Neural Network and Optimizers

Sahil Chaudhary[*] **Om Prakash Chaudhary**[*] **Pratham Gupta**[*] **Rolla Siddartha**[*]

Aditya Manjunatha[*] **Adithya K Anil**[*]

Abstract

Neural networks have revolutionized machine learning by surpassing traditional models such as Support Vector Machines (SVM). However, the performance of neural networks is heavily dependent on the optimization methods used to update their weights and biases. In this paper, we provide a comprehensive comparison of four prominent optimization algorithms: Stochastic Gradient Descent (SGD), Momentum, RMSProp, and Adam. We detail the mathematical foundations and algorithms for each method and offer practical guidelines for implementing neural networks from scratch. Using the MNIST and CIFAR-10 datasets, we empirically evaluate the performance of these optimizers. Our results demonstrate that Adam consistently outperforms other stochastic optimization methods, providing faster convergence and higher accuracy. This paper aims to serve as a valuable resource for beginners and practitioners in the field of neural network optimization.

Introduction

A neural network is a machine learning model that mimics the functioning of a biological brain to make decisions or predict outputs after being trained with initial data. A neural network consists of several layers of neurons, each with its own properties (weights and biases). These layers process inputs to produce a conclusive output or prediction. Initially, the network is fine-tuned or trained using training data to enable it to function efficiently and produce accurate predictions.

Structure of a Neural Network

A neural network is composed of layers of neurons, where each neuron can be thought of as a linear regression model. The basic operation within a layer is represented by the formula:

$$F(X) = W \circ X + B \tag{1}$$

where :

X is the input vector, W represents the weights, B represents the biases and $\circ$ represents the dot product.

The weights determine the contribution of each input feature towards the output. All inputs are multiplied by their respective weights, summed, and then passed through an activation function, which determines the output. This output becomes the input for the next layer. Finally, the output is extracted from the last layer of the network.

Training Process

The training process involves adjusting the weights and biases to minimize the error between the predicted output and the actual target values. This process is iterative and can be broken down into the following steps:

[*]All authors contributed equally to this work.

1. Forward Propagation

In forward propagation, the input data is passed through the network layer by layer to obtain the output. For each neuron in a layer, the operation is:

$$a^{(l)} = \sigma(W^{(l)} \cdot a^{(l-1)} + B^{(l)}) \tag{2}$$

where:

σ is the activation function acted upon the computation as in the initial formula.

2. Loss Calculation

The loss function quantifies the difference between the predicted output and the actual target values. Common loss functions include Mean Squared Error (MSE) for regression tasks. For instance, the MSE is given by:

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3}$$

where y represents the predicted value and $\hat{y}$ represents the actual value in the data. Also n represents the number of test samples taken in the training data.

3. Backpropagation

Backpropagation is used to calculate the gradient of the loss function with respect to each weight by applying the chain rule of calculus. It involves two steps: calculating the gradient of the loss with respect to the output (backward pass) and then propagating these gradients backward through the network layers.

The gradient of the loss function L with respect to the weights $W^{(l)}$ and biases $B^{(l)}$ at layer l can be computed as follows:

$$\frac{\partial L}{\partial W^{(l)}} = \delta^{(l)} \cdot (a^{(l-1)})^T \tag{4}$$

$$\frac{\partial L}{\partial B^{(l)}} = \delta^{(l)} \tag{5}$$

where $\delta^{(l)}$ is the error term for layer l, calculated as:

$$\delta^{(l)} = (\delta^{(l+1)} \cdot (W^{(l+1)})^T) \circ \sigma'(z^{(l)}) \tag{6}$$

Here, $\circ$ denotes dot product, and σ' is the derivative of the activation function.

4. Parameter Update

Once the gradients are calculated, the weights and biases are updated using an optimization algorithm. A common method is Stochastic Gradient Descent (SGD), where the parameters are updated as follows:

$$W^{(l)} = W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}} \tag{7}$$

$$B^{(l)} = B^{(l)} - \eta \frac{\partial L}{\partial B^{(l)}} \tag{8}$$

where η is the learning rate, a hyperparameter that controls the step size during the update. These parameters are carefully designed for specific tasks based on the dataset and the values. Having a very high η, the model can miss the saddle points in the stochastic function whereas having ti very low makes it very long to compute.

After the neural network has been trained on some data, all it's parameters are updated and further predictions can now be made based on the updated network. There are many variations in this model such as the shape and sizes of the network and also the many other types of functions being used for activation or loss calculation. Many optimization techniques are also used to make this easily computable for large datasets such as SGD, RMSProp and Adam which are further discussed below.

Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a fundamental optimization technique in machine learning. It is an iterative procedure used to optimize an objective function, typically a loss function. SGD is a variant of the standard gradient descent algorithm which uses stochastic approximations by randomly sampling a subset of the data to perform updates. This approach reduces the computational complexity and speeds up the iterations, albeit at the cost of exact convergence.

In neural networks, the objective is to find the optimal set of weights that minimize the loss function for the given data. The standard gradient descent algorithm updates the weights by moving in the direction of the negative gradient of the loss function:

$$\theta := \theta - \eta \nabla F(\theta) \tag{9}$$

where θ represents the parameters (weights) of the model, η is the learning rate, and $\nabla F(\theta)$ is the gradient of the loss function with respect to θ. While effective, this method can be computationally expensive and time-consuming for large datasets.

SGD addresses this issue by approximating the gradient using a single data point (or a mini-batch) at each iteration:

$$\theta := \theta - \eta \nabla F_i(\theta) \tag{10}$$

where $F_i(\theta)$ denotes the loss function evaluated at the i-th data point. By iterating over random subsets of the data, SGD reduces the overall complexity and provides a more efficient optimization process. The dataset is typically shuffled between passes to prevent cycles that could hinder convergence.

To further enhance convergence, adaptive learning rates are often employed, where the learning rate η is adjusted dynamically based on various factors. This approach is elaborated upon in advanced optimization techniques such as Momentum, AdaGrad, and Adam.

Example: SGD on a Quadratic Loss Function

Consider a simple quadratic loss function:

$$F(\theta) = \frac{1}{2}(\theta - \theta^*)^2 \tag{11}$$

where θ^* is the optimal parameter value. The gradient of this loss function is:

$$\nabla F(\theta) = \theta - \theta^* \tag{12}$$

Using SGD, the parameter update rule becomes:

$$\theta_{t+1} = \theta_t - \eta(\theta_t - \theta^*) \tag{13}$$

Here, θ_t is the parameter value at iteration t. By iteratively applying this update rule, θ_t will converge to θ^*, given an appropriate learning rate η.

Momentum

Momentum optimizer is an optimization algorithm which accelerates the process of gradient descent by considering the previous gradients to smooth out the updates. By considering the accumulated gradients, momentum can help the optimizer to accelerate in directions with persistent gradients, leading to faster convergence. The momentum optimizer introduces a velocity term v which denotes the accumulated gradient. The consideration of velocity term also help in smoothing out noises as well as escaping shallow local minimas by carrying the optimizer forward.
Hyperparameters:

- **Learning Rate** (η): Controls the size of the steps the optimizer takes in the direction of the negative gradient in each loop. It affects how fast or slow the model learns (usually in between 0.001 to 0.1)
- **Momentum Coefficient** (γ): Controls the contribution of the past gradients to the current update. A higher momentum coefficient means that more of the previous gradients are included, which can help to smooth the updates and accelerate convergence (usually in between 0.8 to 0.99)

Algorithm:

- **Initilization**:
  - Parameters: Initialize the parameters(weights) of the model, either as a zero matrix or randomly. (It is preferred to initialize it randomly)
  - Hyperparameters : Initialize the hyperparameters so a value in the typical range however these values can be modified later depending on need.
- **Loop:**
  1. Calculate the gradient : In each iteration, compute the gradient of the loss function with respect to each of the parameters. If the parameters are θ_t, cost function is $J(\theta)$, where subscript t denotes the iteration, then the gradient $g_t = \nabla_\theta J(\theta_t)$
  2. Update Velocity recursively : $v_t = \gamma v_{t-1} + \eta g_t$
  3. Update Parameters recursively : $\theta_t = \theta_{t-1} - v_t$

Root Mean Square Propogation

The RMSProp optimizer is one which is used in specific scenarios, where the Vanilla SGD and SGD with Momentum take a lot of iterations to converge to the minima of the loss function. The main feature of this optimizer is that it is able to adapt the learning rate separately for each parameter(weights) of the model. Note that this is not the only optimizer that uses this idea of adaptive gradients.

Hyperparameters

- **Learning Rate** (α): Controls the step size during the parameter update.
- **(β):** : Controls how influential the past velocities will be in the current update (Usually kept as 0.9)
- **Epsilon** (ϵ): A small constant to prevent division by zero (e.g., 10^{-8}).

