MBAG: A Scalable Mini-Block Adaptive Gradient Method for Deep Neural Networks

Jaewoo Lee
Department of Computer Science
University of Georgia
Athens, USA
jwlee@cs.uga.edu

***Abstract*—Preconditioning is a technique widely used to accelerate the convergence of optimization algorithms. Recently proposed efficient second-order algorithms (such as KFAC) showed that preconditioning the gradient using the curvature information of loss function can help achieve faster convergence. However, their practicality in large-scale deep learning is still limited due to the high computational and storage cost. In this work, we propose a stochastic adaptive gradient algorithm, called Mini-Block Adaptive Gradient (MBAG), that addresses those computational challenges in computing the preconditioning matrix. To reduce the per-iteration cost, MBAG analytically computes the inverse of preconditioning matrix using the matrix inversion lemma and then approximately finds its square root using an iterative solver. Further, to mitigate the storage requirement, MBAG partitions model parameters into subsets of small size and only computes sub-blocks of preconditioner associated with each subset of parameters. This greatly improves the scalability of the proposed algorithm. The performance of MBAG is compared to that of popular first- and second-order algorithms on auto-encoder and classification tasks using real datasets.**

***Index Terms*—Stochastic gradient descent, preconditioned gradient, second order optimizer, block-diagonal approximation**

I. Introduction

Stochastic gradient-based optimization methods, such as stochastic gradient descent (SGD) [1], are most widely used for training deep neural networks (NNs) on large-scale datasets due to their computational efficiency. There has been significant effort to improve the convergence rate of first order gradient-based methods by incorporating momentum [2], [3], adaptive learning rates [4]–[6], and variance reduction [7]. Despite the effort, due to the high-dimensional and highly nonconvex nature of NN training, their convergence in practice can still be slow, especially when the problem is heavily ill-conditioned. A typical approach to improving the convergence rate is to change the geometry of parameter space by preconditioning the SGD update with a matrix $\mathbf{G}_t$:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{G}_t^{-1} \nabla \mathcal{L}(\boldsymbol{\theta}_t)\,, \tag{1}$$

where $\boldsymbol{\theta}_t \in \mathbb{R}^d$ is the model parameter vector at iteration t, $\eta_t > 0$ is a step size, $\nabla \mathcal{L}(\boldsymbol{\theta}_t)$ denotes the gradient of empirical loss function evaluated at $\boldsymbol{\theta}_t$, and $\mathbf{G}_t \in \mathbb{R}^{d \times d}$ is a matrix called *preconditioner*. Typically, $\mathbf{G}$ is set to a matrix containing the curvature information of loss function.

This research was supported by the National Science Foundation under Grant No. 1943046.

Existing gradient-based methods can be viewed as a preconditioned SGD. The first order adaptive methods (such as Adam [5] and AdaGrad [4]) restrict $\mathbf{G}$ to be a diagonal matrix. This greatly reduces the computational complexity associated with computing and inverting $\mathbf{G}$ but at the same time limits the amount of curvature information they can extract. There has been recent interest in developing efficient second order methods for deep NNs, e.g., KFAC [8], Shampoo [9], and K-BFGS [10]. These methods set the preconditioner $\mathbf{G}$ to be an approximate Hessian [10], [11], generalized Gauss-Newton matrix [12], or FIM [8], [9]. However, the use of second order information in the context of deep learning poses many computational challenges. First, exactly computing these matrices is infeasible as it requires prohibitive amount of computation and memory space. For example, computing the inverse of $\mathbf{G}$ in general has $\mathcal{O}(d^3)$ time complexity, where d is the number of parameters which can be extremely large for modern deep NN models. Even storing $\mathbf{G}$ requires $\mathcal{O}(d^2)$ memory space. Hence, some sort of approximations need to be applied. Based on the empirical observation that curvature matrices are diagonally dominant, existing second order methods typically use layer-wise block-diagonal approximation (BDA) to the curvature matrix, i.e., they assume parameters belonging to different layers are independent and only compute sub-blocks of $\mathbf{G}$ along the diagonal. Second, most second order methods require performing computationally expensive matrix inversion. Although many practical methods only perform the inversion periodically to amortize the cost over several iterations, it adds another layer of approximation.

In this paper, we propose an adaptive preconditioned SGD algorithm, called Mini-Block Adaptive Gradient (MBAG), that efficiently utilizes the curvature information yet can easily scale to modern large-scale network architectures. To address the computational challenges in approximating curvature information, MBAG employs three innovative techniques. First, MBAG extends the block-diagonal approximation and further approximates each diagonal block as a block diagonal matrix consisting of smaller matrices (which we call *mini-blocks*). Specifically, it groups the parameters of a layer and computes the curvature matrix for each group assuming the independence between parameters of different groups. Second, inspired by the observation that the curvature matrix update can be viewed as a sequence of rank-one updates to an identity

matrix, MBAG directly updates and exactly computes the inverse of $\mathbf{G}$ using the Sherman-Morrison formula [13]. This eliminates the need for costly matrix inversion. Third, while the curvature matrix used in MBAG has some superficial resemblance to empirical FIM, our method is not motivated by the natural gradient [14] but rather by AdaGrad in which $\mathbf{G}$ is set to $\mathbf{G} = \delta \mathbf{I}_d + \left(\sum_{i=1}^{t} \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i) \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_i)^\intercal \right)^{\frac{1}{2}}$, where $\delta \geq 0$ is a small constant. Existing algorithms typically compute the above square root using the singular value decomposition (SVD) or eigenvalue decomposition whose time complexity is $\mathcal{O}(d^3)$. To mitigate the computational cost, we approximately compute the square root $\mathbf{G}^{1/2}$ by running few iterations of Newton-Schulz iterations [15] and show that it is sufficient for our purpose. The Newton-Schulz iterations only involve easily parallelizable matrix-matrix multiplications and are order-of-magnitude faster than SVD-based approach. In sum, our main contributions are as follows:

