2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC) | 979-8-3503-6013-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/CCWC60891.2024.10427781

Multiple Gain Adaptations for Improved Neural Networks Training

Jeshwanth Challagundla*, Kanishka Tyagi†, Tushar Chugh‡ and Michael Manry§
Department of Electrical Engineering, The University of Texas at Arlington, Arlington, Texas
Email: *jeshwanth.challagundla@mavs.uta.edu, †kanishka.tyagi@mavs.uta.edu, §manry@uta.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, USA
Email: ‡tusharchugh19@gmail.com

***Abstract*—A two stage algorithm developed called Multiple Gain Adaptations for Improved Networks (MGAIN) is presented. MAGAIN alternatively finds output weights and uses several gain factors to update the input weights in a Multi-Layer Perceptron. The gain factors are computed using Newtons method. Our method dynamically adjusts the quantity of gain factors calculated to maximize the reduction in loss with each epoch. The results demonstrate that our approach outperforms existing second order algorithms across the majority of diverse datasets.**

***Index Terms*—Multilayer Perceptron, Back Propagation, Target Weight Refinement, Adaptive multiple gain adaptations, Orthogonal Least Squares.**

I. Introduction

Artificial Neural Networks (ANNs) consist of various architectures, with multilayer feed-forward networks being the predominant choice for tasks such as function approximation and pattern recognition. This is primarily due to their inherent capabilities for universal approximation and approximation of Bayes discriminants. [1] [2]

In the domain of neural network optimization, techniques such as Adam [3], AdaDelta [4], AdaGrad [5], RmsProp, Nesterov momentum [6], are predominantly aligned with first-order training algorithms. However, there exists a notable deficit in the literature regarding more advanced optimization strategies for second-order training algorithms. The primary challenge with second-order algorithms lies in their scalability issues, largely attributable to the necessity of computing Hessian matrices, which are notably resource-intensive. The computational demands escalate significantly as the network size increases, with the operations required for Hessian matrix computation growing exponentially.

Neural networks can approximate continuous discriminants arbitrarily well, due to universal approximation [1]. However, it raises questions about the No-Free-Lunch theorem [7] as it implies that multi-layer perceptrons (MLPs) can approximate alternative discriminants. For a first-order neural network optimization, Adafactor [8] presents a fusion of Adam optimization and memory efficiency, reducing memory overhead by factorizing moving average matrices into low-rank forms. While advantageous for its memory-saving attributes, Adafactor lacks detailed exploration regarding its performance across a wider range of neural network architectures and datasets, leaving gaps in understanding its adaptability beyond specific scenarios. Similarly, AdaSmooth [9] introduces adaptability in the window size for accumulated past gradients, surpassing the fixed-size constraints. However, AdaSmooth's effectiveness across highly dynamic or noisy datasets remains relatively unexplored, necessitating further investigation into its robustness in more complex optimization landscapes. Moreover, the Large Batch Optimization technique (LAMB) [10] demonstrates layer-wise adaptiveness in updating weights, enabling faster training of models like BERT [11]. Yet, LAMB's scalability and generalization to diverse neural architectures and tasks warrant deeper scrutiny, particularly to discern its limitations in scenarios involving smaller datasets or different model architectures. The SLAMB technique [12] amalgamates LAMB with sparse GPU communication, accelerating large batch training. However, the practical implementation challenges and trade-offs in deploying sparse communication methods, as well as the specific hardware requirements for optimal performance, require further elucidation for broader applicability in varied computing environments. Furthermore, the Symbolic Discovery of Optimization Algorithms (LION) [13] marks a pioneering attempt at exploring various optimization algorithms comprehensively. Nevertheless, the paper lacks an in-depth comparative analysis or benchmarking against existing optimization strategies, limiting its insights into the relative effectiveness or novelty of the discovered algorithms. Shifting focus to second-order optimization techniques, Shampoo [14] introduces efficient gradient preconditioning using matrices instead of Hessian computations. However, Shampoo's scalability concerns and its performance in scenarios with highly non-convex functions or noisy gradients remain areas that necessitate further investigation to ascertain its generalizability. Moreover, Scalable Second Order Optimization in [15] efficiently implements the Shampoo algorithm. Yet, a comprehensive examination of its computational complexity and comparison with other scalable second-order techniques is lacking, hindering a complete understanding of its advantages over alternative methodologies. AdaHessian [16] innovatively utilizes only the diagonal elements of the Hessian matrix, reducing memory requirements significantly. However, the trade-offs between computational efficiency and optimization accuracy in complex, highly non-linear optimization landscapes warrant deeper analysis to comprehend its viability across

