# A General Regret Bound of Preconditioned Gradient Method for DNN Training

Hongwei Yong Ying Sun Lei Zhang
The Hong Kong Polytechnic University
hongwei.yong@polyu.edu.hk, {csysun, cslzhang}@comp.polyu.edu.hk

## Abstract

*While adaptive learning rate methods, such as Adam, have achieved remarkable improvement in optimizing Deep Neural Networks (DNNs), they consider only the diagonal elements of the full preconditioned matrix. Though the full-matrix preconditioned gradient methods theoretically have a lower regret bound, they are impractical for use to train DNNs because of the high complexity. In this paper, we present a general regret bound with a constrained full-matrix preconditioned gradient, and show that the updating formula of the preconditioner can be derived by solving a cone-constrained optimization problem. With the block-diagonal and Kronecker-factorized constraints, a specific guide function can be obtained. By minimizing the upper bound of the guide function, we develop a new DNN optimizer, termed AdaBK. A series of techniques, including statistics updating, dampening, efficient matrix inverse root computation, and gradient amplitude preservation, are developed to make AdaBK effective and efficient to implement. The proposed AdaBK can be readily embedded into many existing DNN optimizers, e.g., SGDM and AdamW, and the corresponding SGDM_BK and AdamW_BK algorithms demonstrate significant improvements over existing DNN optimizers on benchmark vision tasks, including image classification, object detection and segmentation. The code is publicly available at* https://github.com/Yonghongwei/AdaBK.

## 1. Introduction

Stochastic gradient descent (SGD) [26] and its variants [21, 23], which update the parameters along the opposite of their gradient directions, have achieved great success in optimizing deep neural networks (DNNs) [14, 24]. Instead of using a uniform learning rate for different parameters, Duchi *et al*. [5] proposed the AdaGrad method, which adopts an adaptive learning rate for each parameter, and proved that AdaGrad can achieve lower regret bound than SGD. Following AdaGrad, a class of adaptive learning rate gradient descent methods has been proposed. For example, RMSProp [30] and AdaDelta [35] introduce the exponential moving average to replace the sum of second-order statistics of the gradient for computing the adaptive learning rate. Adam [15] further adopts the momentum into the gradient, and AdamW [22] employs a weight-decoupled strategy to improve the generalization performance. RAdam [18], Adabelief [38] and Ranger [19, 32, 37] are proposed to accelerate training and improve the generalization capability over Adam. The adaptive learning rate methods have become the mainstream DNN optimizers.

In addition to AdaGrad, Duchi *et al*. [5] provided a full-matrix preconditioned gradient descent (PGD) method that adopts the matrix $\boldsymbol{H}_T = (\sum_{t=1}^{T} \boldsymbol{g}_t \boldsymbol{g}_t^\top)^{\frac{1}{2}}$ to adjust the gradient $\boldsymbol{g}_T$, where $t$ denotes the iteration number and $T$ is the number of the current iteration. It has been proved [5] that the preconditioned gradient $\boldsymbol{H}_T^{-1}\boldsymbol{g}_T$ has a lower regret bound than the adaptive learning rate methods that only consider the diagonal elements of $\boldsymbol{H}_T$. However, the full-matrix preconditioned gradient is impractical to use due to its high dimension, which limits its application to DNN optimization. Various works have been reported to solve this problem in parameter space by adding some structural constraints on the full-matrix $\boldsymbol{H}_T$. For instances, GGT [1] stores only the gradients of recent iterations so that the matrix inverse root can be computed efficiently by fast low-rank computation tricks. Yun *et al*. [34] proposed a mini-block diagonal matrix framework to reduce the cost through coordinate partitioning and grouping strategies. Gupta *et al*. [9] proposed to extend AdaGrad with Kronecker products of full-matrix preconditioners to make it more efficient in DNN training. Besides, natural gradient approaches [6, 7], which adopt the approximations of the Fisher matrix to correct the descent direction, can also be regarded as full-matrix preconditioners.

The existing constrained PGD (CPGD) methods, however, are heuristic since manually designed approximations to the full matrix $\boldsymbol{H}_T$ are employed in them, while their influence on the regret bound is unknown. By far, they lack a general regret-bound theory that can guide us to design the full-matrix preconditioned gradient methods. On the other hand, the practicality and effectiveness of these precondi-

tioner methods are also an issue, which prevents them from being widely used in training DNNs.

To address the above-mentioned issues, in this paper we present a theorem to connect the regret bound of the constrained full-matrix preconditioner with a guide function. By minimizing the guide function under the constraints, an updating formula of the preconditioned gradient can be derived. That is, optimizing the guide function of the preconditioner will minimize its regret bound at the same time, while different constraints can yield different updating formulas. With the commonly-used constraints on DNN preconditioners, such as the block-diagonal and Kronecker-factorized constraints [7, 9], specific guide functions can be obtained. By minimizing the upper bound of the guide function, a new optimizer, namely AdaBK, is derived.

We further propose a series of techniques, including statistics updating, dampening, efficient matrix inverse root computation and gradient norm recovery, to make AdaBK more practical to use for DNN optimization. By embedding AdaBK into SGDM and AdamW (or Adam), we develop two new DNN optimizers, SGDM_BK and AdamW_BK. With acceptable extra computation and memory cost, they achieve significant performance gain in convergence speed and generalization capability over state-of-the-art DNN optimizers, as demonstrated in our experiments in image classification, object detection and segmentation.

For a better understanding of our proposed regret bound and the developed DNN optimizer, in Fig. 1, we illustrate the existing major DNN optimizers and their relationships. SGD and its momentum version (SGDM) apply the same learning rate to all parameters based on their gradient descent directions. The adaptive learning rate methods assign different learning rates to different parameters by using second-order information of the gradients, achieving better convergence performance. The adaptive learning rate methods can be viewed as special cases of PGD methods by considering only the diagonal elements of the full preconditioned matrix of gradients. Our method belongs to the class of PGD methods, while our proposed general regret bound of constrained PGD methods can be applied to the PGD optimizers under different constraints, including AdaGrad, Full-Matrix AdaGrad and our AdaBK.

**Notation system.** We denote by $\boldsymbol{w}_t$ and $\boldsymbol{g}_t$ the weight vector and its gradient of a DNN model in the $t$-th iteration. Denote by $\boldsymbol{g}_{t,i}$ the gradient of the $i$-th sample in a batch in the $t$-th iteration, we have $\boldsymbol{g}_t = \frac{1}{n}\sum_{i=1}^{n}\boldsymbol{g}_{t,i}$, where $n$ is the batch size. The notations $\boldsymbol{A} \succeq \boldsymbol{0}$ and $\boldsymbol{A} \succ \boldsymbol{0}$ for a matrix $\boldsymbol{A}$ denote that $\boldsymbol{A}$ is symmetric positive semidefinite (PSD) and symmetric positive definite, respectively. $\boldsymbol{A} \succeq \boldsymbol{B}$ or $\boldsymbol{A} - \boldsymbol{B} \succeq \boldsymbol{0}$ means that $\boldsymbol{A} - \boldsymbol{B}$ is PSD. $\mathrm{Tr}(\boldsymbol{A})$ represents the trace of the matrix $\boldsymbol{A}$. For a PSD matrix $\boldsymbol{A}$, $\boldsymbol{A}^{\alpha} = \boldsymbol{U}\boldsymbol{\Sigma}^{\alpha}\boldsymbol{U}^{\top}$, where $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{U}^{\top}$ is the Singular Value Decomposition (SVD) of $\boldsymbol{A}$. $\|\boldsymbol{x}\|_{\boldsymbol{A}} = \sqrt{\boldsymbol{x}^{\top}\boldsymbol{A}\boldsymbol{x}}$ is the Mahalanobis norm of $\boldsymbol{x}$ induced by PSD matrix $\boldsymbol{A}$, and its dual norm is $\|\boldsymbol{x}\|_{\boldsymbol{A}}^{*} = \sqrt{\boldsymbol{x}^{\top}\boldsymbol{A}^{-1}\boldsymbol{x}}$. $\boldsymbol{A} \otimes \boldsymbol{B}$ means the Kronecker product of $\boldsymbol{A}$ and $\boldsymbol{B}$, while $\boldsymbol{A} \odot \boldsymbol{B}$ and $\boldsymbol{A}^{\odot\alpha}$ are the element-wise matrix product and element-wise power operation, respectively. $\mathrm{Diag}(\boldsymbol{x})$ is a diagonal matrix with diagonal vector $\boldsymbol{x}$, and $\mathrm{vec}(\ )$ denotes the vectorization function.

