Evaluation of Various Gradient Descent Optimization Techniques for Neural Networks

Mit Patel
Computer Science Department
Arizona State University
Email: mjpatel8@asu.edu

Dhaval Patel
Computer Science Department
Arizona State University
Email: dpatel48@asu.edu

Prajeet Bhavsar
Computer Science Department
Arizona State University
Email: pdbhavsa@asu.edu

***Abstract*— Ever since the inception of Gradient Descent algorithm, it is without a doubt the most popular optimization strategy used in machine learning and deep learning. In this paper we have used various techniques to accelerate the gradient vectors in the right direction. The main problem with gradient descent algorithm is the rate of convergence. So to speed up the process, we take into effect all the previous gradients and tune the current gradient accordingly. There are various techniques available to do so and we have explored 5 such techniques namely i) No Momentum ii) Polyak's Classical Momentum iii) Nesterov's Accelerated Gradient (iv) RmsProp and (v) ADAM . We will compare the accuracies, rate of convergence and the stability for all this techniques in this paper.**

***Index Terms*—Gradient Descent, Polyak's classical momentum, Nesterov's Accelerated Gradient, RmsProp, ADAM, MNIST fashion Data Set**

I. INTRODUCTION

THE most common method for neural network optimization is gradient descent and is one of the most favored algorithms. And almost every deep learning library is equipped with various implementations of numerous algorithms to optimize the gradient descent. In this paper, we have implemented some of those techniques and compare their results.

The main objective of Gradient descent is to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by changing the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. To control the size of leaps we take to converge (local minima), we use hyper parameter η – learning rate. Here choosing the learning rate value poses a big problem. A small learning rate can lead to slow convergence and a large value of η may overshoot the minima. Particularly, we go along the direction of the slope of the surface created by the objective function downhill until we reach a minima. Now this is easily achieved if the objective function is convex in nature. But that is not usually the case in real world problems. And if the objective function is not convex in nature, the probability of converging to a local minima is very high.

The nature of many real world machine learning problems have non convex formulation of the objective function. The problem of finding the global minimum in such cases is treated as NP hard problem [1]. Gradient Descent is naturally the first go to iterative algorithm to solve these problems. There are a number of variations of Gradient Descent methods and most of them can be collectively classified into these three: i) Momentum-based methods (e.g. Nesterov's Accelerated Gradient), ii) Variance Reduction Methods (e.g. Stochastic Variance Reduced Gradient) and iii) Adaptive Learning Methods (e.g. AdaGrad) [1].

In this work, we study momentum-based methods and implement the same on MNIST fashion dataset [2]. The paper is organized as follows: Section 2 discusses the various techniques and background, Section 3 discusses the methodology and architecture of our neural network, Section 4 showcases the comparisons between the performances of different techniques on the dataset, Section 5 contains the conclusions that we drew from the experiment, Section 6 shows the division of work for our project and finally Section 7 has the self-peer evaluation table followed by references.

II. RELATED WORK

As mentioned in the above section, the estimation of the network parameters can take a lot of time if we have a large dataset and the learning rate we are using. The equation for the gradient descent parameter update is given by [3]:

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{1}$$

So in order to speed up the convergence in gradient descent, some of the widely used optimization techniques are Polyak's classical momentum, Nesterov's Accelerated Gradient, RMSProp, ADAM, Adagrad and AdaDelta.

A. *Polyak's classical momentum:*

This method accelerates the stochastic gradient descent by taking the exponentially weighted average of the gradients into consideration. Here the parameters are updated by adding a fraction γ of the previous iteration's update vector to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \tag{2}$$
$$\theta = \theta - v_t \tag{3}$$

The dimensions whose gradients are in same direction, the momentum term γ will increase and hence this will result in faster convergence [3].

B. *Nesterov's Accelerated Gradient:*

In the momentum method, as we reach toward the minima, the momentum can be high which may lead to miss the lowest point. In Nesterov's Accelerated Gradient, we compute $\theta - \gamma v_{t-1}$ which gives us a rough estimate of the next position of the parameters.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \tag{4}$$
$$\theta = \theta - v_t \tag{5}$$

Thus, NAG provides an ability to look ahead by calculating gradient with respect to the rough estimate of future position of the parameters as opposed to the current parameters [4].

C. RMSProp:

The main idea behind RMSProp is to update the learning rate at each step. Many times it may be the case that infrequent and sparse features can hold vital information as compared to features occurring frequently. In order to address this problem, techniques like RMSProp and ADAM adapts learning rate to the parameters by performing small updates for parameters related to frequent features and larger updates for parameters associated with features occurring infrequently.

