section, we will examine the mathematical concepts behind machine learning algorithms in a simple and accessible manner to enhance the understanding of undergraduate students and researchers in various fields to apply the concept in their areas of interest.

3. MATHEMATICAL FRAMEWORK OF DEEP NEURAL NETWORK

Consider the identification problem of an animal as a cat or non-cat by analyzing an image. Consider a data set consisting of 28×28 pixel grayscale images each of which is accompanied by a label specifying whether the image is that of a cat (label 1) or non-cat (label 0). Each image has 784 pixels and each pixel represents a gray scale whose values range from 0 (black) to 255 (white) indicating the intensity of the grey shade. Suppose $[a_{ij}]_{28\times 28}$ be the matrix representation of grayscale values of an image. Vectorization of this image produces a 784×1 column matrix $x = [x_k],\ k \in \{1, 2, \ldots, 784\}$ whose first 28 entries are the entries of the first row of $[a_{ij}]_{28\times 28}$ followed by the entries in succeeding rows. (The input layer consists of 784 places to store the entries of x). Allocate 784 memory units to store the entries of x which act as the input layer of the machine learning environment. In this example, the network needs only one neuron in the output layer for the desired output, whether the input image is a cat(1) or non-cat (0). Suppose the network contains three hidden layers. The data from the input layer is passed through subsequent layers to obtain a desired output from the output layer corresponding to present weights and biases. That is the forward pass.

If we use the output activation as sigmoid, then the network prediction is always in between 0 and 1. The loss function computes the error between this predicted value $\hat{y}$ and the corresponding label y of the input. By back-propagating this error, the

optimizer adjusts the weights and biases of the network in such a way that reduces the error (minimizes the loss) most effectively. By iteratively performing forward passes, computing losses, and optimizing the network parameters, the model gradually learns to make better predictions for whether the input image is a cat or not.

3.1. Forward propagation

Consider a fully connected artificial neural network with $L + 1$ layers. We index these layers by $[0], [1], \ldots, [L]$ where $[0]$ and $[L]$ represent the input and output layers respectively. In general, $[l]$ denotes the lth layer for $l = 0, 1, 2, \ldots, L$.

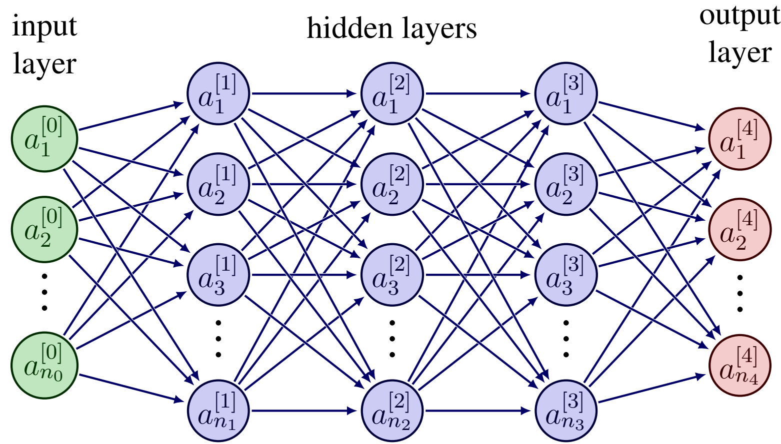

Figure 4: Fully connected Artificial Neural Network (ANN) with three hidden layers.

Let n_l be the number of neurons in the lth layer. Let $W^{[l]} = \left[w_{ij}^{[l]}\right], l \in \{1, 2, \ldots, L\}$ be the $n_l \times n_{l-1}$ matrix, where $w_{ij}^{[l]}$ is the weight associated with j^{th} neuron of $(l-1)^{th}$ layer to i^{th} neuron of lth layer and $B^{[l]} = \left[b_j^{[l]}\right]$ be the $n_l \times 1$ matrix where $b_j^{[l]}$ is the bias associated with j^{th} neuron of lth layer.

Let the $n_0 \times 1$ matrix $a^{[0]} = \left[a_j^{[0]}\right]$ be the input. Since no computations are taking place in the input layer, $a^{[0]}$ itself becomes the output from the input layer. Let the $n_L \times 1$ matrix $y = \left[y_j\right]$ be the label of input vector $a^{[0]}$. For each $l \in \{1, 2, \ldots, L\}$, the weighted sum (linear part) calculated by the jth neuron of lth layer is given by

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \text{ for } j = 1, 2, \ldots, n_l$$

where $a_k^{[l-1]}$ is the output of kth neuron of the $(l-1)^{th}$ layer. Let $z^{[l]} = \left[z_j^{[l]}\right]$ be the

$n_l \times 1$ matrix containing these weighted sums. Let $g_j^{[l]}$ be the activation function used in the jth neuron of lth layer. Let $a_j^{[l]} = g_j^{[l]}(z_j^{[l]})$ for $j = 1, 2, \ldots, n_l$. Then the $n_l \times 1$ matrix $a^{[l]} = \left[a_j^{[l]}\right]$ containing all the activations in the lth layer, becomes the output of the lth layer. Then, $z^{[l]} = W^{[l]}a^{[l-1]} + B^{[l]}$ for $l = 1, 2, \ldots, L$.