- We propose a stochastic adaptive gradient algorithm for large-scale deep NNs that efficiently approximates full-matrix AdaGrad preconditioner using mini-blocks. It achieves fast convergence by benefiting from the use of second order information while being as scalable as first order algorithms.
- We analyzes the convergence property of the proposed algorithm and show that it converges to a stationary point.
- We perform extensive experiments on real datasets against other recently proposed first and second order algorithms. We empirically show that MBAG enjoys fast convergence on various datasets.

II. Related Work

Many recent works (including ours) generalized this idea by considering sub-blocks of smaller sizes [16]–[20]. The works closest to ours are those of [18] and [19]. Bahamou et al. [18] proposed an adaptive natural gradient method called MBF. For a fully connected layer l with weight matrix $\mathbf{W}^l \in \mathbb{R}^{m \times n}$, MBF creates a parameter group for each row $i \in [m]$ in $\mathbf{W}$, i.e., decomposing $\mathbf{G}^l$ into m matrices of size $n \times n$. However, for many modern network architectures (e.g., transformers and autoencoders), this strategy becomes impractical and incurs prohibitive computational cost as n could be much larger than m. To address this issue, they proposed to take a single matrix by taking an average over m sub-block matrices, which adds another layer of approximation. In contrast, our algorithm allows groups of arbitrary size. Specifically, MBAG splits the parameters in $\mathbf{W}$ into groups of fixed size τ, where τ is a hyperparameter whose value can be chosen according to the available memory size. AdaBlock [19] is a BDA-based approximation of full-matrix version of Adam algorithm. Similar to our algorithm, it also allows an arbitrary grouping of parameters, but it partitions $\mathbf{W}$ column-wise while our algorithm partitions row-wise. In addition, our algorithm is different from AdaBlock and MBF in how it computes the inverse of curvature matrix. Both AdaBlock and MBF maintain a moving average of $\mathbf{G}_t$ and update it whenever a new mini-batch gradient is received. To compute $\mathbf{G}_t^{-1}$, they perform expensive eigenvalue decomposition (or SVD). Contrastingly, MBAG directly models $\mathbf{G}_t^{-1}$ and updates it exactly using the Sherman-Morrison formula. Many prior works have utilized the idea of using the Sherman-Morrison formula for approximating the inverse of Hessian (either explicitly or implicitly) in the context of BFGS optimization [21], natural gradient descent [22], [23], online convex optimization [24], and network pruning [25]. To our best knowledge, our work is the first to combine the matrix inversion lemma with fine-grained block diagonal approximation in the context of large-scale deep learning to improve the accuracy and efficiency of curvature approximation.

MBAG can be viewed as an approximation to full-matrix version of AdaGrad, and there has been several attempts to scale up the block-diagonal approximation of AdaGrad preconditioner. Shampoo [9] approximates the preconditioning matrix as a Kronecker product of small matrices, each of which working on a single dimension. However, their algorithm requires knowledge of model structure. Radagrad [26] approximates the preconditioning matrix in a lower dimensional space using random projections. However, the performance of reconstructed preconditioner can be sensitive to noise and hence is not well suited for stochastic optimization. GGT [27] approximates the inverse square root of preconditioning matrix from a sliding window of gradient history. However, it requires storing r copies of gradients in memory and computing the eigenvalue decomposition of $r \times r$ matrix at every iteration, where r in their experiments is as large as several hundred.

III. Preliminaries

A. Notations

We consider a neural network $f_{\boldsymbol{\theta}}$ with L layers. Each layer $l \in [L]$ has a weight matrix $\mathbf{W}^l \in \mathbb{R}^{m_l \times n_l}$, where $[L] = \{1, 2, \ldots, L\}$ denotes the set of integers between 1 and L. When it's clear from the context, we omit the the layer index l in the superscript to avoid cluttered notation. To construct a column vector from matrices (or tensors in general), we use the vectorization operator. For a matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$, $\mathrm{vec}(\mathbf{W}) \in \mathbb{R}^{mn}$ transforms $\mathbf{W}$ into a column vector by stacking its column vectors below one another. The parameters of $f_{\boldsymbol{\theta}}$ are denoted by $\boldsymbol{\theta} = \left((\boldsymbol{\theta}^1)^\intercal, \ldots, (\boldsymbol{\theta}^L)^\intercal \right)^\intercal$, where $\boldsymbol{\theta}^l = \mathrm{vec}(\mathbf{W}^l)$ is the subset of parameters $\boldsymbol{\theta}$ belonging to layer l. Given a training dataset $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$ consisting of paired examples $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y}$, the parameters are updated to minimize the empirical risk $\mathcal{L}(\boldsymbol{\theta}; D) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i))$,where $\ell : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a loss function. We regard the gradient of $\mathcal{L}$ w.r.t. $\boldsymbol{\theta}$, i.e., $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t) = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \big|_{\boldsymbol{\theta} = \boldsymbol{\theta}_t}$ as a column vector. Given a set of matrices $\{\mathbf{A}_1, \ldots, \mathbf{A}_L\}$, where $\mathbf{A}_i \in \mathbb{R}^{n_i \times n_i}$, $\mathrm{diag}(\mathbf{A}_1, \ldots, \mathbf{A}_L)$ denotes a block diagonal matrix with L sub-matrices on the diagonal and its size is $\sum_{i=1}^{L} n_i \times \sum_{i=1}^{L} n_i$.