Let g_t be the gradient at time step t. Here the sum of gradients is defined recursively as a decaying average of previously squared gradients. Hence the running average $E[g^2]_t$ at time step t will depend only on the previous average and the current gradient [3]. RMSProp will then divide the learning rate by this exponentially decaying average of squared gradients.

$$E[g^2]t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2 \quad (6)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \quad (7)$$

In practice for good results, γ is set to 0.9 and the learning rate η to 0.001.

D. ADAM:

In ADAM, we store exponentially decaying average of past squared gradients v_t as well as exponentially decaying average of past gradients m_t, and use it to update the weights θ_{t+1}.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \quad (8)$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \quad (9)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} m_t \quad (10)$$

The default values of 0.9 for β_1 , 0.999 for β_2, and 10^{-8} for ϵ are favorable. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

III. Methodology

For the implementation of above mentioned optimization techniques, we have used Fashion MNIST dataset which is a collection of images of various articles of clothing and accessories. Each sample is a 28x28 grayscale image from possible ten classes. We have preprocessed the data by dividing the data by 255 and then subtracting the respective means of each feature.

To classify these images, we have used a 3 layer fully connected neural network (i.e. 2 hidden layers). Generally in conventional neural networks, weights and biases are considered as different parameters. As a result, we need to update both the weights and bias in every iteration. However, in our implementation, we have stored the bias in the weight vector itself and so we need update just a single parameter. For the two hidden layers, we have used Rectified Linear unit (relu) activation function which is defined as:

$$g(z) = \max(0, z) \quad (11)$$

and thus relu's derivative can be written as:

$$g'(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases} \quad (12)$$

At the last layer which gives the output, SoftMax activation is used. The fully connected neural network diagram is shown below:

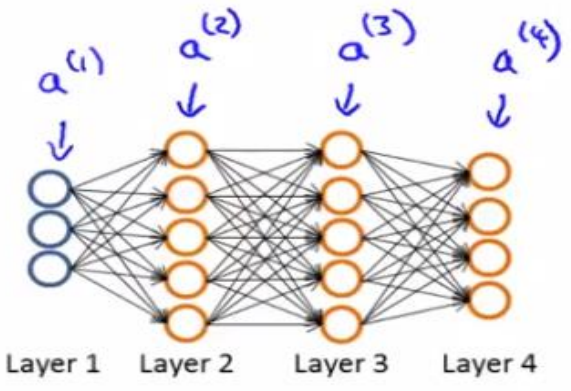

Figure 1. Fully connected 3 layer network

The input layer will have total 785 nodes (28x28 + one node which will have value +1 always, hence the weight connected with this node will act as bias) and 10 nodes in the output layer. As mentioned earlier, bias units are included in the weight vector itself by adding an extra node that has value +1 in every layer. Thus the formula to calculate z, $z = wx + b$ changes to $z = w'x'$ where $w' = [\, b\ w\,]$ and $x' = [\, 1\ x^T\,]^T$. Also, the weights are initialized by uniform distribution multiplied by $\sqrt{2/size\ of\ previous\ layer}$.

The formulas to find 'z' and 'a' during forward propagation in each layer are described below:

$$a^{(1)} = [\, 1\ x\,] \quad (13)$$

$$z^{(2)} = \theta^{(1)} a^{(1)} \quad (14)$$

$$a^{(2)} = g\left(z^{(2)}\right) \quad (\, add\ a_0^{(2)} = +1\,) \quad (15)$$

$$z^{(3)} = \theta^{(2)} a^{(2)} \quad (16)$$

$$a^{(3)} = g\left(z^{(3)}\right) \quad (\, add\ a_0^{(3)} = +1\,) \quad (17)$$

$$z^{(4)} = \theta^{(3)} a^{(3)} \quad (18)$$

$$a^{(4)} = h_\theta(x) = g\left(z^{(4)}\right) \quad (19)$$

IV. EXPERIMENT

We implemented the 3 layer fully connected neural network for four different learning rates – 0.0001, 0.001, 0.01 and 0.1 with 500 and 100 nodes in the two hidden layers respectively and relu as the activation function in both the layers. Also, the experiment was performed for the training set of 10,000 samples, validation set of 1000 samples and 5000 test samples. Moreover, the weights initialized for each technique are same for a particular learning rate. The results for various gradient optimization techniques and the intuition behind them is discussed below:

A. *Learning rate = 0.0001:*

For all the above mentioned learning rates, we have implemented the Polyak's momentum, Nestorov's Accelerated Gradient (NAG), RMSProp and ADAM optimization techniques.

