Designing Effective Preconditioned Optimizers for Deep Neural Network Training

Ying Sun

PhD

The Hong Kong Polytechnic University

2024

The Hong Kong Polytechnic University

Department of Computing

Designing Effective Preconditioned Optimizers for Deep Neural Network Training

Ying Sun

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

April 2024

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signed)

Ying Sun (Name of student)

Abstract

Designing an effective optimizer for training deep neural networks (DNNs) has been under the research spotlight for the past decades, and one of the most effective ways is to design preconditioned optimizers. In this thesis, we focus on developing effective preconditioned optimizers from the following four aspects: Hessian-based preconditioned approach, nature gradient based preconditioned approach, preconditioned gradient adaptive stepsize approach, and attention-feature-based preconditioned approach for transformer structures. Accordingly, we present four algorithms: Stochastic Gradient Descent with Partial Hessian (SGD-PH), the Newton-Kronecker Factorized Approximate Curvature (NKFAC) algorithm, Adaptive learning rate method with a Rotation transformation (AdamR), and Attention-Feature-based Optimizer(AFOpt). The details of the works are as follows.

SGD-PH: In this work, we propose a compound optimizer, which is a combination of a second-order optimizer with a precise partial Hessian matrix for updating channel-wise parameters and the first-order stochastic gradient descent (SGD) optimizer for updating the other parameters. We show that the associated Hessian matrices of channel-wise parameters are diagonal and can be extracted directly and precisely from Hessian-free methods. The proposed method, namely SGD with Partial Hessian (SGD-PH), inherits the advantages of both first-order and second-order optimizers. Compared with first-order optimizers, it adopts a certain amount of information from the Hessian matrix to assist optimization, while compared with

the existing second-order optimizers, it keeps the good generalization performance of first-order optimizers. Experiments on image classification tasks demonstrate the effectiveness of our proposed optimizer SGD-PH.

NKFAC: This work presents a Newton-Kronecker factorized approximate curvature (NKFAC) algorithm, which incorporates Newton's iteration method for inverting second-order statistics. As the Fisher information matrix between adjacent iterations changes little, Newton's iteration can be initialized by the inverse obtained from the previous step, producing accurate results within a few iterations thanks to its fast local convergence. This approach reduces the computation time and inherits the property of second-order optimizers, enabling practical applications. The proposed algorithm is further enhanced with several useful implementations, resulting in state-of-the-art generalization performance without the need for extensive parameter tuning. The efficacy of NKFAC is demonstrated through experiments on various computer vision tasks.

AdamR: In pursuit of attaining a more favorable regret bound, we propose to integrate a rotation transformation into the existing adaptive learning rate algorithms. We employ the widely-recognized adaptive learning rate optimization method AdamW as a base optimizer, and develop a novel optimizer named AdamR. It consists of three steps in each iteration to compute the modified gradient. Firstly, the computation of the gradient with a rotation; secondly, the execution of the standard Adam step; and finally, the reorientation of the gradient back to its original space. The experimental results on image classification, object detection and segmentation have demonstrated AdamR's superior performance in accelerating the training process and improving the generalization capability.

AFOpt: For attention module that acting as the most critical module in transformers, in this work, we consider the gradient descent step in the attention matrix space and propose a preconditioned optimizer named AFOpt. By converting the

gradient step into the attention space, more information in the attention module can be combined into the final descent direction to assist the training process, which helps the transformer training. Numerical experiments are conducted to verify the effect of the proposed optimizer.

Overall, we propose four preconditioned optimizers in this thesis. Among them, SGD-PH adopts the Hessian information of normalization layers to assist in training DNNs; NKFAC combines Newton's method and practical implementations into KFAC to improve the effectiveness and efficiency; AdamR employs an adaptive step-size method with rotation transformation to achieve lower regret bound; and AFOpt performs the attention matrix based gradient step to better train transformers. The improvement of generalization performance in DNN training experiments proves their effectiveness.

Keywords: Optimizer, Second-order, Preconditioned Method, Newton Method, Nature Gradient Method, Adaptive Learning Rate, Error Bound, Attention.

Publications Arising from the Thesis

Conference Papers

1. **Ying Sun**, Hongwei Yong, and Lei Zhang. NKFAC: A Fast and Stable KFAC Optimizer for Deep Neural Networks. Joint European Conference on Machine Learning and Knowledge Discovery in Databases. Cham: Springer Nature Switzerland, 2023.

Preprints

1. **Ying Sun**, Hongwei Yong, and Lei Zhang. SGD with Partial Hessian for Deep Neural Networks Optimization. arXiv preprint arXiv:2403.02681 (2024).

2. **Ying Sun**, Hongwei Yong, and Lei Zhang. AdamR: Adaptive Learning Rate Optimization Method with a Rotation Transformation. Submitted.

3. **Ying Sun**, Hongwei Yong, and Lei Zhang. AFOpt: Attention Feature Based Optimizer for Transformers. In preprint.

Acknowledgements

It is never easy to jump out of the darkness and embrace a new life. Luckily, I am blessed by those people whom I sincerely respect and whom I deeply love.

Firstly, I want to express my utmost thanks and respect to my supervisor, Prof. Lei Zhang. I do not know how to express my deep gratitude to him, not only for his patient and thoughtful guidance but also for giving me a precise opportunity to research with him. Prof. Zhang is a wise, honest, and insightful scholar with great passion and interest in research and an empathetic elder willing to help students. I sincerely appreciate his guidance, support, and encouragement over the past few years. I will treasure these for the rest of my life.

Besides, I would like to thank my colleague, Dr. Hongwei Yong. He is a knowledgeable scholar and always selflessly shared his experience with me, which saved me many detours. I gained a lot from in-depth discussions with him, which I am grateful for. Special thanks also go to Dr. Shuai Li, Mr. Ming Liu, and Miss Minghan Li for their helpful discussion and generous help.

In particular, I want to thank Prof. Ben Young, an upright and excellent vice president of PolyU. He cares about, supports, and helps students from a practical perspective, and I am one of the luckiest students to be supported by him. Meanwhile, my thanks go to Dr. Dana Lo who sticks to her duty as a doctor.

Moreover, I would like to thank the Hong Kong Special Administrative Region Government for providing me with the Hong Kong PhD Fellowship and the Hong

Kong Polytechnic University for supporting research facilities.

Lastly, I would like to thank my parents, for their endless support, heartfelt encouragement, and unconditional love.

Table of Contents

CERTIFICATE OF ORIGINALITY iii

Abstract iv

Publications Arising from the Thesis vii

Acknowledgements viii

List of Figures xiv

List of Tables xvi

1 Introduction 1

1.1 Preconditioned Method and Its Application in Deep Neural Networks Optimization . . . 3

1.2 Special Network Structures for Optimization . . . 6

1.2.1 Normalization Layers . . . 6

1.2.2 Attention Module . . . 8

1.3 Contribution and Organization of Thesis . . . 9

1.4 Mathematical Notation System . . . 11

2 SGD with Partial Hessian for Deep Neural Networks Optimization 13

2.1 Introduction . . . 14

2.2 Related Work . . . 16

2.3 Methodology . . . 18

2.3.1 Diagonal Hessian Matrix . . . 18

2.3.2 The Hessian-free Approach 20
2.3.3 Techniques in Nonconvex Optimization 21
2.3.4 Generalizations on Convolutional Layers 23
2.4 Experiments 24
2.4.1 Results on CIFAR10/CIFAR100 25
2.4.2 Results on Mini-ImageNet 29
2.4.3 Results on ImageNet 30
2.4.4 Results about Convolutional Layers 31
2.4.5 Results on Person ReID Benchmarks 32
2.4.6 Ablation Studies 34
2.5 Conclusion 36
3 NKFAC: A Fast and Stable KFAC Optimizer for Deep Neural Networks 38
3.1 Introduction 39
3.2 Background and Preliminaries 41
3.2.1 Matrix Inverse 41
3.2.2 KFAC Algorithm 42
3.2.3 About Other Second-order Optimizers 43
3.3 Methodology 44
3.3.1 Motivation 44
3.3.2 Newton's Iteration of Matrix Inverse 45
3.3.3 NKFAC 48
3.3.4 Implementations and AdaNKFAC 49
3.4 Experiments 51
3.4.1 Experimental Setup 51
3.4.2 Results on CIFAR100/10 53

3.4.3 Results on ImageNet . . . 57
3.4.4 Results on COCO . . . 59
3.4.5 Ablation Studies . . . 61
3.4.6 Ablation Study on the stepsize of Newton's iteration . . . 63
3.4.7 Ablation Study on Implementations . . . 65
3.5 Conclusion . . . 66

4 AdamR: Adaptive Learning Rate Optimization Method with a Rotation Transformation 68

4.1 Introduction . . . 69
4.2 Preliminaries . . . 71
4.3 Methodology . . . 73
4.3.1 Rotation Transformation . . . 73
4.3.2 Solving the Diagonal Matrix . . . 75
4.3.3 Solving the Orthogonal Matrix . . . 78
4.4 Detailed Implementation . . . 80
4.5 Experiments . . . 85
4.5.1 Experiment Setup . . . 85
4.5.2 Image Classification . . . 85
4.5.3 Detection and Segmentation . . . 87
4.5.4 Ablation Study . . . 90
4.6 Conclusion . . . 91

5 AFopt: Attention Feature Based Optimizer for Transformers 94

5.1 Introduction . . . 94
5.2 Preliminaries . . . 96
5.2.1 Attention Module and Notations . . . 96
5.2.2 Feature Gradient Descent . . . 97

5.3 Methodology 98
5.3.1 Motivation 98
5.3.2 The update of $\mathbf{W_Q}$ and $\mathbf{W_K}$ 99
5.3.3 The update of $\mathbf{W_V}$ 100
5.3.4 Implementations 101
5.3.5 Overall Algorithm 103
5.4 Experiments 104
5.4.1 Experiment on COCO 104
5.4.2 Ablation Study 107
5.4.3 Analysis 108
5.5 Conclusion 109
6 Conclusion and Future Work **111**
6.1 Conclusion 111
6.2 Future Work 113
References **114**

List of Figures

1.1 Contribution and organization of this thesis. 9

2.1 Illustration of SGD-PH. Here, Figure (a) represents the case when BN is adopted in neural networks training, while other normalization methods such as LN, GN and IN can be represented in the same way. Figure (b) illustrates the case of decoupling the convolutional layers when there are no normalization layers followed, where $\boldsymbol{\beta}$ represents bias, please see Section 2.3.4 for more details. 18

2.2 Illustration of diagonal Hessian computation. Here, the light green boxes represent the elements that can be any real numbers while the white boxes represent zeros. The central 3×3 matrix (i.e., $\mathbf{H}_{SO}$) in $\mathbf{H}$ is diagonal corresponding to the specific 1D variable. By multiplies with the vector $\mathbf{e}_{SO}$, we can extract the diagonal elements precisely in the middle of $\mathbf{H}\mathbf{e}_{SO}$ and compute the element-wise inverse to get $\mathbf{D}_{SO}$, which is exactly the diagonal of $\mathbf{H}_{SO}^{-1}$. 21

2.3 Testing accuracy curves of different optimizers for different DNNs on CIFAR100 and CIFAR10 datasets. 28

2.4 Testing accuracy curves of different optimizers on Mini-ImageNet dataset. 31

2.5 Testing and training accuracy curves of VGG19 without BN layers. . 31

3.1 Change of $\mathbf{L}_t$ and $\mathbf{R}_t$ in training ResNet18 on CIFAR100. 46

3.2 Loss curves of Algorithm 1. 48

3.3 Training loss and testing accuracy curves with respect to epoch and time on CIFAR100 for different optimizers, respectively. 56

3.4 Training loss and validating accuracy curves w.r.t epoch for different optimizers on ImageNet with Swin-T/Swin-B. 63

3.5 Loss curves w.r.t iteration for different deep neural networks with different optimizers on COCO. 64

3.6 Time cost of optimization steps for KFAC*, SKFAC* and NKFAC on CIFAR100. 67

4.1 Training loss (left) and testing accuracy (right) curves to time of AdamW and AdamR on CIFAR100 with ResNet18 when the learning rates are deducted nearly simultaneously. 88

4.2 Training loss (left) and testing accuracy (right) curves to time of AdamW and AdamR on CIFAR100 with ResNet50 when the learning rates are deducted nearly simultaneously. 89

4.3 Training loss curves to epoch (left) and time (right) of 60 epochs training of AdamW and AdamR on CIFAR100 for ResNet50 with the initial learning rates kept. 90

4.4 Training and validation accuracy curves of AdamW and AdamR on ImageNet with ResNet18 and ResNet50 backbones. 91

4.5 Training loss curves of ResNet50. 92

4.6 Training loss curves of Mask-RCNN. 92

5.1 Illustration of attention module. 97

5.2 The loss curve (left) and the mAP index (right) of detection for different optimizers in training RetinaNet with Swin-T backbone, 1× schedule. 107

5.3 The mAP index for detection (left) and for segmentation (right) of different optimizers in training Mask-Rcnn with Swin-T backbone, 1× schedule. 108

List of Tables

2.1 Testing accuracies (%) of different DNNs with different optimizers on CIFAR100/10 datasets. 27

2.2 Testing accuracies (%) of DNNs with different optimizers for ResNet18/50 on Mini-ImageNet dataset. 27

2.3 Testing accuracies (%) of DNNs with different optimizers on ImageNet. The result of ADAM, AdamW, Adabelief and Adahessian are cited from [45], [10], [101], and [88], respectively. 30

2.4 Testing accuracies (%) of VGG19 with/without normalization layers on CIFAR100 dataset. 31

2.5 Experiment results (%) on Market1501. 33

2.6 Experiment results (%) on DukeMTMC-ReID. 34

2.7 Testing accuracies (%) of different LR and WD for ResNet18 on CIFAR100. 34

2.8 Testing accuracies (%) of different second-order learning rate τ_{SO} for different DNNs on CIFAR100/10 datasets. 35

2.9 Testing accuracies (%) of α for ResNet18 on CIFAR100. 35

2.10 Testing accuracies (%) of different batch size settings for ResNet18 on CIFAR100. 36

3.1 Testing accuracies (%) of different optimizers on CIFAR100/10. * means at least one of the four repeated experiments fails to converge. 55

3.2 Settings of learning rate (LR) and weight decay (WD) for different optimizers on CIFAR100/10. 57

3.3 Time cost by inversion (s) of optimizers on CIFAR100. 57

3.4 Settings of learning rate (LR) and weight decay (WD) for different optimizers on ImageNet. 58

3.5 Top 1 accuracy (%) for different optimizers on ImageNet with ResNet18/50. 58

3.6 Top 1 accuracy (%) with Swin-T and Swin-B. 59

3.7 Detection results of Faster-RCNN on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one. 61

3.8 Detection results of RetinaNet on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one. 61

3.9 Detection and segmentation results of Mask-RCNN on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one. 62

3.10 Testing accuracy (Acc.) of NKFAC with different dampening parameter η for ResNet50 on CIFAR100. 62

3.11 Testing accuracy (Acc.) of NKFAC with different statistics momentum α for ResNet50 on CIFAR100. 63

3.12 Testing accuracy (Acc.) and time cost by computing inversion of NKFAC with different number of Newton's step K. 65

3.13 Testing accuracy (Acc.) of NKFAC with different updating intervals T_{stat} and T_{inv} for ResNet50 on CIFAR100. 65

3.14 Detection results of Faster-RCNN on COCO. Δ means the improvement of α_{k+1} compared with α^*_{k+1}. 66

3.15 Detection and segmentation results of Mask-RCNN on COCO. Δ means the improvement of α_{k+1} compared with α^*_{k+1}. 66

3.16 Testing accuracies (%) and time cost by computing dampening and inversion (s) of different optimizers **with implementations** on CIFAR100. 67

4.1 Testing accuracies (%) on CIFAR100/CIFAR10. The best results are highlighted in bold fonts, and the numbers in red color indicate the improvement of AdamR over AdamW. 84

4.2 Settings of learning rate (LR), weight decay (WD) and WD methods for different optimizers on CIFAR10/100. Here, the WD methods include L_2 regularization weight decay (L_2 in short) and weight decouple (decouple in short). 86

4.3 Settings of learning rate (LR), weight decay (WD) and WD methods (L_2 and decouple) for different optimizers on ImageNet. 86

4.4 Top 1 accuracy (%) on the validation set of ImageNet. The numbers in red color indicate the improvement of AdamR over AdamW. . . . 87

4.5 Detection results of Faster-RCNN on COCO. Δ means the gain of AdamR over AdamW. * indicates the default optimizer. 89

4.6 Detection and segmentation results of Mask-RCNN on COCO. Δ means the gain of AdamR over AdamW. * indicates the default optimizer. 93

4.7 Accuracy (%) and time (s) of AdamR with different updating intervals in training ResNet50 on CIFAR100. 93

5.1 Detection results of RetinaNet for different parmeters on COCO. Δ_{FSGD} and Δ_{AFOpt} means the gain of FSGD and AFOpt over AdamW, respectively. * indicates the default optimizer. 106

5.2 Detection and segmentation results of Mask-RCNN for different parameters on COCO. Δ_{FAdam} and Δ_{Ada_AFOpt} means the gain of FSGD and AFOpt over AdamW, respectively. * indicates the default optimizer. 107

5.3 Detection results of RetinaNet, Swin-T backbone for different dampening parameter on COCO. 108

5.4 Detection results of RetinaNet, Swin-T backbone for different updating interval on COCO. 109

Chapter 1

Introduction

In the development of deep neural networks (DNNs), no proper algorithm can handle the deep structure well until the proposal of the well-known back-propagation (BP) algorithm [85, 67]. Based on the chain rule for derivation, the BP algorithm allows us to acquire the first-order information of DNNs at a reasonable cost. Thus, the first-order optimization algorithms have occupied the mainstream, among which the stochastic gradient descent (SGD) algorithm [56, 60] has become one of the most important baselines of training DNNs. Specifically, SGD uses the negative gradient direction as the descent direction of a stochastic chosen batch of samples, which, in traditional convex optimization, is the steepest descent direction.

Considering the fact that the steepest descent direction may not generate the fastest route that converge to a local minimum in traditional optimization, some other information and correction have been developed to better optimize the convergence route. Being the same in DNNs optimization, other information is extracted to assist optimization, and the most successful and wildest applied optimizer series generated is the series of adaptive stepsize optimizers e.g., AdaGrad [17], Adam [36], AdamW [47], Radam [45], AdaDelta [95] and Adabelief [101]. Beginning from AdaGrad, the design of this series essentially comes from the preconditioned method, which applies precondition matrix to the gradient to assist optimization. Later on, the element-

wise version Adam and its weight decay correction AdamW become popular, even preferred in many areas because of their merit of stable and better performance.

Mentioning preconditioned method, the most intuitive and direct way is to apply the inversion of Hessian matrix directly onto the gradient, e.g., AdaHessian [88]. However, considering the very high dimension of Hessian, the computation and storage cost is really high. Thus, to utilize the second order information, the other way is to consider the nature gradient (NG) method, where the Fisher Information Matrix (FIM) is applied as the precondition matrix. One of the successful applications is KFAC [52], which solves the problem of high dimension via Kronecker product. After KFAC, another preconditioned optimizer that deserves mentioned is Shampoo [25], where gradient is corrected with a better online regret bound. In Section 1.1, we will introduce the preconditioned methods, and briefly describe some representative preconditioned optimizers to see the applications of preconditioned methods in DNNs optimization.

There is still one thing that worth mentioning. Although in preconditioned optimizers, matrix inversion is necessary which will add on some unstable computation, it is still illustrated by experiments that preconditioned methods have better generalization performance and, thus deserve to be researched. Therefore, we can study how to find better ways of dealing with the inversion in DNNs optimizers.

Neural network optimization develops along with the development of neural networks. At the very beginning stage of neural networks development, it was soon mathematically proved that some early models (e.g., the simple shallow perceptrons) are limited to approximate the brain work [53], which led to a near stagnation of neural networks research. After being trainable by the BP algorithm, and thanks to the increasing computing power, network models developed quickly. After multilayer perceptrons (MLP), convolution layers show their effectiveness in image processing, and convolutional neural networks (CNNs) gradually become popular in many tasks,

especially in computer vision area (e.g., VGG [75], ResNet[29], DenseNet[31]).

Besides linear and convolutional layers, normalization layers also play important roles in DNNs. The commonly used normalization layers include batch normalization (BN) [33], weight normalization (WN) [70] layer normalization (LN) [4], group normalization (GN) [86] and instance normalization (IN) [82]. Normalization layers can boost the performance of neural networks and sometimes are regarded as a kind of optimization technique. Later on, the attention module is developed and widely applied with super performance (e.g., [16, 8, 46]). The attention module is a specially designed nonlinear module, which may contain extra information that can be extracted by optimizers. In Section 1.2, we will discuss these modules in detail.

The rest of this Chapter is arranged as follows. We first introduce preconditioned optimizers and some special structures in DNNs in Section 1.1 and Section 1.2, respectively. In Section 1.3, we conclude the main work and contribution in this thesis, and in Section 1.4, we briefly introduce the notation system throughout the thesis.

1.1 Preconditioned Method and Its Application in Deep Neural Networks Optimization

Preconditioner is an effective technique in numerical optimization. On the one hand, in traditional linear algebra, a preconditioner is always applied to a given matrix to lower its conditional number. Thus, the matrix will be more suitable for the rest of the computation, e.g., solving a linear equation system. On the other hand, from an optimization standpoint, a preconditioner is usually performed on the first-order descent direction to assist and accelerate the optimization process.

Here, we describe the sketch of the preconditioned method in optimization. For

an optimization problem

$$\arg\min_{\mathbf{W}} \mathcal{L}(\mathbf{W}), \tag{1.1}$$

where $\mathcal{L}(\mathbf{W})$ is a differentiable loss function and $\mathbf{W}$ is the parameter that needs to be optimized. The first order gradient descent here for the t-th iteration takes

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta_t \nabla \mathcal{L}(\mathbf{W}^t). \tag{1.2}$$

If we design an invertible matrix $\mathbf{H}^t$ as a preconditioner of the t-th iteration, then the preconditioned gradient descent takes the form

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta_t (\mathbf{H}^t)^{-1} \nabla \mathcal{L}(\mathbf{W}^t). \tag{1.3}$$

One may notice that Eq. (1.3) contains the well-known Newton's method, which is an effective second-order optimization method. If $\mathcal{L}$ is continuously differentiable, then the update equation of Newton's method at the t-th iteration takes

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta_t (\nabla^2 \mathcal{L}(\mathbf{W}^t))^{-1} \nabla \mathcal{L}(\mathbf{W}^t). \tag{1.4}$$

When convex and continuous differentiable, Newton's method has a second-order convergence rate, such that it is preferred in solving many kinds of statistical problems.

We can now correspond the above formula in traditional optimization with optimizers in DNN optimization. For a DNN optimization problem in Eq. (1.1), where $\mathcal{L}$ is the loss surface of the network and $\mathbf{W}$ refers to the group of the network parameters. After randomly splitting batches to deal with the large dataset, for a single iteration, the stochastic gradient descent is the same form as Eq. (1.2). After that, the pioneer of adaptive stepsize methods, AdaGrad, takes the form of Eq. (1.3) and adds a preconditioned matrix on the stochastic gradient. Specifically, for a batch of samples with number N, in the t-th iteration, the full-matrix AdaGrad use the

below preconditioner

$$\mathbf{H}^t = \frac{\mathbf{G}_\circ{}^{t^{1/2}}}{tr(\mathbf{G}_\circ^{t^{1/2}})}, \quad \mathbf{G}_\circ^t = \sum_{i=1}^{N} \mathbf{g}_i^t \mathbf{g}_i^{t^\top}, \tag{1.5}$$

where $\mathbf{g}_i^t$ is the vector format gradient for the i-th sample in the t-th iteration. The design of this preconditioner comes from the aspect of lowering the online regret bound. AdaGrad also holds a diagonal version, only utilizing $\text{diag}(\mathbf{G}_\circ{}^{t^{1/2}})$ instead of $\mathbf{G}_\circ{}^{t^{1/2}}$. This method is efficient for sparse gradient tasks. However, with the learning rate decreasing, it is not effective in other DNN training. Adam fixed this problem by combining the diagonal version with momentum. Specifically, by substituting the current $\mathbf{g}^t$ by $\beta_1 \mathbf{g}^t + (1-\beta_1)\mathbf{g}^{t-1}$ for some $\beta_1 \in (0,1)$, and computing

$$\mathbf{H}^t = \beta_2 \text{diag}(\mathbf{G}_\circ{}^{t^{1/2}}) + (1-\beta_2)\mathbf{H}^{t-1}, \quad \beta_2 \in (0,1) \tag{1.6}$$

with $\mathbf{G}_\circ^t$ follows the definition in Eq. (1.5). Here, the preconditioner and the gradient both cover information brought by more batches, making them more effective.

Next, we turn our attention to Eq.(1.4). Here, the preconditioner $\mathbf{H}^t = (\nabla^2 \mathcal{L}(\mathbf{W}^t))$ is exactly the second order derivative. However, applying the second-order derivative directly is not realistic because of the very high dimension of the DNN parameters. Thus, AdaHessian only extracts the diagonal element of Hessian, i.e.,

$$\mathbf{H}^t = \text{Diag}(\nabla^2 \mathcal{L}(\mathbf{W}^t)). \tag{1.7}$$

By only keeping the diagonal elements, the storage of AdaHessian can be reduced to an acceptable range. However, since the second time back-propagation is needed, the computation time increase a lot, which is an unavoidable question.

From another aspect, the nature gradient (NG) method also follows the preconditioner design. In NG methods, the preconditioner $\mathbf{H}^t$ refers to

$$\mathbf{H}^t = \mathbf{F}^t, \tag{1.8}$$

where $\mathbf{F}^t$ is the Fisher information matrix (FIM). Later on, $\mathbf{F}^t$ is equivalently converted into a left preconditioner and a right preconditioner by Kronecker decomposition, and the update formula finally, in matrix format, becomes

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta_t \mathbf{L}_{t+1}^{-1} \mathbf{G}_t \mathbf{R}_{t+1}^{-1},, \tag{1.9}$$

for some $\mathbf{L}_{t+1}$ and $\mathbf{R}_{t+1}$, and $\mathbf{G}_t$ is the matrix format gradient in the t-th iteration. We will introduce the detailed definition and further details needed in Chapter 3.

While converting the classic left preconditioner $\mathbf{H}^t$ into a left $\mathbf{L}^t$ and a right $\mathbf{R}^t$ preconditioner with the help of Kronecker decomposition is a common technique, we can also design the preconditioned optimizer in the form of Eq.(1.9) directly. An existing work is Shampoo, which designs the following left and right preconditioners directly

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \eta_t \mathbf{L}_{t+1}^{-1/4} \mathbf{G}_t \mathbf{R}_{t+1}^{-1/4}, \tag{1.10}$$

$$\mathbf{L}_{t+1} = \mathbf{L}_t + \mathbf{G}_t \mathbf{G}_t^\top, \quad \mathbf{R}_{t+1} = \mathbf{R}_t + \mathbf{G}_t^\top \mathbf{G}_t. \tag{1.11}$$

To sum up, there are several common ways to design preconditioned optimizers.

- Design a left preconditioner $\mathbf{H}^t$ corresponding to the second-order information (e.g., AdaHessian and KFAC).
- Design a left preconditioner $\mathbf{H}^t$ to attain a lower regret bound (e.g., AdaGrad).
- Design a pair of left and right preconditioners $(\mathbf{L}^t, \mathbf{R}^t)$ to get better performance (e.g., Shampoo).

1.2 Special Network Structures for Optimization

1.2.1 Normalization Layers

Normalization layers have been widely applied in many network structures, and sometimes become default modules in designing networks, e.g. in ResNet50 [29]. One

of the most commonly used normalization layers is batch normalization (BN), which is often employed in models for high-level tasks. BN uses the mean and variance of the features within a mini-batch to standardize the feature in training, and adopt an exponential moving average across the mini-batches in reference. The BN layer has some merits. Firstly, adding BN into the model is more friendly for parameter tuning, and higher learning rates are usually allowed. Secondly, it speeds up the loss convergence in the training process. Lastly and most importantly, it enhances the generalization performance of DNNs. Owing to these merits, the training of BN layers deserves special consideration.

Another important technique is weight normalization (WN). Unlike BN, which is applied to the features, WN deals with weight parameters and is more suitable for some tasks, e.g., reinforcement learning or generative models. It decouples the length from the weight parameter, converting the optimization of the weight parameter into two parts: the weight direction and the weight length. Decoupling contributes to accelerating the convergence of the SGDM to some degree. BN and WN share a common characteristic of reparameterizing the tensor, generating channel-wise 1D parameters, which opens up the possibility of utilizing it to design optimizers. Specifically, consider the BN layer, for a batch of input $\mathcal{X} \in \mathbb{R}^{N\times C\times W\times H}$, divide it into C sets $\mathcal{X}_1, \ldots, \mathcal{X}_C$ by channels, where $\mathcal{X}_i = \{x_i^1, \cdots, x_i^{m_i}\}$ for each channel $i \in \{1, 2, \ldots, C\}$, and $m_i = NWH$ is the number of the elements in the corresponding channel. Then for each channel $\mathcal{X}_i$, the BN layer takes

$$\mu_{\mathcal{X}_i} = \frac{1}{m_i}\sum_{j=1}^{m_i} x_i^j;\ \sigma^2_{\mathcal{X}_i} = \frac{1}{m_i}\sum_{j=1}^{m_i}(x_i^j - \mu_{\mathcal{X}_i})^2;\ \hat{x}_i^j = \frac{x_i^j - \mu_{\mathcal{X}_i}}{\sqrt{\sigma^2_{\mathcal{X}_i} + \epsilon}};\ y_i^j = \gamma_i \hat{x}_i^j + \beta_i, \quad (1.12)$$

and the final $\mathbf{\Gamma} = (\gamma_1, \cdots, \gamma_C)^T$ and $\boldsymbol{\beta} = (\beta, \cdots, \beta)^T$ are the 1D parameters we mentioned. Moreover, for the WN operation, consider a single 2D convolution layer $\mathbf{Y} = \mathbf{W} * \mathbf{X}$, where $\mathbf{W}$ is the kernel with dimension $C_{out} \times C_{in} \times k_1 \times k_2$ and $\mathbf{X}$ is the

input, then for each output channel $i \in \{1, \ldots, C_{out}\}$, the WN operation takes

$$\mathbf{W}_i = \gamma_i \frac{\mathbf{V}_i}{\|\mathbf{V}_i\|_2}, \quad \text{where } \mathbf{V}_i \in \mathbb{R}^{C_{in} \times k_1 \times k_2} \text{ and } \gamma_i \in \mathbb{R}, \tag{1.13}$$

and $\mathbf{\Gamma} = (\gamma_1, \ldots, \gamma_{C_{out}})$ is exactly the 1D parameter we need.

Besides WN and BN, there are also many other normalization methods, for example, layer normalization (LN) [4], instance normalization (IN) [82] and group normalization (GN) [86], which also generate similar 1D parameters and can be taken into consideration together.