IV. MBAG Algorithm

In this section, we describe each component of the proposed MBAG algorithm. At iteration t, MBAG computes the gradient

Algorithm 1: MBAG algorithm

Input: initial parameter vector $\boldsymbol{\theta}_0$, parameter grouping for each layer $\Pi^l = (\pi_1^l, \ldots, \pi_K^l)$, step size η_t, total number of iterations T, matrix square root computation interval T_{sqrt}, number of Newton-Schulz iterations T_{NS}, momentum β

Initialize $\mathbf{v}_0 \leftarrow \mathbf{0}$, $\mathbf{H}_0^{l,\pi_i} \leftarrow \frac{1}{\lambda}\mathbf{I}$ for $\forall l, \pi_i$

for $t = 1$ **to** T **do**

  Sample a mini-batch B of size m

  $\mathbf{g}_t \leftarrow \frac{1}{m}\sum_{i\in B} \nabla \ell(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i))$

  foreach *parameter group* $\pi_i \in \Pi^l$ *in layer* l **do**

    Update $\mathbf{H}_t^{l,\pi_i}$ according to (4)

    if $t \equiv 0 \pmod{T_{sqrt}}$ **then**

      Compute the matrix square root $(\mathbf{H}_t^{l,\pi_i})^{\frac{1}{2}} \leftarrow \text{Newton-Schulz}(\mathbf{H}_t^{l,\pi_i}, T_{\text{NS}})$

    else

      $(\mathbf{H}_t^{l,\pi_i})^{\frac{1}{2}} \leftarrow (\mathbf{H}_{t-1}^{l,\pi_i})^{\frac{1}{2}}$

    $\mathbf{v}_t^{l,\pi_i} \leftarrow \beta \mathbf{v}_{t-1}^{l,\pi_i} + (\mathbf{H}_t^{l,\pi})^{\frac{1}{2}} \mathbf{g}_t^{l,\pi_i}$

    $\boldsymbol{\theta}_t^{l,\pi_i} \leftarrow \boldsymbol{\theta}_{t-1}^{l,\pi_i} - \eta_t \mathbf{v}_t$

using a mini-batch and updates the preconditioner as follows:

$$\mathbf{G}_t = \mathbf{G}_{t-1} + \mathbf{g}_t \mathbf{g}_t^\intercal \,, \tag{2}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{G}_t^{-\frac{1}{2}} \mathbf{g}_t \,, \tag{3}$$

where $\mathbf{G}_0 = \lambda \mathbf{I}$ with $\lambda > 0$ and $\mathbf{g}_t = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t)$ is the stochastic gradient. The pseudocode of proposed algorithm is presented in Algorithm 1.

A. Computing the Inverse of Preconditioner

A computational bottleneck in the proposed method is the computation of matrix inverse square root in (3), i.e., computation of $\mathbf{G}^{-\frac{1}{2}}$. To reduce the computational burden, we take a two-step approach. Instead of computing and maintaining $\mathbf{G}_t$, MBAG directly computes and maintains the inverse $\mathbf{G}_t^{-1}$ using the Sherman-Morrison formula. For notational convenience, we write $\mathbf{H}$ to denote the inverse of our curvature matrix $\mathbf{G}$, i.e., $\mathbf{H} = \mathbf{G}^{-1}$. Given $\mathbf{H}_t$, MBAG applies a variant of Newton iterations to approximately compute its square root. Unrolling the recurrence equation in (2) shows that the preconditioner $\mathbf{G}_t$ is a sequence of rank-one update to the multiple of identity matrix. Therefore, it is symmetric and positive definite and has an inverse. Furthermore, the updated preconditioner $\mathbf{G}_t$ in (2) is a rank-one modification to $\mathbf{G}_{t-1}$. This means that, given $\mathbf{G}_{t-1}^{-1}$ and $\mathbf{g}_t$, the inverse of $\mathbf{G}_t$ can be analytically obtained using the Sherman-Morrison formula. Thus, the algorithm keeps a variable for $\mathbf{H}_t = \mathbf{G}_t^{-1}$ and iteratively updates it as follows.

$$\mathbf{H}_t = (\mathbf{G}_{t-1} + \mathbf{g}_t \mathbf{g}_t^\intercal)^{-1} = \mathbf{H}_{t-1} - \frac{\mathbf{H}_{t-1}\mathbf{g}_t\mathbf{g}_t^\intercal \mathbf{H}_t}{1 + \mathbf{g}_t^\intercal \mathbf{H}_t \mathbf{g}_t} \,, \tag{4}$$

where $\mathbf{H}_0 = \frac{1}{\lambda}\mathbf{I}$. The above update can be efficiently done as it only involves GPU friendly matrix-vector operations.