3.2. Back propagation

Let $E = E(a^{[L]}, y)$ be the loss function (error function) corresponding to the network-generated output $a^{[L]}$ and the label y. Due to the presence of millions of parameters, the loss function may be highly multi-dimensional and it is impractical to adopt analytical methods to minimize the function. We use iterative algorithms based on gradients to minimize this loss by adjusting the weights and biases (Since only variables associated with this loss function E are weights and biases). These algorithms work on the fact that the steepest descent of E at a point occurs in the direction opposite to that of the gradient of E at the point. Let $\nabla_{z^{[L]}} E = \left[\frac{\partial E}{\partial z_j^{[L]}}\right]$ represent the gradient of E with respect to $z^{[L]}$.

$$\text{Note that, } \frac{\partial E}{\partial z_j^{[L]}} = \frac{\partial E}{\partial a_j^{[L]}} \frac{da_j^{[L]}}{dz_j^{[L]}} \text{ for } j = 1, 2, \ldots, n_L$$
$$= \frac{\partial E}{\partial a_j^{[L]}} g_j^{[L]'}$$

$$\text{Therefore, } \nabla_{z^{[L]}} E = \nabla_{a^{[L]}} E \odot g^{[L]'} \tag{1}$$

where $g^{[L]'} = \left[\frac{dg_j^{[L]}}{dz_j^{[L]}}\right]$ is a $n_L \times 1$ matrix and $\odot$ represents the Hadamard product of corresponding matrices.

Now we will derive $\nabla_{z^{[l-1]}} E$ from $\nabla_{z^{[l]}} E$ for $l = L, L-1, \ldots, 2$.

For $j = 1, 2, \ldots, n_{l-1}$ consider,

$$\begin{aligned}
\frac{\partial E}{\partial z_j^{[l-1]}} &= \frac{\partial E}{\partial a_j^{[l-1]}} \frac{da_j^{[l-1]}}{dz_j^{[l-1]}} \\
&= \left(\sum_{k=1}^{n_l} \frac{\partial E}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}}{\partial a_j^{[l-1]}} \right) g_j^{[l-1]'} \\
&= \left(\sum_{k=1}^{n_l} \frac{\partial E}{\partial z_k^{[l]}} w_{kj}^{[l]} \right) g_j^{[l-1]'} \\
&= \left(\nabla_{z^{[l]}} E\right)^T W_{j_c}^{[l]} g_j^{[l-1]'}
\end{aligned}$$

$$= \left({W^{[l]}}^T_{j_r} \nabla_{z^{[l]}} E \right) g_j^{[l-1]'}$$

where $W^{[l]}_{j_c}$ and ${W^{[l]}}^T_{j_r}$ are the jth column and j^{th} raw of the matrix $W^{[l]}$ and ${W^{[l]}}^T$ respectively.
Therefore

$$\nabla_{z^{[l-1]}} E = \left({W^{[l]}}^T \nabla_{z^{[l]}} E \right) \odot g^{[l-1]'}, \;\; l = L, L-1, \ldots, 2. \tag{2}$$

Where $g^{[l-1]'} = \left[\dfrac{dg_j^{[l-1]}}{dz_j^{[l-1]}} \right]$ is a $n_{l-1} \times 1$ matrix, for $l = L, L-1, \ldots, 2$.
Now we will derive $\nabla_{W^{[l]}} E$ (gradient of E with respect to weights $W^{[l]}$ of lth layer).
For $j = 1, 2, \ldots, n_l$, $k = 1, 2, \cdots, n_{l-1}$ and $l = 1, 2, \ldots, L$,

$$\text{Consider, } \frac{\partial E}{\partial w_{jk}^{[l]}} = \frac{\partial E}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial E}{\partial z_j^{[l]}} a_k^{[l-1]}$$

$$\text{Therefore, } \nabla_{W^{[l]}} E = (\nabla_{z^{[l]}} E) {a^{[l-1]}}^T, \;\; l = 1, 2, \ldots, L. \tag{3}$$

Now we will derive $\nabla_{B^{[l]}} E$. For $j = 1, 2, \ldots, n_l$,

$$\frac{\partial E}{\partial b_j^{[l]}} = \frac{\partial E}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial E}{\partial z_j^{[l]}}$$

$$\text{Therefore, } \nabla_{B^{[l]}} E = \nabla_{z^{[l]}} E, \;\; l = 1, 2, \ldots, L. \tag{4}$$

Using the formulae (1) to (4), we can evaluate gradients of the loss function with respect to the weights and biases of all layers.

