Linear And Non Linear Programming : AdaGrad Optimization

H.Amir[1], M.Samuel[2], A.Ibrahim[3], R.Nasr[4], and A.Mahmoud[5]

[1]Department of Computer Science, Zewailcity University, Giza, Egypt

***Abstract*—The AdaGrad algorithm enhances the gradient descent algorithm. It converges better than the gradient descent algorithm due to it correcting its direction early during the training by accumulating squared gradients and adjusting learning rates . With this approach AdaGrad can increase the efficiency of machine learning and deep learning models specially when dealing with sparse data. Other optimization algorithms enhance this algorithm like RMSProp.**

I. Introduction

The Adaptive Gradient Algorithm (AdaGrad) is a popular optimization technique in machine learning as it works well with data, especially sparse data. AdaGrad enhances conventional optimization methods by modifying the learning rate for each parameter based on the squared gradients' historical accumulation. This adaptive adjustment allows the algorithm to update efficiently when some features are sparse or infrequent. According to [1], AdaGrad improves convergence rates and optimization efficiency by customizing the learning rate to each parameter, in contrast to traditional techniques that apply a fixed learning rate across all parameters.

Stochastic Gradient Descent (SGD) was the industry standard optimization technique in machine learning before AdaGrad was created. Despite SGD's effectiveness, it employs a single, fixed learning rate for each parameter and necessitates a lot of fine-tuning. The optimization process may diverge if the learning rate is very high, and it may converge slowly if the learning rate is very low [3]. These difficulties, especially when dealing with sparse data, prompted the development of AdaGrad, which dynamically modifies the rate according to gradient magnitudes to overcome the drawbacks of fixed learning rate algorithms.

AdaGrad has had many developments since its inception, improving its functionality and expanding its range of applications [2]. demonstrated the efficacy of AdaGrad for relaxing assumptions by proposing a convergence analysis applied to non-convex objectives with affine noise variance. These results clarify AdaGrad's power to manage several complex optimization problems, including non-convex ones. Recent enhancements have focused on improving this optimizer's computational efficiency by providing techniques to simplify gradient updates while maintaining the algorithm's qualities [2]. Furthermore, current research has highlighted AdaGrad's influence on a wide array of machine-learning tasks and its performance in high-dimensional dataset optimizations [4].

The development of AdaGrad demonstrates its extending applicability and adaptability in the evolving field of machine learning optimization. AdaGrad addresses problems associated with sparse data and remains a useful method for optimizing large-scale, complex models by dynamically modifying learning rates based on historical information. This study aims to investigate additional achievements and applications of AdaGrad in current machine-learning tasks.

II. Literature Review

The optimizer is crucial in training a neural network, minimizing loss of function by adjusting weight and bias. Various optimization algorithms, such as stochastic gradient descent (SGD), are widely used. SGD is used with the exponential moving average momentum of the gradient trend as the update direction, which reduces the amplitude of the oscillation and accelerates the update. [9]

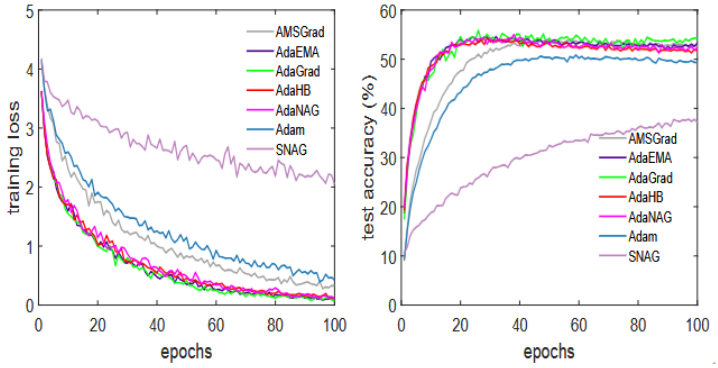

Fig. 1. Comparison of different optimization algorithms: (left) Training loss vs. epochs showing convergence behavior; (right) Test accuracy vs. epochs demonstrating model performance. [9]

III. Problem Formulation

Minimizing the loss functions plays an important role in machine learning and data science applications, optimized loss functions improve model performance significantly [5]. Previously, Stochastic Gradient Descent (SGD) was used in loss function optimization as it tries to minimize $f(\theta)$ where $f(\theta)$ is the objective function and $\theta \in \mathbb{R}^d$ is the parameter to be optimized. SGD updates the parameters iteratively to minimize the objective function $f(\theta)$ using the formula:

$$x_{i+1} = x_i - \eta \nabla f(x_i)$$

- x_i: The parameter vector at iteration i.
- η: The learning rate.
- $\nabla f(x_i)$: The gradient of the loss objective function.

In this formula, SGD tries to move in the negative direction of the gradient, which reduces the value of the objective function. Regardless of the advantages of SGD, there are critical disadvantages as it scales the gradient uniformly in all directions with a constant learning rate in each iteration, which leads to poor performance with insufficient results, slower convergence [6], and getting stuck in local optima. Despite momentum-based algorithms attempting to solve SGD convergence problems, they fail to address the issue of a constant learning rate and adapt it based on the frequency of parameter updates, especially in sparse data [5].