Algorithm:

- **Initilization**:
  - Parameters: Initialize the parameters(weights) of the model, either as a zero matrix or randomly. (It is preferred to initialize it randomly)
  - Hyperparameters : Initialize the hyperparameters so a value in the typical range however these values can be modified later depending on need.
- **Loop:**
  1. Update Velocity recursively : $v_(t+1) = \beta v_t + (1-\beta)\nabla W_t^2$
  2. Update Parameters recursively : $W_{t+1} = W_t - \alpha \frac{\nabla W_t}{\sqrt{V_{t+1}+\epsilon}}$

Adam

ADAM algorithm is an optimization technique that combines the advantages of two other popular optimization algorithms AdaGrad and RMSProp. ADAM is able to achieve a variable learning rate for each parameter independently by utilizing first and second moments allowing it to achieve convergence faster and optimize for all parameters simultaneously.

Hyperparameters:

- **Learning Rate** (α): Controls the step size during the parameter update.
- **Decay Rates for the moment estimates:**
  - β_1: Exponential decay rate for the first moment estimates (usually close to 1, e.g., 0.9).
  - β_2: Exponential decay rate for the second moment estimates (usually close to 1, e.g., 0.999).
- **Epsilon** (ϵ): A small constant to prevent division by zero (e.g., 10^{-8}).

Algorithm:

- **Initialization**:
  - **Parameters**: Initialize parameters (weights) of the model, typically randomly.
  - **Moment Estimates**: Initialize the first moment (mean) $m_0 = 0$ and the second moment (uncentered variance) $v_0 = 0$.
- **Loop**:
  1. **Gradient Computation**:
     - At each time step t, compute the gradient of the loss function with respect to the parameters θ_t: $g_t = \nabla_\theta J(\theta_t)$.
  2. **Update Biased First Moment Estimate**:
     - $m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$
  3. **Update Biased Second Moment Estimate**:
     - $v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$
  4. **Compute Bias-Corrected First Moment Estimate**:
     - $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$
  5. **Compute Bias-Corrected Second Moment Estimate**:
     - $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$
  6. **Parameter Update**:
     - Update the parameters θ_t using the corrected estimates:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Bias Correction in necessary in intial stages of algorithm as β_1 and β_2 are close to 1 and m_0 and v_0 are initialized to 0 , effect of gradients will be significantly reduced at beginning if bias correction is not performed.
In the later stages as β_1^t and β_2^t tends to 0 bias correction is insignificant.

Experiments

We compared the performance of all the learning algorithms over MNIST dataset with simple dense neural network architecture. Our network had an input layer with 28*28 (784) input features which correspond to pixel brightness of greyscale digits followed by a hidden layer with 50 features and ReLU activation and an output layer with 10 features with Softmax activation corresponding to the probability of digit.

For a fair comparison between all the learning algorithms, we ran each algorithm with various values of hyperparameter, with fixed initial weights and bias initialization. This helped us to choose the best hyperparameter values based on model architecture, irrespective of random initialization.

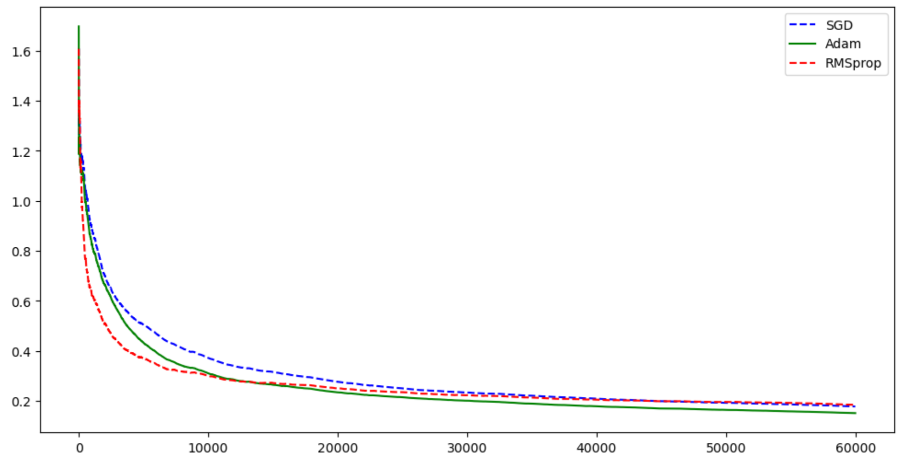

Figure 1: MSE vs Iteration (Best Parameters)

The above figure shows MSE Learning Curve on MNIST Dataset. All the algorithms were trained for 1 epoch over the entire dataset. Although RMSprop had fast initial learning rate, adam outperformed it shortly and converged to a lower MSE value. The best parameters were found by running the algorithms with different parameters and choosing the best-performing parameter.