1.2.2 Attention Module

The attention module has shown its super performances in many tasks in neural language processing (NLP) and computer vision (CV), and thus deserves further research. Take CV tasks as an example, the attention module computes the weights that reflect the similarity between different positions of an image, widely interacting different positions with each other. Thus, the model can use both global and local information more efficiently, significantly improving performance. Designing as a nonlinear module, it has a complex but clear structure. Specifically, we denote $\mathbf{X}$ the input of the module, and $\mathbf{W_Q}$, $\mathbf{W_K}$ and $\mathbf{W_V}$ to be the linear parameters of attention module. Then $\mathbf{F_Q} := \mathbf{W_Q X}$, $\mathbf{F_K} := \mathbf{W_K X}$ and $\mathbf{F_V} := \mathbf{W_V X}$ are the feature generated by the corresponding linear operator, and the attention matrix is defined by $\mathbf{A} = \mathbf{F_Q}^\top \mathbf{F_K}$. Denote $\mathcal{S}(\cdot)$ the softmax function in the module, then $\mathbf{A}_s := \mathcal{S}(\mathbf{A})$ is the feature that passed softmax function, and the whole output of the attention module is $\mathbf{X}_P := \mathbf{F_V A}_s$. Given this clear structure, it may be possible to study whether more information can be applied to its optimization.

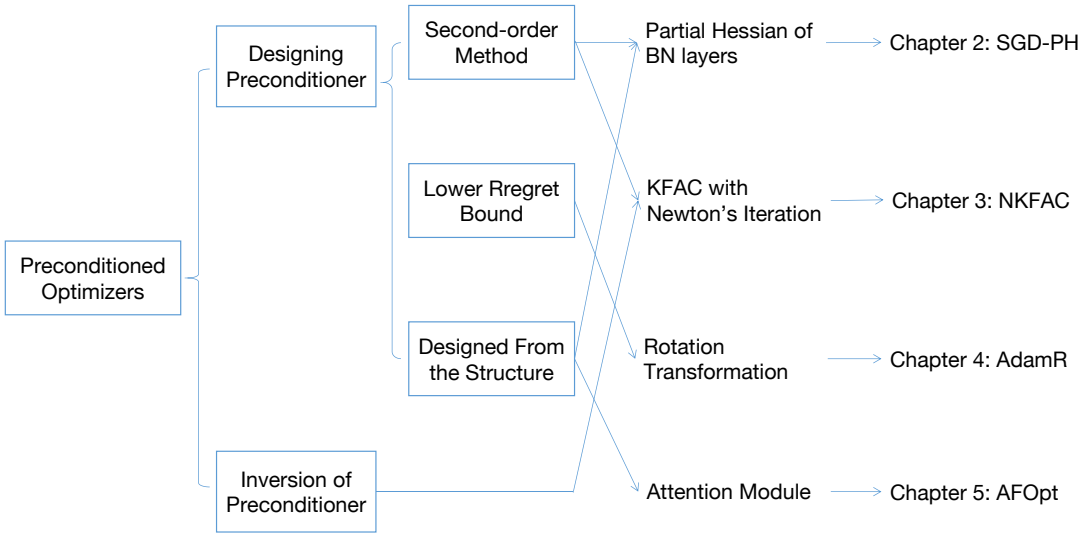


Figure 1.1: Contribution and organization of this thesis.

1.3 Contribution and Organization of Thesis

This thesis consists of four works about designing effective preconditioned optimizers for deep neural network training, and Fig. 1.1 illustrates our main contribution and organization. In this thesis, we mainly focus on the following aspects. First, we use second-order Hessian as a preconditioner for some channel-wise 1D parameters in neural networks and design a combined optimizer SGD-PH. Second, we deal with the inversion in the preconditioned optimizer KFAC and design a more computational-friendly optimizer NKFAC. Third, aiming at lowering online regret bound, we design an adaptive learning rate preconditioned optimizer with rotation transformation, named AdamR. Finally, utilizing the attention feature gradient descent, we design preconditioners to interact the parameters with each other and propose AFOpt.

The organization of this thesis is as follows.

In Chapter 1, we introduce the background of preconditioned optimizers and

discuss some unique structures of DNNs that are related to our design.

In Chapter 2, we introduce the optimizer SGD with Partial Hessian (SGD-PH). Recognizing the unique properties of channel-wise 1D parameters in normalization layers, we first demonstrate that their associated Hessian matrices are diagonal and can be extracted directly and precisely from Hessian-free methods. Then we propose a compound optimizer SGD-PH, which is a combination of a second-order optimizer with a precise partial Hessian matrix for updating channel-wise parameters and the first-order stochastic gradient descent (SGD) optimizer for updating the other parameters. This approach integrates the benefits of partial Hessian matrices to enhance optimization while maintaining the generalization performance of first-order optimizers .We evaluate the performance of SGD-PH through experiments on classification and person re-identification (reID) benchmarks to show its effectiveness.

In Chapter 3, we deal with the time-consuming inversion process in the Kronecker-factorized approximate curvature (KFAC) algorithm. We propose the Newton-Kronecker factorized approximate curvature (NKFAC) algorithm, which incorporates Newton's iteration method for inverting the second-order statistics. As the Fisher information matrix between adjacent iterations changes little, we can initialize Newton's iteration with the previous inverse computed. Thus, by the fast local convergence of Newton's iteration, we get accurate results within a few steps, which reduces computation time. We further enhance NKFAC with practical implementations, resulting in state-of-the-art generalization performance without tedious hyperparameter tuning. We demonstrate the efficacy of NKFAC through experiments on various computer vision tasks.

In Chapter 4, we propose the integration of a rotation transformation into the existing adaptive learning rate algorithms. We utilize the well-established AdamW optimizer as a foundation to develop a optimizer named AdamR. Theoretically, AdamR achieves a lower regret bound compared to other adaptive learning rate

methods that focus solely on the diagonal elements of the preconditioned matrix. An important feature of the rotation transformation in AdamR is its ability to preserve the gradient norm, enabling AdamR to seamlessly adopt the hyper-parameters and retain the advantages of AdamW. We have conducted extensive experiments on image classification, object detection, and segmentation tasks, where AdamR has demonstrated superior performance over existing methods.

In Chapter 5, we introduce the Attention-Feature-based Optimizer (AFOpt) specifically designed to optimize attention modules. AFOpt leverages the attention feature gradient descent to treat the attention module as a unified entity, promoting parameter interaction and enhancing training efficacy. Our approach first directly applies gradient descent to the output features of the attention module, then updates attention parameters by approximating the impact of the output feature's gradient descent. This parameter interaction facilitates the utilization of similarities between different patches and assists the optimization. We validate the effect of AFOpt through experiments on object detection and segmentation tasks.

In Chapter 6, we conclude the content of this thesis and discuss the future work.

1.4 Mathematical Notation System

In this content, we use $\mathbf{a}$ to represent a one dimensional (1D) vector and $\mathbf{A}$ a tensor more than one dimension. $\mathbb{R}$ represents the field of real numbers and $\mathbb{R}^n$ is the field of real numbers with n dimension. The notations $\mathbf{A} \succeq \mathbf{0}$ and $\mathbf{A} \succ \mathbf{0}$ for a matrix $\mathbf{A}$ denote that $\mathbf{A}$ is symmetric positive semidefinite (PSD) and symmetric positive definite, respectively. Furthermore, $\mathbf{A} \succeq \mathbf{B}$ or $\mathbf{A} - \mathbf{B} \succeq \mathbf{0}$ means that $\mathbf{A} - \mathbf{B}$ is PSD. $\mathrm{Tr}(\mathbf{A})$ represents the trace of the matrix $\mathbf{A}$. For a PSD matrix $\mathbf{A}$, $\mathbf{A}^\alpha = \mathbf{U}\mathbf{\Sigma}^\alpha\mathbf{U}^\top$, where $\mathbf{U}\mathbf{\Sigma}\mathbf{U}^T$ is the Singular Value Decomposition (SVD) of $\mathbf{A}$.

$||\mathbf{x}||_{\mathbf{A}} = \sqrt{\mathbf{x}^T\mathbf{A}\mathbf{x}}$ is the Mahalanobis norm of $\mathbf{x}$ induced by PSD matrix $\mathbf{A}$, and its dual norm is $||\mathbf{x}||^*_{\mathbf{A}} = \sqrt{\mathbf{x}^T\mathbf{A}^{-1}\mathbf{x}}$. $\mathbf{A} \otimes \mathbf{B}$ means the Kronecker product of $\mathbf{A}$ and $\mathbf{B}$, while $\mathbf{A} \odot \mathbf{B}$, $\mathbf{A}^{\odot\alpha}$ are the element-wise matrix product and element-wise power operation, respectively. Diag($\mathbf{x}$) is a diagonal matrix with diagonal vector $\mathbf{x}$, and vec($\cdot$) denotes the vectorization function. Other specific notations related to model and data will be declared in each chapter within consistency.

Chapter 2

SGD with Partial Hessian for Deep Neural Networks Optimization

Due to the effectiveness of second-order algorithms in solving classical optimization problems, designing second-order optimizers to train deep neural networks (DNNs) has attracted much research interest in recent years. However, because of the very high dimension of intermediate features in DNNs, it is difficult to directly compute and store the Hessian matrix for network optimization. Most of the previous second-order methods approximate the Hessian information imprecisely, resulting in unstable performance. In this work, we propose a compound optimizer, which is a combination of a second-order optimizer with a precise partial Hessian matrix for updating channel-wise parameters and the first-order stochastic gradient descent (SGD) optimizer for updating the other parameters. We show that the associated Hessian matrices of channel-wise parameters are diagonal and can be extracted directly and precisely from Hessian-free methods. The proposed method, namely SGD with Partial Hessian (SGD-PH), inherits the advantages of both first-order and second-order optimizers. Compared with first-order optimizers, it adopts a certain amount of information from the Hessian matrix to assist optimization, while compared with the existing second-order optimizers, it keeps the good generalization performance of first-order optimizers. Experiments on image classification and person

re-identification tasks demonstrate the effectiveness of our proposed optimizer SGDPH. The code is publicly available at `https://github.com/myingysun/SGDPH`.

2.1 Introduction

Owing to the back-propagation algorithm, the first-order local information (e.g., the gradients) of the loss function for deep neural networks (DNNs) can be obtained easily at a reasonable cost, which greatly assists the successful development of the first-order optimizers for training DNNs. Among the existing optimizers, the most widely used algorithms are SGD with momentum (SGDM) [60] and ADAM [36], the descent directions of which are only decided by the gradients. Because of the advantages including simple implementation, low computational cost and small memory consumption, first-order optimizers [60, 36, 101] occupy the mainstream in the optimization of DNNs. Live up to expectations, they have been shown the efficiency on a variety of tasks in computer vision [63, 28], natural language processing [49] and other machine learning areas (e.g., [54, 74]).

Besides first-order information, the second-order information (e.g., Hessian) is admittedly considered to help the optimization. Sometimes, second-order algorithms will be preferred in solving traditional optimization problems due to their faster (local) convergence under some assumptions and higher accuracy with fewer iterations. Hence, the generalizations of the second-order methods into DNNs optimization are always under the research spotlight over the past few years. However, since the dimension of Hessian in DNNs is very high, it is difficult to directly store and compute the Hessian matrix. Therefore, how to make it practical in deep learning is still an open question. A lot of works focus on how to approach the second-order information in DNNs. For example, some earlier works [51, 88, 50] apply the Hessian-free methods to approximate the Hessian matrix and then embed this information in the optimiza-

tion process. Limited by the ways of implementation, most of these methods extract the imprecise second-order information, and the accuracy of these approximations may be considered to affect the performance of the optimizer. Instead of the Hessian matrix, the natural gradient matrix can also be considered as the second-order information to be approximated, for example, KFAC [24] and EKFAC [22] methods approximate the natural gradient layerwisely by using a block-diagonal of the Fisher matrix. However, these methods are still based on some assumptions about the statistical properties of the parameter distributions, which may also bring inaccuracies into the optimization process. Due to such factors, sometimes the performances of the second-order methods are even worse than the first-order ones, which may also hinder the practical application of second-order optimizers.

It is worth noting that the existing optimizers treat all the network parameters "the same", in other words, all the parameters in the network follow the same updating rule in the training process. However, we notice that the channel-wise one-dimensional (1D) parameters are commonly introduced in some popular basic modules of DNNs (e.g., in sundry normalization layers). From the perspective of designing optimizers, we find these channel-wise 1D parameters can be proved to have an important property, i.e., the Hessian matrix related to any one group of channel-wised 1D parameters is diagonal (see the derivation in Section 2.3.1). The dimension of these parameters, which is exactly the number of output channels of this layer, is usually a very small number (e.g., 64), and for these diagonal Hessian matrices, we can obtain the diagonals directly and precisely by the Hessian-free approach (see Section 2.3.2 for details). This conclusion motivates us to treat the parameters "differently" and propose a new algorithm to combine the first-order methods with the second-order ones.

In this chapter, we propose a new type of compound optimizer, named **SGD** with **P**artial **H**essian (SGD-PH), which combines the second-order optimizer for

the channel-wise 1D parameters with the first-order optimizer for other parameters. Through making use of the special structure we mentioned above, we are allowed to use Newton-type methods to update parameters for some specific layers, such as the batch normalization (BN) layer and the convolutional layer with weight normalization (WN). Compared with first-order optimizers, SGD-PH adopts partial but precise information from the Hessian matrix to help optimization, while compared with other second-order optimizers, it can keep and even surpass the high generalization performance of first-order optimizers in many tasks. Numerical experiments on different datasets are given to illustrate the effectiveness of our SGD-PH.

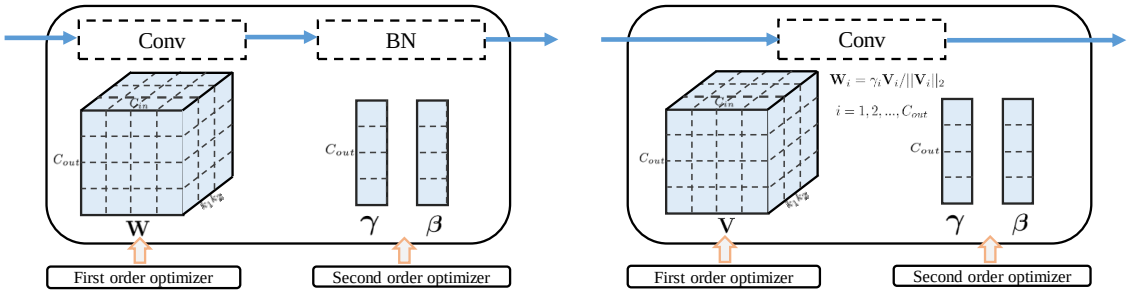
2.2 Related Work

When employing second-order methods for DNNs, an essential part is to consider how to extract the Hessian information efficiently and precisely. In this section, we list some Hessian-free methods for solving the descent directions. Besides, we also give a brief introduction of several commonly used normalization methods that inspire our design.

Hessian-free Approaches: Hessian-free methods are intuitive and efficient in solving the descent direction when applying the Newton-type methods in DNNs optimization. Owing to the advantages of saving storage space, they are super suitable for deep networks and large-scale tensors. One kind of the Hessian-free method approximates the diagonal Hessian elements via back-propagation, e.g., the implementation process in Adahessian [88] and Apollo [50]. Specifically, Adahessian applies Hutchinson's method as an inexact approximation with the help of Rademacher distribution, and Apollo updates the descent direction by the quasi-Newton method with the help of the weak secant equation, in which the weak secant equation provides a reasonable diagonal approximation of the next Hessian matrix. Another kind is a

generalization of the classical iterative methods, including the Gauss-Seidel method, the (preconditioned) conjugate gradient (CG) method, and the generalized minimal residual (GMRES) method. These well-known methods can iteratively solve the linear system without storing the whole Hessian matrix, and have also been embedded into the neural network training pipeline, e.g. [9, 51].

Normalization Methods: In DNNs, there are many specific layers designed to be channel-wise, which enables some good properties for designing a partial Hessian optimizer. The most intuitive examples are the batch normalization (BN)[33] and weight normalization (WN)[70]. BN is a usually adopted technique in high-level tasks training. In BN layers, a pair of parameters (γ, β) is introduced to scale and shift the normalized value by $y_i = \gamma\hat{x}_i + \beta$ for each channel, such that the effect of noise will be reduced. By adding BN, the performance of many DNNs become more robust to the change of hyperparameter values. Differently, WN is a technique that decouples the length and the direction of the weight parameters, i.e., in mathematics, let $\mathbf{w} = \gamma\frac{\mathbf{v}}{\|\mathbf{v}\|}$. By applying the WN method, we are able to decouple the weight length γ and the direction tensor $\mathbf{v}$ from the weight $\mathbf{w}$. Without calculating statistics from mini-batches like BN, WN is more suitable for some specific applications such as deep reinforcement learning or generative models. Numerous experiments have confirmed that they can both speed up the training process and improve the final generalization performance. Besides WN and BN, there are also other feature normalization methods such as layer normalization (LN) [4], group normalization (GN) [86] and instance normalization (IN) [82]. Although these normalization layers are simple affine layers, they have a great favorable impact on training deep neural networks, which makes them become popular basic modules in DNNs.



(a) Conv+BN (b) Conv+WN

Figure 2.1: Illustration of SGD-PH. Here, Figure (a) represents the case when BN is adopted in neural networks training, while other normalization methods such as LN, GN and IN can be represented in the same way. Figure (b) illustrates the case of decoupling the convolutional layers when there are no normalization layers followed, where $\boldsymbol{\beta}$ represents bias, please see Section 2.3.4 for more details.

2.3 Methodology

Instead of using the overall first-order or second-order information, we design the descent steps with different order information at different layers, which is the reason we called our method a "partial Hessian" method. Here, based on the widely applied first-order optimizer SGDM, we add the precise Hessian information when optimizing the 1D parameters. Moreover, for DNNs that have no normalization layers, we can decouple the convolutional layers into a convolution operation and a linear weight normalization operation, which enables our optimizer to be applied. Here we use Figure 2.1 to illustrate our idea.

2.3.1 Diagonal Hessian Matrix

For an intuitive explanation, we adopt the BN layer as an example to describe the design of SGD-PH. For a BN layer, the 1D-parameters are formed by different values of parameters related to different channels. First we recall the operations of a single BN layer with C input channels. For a batch of input $\mathcal{X} \in \mathbb{R}^{N\times C\times W\times H}$ to this layer, divide it into C sets $\mathcal{X}_1, \ldots, \mathcal{X}_C$ according to the channels, where $\mathcal{X}_i = \{x_i^1, \cdots, x_i^{m_i}\}$

for each channel index $i \in \{1, 2, \ldots, C\}$ and $m_i = NWH$ represents the number of the elements within the i-th channel. Then for a specific channel $\mathcal{X}_i$, the BN layer contains the following operations

$$\mu_{\mathcal{X}_i} = \frac{1}{m_i}\sum_{j=1}^{m_i} x_i^j;\ \sigma^2_{\mathcal{X}_i} = \frac{1}{m_i}\sum_{j=1}^{m_i}(x_i^j - \mu_{\mathcal{X}_i})^2;\ \hat{x}_i^j = \frac{x_i^j - \mu_{\mathcal{X}_i}}{\sqrt{\sigma^2_{\mathcal{X}_i} + \epsilon}};\ y_i^j = \gamma_i \hat{x}_i^j + \beta_i. \quad (2.1)$$

The parameters γ_i, β_i introduced here are usually updated via back-propagation in the training process. Notice that the BN layer actually normalizes each channel independently, which means the parameters γ_i, β_i of $\mathcal{X}_i$ are irrelevant to γ_j, β_j of $\mathcal{X}_j$ whenever $j \neq i$. Thus, denote $\mathbf{\Gamma} = (\gamma_1, \cdots, \gamma_C)^T$ as the group of parameters γ_i for all the C channels, the Hessian with respect to $\mathbf{\Gamma}$, denoted by $\mathbf{H}_{\mathbf{\Gamma}}$ here, is exactly a diagonal matrix. i.e.,

$$\mathbf{H}_{\mathbf{\Gamma}} := \frac{\partial^2 \mathcal{L}}{\partial \mathbf{\Gamma}^2} = \mathrm{Diag}\left(\frac{\partial^2 \mathcal{L}}{\partial \gamma_1^2}, \cdots, \frac{\partial^2 \mathcal{L}}{\partial \gamma_C^2}\right) \quad (2.2)$$

with $\frac{\partial^2 \mathcal{L}}{\partial \gamma_i^2} = \sum_{j=1}^{m_i} \frac{\partial^2 \mathcal{L}}{\partial (y_i^j)^2}(\hat{x}_i^j)^2$ for $i = 1, \ldots, C$. Then the inverse $\mathbf{H}_{\mathbf{\Gamma}}^{-1}$, if exists, can be computed directly by the inverse of each diagonal element. This structure states that getting the precise diagonal elements of the Hessian with respect to $\mathbf{\Gamma}$ is equivalent to obtaining the partial Hessian $\mathbf{H}_{\mathbf{\Gamma}}$ precisely, which enables us to apply Newton-type methods easily. The same conclusion also holds for the Hessian of the group of parameters $\boldsymbol{\beta} = (\beta_1, \cdots, \beta_C)^T$. Consequently, we can obtain the exact Hessian matrices of such one-dimensional parameters directly instead of using any approximation method.

It is worth mentioning that the property of partial diagonal Hessian does not hold for nature gradient descent methods (that is, the idea of extracting precise partial diagonal Hessian cannot be generalized to optimizers like KFAC), since the Fisher matrices related to 1D parameters are symmetric positive semidefinite matrices $\mathbb{E}[\mathbf{g}\mathbf{g}^\top]$ with $\mathbf{g}$ being the gradient vector, which cannot be proved to be diagonal.

2.3.2 The Hessian-free Approach

As we mentioned in Section 2.2, the Hessian-free method can be regarded as a necessary way in designing second-order optimizers due to the memory limitations, and some existing optimizers have adopted Hessian-free methods to approximate the whole Hessian diagonal. Under this situation, it seems that there is no need to extract the diagonal of some specific layers. However, regarding the truth that the whole Hessian is not diagonal, the inexactness brought by approximating the whole diagonal may influence the optimization process, which may lead to the unsatisfactory performance of the second-order optimizers on some tasks. As a consequence, by extracting the specific elements, we can avoid the inexactness of the existed diagonal approximation methods, which helps our optimization.

In our optimizer, we adopt the Hessian-free approach with the help of back-propagation. To ensure continuity, we still take the BN layer as an example. Under the diagonal Hessian analysis Eq.(2.2), we set $\mathbf{e_\Gamma}$ to be the vector that takes elements 1 related to the 1D vector $\mathbf{\Gamma}$ and takes elements 0 at all other places. Then with the help of the second time back-propagation, we can get

$$\mathbf{H}\mathbf{e_\Gamma} = \frac{\partial(\mathbf{g}^T\mathbf{e_\Gamma})}{\partial\mathbf{\Gamma}}, \tag{2.3}$$

where $\mathbf{g}$ is the gradient information computed in the first back-propagation process. Then the vector $\mathrm{diag}(\mathbf{H_\Gamma})$ is contained in the corresponding positions of the vector $\mathbf{H}\mathbf{e_\Gamma}$. By this Hessian-free approach, we can get the corresponding diagonal matrix $\mathbf{H_\Gamma}$ precisely without computing and storing the whole Hessian matrix. Hence, our method avoids the inaccuracy brought by the iterative methods or the approximation methods when computing the Hessian information. The analysis of the group of bias parameters $\boldsymbol{\beta}$ in BN layers is the same as $\mathbf{\Gamma}$. Apart from this, other normalization methods, including LN, IN and GN, also have analogous parameters, so similar derivation and conclusion can be obtained.

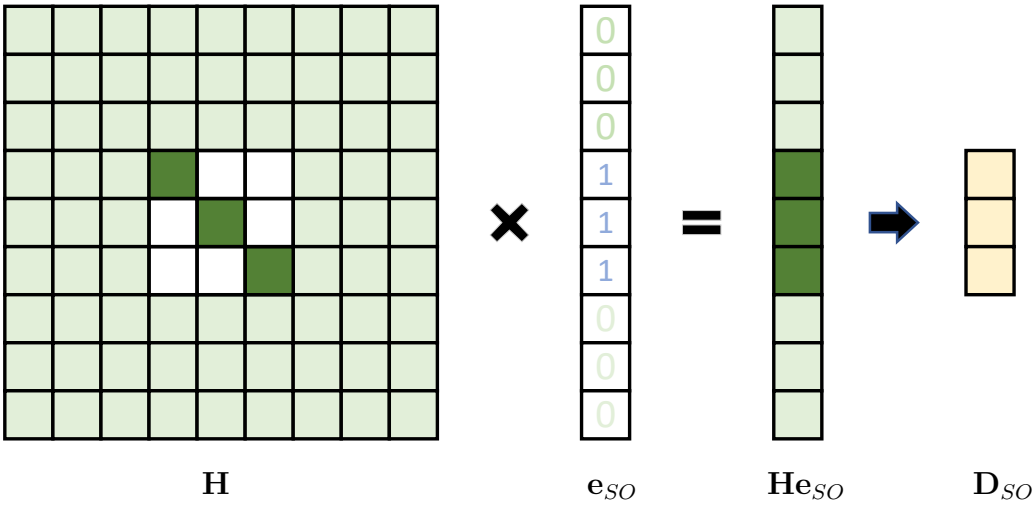


Figure 2.2: Illustration of diagonal Hessian computation. Here, the light green boxes represent the elements that can be any real numbers while the white boxes represent zeros. The central 3×3 matrix (i.e., $\mathbf{H}_{SO}$) in $\mathbf{H}$ is diagonal corresponding to the specific 1D variable. By multiplies with the vector $\mathbf{e}_{SO}$, we can extract the diagonal elements precisely in the middle of $\mathbf{He}_{SO}$ and compute the element-wise inverse to get $\mathbf{D}_{SO}$, which is exactly the diagonal of $\mathbf{H}_{SO}^{-1}$.

Generally, for an arbitrary 1D variable in DNN, we denote $\mathbf{e}_{SO}$ the vector takes 1 at positions corresponding to this 1D vector and 0 at all other positions, and denote the partial diagonal Hessian matrix and the descent direction concerning it by $\mathbf{H}_{SO}$ and $\mathbf{D}_{SO}$, respectively. Figure 2.2 illustrate the idea of our precise diagonal Hessian computation. Through this operation, we can obtain the precise partial Hessian for the calculation of descent direction.

Here, although we have formed a preconditioner on BN layer, unlike the existing work [40] that design a preconditioner to lower condition number and get better linear convergence rate, we design the partial Hessian trying to enjoy the second-order local convergence rate, which is different from the existing method.

2.3.3 Techniques in Nonconvex Optimization

Since a nonconvex objective function may not have a positive-definite Hessian, to apply the Newton method in solving nonconvex stochastic optimization problems,

several rectification techniques are usually adopted to guarantee the well-definiteness and the effectiveness of the Newton method.

First, to handle the nonconvexity and noninvertible issues, we use the absolute value matrix with a positive perturbation as the substitution of the partial Hessian matrix, i.e.,

$$\tilde{\mathbf{H}}_{SO} := \sqrt{\mathbf{H}_{SO}^{\top}\mathbf{H}_{SO}} + \epsilon\mathbf{I} \tag{2.4}$$

with $\epsilon > 0$ a small enough positive real number. Moreover, at the t-th iteration, to minimize the effect of randomness and noise, we apply a momentum step on the Hessian information $\tilde{\mathbf{H}}_{SO}$, i.e., for some $\alpha \in (0, 1)$, the momentum $\mathbf{M}_H$ is computed by

$$\mathbf{M}_H{}^t = (1-\alpha)\mathbf{M}_H{}^{t-1} + \alpha\tilde{\mathbf{H}}_{SO}^t, \tag{2.5}$$

and the inverse of partial Hessian $\mathbf{D}_{SO}{}^t = (\mathrm{diag}(\mathbf{M}_H{}^t))^{-1}$ is computed directly by taking the diagonal element-wise inverse. At each iteration t, we only calculate the partial Hessian and the momentum related to the current 1D parameter. The techniques we mentioned above are some generally used techniques that can also be found in many other papers about second-order optimizers, e.g., [88, 50].

Here, as applied in SGD, we also apply the momentum [60] calculation on the gradient and the weight decay [39] technique for the final descent direction in our optimizer to accelerate the convergence. Specifically, the momentum step on the gradient is

$$\mathbf{M}_G{}^t = (1-\beta)\mathbf{M}_G{}^{t-1} + \beta\mathbf{G}^t, \tag{2.6}$$

where $\beta \in (0, 1)$ is a given constant and $\mathbf{G}$ is the gradient of the parameters. Meanwhile, after computing the final descent direction $\mathbf{D}_G$ (specifically, taking $\mathbf{D}_{SO}\mathbf{M}_G$ for 1D parameters and $\mathbf{M}_G$ for the others), for a given weight decay parameter η and the weight tensor $\mathbf{W}$, the weight decay step is exactly

$$\tilde{\mathbf{D}}_G^t = \mathbf{D}_G{}^t + \eta\mathbf{W}^t. \tag{2.7}$$

Algorithm 1 SGD with partial Hessian (SGD-PH)

Inputs: Initial weight vector $\mathbf{W}^0$, step size τ and τ_{SO}, momentum factor α and β, weight decay parameter η, rectification parameter ϵ.

Outputs: $\mathbf{W}^{(T)}$.

for $t = 1, \ldots, T$ **do**
  get $\mathbf{G}^t = \nabla L^t(\mathbf{W}^t)$;
  if $\mathbf{W}^t$ is a channel-wised 1D parameter **then**
    compute ${\mathbf{H}_{SO}}^t$ and $\tilde{\mathbf{H}}_{SO}^t$ by Eq.(2.3) and Eq.(2.4), respectively;
    update ${\mathbf{M}_H}^t$ by Eq.(2.5) and compute diagonal element-wise inverse ${\mathbf{D}_{SO}}^t$;
    update ${\mathbf{M}_G}^t$ by Eq.(2.6);
    compute ${\mathbf{D}_G}^t = \tau_{SO}{\mathbf{D}_{SO}}^t{\mathbf{M}_G}^t$;
  else
    update ${\mathbf{M}_G}^t$ by Eq.(2.6);
    give ${\mathbf{D}_G}^t = {\mathbf{M}_G}^t$;
  end if
  get $\tilde{\mathbf{D}}_G^t$ by Eq.(2.7);
  update $\mathbf{W}^{t+1} = \mathbf{W}^t - \tau\tilde{\mathbf{D}}_G^t$.
end for

With the techniques in this section, we are ready to introduce the structure of our optimizer SGD with Partial Hessian (SGD-PH) in **Algorithm 1**.