B. Computing the Matrix Square Root

To precondition the gradient, MBAG requires computing the inverse square root $\mathbf{G}^{-\frac{1}{2}}$. The square root of matrix $\mathbf{G} \in \mathbb{R}^{d\times d}$ is defined as any matrix $\mathbf{A} \in \mathbb{R}^{d\times d}$ such that $\mathbf{A}^2 = \mathbf{G}$. A common way to compute the square root of $\mathbf{G}$ is through its eigenvalue decomposition whose time complexity in practice is $\mathcal{O}(d^3)$. Let $\mathbf{G} = \mathbf{Q}\boldsymbol{\Lambda}\mathbf{Q}^\intercal$ be the eigenvalue decomposition of $\mathbf{G}_t$, where $\mathbf{Q}$ is orthogonal and $\boldsymbol{\Lambda}$ is a diagonal matrix whose diagonal entries are the eigenvalues of $\mathbf{G}_t$. It is easy to see that $\mathbf{A} = \mathbf{Q}\boldsymbol{\Lambda}^{1/2}\mathbf{Q}^\intercal$. The square root $\mathbf{A}$ is uniquely determined when $\mathbf{G}$ is positive definite. To reduce the computation time, we propose to compute the square root approximately by running a small number of Newton-Schulz iterations. Specifically, we set $\mathbf{X}_0 = \mathbf{G}^{-1}$ and $\mathbf{Y}_0 = \mathbf{I}$ and run the following Newton-Schulz iterations:

$$\begin{aligned} \mathbf{X}_t &= \frac{1}{2}\mathbf{X}_{t-1}(3\mathbf{I} - \mathbf{Y}_{t-1}\mathbf{X}_{t-1}) \,, \\ \mathbf{Y}_t &= \frac{1}{2}(3\mathbf{I} - \mathbf{Y}_{t-1}\mathbf{X}_{t-1})\mathbf{Y}_{t-1} \,. \end{aligned} \tag{5}$$

It is known that $\mathbf{X}_t$ and $\mathbf{Y}_t$ quadratically converges to $\mathbf{G}^{-\frac{1}{2}}$ and $\mathbf{G}^{\frac{1}{2}}$, respectively [15]. The sufficient condition for (5) to converge is $\|\mathbf{I} - \mathbf{G}^{-1}\|_p < 1$ for $p = 1, 2$, or ∞. In our implementation, to satisfy the condition, we pre-process the input matrix $\mathbf{G}^{-1}$ by normalizing it by its Frobenius norm. Let $\mathbf{A}'$ denote the normalized input matrix, i.e., $\mathbf{A}' = \frac{1}{\|\mathbf{G}^{-1}\|_{\text{F}}}\mathbf{G}^{-1}$. Let $\mathbf{X}_N$ be the value of $\mathbf{X}_t$ after N Newton-Schulz iterations with $\mathbf{X}_0 = \mathbf{A}'$. To account for the normalization, the resulting $\mathbf{G}^{-\frac{1}{2}}$ is adjusted by computing $\mathbf{G}^{-\frac{1}{2}} = \sqrt{\|\mathbf{G}^{-1}\|_{\text{F}}}\,\mathbf{X}_N$. In the above, if $\mathbf{X}_0$ is initialized to $\mathbf{G}$, $\mathbf{Y}_t$ in (5) converges to $\mathbf{G}^{-\frac{1}{2}}$. This means that we can compute the inverse square root of $\mathbf{G}$ without using the Sherman-Morrison formula. This variant of MBAG is shown in Algorithm 2. Another variant is the one that does not compute the square root and preconditions the gradient just with $\mathbf{G}^{-1}$. We name this variant as MBAG-N and empirically evaluate in Section VI.

C. Grouping Parameters

The size of matrix $\mathbf{G}_t$ in (2) is quadratic in the number of parameters, and it becomes computationally intractable to compute and store $\mathbf{G}_t$ even for neural networks of moderate size. A common solution to mitigate this problem is to approximate $\mathbf{G}_t$ with a block diagonal matrix, i.e., $\mathbf{G}_t \approx \text{diag}(\mathbf{G}_t^1, \ldots, \mathbf{G}_t^L)$, where each diagonal block $\mathbf{G}_t^l$ computes the outer product of gradient w.r.t. the parameters of layer l. That is, $\mathbf{G}_t^l = \mathbf{G}_{t-1}^l + \mathbf{g}_t^l (\mathbf{g}_t^l)^\intercal$, where $\mathbf{g}_t^l = \nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}_t^l)$. MBAG further reduces the cost by decomposing $\mathbf{G}_t^l$ into small matrices. Specifically, given the parameter vector $\boldsymbol{\theta}^l \in \mathbb{R}^{d_l}$ of layer l, MBAG partitions $\boldsymbol{\theta}^l$ into K disjoint subsets of parameters $(\boldsymbol{\theta}_{\pi_1}^l, \boldsymbol{\theta}_{\pi_2}^l, \ldots, \boldsymbol{\theta}_{\pi_K}^l)$, i.e., $\bigcup_{i=1}^K \boldsymbol{\theta}_{\pi_i}^l = \boldsymbol{\theta}^l$ and $\boldsymbol{\theta}_{\pi_i} \cap \boldsymbol{\theta}_{\pi_j} = \emptyset$ for $i \neq j$. The algorithm assumes the independence between the parameters belonging to different groups and separately maintains the second order statistics, the sum of past gradient outer products, for each group $\boldsymbol{\theta}_{\pi_i}^l$:

$$\mathbf{G}_t^{l,\pi_i} = \mathbf{G}_{t-1}^{l,\pi_i} + \mathbf{g}_t^{l,\pi_i} (\mathbf{g}_t^{l,\pi_i})^\intercal \,, \text{ for } i = 1, \ldots, K \,. \tag{6}$$