979-8-3503-6013-4/24/$31.00 ©2024 IEEE

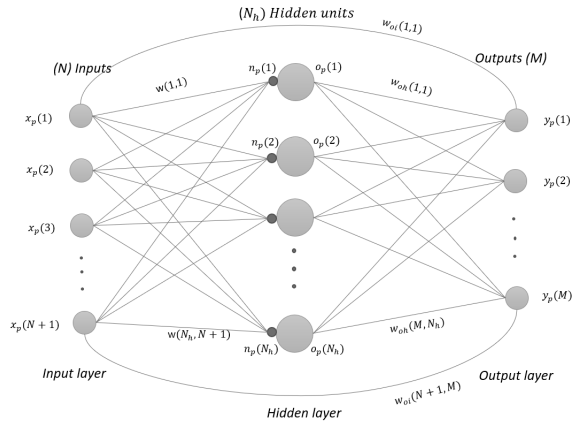

Fig. 1: Diagram of a Fully Connected Multilayer Perceptron

diverse neural network architectures and training scenarios. The algorithm proposed in this paper was originally explored in our thesis work [17]. The proposed algorithm addresses this challenge by optimizing neural network training operations more efficiently. The methodology involves segmenting the input weights of a MLP into clusters, followed by applying Newton's method to compute gain factors for each cluster.

II. Background

A. Structure and Notation

The depicted figure 1 shows the configuration of a fully connected, forward-propagating multilayer perceptron (MLP). In this setup, the weights denoted as $w(k,n)$ form the connections from each n^{th} input node to the k^{th} node in the hidden layer. The weights marked as $w_{oh}(m,k)$ are responsible for linking the k^{th} hidden node's non-linear output, symbolized as $O_p(k)$, to the m^{th} output node $y_p(m)$, which is activated linearly. Additionally, direct connections from the n^{th} input to the m^{th} output are facilitated by $w_{oi}(m,n)$, known as bypass weights. The training data consists of a set of independent and identically distributed pairs of inputs and targets, $\{\mathbf{x_p}, \mathbf{t_p}\}$, with the input vectors $\mathbf{x_p}$ having a dimension of N and target vectors $\mathbf{t_p}$ a dimension of M. The index p ranges from 1 to N_v, the total number of training vectors. The hidden layer comprises N_h nodes. To incorporate input bias, an extra element $x_p(N+1)$, set to 1, is added to each input unit. The net function vector of the hidden layer for any training pattern p is denoted as $\mathbf{n_p}$, and is calculated by $\mathbf{n_p} = \mathbf{W} \cdot \mathbf{x_p}$. The activation for the k^{th} element in the hidden layer activation vector $\mathbf{O_p}$ is given by $O_p(k) = f(n_p(k))$, with $f(\cdot)$ being the sigmoid function. The network's output vector $\mathbf{y_p}$ is given by:

$$\mathbf{y_p} = \mathbf{W_{oi}} \cdot \mathbf{x_p} + \mathbf{W_{oh}} \cdot \mathbf{O_p} \tag{1}$$

The equation above can be alternatively expressed as $\mathbf{y_p} = \mathbf{W_o} \cdot \mathbf{X_a}$, where $\mathbf{X_a}$, the augmented input column vector, incorporates N_u basis functions and is defined as $[\mathbf{x_p}^T : \mathbf{O_p}^T]^T$. Here, N_u equals $1 + N + N_h$. The augmented weight matrix $\mathbf{W_o}$, of dimension $M \times N_u$, is composed of $\mathbf{W_o} = [\mathbf{W_{oh}}, : \mathbf{W_{oi}}]$. Training an MLP involves minimizing the mean squared error (2) between the desired output and the actual output of the network. This process adjusts the network weights and biases to align the calculated output closely with the desired output. The training is framed as an optimization task within a structural risk minimization context [18], [19]. This context focuses on minimizing the loss function E, which is the mean square error (MSE), serving as a surrogate for the non-smooth classification loss. The MSE is defined as:

$$E = \frac{1}{N_v} \sum_{p=1}^{N_v} \sum_{i=1}^{M} [\mathbf{t}_p - \mathbf{y}_p]^2 \tag{2}$$

The nonlinearity within $\mathbf{y_p}$ introduces non-convexity into the loss function E, potentially leading to local minima in practice. We assume $\mathbf{t_p}$ follows a Gaussian distribution concerning input $\mathbf{x_p}$. Our aim is to determine the optimal weights within the MLP structure. Employing the empirical risk minimization framework [20], we design learning algorithms, benefiting from the advantage of transforming MLP training into an optimization problem. This conversion allows us to leverage various optimization algorithms to enhance MLP learning processes.

B. MLP Initialization

As per [20], the input weights matrix $\mathbf{W}$ gets randomly initialized with zero mean and unit standard deviation. For initializing the output weight matrix $\mathbf{W}_{\mathrm{o}}$, the approach involves using Target Weight Refinement (TWR) [20]. The TWR approach seeks to reduce the loss function outlined in equation (2) by focusing on the optimization of $\mathbf{W}_{\mathrm{o}}$. This is achieved by resolving M distinct sets of equations, each set containing N_u equations with N_u unknown variables, as given by:

$$\mathbf{C} = \mathbf{R} \cdot \mathbf{W_o^T} \tag{3}$$

Where $\mathbf{C}$ is the cross-correlation matrix and is defined as $\frac{1}{N_v} \sum_{p=1}^{N_v} \mathbf{t_p} \cdot \mathbf{X_a^T}$ and $\mathbf{R}$ is the auto-correlation matrix and is defined as $\frac{1}{N_v} \sum_{p=1}^{N_v} \mathbf{X_a} \cdot \mathbf{X_a^T}$. In terms of optimization theory, solving equation (3) essentially equates to Newton's algorithm for the output weights [21]. Once $\mathbf{W}$, $\mathbf{W}_{\mathrm{oi}}$, and $\mathbf{W}_{\mathrm{oh}}$ are initialized, a two-stage algorithm is applied, in which we update $\mathbf{W}$ first and then perform TWR to adjust $\mathbf{W}_{\mathrm{o}}$. The MLP network is now ready for training using gradient and hessian based algorithms. Training an MLP employs gradient based methods like back-propagation (BP), conjugate gradient (CG), or hessian based methods such as Levenberg-Marquardt (LM) and Newton's method.

C. Optimization algorithms

In our study, we'll compare our work with scaled conjugate gradient, an optimization algorithms positioned between first and second-order optimization techniques. Alongside these, we briefly review two second order algorithms, LM [22] and Target Weight Refinement-Multiple Gain Adaptation (TWR-MGA) [23].

1) Scaled conjugate Gradient: The Conjugate Gradient algorithm, introduced in [24], executes a line-search within conjugate directions and typically demonstrates faster convergence compared to the back-propagation algorithm. Despite being a general unconstrained optimization technique, the scaled conjugate gradient (SCG) method has been extensively detailed in [25] for efficiently training an MLP. In the context of training an MLP using the conjugate gradient algorithm, a direction vector is derived from the gradient $\mathbf{g}$, denoted as $\mathbf{p} \leftarrow -\mathbf{g} + B_1 \cdot \mathbf{p}$. Here, $\mathbf{p}$ represents $\text{vec}(\mathbf{P}, \mathbf{P_{oh}}, \mathbf{P_{oi}})$, wherein $\mathbf{P}$, $\mathbf{P_{oi}}$, and $\mathbf{P_{oh}}$ are the direction vectors. B_1 signifies the ratio of gradient entopy between current and previous training iterations. All the weights in the network are updated in each training iteration as, $\mathbf{w} \leftarrow \mathbf{w} + z \cdot \mathbf{p}$. For a comprehensive pseudocode, please refer to [20].

2) Levenberg-Marquardt algorithm: The LM (Levenberg-Marquardt) algorithm addresses the sub-optimality often encountered in Newton's method due to the singularity of $\mathbf{H}$ (the Hessian matrix). LM [26], tries to address the problems with Newton's method by modifying the Hessian matrix as $\mathbf{H_{LM}} = \mathbf{H} + \lambda \cdot \mathbf{I}$, where $\mathbf{I}$ is the identity matrix matching the dimensions of $\mathbf{H}$ and λ serves as a regularization parameter. This adjustment ensures that the combined matrix $(\mathbf{H} + \lambda \cdot \mathbf{I})$ remains positive definite, ensuring better conditioning throughout computations. The calculation of the second-order direction, denoted as $\mathbf{d}$, mirrors that of Newton's method, formulated as $\mathbf{H_{LM}} \cdot \mathbf{d} = \mathbf{g}$, where $\mathbf{g}$ represents the gradient. Once $\mathbf{H_{LM}}$ is determined, the model weights undergo an update. The regularization parameter λ plays a significant role in influencing the behavior and performance of the LM algorithm. For an in-depth exploration of the LM algorithm, you can refer to [20].

3) TWR-MGA: An alternative to LM, the TWR-MGA algorithm adopts a two-stage "layer by layer" approach, leveraging Target Weight Refinement (TWR) and multiple gain adaptations (MGA) per hidden unit [23]. Unlike heuristic methods relying on various learning rates or momentum terms [27], [28], TWR-MGA targets enhanced learning speed and convergence.This algorithm handles input weight updates using negative gradients and N_h gain factors that are computed in every training iteration using Newton's method [22]. Meanwhile, output weights are trained using TWR.

D. Challenges and Objectives

The paper addresses the following challenges posed by second order algorithms. Algorithms such as Lavenberg-Marquardt incur significant computational costs due to their reliance on optimizing all weights utilizing second order information. In contrast, algorithms like TWR-MGA, Shampoo, and Ada-Hessian attempt to mitigate these computational demands by either optimizing a subset of parameters via second order methods or by employing a lower rank approximation of the Hessian matrix. This approach reduces the need to compute and invert the full Hessian matrix for all parameters. However, a common limitation within these algorithms, when applied to a specific model architecture, is their consistent optimization of a similar fraction of parameters using second order methods for every training iteration for a given dataset. This uniform strategy leads to an unnecessary escalation in computational requirements. The objective of this paper is to formulate a novel second order training algorithm that dynamically adjusts the number of parameters optimized using second order information. This adjustment will be specific to each training iteration, dependent on the dataset and model architecture involved. The goal is to enhance computational efficiency without increase in optimization time. In order to address the objectives, we introduce Adapt-MGA, a second-order algorithm that involves creating weight clusters for each hidden unit and computing gain factors for each cluster.