The accuracies for different optimization techniques is given below:

	Training accuracy	Validation accuracy	Test accuracy
No momentum	0.24	0.158	0. 11
Polyak's Momentum	0.705	0.426	0.395
NAG	0.7053	0.427	0.401
RMSProp	0.963	0.668	0.65
ADAM	0.945	0.649	0.639

Table 1. Accuracies for $\eta = 0.0001$

So from the above observation, we can say that RMSProp and ADAM performs better compared to other technique for low learning rates. Furthermore, the plot for validation cost vs iterations for various technique is described in the following figure.

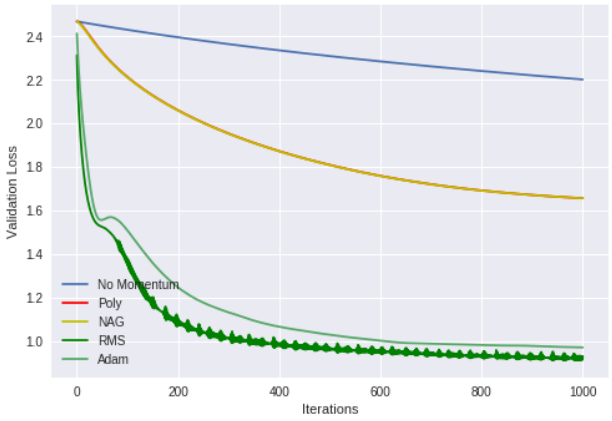

Figure 2. Validation cost curve for $\eta = 0.0001$

From the above plot, we can infer that NAG and Polyak's momentum performs almost same because of very low learning rate. RMSProp performs slightly better compared to ADAM, however it is somewhat less stable compared to ADAM due to small fluctuations in the cost curve. The iterations required to reach a particular cost value can be considered a good factor to measure speed of convergence. The approximate number of iterations to reach the validation cost of 1.1 is given in the following table:

	Approximate iterations
No momentum	>1000
Polyak's Momentum	>1000
NAG	>1000
RMSProp	200
ADAM	400

Table 2. Convergence for $\eta = 0.0001$

It is clear that the No momentum, Polyak's momentum and NAG will require more iterations to converge because the learning rate is very low and hence their gradients will update slowly. Whereas, RMSProp and ADAM converges quickly to the local minima.

B. *Learning rate = 0.001:*

	Training accuracy	Validation accuracy	Test accuracy
No momentum	0.698	0.34	0.334
Polyak's Momentum	0.8192	0.553	0.542
NAG	0.819	0.55	0.543
RMSProp	0.994	0.729	0.716
ADAM	1.0	0.607	0.582

Table 3. Accuracies for $\eta = 0.001$

Table 3 depicts the accuracies for the various optimization techniques. We can infer that RMSProp performs best when learning rate is 0.001. The first three momentum techniques generally give bad results when learning rates are comparatively low. The validation cost vs iteration curve for $\eta = 0.001$ is described in the plot below:

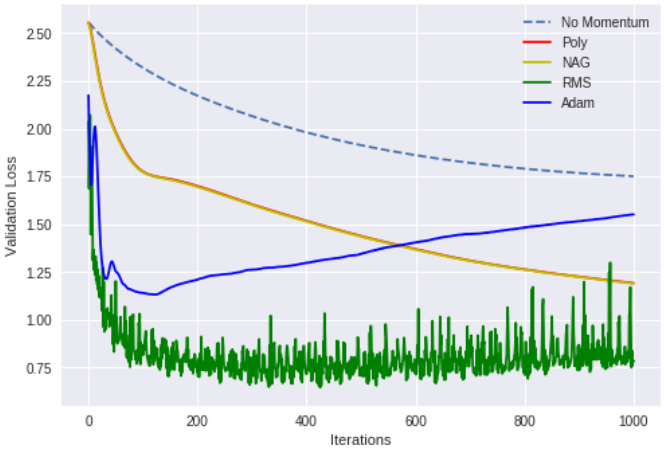

Figure 3. Validation cost curve for η = 0.001

Here despite showing the best accuracies, RMSProp is unstable compared to the remaining techniques. After some iterations, ADAM and RMSProp starts overfitting the samples and hence their validation cost curves are increasing. NAG and Polyak's momentum shows similar behavior and hence their curves are overlapping. The convergence table for validation cost of 1.25 is described below:

	Approximate iterations
No momentum	>1000
Polyak's Momentum	805
NAG	800
RMSProp	20
ADAM	30

Table 4. Convergence for η = 0.001

So ADAM and RMSProp converge quickly compared to others. Moreover, NAG and Polyak's momentum show higher convergence rate compared to the no momentum method.

