Analysis of MLP and Regressor Performance Over Multiple Optimizers

Devanshu K. Singh

Abstract

The optimization methods and algorithm used to optimize the parameters—which, in the context of Multi Perceptrons (MLPs), are the weight parameters and bias parameters of neurons (perceptrons) and, in the case of Regressors, are the detect and slope parameters—has a huge effect on the performance of both Regressors and Multi-Layer Perceptrons. This research article explores the performance of a linear model and a three-layer MLP over the MNIST dataset using four optimization algorithms: Gradient Descent (GD), GD with momentum, RMSProp, and Adam.

1 Introduction

Machine Learning (ML), statistical analysis, and Deep Learning (DL) have entirely taken over and are still advancing in the field of data analytics in the present day. Simple statistical approaches, such as linear regression models, show their usefulness even in increasingly complex data analysis settings by simply expanding the dimensionality of the parameter space using Euclidean algebraic concepts.

1.1 Introduction to Perceptron

When basic models, like linear regression, fail to perform effectively, the first answer is to enhance the model's complexity to integrate as much information as possible for the best potential outcomes. The use of models such as Decision Trees and Random Forest(s) tends to overcome the problem, although the network's performance may still be unreliable. As a result, MLPs are employed to fit the data, which outperforms any other statistical model. The structure of MLPs is a layered structure and the unit of a layer is the model of neuron proposed by Rosenblatt [1], also known as the perceptron model governed by equation 1. Figure 1 illustrates this concept clearly.

$$y = \sum_i W_i \cdot x_i + b \tag{1}$$

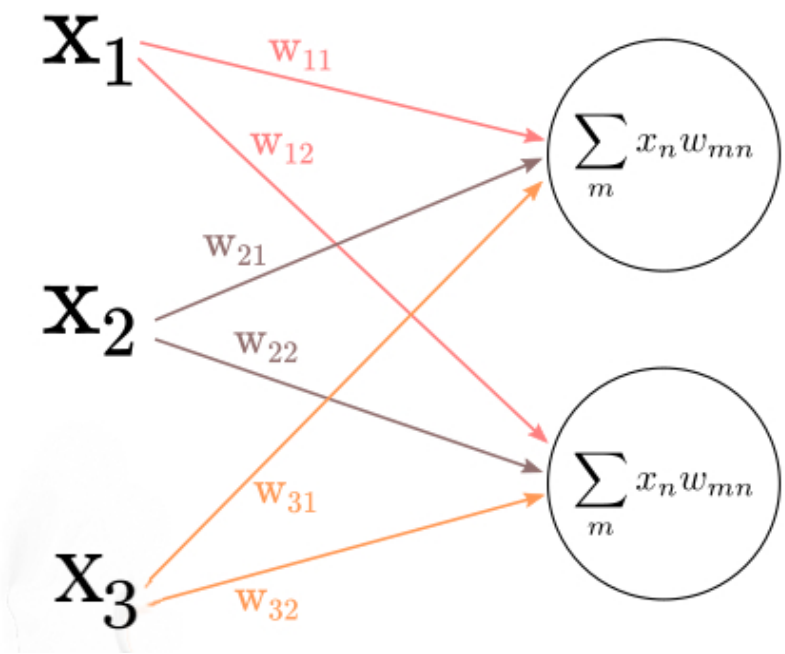

Figure 1: Illustration of a simple perceptron network

Equation 1 is the first mathematical interpretation of the unit of the human brain, a neuron. It can be considered a node, the value of which is determined by 3 variables, the input data which is equivalent to the stimuli that are fed to the brain, and the *weights* attached to the neuron from each stimulus which are likewise the parameters and finally the bias of neurons which is again a parameter. So the value that the neuron holds can just simply be considered a weighted sum of the input data that is being fed to the neuron adding the bias parameter. Different weights signify, the different importance of each input. This is highly by the functioning of the brain because our brain doesn't give equal importance to all stimuli, rather it chooses some important stimuli over others which it learns to do over a training period. The *bias* parameter is used to further fit the input data more accurately.