III. ADAPT-MGA

The TWR-Newton algorithm is known for its impressive loss convergence properties, but its effectiveness reduces when the inputs have linear dependencies. Also, its performance drops when the loss curve is not quadratic. Moreover, in terms of computational demands, TWR-Newton is computationally intensive when compared to first-order training algorithms. Conversely, the TWR-MGA algorithm may not match TWR-Newton's in terms of optimization efficiency, yet it shows better resilience in cases where inputs have linear dependencies. It is also less computationally demanding, as detailed in [29]. The Adaptive MGA algorithm is designed to take advantage of the strengths of both TWR-MGA and TWR-Newton while addressing their limitations by dynamically changing it behavior between the TWR-MGA and TWR-Newton methods in every training iteration. In TWR-MGA, N_h gain factors are determined, but in the Adaptive MGA algorithm, the count of gain factors calculated can range from N_h to $N_h \cdot (N+1)$ in each iteration, allowing for a versatile approach that leverages the advantages of both techniques and adapts according to the specific problem.

A. Input weight clusters

In the described algorithm, a "cluster" denotes a cohort of input weights that are all updated with a single gain factor. In methods like steepest gradient descent, a single gain factor is used to update all the weights, therefore we can say that all the wights belong to a single cluster. In TWR-MGA, N_h gain factors are used where N_h is the number of hidden units. All the weights incident to a hidden unit are updated with single gain factor. There are N_h total clusters and each cluster is of size $N+1$. However, in the adaptive MGA algorithm the number of clusters is not fixed. The number of weights in a cluster can fluctuate between $N+1$ and 1, while the number of weight clusters per hidden unit varies from 1 to $N+1$. Weights incident to a hidden unit are clustered based on the curvature of the loss function with respect to the weight. The curvature of loss function is the second partial derivative of loss function with respect to the weight and is computed as follows.

$$l(k,n) = \frac{\partial^2 E}{\partial w(k,n)^2} = \frac{2}{N_v}\sum_{i=1}^{M} w_{oh}(i,k)^2 \sum_{p=1}^{N_v} f'(n_p(k))^2 x_p(n)^2 \tag{4}$$

The elements of the the curvature matrix correspond to the diagonal elements of Hessian matrix. Weights incident to a hidden unit are sorted based on the curvature and are segmented into N_g equal sized clusters.

B. Adapting the number of clusters

The main objective of the Adapt-MGA algorithm is to utilize the floating point operations (FLOPs) efficiently by changing the number of weight clusters associated with each hidden unit, thereby changing the number gain factors computed using Newton's method. The proposed algorithm achieves this goal by increasing the number of clusters associated with each hidden unit thereby increasing the number of gain factor computed when there is an increase in loss change per FLOP. The algorithm does the opposite i.e decrease the number of gain factors computed when there is drop in loss change per FLOP. Loss change per FLOP is computed as follows,

$$LPF(i_t) = \frac{L(i_t - 1) - L(i_t)}{FLOPs(i_t)} \tag{5}$$

where, $FLOPs(i_t)$ is the count of floating point operations in training iteration i_t, $LPF(i_t)$ is the loss change per FLOP in iteration i_t, $L(i_t - 1)$ is the loss in iteration $i_t - 1$.
From our experiments, we observed that the Adapt-MGA algorithm usually operates close to the TWR-Newton algorithm and achieves large drop in the loss function and as the loss converges to local minima the algorithm adapts to operate close to the TWR-MGA to improve computational efficiency.

C. Adapt-MGA Initialization

We can initialize the number of gain factors per hidden unit to any value between 1 to $N + 1$. From our experiments we observed that the following initialization technique works best. In the first iteration randomly select 5 values between 1 and $N + 1$. Initialize the number of grain factors per hidden unit to the value that gives the lowest loss. This methodology significantly enhances the algorithm's performance compared to the random initialization of the number of gain factors per hidden unit. To further boost performance, this technique can be applied periodically, say once every 50 epochs during training. However, computing loss for different number of gain factors per hidden unit in the same iteration is computationally demanding. To alleviate this, we can take advantage of the fact that the Hessian $\mathbf{H_{Amga}}$ and gradients $\mathbf{g_{Amga}}$ of gain factors with any cluster size can be derived from the Hessian and gradients of gain factors with cluster size $N + 1$, represented as $\mathbf{H}$ and $\mathbf{g}$, in the equation below. This approach reduces the computational load involved in computing loss for gain factors with different cluster sizes in the same training iteration.

$$g_{\text{Amga}}(k,c) = \sum_{a \in c} g(k, i_k(a))^2 \tag{6}$$

$$h_{\text{Amga}}(k,c_1,j,c_2) = \sum_{a \in c_1}\sum_{b \in c_2} h(k,i_k(a),j,i_j(b)) gg(k,c_1,j,c_2)$$
$$gg(k,c_1,j,c_2) = g(k,i_k(a))g(j,i_j(a)) \tag{7}$$