2.3.4 Generalizations on Convolutional Layers

In the previous sections, we explained how to apply SGD-PH to DNNs with normalization layers. In this section, we will suggest that our optimizer can also be applied in training DNNs without normalization layers in the same way of weight normalization (WN). Here we illustrate the derivation via a single 2D convolution operation $\mathbf{Y} = \mathbf{W} * \mathbf{X}$, where $\mathbf{W}$ is the kernel with dimension $C_{out} \times C_{in} \times k_1 \times k_2$ and $\mathbf{X}$ is the input. For each output channel $i \in \{1, \ldots, C_{out}\}$, the WN operation can be defined as follows

$$\mathbf{W}_i = \gamma_i \frac{\mathbf{V}_i}{\|\mathbf{V}_i\|_2}, \quad \text{where } \mathbf{V}_i \in \mathbb{R}^{C_{in} \times k_1 \times k_2} \text{ and } \gamma_i \in \mathbb{R}. \tag{2.8}$$

Overall, we can get $\mathbf{V} = (\mathbf{V}_1, \ldots, \mathbf{V}_{C_{out}})$ and $\mathbf{\Gamma} = (\gamma_1, \ldots, \gamma_{C_{out}})$. Thus, the parameters to be optimized change from the initial parameter $\mathbf{W}$ to the same dimension

parameter $\mathbf{V}$ and a channel-wised 1D parameter $\mathbf{\Gamma}$. Similar to the parameters in BN, our proposed optimization algorithm can also be adopted to optimize the parameter $\mathbf{\Gamma}$ with its second-order information. Moreover, the bias parameter $\boldsymbol{\beta}$ in the convolutional layers can also be contained into the second-order part optimization of SGD-PH (we omit the details since it is trivial). We will demonstrate the efficiency of SGD-PH in this generalization case later by the experiments in Section 2.4.4.

2.4 Experiments

In this section, we will validate the robustness and effectiveness of our proposed SGD-PH by comprehensive experiments on image classification tasks. In our experiments, we compare our SGD-PH with first-order optimizers SGDM [60], Adam [36], AdamW [47], Adabelief [101], together with second-order optimizers Adahessian [88] and Apollo [50]. In Section 2.4.1, we accomplish our experiments on deep neural networks VGG11, VGG19 [77], ResNet18 and ResNet50 [29] for datasets CIFAR10 and CIFAR100 [38], and in Section 2.4.2, we compare the performances of different optimizers on ResNet18 and ResNet50 for dataset Mini-ImageNet [84]. To avoid randomness, these experiments are repeated 4 times and the results of testing accuracies are reported in the "mean $\pm$ std" format. Moreover, we report the performance of SGD-PH for the large-scale dataset ImageNet [68] on ResNet18 in Section 2.4.3. In Section 2.4.4, we provide an illustration of the generalization of SGD-PH on convolutional layers by the network VGG19. More ablation studies results of SGD-PH can be found in Section 2.4.6.

Experiments Setup: In SGD-PH, the hyperparameters α and β control the momentum for the partial Hessian matrix and the gradient, respectively. Following the experience from SGDM, we set $\alpha = 0.9$ and $\beta = 0.9$ in our experiments. The second-order learning rate τ_{SO} in line 8, Algorithm 1 is set to be 0.001. Besides,

we add a small positive number $\epsilon = 0.0001$ to the Hessian diagonal to avoid the diagonal elements being zero. The batch size, learning rate, weight decay, along with the GPUs we use, will be introduced in each section respectively.

2.4.1 Results on CIFAR10/CIFAR100

CIFAR10 and CIFAR100 are commonly used image classification datasets, in which CIFAR10 contains 10 classes of color images for classification with 6000 images per class, and CIFAR100 contains 100 classes images with 600 images per class. Our experiments here are accomplished on Pytorch 1.7 framework with mainly TITAN RTX and Geforce RTX 2080Ti GPUs. In our experiments, we train DNNs with different optimizers for total 200 epochs, using batch size 128 with one single GPU, and the learning rate is multiplied by 0.1 every 60 epochs.

Compared with Other Optimizers: We compare SGD-PH with four popular first-order optimizers SGDM, Adam, AdamW and Adabelief, and two second-order optimizers Adahessian and Apollo, on four representative DNNs, i.e., ResNet18, ResNet50, VGG11 and VGG19. We tune the learning rate and weight decay for all these methods, and choose their best results for comparison. Specifically, the learning rates are set to be 0.1 for SGDM, 0.001 for Adam, AdamW and Adabelief, 0.15 for Adahessian and 1 for Apollo. the weight decays are set to be 0.0005 for SGDM, Adam and Adahessian, 0.5 for AdamW and Adabelief, and 0.00025 for Apollo. For SGD-PH, the learning rate takes 0.01 and the weight decay takes 0.005.

Table 2.1 shows the testing accuracies with these optimizers on CIFAR100 and CIFAR10. It can be seen from the results that SGD-PH achieves the best results of all DNN models on CIFAR100. Specifically, SGD-PH surpasses other compared methods largely. The performance of SGD-PH surpasses SGD from $0.76\% \sim 2.93\%$ on CIFAR100, which fully indicates that the precise partial Hessian can help improve

the final performance. For CIFAR10, we notice that many optimizers(e.g., SGDM, Adabelief, Apollo and SGD-PH) reach 100% training accuracy with loss near to zero, so most of them show quite similar generalization performance as in Table 2.1 and the final stage in Figure 2.3. In the meantime, we also find that the performance of Adahessian sometimes is not stable, e.g., its performance of VGG on CIFAR100 drops largely. The imprecise Hessian adopted in Adahessian sometimes may have very bad impacts on the optimization process.

Meanwhile, for more intuitive insight, we present the testing accuracy curves of different optimizers on CIFAR100/10 during training different networks in Figure 2.3. From the trajectories, we see that the performance of Adahessian seems not to gain too much from the learning rate decay at the 120-th epoch. Moreover, affected by the proportion of 1D variables, the trend of the training curves of SGD-PH are usually closer to the first-order optimizers. However, due to the absorption of second-order information, the performance of SGD-PH exists almost always above that of SGDM after the first learning rate decay at the 60-th epoch. This helps to illustrate that SGD-PH may aggregate the advantages of first-order and second-order methods.

Time and Memory Cost: To extract the precise partial Hessian information without approximation, the Hessian free method via the second time back-propagation is the best approach for SGD-PH. In Adahessian, the same technique has also been applied, and sadly the time consumption and the storage cost increase exaggeratedly (e.g., for ResNet18 on CIFAR100, Adahessian has 4.64 times increase in time and 1.30 times increase in storage compared to SGDM), which may result in a bad impact on its comprehensive applications. By noticing this fact, we have optimized the calculation process in our Hessian-free approach due to our special structure to shorten the time and storage cost of the second time back-propagation. Although we still have increments of 2.22 and 1.23 times in time and storage respectively compared to

Table 2.1: Testing accuracies (%) of different DNNs with different optimizers on CIFAR100/10 datasets.

CIFAR100							
Optimizer	SGDM	Adam	AdamW	Adabelief	Adahessian	Apollo	SGD-PH
ResNet18	77.20 ± .30	72.95 ± .20	77.23 ± .10	77.43 ± .36	76.73 ± .23	76.63 ± .27	**77.96 ± .30**
ResNet50	77.78 ± .43	72.13 ± .53	78.10 ± .17	79.08 ± .23	78.48 ± .22	78.68 ± .11	**79.54 ± .27**
VGG11	70.80 ± .29	68.00 ± .21	71.20 ± .29	72.43 ± .16	67.78 ± .34	70.05 ± .11	**72.75 ± .13**
VGG19	70.94 ± .32	63.90 ± 1.62	70.26 ± .23	72.37 ± .19	69.93 ± .84	71.46 ± .52	**73.87 ± .28**
CIFAR10							
ResNet18	95.10 ± .07	92.95 ± .25	94.80 ± .10	**95.12 ± .14**	94.70 ± .15	95.03 ± .12	95.04 ± .07
ResNet50	94.75 ± .30	92.62 ± .19	94.72 ± .10	**95.35 ± .05**	**95.35 ± .11**	95.27 ± .11	95.22 ± .07
VGG11	92.17 ± .19	90.75 ± .15	92.02 ± .08	92.45 ± .18	91.85 ± .16	92.38 ± .19	**92.60 ± .08**
VGG19	93.57 ± .13	92.19 ± .07	93.54 ± .28	93.72 ± .08	93.68 ± .14	**93.76 ± .07**	93.75 ± .10

Table 2.2: Testing accuracies (%) of DNNs with different optimizers for ResNet18/50 on Mini-ImageNet dataset.

Optimizer	SGDM	Adam	AdamW	Adabelief	Adahessian	Apollo	SGD-PH
ResNet18	67.33 ± .17	66.47 ± .34	66.90 ± .36	67.98 ± .29	67.13 ± .40	68.06 ± .38	**70.53 ± .32**
ResNet50	67.09 ± .56	65.70 ± .60	68.23 ± .30	69.36 ± .13	70.25 ± .28	70.24 ± .28	**72.17 ± .30**

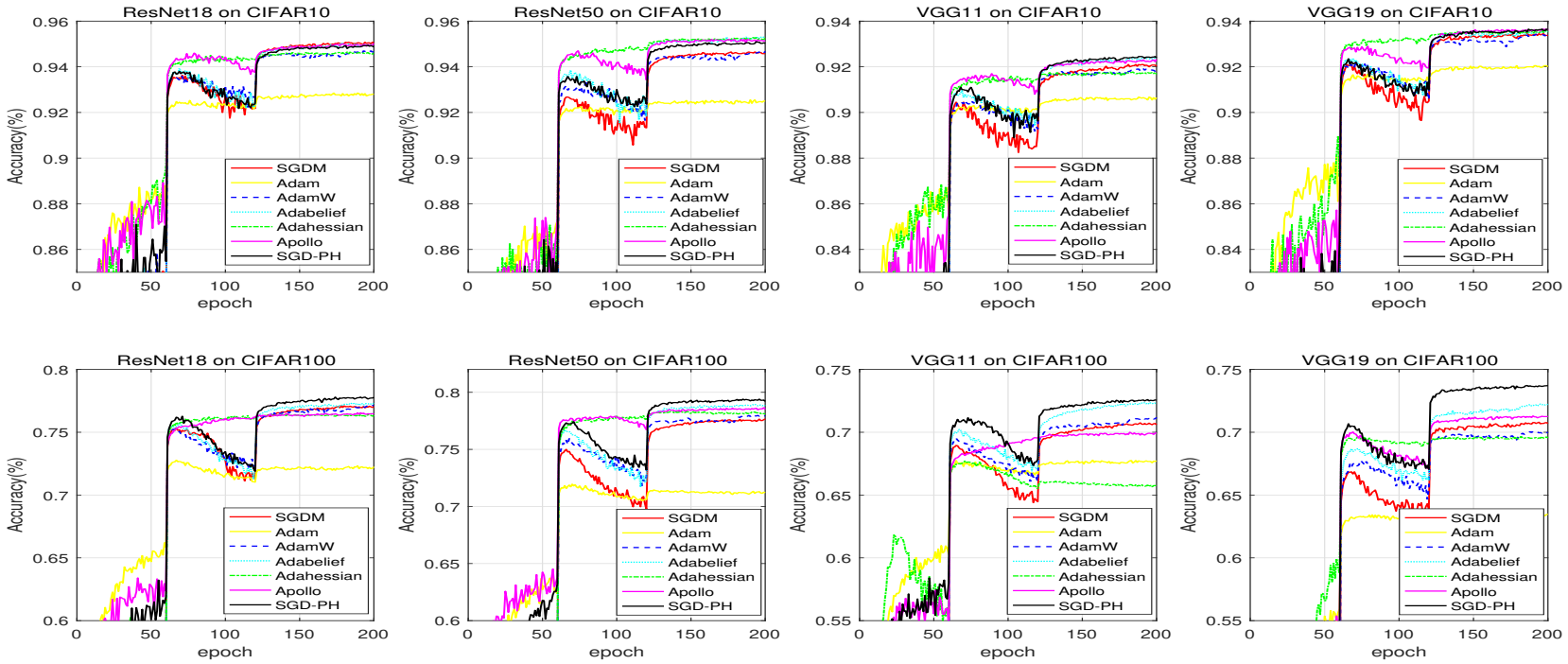


Figure 2.3: Testing accuracy curves of different optimizers for different DNNs on CIFAR100 and CIFAR10 datasets.

SGDM due to the properties of back-propagation, we have saved the time and storage a lot compared to Adahessian that adopting the same technique. However, considering the time and storage cost of SGD-PH than SGDM, one may benefit more from SGD-PH when the 1D parameter has a greater impact on the overall model and task performance.

2.4.2 Results on Mini-ImageNet

Mini-ImageNet is a subset of the well-known dataset ImageNet [68]. In our tests, we use the train/test sets splits provided by [62, 34, 91]. Mini-ImageNet consists of 100 classes and each class has 500 images for training and 100 images for testing. The image resolution is 84×84, and here we resize the images into 224×224 (i.e., the standard ImageNet training input size). We train the ResNet18 and ResNet50 with different optimizers for total 100 epochs with batch size 128 on 4 GPUs. Same as CIFAR100 and CIFAR10 datasets, we repeat each experiment for 4 times to avoid randomness, and the results are reported in the mean $\pm$ std format. The learning rate is multiplied by 0.1 every 30 epochs. For SGD-PH, we set the learning rate to be 0.05 and the weight decay to be 0.001. For other optimizers, the learning rates are the same as the settings in CIFAR100/10. The weight decay parameter is set to be 0.0001 for SGDM, Adam, and Apollo, and 0.1 for AdamW and Adabelief, 0.0005 for Adahessian. Table 2.2 shows the testing accuracies with these optimizers on Mini-ImageNet, and Fig. 2.4 presents their testing accuracy curves during training. We can see that SGD-PH outperforms other compared methods by a large margin, i.e., 2.47% and 1.92% performance gains on ResNet18 and ResNet50, respectively. Meanwhile, on mini-ImageNet dataset, the second-order optimizers usually perform better than the first-order optimizers. Both Adahessian and Apollo achieve favorable generalization performance. Under such cases, SGD-PH can keep the advantages of second-order optimizers, and even further improve their performance.

Table 2.3: Testing accuracies (%) of DNNs with different optimizers on ImageNet. The result of ADAM, AdamW, Adabelief and Adahessian are cited from [45], [10], [101], and [88], respectively.

Optimizer	SGDM	Adam	AdamW	Adabelief	Adahessian	Apollo	SGD-PH
Accuracy	70.49	66.54	67.93	70.08	70.08	70.39	**70.59**

2.4.3 Results on ImageNet

In this section, we report the results on ImageNet [68] to validate the effectiveness of SGD-PH. ImageNet is a large image classification dataset that contains 1000 categories with 1.28 million images for training and 50K images for validation. Our experiments on ImageNet are accomplished on Pytorch 1.7 framework with four GeForce RTX 2080Ti GPUs. In our experiments, we train the network ResNet18 [29] for total 100 epochs with batch size 256 on 4 GPUs, and the learning rate is multiplied by 0.1 every 30 epochs. We test the performance of SGD-PH, SGDM and Apollo, while we cite the performance of other optimizers from existing papers in Table 2.3 for a more intuitive and complete comparison. We set the initial learning rate to be 0.1 for SGD-PH and SGDM, and 1 for Apollo, and set the weight decay to be 0.0001 for all SGD-PH, SGDM and Apollo. The rest hyperparameters all follow their official settings.

Table 2.3 shows the testing accuracy of these optimizers on ImageNet. Among the experiments on ImageNet reported in the existing papers, the first-order optimizer SGDM usually has a favorable performance, and the generalization ability is often not inferior to other compared optimizers, which means these optimizers sometimes may not perform stably on the large scale dataset. Meanwhile, our SGD-PH, under the same initial learning rate and weight decay with SGDM, can maintain this stability of performance by 0.1% performance gain compared with SGDM. This is also a reflection of the fact that our SGD-PH can inherit the advantages of the first-order optimizers, which states the capability and universality of the newly proposed SGD-

Table 2.4: Testing accuracies (%) of VGG19 with/without normalization layers on CIFAR100 dataset.

Optimizer	SGDM	SGDM+WN	SGD-PH+WN	SGDM+BN	SGD-PH+BN
Accuracy	65.56 ± .41	65.98 ± .35	67.93 ± .35	70.94 ± .32	**73.87 ± .28**

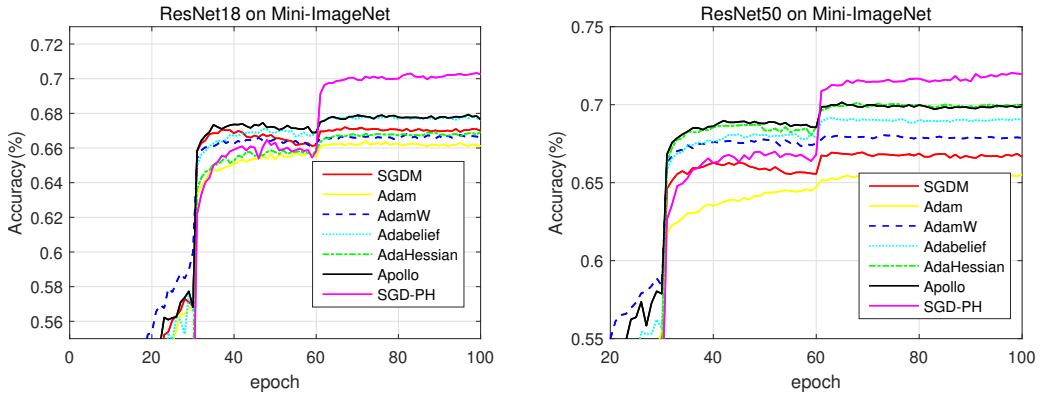


Figure 2.4: Testing accuracy curves of different optimizers on Mini-ImageNet dataset.

PH for training DNNs on large-scale datasets.

2.4.4 Results about Convolutional Layers

For DNNs without normalization layers, we can apply SGD-PH by adopting WN to embed the 1D parameters. As we introduced in Figure 2.2 (b), we reformulate the convolution operation as the composition of a linear operator (corresponding to

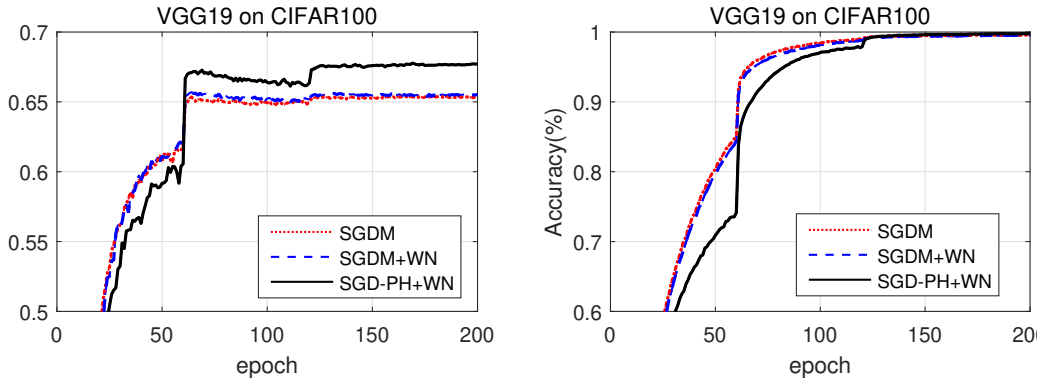


Figure 2.5: Testing and training accuracy curves of VGG19 without BN layers.

the length) with a convolution operator (corresponding to the direction) as is often applied in WN. Here we use VGG19 [77] as an example to illustrate the generalization of our optimizer on general DNNs. In these experiments, all the BN layers of the VGG19 model have been removed. It is well-known that the performance of DNNs may drop largely without BN. We compare SGD and SGD+WN with SGD-PH+WN. We tune both the learning rate and weight decay for these three methods and report their best results. The learning rate is tuned to be 0.01 for them all. The weight decays are 0.0001 for SGD and SGD+WN, and 0.001 for SGD-PH+WN. The other training strategies are the same as those introduced in Section 2.4.1. As a more intuitive comparison, we also report the results of the VGG19 model with BN layers in Table 2.4 using the same results reported in Section 2.4.1.

Table 2.4 gives the results of these five methods and Figure 2.5 shows the testing and training accuracy curves during training for the model VGG19 without BN layers. It is easy to see that the performance drop largely without BN, e.g., from 70.94% to 65.56% for SGDM. In this case, WN can slightly improve the performance of SGDM, while SGD-PH+WN can largely boost the performance over SGDM+WN by 2.37%. These results fully demonstrate the effectiveness of our proposed SGD-PH, that is, whether for a DNN with or without BN layers, SGD-PH can gain the final performance.

2.4.5 Results on Person ReID Benchmarks

In this section, we further apply SGD-PH to two person ReID benchmarks Market1501 [99] and DukeMTMC-ReID [65, 100]. The dataset Market1501 includes 1501 pedestrians and 32668 detected pedestrian rectangles, where 751 pedestrians with 12936 images of them are in training set and the rest 750 pedestrians with 19732 images are in testing sets, while DukeMTMC-ReID, a subset of DukeMTMC dataset for person ReID task, contains 702 persons with 16522 images for training, other 702

persons with 2228 query images and 17661 gallery images for testing. As well-known and commonly used benchmarks, they have been tackled in many existing work. Here, we adopt the method produced in [48] as our baseline [1].

Unlike image classification tasks that may benefit more from SGD than other optimizers, for person ReID tasks, the most popular optimizer is ADAM, which may be considered to have a more robust performance, and the default optimizer used in the baseline [48] is also ADAM. Tables 2.5 and 2.6 lists the results of Rank1 and mAP on the two datasets with ResNet18 and ResNet34 backbones. The experiments are done on eight GeForce RTX 2080Ti GPUs, repeated for 4 times and reported in the "mean ± std" format. Besides citing the performances of ADAM from [48], we tune the learning rate and weight decay of SGD by grid search with other settings keep the same as the baseline, while we keep the same hyperparameters as SGD for SGD-PH. Specifically, the learning rate and weight decay for SGD and SGD-PH on the two datasets are 0.03 and 0.003, respectively. For SGD-PH, the second order learning rate τ_{SO} takes 0.005 and 0.01 for Market1501 and DukeMTMC, respectively. The experiments on Market1501 is not inferior to SGDM (with slightly 0.1% better for most cases), while on DukeMTMC-ReID it gains 0.3%$\sim$ 0.4% for rank1 accuracy and 0.5%$\sim$ 0.6% for mAP compared with SGDM. Therefore, we illustrate the adaptability of our newly-proposed SGD-PH on person ReID tasks.

Table 2.5: Experiment results (%) on Market1501.

Network	ResNet18		ResNet34	
Indicator	Rank1	mAP	Rank1	mAP
ADAM	91.7	77.8	92.7	82.7
SGDM	92.5 ± .4	81.2 ± .3	**93.5 ± .3**	83.9 ± .1
SGD-PH	**92.6 ± .2**	**81.3 ± .1**	**93.5 ± .2**	**84.0 ± .4**

[1]The repository can be downloaded via: `https://github.com/michuanhaohao/reid-strong-baseline`.

Table 2.6: Experiment results (%) on DukeMTMC-ReID.

Network	ResNet18		ResNet34	
Indicator	Rank1	mAP	Rank1	mAP
ADAM	82.5	68.8	**86.4**	**73.6**
SGDM	$84.0 \pm .8$	$69.9 \pm .5$	$85.4 \pm .1$	$71.7 \pm .1$
SGD-PH	$\mathbf{84.4 \pm .4}$	$\mathbf{70.5 \pm .1}$	$85.7 \pm .9$	$72.2 \pm .8$

Table 2.7: Testing accuracies (%) of different LR and WD for ResNet18 on CIFAR100.

LR	0.01	0.01	0.01	0.02	0.005
WD	0.01	0.002	0.005	0.005	0.005
Accuracy	$77.19 \pm .45$	$77.33 \pm .14$	$\mathbf{77.96 \pm .30}$	$77.21 \pm .20$	$77.28 \pm .28$

2.4.6 Ablation Studies

In this section, we will report some ablation studies about SGD-PH to state its robustness and efficiency. We first tune some important hyperparameters, including the initial learning rate, weight decay, the second-order learning rate and the momentum of partial Hessian, then we test the performance of SGD-PH under different input batch size settings. Similarly, we repeat each experiment 4 times and report the results in the mean $\pm$ std format.

Learning Rate and Weight Decay: Generally, good settings for learning rate and weight decay can greatly benefit the final generalization results. In our experiments, we choose the learning rate (LR) and weight decay (WD) from the set $\{0.005, 0.01, 0.02\}$, while the learning rate of the second-order part τ_{SO} and the Hessian momentum α are set to be 0.001 and 0.9, respectively. Table 2.7 shows the results of five combinations of LR and WD. It can be found that the best result is setting LR to be 0.01 and WD to be 0.005, and we adopt this in the experiments of Section 2.4.1. Moreover, other settings also achieve acceptable results with fluctuations of no more than 0.77%, which illustrates that SGD-PH is robust to these

Table 2.8: Testing accuracies (%) of different second-order learning rate τ_{SO} for different DNNs on CIFAR100/10 datasets.

Dataset	CIFAR100			CIFAR10		
τ_{SO}	0.01	0.001	0.0001	0.01	0.001	0.0001
ResNet18	$77.33 \pm .24$	$77.96 \pm .30$	$\mathbf{78.22 \pm .20}$	$\mathbf{94.99 \pm .08}$	$94.98 \pm .07$	$94.86 \pm .06$
ResNet50	$\mathbf{79.61 \pm .20}$	$79.54 \pm .27$	$79.53 \pm .20$	$\mathbf{95.27 \pm .10}$	$95.17 \pm .10$	$95.07 \pm .12$
VGG11	$70.54 \pm .23$	$72.75 \pm .13$	$\mathbf{73.21 \pm .09}$	$92.14 \pm .13$	$\mathbf{92.64 \pm .24}$	$92.58 \pm .20$
VGG19	$72.78 \pm .17$	$\mathbf{73.87 \pm .28}$	$73.62 \pm .21$	$93.54 \pm .18$	$\mathbf{93.77 \pm .19}$	$93.55 \pm .20$

Table 2.9: Testing accuracies (%) of α for ResNet18 on CIFAR100.

α	0.8	0.9	0.99
Accuracy	$77.79 \pm .35$	$\mathbf{77.96 \pm .30}$	$77.81 \pm .32$

hyperparameters.

Second-order Learning Rate: In SGD-PH, an additional important hyperparameter is imported, i.e., the learning rate of the second-order part τ_{SO}. Here we tune τ_{SO} on CIFAR100/10 by traversing through the set $\{0.01, 0.001, 0.0001\}$, while the initial LR and WD are set to be 0.01 and 0.005, respectively. Table 2.8 gives the testing accuracies of different τ_{SO} on CIFAR100/10. We can see from the results that, on CIFAR100, a smaller τ_{SO} (i.e., 0.0001) may achieve better generalization performances in some shallow neural networks, e.g., VGG11 and ResNet18. However, for the deeper networks, a small τ_{SO} may no longer have such an advantage, on the contrary, a larger τ_{SO} like 0.01 may perform better. Meanwhile, on CIFAR10, the performances of SGD-PH with different τ_{SO} have no significant differences. Thus, the results in Table 2.8 shows the applicability of our proposed SGD-PH related to different second-order LR τ_{SO}. Furthermore, as a moderate choice, we take the hyperparameter $\tau_{SO} = 0.001$ in the above sections, and it works well through all of our experiments.

Table 2.10: Testing accuracies (%) of different batch size settings for ResNet18 on CIFAR100.

Batch Size	16	32	64	128	256
Accuracy	78.08 ± 0.08	$\mathbf{78.32 \pm .21}$	$78.14 \pm .35$	$77.96 \pm .30$	$77.56 \pm .16$

Hessian Momentum: Following the experience of tuning the momentum of SGDM, we list some testing accuracy results of SGD-PH with different Hessian momentum parameters $\{0.8, 0.9, 0.99\}$ for ResNet18 on CIFAR100 in Table 2.9. As shown in Table 2.9, $\alpha = 0.9$ attains the best results (which is the value we adopt in the previous sections), while other parameters also achieve acceptable results with a maximum fluctuation of 0.17%. Consequently, SGD-PH performs stably with these commonly chosen values of momentum.

Batch Size: The input batch size can also affect the performance of optimizers. Hence, we also pay attention to the impact of different batch size on the performance of SGD-PH. Here, we provide Table 2.10 about the testing accuracy results of SGD-PH for ResNet18 on CIFAR100, with the hyperparameters settled by lr$=0.01$, wd$=$ 0.005, $\tau_{SO} = 0.001$ and $\alpha = 0.9$. When the batch size increases from 16 to 256, there is a certain range of fluctuations in the testing accuracies, with the best result occurring at batch size 32. Totally, the trend of fluctuation about SGD-PH related to batch size is similar to other widely used optimizers, which ensures the adaptability and stability of SGD-PH for different settings in applications.

2.5 Conclusion

In this chapter, we propose SGD-PH, a compound optimizer that combines first-order optimizer SGDM with partially accurate Hessian information. The design of SGD-PH is based on the derivation of the Hessian matrices of the channel-wise 1D parameters, which are proved to be diagonal matrices and can be extracted precisely

through the Hessian-free method. Besides showing the effectiveness of SGD-PH on DNNs with the widely used normalization layers, we also give an example applying it to the reformulated convolutional layer directly, which illustrates that our optimizer can be applied to any DNNs (even if without normalization layers) with a satisfactory performance achieved. Sufficient ablation studies are accomplished to verify the robustness and adaptability of our proposed SGD-PH related to different hyperparameters. However, consistent with other second-order optimizers, our SGD-PH also needs more computational time and memory compared with first-order optimizers. This is still one of the key problems faced for designing second-order optimizers.

Chapter 3

NKFAC: A Fast and Stable KFAC Optimizer for Deep Neural Networks

In recent advances in second-order optimizers, computing the inverse of second-order statistics matrices has become critical. One such example is the Kronecker-factorized approximate curvature (KFAC) algorithm, where the inverse computation of the two second-order statistics to approximate the Fisher information matrix (FIM) is essential. However, the time-consuming nature of this inversion process often limits the extensive application of KFAC. What's more, improper choice of the inversion method or hyper-parameters can lead to instability and fail the entire optimization process. To address these issues, this chapter proposes the Newton-Kronecker factorized approximate curvature (NKFAC) algorithm, which incorporates Newton's iteration method for inverting second-order statistics. As the FIM between adjacent iterations changes little, Newton's iteration can be initialized by the inverse obtained from the previous step, producing accurate results within a few iterations thanks to its fast local convergence. This approach reduces computation time and inherits the property of second-order optimizers, enabling practical applications. The proposed algorithm is further enhanced with several useful implementations, resulting in state-of-the-art generalization performance without the need

for extensive parameter tuning. The efficacy of NKFAC is demonstrated through experiments on various computer vision tasks. The code is publicly available at `https://github.com/myingysun/NKFAC`.

3.1 Introduction

Benefited from the convenient back-propagation (BP) process, the first-order gradient-based algorithms [61, 36, 101, 47] have occupied the mainstream of deep neural networks (DNNs) optimization. Due to the appealing property of fast convergence, second-order optimizers have been continuously developed for training DNNs. At this stage, the research on second-order optimizers has achieved fruitful results by different approaches of design (e.g., [51, 88, 50, 26, 52, 80, 18]). They usually need fewer epochs than first-order optimizers to reach a specified loss or accuracy, confirming their broad application prospects in training DNNs.