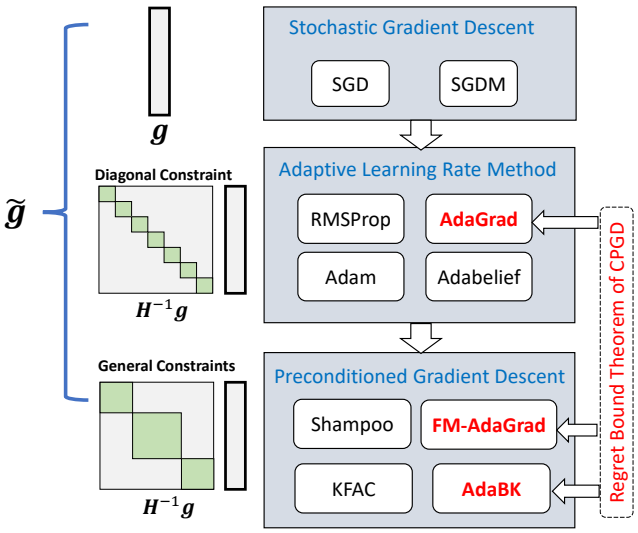


Figure 1. Illustration of the main DNN optimizers.

## 2. Background

### 2.1. Online Convex Optimization

The online convex optimization framework [10, 28] remains the most powerful and popular tool to analyze DNN optimization algorithms, including AdaGrad [5], Adam [15], Shampoo [9], *etc*. Given an arbitrary, unknown sequence of convex loss functions $f f_1(\boldsymbol{w}), ..., f_t(\boldsymbol{w}), ..., f_T(\boldsymbol{w})g$, we aim to optimize the weight $\boldsymbol{w}_t$ in the $t$-th iteration, and evaluate it on the loss function $f_t(\boldsymbol{w})$. The goal of our optimization process is to minimize the regret, which is defined as follows [10, 28]:

$$R(T) = \sum_{t=1}^{T} \left(f_t(\boldsymbol{w}_t) \quad f_t(\hat{\boldsymbol{w}})\right), \tag{1}$$

where $\hat{\boldsymbol{w}} = \arg\min_{\boldsymbol{w}} \sum_{t=1}^{T} f_t(\boldsymbol{w})$. Generally speaking, a lower regret bound means a more effective learning process.

### 2.2. Regret Bound of Preconditioned Gradient

As in previous works [5, 9], an online mirror descent with an adaptive time-dependent regularization is adopted for online convex learning. In the $t$-th iteration, suppose we have obtained the gradient $\boldsymbol{g}_t = r f_t(\boldsymbol{w}_t)$, then given a PSD matrix $\boldsymbol{H}_t \succeq \boldsymbol{0}$, the parameters are updated by optimizing the following objective function:

$$\boldsymbol{w}_{t+1} = \arg\min_{\boldsymbol{w}} \eta \boldsymbol{g}_t^{\top}\boldsymbol{w} + \frac{1}{2}\|\boldsymbol{w} \quad \boldsymbol{w}_t\|_{\boldsymbol{H}_t}^2. \tag{2}$$

The solution of Eq. (2) is exactly a preconditioned gradient descent step, which is

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t \quad \eta \boldsymbol{H}_t^{-1} \boldsymbol{g}_t. \tag{3}$$

Duchi *et al*. [5] have provided a regret bound for online mirror descent, as shown in **Lemma 1**:

**Lemma 1** [5, 9] *For any sequence of matrices* $\boldsymbol{H}_T$ ... $\boldsymbol{H}_1$ $\mathbf{0}$*, the regret of online mirror descent holds that*

$$R(T) \quad \frac{1}{2\eta}\sum_{t=1}^{T}\left(jj\boldsymbol{w}_t \quad \hat{\boldsymbol{w}}jj^2_{\boldsymbol{H}_t} \quad jj\boldsymbol{w}_{t+1} \quad \hat{\boldsymbol{w}}jj^2_{\boldsymbol{H}_t}\right) + \frac{\eta}{2}\sum_{t=1}^{T}\left(jj\boldsymbol{g}_t jj^*_{\boldsymbol{H}_t}\right)^2. \tag{4}$$

*If we further assume* $D = \max_t \ _T ||\boldsymbol{w}_t - \hat{\boldsymbol{w}}||_2$*, then we have*

$$R(T) \leq \frac{D^2}{2\eta} Tr(\boldsymbol{H}_T) + \frac{\eta}{2}\sum_{t=1}^{T}\left(||\boldsymbol{g}_t||_{\boldsymbol{H}_t}\right)^2. \tag{5}$$

Our goal is to find a proper sequence of PSD matrices $f\boldsymbol{H}_1, \boldsymbol{H}_2, ..., \boldsymbol{H}_T g$ to minimize the regret bound in Eq (4) or (5). Duchi *et al*. [5] suggested to adopt $\boldsymbol{H}_T = (\sum_{t=1}^{T} \boldsymbol{g}_t \boldsymbol{g}_t^\top)^{\frac{1}{2}}$ as the full matrix regularization matrix. However, it is hard to directly use it for DNN optimization due to the high dimension of parameter space. Therefore, Duchi *et al*. simplified this full-matrix $\boldsymbol{H}_T$ with its diagonal elements, resulting in the AdaGrad algorithm [5].

# 3. A General Regret Bound for Constrained Preconditioned Gradient

## 3.1. The General Regret Bound

Directly adopting a full-matrix $\boldsymbol{H}_t$ is absurd for optimizing a DNN because it is hard or even prohibitive to compute and store such a high-dimensional matrix. Hence, we need to reduce the dimension of $\boldsymbol{H}_t$ with a constraint set $\Psi$, *e.g*., the set of the block-diagonal matrices [5]. In this section, we aim to construct a general and practical full-matrix regularization term in Eq. (2) to achieve the low regret bound in Eq. (4). For a general constraint set $\Psi \quad \mathsf{R}^{d\times d}$, if it is a cone (*i.e*., $\theta\boldsymbol{x} \ 2 \ \Psi, \theta > 0$, $\theta\boldsymbol{x} \ 2 \ \Psi$ holds), we have the following **Theorem 1** and **Lemma 2**, whose proofs can be found in the **supplementary materials**.

**Theorem 1** *For any cone constraint* $\Psi \quad \mathsf{R}^{d\times d}$*, we define a guide function* $F_T(\boldsymbol{S})$ *on* $\Psi$ *as*

$$F_T(\boldsymbol{S}) = \sum_{t=1}^{T}(jj\boldsymbol{g}_t jj^*_{\boldsymbol{S}})^2, \tag{6}$$

*and then define the matrix* $\boldsymbol{H}_T$ *as*

$$\boldsymbol{H}_T = C_T \boldsymbol{S}_T, \quad \boldsymbol{S}_T = \arg\min_{\boldsymbol{S}\in\Psi, \boldsymbol{S}\succeq\mathbf{0}, Tr(\boldsymbol{S})\leq 1} F_T(\boldsymbol{S}), \tag{7}$$

*where* $C_T = \sqrt{F(\boldsymbol{S}_T)}$*. The regret of online mirror descent holds that*

$$R(T) \quad (\frac{D^2}{2\eta} + \eta)C_T = (\frac{D^2}{2\eta} + \eta)\sqrt{\min_{\boldsymbol{S}\in\Psi, \boldsymbol{S}\succeq\mathbf{0}, Tr(\boldsymbol{S})\leq 1} F_T(\boldsymbol{S})}. \tag{8}$$

The above theorem reveals that minimizing the guide function $F_T(\boldsymbol{S})$ on cone $\Psi$ will minimize the regret bound of the preconditioned gradient descent algorithm simultaneously. More importantly, given a cone constraint $\Psi$, the optimal $\boldsymbol{H}_T = C_T \boldsymbol{S}_T$ that achieves the lowest regret bound can be obtained by optimizing Eq. (7). From **Theorem 1**, we can know that the regret $R(T) \leq O(\sqrt{\min_{\boldsymbol{S}2\Psi, \boldsymbol{S} \ \mathbf{0}, \mathrm{Tr}(\boldsymbol{S}) \ 1} F_T(\boldsymbol{S})})$. If two cones satisfy $\Psi_1 \quad \Psi_2$, we have $\sqrt{\min_{\boldsymbol{S}2\Psi_2, \boldsymbol{S} \ \mathbf{0}, \mathrm{Tr}(\boldsymbol{S}) \ 1} F_T(\boldsymbol{S})} \leq \sqrt{\min_{\boldsymbol{S}2\Psi_1, \boldsymbol{S} \ \mathbf{0}, \mathrm{Tr}(\boldsymbol{S}) \ 1} F_T(\boldsymbol{S})}$. This also explains why full-matrix regularization can achieve the lowest regret bound. In addition, we have the following lemma:

**Lemma 2** *Suppose that* $\Psi$ *is the set of either diagonal matrices or full-matrices, according to the definition of* $\boldsymbol{S}_T$ *and* $\boldsymbol{H}_T$ *in Eq. (7), we have*

$$\boldsymbol{H}_T = Diag\big(\big(\sum_{t=1}^{T}\boldsymbol{g}_t \quad \boldsymbol{g}_t\big)^{\odot\frac{1}{2}}\big), \quad \boldsymbol{H}_T = \big(\sum_{t=1}^{T}\boldsymbol{g}_t\boldsymbol{g}_t^\top\big)^{\frac{1}{2}}. \tag{9}$$

From **Lemma 2**, we can easily see that the diagonal and full matrices used in AdaGrad [5] are two special cases of the results in **Theorem 1**.

## 3.2. Layer-wise Block-diagonal Constraint

In practice, we need to choose a proper constraint set $\Psi$ to regularize the structure of matrix $\boldsymbol{H}_T$. The diagonal constraint is the simplest constraint. However, it results in a very low effective dimension of $\boldsymbol{H}_T$ so that the regret bound is high. We aim to find a more effective and practical constraint set over $\boldsymbol{H}_T$ for DNN optimization.

Instead of considering the full-matrix regularization of all parameters, one can consider the full-matrix regularization of parameters within one DNN layer. Similar ideas have been adopted in KFAC [7] and Shampoo [9], which assume that the matrix $\boldsymbol{H}_T$ has a block diagonal structure and each sub-block matrix is used for one layer of a DNN. Suppose matrices $\boldsymbol{S}_l$ and $\boldsymbol{H}_l$ are for the $l$-th layer, and $\boldsymbol{g}_l$ is the gradient of weight in the $l$-th layer, in order to obtain the updating formula with block-diagonal constraint, we could minimize the guide function $F_T(\boldsymbol{S})$. There is

$$F_T(\boldsymbol{S}) = \sum_{t=1}^{T}(jj\boldsymbol{g}_t jj^*_{\boldsymbol{S}})^2 = \sum_{l=1}^{L}\sum_{t=1}^{T}(jj\boldsymbol{g}_{l,t} jj^*_{\boldsymbol{S}_l})^2. \tag{10}$$

The above equation shows that the original optimization problem can be divided into a number of $L$ sub-problems, and we can solve these sub-problems independently. For the convenience of expression, we omit the subscript $l$ and analyze the sub-problem within one layer of a DNN in the following development.

## 3.3. Kronecker-factorized Constraint

Because the dimension of the parameter space of one DNN layer can still be very high, we need to further constrain the structure of $\boldsymbol{H}_T$. The Kronecker-factorized constraint can be used to significantly reduce the parameter di-

mension within one layer [7, 9]. To be specific, for a fully-connected layer with weight $\boldsymbol{W} \in \mathsf{R}^{C_{out} \times C_{in}}$ and $\boldsymbol{w} = \text{vec}(\boldsymbol{W})$, its corresponding gradient is $\boldsymbol{G} \in \mathsf{R}^{C_{out} \times C_{in}}$ and $\boldsymbol{g} = \text{vec}(\boldsymbol{G})$. Let $\boldsymbol{S} = \boldsymbol{S}_1 \otimes \boldsymbol{S}_2$, where $\boldsymbol{S}_1 \in \mathsf{R}^{C_{out} \times C_{out}}$, $\boldsymbol{S}_2 \in \mathsf{R}^{C_{in} \times C_{in}}$ and $\boldsymbol{S} \in \mathsf{R}^{C_{in}C_{out} \times C_{in}C_{out}}$, and $\otimes$ is Kronecker product. Since $(\boldsymbol{S}_1 \otimes \boldsymbol{S}_2)^{-1} = \boldsymbol{S}_1^{-1} \otimes \boldsymbol{S}_2^{-1}$, what we need to minimize becomes

$$\begin{aligned} F_T(\boldsymbol{S}) &= \sum\nolimits_{t=1}^{T} (jj\boldsymbol{g}_t jj^*_{\boldsymbol{S}_1 \otimes \boldsymbol{S}_2})^2 = \sum\nolimits_{t=1}^{T} \boldsymbol{g}_t^\top (\boldsymbol{S}_1^{-1} \otimes \boldsymbol{S}_2^{-1}) \boldsymbol{g}_t \\ &= \text{Tr}\left( (\boldsymbol{S}_1^{-1} \otimes \boldsymbol{S}_2^{-1}) \sum\nolimits_{t=1}^{T} \boldsymbol{g}_t \boldsymbol{g}_t^\top \right) \end{aligned} \tag{11}$$

under the constraints $\{\boldsymbol{S}_1, \boldsymbol{S}_2 \succeq \boldsymbol{0}, \text{Tr}(\boldsymbol{S}_1) \leq 1, \text{Tr}(\boldsymbol{S}_2) \leq 1\}$.

Nevertheless, directly minimizing the $F_T(\boldsymbol{S})$ in Eq. (11) is still difficult, and we construct an upper bound of $F_T(\boldsymbol{S})$ to minimize. Since $\boldsymbol{g} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{g}_i$, where $\boldsymbol{g}_i$ is the gradient of sample $i$ and $n$ is the batch size, and $\boldsymbol{g}_i = \text{vec}(\boldsymbol{\delta}_i \boldsymbol{x}_i^T) = \boldsymbol{\delta}_i \otimes \boldsymbol{x}_i$, where $\boldsymbol{x}_i$ is the input feature and $\boldsymbol{\delta}_i$ is the output feature gradient of sample $i$, we have the following lemma.

**Lemma 3** *Denote by* $\boldsymbol{L}_T = \sum_{t=1}^{T}\sum_{i=1}^{n} \boldsymbol{\delta}_{ti}\boldsymbol{\delta}_{ti}^\top$ *and* $\boldsymbol{R}_T = \sum_{t=1}^{T}\sum_{i=1}^{n} \boldsymbol{x}_{ti}\boldsymbol{x}_{ti}^\top$*, there is*

$$\begin{aligned} F_T(\boldsymbol{S}) \leq\ & Tr\left( (\boldsymbol{S}_1^{-1} \otimes \boldsymbol{S}_2^{-1}) \frac{1}{n} \sum_{t=1}^{T}\sum_{i=1}^{n} \boldsymbol{g}_{ti}\boldsymbol{g}_{ti}^\top \right) \\ & \frac{1}{n} Tr(\boldsymbol{S}_1^{-1}\boldsymbol{L}_T) Tr(\boldsymbol{S}_2^{-1}\boldsymbol{R}_T). \end{aligned} \tag{12}$$

We minimize the upper bound of $F_T(\boldsymbol{S})$ defined in **Lemma 3**. One can see that the upper bound can be divided into two independent problems w.r.t. $\boldsymbol{S}_1$ and $\boldsymbol{S}_2$, respectively, which are

$$\min_{\boldsymbol{S}_1 \succeq \boldsymbol{0}, \text{Tr}(\boldsymbol{S}_1) \leq 1} \text{Tr}(\boldsymbol{S}_1^{-1}\boldsymbol{L}_T) \text{ and } \min_{\boldsymbol{S}_2 \succeq \boldsymbol{0}, \text{Tr}(\boldsymbol{S}_2) \leq 1} \text{Tr}(\boldsymbol{S}_2^{-1}\boldsymbol{R}_T). \tag{13}$$

To solve the above problem, we have the following lemma:

**Lemma 4** *If* $\boldsymbol{A} \succeq \boldsymbol{0}$*, we have:*

$$\arg \min_{\boldsymbol{S} \succeq \boldsymbol{0}, Tr(\boldsymbol{S}) \leq 1} Tr(\boldsymbol{S}^{-1}\boldsymbol{A}) = \boldsymbol{A}^{\frac{1}{2}} / Tr(\boldsymbol{A}^{\frac{1}{2}}). \tag{14}$$