4. TRAINING ALGORITHMS

Now we will discuss two fundamental algorithms for training a neural network: Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent. These algorithms operate based on the mathematical principles outlined in Section 2.

4.1. Stochastic Gradient Descent (SGD)

Assume that we have n training examples in our data set D. In stochastic gradient descent (SGD), the weights and biases are updated iteratively for each of the training data $x \in D$. This process helps to make faster updates, especially for large datasets. One epoch in SGD refers to a single complete pass through the entire n training dataset.

The following six steps explain the algorithm of one epoch. For the first epoch, initialize all the weights $W^{[l]}$ and the biases $B^{[l]}$ for $l = 1, 2, \ldots, L$. One common technique is to assign random values using a probability distribution.

Corresponding to each training example $x \in D$ iteratively repeats the following steps.

1. Evaluate $z^{[l]}$ and $a^{[l]}$ for all $l = 1, 2, \ldots, L$; denoted by $z^{[l]}(x)$ and $a^{[l]}(x)$. This process is called forward propagation.
2. Evaluate $\nabla_{z^{[L]}} E(x) = \nabla_{a^{[L]}} E(a^{[L]}(x)) \odot g^{[L]\prime}(z^{[L]}(x))$. [From (1)]
3. Evaluate $\nabla_{z^{[l]}} E(x) = {W^{[l+1]}}^T \nabla_{z^{[l+1]}} E(x) \odot g^{[l]\prime}(z^{[l]}(x))$, for $l = L - 1$, $L - 2, \ldots, 1$. [From (2)]
4. Evaluate the rate of change of the loss function E with respect to weights and biases for the present training example.

$$\nabla_{W^{[l]}} E(x) = \nabla_{z^{[l]}} E(x)(a^{[l-1]}(x))^T$$
$$\nabla_{B^{[l]}} E(x) = \nabla_{z^{[l]}} E(x)$$

   for $l = 1, 2, \ldots, L$. [From (3), (4)]

5. Update the present weights and biases

$$W^{[l]} \leftarrow W^{[l]} - \alpha \nabla_{W^{[l]}} E(x)$$
$$B^{[l]} \leftarrow B^{[l]} - \alpha \nabla_{B^{[l]}} E(x)$$

   where α is the learning rate which is a hyperparameter.

Instead of computing the gradient over the entire dataset at once, Stochastic Gradient Descent (SGD) updates the model parameters using the gradient computed from a single training example at each iteration.

4.2. Mini Batch Gradient Descent

For mini-batch gradient descent, the total training data is randomly divided into subsets of equal size, with each subset referred to as a mini-batch. The size of the mini-batch is a hyperparameter that can be adjusted depending on the dataset and the available computational resources. The cost function(C) is defined as the average loss of individual training examples in a mini-batch. The update is done after passing all the training examples in a mini-batch where the present weights and biases are updated using the average of the gradients of the individual loss functions.

Consider a subset of m elements from a given n training data set. have to define mini batch M

1. Initialize all the weights $W^{[l]}$ and biases $B^{[l]}$ for $l = 1, 2, \ldots, L$.
2. Evaluate $\nabla_{W^{[l]}} E(x)$ and $\nabla_{B^{[l]}} E(x)$ for all m training elements $x \in M$.
3. Update the present weights and biases using the average of the above m gradients of the individual loss functions.

$$W^{[l]} \leftarrow W^{[l]} - \alpha \frac{1}{m} \sum_{x \in M} \nabla_{W^{[l]}} E(x)$$

$$B^{[l]} \leftarrow B^{[l]} - \alpha \frac{1}{m} \sum_{x \in M} \nabla_{B^{[l]}} E(x)$$

If we execute the above three steps for all the mini-batches, then that is one epoch for Mini Batch Gradient Descent.

5. CONCLUSION

This review paper provides a fundamental overview of the mathematical theory behind machine learning algorithms. It is written in a way that allows introductory students to grasp the underlying mathematical concepts. Mathematical symbols are used in such a way to facilitate machine input. Additionally, the paper helps learners to develop a clear understanding of gradient descent algorithms, making it easier to comprehend other related optimization techniques and advanced network architectures.

REFERENCES

[1] Divyasheel Sharma, *Deep Learning without Tears: A Simple Introduction*, Resonance Journal of Science Education, Vol. 25, Issue 1, pp.15-32, 2020.

[2] Herculano-Houzel, S, *The Human Brain in Numbers: A Linearly Scaled-up Primate Brain*. Frontiers in Human Neuroscience, Vol. 331, Article 31, pp.1-11, 2009.

[3] Marc Peter Deisenroth, A. Aldo Faisal & Cheng Soon Ong, *Mathematics for Machine Learning*, Cambridge University Press, 2020.

[4] Michael Nielsen, *Neural Networks and Deep Learning*, Determination Press, 2015.

[5] Rumelhart, D., Hinton, G. & Williams, *Learning representations by back-propagating errors*, Nature 323, pp.533–536, 1986.