Optimizing deep neural networks (DNNs) in a second-order manner usually requires addressing the challenging problem of approximating the matrix inverse, a task that is usually time-consuming and sometimes unstable. Two popular ways to utilize the second-order information are Hessian-based methods, which rely on the inverse of second-order Hessian matrix, and nature gradient based methods, which require the inversion of the Fisher information matrix (FIM). However, dealing with the inversion of these matrices can be complex and resource-intensive. Therefore, second-order optimizers usually encounter greater challenges than first-order methods in applications.

In recent decades, natural gradient (NG) methods have received considerable attention due to their "Hessian-free" computation, specifically, only first-order information (i.e., gradients) is required in their computation. From a design perspective, the NG descent algorithm is based on minimizing the KL-divergence of two distri-

butions, with the Fisher Information Matrix (FIM) serving as auxiliary information to aid in optimization. However, as a statistic that summarizes the information of stochastic parameters through observations, FIM is shown to be the expected value of the Hessian under certain regular assumptions [73]. As a result, NG methods are a specific kind of second-order optimizer, and they generally exhibit fast convergence.

The most popular way for applying the NG method is the Kronecker-factorized approximate curvature (KFAC) algorithm [52], which decouples the computation of FIM into a series of matrices of input and matrices of gradient under some assumptions. Thus, the computation cost of the inverse can be reduced from a greatly large FIM to a series of small matrices. Recent works [20, 21, 80, 5, 37, 22] have improved KFAC with better approximations. However, computing the inverse of the decoupled matrices of FIM remains an area that requires further study. A recent work [80] proposed SKFAC, which randomly selects (or averages) samples in spatial dimension for convolutional layers and adopts the Sherman–Morrison formula to reduce the dimension of inversion and shorten the computation time. However, besides the loss of information from random sampling and averaging, the moving average of input and gradient instead of second-order statistics in this method may also lead to a considerable loss of information. Consequently, although the inversion time can be significantly reduced, SKFAC's convergence speed and performance may drop more severely if the number of samples is reduced. Additionally, since the Sherman–Morrison formula cannot reduce the time of matrix inversion when two matrix dimensions are close, SKFAC cannot speed up the optimization process under such circumstances.

Besides reducing the computational time to narrow the gap between first-order and second-order algorithms, the stability of second-order methods is also an issue that requires attention. From one aspect, several commonly used inversion approaches such as the those relying on eigenvalue decomposition are not stable or

not well optimized in build-in functions, which may greatly prevent the widespread application of the corresponding optimizers. From another aspect, the instability of second-order optimizers can also arise from an improper dampening parameter that is added to the matrix, which requires careful parameter tuning.

In this chapter, based on the popular KFAC optimizer, we propose KFAC with Newton's iteration (in short, **NKFAC**), which is practical, stable and time-saving with state-of-the-art generalization results. By combining the property of slow change of FIM in training and the local fast convergence of Newton's iteration, NKFAC narrows the time gap between first-order and second-order methods while inherits the convergence property of second-order optimizers, and is not inferior to first-order methods in generalization performance. We verify the effectiveness and efficiency of NKFAC by extensive experiments on different tasks in computer vision.

3.2 Background and Preliminaries

3.2.1 Matrix Inverse

Efficient computation of matrix inverses is a longstanding and critical research topic due to the universality and time-consuming nature of inversion. In DNNs optimization, there are two main techniques used to reduce the computational cost of matrix inversion. One is the equivalent matrix transformation [80, 2, 37], which reduces the inversion to a small dimension matrix by rigorous mathematics equations like the Sherman–Morrison formula. The other one is the application of iteration methods (e.g., [1, 69, 7, 23]), which combines DNN training with some common iteration methods (e.g., (quasi-)Newton's iteration, conjugate gradient (CG) [30] and GMRES [58]).

Proper inversion methods are critical to the success of training DNNs with second-order optimizers. Using an improper method can lead to the failure of the entire

training process. For example, when the input matrix has repeated eigenvalues, eigenvalue decomposition-based inversion often fails and disrupts the optimization process. Another concern in the stability of inversion arises from the poor condition number of the matrix. Some matrices encountered in DNNs training may have poor condition numbers that cause difficulties with inversion, regardless of the method used. Numerical operations can mitigate this problem to some extent, for example, minimizing division in the inversion process to reduce the risk of numerical overflow. Meanwhile, a proper dampening parameter is usually necessary to improve the condition number. The classic KFAC [52] also stresses the importance of the dampening parameter for second-order optimizers in detail. In this chapter, we are motivated to design a stable algorithm with a suitable inversion method and adaptive dampening parameter.

3.2.2 KFAC Algorithm

Here we introduce the formulation of KFAC [52, 24] from the definition of Fisher information matrix (FIM). Denote $\theta = (\mathrm{vec}(\mathbf{W}_1), \cdots, \mathrm{vec}(\mathbf{W}_l))$ the collective tensor of all the weights $\mathbf{W}_i$ for $i = 1, \ldots, l$ in a fully-connected network. For a negative log probability loss $L(\mathbf{Y}|f(\mathbf{X}, \boldsymbol{\theta})) = -\log r(\mathbf{Y}|f(\mathbf{X}, \boldsymbol{\theta}))$, where r is the probability density function (p.d.f.) of the predictive distribution of $\mathbf{Y}$ parameterized by $f(\mathbf{X}, \boldsymbol{\theta})$, by denoting $p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta}) := r(\mathbf{Y}|f(\mathbf{X}, \boldsymbol{\theta}))$ and $\mathcal{D}\boldsymbol{\theta}$ the gradient of $\log p$ with respect to $\boldsymbol{\theta}$, the FIM is defined by

$$\mathbf{F} = \mathbb{E}\left[\frac{d\log p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta})}{d\boldsymbol{\theta}} \frac{d\log p(\mathbf{Y}|\mathbf{X}, \boldsymbol{\theta})}{d\boldsymbol{\theta}}^{\top}\right] = \mathbb{E}\left[\mathcal{D}\boldsymbol{\theta}\mathcal{D}\boldsymbol{\theta}^{\top}\right]. \tag{3.1}$$

Therefore, the parameter update equation of NG method is defined by

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta_t \mathbf{F}_t^{-1}\mathcal{D}\boldsymbol{\theta}. \tag{3.2}$$

Here, the matrix $\mathbf{F}$ is computed only by the first-order derivatives $\mathcal{D}\boldsymbol{\theta}$, thus no additional back-propagation is needed, which saves plenty of time as a second-order

optimizer.

With the advantage of being "Hessian-free" to save computational cost, NG methods only need to solve the storage problem cost by the highly large dimension of $\mathbf{F}$. Each block of $\mathbf{F}$, denote by $\mathbf{F}_{i,j}$ (defined in Eq. (3.3)), can be approximated with the assumption of the statistic independence between inputs $\mathbf{X}$ and their gradients $\mathbf{\Delta}$, i.e.,

$$\mathbf{F}_{i,j} := \mathbb{E}[\mathrm{vec}(\mathcal{D}\mathbf{W}_i)\mathrm{vec}(\mathcal{D}\mathbf{W}_j)] = \mathbb{E}[\mathbf{X}_i\mathbf{X}_j^\top \otimes \mathbf{\Delta}_i\mathbf{\Delta}_j^\top] \approx \mathbb{E}[\mathbf{X}_i\mathbf{X}_j^\top] \otimes \mathbb{E}[\mathbf{\Delta}_i\mathbf{\Delta}_j^\top]. \quad (3.3)$$

For convenient, we use the notation $\mathbf{L}_{i,j} := \mathbb{E}[\mathbf{\Delta}_i\mathbf{\Delta}_j^\top]$ and $\mathbf{R}_{i,j} := \mathbb{E}[\mathbf{X}_i\mathbf{X}_j^\top]$. Therefore, by the block diagonal approximation of the FIM $\mathbf{F} \approx \mathrm{Diag}(\mathbf{F}_{11}, \cdots, \mathbf{F}_{ll})$, together with Eq. (3.2) and the computation rules of Kronecker product, we get that for the t-th iteration, the update formula of KFAC, converted to the tensor operation related to the weight $\mathbf{W}$, takes the form

$$\mathbf{W}_{t+1} = \mathbf{W}_t - \tau_t \mathbf{L}_{t+1}^{-1} \mathbf{G}_t \mathbf{R}_{t+1}^{-1}, \quad (3.4)$$

where $\mathbf{L} = \mathrm{Diag}(\mathbf{L}_{11}, \cdots, \mathbf{L}_{ll})$, $\mathbf{R} = \mathrm{Diag}(\mathbf{R}_{11}, \cdots, \mathbf{R}_{ll})$, and $\mathbf{G}$ is the gradient of the weight $\mathbf{W}$ in tensor form.

The update equation (3.4) is derived only for fully-connected layers [52]. After this, with the help of *im2col* function, a similar equation is derived for convolutional layers in [24] under some other assumptions, in which two matrix inverses related to the gradients ($\mathbf{L}^{-1}$) and the inputs ($\mathbf{R}^{-1}$) are also required. Although the periodic strategy can be adapted to update the inverse [57], the computation time of inverse is still an important question that deserves consideration to maximize the performance of the optimizer.

3.2.3 About Other Second-order Optimizers

There are many other recently proposed second-order optimizers, e.g., GGT[2], Shampoo[26], AdaHessian[88] and M-FAC[18]. Apart from AdaHessian, the other

three optimizers all require matrix inversion, which suffer from the instability and time-consuming problems that will be specified in Section 3.3.1 without exception. Unfortunately, AdaHessian requires the second-time back-propagation, which is the most time-consuming (over 4 times compared with SGD). Besides, GGT and M-FAC both import a significant hyper-parameter related to matrix dimension, i.e. the window size r. This parameter r has a great influence on the computational cost and storage, and when using the default number $r = 512$, M-FAC even cannot train ResNet50 on CIFAR100 within a 48G storage GPU. Therefore, the design of the second-order optimizer is still a research hot spot.

3.3 Methodology

3.3.1 Motivation

Apart from reducing the time cost of matrix inversion, the toolbox and built-in functions of different machine learning frameworks have also been updated to meet the need for stability as well as time. However, when we tried to reproduce some second-order optimizers, we found that there were still some unstable built-in functions related to the inversion that may kill the optimization process. For example, to deal with matrix inversion, many implementations of second-order optimizers first apply eigenvalue decomposition to the matrix, and then use matrix multiplication to obtain the corresponding inverse. By this implementation, the maximum and minimum eigenvalues can also be obtained for some parameter tuning use. When using a previous version of PyTorch, eigenvalue decomposition sometimes failed, halting the optimization process. Although using the *torch.linalg.inv*() function in PyTorch 1.9 alleviated this issue, some complex matrix inversion algorithms still suffer from numerical overflow problems, particularly when the input matrix is ill-conditioned. Furthermore, parallel processing on GPUs for matrix inversion is not as feasible as

for other optimized operations (e.g., matrix multiplication).

Moreover, the time cost of computing inverse within a single optimization step for KFAC is very large. For example, when training ResNet50 on CIFAR100, computing the inverse takes up over 90% of the time cost of an optimization step. However, previous research [57] has shown that the Fisher Information Matrix (FIM) changes slowly. Our experiments have confirmed this property, especially once the training process stabilizes. Moreover, taking the momentum of second-order statistics into consideration, the inverse of FIM approximated by these statistics also change little between iterations. Therefore, if we adopt an iterative approach to solve the inverses each time, we can use the inverses computed in the previous iteration as good initialization points for the current computation. This property, to our best knowledge, has not been well-utilized in previous work.

To have a more intuitive explanation, we draw the changing curves of these two statistics during the training process for ResNet18 on CIFAR100. Here in the Fig. 3.1, the plot value represents the norm of the difference of the statistics between an interval of T_{stat} (takes 20 here) and is divided by the current statistics norm. We take the average of the results of four experiments to plot the figure. We can see from Fig. 3.1 that, except for the very beginning stage, all the values are small, which verifies the slow change of our Fisher information statistics.

Considering the above facts, we are motivated to design an iterative method that can quickly converge locally to maximize the utilization of the previous inverse, and keep only well-optimized matrix multiplications in our method to maximize stability. Here, we consider Newton's iteration, which is detailed described in Section 3.3.2.

3.3.2 Newton's Iteration of Matrix Inverse

For a given real matrix $\mathbf{A}$, an iterative method [71, 6] of computing matrix inverse $\mathbf{A}^{-1}$, which is a generalization of Newton's iterative method and also known as

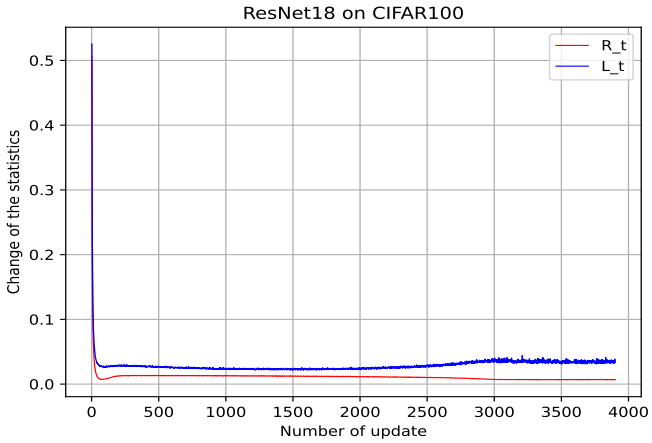


Figure 3.1: Change of $\mathbf{L}_t$ and $\mathbf{R}_t$ in training ResNet18 on CIFAR100.

Schultz iteration, obtains a sequence $\{\mathbf{B}_k\}$ that performed simply by

$$\mathbf{B}_{k+1} = 2\mathbf{B}_k - \mathbf{B}_k\mathbf{A}\mathbf{B}_k. \tag{3.5}$$

Assume $\mathbf{A}$ is an invertible matrix, and $\mathbf{B}_0$ is an initialization closed to $\mathbf{A}^{-1}$. Now we state the following **Proposition 1**.

Proposition 3.1. *[55, Section 2.3] If* $\|\mathbf{B}_0\| \neq 0$ *and* $\|\mathbf{I}-\mathbf{A}\mathbf{B}_0\| < 1$*, then the sequence* $\{\mathbf{B}_k\}$ *generated by* (3.5) *converges to* $\mathbf{A}^{-1}$ *with an order at least* 2*.*

For the case that $\|\mathbf{I}-\mathbf{A}\mathbf{B}_0\| \geqslant 1$, we add a stepsize α_k in the k-th iteration. Thus, the Newton's iteration becomes

$$\mathbf{B}_{k+1} = (1+\alpha_{k+1})\mathbf{B}_k - \alpha_{k+1}\mathbf{B}_k\mathbf{A}\mathbf{B}_k. \tag{3.6}$$

Note that if $\alpha_{k+1} = 1$, the update iteration is exactly the classic equation (3.5). In our experiments, we empirically set an adapting stepsize $\alpha_{k+1} = \frac{1}{\|\mathbf{A}\mathbf{B}_k\|}$. Since we have computed $\mathbf{A}\mathbf{B}_k$ when determining the norm, no additional multiplication is needed. Here we give some simple examples to test the performance of the above Newton's iteration using random generated 1024×1024 matrices. We normalize the matrices to have norm 1 and add perturbations with different norm $0.1, 0.05, 0.01$ to

Algorithm 2 Newton's Iteration for solving matrix inverse

Inputs: $(\mathbf{A}, \mathbf{B}_0, K)$;
Output: $\mathbf{B}_K$.

for $k = 1, \ldots, K$ **do**
  if $\|\mathbf{I} - \mathbf{A}\mathbf{B}_0\| < 1$ **then**
    $\mathbf{B}_k = 2\mathbf{B}_{k-1} - \mathbf{B}_{k-1}\mathbf{A}\mathbf{B}_{k-1}$;
  else
    $\alpha_k = \frac{1}{\|\mathbf{A}\mathbf{B}_{k-1}\|}$, $\mathbf{B}_k = (1 + \alpha_k)\mathbf{B}_{k-1} - \alpha_k\mathbf{B}_{k-1}\mathbf{A}\mathbf{B}_{k-1}$.
  end if
end for

them. Figure 3.2 shows the loss curve of different perturbation cases with respect to iteration. Ablation study about the stepsize α_{k+1} under the case $\|\mathbf{I} - \mathbf{A}\mathbf{B}_0\| \geqslant 1$ can be found in Section 3.4.

Figure 3.2 also shows that the fast convergence of Newton's method relies heavily on the request of a good initialized point. Thus, when applying Newton's methods to solve traditional optimization problems, first-order methods are usually adopted for some iterations as a warm-up to acquire a good initialized point for Newton's method, and in principle, a series $\{\mathbf{B}_k\}$ should be generated to reach a good approximation of $\mathbf{A}^{-1}$. However, as we explained in Section 3.3.1, the change of the second-order statistic between iterations is quite small, which states that the inversion result of the last iteration is exactly a satisfactory initialized point of the current iteration. This ensures the efficiency of Newton's method with few iterations.

Although in the recent research, a series of improved iterative methods with the order of convergence rate from cubic to seventh has been developed (e.g., [78, 41]), we still adopt the above Newton's iteration with a substituted stepsize, since it has just several simple matrix multiplications and is easy to achieve a balance between computational cost and performance.

The detailed steps of the Newton's iteration for solving matrix inverse is stated in **Algorithm 2**.

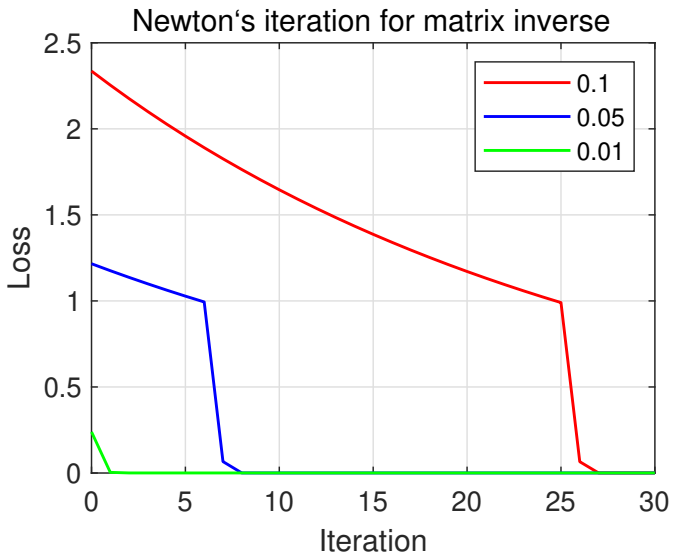


Figure 3.2: Loss curves of Algorithm 1.

3.3.3 NKFAC

As mentioned in Section 3.2.2, KFAC can be applied on both linear and convolutional layers with updating formula Eq.(3.4). Thus, with Newton's iteration to update the matrix inverse $\mathbf{L}^{-1}$ and $\mathbf{R}^{-1}$, we combine **Algorithm 2** with classic KFAC, and propose KFAC with Newton iteration (**NKFAC**). Again, since the FIM changes slow, we apply periodical strategy to update the second order statistics $\mathbf{L}, \mathbf{R}$ and their inverses, i.e., T_{stat} and T_{inv} steps, respectively. The periodical strategy is also adopted in some existing works, e.g., [93, 80]. Moreover, at certain larger intervals CT_{inv}, we recalculate the exact inverse to prevent the approximation from getting too far from the exact inverse.

Different from the existing works, we do not reduce the inversion dimension by random sampling or averaging, so the comprehensiveness of the input information was preserved to the greatest extent. This also allows us to maximally inherit the fast convergence properties of the second-order optimizer while accelerating the inversion process. Meanwhile, **Algorithm 2** has only matrix product operations, which, from

Algorithm 3 Periodical inversion updating strategy at the t-th iteration

Inputs: Tensors $\mathbf{L}_t, \mathbf{R}_t, \hat{\mathbf{L}}_t, \hat{\mathbf{R}}_t, \mathbf{X}_t, \mathbf{\Delta}_t$; floats η, α; integers T_{stat}, T_{inv}, C, K.
Outputs: $(\hat{\mathbf{L}}_{t+1}, \hat{\mathbf{R}}_{t+1})$.

if $t\%T_{stat} = 0$ **then**
  $\mathbf{L}_{t+1} = \alpha\mathbf{L}_t + (1-\alpha)\mathbf{\Delta}_t\mathbf{\Delta}_t^\top$, $\mathbf{R}_{t+1} = \alpha\mathbf{R}_t + (1-\alpha)\mathbf{X}_t\mathbf{X}_t^\top$;
else
  $\mathbf{L}_{t+1} = \mathbf{L}_t$, $\mathbf{R}_{t+1} = \mathbf{R}_t$;
end if *%periodical update the statistics*
if $t\%(CT_{inv}) = 0$ or $t < 2T_{inv}$ **then**
  $\hat{\mathbf{L}}_{t+1} := (\mathbf{L}_{t+1} + \eta\lambda_{\text{Lmax}}\mathbf{I})^{-1}$, $\hat{\mathbf{R}}_{t+1} := (\mathbf{R}_{t+1} + \eta\lambda_{\text{Rmax}}\mathbf{I})^{-1}$;
else if $t\%T_{inv} = 0$ **then**
  $\hat{\mathbf{L}}_{t+1} := \textbf{Alg.1}(\mathbf{L}_{t+1} + \eta\lambda_{\text{Lmax}}\mathbf{I}, \hat{\mathbf{L}}_t, K)$, $\hat{\mathbf{R}}_{t+1} := \textbf{Alg.1}(\mathbf{R}_{t+1} + \eta\lambda_{\text{Rmax}}\mathbf{I}, \hat{\mathbf{R}}_t, K)$;
else
  $\hat{\mathbf{L}}_{t+1} = \hat{\mathbf{L}}_t$, $\hat{\mathbf{R}}_{t+1} = \hat{\mathbf{R}}_t$;
end if *%periodical update the inverse*

a numerical perspective, is stable in computation compared with other inversion methods that may need division. The possibility of numerical overflow is reduced, which also makes the entire training process more stable.

Before stating the whole structure of NKFAC, from a simple and practical point of view, we will introduce some useful implementations in Section 3.3.4.

3.3.4 Implementations and AdaNKFAC

Before introducing the implementations of our NKFAC, it is worth mentioning that these techniques can be implemented in other second-order optimizers to avoid the tedious tuning process and improve the stability of the algorithms. We also report some ablation studies of them on classic KFAC and the generalized SKFAC in Section 3.4.

Gradient Norm Recovery: Parameter tuning is a very important but tedious part of training DNNs. At this stage, first-order optimizers have been widely used in DNNs training. In the deep learning tasks, most of the commonly used parameters and tuning schedules, including learning rate and weight decay, have been

finely tuned by many existing works, so their performance are generally satisfactory. However, for the second-order optimizers, the best parameters and tuning schedules may vary greatly from the first-order optimizers. Carefully tuning parameters will cost a lot of time and resources. However, without tuning parameters, second-order optimizers are easy to fail since they are usually more sensitive to changes in parameters. To solve the parameter tuning question of second-order algorithms, we adopt the gradient norm recovery technique, which can also be found in [93]. Specifically, the descent direction $\hat{\mathbf{G}} = \mathbf{L}^{-1}\mathbf{G}\mathbf{R}^{-1}$ is recovered by

$$\tilde{\mathbf{G}} = \hat{\mathbf{G}}\frac{\|\mathbf{G}\|}{\|\hat{\mathbf{G}}\|}. \tag{3.7}$$

After implementing this technique, the finely tuned parameters and tuning schedules can be adopted with just small changes, even be applied directly. This allows us to maximize the use of the existing parameter tuning results without much tedious and repetitive works.

Adaptive Dampening: In computing matrix inverse, the property of the matrix itself has a great influence on the difficulty of solving the inverse and the stability of the inversion algorithm. For example, if the matrix is ill-conditioned, most algorithms may easy to fail numerically, and if the minimum eigenvalue of the matrix is close to zero, the magnitude of the inverse may become quite large, which may also fail the whole optimization process. For second-order optimizers, dampening parameters can greatly influence the training performance. Here, for each matrix $\mathbf{L}$ and $\mathbf{R}$, we adopt the adaptive dampening parameters related to the maximum eigenvalue λ_{Lmax} and λ_{Rmax} respectively, i.e.,

$$\mathbf{L} := \mathbf{L} + \eta\lambda_{\text{Lmax}}\mathbf{I}, \quad \mathbf{R} := \mathbf{R} + \eta\lambda_{\text{Rmax}}\mathbf{I}, \tag{3.8}$$

where $\eta \in (0, 1)$ is a given hyper-parameter. This technique can also be found in [93] to improve the property of matrices with different magnitudes. With this technique,

the matrix to be inverted becomes more stable.

Statistics Momentum: The second-order statistics used to help optimization are stored in two matrices $\mathbf{L}$ and $\mathbf{R}$, which are exactly the approximations of FIM and also the matrices that need inversion. To sum up all the information of different batches of input, we add momentum to the second-order statistics, i.e., for each iteration $t+1$, we compute and store the matrix $\mathbf{L}_{t+1}$ and $\mathbf{R}_{t+1}$ by

$$\mathbf{L}_{t+1} = \alpha \mathbf{L}_t + (1-\alpha)\boldsymbol{\Delta}_t \boldsymbol{\Delta}_t^\top, \quad \mathbf{R}_{t+1} = \alpha \mathbf{R}_t + (1-\alpha)\mathbf{X}_t \mathbf{X}_t^\top. \tag{3.9}$$

In application, the hyper-parameter α is usually chosen closed to 1, which leads to the slow change of FIM. Therefore, we design **Algorithm 3**, which shows our periodical inversion updating strategy.

Adaptive Stepsize (AdaNKFAC): Inspired by the success of adaptive stepsize methods especially in training transformers, we also add the adaptive stepsize and weight decouple technique into NKFAC and propose AdaNKFAC (stated in **Algorithm 5**), which can also be regarded as a combination of NKFAC and AdamW[47]. Thus, without tedious parameter-tuning, we easily achieve higher performance compared with classical adaptive stepsize method(e.g., AdamW) as we reported in Section 5.4.

Overall, the detailed NKFAC and AdaNKFAC algorithm are stated in **Algorithm 4** and **Algorithm 5**, respectively. For more ablation study experiments about these implementations, please refer to Section 5.4.

3.4 Experiments

3.4.1 Experimental Setup

Hyper-parameters settings: Due to the fast local convergence of Newton's method, together with the numerical performance, we take $K=1$ in **Algorithm 2**. Although the step seems few, it works well through all our experiments. In classification tasks

Algorithm 4 NKFAC

Inputs: Initialization $\mathbf{W}_0, \mathbf{L}_0{=}\epsilon\mathbf{I}_{C_{out}}, \mathbf{R}_0{=}\epsilon\mathbf{I}_{C_{in}}$, $\hat{\mathbf{L}}_0{=}\mathbf{0}_{C_{out}}, \hat{\mathbf{R}}_0{=}\mathbf{0}_{C_{in}}$; float numbers τ, η; integer constants T_{stat}, T_{inv}, C, K;
Outputs: $\mathbf{W}_T$.

for $t = 0, 1, \ldots, T-1$ **do**
  compute the gradient $\mathbf{G}_t$;
  save $\mathbf{X}_t{=}[x_{ti}]_{i=1}^n$ and $\mathbf{\Delta}_t{=}[\delta_{ti}]_{i=1}^n$; *%by forward and backward propagation, respectively*
  $(\hat{\mathbf{L}}_{t+1}, \hat{\mathbf{R}}_{t+1}) =$ **Alg.2**$(\mathbf{L}_t, \mathbf{R}_t, \hat{\mathbf{L}}_t, \hat{\mathbf{R}}_t, \mathbf{X}_t, \mathbf{\Delta}_t, \eta, \alpha, T_{stat}, T_{inv}, C, K)$;
  $\hat{\mathbf{G}}_{t+1} = \hat{\mathbf{L}}_{t+1}\mathbf{G}_t\hat{\mathbf{R}}_{t+1}$;
  $\tilde{\mathbf{G}}_{t+1} = \hat{\mathbf{G}}_{t+1}\frac{\|\mathbf{G}_t\|}{\|\hat{\mathbf{G}}_{t+1}\|}$;
  $\mathbf{W}_{t+1} = \mathbf{W}_t - \tau\tilde{\mathbf{G}}_{t+1}$.
end for

Algorithm 5 AdaNKFAC

Inputs: $\mathbf{W}_0, \mathbf{L}_0{=}\epsilon\mathbf{I}_{C_{out}}, \mathbf{R}_0{=}\epsilon\mathbf{I}_{C_{in}}$, $\hat{\mathbf{L}}_0{=}\mathbf{0}_{C_{out}}, \hat{\mathbf{R}}_0{=}\mathbf{0}_{C_{in}}$, $\mathbf{M}_0{=}\mathbf{0}_{C_{out}\times C_{in}}, \mathbf{V}_0{=}\mathbf{0}_{C_{out}\times C_{in}}$; float τ, η, $\beta_1, \beta_2, \varepsilon$; integers T_{stat}, T_{inv}, C, K;
Outputs: $\mathbf{W}_T$.

for $t = 0, 1, \ldots, T-1$ **do**
  compute the gradient $\mathbf{G}_t$;
  save $\mathbf{X}_t{=}[x_{ti}]_{i=1}^n$ and $\mathbf{\Delta}_t{=}[\delta_{ti}]_{i=1}^n$;
  $(\hat{\mathbf{L}}_{t+1}, \hat{\mathbf{R}}_{t+1}) =$ **Alg.2**$(\mathbf{L}_t, \mathbf{R}_t, \hat{\mathbf{L}}_t, \hat{\mathbf{R}}_t, \mathbf{X}_t, \mathbf{\Delta}_t, \eta, \alpha, T_{stat}, T_{inv}, C, K)$;
  $\hat{\mathbf{G}}_{t+1} = \hat{\mathbf{L}}_{t+1}\mathbf{G}_t\hat{\mathbf{R}}_{t+1}$;
  $\tilde{\mathbf{G}}_{t+1} = \hat{\mathbf{G}}_{t+1}\frac{\|\mathbf{G}_t\|}{\|\hat{\mathbf{G}}_{t+1}\|}$;
  $\boldsymbol{M}_{t+1} = \beta_1\boldsymbol{M}_t + (1-\beta_1)\tilde{\mathbf{G}}_{t+1}$;
  $\boldsymbol{V}_{t+1} = \beta_2\boldsymbol{V}_t + (1-\beta_2)\tilde{\mathbf{G}}_{t+1}\odot\tilde{\mathbf{G}}_{t+1}$;
  $\tilde{\boldsymbol{M}}_{t+1} = \frac{\boldsymbol{M}_{t+1}}{1-\beta_1^t}, \tilde{\boldsymbol{V}}_{t+1} = \frac{\boldsymbol{V}_{t+1}}{1-\beta_2^t}$;
  $\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \tau\frac{\tilde{\boldsymbol{M}}_t}{\sqrt{\tilde{\boldsymbol{V}}_t}+\varepsilon}$.
end for

that compared with NKFAC and KFAC, we take $T_{stat} = 20$ and $T_{inv} = 200$, which keeps the same interval parameters as in [80] for a fair comparison. In other tasks, we adopt $T_{stat} = 100$ and $T_{inv} = 1000$, which not only shows the wide applicability and satisfactory performance but also states the robustness of our proposed NKFAC. The integer interval C takes 500, the gradient momentum takes 0.9, the second-order statistics momentum is set to 0.95, and the dampening parameter $\eta = 0.01$.