The proofs of **Lemma 3** and **Lemma 4** can be found in the **supplementary materials**. According to **Lemma 4**, we know that the solution of Eq. (13) is $\boldsymbol{S}_{1,T} = \boldsymbol{L}_T^{\frac{1}{2}} / \text{Tr}(\boldsymbol{L}_T^{\frac{1}{2}})$ and $\boldsymbol{S}_{2,T} = \boldsymbol{R}_T^{\frac{1}{2}} / \text{Tr}(\boldsymbol{R}_T^{\frac{1}{2}})$. In practice, $\boldsymbol{L}_T$ and $\boldsymbol{R}_T$ will be added with a dampening term $\epsilon\boldsymbol{I}$ to ensure that they are symmetric and positive definite. Without considering the magnitude of $\boldsymbol{H}_T$, we can set

$$\boldsymbol{H}_T = \boldsymbol{H}_{1,T} \otimes \boldsymbol{H}_{2,T}, \boldsymbol{H}_{1,T} = \boldsymbol{L}_T^{\frac{1}{2}}, \boldsymbol{H}_{2,T} = \boldsymbol{R}_T^{\frac{1}{2}}. \tag{15}$$

Then according to the property of Kronecker product, the online mirror descent updating formula in Eq. (3) becomes

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \boldsymbol{H}_{1,t}^{-1} \boldsymbol{G}_t \boldsymbol{H}_{2,t}^{-1}. \tag{16}$$

We ignore the magnitude of $\boldsymbol{H}_T$ here because it will have no impact on the result after we introduce a gradient norm recovery operation in the algorithm, which will be described in the next section.

Finally, the proposed vanilla optimizer, termed AdaBK, is summarized in **Algorithm 1**.

# 4. Detailed Implementation

The proposed AdaBK in **Algorithm 1** involves the calculation of matrix inverse root, which may be unstable and inefficient. For an efficient and effective implementation of AdaBK in training DNNs, we propose a series of techniques.

**Efficient Matrix Inverse Root.** As shown in **Algorithm 1**, we need to calculate the matrix inverse root of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$. Traditional approaches usually use SVD to calculate it. Notwithstanding, SVD is inefficient and the existing deep learning frameworks (*e.g.*, PyTorch) do not implement SVD on GPU well, making the training unstable or even not converging. Instead of using SVD, we adopt the Schur-Newton algorithm [8] to compute the matrix inverse root. For matrix $\boldsymbol{A}$, let $\boldsymbol{Y}_0 = \boldsymbol{A}/\text{Tr}(\boldsymbol{A})$ and $\boldsymbol{Z}_0 = \boldsymbol{I}$. The Schur-Newton algorithm adopts the following iterations:

$$\begin{cases} \boldsymbol{T}_k = \frac{1}{2}(3\boldsymbol{I} - \boldsymbol{Z}_{k-1}\boldsymbol{Y}_{k-1}) \ ; \\ \boldsymbol{Y}_k = \boldsymbol{Y}_{k-1}\boldsymbol{T}_k, \ \boldsymbol{Z}_k = \boldsymbol{T}_k\boldsymbol{Z}_{k-1}, \ k = 1, 2, ..., K. \end{cases} \tag{17}$$

Then we have $\boldsymbol{A}^{\frac{1}{2}} \approx \boldsymbol{Y}_K \sqrt{\text{Tr}(\boldsymbol{A})}$, $\boldsymbol{A}^{-\frac{1}{2}} \approx \boldsymbol{Z}_K / \sqrt{\text{Tr}(\boldsymbol{A})}$. In practice, we find that setting $K = 10$ can achieve good enough precision for our problem.

**Statistics Updating.** In **Algorithm 1**, $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ accumulate the statistics of output feature gradient $\Delta_t$ and input feature $\boldsymbol{X}_t$, respectively. Hence the amplitude of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ will increase during training. After certain iterations, the effective learning rate will become small, making the learning process inefficient. To solve this issue, we use the exponential moving average of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$. Meanwhile, it is unnecessary to compute $\boldsymbol{L}_t$, $\boldsymbol{R}_t$, and their inverse root in each iteration. Two hyper-parameters $T_s$ and $T_{ir}$ are introduced to control the frequency of updating $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ and their inverse root, respectively. This infrequent statistics updating strategy can significantly improve efficiency with a little performance drop. We use two additional statistics $\widehat{\boldsymbol{L}}_t$ and $\widehat{\boldsymbol{R}}_t$ to restore the matrix inverse root of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ (please refer to **Algorithm 2**).

**Dampening Strategy.** When the dimensions of $\Delta_t$ and $\boldsymbol{X}_t$ are high, $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ tend to be singular matrices with large condition numbers. A dampening term $\epsilon\boldsymbol{I}$ should be added into $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ to improve their condition number and enhance the stability of computing inverse root. As in [33], we adopt an adaptive dampening parameter $\epsilon\lambda_{max}$, where $\lambda_{max}$ is the max singular value of the matrix $\boldsymbol{L}_t$ or $\boldsymbol{R}_t$. With this setting, the condition number will be $\frac{\lambda_{max}+\epsilon\lambda_{max}}{\lambda_{min}+\epsilon\lambda_{max}} \leq \frac{1+\epsilon}{\epsilon}$, bounded by a value determined by $\epsilon$. Meanwhile, the maximum singular value of the symmetric matrix ($\boldsymbol{L}_t$ or $\boldsymbol{R}_t$) can be efficiently obtained by the power iteration method [2] as follows:

$$\begin{cases} \boldsymbol{v}_k = \boldsymbol{A}\boldsymbol{u}_{k-1}, \\ \boldsymbol{u}_k = \boldsymbol{v}_k / ||\boldsymbol{v}_k||_2, \ k = 1, 2, ..., K. \end{cases} \tag{18}$$

**Algorithm 1:** AdaBK (Adaptive Regularization with Block-diagonal and Kronecker-factorized Constraints)

**Input:** $\boldsymbol{W}_0$, $\boldsymbol{L}_0 = \epsilon \boldsymbol{I}_{C_{out}}$, $\boldsymbol{R}_0 = \epsilon \boldsymbol{I}_{C_{in}}$, $\eta$
**Output:** $\boldsymbol{W}_T$

**for** *t=1:T* **do**
  Receive $\boldsymbol{X}_t = [\boldsymbol{x}_{ti}]_{i=1}^n$ by forward propagation;
  Receive $\Delta_t = [\boldsymbol{\delta}_{ti}]_{i=1}^n$ by backward propagation;
  Compute gradient $\boldsymbol{G}_t$;
  Update preconditioners:
    $\boldsymbol{L}_t = \boldsymbol{L}_{t-1} + \Delta_t \Delta_t^\top$;
    $\boldsymbol{R}_t = \boldsymbol{R}_{t-1} + \boldsymbol{X}_t \boldsymbol{X}_t^\top$;
  Update weight:
    $\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \boldsymbol{L}_t^{\frac{1}{2}} \boldsymbol{G}_t \boldsymbol{R}_t^{\frac{1}{2}}$;
**end**

**Algorithm 2:** One Step Preconditioned Gradient of AdaBK

**Input:** $T_s, T_{ir}, \alpha, \epsilon, \beta, \boldsymbol{L}_{t-1}, \boldsymbol{R}_{t-1}, \widehat{\boldsymbol{L}}_{t-1}, \widehat{\boldsymbol{R}}_{t-1}, \boldsymbol{X}_t = [\boldsymbol{x}_{ti}]_{i=1}^n$, $\Delta_t = [\boldsymbol{\delta}_{ti}]_{i=1}^n, \boldsymbol{G}_t = \nabla_{\boldsymbol{W}_t} \mathcal{L}$
**Output:** $\tilde{\boldsymbol{G}}_t$