D. Mathematical Treatment

Lemma 1:
Context: Consider a quadratic loss function $E(w)$ with respect to an input weight vector $\mathbf{w}$. Divide $\mathbf{w}$ into k clusters $\mathbf{w}_k$, so $\mathbf{w} = [\mathbf{w}_1^T, \mathbf{w}_2^T, \ldots, \mathbf{w}_k^T]^T$. The gradient of the loss function relative to each cluster is $g_k = \frac{\partial E}{\partial w_k}$. Assume a vector $\mathbf{z}$ that includes gain factors for each of the k clusters.
The updated loss function using the k gain factors and gradients is $E_k = E(\mathbf{w}_1 + z_1\mathbf{g}_1, \mathbf{w}_2 + z_2\mathbf{g}_2, \ldots, \mathbf{w}_k + z_k\mathbf{g}_k)$. By increasing the number of clusters from k to $k + 1$, splitting one of the existing clusters, it holds that $E_{k+1} \leq E_k$.
Proof: The updated loss $E(\mathbf{w})$ after adjusting input weights is represented as,

$$E(\mathbf{w} + \mathbf{e}) = E(\mathbf{w}) - \mathbf{e}^T\mathbf{g} + \frac{1}{2}\mathbf{e}^T\mathbf{H}\mathbf{e} \tag{8}$$

where $\mathbf{e}$ denotes the input weight modification vector, $\mathbf{g}$ is equivalent to $\mathbf{g_k}$ for k=1, and $\mathbf{H}$ represents the Hessian matrix. Applying Newton's method for calculating the weight modification vector $\mathbf{e}$, we obtain:

$$\mathbf{e} = \mathbf{H}^{-1} \cdot \mathbf{g} \tag{9}$$

The weight modification vector for k clusters is:

$$\mathbf{e}_k = [z_1\mathbf{g}_1^T, z_2\mathbf{g}_2^T, \ldots, z_k\mathbf{g}_k^T]^T \tag{10}$$

Given $\mathbf{z} = argmin_z(E(\mathbf{w} + \mathbf{e}_k))$, when k is incremented by one, we have:

$$\mathbf{e}_{k+1} = [z_1\mathbf{g}_1^T, z_2\mathbf{g}_2^T, \ldots, z_{ka}\mathbf{g}_{ka}^T, z_{kb}\mathbf{g}_{kb}^T]^T \tag{11}$$

If $z_{ka} = z_{kb} = z_k$, then $\mathbf{e}_k = \mathbf{e}_{k+1}$ and $E_{k+1} = E_k$. Since all elements in $\mathbf{z}$ can be adjusted, it leads to $E_{k+1} \leq E_k$. Lemma 1 thereby demonstrates that subdividing the input weight clusters results in a reduction in loss.

Lemma 2: The decrease in loss from TWR-MGA in any iteration is equal to or less than the decrease in loss from adaptive MGA: $E - E_{mga} \leq E - E_{Amga}$.
Proof: This conclusion is derived from Lemma 1, as the weight clusters in adaptive MGA are finer divisions of those in TWR-MGA.

Lemma 3: TWR-Newton is an extreme form of the Adapt-MGA algorithm when the k clusters of adaptive MGA are divided until $k = N_h \cdot (N + 1)$.
Proof: The equation for $\mathbf{e_{Newton}}$ is as follows:

$$\mathbf{e_{Newton}} = \begin{bmatrix} z_1 \cdot \mathbf{g}_1 \\ z_2 \cdot \mathbf{g}_2 \\ \vdots \\ z_{N_h(N_g+1)} \cdot \mathbf{g}_{N_h(N_g+1)} \end{bmatrix} \tag{12}$$

In Newton's method for each iteration, the relationship between the resulting losses is $E_{\text{Newton}} \leq E_{\text{Amga}}$. This lemma is a direct implication of Lemma 1.

The above lemmas collectively illustrate that the proposed algorithm interpolates between TWR-MGA and TWR-Newton.
The pseudo-code for Adapt-MGA is outlined in Algorithm 1.

Algorithm 1 Adapt-MGA algorithm

0: Read the training data.
0: Initialize $\mathbf{W}$, $\mathbf{W_{oi}}$, $\mathbf{W_{oh}}$, N_{it} , $N_g \leftarrow N_h$, $i_t \leftarrow 0$,
0: **while** $i_t < N_{it}$ **do**
0: Compute $\mathbf{G}$
0: **MGA Adapt steps** :
0: a: Compute $\mathbf{L}$ (the curvature of loss w.r.t input weights) from equation (4). $size(\mathbf{L}) = (N_h, N+1)$
0: b: $\mathbf{I} \leftarrow argsort(\mathbf{L}, axis = 1)$. Input weight indices sorted in the descending order of curvature. $size(\mathbf{I}) = (N_h, N+1)$
0: c: Divide the sorted indices of weights connected to each hidden unit into N_g clusters of equal size.
0: d: Compute Adaptive MGA gain factors $\mathbf{z_{Amga}}$ using $\mathbf{H_{Amga}}$ and $\mathbf{g_{Amga}}$. $size(\mathbf{z_{Amga}}) = (N_h \cdot N_g, 1)$
0: e: Create gain factor column vector $\mathbf{z}$ by applying same gain factors for all the weights in a cluster. $size(\mathbf{z}) = (N_h, N+1)$
0: Update input weights: $\mathbf{W} \leftarrow \mathbf{W} + \mathbf{z} \odot \mathbf{G}$
0: **TWR step** : Solve equation (3) to obtain $\mathbf{W_o}$
0: Compute loss change per FLOP LPF_{i_t} from equation (5)
0: **if** $LPF_{i_t} > LPF_{i_t-1}$ **then**
0: $N_g \leftarrow N_g + 1$
0: **else**
0: $N_g \leftarrow N_g - 1$
0: **end if**
0: $i_t \leftarrow i_t + 1$
0: **end while**=0