Choice of NKFAC and AdaNKFAC: For experiments on transformers, the adap-

tive momentum methods often obtain more satisfactory results than SGDM, and the default optimizer of training transformers are often AdamW. To keep satisfactory results without tedious parameter tuning, we choose AdaNKFAC for these experiments, while for traditional CNNs with default optimizer SGD, we choose NKFAC as a comparison. Thus, we can gain the performance with the same or just slightly different hyper-parameters.

Baseline Optimizers: The baseline optimizers occur in this section are KFAC, SKFAC, and the first-order optimizers SGDM [61], AdamW [47], RAdam [45] and Adabelief [101]. For SKFAC, a too-small number of random sampling will affect the performance, so we take the random sampling number to be 8, which is still within the suggested range in [80]. Meanwhile, since our experiments are accomplished on Pytorch 1.9 framework, by noticing that the inverse function runs much faster than computing the inverse with the help of the eigenvalue decomposition function, we implement the computation of maximum eigenvalue by Newton-Schulz iterations to save running time in each step. It is worth mentioning that we have tested the second-order optimizers mentioned in Section 3.2.3, i.e., GGT[2], Shampoo[26], AdaHessian[88] and M-FAC[18]. However, as we mentioned that second-order optimizers usually need parameter-tuning from scratch, in our experiments, even after the tedious parameter-tuning, their performance are still not satisfactory and sometimes fail to convergence. Therefore, we do not contain these optimizers in our comparison in this section.

3.4.2 Results on CIFAR100/10

In this section, our experiments are conducted on CIFAR100/10 [38] datasets with 4 Geforce RTX 2080Ti GPUs. In our experiments, we train DenseNet121 [32], ResNet18, ResNet50 [29], and GoogLeNet [79] for total 200 epochs with batch size 128 on one single GPU. We use cosine learning rate schedule, the detailed settings of

which are reported in Table 3.2. Each experiment is repeated four times to eliminate randomness. The generalization performance is reported in the "mean $\pm$ std" format, and the time is reported by taking the average.

From the aspect of generalization performance, we see from Table 3.1 that NKFAC performs the best on CIFAR100 compared with all the other optimizers. After careful hyper-parameters tuning, KFAC performs better than the default first-order SGDM optimizer under most circumstances. With the proposed useful implementations, our NKFAC inherits this property with even better accuracy and faster convergence speed than KFAC as shown in Fig. 3.3. SKFAC, however, may suffers from information lose and does not perform as well as NKFAC and KFAC. For experiments on CIFAR10, we notice from the training process that after 200 epochs, all the optimizers reach nearly 100% training accuracy with the training loss very close to zero, which means that all the networks are trained well under these circumstances. Therefore, there may not big difference about the generalization performance of different optimizers on CIFAR10 for the DNNs we tested, as we show in Table 3.1.

Table 3.3 demonstrates our superior ability to approximate the inverse from a time perspective. In most of the DNNs we tested, NKFAC outperforms other optimizers. Particularly, when the Sherman-Morrison formula cannot significantly reduce the dimension of inversion, SKFAC may not be powerful enough, whereas NKFAC consistently demonstrates its effectiveness. Across all experiments, NKFAC reduces inversion time from 31% to 86% compared to KFAC, while SKFAC achieves a reduction of 3% to 61%. These results demonstrate the time-saving benefit of using Newton's iteration in NKFAC.

Table 3.1: Testing accuracies (%) of different optimizers on CIFAR100/10. * means at least one of the four repeated experiments fails to converge.

CIFAR100							
Optimizer	SGDM	AdamW	RAdam	Adabelief	KFAC	SKFAC	NKFAC
DenseNet121	80.26 ± .32	78.92 ± .17	79.29 ± .19	80.21 ± .24	79.32 ± .05	80.44 ± .35	**81.13** ± .16
ResNet18	78.41 ± .25	77.63 ± .18	77.54 ± .11	78.41 ± .29	79.96 ± .21	78.91 ± .14	**80.23** ± .24
ResNet50	78.87 ± .35	78.85 ± .50	79.15 ± .24	80.61 ± .44	80.77 ± .17	81.16 ± .16	**81.78** ± .06
GoogLeNet	80.26 ± .15	79.83 ± .22	79.90 ± .07	80.98 ± .15	81.47 ± .24	79.68 ± .19	**81.65** ± .10
CIFAR10							
DenseNet121	95.73 ± .12	95.11 ± .14	95.17 ± .24	95.68 ± .14	95.17 ± .11	**95.77** ± .20	95.72 ± .07
ResNet18	95.50 ± .07	95.05 ± .14	95.04 ± .10	95.34 ± .10	**95.84** ± .12	95.82 ± .05	95.80 ± .10
ResNet50	95.43 ± .21	95.20 ± .07	95.19 ± .15	95.77 ± .11	95.93 ± .09	**96.11** ± .16	96.07 ± .10
GoogLeNet	95.56 ± .09	95.05 ± .14	95.17 ± .10	95.67 ± .11	95.52 ± .00*	95.57 ± .16	**96.23** ± .06

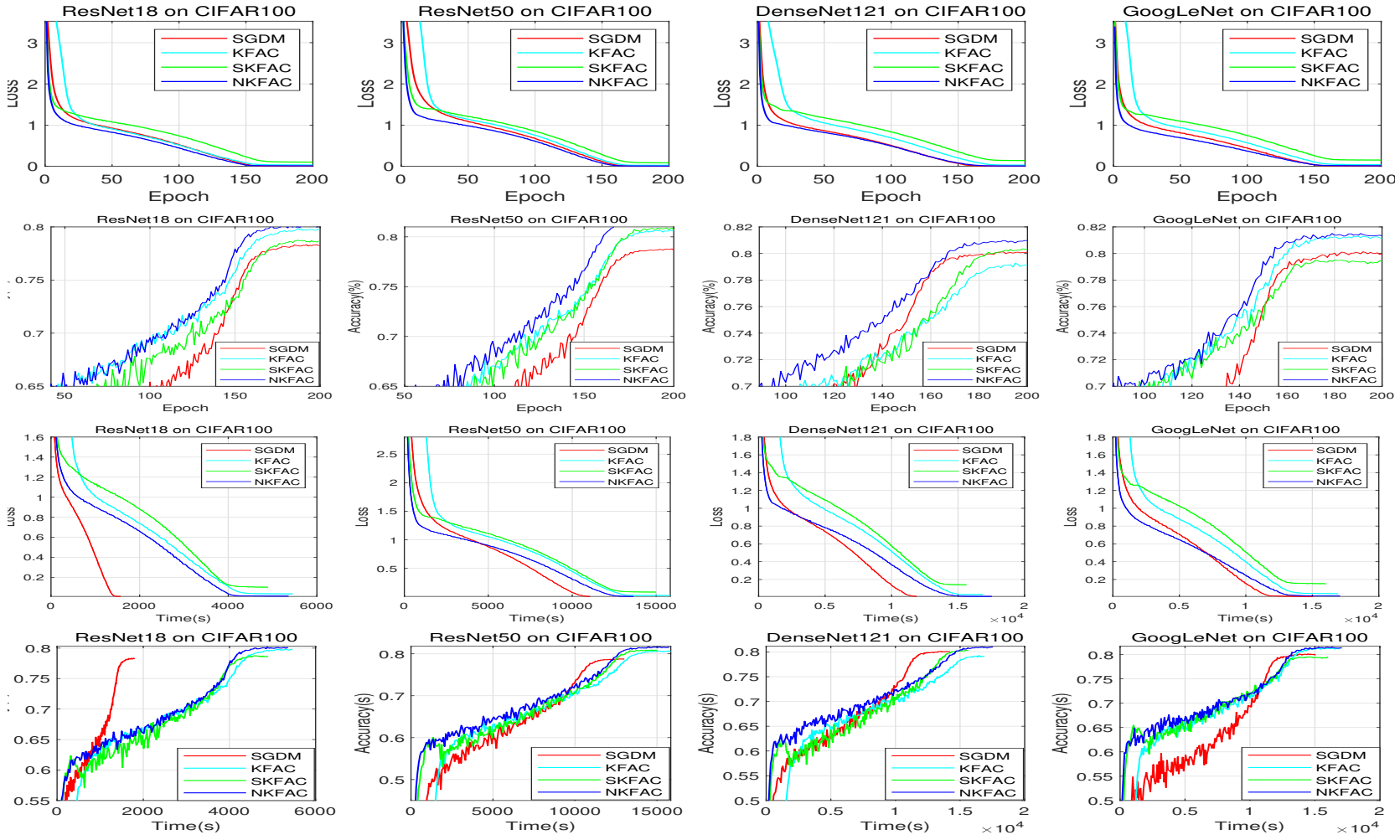


Figure 3.3: Training loss and testing accuracy curves with respect to epoch and time on CIFAR100 for different optimizers, respectively.

Table 3.2: Settings of learning rate (LR) and weight decay (WD) for different optimizers on CIFAR100/10.

Optimizer	SGDM	AdamW	RAdam	Adabelief	KFAC	SKFAC	NKFAC
LR	0.1	0.001	0.001	0.001	0.0005	0.0005	0.05
WD	0.0005	0.5	0.5	0.5	0.1	0.1	0.001

Table 3.3: Time cost by inversion (s) of optimizers on CIFAR100.

Optimizer	KFAC	SKFAC	NKFAC
DenseNet121	264.68	244.16	**35.56**
ResNet18	136.41	**53.32**	54.24
ResNet50	248.84	155.23	**74.13**
GoogLeNet	102.83	99.32	**70.69**

From the aspect of convergence, we plot Fig. 3.3 about the training loss and testing accuracy curves with respect to epoch and total training time for different optimizer, respectively. When train the same number of epochs, NKFAC always gain the performance among all the optimizers compared whether according to training loss or testing accuracy. Meanwhile, considering total training time, NKFAC reduces the loss quicker and gains the generalization performance at most of the time compared with KFAC, NKFAC under most cases, while compared with the first-order optimizer SGDM, it is usually better than SGDM in the early stage but surpassed by SGDM in the later stage. Here, as shown in Fig. 3.3, first-order optimizer still shows it advantage in less training time.

3.4.3 Results on ImageNet

ImageNet [68] is an image classification dataset that contains 1k categories with 1.28 million images for training and 50k images for validation. Our experiments on ImageNet are accopmlished on GeForce RTX 2080Ti GPUs. The initial learning rate and weight decay are reported in Table 3.4.

Results on CNNs: In our testing, we train ResNet18 and ResNet50 networks [29]

Table 3.4: Settings of learning rate (LR) and weight decay (WD) for different optimizers on ImageNet.

Optimizer	SGDM	AdamW	RAdam	Adabelief	KFAC	SKFAC	NKFAC
LR	0.1	0.001	0.001	0.001	0.0005	0.0005	0.05
WD	0.0001	0.1	0.1	0.5	0.02	0.02	0.0002

Table 3.5: Top 1 accuracy (%) for different optimizers on ImageNet with ResNet18/50.

Optimizer	SGDM	AdamW	RAdam	Adabelief	KFAC	SKFAC	NKFAC
ResNet18	70.49	70.01	69.92	70.08	70.51	10.28	**71.12**
ResNet50	76.31	76.02	76.12	76.22	76.66	29.09	**77.07**

for a total of 100 epochs across 4 GPUs. The total batch size is set at 256, and the learning rate is decreased by a factor of 0.1 every 30 epochs. The top-1 accuracy results of ResNet18/50 are displayed in Table 3.5. While the same parameters prove effective for KFAC, SKFAC struggles to optimize the network despite the random sampling number being increased to 8, which could be due to information loss. In contrast, our NKFAC method successfully inherits most of the valuable information from KFAC, and with the help of our useful implementations and well-tuned parameters for first-order optimizers, it achieves the best performance among all optimizers with 0.61% and 0.41% higher than the second best one.

Results on Transformers: During our tests, we utilize Geforce RTX A6000 GPUs and the official MMClassification toolbox [13] [1] to train the Swin-T and Swin-B transformers. Since the adaptive momentum optimizers are widely used for training transformers, we apply the adaptive momentum optimizer AdaNKFAC and compare it with the default optimizer AdamW, which we directly cite the official results of in Table 3.6 [2]. For AdaNKFAC, we set the initial learning rate and weight decay

[1] `https://github.com/open-mmlab/mmclassification`.

[2] The results can be found from `https://github.com/open-mmlab/mmclassification/tree/master/configs/swin_transformer`.

Table 3.6: Top 1 accuracy (%) with Swin-T and Swin-B.

Optimizer	AdamW	AdaNKFAC
Swin-T	81.18	**81.63**
Swin-B	**83.36**	83.30

to be 0.002 and 0.025, respectively, with other hyper-parameters keeping the same as default settings. Our AdaNKFAC inherits the quick convergence property as a second-order optimizer for both networks and converges faster than AdamW (as seen in Fig. 3.4 for the training loss and validating accuracy curves). By the final validation performance in Table 3.6, we see AdaNKFAC gains 0.45% and loses 0.06% on Swin-T and Swin-B, respectively, compared with the default AdamW. These results demonstrate the suitability of the second-order algorithm for training transformers. More experiment results on the training of transformers will be presented in Section 3.4.4.

3.4.4 Results on COCO

In this section, we will show the applicability and robustness of NKFAC on detection and segmentation tasks on COCO [44], which is a large-scale dataset of detection, segmentation and captioning tasks. In this section, we accomplish experiments with Faster-RCNN [64], RetinaNet [43] and Mask-RCNN [28] with backbones ResNet50, ResNet101 and Swin-T transformer [46] to show the stable and efficient performance of our NKFAC/AdaNKFAC on these tasks. Our experiments are accomplished under the sketch of the official MMDetection toolbox [12] [3]. For the default optimizers, if the official results are given, we cite the official results directly for comparison [4]. Otherwise, we adopted the official settings in MMDetection toolbox and reproduced

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

[4] The results can be found from `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`.

the results [5]. Our experiments here are accomplished on RTX A6000, Geforce RTX 3090Ti and Quadro RTX 8000 GPUs.

The hyperparameters of NKFAC/AdaNKFAC are the same as the default optimizers in the MMDetection toolbox without any tuning. Specifically, for NKFAC in Faster-RCNN and Mask-RCNN, the initial learning rate takes 0.02 while the weight decay takes 0.0001, and for RetinaNet, the learning rate takes 0.01 while the weight decay takes 0.0001. For AdaNKFAC, the initial learning rate takes 0.0001 while the weight decay takes 0.2. Our experimental results show that NKFAC performs much better than the default optimizers in all the experiments regardless of model and backbone. Table 3.7 and Table 3.8 report the Average Precision (AP) of detection by Faster-RCNN and RetinaNet, respectively, from which we can see NKFAC improves the AP by 1.9% $\sim$ 3.4% compared to the default SGDM. Meanwhile, in Table 3.9, we report the detection and segmentation results of Mask-RCNN with different backbones, while NKFAC gains 0.9% $\sim$ 1.9% AP^b and 0.7% $\sim$ 1.9% AP^m over the default optimizers. These results can well demonstrate the effectiveness of our proposed NKFAC.

Further considering the convergence performance, we plot Figure 3.5 to clearly show the training loss curves of different optimizers on Faster-RCNN with ResNet101 backbone, RetinaNet with Swin-T backbone, Mask-RCNN with ResNet101 and Swin-T backbone. It can be shown from Figure 3.5 that NKFAC/AdaNKFAC converges faster than the default optimizer in all the cases. Therefore, we are optimistic about the application of NKFAC on various tasks.

[5]For Swin-T 2× of Mask-RCNN, the decay epochs of learning rate are set to be 16 and 22, with total 24 epochs.

Table 3.7: Detection results of Faster-RCNN on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one.

Backbone, LR	Algorithm	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
ResNet50, 1×	SGDM*	37.4	58.1	40.4	21.2	41.0	48.1
	NKFAC	39.7	60.7	43.0	23.4	43.3	51.5
	Δ	↑2.3	↑ 2.6	↑2.6	↑2.2	↑2.3	↑3.4
ResNet101, 1×	SGDM*	39.4	60.1	43.1	22.4	43.7	51.1
	NKFAC	41.3	62.0	45.0	24.4	44.7	54.9
	Δ	↑1.9	↑ 1.9	↑1.9	↑2.0	↑1.0	↑3.8

Table 3.8: Detection results of RetinaNet on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one.

Backbone, LR	Algorithm	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
Swin-T, 1×	SGDM*	37.3	57.4	39.6	22.2	40.7	50.6
	NKFAC	40.7	61.7	43.3	25.2	43.9	53.9
	Δ	↑3.4	↑ 4.3	↑3.7	↑3.0	↑ 3.2	↑ 3.3

3.4.5 Ablation Studies

In this section, we report some ablation studies on NKFAC with respect to the hyper-parameters on CIFAR100[38]. Our experiments are accomplished on 8 Geforce RTX 2080Ti GPUs on the same server. Besides the parameter we focus on, other hyper-parameters and are identical to those in Section 3.4.2. Here, we mainly focus on the dampening parameter η, statistics momentum α, the number of Newton's iterations K, and the updating intervals T_{stat} and T_{inv}.

Table 3.10 shows the result with different dampening parameters η. As we mentioned before, the dampening parameter matters a lot in a second-order optimizer, and here $\eta = 0.01$ obtains the best result while $\eta = 0.001$ reduces the performance by 1.00%. Table 3.11 shows the result with different second-order statistics momentum α. Benefiting from the fact that the Fisher information matrix (FIM) changes little between iterations, NKFAC is stable related to α, and all the momentum parameters α achieve good results. To prevent a greater change of FIM on larger datasets, we finally take a moderate choice $\alpha = 0.95$.

Table 3.9: Detection and segmentation results of Mask-RCNN on COCO. * means the default optimizer, and Δ means the improvement of NKFAC compared with the default one.

Backbone, LR	Algorithm	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
ResNet50, 1×	SGDM*	38.2	58.8	41.4	34.7	55.7	37.2
	NKFAC	40.1	60.9	43.9	36.6	57.9	39.2
	Δ	↑1.9	↑ 2.1	↑2.5	↑1.9	↑2.2	↑2.0
ResNet101, 1×	SGDM*	40.0	60.5	44.0	36.1	57.5	38.6
	NKFAC	41.8	62.1	45.9	37.8	59.2	40.3
	Δ	↑1.8	↑ 1.6	↑1.9	↑1.7	↑1.7	↑1.7
Swin-T, 1×	AdamW*	42.7	65.2	46.8	39.3	62.2	42.2
	AdaNKFAC	43.6	65.7	47.5	40.1	62.9	43.4
	Δ	↑0.9	↑0.5	↑0.7	↑0.8	↑0.7	↑1.2
Swin-T, 2×	AdamW*	45.2	67.3	49.4	41.0	64.3	44.0
	AdaNKFAC	46.2	68.2	50.5	41.7	65.1	45.0
	Δ	↑1.0	↑0.9	↑1.1	↑0.7	↑0.8	↑1.0
Swin-T, 3×	AdamW*	46.0	68.2	50.3	41.6	65.3	44.7
	AdaNKFAC	46.9	68.6	51.5	42.3	65.8	45.6
	Δ	↑0.9	↑0.4	↑1.2	↑0.7	↑0.5	↑0.9

Table 3.10: Testing accuracy (Acc.) of NKFAC with different dampening parameter η for ResNet50 on CIFAR100.

η	0.1	0.01	0.001	0.0001
Acc. (%)	$81.74 \pm .27$	$\mathbf{81.78} \pm .06$	$80.78 \pm .13$	$79.69 \pm .05$

Table 3.12 reports the testing results with different numbers of Newton's step K, where the time shown is the cost of inversion plus adaptive dampening, and is reported by taking the average. When K takes 2, 3 or 5, the accuracy becomes better, while more steps result in more time cost. As a balance, we finally take $K = 1$. Meanwhile, we report Table 3.13 about the testing accuracy of different updating interval of the second-order statistics T_{stat} and the inversion T_{inv}. Within the range listed in the table, NKFAC performs stable and satisfactory, so one can choose these intervals according to different needs.

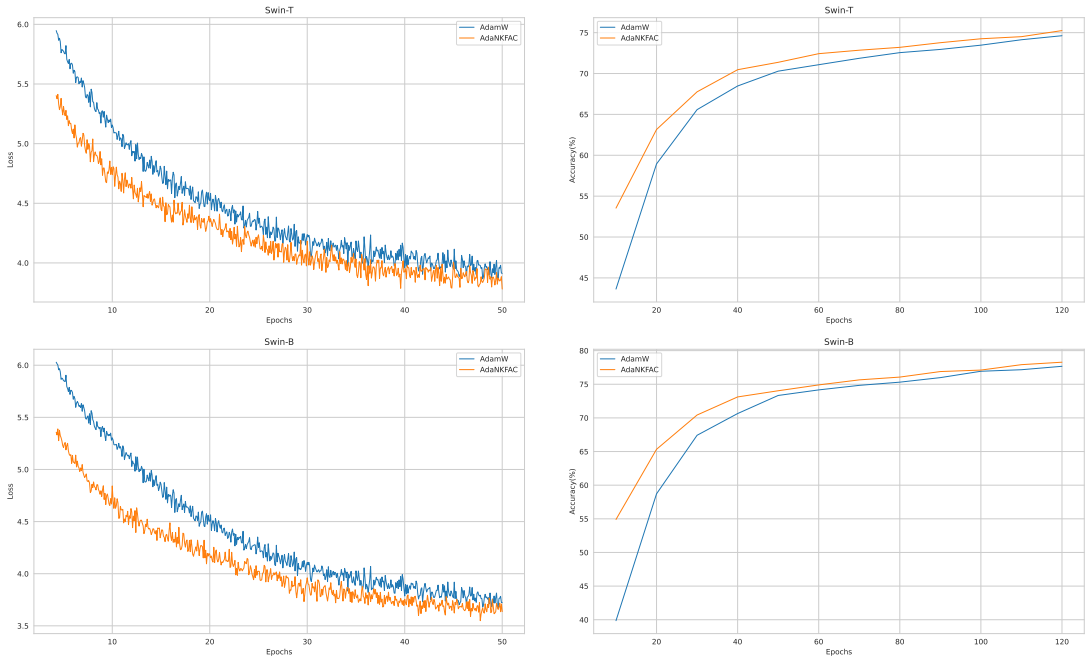


Figure 3.4: Training loss and validating accuracy curves w.r.t epoch for different optimizers on ImageNet with Swin-T/Swin-B.

Table 3.11: Testing accuracy (Acc.) of NKFAC with different statistics momentum α for ResNet50 on CIFAR100.

α	0.9	0.95	0.995	0.9995
Acc. (%)	$81.82 \pm .15$	$81.78 \pm .06$	$81.84 \pm .12$	$\mathbf{81.95} \pm .16$

3.4.6 Ablation Study on the stepsize of Newton's iteration

In this section we conduct the ablation study on the stepsize of Newton's iteration. Recall that for a given real matrix $\mathbf{A}$, our **Algorithm 2** (i.e., the Newton's method for solving matrix inverse) obtains a sequence $\{\mathbf{B}_k\}$ that performed simply by

$$\mathbf{B}_{k+1} = (1 + \alpha_{k+1})\mathbf{B}_k - \alpha_{k+1}\mathbf{B}_k\mathbf{A}\mathbf{B}_k, \tag{3.10}$$

with $\alpha_{k+1} = 1$ when $\|\mathbf{I} - \mathbf{A}\mathbf{B}_0\| < 1$ and $\alpha_{k+1} = \frac{1}{\|\mathbf{A}\mathbf{B}_k\|}$ when $\|\mathbf{I} - \mathbf{A}\mathbf{B}_0\| > 1$. When choosing α_{k+1} in the latter case, the most natural idea is to use the length that drops the deviation most along this descent direction as the stepsize, specifically, as in the below Proposition 3.2.

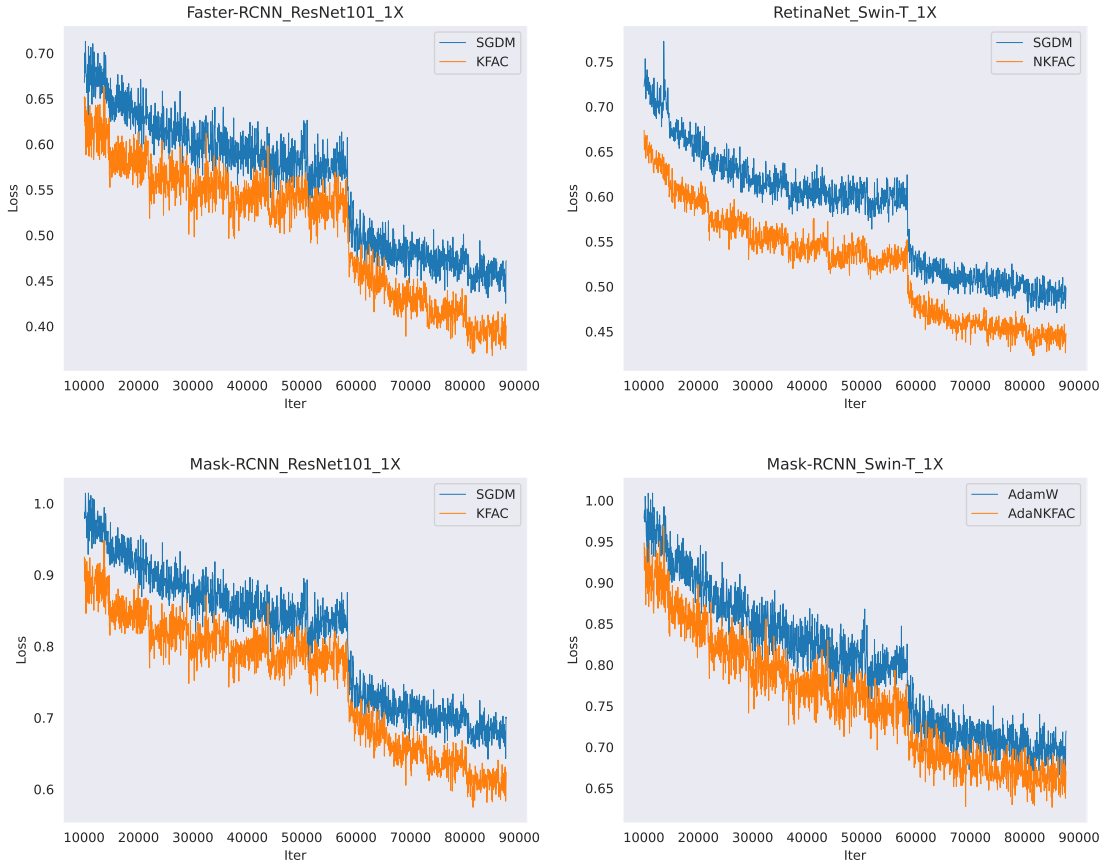


Figure 3.5: Loss curves w.r.t iteration for different deep neural networks with different optimizers on COCO.

Proposition 3.2. *Denote the descent direction* $\mathbf{D}_k := \mathbf{B}_k(\mathbf{A}\mathbf{B}_k - \mathbf{I})$ *and the function* $f_k(\alpha) := \frac{1}{2}\|\mathbf{A}(\mathbf{B}_k - \alpha\mathbf{D}_k) - \mathbf{I}\|^2$ *for each iteration* k*. Then the optimization problem* $\alpha^*_{k+1} := \arg\min_\alpha f_k(\alpha)$ *takes the optimal* $\alpha^*_{k+1} = \frac{\langle \mathbf{A}\mathbf{B}_k - \mathbf{I}, \mathbf{A}\mathbf{B}_k(\mathbf{A}\mathbf{B}_k - \mathbf{I})\rangle}{\|\mathbf{A}\mathbf{B}_k(\mathbf{A}\mathbf{B}_k - \mathbf{I})\|^2}$ *in iteration* k*.*

However, our experiments show that this α^*_{k+1} may not works as good as $\alpha_{k+1} = \frac{1}{\|\mathbf{A}\mathbf{B}_k\|}$, which is chosen from our experience. Here we show some of the comparison results on detection and segmentation tasks in Table 3.14 and Table 3.15. In summary, the stepsize chosen in Newton's iteration indeed leave impact on the performance of NKFAC. Luckily, our stepsize $\alpha_{k+1} = \frac{1}{\|\mathbf{A}\mathbf{B}_k\|}$ that chosen in **Algorithm 2** performs good throughout our experiments.

Table 3.12: Testing accuracy (Acc.) and time cost by computing inversion of NKFAC with different number of Newton's step K.

K	1	2	3	5	10
Acc. (%)	$81.78 \pm .06$	$\mathbf{81.93} \pm .21$	$81.91 \pm .11$	$81.92 \pm .18$	$81.79 \pm .16$
Time (s)	**132.00**	202.37	268.00	405.62	747.88

Table 3.13: Testing accuracy (Acc.) of NKFAC with different updating intervals T_{stat} and T_{inv} for ResNet50 on CIFAR100.

T_{stat}	20	50	100	200
T_{inv}	200	500	1000	2000
Acc. (%)	$81.78 \pm .06$	$81.74 \pm .18$	$\mathbf{81.90} \pm .26$	$81.80 \pm .13$

3.4.7 Ablation Study on Implementations

With our experiment results in Section 3.4.2, we are motivated to add our implementations to KFAC and SKFAC, denoted by KFAC* and SKFAC*, respectively. In this section, we compare the implemented KFAC*, SKFAC* with NKFAC on CIFAR100[38] and ImageNet[68] datasets.

Here, the learning rate and weight decay of KFAC*, SKFAC* and NKFAC are set to be the same, specifically, 0.05 and 0.001, respectively. We see NKFAC still shows its advantage of time cost from Table 3.16 and achieves higher generalization performances than SKFAC*, while stably reducing the inversion cost by 56% and 64%. KFAC* shows its efficiency benefited from our useful implementations and achieves the highest accuracy in ResNet50 while NKFAC is still the best in DenseNet121. Thus, as a balance, NKFAC can be a good choice in applications.

To clearly compare the time cost of inversion and adaptive dampening in each step, we plot Figure 3.6 to show the time cost proportion of each part in a single optimization step using the experiment results on CIFAR100. To eliminate randomness, the time reported in Figure 3.6 is the cumulative time of the first 50 epochs in the training process. SKFAC can also shorten the time in computing the adaptive

Table 3.14: Detection results of Faster-RCNN on COCO. Δ means the improvement of α_{k+1} compared with α^*_{k+1}.

Backbone, LR	Stepsize	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
ResNet50, 1×	α^*_{k+1}	38.7	59.5	42.0	22.9	42.3	50.3
	α_{k+1}	39.7	60.7	43.0	23.4	43.3	51.5
	Δ	↑1.0	↑1.2	↑1.0	↑0.5	↑1.0	↑1.2
ResNet101, 1×	α^*_{k+1}	39.1	59.5	42.6	22.6	42.9	51.2
	α_{k+1}	41.3	62.0	45.0	24.4	44.7	54.9
	Δ	↑2.2	↑ 2.5	↑2.4	↑1.8	↑1.8	↑3.7

Table 3.15: Detection and segmentation results of Mask-RCNN on COCO. Δ means the improvement of α_{k+1} compared with α^*_{k+1}.