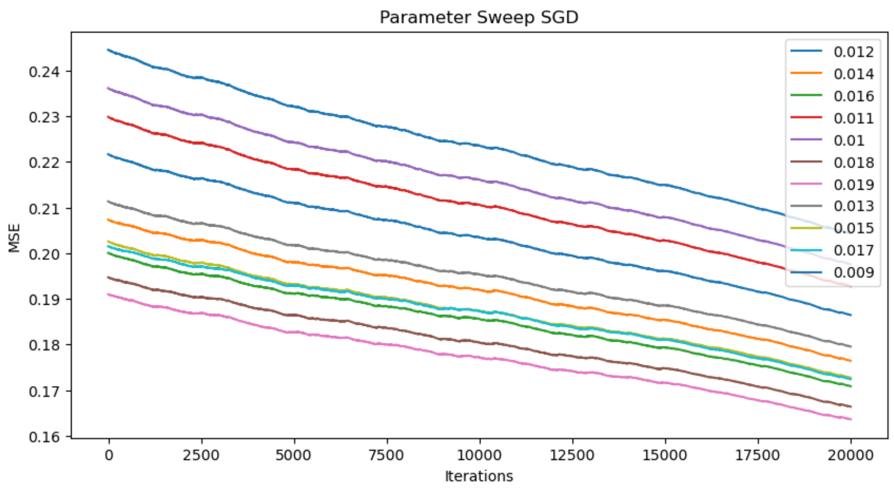

Figure 2: Stepsize Sweep for SGD (Optimal step size 0.019)

With step size 0.019, SGD consistently had an accuracy of about $91 - 93\%$ on the test set when trained for 1 Epoch.

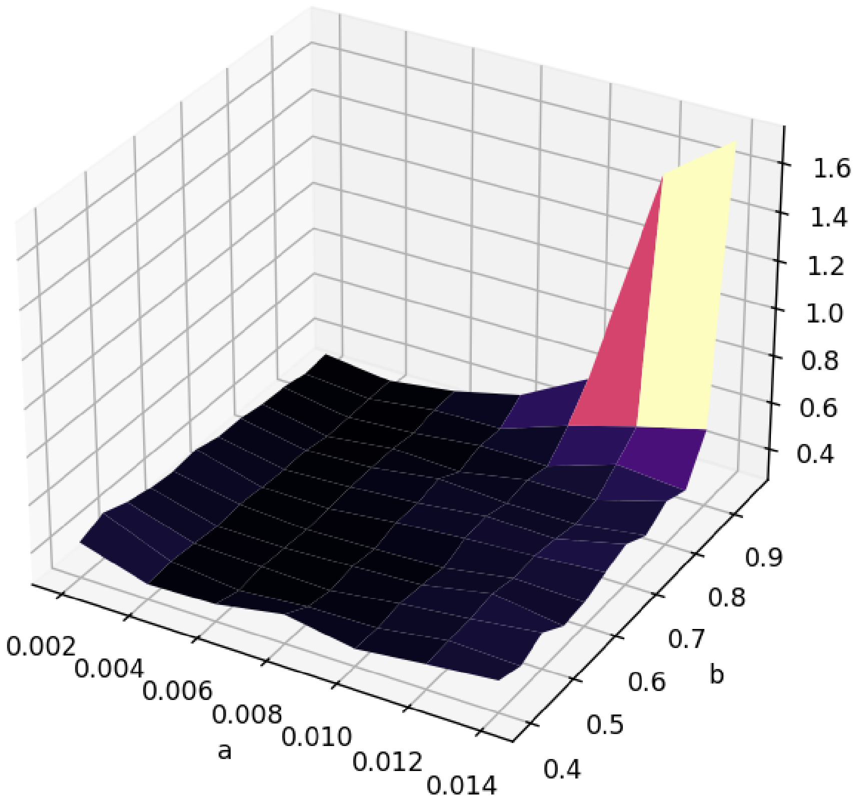

Figure 3: Stepsize(a) vs Moment Estimate(b) vs MSE for RMSprop

RMSprop works good over a range of parameter values. We used Stepsize(a) = 0.006 and Moment Estimate(b) = 0.6 and it had an accuracy of about $91 - 93\%$ on the test set when trained for 1 Epoch.