1.2 Introduction to Activation Functions, and significance of Sigmoid

The weighted sum is then passed through a function that scales down the value of the neuron. This function is known as an *activation function*. Activation functions are another branch of artificial intelligence and encompass the study of these differentiable functions. There are quite a few activation functions in the literature, namely - Rectified Linear Unit (ReLU), Hyperbolic Tangent (tanh), Sigmoid, etc. Further improvement of the concept of MLPs led to the development of Artificial Neural Networks (ANN)s, which can also be considered modified MLPs. An ANN differs from an MLP slightly, and the point of difference is created by the choice of

activation function used in both networks. The activation function used in an MLP is known as the Threshold Logic Unit (TLU) function governed by Equation 2, Figure 2 shows the graph of TLU activation, while the activation function in an ANN is a sigmoid function controlled by Equation 3 and is illustrated in Figure 3.

$$a(z) = \begin{cases} 1, & z \geq 0 \\ 0 & z < 0 \end{cases} \tag{2}$$

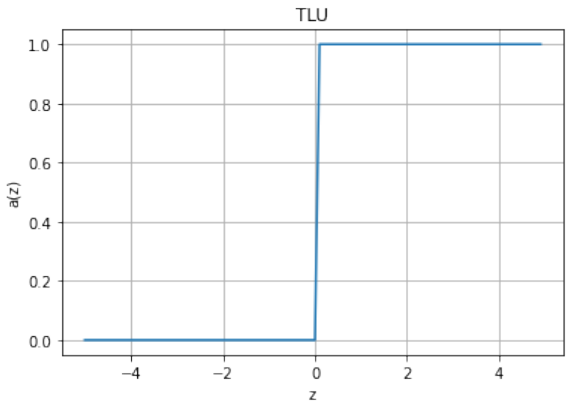

Figure 2: TLU activation curve

$$a(z) = \frac{1}{1+e^{-z}} \tag{3}$$

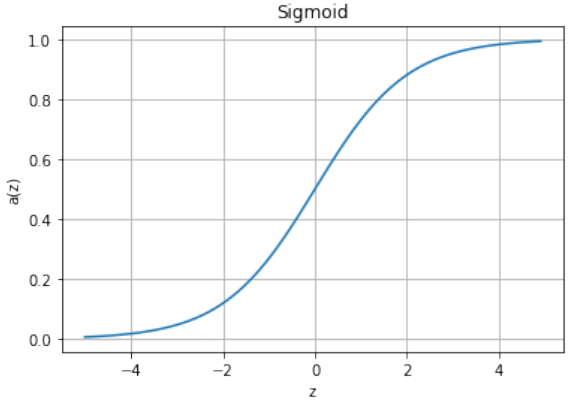

Figure 3: Sigmoid activation curve

The TLU function is used because it's feasible for some neurons to fire at once while other neurons might not. This is a logical approach that is consistent with the biological phenomena

of the thinking process. However, the usage of sigmoid offers a more accurate representation of the human brain, which is what gave rise to the moniker ANN, meaning that all neurons fire, while some do so more frequently than others. The sigmoid function, which has a range of $[0,1]$, makes this conceivable. As a result, rather than just being in a "off" state like in an MLP, all neurons are partially activated.

The essence of MLPs and ANNs lies in their training process. The training process encompasses optimizing the network parameters (weights and biases) that are initialized randomly, using the input data to have minimum prediction error. The input data is divided into 2 major portions - training data and the testing data, and their size ratio is generally kept to be either $7:3$ or $8:2$. Sometimes the data distribution also contains a validation set, which is used to analyze the quality of the training process at each iteration. The application of optimization strategies from the literature optimizes the network parameters. These optimization strategies, which are frequently utilized over a custom network, are compared in this article.

2 Literature Review

2.1 Forward Propagation

With the development of the perceptron, a method for calculating the output of a perceptron was also developed [1]. The first step is the calculation of the weighted sum with the bias parameter added subsequently, which is referred to as *forward propagation.* This is also referred to as the network's *linear function.* This was already addressed in the previous section, and when the output of a linear functional calculation is created, it is further processed by an activation function that scaled down the result to reduce its computational complexity. The demand for the summing is removed by the weight matrix-input and vector-matrix product. The network's last or output layer is indicated by the superscript L, followed by the hidden layer $(L-1)$, and the input layer $(L-2)$ of the network.

$$z^{L-1} = (W^{L-1})^T X + b \tag{4}$$

$$a^{L-1} = a(z^{L-1}) \tag{5}$$

Once the hidden layer's activation has been determined, it is supplied as input to the next layer, which is the output layer for the network being explored and constructed in this article.