**if** $t \% T_s = 0$ **then**
  $\boldsymbol{L}_t = \alpha \boldsymbol{L}_{t-1} + (1-\alpha) \Delta_t \Delta_t^\top$;
  $\boldsymbol{R}_t = \alpha \boldsymbol{R}_{t-1} + (1-\alpha) \boldsymbol{X}_t \boldsymbol{X}_t^\top$;
**else**
  $\boldsymbol{L}_t = \boldsymbol{L}_{t-1}, \boldsymbol{R}_t = \boldsymbol{R}_{t-1}$;
**end**
**if** $t \% T_{ir} = 0$ **then**
  Compute $\lambda_{max}^L$ and $\lambda_{max}^R$ by Power Iteration;
  Compute $\widehat{\boldsymbol{L}}_t = (\boldsymbol{L}_t + \lambda_{max}^L \epsilon \boldsymbol{I})^{\frac{1}{2}}$ and $\widehat{\boldsymbol{R}}_t = (\boldsymbol{R}_t + \lambda_{max}^R \epsilon \boldsymbol{I})^{\frac{1}{2}}$ by Schur-Newton Iteration Eq. (17);
**else**
  $\widehat{\boldsymbol{L}}_t = \widehat{\boldsymbol{L}}_{t-1}$ and $\widehat{\boldsymbol{R}}_t = \widehat{\boldsymbol{L}}_{t-1}$;
**end**
$\widehat{\boldsymbol{G}}_t = \widehat{\boldsymbol{L}}_t \boldsymbol{G}_t \widehat{\boldsymbol{R}}_t$;
$\tilde{\boldsymbol{G}}_t = \widehat{\boldsymbol{G}}_t \frac{\|\boldsymbol{G}_t\|_2}{\|\widehat{\boldsymbol{G}}_t\|_2}$;

We use $\lambda_{max}$ $\|\boldsymbol{v}_K\|_2$ for our proposed adaptive dampening and set $K$ to 10 in our implementation.

**Gradient Norm Recovery.** Since the amplitude of the preconditioned gradient $\boldsymbol{L}_t^{-\frac{1}{2}} \boldsymbol{G}_t \boldsymbol{R}_t^{-\frac{1}{2}}$ may significantly differ from the amplitude of original $\boldsymbol{G}_t$, the optimal learning rate and weight decay will also differ from the original optimizer. It is expected that the well-tuned hyper-parameters in current optimizers (*e.g.*, SGDM, AdamW) can be directly used in our proposed AdaBK optimizer without further hyper-parameter tuning. To this end, we follow the strategy in [33] to re-scale the amplitude of the preconditioned gradient $\widehat{\boldsymbol{G}}_t = \boldsymbol{L}_t^{-\frac{1}{2}} \boldsymbol{G}_t \boldsymbol{R}_t^{-\frac{1}{2}}$ to the original gradient $\boldsymbol{G}_t$ by multiplying it with a scaling factor, *i.e.*,

$$\tilde{\boldsymbol{G}}_t = \widehat{\boldsymbol{G}}_t \frac{\|\boldsymbol{G}_t\|_2}{\|\widehat{\boldsymbol{G}}_t\|_2}. \tag{19}$$

It is easy to know that $\tilde{\boldsymbol{G}}_t$ and $\boldsymbol{G}_t$ have the same $L_2$ norm. With gradient norm recovery, the proposed AdaBK method can be easily embedded into existing optimizers without much extra hyperparameter tuning.

**Convolutional Layer.** We have discussed the optimization of FC layers in Section 3. For the Conv layer, the derivation process is similar. The convolution operation can be formulated as matrix multiplication with the $im2col$ operation [31, 36], and then the Conv layer can be viewed as an FC layer with $\mathfrak{A} = U_1(W)\mathfrak{X}$, where $\mathfrak{A}$ and $\mathfrak{X}$ are the output and input features after $im2col$ operation, and $U_1(\,)$ is the mode 1 unfold operation of a tensor. For example, for a convolution weight $\mathbf{W} \in \mathbb{R}^{C_{out} \times C_{in} \times k_1 \times k_2}$, we have $U_1(\mathbf{W}) \in \mathbb{R}^{C_{out} \times C_{in} k_1 k_2}$. $U_1(\mathbf{W})$ can be considered as the weight of the FC layer, and the remaining computation is the same as the FC layer.

**Embedding AdaBK into SGDM and AdamW.** With the above-introduced techniques, a more efficient and practical implementation of AdaBK can be obtained. The one-step preconditioned gradient of AdaBK is summarized in **Algorithm 2**. For a FC layer, the complexity of AdaBK is $T(O(\frac{C_{in}^3 + C_{out}^3}{T_{ir}}) + O(\frac{(C_{in}^2 + C_{out}^2)N}{T_s}) + O(C_{in} C_{out}(C_{in} + C_{out})))$, where $T$ is the total number of iterations. For a Conv layer, its complexity is $T(O(\frac{C_{in}^3 k_1^3 k_2^3 + C_{out}^3}{T_{ir}}) + O(\frac{(C_{in}^2 k_1^2 k_2^2 + C_{out}^2)N}{T_s}) + O(C_{in} k_1 k_2 C_{out}(C_{in} k_1 k_2 + C_{out})))$. In our implementation, $T_s$ and $T_{ir}$ are set to 200 and 2000, respectively, and the complexity is acceptable. In practice, it only costs $10\%$ $25\%$ additional training time.

AdaBK can be embedded into many existing optimizers. In this paper, we embed it into the two commonly used DNN optimizers, *i.e.*, SGDM and AdamW (or Adam), and name the obtained new optimizers as SGDM_BK and AdamW_BK accordingly. The detailed algorithms of SGDM_BK and AdamW_BK are summarized in the **supplementary materials**.

## 5. Experiments

We evaluate the proposed SGDM_BK and AdamW_BK optimizers on typical vision tasks, including image classification (on CIFAR100/CIFAR10 [16] and ImageNet [27]), object detection and segmentation (on COCO [17]). For the hyper-parameters of SGDM_BK and AdamW_BK, we set $\alpha = 0.9$, $T_s = 200$, $T_{ir} = 2000$, and $\epsilon = 0.00001$ throughout the experiments if not specified. Ablation studies on hyper-parameter selection can be found in the **supplementary material**. All experiments are conducted under the Pytorch 1.11 framework with NVIDIA GeForce RTX 2080Ti and 3090 Ti GPUs.

### 5.1. Image Classification

In the image classification task, we compare SGDM_BK and AdamW_BK with the representative and state-of-the-art DNN optimizers, including SGDM, AdamW [22], Ada-

Table 1. Testing accuracies (%) on CIFAR100/CIFAR10. The best and second best results are highlighted in bold and italic fonts, respectively. The numbers in red color indicate the improvement of SGDM_BK/AdamW_BK over SGDM/AdamW, respectively.

| Optimizer | SGDM | AdamW | Adagrad | RAdam | Adabelief | Shampoo | KFAC | WSGDM | SGDM_BK | AdamW_BK |
|---|---|---|---|---|---|---|---|---|---|---|
| CIFAR100 | | | | | | | | | | |
| ResNet18 | 77.20 ± .30 | 77.23 ± .10 | 71.55 ± .25 | 77.05 ± .15 | 77.43 ± .36 | 71.81 ± .40 | 78.25 ± .23 | *79.28* ± .27 | **79.30** ± .07 (↑2.10) | 78.66 ± .34 (↑1.43) |
| ResNet50 | 77.78 ± .43 | 78.10 ± .17 | 72.20 ± .15 | 78.20 ± .15 | 79.08 ± .23 | 71.31 ± .53 | 79.25 ± .26 | *80.90* ± .23 | **81.26** ± .20 (↑3.48) | 80.15 ± .19 (↑2.05) |
| VGG11 | 70.80 ± .29 | 71.20 ± .29 | 67.70 ± .18 | 71.08 ± .24 | 72.45 ± .16 | 63.56 ± .44 | 72.75 ± .31 | *73.42* ± .28 | **73.89** ± .13 (↑3.09) | 73.09 ± .29 (↑1.89) |
| VGG19 | 70.94 ± .32 | 70.26 ± .23 | 63.30 ± .58 | 73.01 ± .20 | 72.39 ± .27 | 65.62 ± .56 | 73.87 ± .43 | *74.82* ± .23 | **75.10** ± .13 (↑4.16) | 74.27 ± .25 (↑4.01) |
| DenseNet121 | 79.53 ± .19 | 78.05 ± .26 | 71.27 ± .79 | 78.65 ± .05 | 79.88 ± .08 | 74.95 ± .42 | 79.84 ± .33 | **81.23** ± .10 | *81.18* ± .27 (↑1.65) | 79.93 ± .23 (↑1.88) |
| CIFAR10 | | | | | | | | | | |
| ResNet18 | 95.10 ± .07 | 94.80 ± .10 | 92.83 ± .12 | 94.70 ± .18 | 95.12 ± .14 | 92.94 ± .27 | 95.01 ± .12 | *95.43* ± .08 | **95.44** ± .12 (↑0.34) | 95.22 ± .13 (↑0.42) |
| ResNet50 | 94.75 ± .30 | 94.72 ± .10 | 92.55 ± .39 | 94.72 ± .10 | 95.35 ± .05 | 92.61 ± .27 | 95.43 ± .16 | *95.80* ± .15 | **95.86** ± .05 (↑1.11) | 95.40 ± .07 (↑0.68) |
| VGG11 | 92.17 ± .19 | 92.02 ± .08 | 90.25 ± .25 | 92.00 ± .18 | 92.45 ± .18 | 89.01 ± .29 | 92.82 ± .11 | 92.95 ± .20 | **93.14** ± .26 (↑0.97) | *92.96* ± .07 (↑0.94) |
| VGG19 | 93.61 ± .06 | 93.40 ± .04 | 91.28 ± .14 | 93.57 ± .11 | 93.58 ± .12 | 90.62 ± .32 | 93.47 ± .09 | 93.91 ± .19 | **94.03** ± .15 (↑0.42) | *93.94* ± .10 (↑0.54) |
| DenseNet121 | 95.37 ± .17 | 94.80 ± .07 | 92.95 ± .23 | 95.02 ± .08 | 95.37 ± .04 | 94.37 ± .36 | 95.18 ± .22 | **95.72** ± .14 | *95.70* ± .13 (↑0.33) | 95.40 ± .04 (↑0.60) |