Backbone, LR	Algorithm	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
ResNet50, 1×	α^*_{k+1}	39.5	60.4	43.1	36.3	57.5	38.9
	α_{k+1}	40.1	60.9	43.9	36.6	57.9	39.2
	Δ	↑0.6	↑ 0.5	↑0.8	↑0.3	↑0.4	↑0.3
ResNet101, 1×	α^*_{k+1}	41.4	61.9	45.4	37.4	59.0	40.2
	α_{k+1}	41.8	62.1	45.9	37.8	59.2	40.3
	Δ	↑0.4	↑ 0.2	↑0.5	↑0.4	↑0.2	↑0.1

dampening after dimension reduction, as shown in Figure 3.6, while NKFAC shows the least time in the optimization step for both the DNNs tested.

3.5 Conclusion

In this work, we propose NKFAC, a generalized KFAC optimizer that combining classic KFAC with Newton's iteration. With the help of a good initialization obtained from the last iteration and the fast local convergence of Newton's iteration, we reduce the computational cost of the second-order statistics inversion while maintaining satisfactory performance. Meanwhile, since Newton's iteration contains only matrix product operations, the probability of numerical problems can be reduced and the stability of inversion can be improved. Moreover, without random selection or taking an average to reduce the dimension of the second-order statistics, we

Table 3.16: Testing accuracies (%) and time cost by computing dampening and inversion (s) of different optimizers **with implementations** on CIFAR100.

	Accuracy			Time Cost		
Optimizer	KFAC*	SKFAC*	NKFAC	KFAC*	SKFAC*	NKFAC
DenseNet121	81.04 ± .22	80.66 ± .13	**81.13** ± .16	362.14	331.04	**132.00**
ResNet50	**81.89** ± .21	81.08 ± .18	81.78 ± .06	306.88	191.33	**134.21**

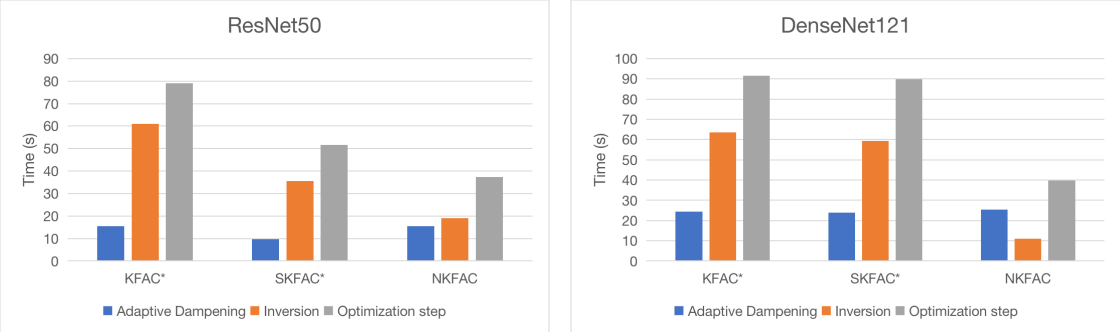


Figure 3.6: Time cost of optimization steps for KFAC*, SKFAC* and NKFAC on CIFAR100.

preserve the information obtained from the samples to the greatest extent, ensuring a speedy convergence rate. Last but not the least, through some useful implementations, the generalization performance of NKFAC achieves SOTA without tedious hyper-parameter tuning. NKFAC reduces the gap between first-order and second-order methods in terms of time cost and hyper-parameters tuning, while still keeping satisfactory convergence performance. Sufficient experiments on image classification, detection and segmentation tasks have been conducted to confirm the effectiveness of our proposed NKFAC.

Chapter 4

AdamR: Adaptive Learning Rate Optimization Method with a Rotation Transformation

The advancement of adaptive learning rate methodologies has been instrumental in the optimization of Deep Neural Networks (DNNs). In pursuit of attaining a more favorable regret bound, we propose the integration of a rotation transformation into the existing adaptive learning rate algorithms. We employ the widely-recognized adaptive learning rate optimization method AdamW as a based optimizer, leading to the development of a novel optimizer we have named AdamR. It consists of three steps in each iteration to compute the modified gradient: firstly, the computation of the gradient with a rotation; secondly, the execution of the standard Adam step; and finally, the reorientation of the gradient back to its original space. Theoretically, AdamR exhibits a lower regret bound compared to other adaptive learning rate methods that focus solely on the diagonal elements of the preconditioned matrix. An important characteristic of the rotation transformation employed in AdamR is its preservation of the gradient norm, thereby allowing AdamR to seamlessly inherit the hyper-parameters and inherent advantages of AdamW. Similar to the optimizer AdaBK, we also employ the layer-wise block-diagonal constraint and the Kronecker-

factorized constraint on the rotation transformation to reduce the computation cost in optimizing DNNs. Meanwhile, to further enhance the efficiency of AdamR, the computation of these two statistics is conducted sporadically, such as once every 1000 iterations. The experimental results on image classification, object detection and segmentation have demonstrated AdamR's superior performance in accelerating the training process and improving the generalization capability.

4.1 Introduction

Designing a satisfying optimizer for training deep neural networks (DNNs) has been a research spotlight in the past few decades. In designing optimizers, on the one hand, it is necessary to maximize the use of data and network information to obtain satisfactory performance. On the other hand, limiting the calculation flops and memory consumption is also required. Stochastic gradient descent (SGD) [66, 61], as the fundamental optimization algorithm in deep learning, has achieved remarkable performance in many area [29, 63, 3, 49, 74]. Adaptive learning rate methods have been developed to mine more valuable information in DNNs optimization. Aiming at achieving lower regret bound, Duchi *et al*. [17] proposed AdaGrad that affiliates an adaptive learning rate for each parameter independently, which pioneered and inspired the adaptive learning rate methods. Specifically, because AdaGrad increases the effective learning rate during training, it usually performs poorly in practice. The later proposed algorithms (e.g., RMSProp [81], Adam [36], AdamW [47], RAdam [45], Adabelief [101] and Ranger [45, 98, 91]) implement the series of adaptive learning rate methods with better utilization of training information and more satisfactory performance, making adaptive stepsize methods the most popular kind of optimizer for many tasks [97, 42, 15, 19, 83, 83, 46]. Besides, as the more general version of adaptive stepsize methods, full-matrix preconditioned gradient optimizers

are also under careful research. Although attaining lower regret bound [17] and better mining the data information, full-matrix preconditioned gradient methods often encounter the problem of expensive computation and storage, violating the limitation we mentioned initially. Therefore, some structure priors of the full-matrix are usually introduced to solve the problem [2, 94, 26]. However, they outperform the adaptive learning rate methods little in both the efficiency and generalization performance.

From the aspect of integrating more useful information and attaining a better regret bound, a very recent work [92] proposed AdaBK, which is derived with the help of minimizing the regret bound under given constraints. In practice, AdaBK is embedded in both SGDM and AdamW (named SGDM_BK and AdamW_BK, respectively) and is observed to have state-of-the-art generalization performance. However, although AdaBK itself is proved to attain a lower regret bound, no such theoretical analysis is guaranteed on the proposed SGDM_BK and AdamW_BK, which, as stated in [92], seems like a heuristic combination. Moreover, from the view of adaptive stepsize methods, AdamW_BK actually utilizes the adaptive stepsize computed by AdamW, which is implemented by the gradient norm recovery technique. Therefore, AdaBK only contributes to a better descent direction but has an additional undesirable impact on the length (equivalently, stepsize) of the descent direction, which requires correction by performing gradient norm recovery. Consequently, it is not very natural to embed and generate an adaptive stepsize method under the analysis of AdamW_BK.

Back to adaptive learning rate methods, since their effectiveness has been testified in various applications [97, 42, 15, 19, 83] and their hyper-parameters (*e.g.*, learning rate and weight decay) are well-tuned, we aim to introduce the information of full-matrix preconditioned gradient into them without changing their implementation way and hyperparameters, while the computation and storage cost are both acceptable in practice. To achieve this, we propose introducing a rotation transformation

into the adaptive learning rate methods. Precisely, we first rotate the gradient vector to a specific space with an orthogonal matrix and then implement the adaptive learning rate methods (*e.g.*, Adam) to get the primary preconditioned gradient. Finally, we rotate it back into the original gradient space as the final preconditioned gradient. Theoretically, the adaptive learning rate methods with a proper orthogonal matrix are equivalent to the full-matrix preconditioned gradient, which has been proven to have a lower regret bound in the existing works. In practice, this approach can fully inherit the advantages of original adaptive learning rate methods and their well-tuned hyper-parameters. Meanwhile, to make it practical according to computation and storage cost, we learn from AdaBK, adopting the block-diagonal and Kronecker-factorized constraints for the orthogonal matrix. We apply it to Adam and name the obtained optimization algorithm AdamR. Extensive experiments on image classification, object detection and segmentation show the effectiveness of the proposed AdamR.

Additional notations. In our content, $\mathbf{w}_t$ and $\mathbf{g}_t$ are the weight vector and its gradient of a DNN model in the t-th iteration. $\mathbf{g}_{ti}$ is the gradient of the i-th sample in the t-th iteration batch, and $\mathbf{g}_t = \frac{1}{n}\sum_{i=1}^{n}\mathbf{g}_{ti}$, where n is the batch size.

4.2 Preliminaries

Nowadays, a useful and powerful framework to theoretically analyze the optimization algorithms in deep learning is the online convex optimization framework [72, 27]. Specifically, one can evaluate the current existing algorithms by analyzing the online regret bound, and obtain a suitable optimization algorithm by minimizing the bound. The definition of regret is stated in Definition 4.1.

Definition 4.1. *[72, 27] For an arbitrarily given sequence of convex loss functions* $\{f_1(\mathbf{w}), f_2(\mathbf{w}), \ldots, f_T(\mathbf{w})\}$*, the regret on the t-th iteration is defined by*

$$R(T) = \sum_{t=1}^{T} \left(f_t(\mathbf{w}_t) - f_t(\mathbf{w}^*) \right), \tag{4.1}$$

where $\mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_{t=1}^{T} f_t(\mathbf{w})$.

In training DNNs, the popular preconditioned gradient methods adopt the following updating formula

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{H}_t^{-1} \mathbf{g}_t, \tag{4.2}$$

where $\mathbf{H}_t^{-1}\mathbf{g}_t$ is the preconditioned gradient. The preconditioned gradient update Eq. (4.2), from the view of stochastic optimization, can also been analyzed by the online convex optimization framework under some convex assumptions, which means that one can manually design the preconditioned matrix $\mathbf{H}_t$ to achieve a low regret.

Most existing works (e.g., AdaGrad) manually design the sequence of precondition matrices $\mathbf{H}_1$, $\mathbf{H}_2$,...,$\mathbf{H}_T$, according to experience. To our knowledge, AdaBK [92] is the first work designing an optimizer by minimizing the upper bound of a guide function obtained from some specific constraints. By block-diagonal and Kronecker-factorized constraints, the cost of AdaBK becomes reasonable, and the performance of the embedded SGDM_BK and AdamW_BK are both satisfactory compared to SGDM and AdamW, respectively. Besides, a general regret bound is also given in [92], of which the (full-matrix) AdaGrad and the proposed AdaBK are both special cases. Specifically, for a general constraint set Ψ for $\mathbf{H}_t$, if the set $\Psi \subseteq \mathbb{R}^{d\times d}$ is a cone (*i.e.*, $\forall \mathbf{x} \in \Psi, \theta > 0$, $\theta\mathbf{x} \in \Psi$ holds), the following Theorem4.1 holds.

Theorem 4.1. *[92, Theorem 1] For any cone constraint $\Psi \subseteq \mathbb{R}^{d\times d}$, define the guide function $F_T(\mathbf{S})$ on Ψ as*

$$F_T(\mathbf{S}) = \sum_{t=1}^{T} (||\mathbf{g}_t||_{\mathbf{S}}^*)^2, \tag{4.3}$$

and define

$$\mathbf{S}_T = \arg \min_{\mathbf{S}\in\Psi, \mathbf{S}\geq\mathbf{0}, Tr(\mathbf{S})\leqslant 1} F_T(\mathbf{S}), \quad \mathbf{H}_T = C_T \mathbf{S}_T, \tag{4.4}$$

where $C_T = \sqrt{F_T(\mathbf{S}_T)}$*, then the regret of online mirror descent holds that*

$$R(T) \leqslant (\frac{D^2}{2\eta} + \eta) C_T = (\frac{D^2}{2\eta} + \eta) \sqrt{\min_{\mathbf{S} \in \Psi, \mathbf{S} \geq \mathbf{0}, Tr(\mathbf{S}) \leqslant 1} F_T(\mathbf{S})}. \tag{4.5}$$

This theorem reveals that minimizing the guide function $F_T(\boldsymbol{S})$ on cone Ψ is equivalent to minimizing the regret bound of the preconditioned gradient descent algorithm simultaneously. Moreover, with Theorem 4.1, one can easily derive the matrix $\mathbf{H}_T$ under a given constraint, and the key problem left is how to investigate proper constraints on $\mathbf{H}_T$ or $\mathbf{S}_T$.

Yong *et al*. [92] suggest using layer-wise block-diagonal constraint and the Kronecker-factorized constraint on $\mathbf{H}_T$ or $\mathbf{S}_T$, which leads to AdaBK optimizer. To end up this section, we cite two key lemmas that will play a role in our proofs later in Section 4.3.3. Denote $\mathbf{g} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{g}_i$, where $\mathbf{g}_i$ is the gradient of each simple and n is the batch size, and $\mathbf{g}_i = \mathrm{vec}(\boldsymbol{\delta}_i \mathbf{x}_i^\top) = \boldsymbol{\delta}_i \otimes \mathbf{x}_i$, where $\boldsymbol{\delta}_i$ and $\mathbf{x}_i$ is the output feature gradient and input feature of sample i, respectively. Assume for a given matrix $\mathbf{S}$, $\mathbf{S} = \mathbf{S}_1 \otimes \mathbf{S}_2$. Then the following two lemmas hold.

Lemma 4.1. *[92, Lemma 3] The following inequality holds*

$$F_T(\mathbf{S}) \leqslant \frac{1}{n} Tr(\mathbf{S}_1^{-1} \boldsymbol{L}_T) \, Tr(\mathbf{S}_2^{-1} \boldsymbol{R}_T), \tag{4.6}$$

where $\boldsymbol{L}_T = \sum_{t=1}^{T} \sum_{i=1}^{n} \boldsymbol{\delta}_{ti} \boldsymbol{\delta}_{ti}^\top$ *and* $\boldsymbol{R}_T = \sum_{t=1}^{T} \sum_{i=1}^{n} \boldsymbol{x}_{ti} \boldsymbol{x}_{ti}^\top$.

Lemma 4.2. *[92, Lemma 4] If* $\boldsymbol{A} \succ \mathbf{0}$*, then*

$$\arg \min_{\mathbf{S} \geq \mathbf{0}, Tr(\mathbf{S}) \leqslant 1} Tr(\mathbf{S}^{-1} \boldsymbol{A}) = \boldsymbol{A}^{\frac{1}{2}} / Tr(\boldsymbol{A}^{\frac{1}{2}}). \tag{4.7}$$

4.3 Methodology

4.3.1 Rotation Transformation

As we mentioned in Section 4.2, investigating proper constraints can naturally lead to a new different optimizer. Although AdaBK has been embedded into AdamW

and resulted in AdamW_BK, more implementations (*e.g.*, gradient norm recovery) are needed to guarantee its good performance and similar hyper-parameters as AdamW. Motivated by AdaBK and Theorem 4.1 that inspire us to find proper constraints on $\mathbf{H}_T$ or $\mathbf{S}_T$, we introduce a rotation transformation into the commonly used adaptive learning rate methods, which essentially derives an adaptive stepsize method more naturally that can achieve lower regret bounds and maintain the merits of adaptive learning rate methods without so many implementations.

In spite of the diagonal constraint, we aim to find a proper constraint that can inherit the advantages of the existing adaptive learning rate methods. The first and the most natural one is the diagonal constraint since the original AdaGrad is derived from the diagonal constraint on the matrix $\mathbf{S}_T$, and this constraint is inherited in the whole series of adaptive learning rate methods, which usually perform well in practice. Actually, from the view of matrix analysis, the SVD decomposition of a given matrix will definitely generate a diagonal matrix. Specifically, for a matrix $\mathbf{S}_T$ with SVD decomposition $\mathbf{S}_T = \mathbf{U}\mathbf{D}\mathbf{U}^\top$, matrix $\mathbf{D}$ is indeed diagonal. Ideally, we hope there is no further constraint on the diagonal elements of $\mathbf{D}$ (except $\mathbf{D} \succeq \mathbf{0}$). However, due to the constraint on $\mathbf{S}_T$, its singular value will not be free, which is inconsistent with AdaGrad. To address this, we state the following Lemma 4.3 with proof.

Lemma 4.3. *With $F_T(\cdot)$ defined in Eq.* (4.3)*, it holds that*

$$F_T(\mathbf{S}_T) \geqslant \min_{\mathbf{D}=Diag(d),\mathbf{d}\geq\mathbf{0},1^T\mathbf{d}\leqslant 1} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top), \tag{4.8}$$

where $\mathbf{S}_T$ is defined in Eq (4.4), and $\mathbf{U}$ is the singular vector matrix of $\mathbf{S}_T$.

Proof.

$$\begin{aligned} F_T(\mathbf{S}_T) &= \min_{\mathbf{S}\in\Psi,\mathbf{S}\succeq\mathbf{0},\mathrm{Tr}(\mathbf{S})\leqslant 1} F_T(\mathbf{S}) \\ &= \min_{\mathbf{U}\mathbf{D}\mathbf{U}^\top\in\Psi,\mathbf{D}=\mathrm{Diag}(d),\mathbf{d}\geq\mathbf{0},1^\top\mathbf{d}\leqslant 1} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top) \\ &\geqslant \min_{\mathbf{D}=\mathrm{Diag}(d),\mathbf{d}\geq\mathbf{0},1^T\mathbf{d}\leqslant 1} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top). \end{aligned} \tag{4.9}$$

□

For a given $\mathbf{S}_T$ defined in Eq (4.4), Lemma 4.3 can lead to a new preconditioned gradient algorithm which has lower regret bound with the singular vector matrix of $\mathbf{S}_T$. Thus, the problem has been converted into two individual sub-problems, the first one is to solve the diagonal matrix $\mathbf{D}$ from Eq (4.8), and the second one is to solve $\mathbf{S}_T$ from Eq (4.4) and compute its singular vector matrix (orthogonal matrix) $\mathbf{U}$. We will address these two sub-problems in Section 4.3.2 and Section 4.3.3 separately.

4.3.2 Solving the Diagonal Matrix

To solve the diagonal matrix $\mathbf{D}$, we state the following Lemma 4.4 with proof.

Lemma 4.4. *For any orthogonal matrix* $\mathbf{U}$*, suppose that*

$$C_T = \sqrt{\min_{\mathbf{D}=Diag(d),\mathbf{d}\geq\mathbf{0},1^\top\mathbf{d}\leqslant 1} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top)}, \tag{4.10}$$

then we have

$$\begin{aligned} \mathbf{D}_T =& \arg\min_{\mathbf{D}=Diag(d),\mathbf{d}\geq\mathbf{0},1^\top\mathbf{d}\leqslant C_T} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top) \\ =& Diag\big(\big(\sum_{t=1}^{T}\widehat{\mathbf{g}}_t\odot\widehat{\mathbf{g}}_t\big)^{\odot\frac{1}{2}}\big), \end{aligned} \tag{4.11}$$

where $\widehat{\mathbf{g}}_t = \mathbf{U}^\top\mathbf{g}_t$.

Proof. Suppose

$$\widehat{\mathbf{d}}_T = \arg\min_{\mathbf{D}=\mathrm{Diag}(d),\mathbf{d}\geq\mathbf{0},1^\top\mathbf{d}\leqslant 1} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top), \tag{4.12}$$

we can know that

$$\begin{aligned}\mathbf{d}_T &= \arg\min_{\mathbf{D}=\mathrm{Diag}(d),\mathbf{d}\geq\mathbf{0},1^\top\mathbf{d}\leqslant C_T} F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top) \\ &= C_T \arg\min_{\hat{\mathbf{D}}=\mathrm{Diag}(\hat{d}),\hat{\mathbf{d}}\geq\mathbf{0},1^\top\hat{\mathbf{d}}\leqslant 1} F_T(\mathbf{U}\hat{\mathbf{D}}\mathbf{U}^\top) \\ &= C_T\hat{\mathbf{d}}_T.\end{aligned} \tag{4.13}$$

We see that to get $\mathbf{d}_T$, we only need to solve $\widehat{\mathbf{d}}_T$.

Because

$$\begin{aligned}F_T(\mathbf{U}\mathbf{D}\mathbf{U}^\top) &= \sum_{t=1}^{T}(||\mathbf{g}_t||^*_{\mathbf{U}\mathbf{D}\mathbf{U}^\top})^2 = \sum_{t=1}^{T}\mathbf{g}_t^\top(\mathbf{U}\mathbf{D}\mathbf{U}^\top)^{-1}\mathbf{g}_t \\ &= \sum_{t=1}^{T}\mathbf{g}_t^\top\mathbf{U}\mathbf{D}^{-1}\mathbf{U}^\top\mathbf{g}_t = \sum_{t=1}^{T}\widehat{\mathbf{g}}_t^\top\mathbf{D}^{-1}\widehat{\mathbf{g}}_t \\ &= \sum_{t=1}^{T}\sum_{i=1}^{d}\frac{\widehat{g}_{ti}^2}{d_i},\end{aligned} \tag{4.14}$$

where $\widehat{\mathbf{g}}_t = \mathbf{U}^\top\mathbf{g}_t$. we have

$$\widehat{\mathbf{d}}_T = \arg\min_{\mathbf{d}\geq 0,\mathbf{1}^\top\mathbf{d}\leqslant 1}\sum_{t=1}^{T}\sum_{i=1}^{d}\frac{\widehat{g}_{ti}^2}{d_i}, \tag{4.15}$$

where $\mathbf{d} \succeq 0$ means all the coefficients of vector $\mathbf{d}$ are non-negative. By introducing multipliers $\lambda \succeq \mathbf{0}$ and $\theta \geqslant 0$, we can have the Lagrangian of the above constrained optimization problem

$$\mathcal{L}(\mathbf{d},\lambda,\theta) = \sum_{t=1}^{T}\sum_{i=1}^{d}\frac{\widehat{g}_{ti}^2}{d_i} - \langle\lambda,\mathbf{d}\rangle + \theta(\mathbf{1}^\top\mathbf{d}-1). \tag{4.16}$$

Taking the partial derivatives w.r.t. d_i, we have

$$\frac{\partial\mathcal{L}(\mathbf{d},\lambda,\theta)}{\partial d_i} = -\sum_{t=1}^{T}\frac{\widehat{g}_{ti}^2}{d_i^2} - \lambda_i + \theta = 0. \tag{4.17}$$

Obviously, $d_i \neq 0$, and according to the complementary conditions, we know $\lambda_i = 0$. Then, we have

$$\widehat{d}_{Ti} = \theta^{-\frac{1}{2}} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}. \tag{4.18}$$

With the constraint $\mathbf{1}^\top \mathbf{d} \leqslant 1$, we can choose a proper θ so that

$$\widehat{d}_{Ti} = \frac{(\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}}{\sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}} \tag{4.19}$$

meets the constraint. Meanwhile, we can derive that

$$\begin{aligned} C_T &= \sqrt{\sum_{t=1}^{T} \sum_{i=1}^{d} \frac{\widehat{g}_{ti}^2}{\widehat{d}_{Ti}}} \\ &= \sqrt{\sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}} \sum_{i=1}^{d} \sum_{t=1}^{T} \frac{\widehat{g}_{ti}^2}{(\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}}} \\ &= \sqrt{\sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}} \sum_{i=1}^{d} \frac{\sum_{t=1}^{T} \widehat{g}_{ti}^2}{(\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}}} \\ &= \sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}. \end{aligned} \tag{4.20}$$

Therefore,

$$\begin{aligned} \mathbf{d}_{Ti} &= C_T \hat{\mathbf{d}}_{Ti} \\ &= \sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}} \frac{(\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}}{\sum_{i=1}^{d} (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}} \\ &= (\sum_{t=1}^{T} \widehat{g}_{ti}^2)^{\frac{1}{2}}, \end{aligned} \tag{4.21}$$

and finally we have $\mathbf{D}_T = \mathrm{Diag}\big(\big(\sum_{t=1}^{T} \hat{\mathbf{g}}_t \odot \hat{\mathbf{g}}_t \big)^{\odot \frac{1}{2}} \big)$. □

Recall that for AdaGrad [17], the diagonal matrix of adaptive stepsize is given by $\mathbf{H}_T = \mathrm{Diag}\big(\big(\sum_{t=1}^{T} \mathbf{g}_t \odot \mathbf{g}_t\big)^{\odot\frac{1}{2}}\big)$. Thus, as a comparison, it can be seen that if we fix the orthogonal matrix $\mathbf{U}$, the optimal $\mathbf{D}_T$ is just the formula of AdaGrad in a new gradient space, which can be viewed as the original gradient space with the rotation transformation $\mathbf{U}$. It indicates that we can choose a proper orthogonal matrix $\mathbf{U}$, and implement the adaptive learning rate methods on the gradient space after the rotation so that their advantages and hyper-parameters can be inherited. Moreover, there are more advanced adaptive learning rate methods are proposed after AdaGrad, such as Adam [36] RAdam [45], Adabelief [101], *etc.*, which always achieve more satisfactory performance with some useful generalization and implementations. Therefore, we can also adopt such adaptive learning rate optimizers here on the gradient space after the rotation.

Compared with AdamW_BK that implemented with gradient norm recovery technique, this rotation with the help of the orthogonal matrix $\mathbf{U}$ will not leave an undesired impact on the length or stepsize of the descent direction, thus it naturally inherits the advantages as a generalized adaptive stepsize method. Moreover, the lower regret bound can also be guaranteed without necessary heuristic steps like AdamW_BK.

4.3.3 Solving the Orthogonal Matrix

With Section 4.3.2 solving the first sub-problem, the other sub-problem left is to choose a proper orthogonal matrix $\mathbf{U}$. Definitely, through optimizing Eq (4.4), we can obtain $\mathbf{S}_T$ and its singular vector matrix $\mathbf{U}$ would be a good choice. To achieve this, we first need to introduce a proper constraint Ψ for $\mathbf{H}_t$. As we mentioned before, a good optimizer should utilize the training information (e.g., data statistic, parameter structure and network) as much as possible while keeping the cost reasonable. Motivated by this, we consider the constraint Ψ for the matrix $\mathbf{H}_t$ that can fully

utilize the structure of the parameters in DNNs. Here, learned by AdaBK [92], we adopt the layer-wise block-diagonal constraint and Kronecker-factorized constraint for the optimization of DNNs. For completeness, we describe the entire process as follows.

The layer-wise block-diagonal constraint assumes that the matrix $\mathbf{S}_T$ has a block diagonal structure and each sub-block matrix is for one layer of a DNN. Thus, with the layer-wise block-diagonal constraint, the theoretical analysis of regret bound can be derived within one layer of DNNs. For a fully-connected layer with weight $\mathbf{W} \in R^{C_{out} \times C_{in}}$ and $\mathbf{w} = \text{vec}(\mathbf{W})$, the corresponding gradient $\mathbf{G} \in R^{C_{out} \times C_{in}}$ and $\mathbf{g} = \text{vec}(\mathbf{G})$, the kronecker-factorized constraint assumes that $\mathbf{S} = \mathbf{S}_1 \otimes \mathbf{S}_2$, where $\mathbf{S}_1 \in R^{C_{out} \times C_{out}}$, $\mathbf{S}_1 \in R^{C_{in} \times C_{in}}$ and $\mathbf{S} \in R^{C_{in}C_{out} \times C_{in}C_{out}}$, which can significantly reduce the free dimension of $\mathbf{S}$. Meanwhile, if $\mathbf{S}_1 \succeq \mathbf{0}$ and $\mathbf{S}_2 \succeq \mathbf{0}$, then $\mathbf{S} \succeq \mathbf{0}$. And because $\mathbf{S} = \mathbf{S}_1 \otimes \mathbf{S}_2 = (\mathbf{U}_1\mathbf{D}_1\mathbf{U}_1^\top) \otimes (\mathbf{U}_2\mathbf{D}_2\mathbf{U}_2^\top) = (\mathbf{U}_1 \otimes \mathbf{U}_2)(\mathbf{D}_1 \otimes \mathbf{D}_2)(\mathbf{U}_1 \otimes \mathbf{U}_2)^\top$, the singular vector matrix of $\mathbf{S}$ is $\mathbf{U} = \mathbf{U}_1 \otimes \mathbf{U}_2$. Hence, to obtain $\mathbf{U}$, we need to optimize the following guide function w.r.t. $\mathbf{S}_1$ and $\mathbf{S}_2$

$$\min_{\mathbf{S}_1 \succeq \mathbf{0}, \mathbf{S}_2 \succeq \mathbf{0}, \text{Tr}(\mathbf{S}) \leqslant 1, \text{Tr}(\mathbf{S}) \leqslant 1} F_T(\mathbf{S}_1 \otimes \mathbf{S}_2). \tag{4.22}$$

Since it is hard to solve the closed-form solution of Eq. (4.22) directly, we alternatively minimize the upper bound of Eq. (4.22), which is easier to be solved. Thus, with the help of Lemma 4.1 (the relaxation of the upper bound) and Lemma 4.2 (the closed form solution), we get the desired solutions

$$\mathbf{S}_{1,T} = \boldsymbol{L}_T^{\frac{1}{2}} / \text{Tr}(\boldsymbol{L}_T^{\frac{1}{2}}), \mathbf{S}_{2,T} = \boldsymbol{R}_T^{\frac{1}{2}} / \text{Tr}(\boldsymbol{R}_T^{\frac{1}{2}}). \tag{4.23}$$

Furthermore, the singular vector matrices of $\mathbf{S}_{1,T}$ and $\mathbf{S}_{2,T}$ are the same as $\mathbf{L}_T$ and $\mathbf{R}_T$. Hence, it is unnecessary to solve $\mathbf{S}_{1,T}$ and $\mathbf{S}_{2,T}$, and we can directly use the SVD decomposition for $\mathbf{L}_T$ and $\mathbf{R}_T$ to get the singular vector matrices. Finally, the desired matrix $\mathbf{U}$ can be obtained by

$$\mathbf{U} = \mathbf{U}_1 \otimes \mathbf{U}_2, \mathbf{U}_1\mathbf{D}_L\mathbf{U}_1^\top = \mathbf{L}_T, \mathbf{U}_2\mathbf{D}_R\mathbf{U}_2^\top = \mathbf{R}_T. \tag{4.24}$$

To sum up, the layer-wise block-diagonal constraint utilizes the structure of parameters and DNN, while together with the Kronecker-factorized constraint, they keep the computational and storage cost of solving the matrix $\mathbf{U}$ reasonable. Therefore, we keep these constraints in our algorithm, which have also been researched and adopted in [26, 24, 92], and solve the matrix $\mathbf{U}$

Overall, after we obtain the rotation transformation $\mathbf{U}$, the final preconditioned gradient $\mathbf{U}\mathbf{D}\mathbf{U}^\top\mathbf{g}$ can be divided into three steps to compute:

1. Compute the gradient after the rotation $\mathbf{U}$ by $\widehat{\mathbf{g}} = \mathbf{U}^\top\mathbf{g} = (\mathbf{U}_1^\top \otimes \mathbf{U}_2^\top)\mathbf{g} = \mathrm{vec}(\mathbf{U}_1^\top\mathbf{G}\mathbf{U}_2)$;

2. Use the adaptive learning rate method to compute $\tilde{\mathbf{g}} = \mathbf{D}\widehat{\mathbf{g}}$;

3. Rotate back the gradient $\bar{\mathbf{g}} = \mathbf{U}\tilde{\mathbf{g}} = (\mathbf{U}_1 \otimes \mathbf{U}_2)\tilde{\mathbf{g}} = \mathrm{vec}(\mathbf{U}_1\tilde{\mathbf{G}}\mathbf{U}_2^\top)$, where $\tilde{\mathbf{g}} = \mathrm{vec}(\tilde{\mathbf{G}})$.