$$z^{L} = (W^{L})^T a^{L-1} + b \tag{6}$$

$$a^{L} = a(z^{L}) \tag{7}$$

2.2 Loss Functions

There are majorly 2 different types of cost or loss functions used in neural networks, the difference arises depending on whether the network performs a regression or classification. If the network performs a regression task, Equation 8 is used as a loss function, and if the network performs classification either binary classification or multi-class classification, the loss function is governed by Equation 9

$$\mathscr{L} = \frac{1}{2}\sum_{j}(a_j - y_j)^2 \tag{8}$$

$$\mathscr{L} = -\sum_{j} y_j \ln a_j \tag{9}$$

Equation 9 is known *Cross-Entropy Loss*, and a loss function which is also widely used in place of cross-entropy loss, is the *Log Loss* function, and it is governed by Equation 10

$$\mathscr{L} = -\sum_{j}(y_j \ln a_j + (1 - y_j) \ln 1 - a_j) \tag{10}$$

2.3 Backpropagation Algorithm

Williams, Rumelhart, and Hinton [2] proposed the backpropagation algorithm which revolutionized the way parameter optimization is carried out in a neural network. The backpropagation algorithm is simply a chain rule of partial derivatives applied to the loss function involved in the network to obtain network parameter gradients. The gradient of weights in the network's final layer according to backpropagation takes the form shown in Equation 11

$$\frac{\partial \mathscr{L}}{\partial w_{kj}^L} = \frac{\partial \mathscr{L}}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial w_{kj}^L} \tag{11}$$

Equation 11 assumes that there are j output neurons and k neurons in the last hidden layer. The first partial derivative $\frac{\partial L}{\partial a_j^L}$, is the partial derivative of the cost/loss function w.r.t. the activation function produced in the previous layer, then with respect to the linear function produced in the previous layer, and finally w.r.t. the weights that interconnect the neurons in the previous layer and the last hidden layer. The gradient of the bias parameters, like the weight parameters, is dictated by Equation 12

$$\frac{\partial \mathscr{L}}{\partial b_{kj}^L} = \frac{\partial \mathscr{L}}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial b_{kj}^L} \tag{12}$$

Just as Equation 11 and Equation 12 behave, for gradients further back in the network, the number of partial derivative terms keeps on increasing. For weight parameters of the hidden layer in the network, the gradient is governed by Equation 13

$$\frac{\partial \mathscr{L}}{\partial w_{lk}^{L-1}} = \frac{\partial \mathscr{L}}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial a_k^{L-1}} \times \frac{\partial a_k^{L-1}}{\partial z_k^{L-1}} \times \frac{\partial z_k^{L-1}}{\partial w_{lk}^{L-1}} \tag{13}$$

But the Equation 13 is not complete. Since the activation of layer $L-1$ affects the error through multiple pathways, As a result, a summation must be included in the equation to account for every component of the activation impacting the inaccuracy. This modification is formulated in Equation 14. However, this is not required for the bias parameters as they affect the error singularly. So, the equation for the gradient of the loss with respect to the bias vector of the hidden layer of the network is similar to Equation 13, governed by equation 15

$$\frac{\partial \mathscr{L}}{\partial w_{ik}^{L-1}} = \left(\sum_j \frac{\partial \mathscr{L}}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial a_k^{L-1}} \right) \times \frac{\partial a_k^{L-1}}{\partial z_k^{L-1}} \times \frac{\partial z_k^{L-1}}{\partial w_{ik}^{L-1}} \tag{14}$$

$$\frac{\partial \mathscr{L}}{\partial b_k^{L-1}} = \frac{\partial \mathscr{L}}{\partial a_j^L} \times \frac{\partial a_j^L}{\partial z_j^L} \times \frac{\partial z_j^L}{\partial a_k^{L-1}} \times \frac{\partial a_k^{L-1}}{\partial z_k^{L-1}} \times \frac{\partial z_k^{L-1}}{\partial b_k^{L-1}} \tag{15}$$

The gradients of weight and bias parameters are calculated using the chain rule then utilized to optimize the parameters, which are some of which are described in the preceding section. GD is a well-known optimization technique that updates the parameter, and the update process is governed by Equation 16. Equation 16 is used to optimize the weight parameter matrix and represents the general form of the update rule of GD for weight matrices of the network. Similarly, Equation 17 is used to optimize the bias parameters of the

$$w_{t+1} = w_t - \eta \cdot \nabla w_t \tag{16}$$

$$b_{t+1} = b_t - \eta \cdot \nabla b_t \tag{17}$$

The optimization algorithms further introduce *hyperparameters* in the network, one such parameter is η which is known as the learning rate. It is the step size that the algorithm takes towards gradient vector to reduce the error. Most optimization algorithms used to optimize network parameters are gradient-based due to the nature of backpropagation. The idea behind gradient-based optimization is descent over error space to the lowest point or the global minima. However, with gradient-based optimization, often the algorithm gets stuck in the local minima in case of a sufficiently complex problem and may not always be able to escape the curve of local minima.