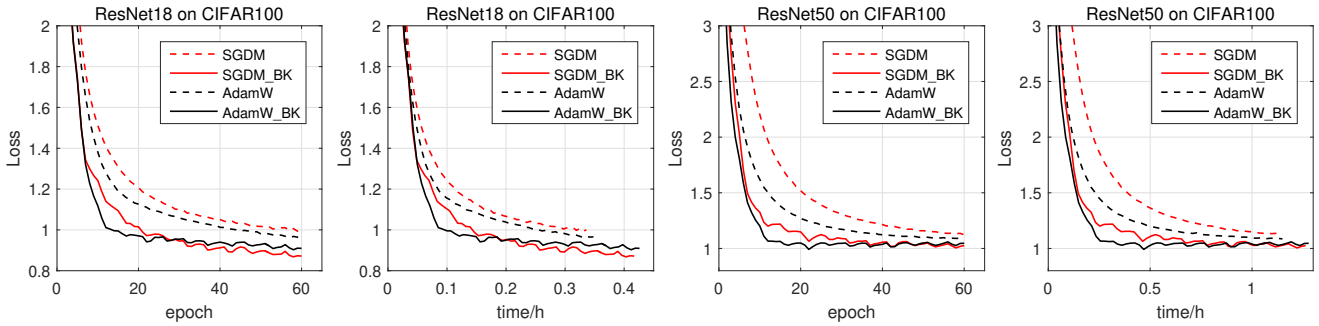


Figure 2. Training loss curves (loss vs. epoch and loss vs. time) of SGDM, SGDM_BK, AdamW and AdamW_BK on CIFAR100 with ResNet18 and ResNet50 before 60 epochs.

grad [5], RAdam [19][1], and Adabelief [38][2], Shampoo [9][3], KFAC [7] [9][4], WSGDM [33][5]. We tune learning rate and weight decay for each optimizer with grid search and the detailed settings for different optimizers can be found in the **supplementary material**.

**Results on CIFAR100/10:** We first testify the effectiveness of SGDM_BK and AdamW_BK with different DNN models on CIFAR100/CIFAR10 [16], including ResNet18, ResNet50 [12], VGG11 VGG19 [29] and DenseNet-121 [13] [6]. All the DNN models are trained for 200 epochs with batch size 128 on one GPU. The learning rate is multiplied by 0.1 for every 60 epochs. The experiments are repeated 4 times and the results are reported in a "mean std" format in Table 1. We can see that SGDM_BK and AdamW_BK achieve significant improvements over SGDM and AdamW, which are 1.44% 4.16% and 1.43% 4.01% on CIFAR100, and 0.28% 1.11% and 0.42% 0.94% on CIFAR10, respectively. They also surpass other compared optimizers for most of the used backbone networks.

Figure 2 shows the curves of training loss vs. epoch and training loss vs. time for SGDM, SGDM_BK, AdamW and AdamW_BK on CIFAR100 with ResNet18 and ResNet50 backbones before 60 epochs. One can see that SGDM_BK and AdamW_BK can significantly speed up the training process of SGDM and AdamW, respectively. Since SGDM_BK and AdamW_BK cost additional time in each iteration, for a fair comparison, we also show the curves of training loss vs. time. One can see that they still have great advantages over the original SGDM and AdamW.

**Results on ImageNet-1k:** To testify that SGDM_BK and AdamW_BK can also work well on large-scale datasets, we evaluate them on ImageNet-1k [27], which contains 1000 categories with 1.28 million images for training and 50K images for validation. ResNet18 and ResNet50 are selected as the backbone models with training batch size 256 on 4 GPUs, and the training settings follow the work in [**?**, 3]. The learning rate is multiplied by 0.1 for every 30 epochs. SGDM_BK and AdamW_BK adopt the same learning rate and weight decay as SGDM and AdamW, respectively. The top 1 accuracies on the validation set are reported in Table 2. One can see that SGDM_BK and AdamW_BK perform better than others. Meanwhile, we plot the training and validation accuracy curves in Figure 3, from which we see that the proposed AdaBK technique can largely speed up the training process. .

We also evaluate the proposed optimizer on Swin-

[1] https://github.com/LiyuanLucasLiu/RAdam

[2] https : / / github . com / juntang - zhuang / Adabelief - Optimizer

[3] https://github.com/moskomule/shampoo.pytorch

[4] https://github.com/alecwangcq/KFAC-Pytorch

[5] https://github.com/Yonghongwei/W-SGDM-and-W-Adam

[6] The model can be downloaded at https://github.com/weiaicunzai/pytorch-cifar100.

Table 2. Top 1 accuracy (%) on the validation set of ImageNet-1k. The numbers in red color indicate the improvement of SGDM_BK/AdamW_BK over SGDM/AdamW, respectively.

| Optimizer | SGDM | AdamW | Adagrad | RAdam | Adabelief | Shampoo | KFAC | WSGDM | SGDM_BK | AdamW_BK |
|---|---|---|---|---|---|---|---|---|---|---|
| ResNet18 | 70.49 | 70.01 | 62.22 | 69.92 | 70.08 | 64.45 | 69.62 | 71.43 | *71.59* ("1.10) | **71.63** ("1.62) |
| ResNet50 | 76.31 | 76.02 | 69.38 | 76.12 | 76.22 | 70.11 | 76.36 | *77.48* | **77.62** ("1.31) | 77.22 ("1.10) |

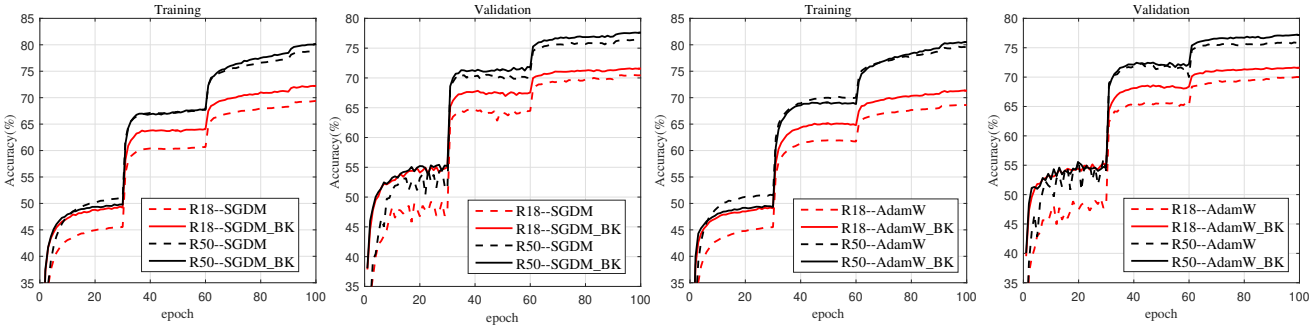


Figure 3. Training and validation accuracy curves of SGDM, SGDM_BK, AdamW and AdamW_BK on ImageNet-1k with ResNet18 and ResNet50 backbones.