Algorithm 2: MBAG-EMA algorithm

Input: initial parameter vector $\boldsymbol{\theta}_0$, parameter grouping for each layer Π^l, step size η_t, total number of iterations T, square root compute interval T_{sqrt}, number of Newton-Schulz iterations T_{NS}, momentum β, decay coefficient α

Initialize $\mathbf{v}_0 \leftarrow \mathbf{0}$, $\mathbf{G}_0^{l,\pi_i} \leftarrow \lambda \mathbf{I}$ for $\forall l, \pi_i$
for $t = 1$ **to** T **do**
  Sample a mini-batch B of size m
  $\mathbf{g}_t \leftarrow \frac{1}{m} \sum_{i \in B} \nabla \ell(y_i, f_{\boldsymbol{\theta}}(\mathbf{x}_i))$
  foreach *parameter group* $\pi_i \in \Pi^l$ *in layer* l **do**
    $\mathbf{G}_t^{l,\pi_i} \leftarrow \alpha \mathbf{G}_{t-1}^{l,\pi_i} + (1-\alpha) \mathbf{g}_t^{l,\pi_i} (\mathbf{g}_t^{l,\pi_i})^\intercal$
    if $t \equiv 0 \pmod{T_{sqrt}}$ **then**
      $(\mathbf{H}_t^{l,\pi_i})^{\frac{1}{2}} \leftarrow \text{Newton-Schulz}(\mathbf{G}_t^{l,\pi_i}, T_{\text{NS}})$
    else
      $(\mathbf{H}_t^{l,\pi_i})^{\frac{1}{2}} \leftarrow (\mathbf{H}_{t-1}^{l,\pi_i})^{\frac{1}{2}}$
    $\mathbf{v}_t^{l,\pi_i} \leftarrow \beta \mathbf{v}_{t-1}^{l,\pi_i} + (\mathbf{H}_t^{l,\pi})^{\frac{1}{2}} \mathbf{g}_t^{l,\pi_i}$
    $\boldsymbol{\theta}_t^{l,\pi_i} \leftarrow \boldsymbol{\theta}_{t-1}^{l,\pi_i} - \eta_t \mathbf{v}_t$

In the above, $\mathbf{g}^{l,\pi_i} = \frac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_{\pi_i}^l}$ and $\mathbf{G}_t^{l,\pi_i}$ denotes the second order statistics associated with the subset of parameters $\boldsymbol{\theta}_{\pi_i}^l$. This group-wise independence assumption results in further decomposing $\mathbf{G}_t^l$ into K smaller $\tau \times \tau$ matrices and approximating $\mathbf{G}_t^l$ as a block diagonal matrix $\mathbf{G}_t^l = \text{diag}(\mathbf{G}_t^{l,\pi_1}, \ldots, \mathbf{G}_t^{l,\pi_K})$. A natural question to ask is which grouping strategy achieves a good balance between the reduction in space and time complexity and the quality of approximation. We can group the parameters according to whether they share the same input, whether they contribute to the same output, or hybrid of them. We use the following strategy for our implementation.

Fully-connected layers. Let $\mathbf{W} \in \mathbb{R}^{m \times n}$ be the weight matrix of a fully-connected layer with m outputs and n inputs. Given the group size τ, our proposed algorithm splits $\text{vec}(\mathbf{W}^\intercal)$ into groups of fixed size τ, assuming τ divides mn. Figure 1 illustrates this parameter grouping strategy when $\tau = 2$. While being simple, it is general enough to include important cases. For example, when $\tau = n$, it groups the parameters connected to the same output neuron. When $\tau = m$ and we split $\text{vec}(\mathbf{W})$ (instead of $\text{vec}(\mathbf{W}^\intercal)$), it becomes grouping the parameters corresponding to the same input neuron. When $\tau = mn$, it is the same with the layer-wise block diagonal approximation. We empirically observed that in practice a small group size (e.g., $\tau = 16$ or 32) provides good performance while using a small space.

Convolutional layers. Consider a convolutional layer having C_{out} output and C_{in} input filters of size $\kappa \times \kappa$, and let $\mathbf{W} \in \mathbb{R}^{C_{\text{out}} \times C_{\text{in}} \times \kappa \times \kappa}$ denote its weights (i.e., kernel). We reshape the weights $\mathbf{W}$ into a matrix $\mathbf{W}'$ of size $(C_{\text{out}} \cdot C_{\text{in}}) \times \kappa^2$ and split it in the same way as the weights of fully-connected layers are split while fixing $\tau = \kappa^2$. In other words, we put the parameters corresponding to a specific input-output filter combination into the same group and as a result we maintain $C_{\text{out}} \cdot C_{\text{in}}$ matrices of size $\kappa^2 \times \kappa^2$. We note that the kernel size κ used in many state-of-the art convolutional neural networks (CNNs) is small and typically $\kappa = 3$ or $\kappa = 5$ are used.

V. Convergence Analysis

This section analyzes the convergence property of the proposed algorithm. To show the convergence of proposed MBAG algorithm to a stationary point, we follow the framework of [28]. For simplicity, we assume a fixed step size, i.e., $\eta_t = \eta$ for $t = 1, \ldots, T$. Let $f(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}; D)$ be our objective function. We make the following assumptions for the analysis.