3 Methodology

3.1 Network Architecture

The MLP constructed in this article consists of 3 layers, namely - the input layer, the hidden layer, and the output layer. Figure 5 illustrates the network architecture. The number of input neurons is 784 because each image in the dataset being used to train the MLP which is the MNIST [3] dataset of digits $0-9$ is 28×28 pixels which result in 784 when flattened. Figure 4 shows some digits from the MNIST dataset. Further, the MLP structure consists of 128 hidden neurons. Therefore, the shape of the first weight matrix is (128×784) and the shape of the bias vector of the hidden layer is (128×1). The linear computation for the hidden layer becomes. Also, the shape of the second weight matrix is (128×10), and the shape of the bias vector is (10×1).

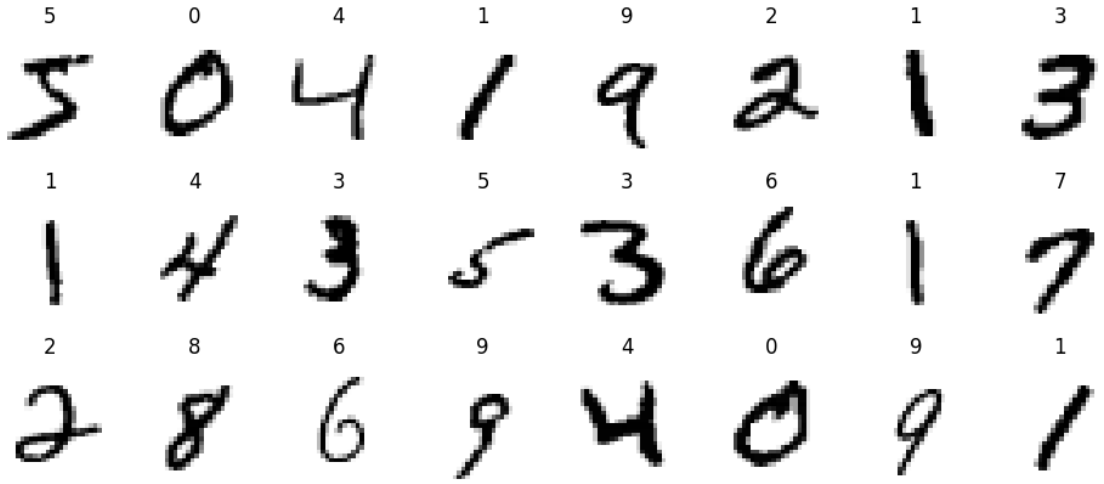

Figure 4: Digits from MNIST dataset

$$z^1 = (W^1)^T \cdot x + b^1 \tag{18}$$

The resulting vector $z1$ has the shape (128×1). The ReLU activation, which is controlled by the Equation 19 is the activation function used to scale the hidden layer's output. This just scales the input vector as any other activation function does; the previous section explains how this affects the form of the output vector z^1. Equation 20 applies the hidden layer's activation to the output of the hidden layer's linear function. After determining the output of the activation function, the following layer of the output layer gets it as input.