C. *Learning rate = 0.01:*

The accuracy results for various optimization techniques is depicted in the following table.

	Training accuracy	Validation accuracy	Test accuracy
No momentum	0.82	0.564	0.558
Polyak's Momentum	0.91	0.714	0.691
NAG	0.909	0.715	0.698
RMSProp	0.98	0.638	0.645
ADAM	1.0	0.621	0.616

Table 5. Accuracies for η = 0.01

Here the accuracy results for No momentum, Polyak's momentum and NAG are better compared to the previous learning rates. Figure 4 depicts the zoomed in plot for validation cost against iterations:

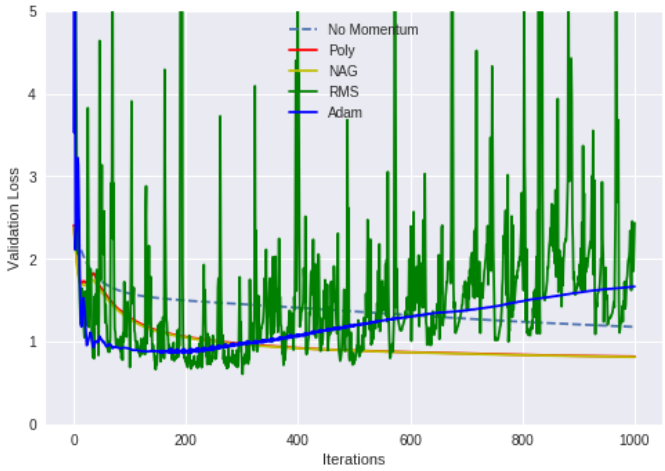

Figure 4. Validation cost curve for η = 0.01

After some finite iterations, the cost for RMSProp and ADAM starts to increase which indicates the overfitting of training data. Learning rate of 0.01 is still high for ADAM as well as RMSProp. So we observed some initial fluctuations in ADAM but as number of iteration increases, the curve becomes smoother. The same cannot be said for RMSProp, which fluctuates throughout the training period.

	Approximate iterations
No momentum	800
Polyak's Momentum	140
NAG	135
RMSProp	20
ADAM	20

Table 6. Convergence for η = 0.01

Above table depicts the convergence properties for different techniques for validation cost of approximately 1.25. Here the momentum techniques viz. Polyak's momentum and NAG shows good convergence as compared to when η is very low.

D. *Learning rate = 0.1:*

The table 7 compares the accuracies for different techniques. Since the learning rate is very high, RMSProp and ADAM shows very poor results. And the gradient updates for the remaining momentum techniques is better and hence they provide decent results.

	Training accuracy	Validation accuracy	Test accuracy
No momentum	0.9108	0.68	0.66
Polyak's Momentum	0.995	0.62	0.591
NAG	0.996	0.672	0.647
RMSProp	10	10	10
ADAM	10	10	10

Table 7. Accuracies for $\eta = 0.1$

The figure 5 depicts the plot for validation cost vs iteration when $\eta = 0.1$:

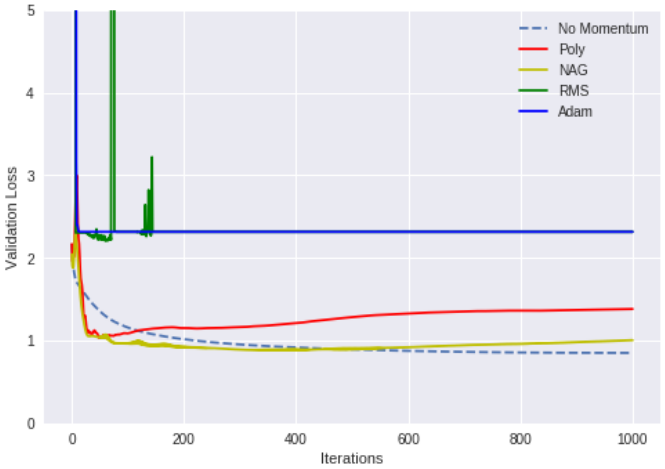

Figure 5. Validation cost curve for $\eta = 0.1$

Here a constant value of cost for RMSProp and ADAM shows bad gradient updates while increase in cost for NAG and Polyak's momentum indicates overfitting of samples. Hence RMSProp and ADAM are unstable when learning rate is very high.