IV. Experimental Results

For algorithms like LM and SCG, all weights are updated using the respective algorithms in each iteration. Conversely, in TWR-MGA and adaptive MGA algorithms, the input weights are updated using the respective alogorithms, followed by solving linear equations for the output weights. We compared the performance of the aforementioned algorithms on datasets from [30]. The inputs in the datasets are normalized before training. Network pruning is used to determine the hidden units size as outlined in [31]. Training is conducted on the complete dataset ten times using ten distinct initial networks. The resulting average Mean Squared Error (MSE) from this ten-fold training is visualized in the subsequent plots.

Additionally, training loss and the number of floating point operations are computed in each iteration for a given dataset and training algorithm. The following plots show the effectiveness of adaptive MGA algorithm when compared to other training algorithms.

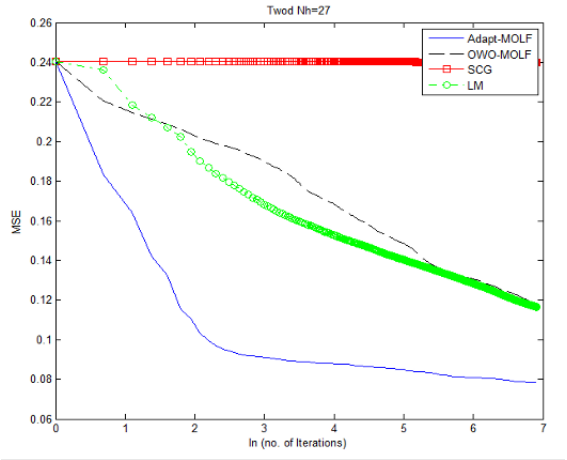

Fig. 2: Twod.tra : MSE vs epochs

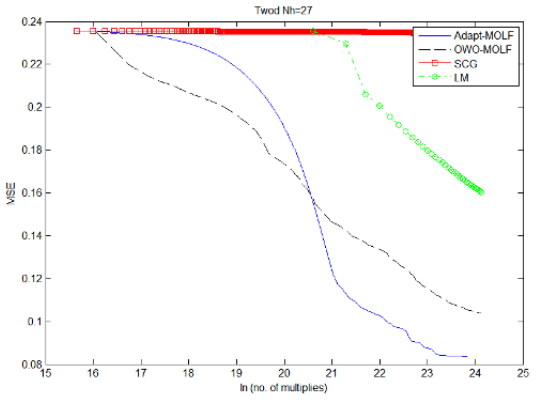

Fig. 3: Twod.tra data set: MSE vs multipliers

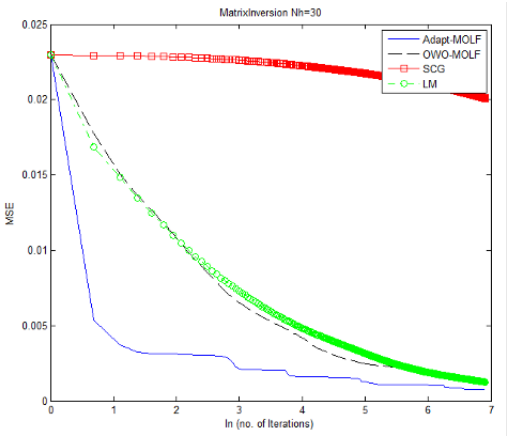

Fig. 8: Matrix inversion : MSE vs epochs

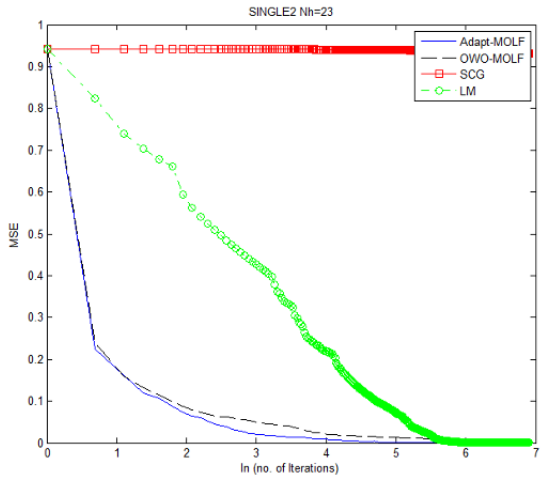

Fig. 4: Single2.tra : MSE vs epochs