$$ReLU(z) = max(0, z) \tag{19}$$

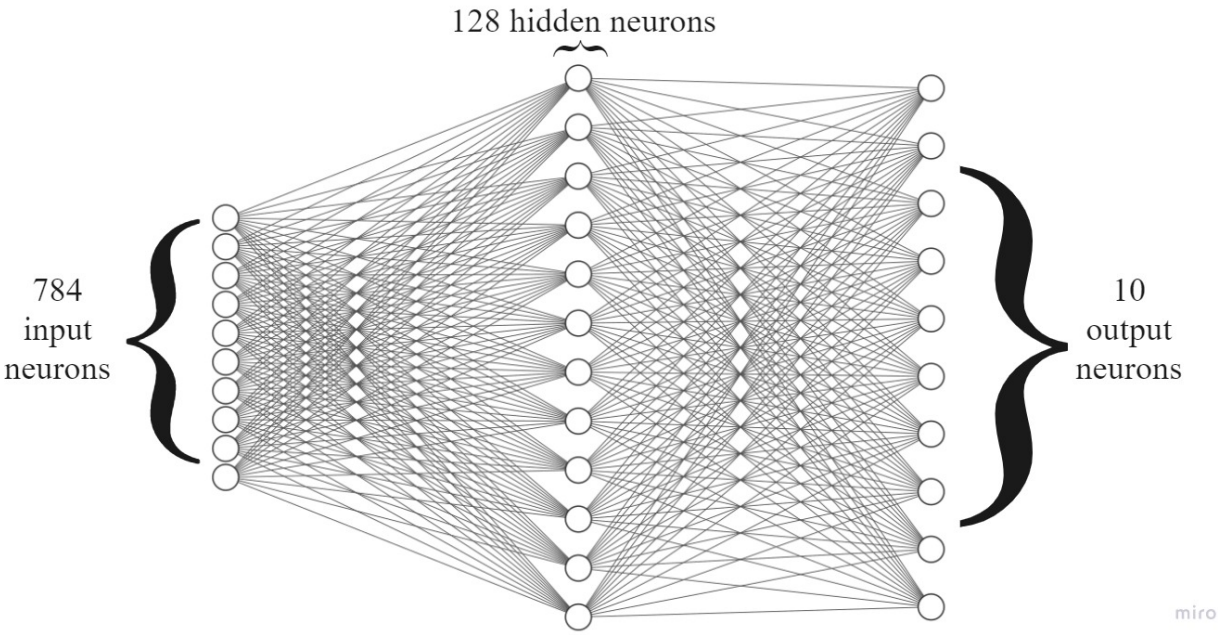

Figure 5: Network architecture

$$a^1 = ReLU(z^1) \tag{20}$$

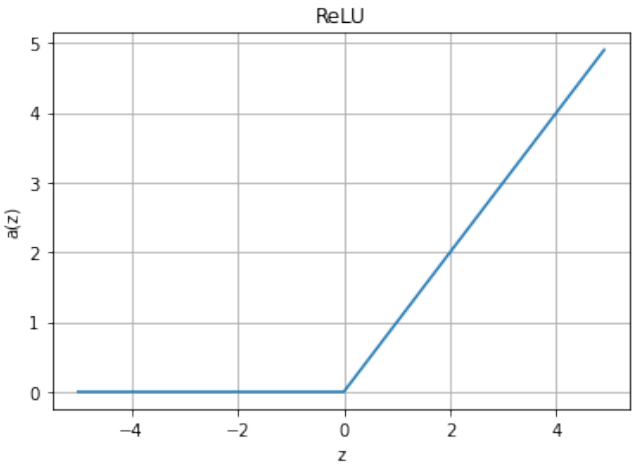

Figure 6: ReLU activation curve

Equation 22 gives an output of the hidden layer's activation, which is needed by the final layer's linear function. After matrix multiplication of the weight matrix and the activation

vector of the previous layer, which has the shape (128×1), another vector with the shape (10×1) is produced, giving the weight matrix of the last layer its shape of (128×10). In the last layer, the Softmax or Sigmoid functions might be utilized as activation functions. In multi-class classification problems, the Softmax function is widely utilized. However, this neither demonstrates nor disproves the employment of the Sigmoid function. The activation function of the output layer is the focus of this paper.

$$z^2 = (W^2)^T \cdot a^1 + b^2 \tag{21}$$

$$a^2 = Sigmoid(z^2) = \frac{1}{1+e^{-z^2}} \tag{22}$$

Now that the forward pass is complete, next comes optimizing parameters using the back-propagation and optimization algorithms. The log loss function is employed in the paper, and the partial derivatives for the gradients of loss with regard to the weight parameters are-

$$\frac{\partial \mathscr{L}}{\partial a^L} = A^L - Y \tag{23}$$

$$\frac{\partial a^L}{\partial z^L} = A^L(1 - A^L) \tag{24}$$

$$\frac{\partial z^L}{\partial w^L} = A^{L-1} \tag{25}$$

Here, Y is a one-hot vector. From Equations 23, 24 and 25 we get. The formulation of bias gradient is similar to that of Equation 26

$$\frac{\partial \mathscr{L}}{\partial w^L} = (A^L - Y) \times A^L(1 - A^L) \times A^{L-1} \tag{26}$$

$$\frac{\partial \mathscr{L}}{\partial b^L} = (A^L - Y) \times A^L(1 - A^L) \tag{27}$$