- A1 **Differentiability.** The objective function f is continuously differentiable with respect to $\boldsymbol{\theta}$.
- A2 **L-smoothness.** There exists a constant $L > 0$ such that $\forall \boldsymbol{\theta}, \boldsymbol{\theta}' \in \boldsymbol{\Theta}$, $\|\nabla f(\boldsymbol{\theta}) - \nabla f(\boldsymbol{\theta}')\| \leq L \|\boldsymbol{\theta} - \boldsymbol{\theta}\|$.
- A3 **Bounded gradient.** There exists a constant G that satisfies $\|\nabla f(\boldsymbol{\theta})\| \leq G$ and $\|\mathbf{g}_t\| \leq G$ for $\forall t \geq 1$.
- A4 **Independent noise.** The noisy gradient is expressed as the sum of gradient and noise with mean zero, i.e., $\mathbf{g}_t = \nabla f(\boldsymbol{\theta}_t) + \boldsymbol{\zeta}_t$, $\mathbb{E}[\boldsymbol{\zeta}_t] = 0$. Further, $\boldsymbol{\zeta}_i$ and $\boldsymbol{\zeta}_j$ are independent for $i \neq j$.

The above assumptions are standard for analyzing the convergence of stochastic optimization algorithms for nonconvex problems.

Theorem 1. *Suppose that Assumptions A1 - A4 are satisfied. Let $\widehat{\mathbf{G}}_t$ be the block diagonal matrix constructed by $\mathbf{G}_t^{l,\pi_i}$ associated with parameter group π_i in layer l, for $l \in [L]$ and $i \in [K]$, $\widehat{\mathbf{H}}_t = \widehat{\mathbf{G}}_t^{-1}$, and $\gamma_t = \lambda_{\min}(\eta_t \widehat{\mathbf{H}}_t^{1/2})$. Then Algorithm 1 with a fixed step size η yields*

$$\min_{t \in [T]} \mathbb{E} \|\nabla f(\boldsymbol{\theta}_t)\|^2 \leq \frac{\frac{L\eta^2}{2} \left(\mathbb{E} \left[\log \frac{|\widehat{\mathbf{G}}_T|}{|\widehat{\mathbf{G}}_0|} \right] \right) + 2G^2 \eta \left(\text{tr}(\mathbf{H}_0^{1/2}) + \frac{1}{\sqrt{\lambda}} \right) + C}{\sum_{t=0}^T \gamma_t},$$

where $C = \mathbb{E}[f(\boldsymbol{\theta}_0) - f(\boldsymbol{\theta}^*)]$ and $\boldsymbol{\theta}^* \in \arg\min_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} f(\boldsymbol{\theta})$.

Theorem 1 shows that the convergence of MBAG algorithm is related to the eigenvalues of $\widehat{\mathbf{G}}_T$ and it converges to a stationary point when the log determinant of $\widehat{\mathbf{G}}_T$ is $o(\sum_{t=0}^T \gamma_t)$.

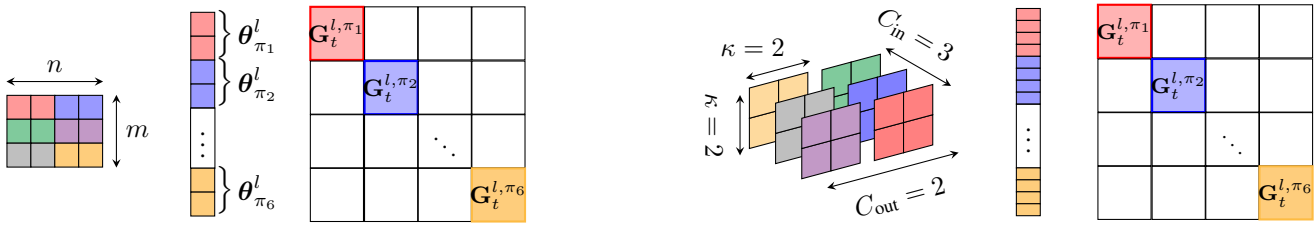
VI. Experimental Results

In this section, we evaluate the performance of MBAG on two tasks: image classification and AutoEncoder.

A. Image Classification Task

We evaluate the performance of MBAG on image classfication task and compare it to those of other first- and second-order baselines: Adam [5], AdaGrad [4], KFAC [1] [8], Shampoo [2] [9], AdaBlock [3] [19], and MBAG-N (a variant of MBAG that preconditions gradient using $\mathbf{G}^{-1}$ instead of $\mathbf{G}^{-\frac{1}{2}}$). For this task, we trained ResNet56 [29] model on the CIFAR10

[1] https://github.com/alecwangcq/KFAC-Pytorch
[2] https://github.com/moskomule/shampoo.pytorch
[3] https://proceedings.mlr.press/v151/yun22a/yun22a-supp.zip



(a) Fully-connected layers ($\tau = 2$) (b) Convolutional layer

Fig. 1: An example parameter grouping for a fully-connected layers and a 2D convolutional layer. The weights of the same color indicates that they belong to the same group $\boldsymbol{\theta}_{\pi_i}$ and their dependencies are modeled by $\mathbf{G}^{l,\pi_i}$.