	Approximate iterations
No momentum	110
Polyak's Momentum	45
NAG	40
RMSProp	No convergence
ADAM	No convergence

Table 8. Convergence for $\eta = 0.1$

From the above convergence table (for validation cost = 1.2), we can conclude that ADAM and RMSProp are not converging because of overshooting of gradient updates. Remaining techniques show good convergence properties due to high η.

So during our project we wondered that if we used the momentum techniques for gradient decent, do we really need the learning rate parameter. Well it turns out that learning rate parameter cannot be excluded even if we use the best gradient decent optimization technique. The main reason is if we exclude the learning rate parameter from gradient update, the gradients won't change. This is because no learning will happen if learning rate is 0. Momentum techniques help learning rate so that the network can converge faster. Momentum targets to improve the rate of convergence by avoiding local minima and thus cannot enable learning by itself. In brief, learning rate is important for updating the weight parameters to minimize error and momentum is used to help learning rate achieve that task and not replace learning rate.

V. Conclusion

After analyzing the various gradient descent optimization techniques, we came to the conclusion that RMSProp gives better accuracies for lower learning rates. If the training data is sparse, then no momentum, Polyak's momentum and NAG will give poor result. For sparse data, one should use adaptive optimization techniques. We also saw that RMSProp was quite unstable for even low learning rates and hence if we were to use mini batch gradient descent, the fluctuations in the cost will grow even more. Hence for better convergence and stability, ADAM is widely used in practice compared to other optimization techniques.

VI. Division of work

I was responsible for implementing the ADAM momentum technique. When the code base was ready, we divide the three architectures among ourselves for testing. I tested all the momentum techniques on different learning rates. I tested the network on two different learning rates viz. 0.0001 and 0.001. Finally we combined all our results and prepared the report.

VII. Self-peer evaluation table

Dhaval Patel: 20	Prajeet Bhavsar:20	Myself: 20

Table 9. Self-Peer Evaluation Table

VIII. References

[1] Vishwak Srinivasan, Adepu Ravi Sankar and Vineeth N Balasubramanian,"ADINE: An Adaptive Momentum Method for Stochastic Gradient Descent ",Indian Institute of Technology Hyderabad

[2] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[3] S. Ruder, An overview of gradient descent optimization algorithms, 2018 [Online]. Available: http://ruder.io/optimizing-gradient-descent/

[4] Anish Singh Walia, Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent. [Online]. Available: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

IX. Gradient Updates

```
def no_momentum_gradient_update(self, alpha, sample_size, t):
        for i in range(1, len(self.layer)):
            gradient = (self.layer[i].error[1:] @ self.layer[i-1].A.T)/sample_size
            self.layer[i].weights = self.layer[i].weights - (alpha * gradient)

#NAG and poly has same momentum formula
#Forward and backward calculation in NAG occurs with (weights - gamma * Vt)
def poly_momentum_gradient_update(self, alpha, sample_size, t):
    for i in range(1, len(self.layer)):
        gradient = (self.layer[i].error[1:] @ self.layer[i-1].A.T)/sample_size
        self.layer[i].v = (self.gamma * self.layer[i].v) + (alpha * gradient)
        self.layer[i].weights = self.layer[i].weights - self.layer[i].v

def rms_prop_momentum_gradient_update(self, alpha, sample_size, t):
    for i in range(1, len(self.layer)):
        gradient = (self.layer[i].error[1:] @ self.layer[i-1].A.T)/sample_size
        self.layer[i].gradient_square_sum = (self.gamma * /
                                        self.layer[i].gradient_square_sum) + /
                                        ((1-self.gamma) * (gradient * gradient))
        self.layer[i].weights = self.layer[i].weights - /
                                ((alpha/np.sqrt(self.layer[i].gradient_square_sum + /
                                1e-8)) * gradient)

def adam_gradient_update(self, alpha, sample_size, t):
    for i in range(1, len(self.layer)):
        gradient = (self.layer[i].error[1:] @ self.layer[i-1].A.T)/sample_size
        self.layer[i].m = (self.beta1 * self.layer[i].m) + ((1-self.beta1) * gradient)
        self.layer[i].v = (self.beta2 * self.layer[i].v) + ((1-self.beta2) * /
                               (gradient * gradient))
        m_corrected = self.layer[i].m / (1 - (self.beta1**t))
        v_corrected = self.layer[i].v / (1 - (self.beta2**t))
        self.layer[i].weights = self.layer[i].weights - (alpha/(np.sqrt(v_corrected)+/
                                  1e-8)) * m_corrected
```