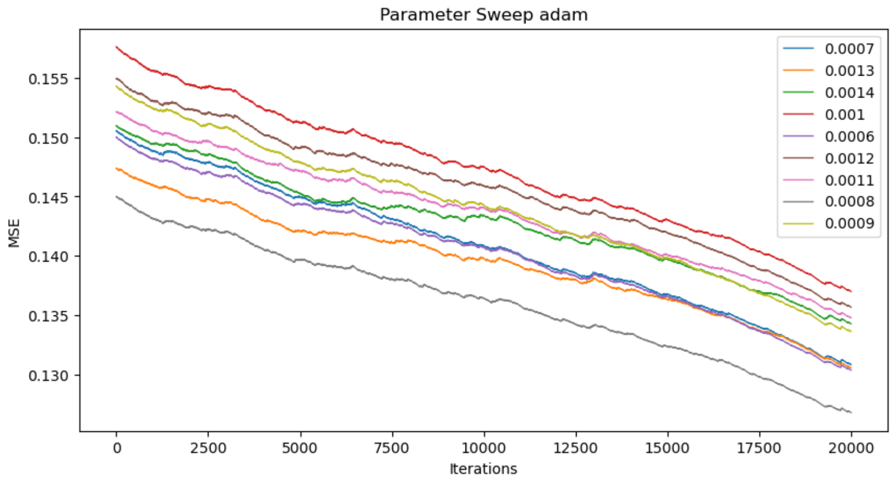

Figure 4: Stepsize sweep for Adam

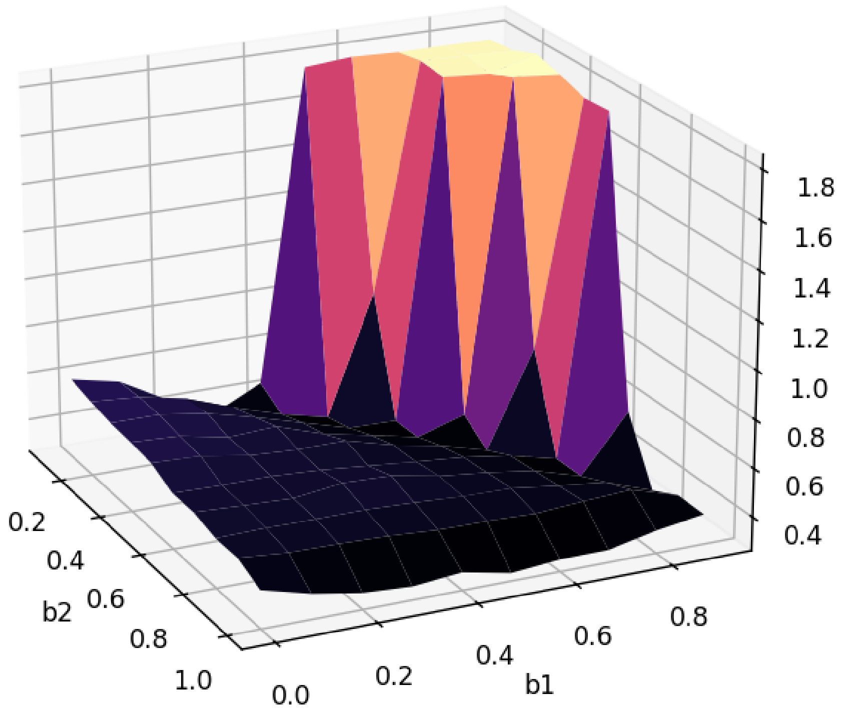

Figure 5: Moment Estimate Parameters b1 vs b2 vs MSE Adam

Adam performs well for small steps size close to 8e-4. Here we ran the algorithm with various stepsize and Moment Estimate Parameters fixed to b1 = 0.9 and b2 = 0.999. Changing the Moment Estimate Parameters slightly produced similar result. Adam moment estimate parameters work well over a range of values but the best parameter values among all are b1 = 0.5 and b2 = 0.999. The parameters produced accuracy of about $93 - 95\%$ on the test set when trained for 1 Epoch.

Conclusions

In this paper, we conducted a comparison of various optimization algorithms including SGD,Momentum, RMSProp, and Adam, to evaluate their performance in training in neural networks. We analyzed each optimizer's convergence behaviour and sensitivity to hyperparameters using visualisations of the MSE over iterations. Our results showed that the

Adam optimizer consistently outperformed other methods across different hyperparameter settings, achieving the fastest convergence and the lowest MSE values on the MNIST dataset.

Our findings confirm that while the choice of optimizer significantly impacts convergence speed and final accuracy, Adam provides a more efficient and reliable solution for neural network training, especially in large-scale settings. This conclusion serves as a guide for practitioners in selecting appropriate optimizers for deep learning applications, encouraging further exploration into hybrid optimizers that can achieve even better performance.

Appendix

Convergence of Adam