Table 3. Top 1 accuracy (%) on the validation set of ImageNet-1k.

| Optimizer | AdamW | AdamW_BK |
|---|---|---|
| Swin-T | 81.18 | 81.79 ("0.61) |
| Swin-B | 83.02 | 83.14 ("0.12) |

transformer [20] backbone. We compare AdamW_BK with their default optimizer AdamW. The configurations follow the settings of the official MMClassification toolbox[7]. The results are shown in Table 3. We can see AdamW_BK can also achieves certain performance gain over AdamW.

## 5.2. Detection and Segmentation

We then evaluate SGDM_BK and AdamW_BK on COCO [17] detection and segmentation tasks to show that they can work well beyond classification tasks and can be used to fine-tune pre-trained models. The models are pre-trained on ImageNet1k and fine-tuned on COCO $train2017$ (118K images), and then evaluated on COCO $val2017$ (40K images). The latest version of MMDetection toolbox [4] is used as to train our models. We test SGDM_BK and AdamW_BK by Faster-RCNN [25] and Mask-RCNN [11] with various backbones, including ResNet50 (R50), ResNet101 (R101) and Swin transformer [20].

As mentioned in Section 4, with the gradient norm recovery operation, we can directly adopt the same hyper-parameters (*i.e.*, learning rate and weight decay) of SGDM and AdamW into SGDM_BK and AdamW_BK, respectively. To be specific, for R50 and R101 backbones, we compare the proposed optimizer with SGDM, WSGDM and AdamW. The learning rate and weight decay are set to 0.02 and 0.0001 for SGDM, WSGDM and SGDM_BK, and 0.0001 and 0.2 for AdamW and AdamW_BK, respectively.

For the Swin transformer backbone, the learning rate and weight decay are set to 0.0001 and 0.02 for AdamW and AdamW_BK, respectively. The learning rate schedule is 1X for Faster-RCNN. Other configurations follow the settings of the official MMDetection toolbox[8]. For the default optimizers, we use their official results[9]. This experiment is conducted on NVIDIA GeForce RTX 3090 Ti GPUs.

Table 4 lists the Average Precision (AP) of object detection by Faster-RCNN. It can be seen that the models trained by SGDM_BK and AdamW_BK achieve clear performance gains of 1.6% 2.2% AP for R50 and R101 backbones. Fig. 4 shows the training loss curves of Faster-RCNN with ResNet50 backbone. One can see that SGDM_BK and AdamW_BK accelerate the training process over SGDM and AdamW. Table 5 shows the $AP^b$ of detection and $AP^m$ of segmentation by Mask-RCNN. We can see that SGDM_BK and AdamW_BK gain 1.5% 2.2% $AP^b$ and 1.2% 2.2% $AP^m$ for R50 and R101 backbones over SGDM and AdamW, respectively. For Swin transformer backbone, AdamW_BK also improves 0.7% 0.9% $AP^b$ and 0.3% 0.9% $AP^m$ over AdamW. Meanwhile, compared with WSGDM, the proposed SGDM_BK also outperforms it with 0.2% 0.6% AP gain. Moreover, Fig. 5 plots the training loss curves of Faster-RCNN with ResNet50, Swin-T (1X) and Swin-S (3X). The proposed SGDM_BK and AdamW_BK accelerate the train-

[7] https://github.com/open-mmlab/mmclassification/tree/master/configs/swin_transformer

[8] https://github.com/open-mmlab/mmdetection

[9] Please refer to https://github.com/open-mmlab/mmdetection/tree/master/configs/faster_rcnn, https://github.com/open-mmlab/mmdetection/tree/master/configs/mask_rcnn, and https://github.com/open-mmlab/mmdetection/tree/master/configs/swin.

Table 4. Detection results of Faster-RCNN on COCO. Δ means the gain of SGDM_BK over SGDM or AdamW_BK over AdamW. * indicates the default optimizer.

| Backbone | Algorithm | AP | $AP_{.5}$ | $AP_{.75}$ | $AP_s$ | $AP_m$ | $AP_l$ |
|---|---|---|---|---|---|---|---|
| R50 | SGDM* | 37.4 | 58.1 | 40.4 | 21.2 | 41.0 | 48.1 |
| | WSGDM | 39.4 | 60.6 | 43.1 | 23.1 | 42.9 | 50.7 |
| | SGDM_BK | 39.6 | 60.7 | 42.8 | 22.6 | 42.9 | 52.2 |
| | Δ | "2.2 | "2.6 | "2.4 | "1.4 | "1.9 | "4.1 |
| | AdamW | 37.8 | 58.7 | 41.0 | 22.1 | 41.2 | 49.2 |
| | AdamW_BK | 39.4 | 60.3 | 42.9 | 22.5 | 42.8 | 52.3 |
| | Δ | "1.6 | "1.6 | "1.9 | "0.4 | "1.6 | "3.1 |
| R101 | SGDM* | 39.4 | 60.1 | 43.1 | 22.4 | 43.7 | 51.1 |
| | WSGDM | 41.1 | 61.6 | 45.1 | 24.0 | 45.2 | 54.3 |
| | SGDM_BK | 41.6 | 62.3 | 45.3 | 24.9 | 45.6 | 55.2 |
| | Δ | "2.2 | "2.2 | "2.2 | "2.5 | "1.9 | "4.1 |
| | AdamW | 40.1 | 60.6 | 43.8 | 22.9 | 44.1 | 52.8 |
| | AdamW_BK | 41.7 | 62.1 | 45.5 | 24.4 | 45.4 | 56.2 |
| | Δ | "1.6 | "1.5 | "1.7 | "1.5 | "1.3 | "3.4 |

Table 5. Detection and segmentation results of Mask-RCNN on COCO. Δ means the gain of SGDM_BK over SGDM or AdamW_BK over AdamW. * indicates the default optimizer.

| Backbone | Lr schedule | Algorithm | $AP^b$ | $AP^b_{.5}$ | $AP^b_{.75}$ | $AP^m$ | $AP^m_{.5}$ | $AP^m_{.75}$ |
|---|---|---|---|---|---|---|---|---|
| R50 | 1X | SGDM* | 38.2 | 58.8 | 41.4 | 34.7 | 55.7 | 37.2 |
| | | W-SGDM | 39.8 | 60.8 | 43.4 | 36.4 | 57.6 | 38.9 |
| | | SGDM_BK | 40.4 | 61.3 | 43.9 | 36.9 | 58.3 | 39.6 |
| | | Δ | "2.2 | "2.5 | "2.5 | "2.2 | "2.6 | "2.4 |
| | | AdamW | 37.8 | 58.7 | 41.0 | 35.4 | 56.2 | 38.0 |
| | | AdamW_BK | 40.0 | 60.6 | 43.5 | 36.7 | 58.0 | 39.3 |
| | | Δ | "2.2 | "1.9 | "2.5 | "1.3 | "1.8 | "1.3 |
| R100 | 1X | SGDM* | 40.0 | 60.5 | 44.0 | 36.1 | 57.5 | 38.6 |
| | | W-SGDM | 41.7 | 62.5 | 45.5 | 37.9 | 59.4 | 40.8 |
| | | SGDM_BK | 42.2 | 62.9 | 46.1 | 38.1 | 60.0 | 40.7 |
| | | Δ | "2.2 | "2.4 | "2.1 | "2.0 | " 2.5 | "2.1 |
| | | AdamW | 40.7 | 61.1 | 44.6 | 37.2 | 58.4 | 40.1 |
| | | AdamW_BK | 42.2 | 62.5 | 46.0 | 38.4 | 59.9 | 41.2 |
| | | Δ | "1.5 | "1.4 | "1.4 | "1.2 | "1.5 | "1.1 |
| Swin-T | 1X | AdamW* | 42.7 | 65.2 | 46.8 | 39.3 | 62.2 | 42.2 |
| | | AdamW_BK | 43.6 | 65.9 | 47.8 | 40.2 | 63.1 | 43.1 |
| | | Δ | "0.9 | "0.7 | "1.0 | "0.9 | "0.9 | "0.9 |
| Swin-T | 3X | AdamW* | 46.0 | 68.2 | 50.3 | 41.6 | 65.3 | 44.7 |
| | | AdamW_BK | 46.8 | 68.8 | 51.4 | 42.4 | 66.1 | 45.6 |
| | | Δ | "0.8 | "0.6 | "1.1 | "0.8 | "0.8 | "0.9 |
| Swin-S | 3X | AdamW* | 48.2 | 69.8 | 52.8 | 43.2 | 67.0 | 46.1 |
| | | AdamW_BK | 48.9 | 70.4 | 53.8 | 43.5 | 67.4 | 46.8 |
| | | Δ | "0.7 | "0.6 | "1.0 | "0.3 | "0.4 | "0.7 |