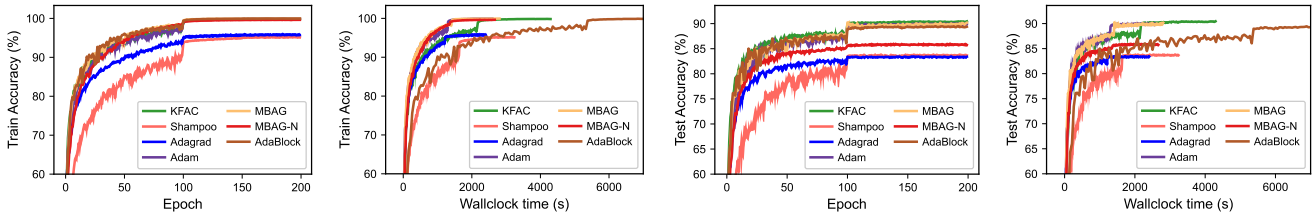


Fig. 2: Performance of ResNet56 on CIFAR10 dataset (Left: training, Right: test)

dataset which contains 50k and 10k 32×32 RGB images for training and testing, respectively. The size of mini-batch is set to 128 for all algorithms. For algorithms that performs periodic curvature matrix update and inversion, the intervals for update and inversion are set to $T_1 = 5$ and $T_2 = 50$, respectively. For fair comparison, MBAG also computes the matrix square root after every 50 iterations by running 5 Newton-Schulz iterations. For MBAG and AdaBlock, we fixed the size of each parameter group to $\tau = 32$. All the performance values reported in the experiments are averaged value of 5 runs. We run each algorithm for 200 epochs and decay the step size by multiplying by 0.1 at epoch 100 and 150. For MBAG, we set $\lambda = 0.01$ and $\beta = 0.9$. Figure 2 shows the train and test performance of algorithm against iterations and wall-clock time. Among the algorithms, KFAC achieves the highest test accuracy as expected although the gap between the first and second-order algorithms is not significant. As shown in Table I, the test accuracy of MBAG is closer to that of KFAC while it is as fast as Adam. This shows that MBAG benefits from both the second-order statistic it maintains and the efficient computation techniques employed to update the statistic. AdaBlock achieves similar test accuracy with Adam but it was approximately $4.13\times$ slower than Adam. The reason is that it performs eigenvalue decomposition at every iteration. Like other methods (e.g., KFAC), AdaBlock can be modified to perform the matrix inversion periodically, but we used the authors' implementation as it is.

	Test accuracy (%)
Adagrad	83.40 ± 0.415
Adam	89.93 ± 0.361
KFAC	$\mathbf{90.64} \pm 0.145$
Shampoo	83.67 ± 0.156
AdaBlock-32	89.38 ± 0.169
MBAG ($\tau = 32$)	$\mathbf{90.17} \pm 0.112$
MBAG-N ($\tau = 32$)	85.75 ± 0.327

TABLE I: Test accuracy for ResNet-56 on CIFAR10 dataset

B. AutoEncoder Task

We also test the performance of MBAG on AutoEncoder learning task using two datasets: FACES [30] and CURVES [31]. We built AutoEncoders using the same architecture used in [30] and trained using the binary cross-entropy loss. In this experiment, the size of mini-batch was set to 512. For KFAC and Shampoo, we fixed $T_1 = 5$ and $T_2 = 50$. For MBAG, we set $\lambda = 0.1$, $T_{\text{sqrt}} = 50$, $\beta = 0.9$, $T_{\text{NS}} = 7$. AdaBlock is not included in this experiment as its running time is prohibitively long compared to that of other algorithms. The two graphs on the left in Figure 3 show the change of training loss over iterations and wall-clock time on FACES dataset. As seen on image classification experiment, MBAG achieves the similar performance with KFAC but runs faster. Interestingly, there exists a noticeable performance gap between MBAG and MBAG-N in terms of accuracy. This shows the importance of taking square root of preconditioning matrix in AdaGrad like algorithms. The two graphs on the right in Figure 3 show the performance on CURVES dataset. For this dataset, our algorithm performed poorly than the first-order algorithms (i.e., Adam and AdaGrad). As shown in the figure, MBAG (including Shampoo and AdaBlock) decreases the objective value during the first few iterations and remains the same at the value around the value 0.158. We conjecture this is because that the algorithm is stuck at one of local minima and failed to escape.

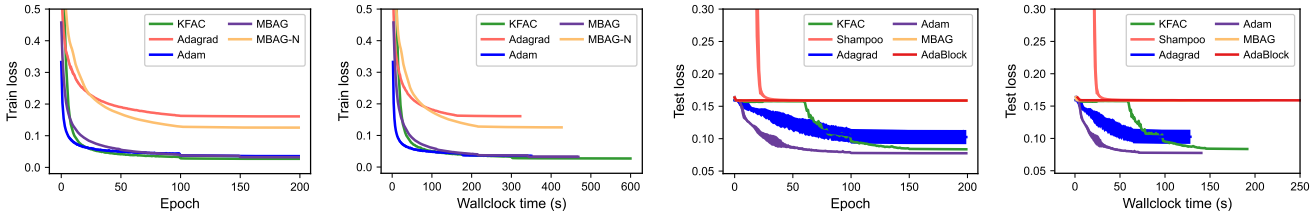


Fig. 3: Performance comparison of algorithms on AutoEncoder task (Left: FACES, Right: CURVES)

VII. Conclusion

In this work, we presented MBAG, a stochastic adaptive gradient algorithm. To reduce computational complexity of computing and obtaining the inverse square root of preconditioner, MBAG uses the Sherman-Morrison formula along with the Newton-Schulz iterations. This not only decreases the cost of preconditioner update but also allows an easy trade-off between approximation quality and computational efficiency by controlling the number of iterations. Furthermore, MBAG approximates the preconditioning matrix using the block-diagonal approximation in which the size of each diagonal can be arbitrary. By controlling the sizes of sub-blocks, it allows a smooth interpolation between the first-order and second-order algorithms. Therefore, our algorithm is suitable for large-scale deep learning models and is highly scalable.