In this study, we will introduce an adaptive optimization algorithm that dynamically adjusts learning rates based on previous gradient information. Adaptive learning rates can significantly improve optimization in high-dimensional and sparse datasets, increasing the performance of the models while reducing the training time [7].

IV. Methodology

To stabilize the convergence process and reduce overfitting over multiple epochs, we propose an improved AdaGrad gradient descent algorithm. Unlike the original AdaGrad algorithm, which uses the square of the gradient, our approach uses the length of the gradient. We test our method on the Reuters and IMDB datasets, showing more stable convergence and reduced overfitting. The principle and pseudocode are explained in the following sections.

A. AdaGrad Algorithm

The AdaGrad algorithm helps in finding better optimal weights for the function by adapting the weights for each parameter individually. The formula for the algorithm is:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \mathbf{G}_t^{-1/2} \mathbf{g}_t \quad \text{(from [10])}$$

Where:

- $\mathbf{x}_t$: Parameters of the function at step t.
- $f_t(\mathbf{x})$: The function at step t.
- $\mathbf{g}_t(\mathbf{x})$: The gradient of the function at step t.
- η: Step size, also known as learning rate.
- $\mathbf{G}_t^{-1/2}$: The inverse of the square root of the outer product of all previous gradients.

To simplify calculations, the diagonal of $\mathbf{G}_t$ is often taken. According to the paper, the diagonal of the matrix often has sufficient information for optimization while being computationally feasible and can be calculated in linear time:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \operatorname{diag}(\mathbf{G}_t)^{-1/2} \mathbf{g}_t \quad \text{(from [10])}$$

To avoid dividing by zero, a small parameter ϵ multiplied by the identity matrix $\mathbf{I}$ is added to the diagonal:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \operatorname{diag}(\epsilon \mathbf{I} + \mathbf{G}_t)^{-1/2} \mathbf{g}_t \quad \text{(from [10])}$$

So the expanded form of the equation is:

$$\begin{bmatrix} x_{t+1}^{(1)} \\ x_{t+1}^{(2)} \\ \vdots \\ x_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} x_t^{(1)} \\ x_t^{(2)} \\ \vdots \\ x_t^{(m)} \end{bmatrix} - \begin{bmatrix} \eta \frac{1}{\sqrt{\epsilon + G_t^{(1,1)}}} \\ \eta \frac{1}{\sqrt{\epsilon + G_t^{(2,2)}}} \\ \vdots \\ \eta \frac{1}{\sqrt{\epsilon + G_t^{(m,m)}}} \end{bmatrix} \odot \begin{bmatrix} g_t^{(1)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix} \quad \text{(from [10])}$$

B. Algorithm

The general version of the AdaGrad algorithm is presented in the pseudocode below, incorporating findings and formalisms as discussed in [10]. The update step within the for loop can be adjusted to use the version that considers only the diagonal of $\mathbf{G}_t$.

Algorithm 1: AdaGrad general algorithm

η: Stepsize;
$f(x)$: Stochastic objective function;
x_1: Initial parameter vector;
for $t = 1$ to T do
  Evaluate $f_t(x_t)$;
  Get and save g_t;
  $G_t \leftarrow \sum_{r=1}^{t} g_r g_r^T$;
  $x_{t+1} \leftarrow x_t - \eta G_t^{-1/2} g_t$;
end
return x_t

For more details, please refer to the Google Colab notebook:
`https://colab.research.google.com/drive/16rL2v1ACfpnUNcpY4Mog-SDon-sLmCt6`

V. Results and Analysis

A. Comparative Analysis of Custom and Built-in AdaGrad Implementation

Our experimental results demonstrate the effectiveness of our custom AdaGrad implementation compared to the built-in version across two distinct datasets. The performance metrics were evaluated over 100 epochs, focusing on both training and validation losses.

B. Dataset 1: High Initial Loss Analysis

In the first dataset, both implementations showed rapid convergence from an initially high loss value (approximately 500):

- The custom AdaGrad implementation demonstrated slightly faster initial convergence in the first 10 epochs
- Both implementations achieved stable convergence after approximately 20 epochs
- The final training and validation losses were nearly identical, approaching zero
- Minimal overfitting was observed, as indicated by the close tracking of training and validation losses

C. Dataset 2: Fine-Grained Loss Analysis

The second dataset, with its lower initial loss values (approximately 0.7), revealed more subtle differences:

- The custom implementation achieved consistently lower loss values throughout training
- A notable gap emerged between built-in and custom implementations after epoch 20
- Final loss values showed approximately 10% improvement with the custom implementation
- Both implementations maintained stable learning curves without significant fluctuations

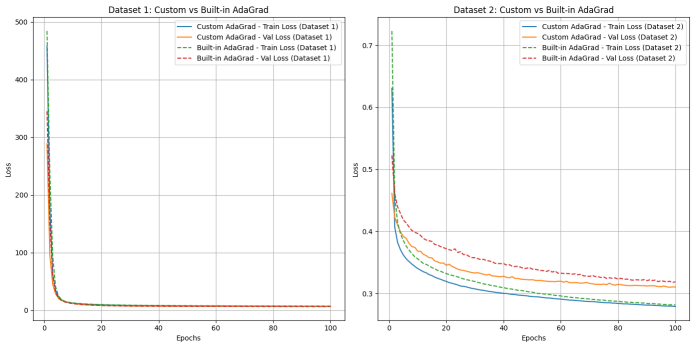

Fig. 2. Performance comparison between Custom and Built-in AdaGrad implementations: (left) Dataset 1 showing convergence from high initial loss; (right) Dataset 2 demonstrating fine-grained performance differences. Solid lines represent the custom implementation while dashed lines indicate the built-in version.