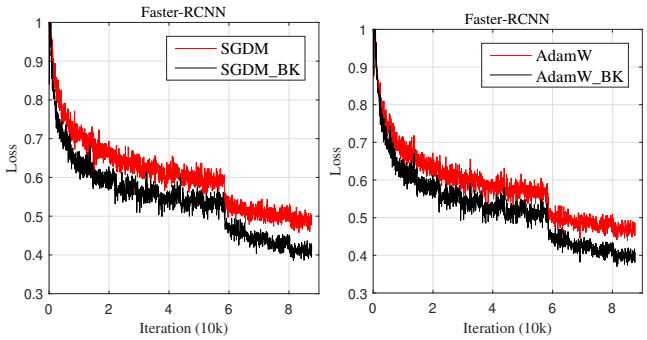

Figure 4. Training loss curves of ResNet50.

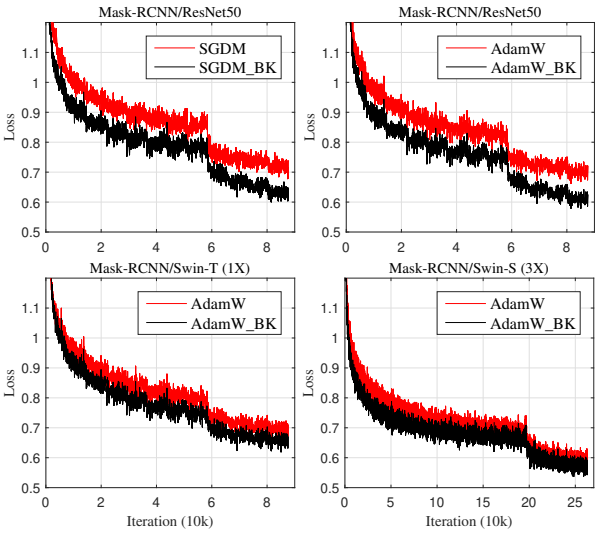

Figure 5. Training loss curves of Mask-RCNN.

ing process clearly. The results on COCO demonstrate that the proposed SGDM_BK and AdamW_BK can be easily adopted into the downstream tasks without additional hyper-parameter tuning.

### 5.3. Memory Usage and Training Time

For full-matrix adaptive optimizers, one important concern is the training cost, including memory usage and training time. Here we compare the memory and time cost of our optimizers with SGDM [23], AdamW [22] and Adagrad [5] on CIFAR100. ResNet50 is used as the backbone and one GeForce RTX 2080Ti GPU is used. The results are reported in Table 6. One can see that the embedding of AdaBK slightly increases the memory usage and training time (10% 25% extra training time and memory usage). Compared to the improvement of performance, the extra cost is affordable and worthwhile.

## 6. Conclusion

This work presented a general regret bound for the constrained full-matrix preconditioned gradient methods for DNN optimization. Different from previous full-matrix preconditioned methods, where the parameter update formulas are designed heuristically, we proved that given a cone constraint on the full-matrix preconditioner, the corresponding parameter update formula can be obtained by optimizing a guide function. Based on our theoretical analysis, we derived a specific guide function with the layer-wise block-diagonal constraint and Kronecker-factorized constraint. Through optimizing an upper bound of the guide function, a new preconditioned optimization algorithm, namely AdaBK, was obtained. We embedded AdaBK into two widely used optimizers, *i.e.*, SGDM and AdamW, and the experimental results on image classification, object detection and segmentation tasks demonstrated that AdaBK can significantly improve the DNN optimization performance with only 10% 25% extra computation cost.

Table 6. Memory cost (MiB) and training time (h) of different optimizers with ResNet50.

| Optimizer | SGDM | AdamW | Adagrad | SGDM_BK | AdamW_BK |
|---|---|---|---|---|---|
| Memory | 5867 | 5883 | 5865 | 6525 | 6535 |
| Time | 3.42 | 3.48 | 3.46 | 4.14 | 4.20 |

## References


[1] Naman Agarwal, Brian Bullins, Xinyi Chen, Elad Hazan, Karan Singh, Cyril Zhang, and Yi Zhang. Efficient full-matrix adaptive regularization. In *International Conference on Machine Learning*, pages 102–110. PMLR, 2019. 1

[2] Richard L Burden, J Douglas Faires, and Annette M Burden. *Numerical analysis*. Cengage learning, 2015. 4

[3] Jinghui Chen, Dongruo Zhou, Yiqi Tang, Ziyan Yang, Yuan Cao, and Quanquan Gu. Closing the generalization gap of adaptive gradient methods in training deep neural networks. *arXiv preprint arXiv:1806.06763*, 2018. 6

[4] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, et al. Mmdetection: Open mmlab detection toolbox and benchmark. *arXiv preprint arXiv:1906.07155*, 2019. 7

[5] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011. 1, 2, 3, 6, 8

[6] Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a kronecker-factored eigenbasis. *arXiv preprint arXiv:1806.03884*, 2018. 1

[7] Roger Grosse and James Martens. A kronecker-factored approximate fisher matrix for convolution layers. In *International Conference on Machine Learning*, pages 573–582. PMLR, 2016. 1, 2, 3, 4, 6

[8] Chun-Hua Guo and Nicholas J Higham. A schur–newton method for the matrix\boldmath p th root and its inverse. *SIAM Journal on Matrix Analysis and Applications*, 28(3):788–804, 2006. 4

[9] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018. 1, 2, 3, 4, 6

[10] Elad Hazan et al. Introduction to online convex optimization. *Foundations and Trends® in Optimization*, 2(3-4):157–325, 2016. 2

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017. 7

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 6

[13] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 6

[14] Ahmet Iscen, Giorgos Tolias, Yannis Avrithis, and Ondrej Chum. Label propagation for deep semi-supervised learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5070–5079, 2019. 1

[15] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. 1, 2

[16] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009. 5, 6

[17] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014. 5, 7

[18] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019. 1

[19] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019. 1, 6

[20] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021. 7

[21] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 1

[22] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017. 1, 5, 8

[23] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999. 1, 8

[24] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015. 1

[25] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015. 7

[26] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951. 1

[27] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015. 5, 6

[28] Shai Shalev-Shwartz et al. Online learning and online convex optimization. *Foundations and Trends® in Machine Learning*, 4(2):107–194, 2012. 2

[29] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 6

[30] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:26–31, 2012. 1

[31] Chengxi Ye, Matthew Evanusa, Hua He, Anton Mitrokhin, Tom Goldstein, James A Yorke, Cornelia Fermüller, and

Yiannis Aloimonos. Network deconvolution. *arXiv preprint arXiv:1905.11926*, 2019. 5

[32] Hongwei Yong, Jianqiang Huang, Xiansheng Hua, and Lei Zhang. Gradient centralization: A new optimization technique for deep neural networks. In *European Conference on Computer Vision*, pages 635–652. Springer, 2020. 1

[33] Hongwei Yong and Lei Zhang. An embedded feature whitening approach to deep neural network optimization. In *the European Conference on Conputer Vision*, 2022. 4, 5, 6

[34] Jihun Yun, Aurelie C Lozano, and Eunho Yang. Stochastic gradient methods with block diagonal matrix adaptation. *arXiv preprint arXiv:1905.10757*, 2019. 1

[35] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012. 1

[36] Huishuai Zhang, Wei Chen, and Tie-Yan Liu. Train feed-foward neural network with layer-wise adaptive rate via approximating back-matching propagation. *arXiv preprint arXiv:1802.09750*, 2018. 5

[37] Michael R Zhang, James Lucas, Geoffrey Hinton, and Jimmy Ba. Lookahead optimizer: k steps forward, 1 step back. *arXiv preprint arXiv:1907.08610*, 2019. 1

[38] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar C Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *Advances in neural information processing systems*, 33:18795–18806, 2020. 1, 6