References

[1] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400 – 407, 1951.

[2] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *Ussr computational mathematics and mathematical physics*, vol. 4, no. 5, pp. 1–17, 1964.

[3] Y. Nesterov, "A method for solving the convex programming problem with convergence rate O($1/k^2$)," *Doklady Akademii Nauk SSSR*, vol. 269, pp. 543–547, 1983.

[4] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization." *Journal of machine learning research*, vol. 12, no. 7, 2011.

[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015*, Y. Bengio and Y. LeCun, Eds., 2015.

[6] T. Tieleman, G. Hinton *et al.*, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[7] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in neural information processing systems*, 2013, pp. 315–323.

[8] J. Martens and R. Grosse, "Optimizing neural networks with Kronecker-factored approximate curvature," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Jul. 2015, pp. 2408–2417.

[9] V. Gupta, T. Koren, and Y. Singer, "Shampoo: Preconditioned Stochastic Tensor Optimization," in *Proceedings of the 35th International Conference on Machine Learning*. PMLR, Jul. 2018, pp. 1842–1850.

[10] D. Goldfarb, Y. Ren, and A. Bahamou, "Practical quasi-newton methods for training deep neural networks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, 2020, pp. 2386–2396.

[11] X. Wang, S. Ma, D. Goldfarb, and W. Liu, "Stochastic Quasi-Newton Methods for Nonconvex Stochastic Optimization," *SIAM Journal on Optimization*, vol. 27, no. 2, pp. 927–956, Jan. 2017.

[12] A. Botev, H. Ritter, and D. Barber, "Practical Gauss-Newton Optimisation for Deep Learning," in *Proceedings of the 34th International Conference on Machine Learning*. PMLR, Jul. 2017, pp. 557–565.

[13] J. Sherman and W. J. Morrison, "Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix," *The Annals of Mathematical Statistics*, vol. 21, no. 1, pp. 124–127, 1950.

[14] S.-i. Amari, "Neural learning in structured parameter spaces-natural Riemannian gradient," *Advances in neural information processing systems*, vol. 9, 1996.

[15] N. J. Higham, *Functions of Matrices*. Society for Industrial and Applied Mathematics, 2008.

[16] H. Zhang, C. Xiong, J. Bradbury, and R. Socher, "Block-diagonal hessian-free optimization for training neural networks," *arXiv preprint arXiv:1712.07296*, 2017.

[17] S.-W. Chen, C.-N. Chou, and E. Y. Chang, "EA-CG: An Approximate Second-Order Method for Training Fully-Connected Neural Networks," no. arXiv:1802.06502, Dec. 2018.

[18] A. Bahamou, D. Goldfarb, and Y. Ren, "A Mini-Block Natural Gradient Method for Deep Neural Networks," *arXiv:2202.04124 [cs]*, Feb. 2022.

[19] J. Yun, A. Lozano, and E. Yang, "AdaBlock: SGD with Practical Block Diagonal Matrix Adaptation for Deep Learning," in *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, 2022, pp. 2574–2606.

[20] F. Dangel, S. Harmeling, and P. Hennig, "Modular Block-diagonal Curvature Approximations for Feedforward Architectures," in *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, Jun. 2020, pp. 799–808.

[21] D. C. Liu and J. Nocedal, "On the limited memory BFGS method for large scale optimization," *Mathematical Programming*, vol. 45, no. 1, pp. 503–528, Aug. 1989.

[22] S.-i. Amari, H. Park, and K. Fukumizu, "Adaptive method of realizing natural gradient learning for multilayer perceptrons," *Neural computation*, vol. 12, no. 6, pp. 1399–1409, 2000.

[23] E. Frantar, E. Kurtic, and D. Alistarh, "M-fac: Efficient matrix-free approximations of second-order information," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34, 2021, pp. 14 873–14 886.

[24] E. Hazan, A. Agarwal, and S. Kale, "Logarithmic regret algorithms for online convex optimization," *Machine Learning*, vol. 69, no. 2, pp. 169–192, Dec. 2007.

[25] S. P. Singh and D. Alistarh, "WoodFisher: Efficient second-order approximation for neural network compression," in *Advances in Neural Information Processing Systems*, 2020, pp. 18 098–18 109.

[26] G. Krummenacher, B. McWilliams, Y. Kilcher, J. M. Buhmann, and N. Meinshausen, "Scalable adaptive stochastic optimization using random projections," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, 2016, pp. 1758–1766.

[27] N. Agarwal, B. Bullins, X. Chen, E. Hazan, K. Singh, C. Zhang, and Y. Zhang, "Efficient Full-Matrix Adaptive Regularization," in *Proceedings of the 36th International Conference on Machine Learning*, May 2019, pp. 102–110.

[28] X. Chen, S. Liu, R. Sun, and M. Hong, "On the Convergence of A Class of Adam-Type Algorithms for Non-Convex Optimization," in *International Conference on Learning Representations*, Feb. 2019.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[30] J. Martens, "Deep learning via hessian-free optimization," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, 2010, pp. 735–742.

[31] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.