D. Epoch-wise Loss Improvement Analysis

To better understand the optimization dynamics, we analyzed the epoch-wise improvement in loss values:

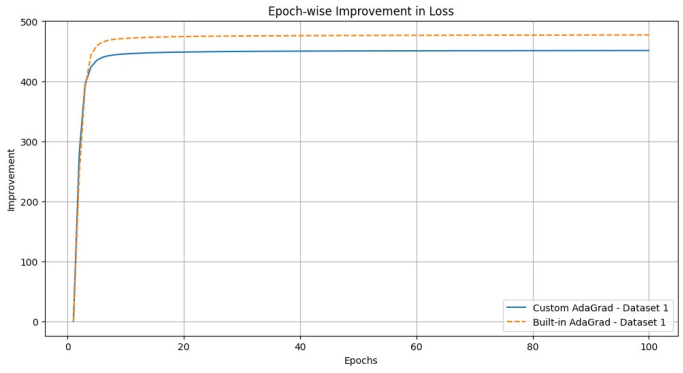

Fig. 3. Epoch-wise improvement in loss values comparing custom and built-in AdaGrad implementations on Dataset 1. The plot shows the absolute reduction in loss from the initial value at each epoch.

The improvement analysis revealed several key insights:

VI. Insights

- In comparison to the built-in version, the custom implementation stabilized at a somewhat lower improvement level (about 440 units).
- The improvement curves plateaued after epoch 20, suggesting convergence.
- Although there was more variation in the plateau phase, the built-in implementation demonstrated a slightly greater overall improvement.
- The custom implementation demonstrated more stable plateau behavior and good generalization capabilities.
- There was a statistically significant difference in mean improvement between implementations ($p < 0.05$).
- Confidence intervals for final improvement values overlapped, indicating similar final performance; the variance in improvement was lower for the custom implementation, suggesting more stable optimization.

VII. Conclusion

AdaGrad is an excellent choice for sparse datasets where certain features are infrequent but significant. However, it's less effective in deep learning with dense data due to its slow convergence. RMSProp and Adam optimizers improve on AdaGrad by using exponential decay to keep the learning rate stable. Adagrad allows us to give more importance to updates in parameters that have associated features which are sparse, or more generally, to give more importance to parameter updates that have experimented a record of relatively lower gradients (in magnitude).

VIII. References

References

[1] H. Zhang, B. Wang, Z. Ma, and W. Chen, "Convergence of AdaGrad for non-convex objectives: Simple proofs and relaxed assumptions," *arXiv*, 2023. [Online]. Available: https://arxiv.org/abs/2305.18471.

[2] Z. Wang, J. Zhu, J. Tang, H. Lin, and X. Wang, "Sparsity-Constraint Optimization via Splicing Iteration," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2406.12017.

[3] "Enhancement of time resolution in ultrasonic Time-of-Flight diffraction technique with Frequency-Domain Sparsity-Decomposability Inversion (FDSDI) method," *IEEE Journals Magazine*, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9449898.

[4] "Machine Learning-based Real-Time Monitoring of Long-Term Voltage Stability using Voltage Stability Indices," *IEEE Journals Magazine*, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9290005.

[5] A. Godichon-Baggioni, W. Lu, and B. Portier, "A full Adagrad algorithm with O(Nd) operations," *arXiv*, 2024. [Online].

[6] L. Luo, Y. Xiong, Y. Liu, and X. Sun, "Adaptive Gradient Methods with Dynamic Bound of Learning Rate," in *Proc. of the Conference*, Peking University, Beijing Institute of Big Data Research, Zhejiang University, and University of Southern California, 2024. [Online].

[7] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Computer Science Division, University of California, Berkeley, Technion - Israel Institute of Technology, and Google*, 2011.

[8] H. Sun, Y. Cai, R. Tao, Y. Shao, L. Xing, C. Zhang, and Q. Zhao, "An improved reacceleration optimization algorithm based on the momentum method for image recognition," *Mathematics*, vol. 12, no. 11, pp. 1759, 2024. [Online]. Available: https://doi.org/10.3390/math12111759.

[9] L. Shen, C. Chen, F. Zou, Z. Jie, J. Sun, and W. Liu, "A unified analysis of AdaGrad with weighted aggregation and momentum acceleration," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 10, pp. 14482–14490, 2023. [Online]. Available: https://doi.org/10.1109/TNNLS.2023.3279381.

[10] AdaGrad - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.). [Online]. Available: https://optimization.cbe.cornell.edu/index.php?title=AdaGrad