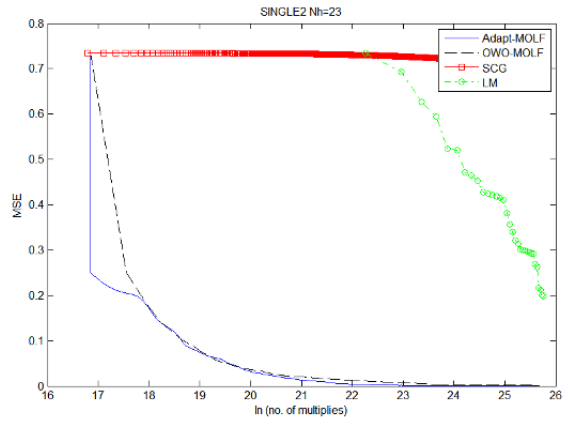

Fig. 5: Single2.tra data set: MSE vs multipliers.

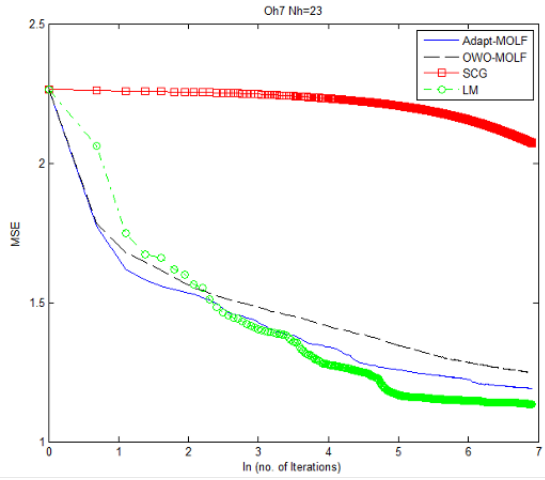

Fig. 6: Oh7.tra : MSE vs epochs

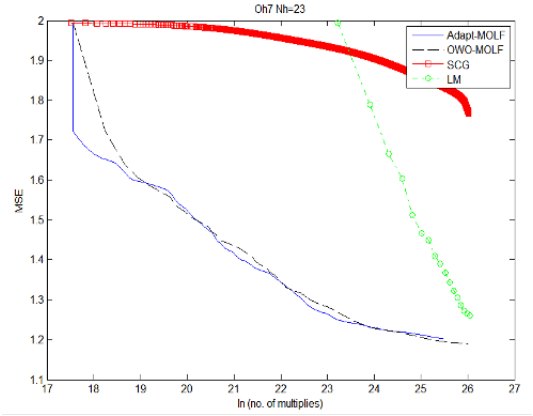

Fig. 7: Oh7.tra data set: MSE vs multipliers.

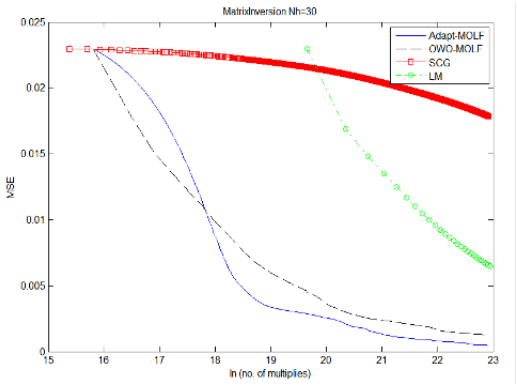

Fig. 9: Matrix inversion data set: MSE vs multipliers

A. *Results*

We employed k-fold cross validation to determine the average loss for both training and testing. The comparative data in Table 2 below showcases the average training and testing loss obtained using the adaptive MGA algorithm as compared to other algorithms across different data sets.

TABLE I: Train and test loss with k-folds (k=10)

Data Set	Adaptive MGA	TWR-MGA	SCG	LM
Twod.tra Etrn	0.0888	0.1554	1.0985	0.2038
Twod.tra Etst	0.1172	0.1731	1.0945	0.2205
Single2.tra Etrn	0.0042	0.0151	3.5719	0.0083
Single2.tra Etst	0.0179	0.1689	3.6418	0.0178
Mattrn.tra Etrn	0.0011	0.0027	4.2400	0.0022
Mattrn.tra Etst	0.0013	0.0032	4.3359	0.0027
Oh7.tra Etrn	1.2507	1.3205	4.1500	1.1602
Oh7.tra Etst	1.4738	1.4875	4.1991	1.4373

The observations from the plots and the provided Table suggest that the adaptive MGA algorithm tends to outperform TWR-MGA, LM, and SCG algorithms across most datasets, both in terms of number of iterations and FLOPs.