3.2 Optimization Methods

The optimization algorithms used in the article to optimize the network parameters are GD, modification of GD - GD with momentum, RMSProp, and Adam optimizer. The intuition behind GD with momentum is that of a rolling ball with dynamic mass, mass increases at each iteration by a fraction of the previous mass. Here the mass of the ball is the error at each iteration. This increases the rate of convergence more than that of GD. Another modification of GD with momentum is *Nesterov's Accelerated Gradient*, with an even greater rate of convergence than GD with momentum. The update rule governing GD with momentum is as follows-

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot \nabla w_t \tag{28}$$

$$w_{t+1} = w_t - \eta \cdot v_t \tag{29}$$

Here v_t is the velocity term and denotes the velocity of the gradient. The β parameter is the momentum after which the optimizer is known.

The RMSProp optimizer was developed as an improvement to the RProp optimizer. The reason is that the RProp optimizer does not work well with mini-batches. The RMSprop optimizer is quite similar to the GD with a momentum optimizer. The RMSprop optimizer constraints oscillations in the vertical axis of the parameter space. Therefore, the learning rate can be increased and the algorithm can take larger steps in the horizontal direction, leading to convergence faster. The difference between RMSprop and GD is in how the gradients are calculated. The following equations, Equation 30 shows how the gradients are calculated for the RMSprop and GD with momentum. The value of momentum is denoted by beta and is usually set to 0.9. If you are not interested in the math behind the optimizer, you can just skip the following equations.

$$v_t = \beta \cdot v_{t-1} + (1-\beta) \cdot (\nabla w_t)^2 \tag{30}$$

$$w_{t+1} = w_t - \eta \cdot \frac{\nabla w}{\sqrt{v_t} + \varepsilon} \tag{31}$$

In the Adam optimizer, instead of adapting learning rates based on the average first moment as in RMSProps, Adam makes use of the mean of second moments of gradients. This algorithm determines the Exponential Moving Average (EMA) of gradients along with that of square gradients. The parameters of β_1 and β_2 are used to control the decay rates of these moving averages. Adam is a combination of two GD methods, Momentum, and RMSProp which are explained below.

$$v_t = \beta_1 \cdot v_{t-1} + (1-\beta_1) \cdot \nabla w_t \tag{32}$$

$$s_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla w_t)^2 \tag{33}$$

$$w_{t+1} = w_t - \eta \cdot \frac{v_t}{\sqrt{s_t} + \varepsilon} \cdot \nabla w \tag{34}$$

4 Results

This section ruminates on the results obtained from the training process of the MLP, from both frameworks - Tensorflow and the self-implemented scratch framework. The results for GD optimization and GD with momentum optimization are obtained from the self-implemented

framework, shown in Figure 7. The performance of other optimizers which are RMSProp and Adam is obtained from the Tensorflow framework, this is shown in 8.