These three steps can be implemented very efficiently, which will be described in Section 4.4 in detail.

4.4 Detailed Implementation

Infrequent Updating. In order to obtain $\mathbf{U}_1$ and $\mathbf{U}_2$, we need to store two additional statistics $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ and compute them by SVD decomposition. However, it is well known that SVD decomposition has very high complexity and is super time-consuming in dealing with large dimensional matrices. Therefore, computing them in each iteration is unendurable since it will cost tremendous computation and make the proposed optimizer inefficient. Luckily, in the procedure of training DNNs, it is no need to compute the decomposition in each iteration. Here, as the statistics tend to be stable, we adopt an infrequent updating strategy to improve computation efficiency. To be specific, we introduce two hyper-parameters T_s and T_u to control the

frequency of updating $\boldsymbol{L}_t, \boldsymbol{R}_t$ and compute the corresponding $\mathbf{U}_1, \mathbf{U}_2$, respectively. Considering the fact that SVD decomposition costs more in computational compared with updating statistics, we need to set a relatively large T_u (*e.g.*, 1000). Moreover, when using multiple GPUs, we employ a cross-GPU synchronization method to calculate more reliable feature statistics $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$. It is similar to Synchronized BN (SyncBN) [59, 11] which computes the statistics on each GPU and then synchronized across all GPUs. With this synchronization operation, the computation of feature statistics is more reliable when the batch size within a single GPU is small. To summarize, the infrequent updating strategy can improve the efficiency of the proposed optimizer in practice while keeping satisfactory performance.

Stability of SVD Decomposition. In our optimizer, $\mathbf{U}_1$ and $\mathbf{U}_2$ are usually obtained by SVD decomposition. Nevertheless, the existing deep learning frameworks (*e.g.*, PyTorch) do not implement SVD on GPU well. We find that there are many cases where SVD decomposition does not converge, which would make our proposed method fail to optimize DNNs in some cases. In order to improve the stability of SVD decomposition, we proposed two tricks. First, instead of using SVD decomposition for $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$, we add a dampening term into them $\epsilon\boldsymbol{I}$. Theoretically, such a dampening term does not change the singular vector matrices of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$. Because the condition number of the matrix can be reduced by this dampening term, it can significantly improve the stability of SVD decomposition. Moreover, $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ accumulate the statistics of output feature gradient Δ_t and input feature $\mathbf{X}_t$, but such updating formula will make their norm increase too large with large iterations. As a consequence, there has the risk of data overflow. To address this, we propose

the following updating formula

$$\begin{cases} \boldsymbol{L}_t = (1-\frac{1}{t})\boldsymbol{L}_{t-1} + \frac{1}{t}\Delta_t\Delta_t^\top; \\ \boldsymbol{R}_t = (1-\frac{1}{t})\boldsymbol{R}_{t-1} + \frac{1}{t}\mathbf{X}_t\mathbf{X}_t^\top. \end{cases} \tag{4.25}$$

It can be proved that $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ derived from this updating formula multiplying t are equal to their original definition in Lemma 4.1, and can avoid the amplitude of $\boldsymbol{L}_t$ and $\boldsymbol{R}_t$ increase by t. Meanwhile, it has no undesired impact on the solution of $\mathbf{U}_1$ and $\mathbf{U}_2$. Thus, with these two implementations, the SVD decomposition becomes more stable and we have never encountered the failure cases caused by SVD in our experiments.

Convolutional Layer. We have given the updating formula of the FC layer before. For the Conv layer, the derivation process is similar to the FC layer. We need to unfold the convolution operation to matrix multiplication first. The convolution operation can be formulated as matrix multiplication with the *im2col* operation [89, 96], and then the Conv layer can be viewed as an FC layer with $\mathfrak{A} = \mathcal{U}_1(W)\mathfrak{X}$, where $\mathfrak{A}$ and $\mathfrak{X}$ are the output and input features after *im2col* operation and $\mathcal{U}_1(\cdot)$ is the mode 1 unfold operation of a tensor. *e.g.*, for a convolution weight $\mathbf{W} \in \mathbb{R}^{C_{out}\times C_{in}\times k_1\times k_2}$, $\mathcal{U}_1(\mathbf{W}) \in \mathbb{R}^{C_{out}\times C_{in}k_1k_2}$. Then we can compute the statistics with input feature $\mathfrak{X}$ and the gradient of the output feature. $\mathcal{U}_1(\mathbf{W})$ can be considered as the weight of the FC layer, so the following computation is the same as the FC layer mentioned before.

Overall Algorithm of AdamR. The overall algorithm of AdamR is summarized in **Algorithm 1**. For a FC layer, its complexity is $T(O(\frac{C_{in}^3+C_{out}^3}{T_{ir}})+O(\frac{(C_{in}^2+C_{out}^2)N}{T_s})+ O(C_{in}C_{out}(C_{in}+C_{out})))$, and for a Conv layer, its complexity is $T(O(\frac{C_{in}^3k_1^3k_2^3+C_{out}^3}{T_{ir}}) + O(\frac{(C_{in}^2k_1^2k_2^2+C_{out}^2)N}{T_s}) + O(C_{in}k_1k_2C_{out}(C_{in}k_1k_2+C_{out})))$, where T is the total number of iterations. In our implementation, T_s and T_{ir} are set as relatively large numbers,

Algorithm 6 Adam with a rotation transformation **(AdamR)**

Input: $\mathbf{W}_0$, $\mathbf{L}_0 = \mathbf{0}_{C_{out}}$, $\mathbf{R}_0 = \mathbf{0}_{C_{in}}$, η, β_1, β_2, ϵ, ϵ'
Output: $\mathbf{W}_T$

for *t=1:T* **do**
  Receive $\mathbf{X}_t = [\boldsymbol{x}_{ti}]_{i=1}^n$ by forward propagation Receive $\Delta_t = [\boldsymbol{\delta}_{ti}]_{i=1}^n$ by backward propagation Compute gradient of weight $\mathbf{G}_t$ **if** $t\%T_s = 0$ **then**
    $\boldsymbol{L}_t = (1 - \frac{T_s}{t})\boldsymbol{L}_{t-1} + \frac{T_s}{t}\Delta_t\Delta_t^\top$,
    $\boldsymbol{R}_t = (1 - \frac{T_s}{t})\boldsymbol{R}_{t-1} + \frac{T_s}{t}\mathbf{X}_t\mathbf{X}_t^\top$;
  else
    $\boldsymbol{L}_t = \boldsymbol{L}_{t-1}$,
    $\boldsymbol{R}_t = \boldsymbol{R}_{t-1}$
  end
  if $t\%T_u = 0$ **then**
    $\mathbf{U}_{1,t}\mathbf{D}_L\mathbf{U}_{1,t}^\top = \mathbf{L}_t + \epsilon\boldsymbol{I}_{C_{out}}$,
    $\mathbf{U}_{2,t}\mathbf{D}_R\mathbf{U}_{2,t}^\top = \mathbf{R}_t + \epsilon\boldsymbol{I}_{C_{in}}$
  else
    $\mathbf{U}_{1,t} = \mathbf{U}_{1,t-1}$,
    $\mathbf{U}_{2,t} = \mathbf{U}_{2,t-1}$;
  end
  $\tilde{\mathbf{G}}_t = \mathbf{U}_1^\top\mathbf{G}_t\mathbf{U}_2$ $\mathbf{M}_t = \beta_1\mathbf{M}_{t-1} + (1-\beta_1)\tilde{\mathbf{G}}_t$;
  $\mathbf{V}_t = \beta_2\mathbf{V}_{t-1} + (1-\beta_2)\tilde{\mathbf{G}}_t \odot \tilde{\mathbf{G}}_t$ $\widehat{\mathbf{M}}_t = \frac{\mathbf{M}_t}{1-\beta_1^t}$, $\widehat{\mathbf{V}}_t = \frac{\mathbf{V}_t}{1-\beta_2^t}$ $\tilde{\mathbf{G}}_t = \frac{\widehat{\mathbf{M}}_t}{\sqrt{\widehat{\mathbf{V}}_t}+\epsilon'}$;
  $\bar{\mathbf{G}}_t = \mathbf{U}_1\tilde{\mathbf{G}}_t\mathbf{U}_2^\top$;
  $\mathbf{W}_{t+1} = \mathbf{W}_t - \eta\bar{\mathbf{G}}_t$;
end

i.e., 100 and 1000, respectively, so that the algorithm can be efficient in training DNNs. Meanwhile, similar to AdamW [47], we adopt the weight decouple method as our weight decay method in AdamR. Therefore, AdamR is based on the adaptive learning rate method AdamW, while compared with AdamW_BK that embedded AdaBK[92], we do not introduce undesirable length impact while naturally guaranteeing the lower regret bound theoretically. Furthermore, AdamR can also be combined with some parameter-free methods, e.g., [14], in the same way of Adam with D-adaption, and we omit the details here.

Table 4.1: Testing accuracies (%) on CIFAR100/CIFAR10. The best results are highlighted in bold fonts, and the numbers in red color indicate the improvement of AdamR over AdamW.

Model	SGDM	AdaGrad	AdamW	RAdam	Ranger	Adabelief	AdamW_BK	AdamR
CIFAR100								
R18	77.20±.30	71.55±.25	77.23±.10	77.05±.15	76.75±.11	77.43±.36	78.66± .34	**78.77**±.21(↑1.54)
R50	77.78±.43	72.20±.15	78.10±.17	78.20±.15	78.13±.12	79.08±.23	80.15± .19	**80.73**±.08(↑2.63)
V11	70.80±.29	67.70±.18	71.20±.29	71.08±.24	70.58±.14	72.43±.16	73.09± .29	**73.35**±.15(↑2.15)
V19	70.94±.32	63.30±.58	70.26±.23	73.01±.20	73.02±.04	72.39±.27	74.27± .25	**74.45**±.20(↑4.19)
D121	79.53±.19	71.27±.79	78.05±.26	78.65±.05	78.28±.08	79.88±.08	79.93± .23	**80.35**±.05(↑2.30)
CIFAR10								
R18	95.10±.07	92.83±.12	94.80±.10	94.70±.18	94.75±.18	95.12±.14	**95.22**± .13	95.12±.12(↑0.32)
R50	94.75±.30	92.55±.39	94.72±.10	94.72±.10	95.27±.12	95.35±.05	95.40± .07	**95.50**±.07(↑0.78)
V11	92.17±.19	90.25±.25	92.02±.08	92.00±.18	92.10±.07	92.45±.18	92.96± .07	**93.00**±.20(↑0.98)
V19	93.61±.06	91.28±.14	93.40±.04	93.57±.11	93.77±.12	93.58±.12	**93.94**± .10	93.85±.11(↑0.45)
D121	95.37±.17	92.95±.23	94.80±.07	95.02±.08	95.45±.11	95.37±.04	95.40± .04	**95.45**±.08(↑0.65)

4.5 Experiments

4.5.1 Experiment Setup

We testify the proposed AdamR on various vision tasks, such as image classification (on CIFAR100/CIFAR10 [38] and ImageNet [68]), object detection and segmentation (on COCO [44]). We compare AdamR with the representative and state-of-the-art DNN optimizers including SGDM, AdaGrad [17], AdamW [47], RAdam [45], Ranger [45, 98, 91] Adabelief [101] and AdamW_BK [92]. We tune their learning rates and weight decays and report their best results. Other hyper-parameters keep the same as their default settings. For AdamR, we set $T_s = 100$ and $T_u = 1000$, $\epsilon = 0.01$, and to make a fair comparison, the other hyper-parameters are set to be the same value as AdamW, *e.g.* learning rate and weight decay. All experiments are conducted under the Pytorch 1.8.0 framework.

4.5.2 Image Classification

In the image classification task, we compare AdamR with the existing representative DNN optimizers, including SGDM, AdamW [47], Adagrad [17], RAdam [45], Ranger [45, 98, 91], Adabelief [101], and AdamW_BK [92]. Except for SGDM which usually acts as the default optimizer in classification with much better performance than the original Adam, other optimizers are all the adaptive learning rate methods. We tune the learning rate and weight decay for each optimizer with grid search and the detailed settings for different optimizers can be found in Table 4.2 and Table 4.3. This experiment is conducted on NVIDIA GeForce RTX 2080Ti GPU.

Results on CIFAR100/10: We first evaluate the effectiveness of AdamR on CIFAR100/CIFAR10 [38], which include 50K training images and 10K testing images from 100 categories and 10 categories, respectively. Various DNN models are employed, including ResNet18 (R18), ResNet50 (R50) [29], VGG11 (V11), VGG19

Table 4.2: Settings of learning rate (LR), weight decay (WD) and WD methods for different optimizers on CIFAR10/100. Here, the WD methods include L_2 regularization weight decay (L_2 in short) and weight decouple (decouple in short).

Optimizer	SGDM	AdaGrad	AdamW	RAdam	Ranger	Adabelief	AdamW_BK	AdamR
LR	0.1	0.01	0.001	0.001	0.001	0.001	0.001	0.001
WD	0.0005	0.0005	0.5	0.5	0.5	0.5	0.5	0.5
WD method	L_2	L_2	decouple	decouple	decouple	decouple	decouple	decouple

Table 4.3: Settings of learning rate (LR), weight decay (WD) and WD methods (L_2 and decouple) for different optimizers on ImageNet.

Optimizer		SGDM	Adagrad	AdamW	RAdam	Ranger	Adabelief	AdamW_BK	AdamR
ResNet18	LR	0.1	0.01	0.001	0.001	0.001	0.001	0.001	0.001
	WD	0.0001	0.0001	0.1	0.1	0.01	0.05	0.1	0.1
ResNet50	LR	0.1	0.01	0.001	0.001	0.001	0.001	0.0005	0.001
	WD	0.0001	0.0001	0.1	0.05	0.1	0.1	0.3	0.1
WD method		L_2	L_2	decouple	decouple	decouple	decouple	decouple	decouple

(V19) [76] and DenseNet-121 (D121) [31] [1]. All the DNN models are trained for 200 epochs with batch size 128 on one single GPU. The learning rate is multiplied by 0.1 for every 60 epochs. We repeated each experiment for 4 times and reported the final result in a "mean±std" format in Table 4.1. It can be seen that AdamR achieves significant performance gains over AdamW, which are 1.54% $\sim$ 4.19% and 0.32% $\sim$ 0.98% on CIFAR100 and CIFAR10, respectively. AdamR also significantly outperforms other compared optimizers on CIFAR100 and on most DNNs of CIFAR10, with only slightly 0.1% lower compared with AdamW_BK on R18 and V19. Figure 4.1 and Figure 4.2 show the training loss to time curves of AdamW and AdamR on CIFAR100 with ResNet18 and ResNet50 when the learning rates are deducted nearly the same training time. One can see from Figure 4.2 that for R50, AdamR can accelerate the training of AdamW nearly all the time, while in Figure 4.1 for R18, AdamR can vastly speed up the beginning of the training. For a more straightforward comparison, Figure 4.3 shows the training loss curves of the epoch and time of 60 epochs with the initial learning rate kept, in which AdamR largely

[1]The model can be downloaded in `https://github.com/weiaicunzai/pytorch-cifar100`.

Table 4.4: Top 1 accuracy (%) on the validation set of ImageNet. The numbers in red color indicate the improvement of AdamR over AdamW.

Model	SGDM	AdaGrad	AdamW	RAdam	Ranger	Adabelief	AdamW_BK	AdamR
R18	70.47	62.22	70.01	69.92	69.35	70.08	**71.63**	71.29(↑1.28)
R50	76.31	69.38	76.02	76.12	75.95	76.22	76.63	**76.68**(↑0.66)

speeds up the training. For generalization performance, AdamR is always better than AdamW, with a noticeable improvement on R18 and R50. That indicates the proposed rotation transformation can gain generalization performance and speed up the initial training process.

Results on ImageNet: We also apply AdamR on the large-scale dataset ImageNet[68] that contains 1000 categories with 1.28 million images for training and 50K images for validation. We use ResNet18 and ResNet50 as the backbone models with training batch size 256 on 4 GPUs, and follow the training settings of the work [10]. Table 4.4 reports the top 1 accuracies on the validation set. One can see that AdamR outperforms all the other optimizers on R50 while performing the second best on R18 (0.34% lower than AdamW_BK), and it achieves 1.28% and 0.66% gains over AdamW for R18 and R50. Meanwhile, Figure 4.4 plots the training and validation accuracy curves, from which we see that the proposed AdamR can largely accelerate the training of DNNs over AdamW.

4.5.3 Detection and Segmentation

To show that AdamR can work well on more tasks beyond classification tasks, we evaluate it on COCO [44] for detection and segmentation tasks. The models are pre-trained on ImageNet1k and fine-tuned on COCO *train*2017 (118K images), and then evaluated on COCO *val*2017 (40K images). We adopt the latest version of MMDetection toolbox [11] as our detection framework. In our experiments, we test AdamR by Faster-RCNN [63] and Mask-RCNN [28] with various backbones, including ResNet50 (R50) and ResNet101 (R101), Swin transformer [46] (*e.g.*, Swin-

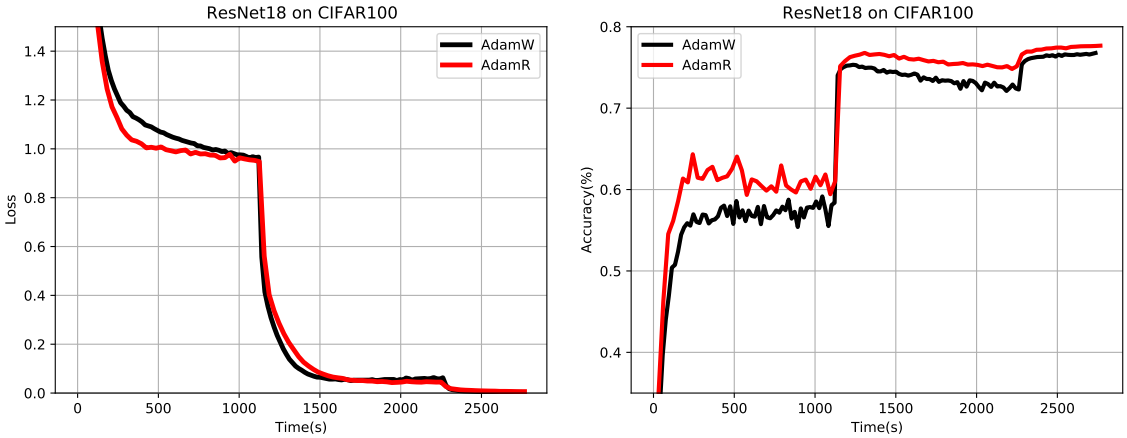


Figure 4.1: Training loss (left) and testing accuracy (right) curves to time of AdamW and AdamR on CIFAR100 with ResNet18 when the learning rates are deducted nearly simultaneously.

T and Swin-S). For R50 and R101 backbone, learning rate and weight decay are set to 0.02 and 0.0001 for SGDM, and 0.0001 and 0.2 for AdamW and AdamR, respectively. For the Swin transformer backbone, the learning rate and weight decay are set to 0.0001 and 0.02 for AdamW and AdamR. The learning rate schedule is 1X for Faster-RCNN. Other configurations follow the settings of the official MMDetection toolbox[2]. For the default optimizers, we cite their official results directly[3]. The experiments in this section are conducted on NVIDIA GeForce RTX 3090 Ti GPUs.

We list the Average Precision (AP) of object detection by Faster-RCNN in Table 4.5. We can see that the models trained by AdamR achieve clear performance gains of $1.3\% \sim 2.4\%$. Fig. 4.5 shows the training loss curves of Faster-RCNN with ResNet50 and ResNet101 backbone. AdamR obviously accelerates the training process over AdamW. Table 4.6 shows the AP^b of detection and AP^m of segmentation by Mask-RCNN. We can see that AdamR gains $1.4\% \sim 1.8\%$ AP^b and 1.1% AP^m for R50

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

[3]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`.

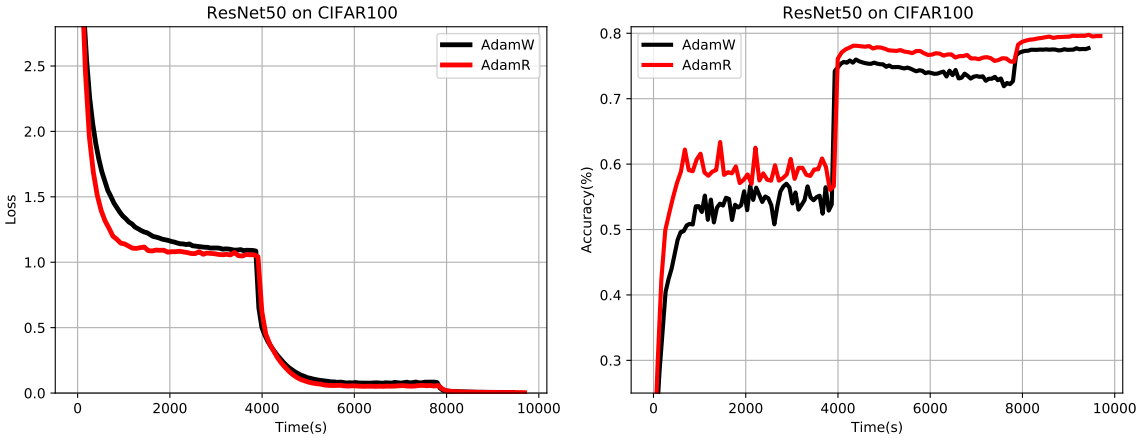


Figure 4.2: Training loss (left) and testing accuracy (right) curves to time of AdamW and AdamR on CIFAR100 with ResNet50 when the learning rates are deducted nearly simultaneously.

Table 4.5: Detection results of Faster-RCNN on COCO. Δ means the gain of AdamR 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
	AdamW	37.8	58.7	41.0	22.1	41.2	49.2
	AdamR	39.1	60.1	42.4	22.2	42.8	51.6
	Δ	↑1.3	↑1.4	↑1.4	↑0.1	↑1.6	↑2.4
R101	SGDM*	39.4	60.1	43.1	22.4	43.7	51.1
	AdamW	40.1	60.6	43.8	22.9	44.1	52.8
	AdamR	41.5	61.9	45.3	24.0	45.1	55.6
	Δ	↑1.4	↑1.3	↑1.5	↑1.1	↑1.0	↑2.8

and R101 backbones over AdamW. For the Swin transformer backbone, AdamR also achieves $0.6\% \sim 0.9\%$ AP^b and $0.4\% \sim 0.7\%$ AP^m gains over AdamW. Fig. 4.6 plots the training loss curves of Faster-RCNN with ResNet50 and ResNet101 backbone. AdamR speeds up the training process clearly. The results on COCO demonstrate that the proposed AdamR can be easily adopted into the downstream tasks without additional hyperparameter tuning over AdamW.

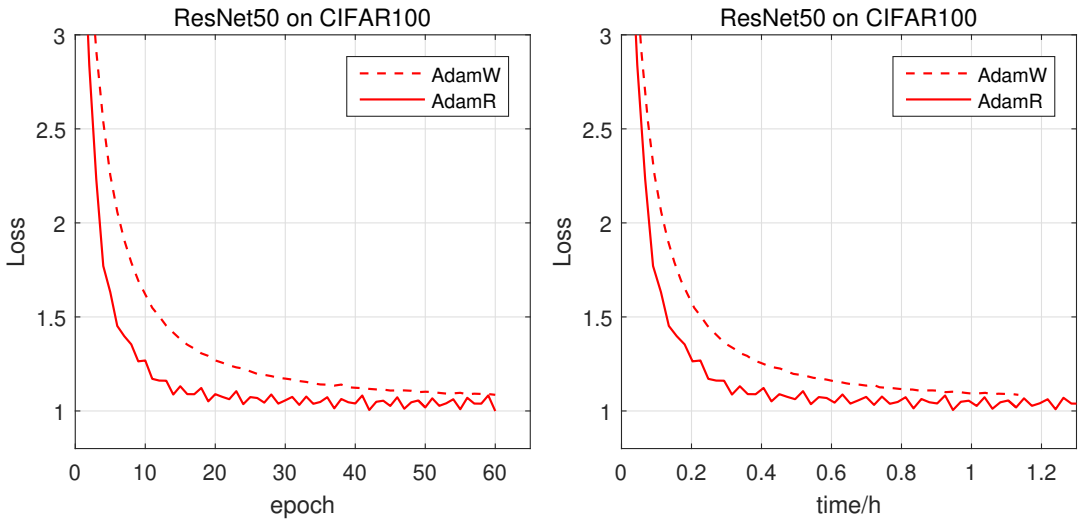


Figure 4.3: Training loss curves to epoch (left) and time (right) of 60 epochs training of AdamW and AdamR on CIFAR100 for ResNet50 with the initial learning rates kept.

4.5.4 Ablation Study

As we mentioned before, AdamR is well-designed to inherit all the well-tuned hyper-parameters in AdamW naturally. Due to this considerable advantage, combined with the implementations added, we have only one pair of parameters to do ablations, i.e., the infrequent updating intervals (T_s, T_u). Meanwhile, we leave a remark on the $\epsilon\mathbf{I}_{C_{out}}$ and $\epsilon\mathbf{I}_{C_{in}}$ of implemented stability SVD decomposition for clarity.

Infrequent Updating. In AdamR, we apply the infrequent updating intervals T_s and T_u to balance the computational cost and performance. Here we compare and report the testing accuracy and the experiment time of AdamR with different updating intervals for training ResNet 50 on CIFAR100. Apart from parameters T_s and T_u , the other settings keep the same as the experiments in Section4.5.2. Each experiment is repeated four times to eliminate randomness, and the results are reported in a "mean $\pm$ std" format. Ideally, a smaller interval (e.g., $T_s = 20$) will keep more accurate statistics. However, we can find from Table 4.7 that the larger interval

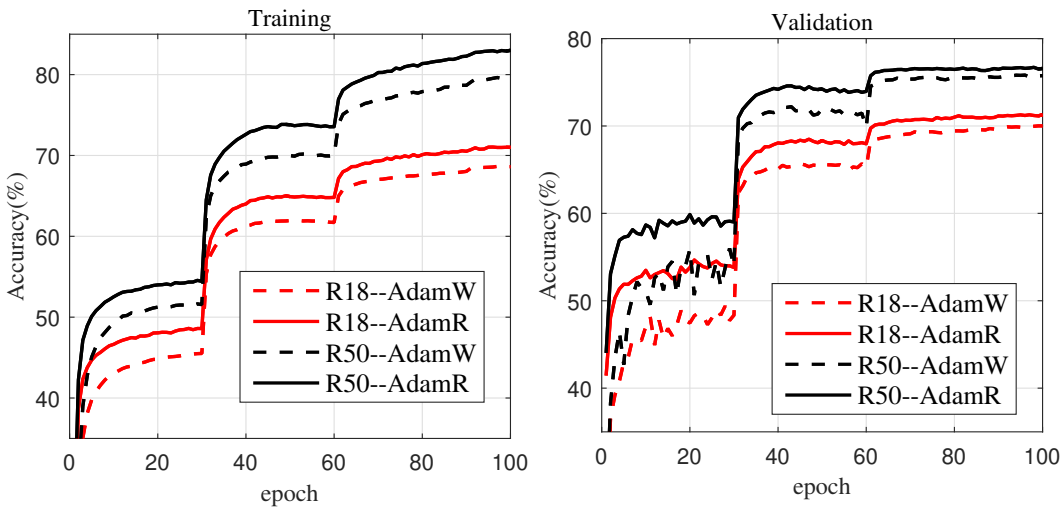


Figure 4.4: Training and validation accuracy curves of AdamW and AdamR on ImageNet with ResNet18 and ResNet50 backbones.

(i.e., $T_s = 50, 100, 200$) also maintains good performance with less computation time, while the larger $T_s = 500$ will cause an obvious performance drop. As a balance, we choose $T_s = 100$ and $T_u = 1000$, which perform well throughout the rest experiments.

Stability of SVD Decomposition. As a remark, although in AdamR, we add the term $\epsilon \mathbf{I}_{C_{out}}$ and $\epsilon \mathbf{I}_{C_{in}}$ on the matrices $\mathbf{L}_t$ and $\mathbf{R}_t$, respectively, they only change the eigenvalue of the decomposition while leaving no influence on the orthogonal matrix (formed by eigenvectors) we computed. Consequently, as small terms that are added just in case there occurs numerically unstable inbuilt functions, it will not change our results theoretically. And experimentally, we find the parameter ϵ has almost no impact on the final results.

4.6 Conclusion

To sum up, in this work, we propose a new adaptive learning rate optimization algorithm, named AdamR. Compared with traditional adaptive learning rate methods, it introduces an extra rotation transformation into the gradient. We can show

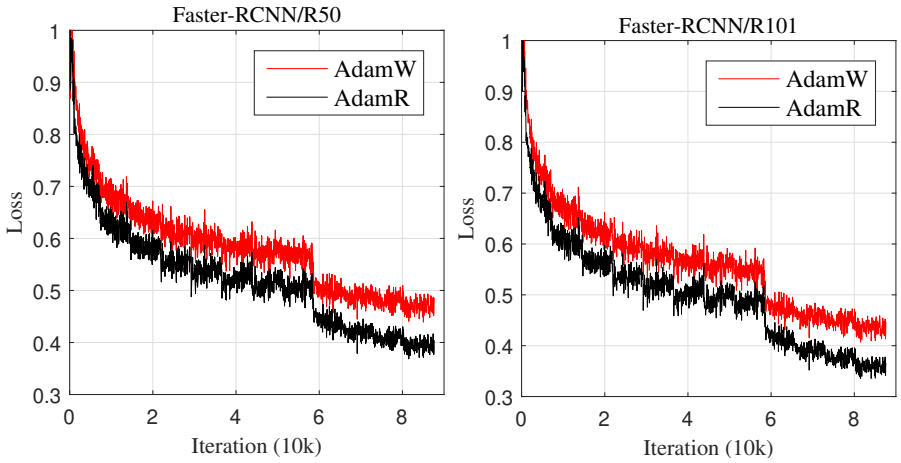


Figure 4.5: Training loss curves of ResNet50.