V. Conclusion and Future Work

The Adapt-MGA algorithm presented in this research successfully mitigates the scalability challenges inherent in

second-order training algorithms. This is achieved through an adaptive mechanism that modulates the computation of gain factors using Newton's method, thereby enhancing the efficiency of loss reduction per FLOP. Empirical assessments show that the Adapt-MGA algorithm outperforms the TWR-MGA algorithm, demonstrating greater efficiency in reducing loss per training iteration and frequently achieving significant drop in loss per FLOP. Furthermore, the algorithm demonstrates a unique capacity to interpolate between the TWR-MGA and TWR-Newton methodologies. The scope of this study is intentionally focused, applying the concept of Adapt-MGA exclusively to input weights in a MLP. This concentration allows for a detailed exploration and elucidation of the algorithm's derivation and operational specifics. Looking forward, subsequent research will expand the application of the Adapt-MGA algorithm to more complex deep neural network architectures. This will facilitate a comprehensive comparison of its performance against established first-order neural network optimization techniques. This forthcoming analysis is anticipated to provide further insights into the efficacy and applicability of the Adapt-MGA algorithm within the broader context of neural network optimization.

REFERENCES

[1] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1989.

[2] D. W. Ruck, S. K. Rogers, M. Kabbisky, M. E. Oxley, and B. W. Suter. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *IEEE Transactions on Neural Networks*, 1(4), 1990.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Matthew D. Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[6] Aleksandar Botev, Guy Lever, and David Barber. Nesterov's accelerated gradient and momentum as approximations to regularised update descent. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 1899–1903. IEEE, 2017.

[7] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012.

[8] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[9] Jun Lu. Adasmooth: An adaptive learning rate method based on effective ratio. In *Sentiment Analysis and Deep Learning: Proceedings of ICSADL 2022*, pages 273–293, Singapore, 2023. Springer Nature Singapore.

[10] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[12] Hang Xu, Wenxuan Zhang, Jiawei Fei, Yuzhe Wu, TingWen Xie, Jun Huang, Yuchen Xie, Mohamed Elhoseiny, and Panos Kalnis. Slamb: accelerated large batch training with sparse communication. In *International Conference on Machine Learning*, pages 38801–38825. PMLR, 2023.

[13] Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, and et al. Symbolic discovery of optimization algorithms. *arXiv preprint arXiv:2302.06675*, 2023.

[14] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[15] Rohan Anil, Vineet Gupta, Tomer Koren, Kevin Regan, and Yoram Singer. Scalable second order optimization for deep learning. *arXiv preprint arXiv:2002.09018*, 2020.

[16] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael Mahoney. Adahessian: An adaptive second order optimizer for machine learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 10665–10673, 2021.

[17] Jeshwanth Challagundla. Adaptive multiple optimal learning factors for neural network training. *M.S. Thesis/Dissertation https://rc.library.uta.edu/uta-ir/handle/10106/25030*, 1(1):30–49, 2015.

[18] Christopher M Bishop. Pattern recognition. *Machine Learning*, 128, 2006.

[19] Hugo Larochelle, Dumitru Erhan, Aaron Courville, James Bergstra, and Yoshua Bengio. An empirical evaluation of deep architectures on problems with many factors of variation. In *Proceedings of the 24th international conference on Machine learning*, pages 473–480. ACM, 2007.

[20] Kanishka Tyagi, Chinmay Rane, and Michael Manry. Supervised learning. In *Artificial Intelligence and Machine Learning for EDGE Computing*, pages 3–22. Elsevier, 2022.

[21] Melvin Deloyd Robinson and Michael Thomas Manry. Two-stage second order training in feedforward neural networks. In *FLAIRS Conference*, 2013.

[22] Kanishka Tyagi, Son Nguyen, Rohit Rawat, and Michael Manry. Second order training and sizing for the multilayer perceptron. *Neural Processing Letters*, 51(1):963–991, 2020.

[23] Rohit Rawat, Jignesh K Patel, and Michael T Manry. Minimizing validation error with respect to network size and number of training epochs. In *The 2013 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2013.

[24] Magnus Rudolph Hestenes and Eduard Stiefel. *Methods of conjugate gradients for solving linear systems*, volume 49. NBS, 1952.

[25] Christakis Charalambous. Conjugate gradient algorithm for efficient training of artificial neural networks. *IEE Proceedings G-Circuits, Devices and Systems*, 139(3):301–310, 1992.

[26] Kenneth Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly of applied mathematics*, 2(2):164–168, 1944.

[27] Robert A Jacobs. Increased rates of convergence through learning rate adaptation. *Neural networks*, 1(4):295–307, 1988.

[28] Simon Haykin. *Neural networks and learning machines*, volume 3. Pearson Education, 2009.

[29] Praveen Jesudhas, Michael T. Manry, Rohith Rawat, and Sanjeev Malalur. Analysis and improvement of multiple optimal learning factors for feed-forward networks. In *The 2011 International Joint Conference on Neural Networks*, San Jose, CA, 2011.

[30] Classification data files. Image Processing and Neural Networks Lab, The University of Texas Arlington. https://ipnnl.uta.edu/training-data-files/regression/, 2022. Image Processing and Neural Networks Lab, The University of Texas Arlington.

[31] S. S. Malalur, M. T. Manry, and P. Jesudhas. Multiple optimal learning factors for the multilayer perceptron. *Neurocomputing*, 149:1490–1501.