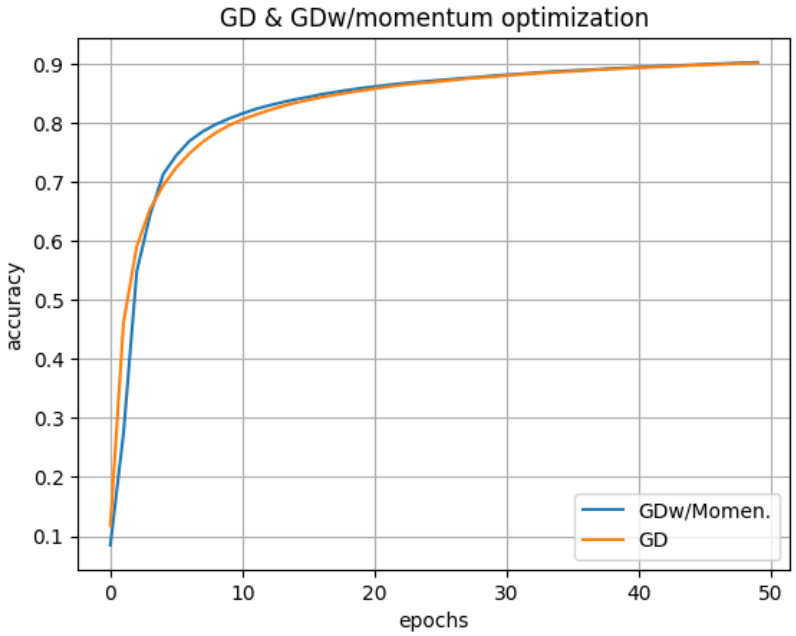

Figure 7: Accuracy curves of GD & GD with momentum

Improvement in accuracy is visible in the self-implemented framework, with the training time till the convergence of the error of prediction of the network still being more than the Tensorflow training time. This proves that the concept of implementation of the network from scratch is correct but still inefficient than Tensorflow. This is due to a particular reason which is discussed in the next section.

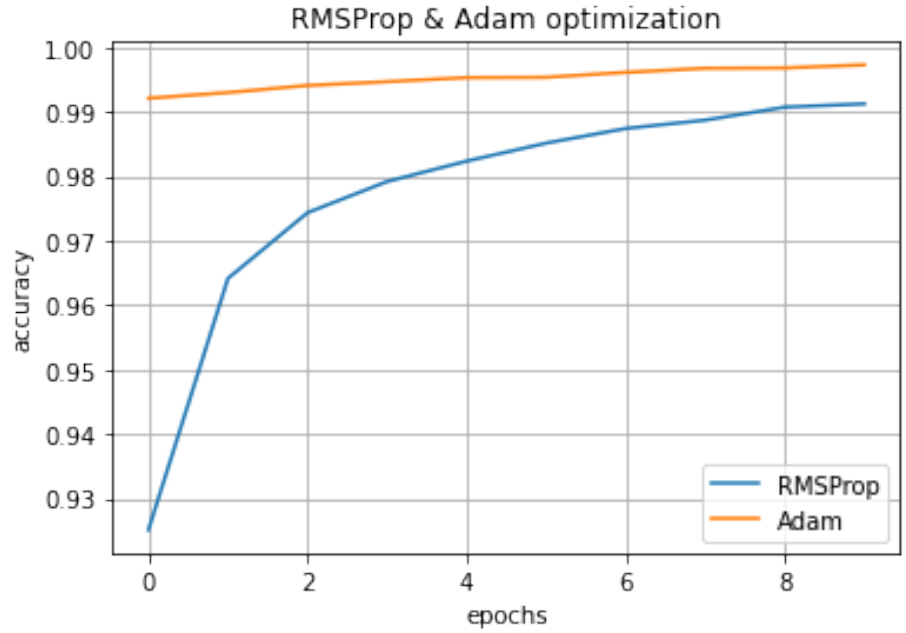

Figure 8: Accuracy curves of RMSProp & Adam

5 Conclusion

The effect of the application of various optimizers was studied in this article. It was found that not only the optimization algorithms affect the training process of the neural network but also the random values of weights and biases that perform the feed-forward pass of the network. In-depth working of an MLP or neural network was understood. This provides a deep understanding of neural networks and lays the foundation for deep neural networks. The effect of hyperparameters such as learning rate and momentum hyperparameter, which are not specifically a part of neural networks but plays an important role in optimizing parameters and reaching convergence of the loss function, was also studied.

6 Future Scope

The performance of the self-implemented neural framework lacked in the process of weight initialization. In the challenges that were recognized by Geoffrey Hinton by the end of AI winter, one of the challenges was correct weight initialization. Later, this challenge was overcome by introducing statistical techniques of weight initialization. Tensorflow framework has these techniques implemented in the framework for the best possible training performance, as it is a quite widely used framework. The fully connected network that was implemented in this article used weights and biases that are randomized in the range $[-1, 1]$, whereas the same fully connected

network has the weights initialized according to the *Glorot Uniform* distribution, and the biases are initialized zero vectors. Apart from glorot uniform distribution, *He Normal*, and *LeCun Normal* distributions are also used. The self-implemented framework can further be improved by the incorporation of these weight initialization techniques to improve the training process.

References

[1] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

[2] Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986). https://doi.org/10.1038/323533a0

[3] Deng, L., 2012. The mnist database of handwritten digit images for machine learning research.*IEEE Signal Processing Magazine*, 29(6), pp.141–142.