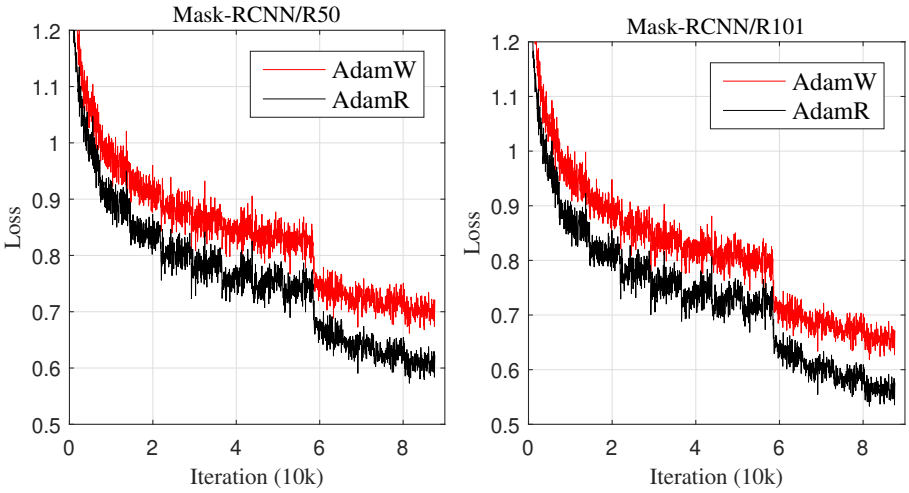


Figure 4.6: Training loss curves of Mask-RCNN.

that the rotation transformation embeds the information of the full-matrix preconditioned gradient, which usually has a lower regret bound than the adaptive learning rate methods that consider the diagonal elements of the preconditioned matrix, and it will not leave an undesired impact on the length (stepsize) of the descent direction. In order to find proper rotation transformation, we adopt the layer-wise block-diagonal and Kronecker-factorized constraints for training DNNs under the theorem of lower regret bound. As a result, we naturally derive a formula to compute the rotation

Table 4.6: Detection and segmentation results of Mask-RCNN on COCO. Δ means the gain of AdamR 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
		AdamW	37.8	58.7	41.0	35.4	56.2	38.0
		AdamR	39.6	60.4	43.4	36.5	57.6	38.8
		Δ	↑1.8	↑1.7	↑2.4	↑1.1	↑1.4	↑0.8
R101	1X	SGDM*	40.0	60.5	44.0	36.1	57.5	38.6
		AdamW	40.7	61.1	44.6	37.2	58.4	40.1
		AdamR	42.1	62.5	46.0	38.3	59.8	41.1
		Δ	↑1.4	↑1.4	↑1.4	↑1.1	↑1.45	↑1.0
Swin-T	1X	AdamW*	42.7	65.2	46.8	39.3	62.2	42.2
		AdamR	43.3	66.0	47.4	40.0	62.9	42.8
		Δ	↑0.6	↑0.8	↑0.6	↑0.7	↑0.7	↑0.6
Swin-T	3X	AdamW*	46.0	68.2	50.3	41.6	65.3	44.7
		AdamR	46.9	68.9	51.6	42.3	65.8	45.4
		Δ	↑0.9	↑0.7	↑1.3	↑0.7	↑0.5	↑0.7
Swin-S	3X	AdamW*	48.2	69.8	52.8	43.2	67.0	46.1
		AdamR	49.0	70.5	54.1	43.6	67.6	47.0
		Δ	↑0.8	↑0.7	↑1.3	↑0.4	↑0.6	↑0.9

Table 4.7: Accuracy (%) and time (s) of AdamR with different updating intervals in training ResNet50 on CIFAR100.

T_s	20	50	100	200	500
T_u	200	500	1000	2000	5000
Acc.	$80.04 \pm .21$	$80.23 \pm .19$	$\mathbf{80.28} \pm .26$	$80.24 \pm .27$	$79.69 \pm .25$
Time	20470.0 ± 105	17948.5 ± 93	17084.3 ± 63	16635.8 ± 83	$\mathbf{16393.5} \pm 74$

transformation which only needs two additional statistics, the computational and storage cost of which can be limited within a proper range. The experimental results illustrate that AdamR significantly speeds up the beginning training stage and clearly gains the generalization performance over the based optimizer AdamW.

Chapter 5

AFopt: Attention Feature Based Optimizer for Transformers

The attention module is a critical transformer component and significantly enhances the model performance across various tasks. This chapter introduces a new optimizer for optimizing the attention module, named the Attention-Feature-based Optimizer (AFOpt). By attention feature gradient descent, AFOpt treats the attention module holistically, facilitating interaction among its parameters to improve training efficacy. Our method begins by applying gradient descent directly to output features of the attention module. Subsequently, we update the module's parameters by approximating the impact of the output feature's gradient descent. This parameter interaction, inherent to the unique structure of the attention module, may better utilize the feature information and the similarities of different patches to enhance the training process. We demonstrate the effect of AFOpt through experiments in object detection and segmentation tasks.

5.1 Introduction

The deep neural networks (DNNs) structure has been constantly developing and innovating in recent years. One of the structures that cannot be ignored is the proposal of transformer [83]. After first verified effective in the NLP field (*e.g.*,[83,

35]), transformers such as ViT [16], DETR [8] and Swin [46] were also proposed in the computer vision field. The most crucial innovation of transformers is the addition of the attention module. The attention module is a specially designed nonlinear operation. It comprises linear and softmax layers arranged sequentially, which is significantly more complex than a single linear or convolutional layer. As a result, it can extract more useful information, thereby enhancing its effectiveness. Generally, the attention module is used as a whole and outputs the corresponding feature to the rest part of the neural networks (NNs) for training, while looking into the module, it consists of a second-order mapping of the input with a nonlinear softmax function followed, and another linear mapping of input in parallel, which is different with the other NNs and deserved to be further discussed.

Focusing on the aspect of training, the default training optimizers of transformers are usually Adam and AdamW, being nearly the same as those adopted in training other DNNs. From the optimization stand of view, the current optimizers, for example, the stochastic gradient descent [66, 61] and the adaptive stepsize methods (Adam [36], RMSProp [81], AdamW [47], and Radam [45], etc.), are already effective for optimizing the parameters in attention modules with the help of back-propagation (BP) process. However, after witnessing the effectiveness of attention modules, it is a natural intention to utilize the hidden information behind the unique and clear attention structure to assist optimization.

Considering the attention module, the parameters compose with each other, creating a second-order and even more complex operator of input to shape a better output. Overall, a better output feature is the aim of the whole attention module, with different parameters working together to reflect the similarities between different positions. Introducing some interaction within the attention parameters may better reflect the similarities and lead to better performance. Meanwhile, Yong [90] suggests that gradient descent on intermediate features achieves better performance

and proposes stochastic feature gradient decent (FSGD), in which the feature output from a single layer is considered the tensor that needs to be applied gradient descent. [90] also explained that FSGD updates feature in a higher dimensional space, which is more accurate.

Inspired by [90], in this work, we consider optimizing the attention module as a whole, making the parameters within attention interact with each other to assist training, which we call Attention-Feature-based Optimizer (AFOpt). Overall, since the attention module changes the input into the output feature needed, we can apply gradient descent to the output feature of the attention module while optimizing the parameters within attention according to the feature change in each step. Therefore, the attention feature is optimized in a higher-dimensional space with less noise, leading to better performance. Further to FSGD [90], although AFOpt and FSGD both apply the gradient descent on the output feature and update the parameter by approximating the feature gradient descent with ℓ_2 norm, we have a second intention. Since attention module researches on global similarity, we interact the parameters with each other to better utilize the similarities and shape the output. Different parameters within the attention module can be connected by applying attention feature gradient descent, which may assist the optimization. The update interaction comes from the specific structure of attention, which can be reflected in the update formula. Experimentation on detection and segmentation tasks shows the effect of our proposed method.

5.2 Preliminaries

5.2.1 Attention Module and Notations

Attention module has shown its effectiveness in many areas as we state in Section 5.1. In this section, we introduce the details of attention module in Figure 5.1 and

state its notations for further use.

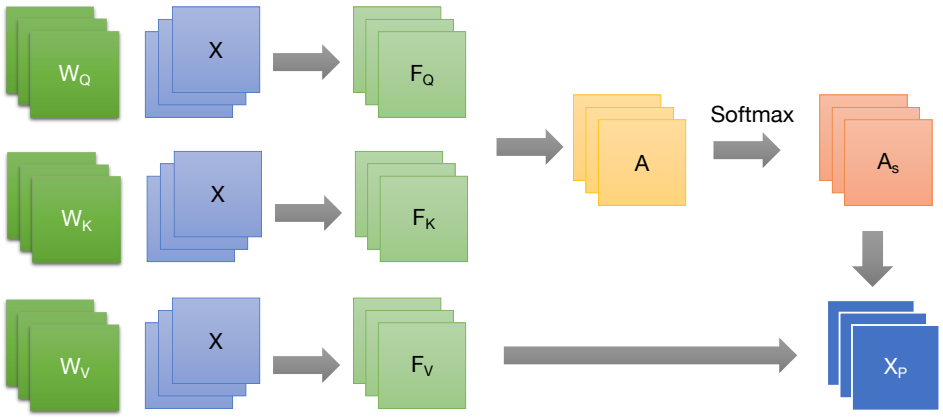


Figure 5.1: Illustration of attention module.

Here, we denote $\mathbf{X}$ the input of the module, denote $\mathbf{W_Q}$, $\mathbf{W_K}$ and $\mathbf{W_V}$ to be the linear parameters of attention module, while using $\mathbf{F_Q} := \mathbf{W_Q X}$, $\mathbf{F_K} := \mathbf{W_K X}$ and $\mathbf{F_V} := \mathbf{W_V X}$ to be the feature generated by the corresponding linear operator. Meanwhile, we denote $\mathbf{A}$ the attention matrix, which can be computed directly by $\mathbf{A} = \mathbf{F_Q}^\top \mathbf{F_K}$, denote $\mathcal{S}(\cdot)$ the softmax function in the module, and $\mathbf{A}_s := \mathcal{S}(\mathbf{A})$ the feature generated by $\mathbf{A}$ and the softmax function. The whole output of the attention module is usually the input of the following projection layer, so we denote the whole output by $\mathbf{X}_P := \mathbf{F_V A}_s$.

5.2.2 Feature Gradient Descent

We briefly introduce feature gradient descent (FSGD)[90] here because of two reasons. First, the design of FSGD inspires our work because feature gradient descent is a practical way to create interaction with different parameters in attention module, which is exactly our intention. Second, due to the fact that there are usually MLP modules in transformers, we can combine FSGD into our work to better optimize parameters in other linear layers.

The insight of FSGD is to change the update of weights from parameter space into the feature space, using the gradient of feature (i.e., the descent direction of feature) to correct the descent direction of weights. Thus, more useful information of the input data and features can be adopted. Specifically, for a linear variable $\mathbf{W}$ and input $\mathbf{X}$, denote by $\mathbf{F}^{t+1} := \mathbf{W}^t\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{F}^t}$ the desired feature after a single gradient descent step, and $\mathbf{F}(\mathbf{W}) := \mathbf{W}\mathbf{X}^t$ the output feature obtained by the network update. To minimize the distance between $\mathbf{F}^{t+1}$ and $\mathbf{F}(\mathbf{W})$, we need to solve the optimization problem

$$\mathbf{W}^{t+1} = \arg\min_{\mathbf{W}} \|\mathbf{W}^t\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{F}^t} - \mathbf{W}\mathbf{X}^t\|_2^2, \tag{5.1}$$

which leads to the closed-form solution

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial L}{\partial\mathbf{W}^t}(\mathbf{X}^t\mathbf{X}^{t\top})^{-1}. \tag{5.2}$$

The corresponding work [90] also discussed the case of convolution operation, and we omit the details here.

5.3 Methodology

5.3.1 Motivation

We consider optimizing the attention module holistically to interact the parameters with each other. We have two intentions for our design. First, inspired by [90], we optimize attention feature in the feature space, which may have a higher dimension and less noise, making the training more accurate. Second, we use this approximation to derive the interaction of the parameters, which may better utilize the similarity information of positions.

In practice, we apply gradient descent to the feature matrix generated by the attention module and generalize an optimization problem using the direct weight operation to approximate attention gradient descent. The solution is adopted to

update the weight parameters. The main difficulties lie in solving the generated optimization problem that contains the nonlinear softmax function and implementing the computation economically. Later in Section 5.3.2, we will deal with the difficulty caused by the softmax function in updating $\mathbf{W_Q}$ and $\mathbf{W_K}$, and in Section 5.3.3, we change the update computation of $\mathbf{W_V}$ to be economical and practical. In Section 5.3.4, we will implement AFOpt with proper technique.

5.3.2 The update of $\mathbf{W_Q}$ and $\mathbf{W_K}$

Here we update $\mathbf{W_Q}$ and $\mathbf{W_K}$ by alternative update. We first deal with the parameter $\mathbf{W_K}$. When we take the attention module as a whole, the output feature is $\mathbf{X}_P$, and the gradient descent based on $\mathbf{X}_P^t$ at the t-th iteration takes $\mathbf{X}_P^{t+1} = \mathbf{X}_P^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{X}_P^t}$. Meanwhile, $\mathbf{X}_P$, as we state before, is computed by $\mathbf{X}_P = \mathbf{F_V}\mathcal{S}\left((\mathbf{W_Q}\mathbf{X})^\top \mathbf{W_K}\mathbf{X}\right)$. For convenient we use the notation $\mathbf{F_Q} = \mathbf{W_Q}\mathbf{X}$ and approximate the attention feature gradient descent using the weight parameter $\mathbf{W_K}$. Thus, the attention feature based gradient descent problem for $\mathbf{W_K}$ takes

$$\mathbf{W_K}^{t+1} = \arg\min_{\mathbf{W_K}} \|\mathbf{X}_P^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{X}_P^t} - \mathbf{F_V}^t \mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{W_K} \mathbf{X}^t\right)\|_2^2. \tag{5.3}$$

Solving the above question directly will be costly and impractically, and the difficulty mainly comes from the softmax function $\mathcal{S}(\cdot)$. Therefore, here we approximate the softmax function using its Taylor expansion approximation. Notice that the first order expansion of softmax function at a given $\mathbf{x}_0$ takes

$$\mathcal{S}(\mathbf{x}_0 + \Delta\mathbf{x}) = \mathcal{S}(\mathbf{x}_0) + \nabla\mathcal{S}(\mathbf{x}_0)\Delta\mathbf{x}. \tag{5.4}$$

Then for the desired optimization problem (5.3), in the current t-th step, the approximation of softmax function takes

$$\begin{aligned} \mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{W_K} \mathbf{X}^t\right) \approx \quad & \mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{W_K}^t \mathbf{X}^t\right) \\ & + \nabla\mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{W_K}^t \mathbf{X}^t\right) \mathbf{F_Q}^{t\top} \left(\mathbf{W_K} - \mathbf{W_K}^t\right) \mathbf{X}^t. \end{aligned} \tag{5.5}$$

After simple merging, the attention feature based gradient is defined to be the solution of the following optimization problem (5.6) *i*.e.,

$$\mathbf{W_K}^{t+1} = \arg\min_{\mathbf{W_K}} \|\mathbf{F_V}^t \nabla\mathcal{S}\Big(\mathbf{F_Q}^{t\top}\mathbf{W_K}^t\mathbf{X}^t\Big)\mathbf{F_Q}^{t\top}\big(\mathbf{W_K}^t - \mathbf{W_K}\big)\,\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{X}_P^t}\|_2^2. \tag{5.6}$$

With the help of this approximation, the above problem (5.6) has closed form solution. Specifically, by first order optimal condition, we derive the solution of problem (5.6) as

$$\mathbf{W_K}^{t+1} = \mathbf{W_K}^t - (\mathbf{L_Q}^t\mathbf{L_Q}^{t\top})^{-1}\frac{\partial\mathcal{L}^t}{\partial\mathbf{W_K}^t}(\mathbf{X}^t\mathbf{X}^{t\top})^{-1}, \quad \text{where} \tag{5.7}$$

$$\mathbf{L_Q}^t = \mathbf{F_Q}^t\left(\nabla\mathcal{S}\left(\mathbf{F_Q}^{t\top}\mathbf{F_K}^t\right)\right)^\top\mathbf{F_V}^{t\top}. \tag{5.8}$$

Similarly, for $\mathbf{W_Q}$, we have

$$\mathbf{W_Q}^{t+1} = \mathbf{W_Q}^t - (\mathbf{L_K}^t\mathbf{L_K}^{t\top})^{-1}\frac{\partial\mathcal{L}^t}{\partial\mathbf{W_Q}^t}(\mathbf{X}^t\mathbf{X}^{t\top})^{-1}, \quad \text{where} \tag{5.9}$$

$$\mathbf{L_K}^t = \mathbf{F_K}^t\left(\nabla\mathcal{S}\left(\mathbf{F_Q}^{t\top}\mathbf{F_K}^t\right)\right)^\top\mathbf{F_V}^{t\top}. \tag{5.10}$$

Till now, we have finished deriving the update of $\mathbf{W_Q}$ and $\mathbf{W_K}$.

5.3.3 The update of $\mathbf{W_V}$

For matrix $\mathbf{W_V}$, due to the fact that $\mathbf{X}_P = \mathbf{F_V}\mathbf{A}_s = \mathbf{W_V}\mathbf{X}\mathbf{A}_s$, essentially the update is the same as linear layer with $\mathbf{X}^t\mathbf{A}_s^t$ substitute $\mathbf{X}^t$ in FSGD. For a clear explanation, the attention feature based gradient descent takes

$$\mathbf{W_V}^{t+1} = \arg\min_{\mathbf{W_V}} \|\mathbf{X}_P^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{X}_P^t} - \mathbf{W_V}\mathbf{X}^t\mathbf{A}_s^t\|_2^2, \tag{5.11}$$

which has the closed form solution

$$\mathbf{W_V}^{t+1} = \mathbf{W_V}^t - \eta\frac{\partial L}{\partial\mathbf{W_V}^t}(\mathbf{X}^t\mathbf{A}_s^t\mathbf{A}_s^{t\top}\mathbf{X}^{t\top})^{-1}. \tag{5.12}$$

However, in application, the update of $\mathbf{W_V}$ is not economic to implement with the above equation. This is because the attention matrix $\mathbf{A}_s$ is required in Eq.(5.12), which is usually associated with embedding and is not easy to extract. Thus, we rearrange it using the input of the projection layer follows (*i*.e., $\mathbf{X}_P$). For convenience, denote $\mathbf{W_{VV}} = \mathbf{W_V}^\top \mathbf{W_V}$ and assume $\mathbf{W_{VV}}$ has matrix inverse, then we have

$$\mathbf{X}\mathbf{A}_s\mathbf{A}_s^\top\mathbf{X}^\top = \mathbf{W_{VV}}^{-1}\mathbf{W_V}^\top\mathbf{X}_P\mathbf{X}_P^\top\mathbf{W_V}\mathbf{W_{VV}}^{-1}. \tag{5.13}$$

By applying inversion, it holds that

$$(\mathbf{X}\mathbf{A}_s\mathbf{A}_s^\top\mathbf{X}^\top)^{-1} = \mathbf{W_{VV}}(\mathbf{W_V}^\top\mathbf{X}_P\mathbf{X}_P^\top\mathbf{W_V})^{-1}\mathbf{W_{VV}}. \tag{5.14}$$

Eq. (5.14) can be compiled into the update of $\mathbf{W_V}$, i.e.,

$$\mathbf{W_V}^{t+1} = \mathbf{W_V}^t - \eta\frac{\partial L}{\partial \mathbf{W_V}^t}\mathbf{W}_{\mathbf{VV}}^t(\mathbf{W_V}^{t\top}\mathbf{X}_P^t\mathbf{X}_P^{t\top}\mathbf{W}_{\mathbf{V}}^t)^{-1}\mathbf{W}_{\mathbf{VV}}^t, \tag{5.15}$$

which is the final update formula of $\mathbf{W_V}$. Eq. (5.15) only requires the input of project layer $\mathbf{X}_P$ and the current weight parameter $\mathbf{W_V}$, which is easier to obtain.

5.3.4 Implementations

To increase the stability and performance of our proposed optimizer, in this section, we introduce some implementations to finish our design. Except for embedding FSGD, other techniques can be found in many existing optimizers (e.g., [90, 92]), and we have implemented some of them in the proposed NKFAC in Chapter 3. Here, we only briefly discuss these techniques.

Embed FSGD. Feature gradient descent is proposed in [90]. By applying gradient descent on the output feature of convolution and linear layers, FSGD is proved by some experiments to be effective in traditional CNNs. Because our method only aims to optimize the attention module, leaving linear layers optimized by default optimizer, we are motivated to implement FSGD into our AFOpt to enhance the

generalization performance further. Thus, we apply FSGD to the linear layers other than attention modules to improve performance. The details of FSGD on linear layers have been introduced in Section 5.2.2.

Gradient Norm Recovery: The gradient norm recovery is a useful technique to avoid the tedious parameter tuning process. Specifically, for a revised gradient $\hat{\mathbf{G}}$, we recover its norm by the the previous gradient norm $\|\mathbf{G}\|$, i.e.,

$$\tilde{\mathbf{G}} = \hat{\mathbf{G}}\frac{\|\mathbf{G}\|}{\|\hat{\mathbf{G}}\|}. \tag{5.16}$$

Adaptive Dampening: The numerical stability of inversion is a problem that always needs consideration in designing optimizers. Here, we add the adaptive dampening for all the statistics that need inversion. Specifically, for a statistics $\mathbf{S}$ (which, in practice, refers to $\mathbf{X}^t\mathbf{X}^{t\top}$, or $\mathbf{L_Q}^t\mathbf{L_Q}^t$, or others), we compute the maximal eigenvalue $\lambda_{\mathbf{S}}$, and add an adaptive dampening $\eta\lambda_{\mathbf{S}}\mathbf{I}$ on the statistic $\mathbf{S}$ to guarantee the numerical stability, i.e.,

$$\mathbf{S}^t = \mathbf{S}^t + \eta\lambda_{\mathbf{S}}\mathbf{I}. \tag{5.17}$$

Statistics Momentum: The statistics momentum is added to gather and summarize all the sample information. This helps in two aspects. On the one hand, gathering information helps ensure the accuracy of statistics calculation; on the other hand, it can help reverse statistics. Here for all statistics $\mathbf{S}$, we update

$$\mathbf{S}^t = \alpha\mathbf{S}^t + (1-\alpha)\mathbf{S}^{t-1}, \quad \alpha \in (0,1). \tag{5.18}$$

Infrequent Update: Our computation of statistics, especially of the inversion, will add to the training process's computational cost and running time. We adopt a periodical update strategy to reduce the computational cost and control the running time. Here, we set T_s and T_u as the update interval of statistics and inversion.

Adaptive Stepsize (Ada_AFOpt): Many tasks use AdamW as their default optimizer when training DNNs. Besides gradient descent, we can also embed the corrected gradient into AdamW. Meanwhile, noticing a recent work Adan [87], we may also embed AFOpt into this structure to enjoy the advantages of this framework, and we omit the details here.

5.3.5 Overall Algorithm

Now we are ready to state the overall AFOpt and Ada_AFOpt. For convenience, we denote the gradient matrix by $\mathbf{G}^t$, and denote

$$\mathbf{L_{KK}}^t = (\mathbf{L_Q}^t \mathbf{L_Q}^{t\top}), \quad \text{where} \quad \mathbf{L_Q}^t = \mathbf{F_Q}^t \left(\nabla \mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{F_K}^t\right)\right)^\top \mathbf{F_V}^{t\top}, \tag{5.19}$$

$$\mathbf{L_{QQ}}^t = (\mathbf{L_K}^t \mathbf{L_K}^{t\top}), \quad \text{where} \quad \mathbf{L_K}^t = \mathbf{F_K}^t \left(\nabla \mathcal{S}\left(\mathbf{F_Q}^{t\top} \mathbf{F_K}^t\right)\right)^\top \mathbf{F_V}^{t\top}, \tag{5.20}$$

$$\mathbf{R_{VV}}^t = \mathbf{W_V^t}^\top \mathbf{X}_P^t \mathbf{X}_P^{t\top} \mathbf{W_V^t}, \qquad \mathbf{R_{XX}}^t = \mathbf{X}^t \mathbf{X}^{t\top}. \tag{5.21}$$

For the inversion part, for convenience we denote

$$\mathbf{D_{KK}}^t = (\mathbf{L_{KK}}^t)^{-1}, \mathbf{D_{QQ}}^t = (\mathbf{L_{QQ}}^t)^{-1}, \mathbf{D_{VV}}^t = (\mathbf{R_{VV}}^t)^{-1}, \mathbf{D_{XX}}^t = (\mathbf{R_{XX}}^t)^{-1}. \tag{5.22}$$

Thus, the updated gradient of the attention module can be represented as

$$\hat{\mathbf{G}}_\mathbf{Q}^t = \mathbf{D_{QQ}}^t \mathbf{G_Q}^t \mathbf{D_{XX}}^t, \quad \hat{\mathbf{G}}_\mathbf{K}^t = \mathbf{D_{KK}}^t \mathbf{G_K}^t \mathbf{D_{XX}}^t, \quad \hat{\mathbf{G}}_\mathbf{V}^t = \mathbf{G_V}^t \mathbf{D_{VV}}^t. \tag{5.23}$$

and the updated gradient of FSGD for linear layers can be represented as

$$\hat{\mathbf{G}}^t = \mathbf{G}^t \mathbf{D_{XX}}^t. \tag{5.24}$$

Combining with the implementations, the sketch of AFOpt and Ada_AFOpt are described in Algorithm 7 and Algorithm 8.

Lastly, it is important to acknowledge that in practical applications, multi-head self-attention is commonly employed instead of single-head attention. In such cases, we only need to perform gradient calculation according to the head block for each $\mathbf{W_Q}$ and $\mathbf{W_K}$. We omit this part here.

Algorithm 7 AFOpt

Input: $\mathbf{W}^0$, ${\mathbf{L}_{\mathbf{QQ}}}^0 = \mathbf{0}_{C_{out}}$,${\mathbf{L}_{\mathbf{KK}}}^0 = \mathbf{0}_{C_{out}}$, ${\mathbf{R}_{\mathbf{VV}}}^0 = \mathbf{0}_{C_{in}}$,${\mathbf{R}_{\mathbf{XX}}}^0 = \mathbf{0}_{C_{in}}$, τ
Output: $\mathbf{W}_T$

for *t=1:T* **do**
  Extract $\mathbf{X}^t$, ${\mathbf{F}_{\mathbf{Q}}}^t$, ${\mathbf{F}_{\mathbf{K}}}^t$, ${\mathbf{F}_{\mathbf{V}}}^t$, $\mathbf{X}_P^t$ by forward and backward propagation. Compute the weight gradient $\mathbf{G}^t$.
  if $t\%T_s = 0$ **then**
    Compute ${\mathbf{L}_{\mathbf{KK}}}^t$, ${\mathbf{L}_{\mathbf{QQ}}}^t$, ${\mathbf{R}_{\mathbf{VV}}}^t$ and ${\mathbf{R}_{\mathbf{XX}}}^t$ by Eq.(5.19), Eq.(5.20) and Eq.(5.21); Apply statistics momentum Eq.(5.18) on ${\mathbf{L}_{\mathbf{KK}}}^t$, ${\mathbf{L}_{\mathbf{QQ}}}^t$, ${\mathbf{R}_{\mathbf{VV}}}^t$ and ${\mathbf{R}_{\mathbf{XX}}}^t$;
  else
    ${\mathbf{L}_{\mathbf{KK}}}^t = {\mathbf{L}_{\mathbf{KK}}}^{t-1}$, ${\mathbf{L}_{\mathbf{QQ}}}^t = {\mathbf{L}_{\mathbf{QQ}}}^{t-1}$, ${\mathbf{R}_{\mathbf{VV}}}^t = {\mathbf{R}_{\mathbf{VV}}}^{t-1}$ and ${\mathbf{R}_{\mathbf{XX}}}^t = {\mathbf{R}_{\mathbf{XX}}}^{t-1}$.
  end
  if $t\%T_u = 0$ **then**
    Apply adaptive dampening Eq.(5.17) onto ${\mathbf{L}_{\mathbf{KK}}}^t$, ${\mathbf{L}_{\mathbf{QQ}}}^t$, ${\mathbf{R}_{\mathbf{VV}}}^t$ and ${\mathbf{R}_{\mathbf{XX}}}^t$; Compute the inverse ${\mathbf{D}_{\mathbf{KK}}}^t$, ${\mathbf{D}_{\mathbf{QQ}}}^t$, ${\mathbf{D}_{\mathbf{VV}}}^t$ and ${\mathbf{D}_{\mathbf{XX}}}^t$ by Eq. (5.22);
  else
    ${\mathbf{D}_{\mathbf{KK}}}^t = {\mathbf{D}_{\mathbf{KK}}}^{t-1}$, ${\mathbf{D}_{\mathbf{QQ}}}^t = {\mathbf{D}_{\mathbf{QQ}}}^{t-1}$, ${\mathbf{D}_{\mathbf{VV}}}^t = {\mathbf{D}_{\mathbf{VV}}}^{t-1}$ and ${\mathbf{D}_{\mathbf{XX}}}^t = {\mathbf{D}_{\mathbf{XX}}}^{t-1}$;
  end
  if *attention module and other linear layer* **then**
    Compute gradient $\hat{\mathbf{G}}^t$ by Eq.(5.23) and Eq.(5.24), respectively;
  else
    $\hat{\mathbf{G}}^t = \mathbf{G}^t$.
  end
  Apply gradient norm recovery by Eq.(5.16) to get $\tilde{\mathbf{G}}^t$.
  $\mathbf{W}_{t+1} = \mathbf{W}_t - \tau\tilde{\mathbf{G}}^t$
end

5.4 Experiments

5.4.1 Experiment on COCO

In this section, we testify the proposed AFOpt on object detection and segmentation task, with dataset COCO [44]. The model we adopt is pre-trained on ImageNet1k. We fine-tune the pre-trained model on COCO *train*2017 (118K images) and evaluated on COCO *val*2017 (40K images). Here, we adopt the latest version of MMDetection toolbox [11] as our detection framework. In our experiments, we test AFOpt (Ada_AFOpt) by Mask-RCNN [28] and RetinaNet [43] with Swin transformer [46], and for a more clear insight, we also give the results of only

Algorithm 8 Ada_AFOpt

Input: $\mathbf{W}^0$, $\mathbf{L_{QQ}}^0 = \mathbf{0}_{C_{out}}$,$\mathbf{L_{KK}}^0 = \mathbf{0}_{C_{out}}$, $\mathbf{R_{VV}}^0 = \mathbf{0}_{C_{in}}$,$\mathbf{R_{XX}}^0 = \mathbf{0}_{C_{in}}$, τ, β_1, β_2, ε
Output: $\mathbf{W}_T$

for *t=1:T* **do**
  Extract $\mathbf{X}^t$, $\mathbf{F_Q}^t$, $\mathbf{F_K}^t$, $\mathbf{F_V}^t$, $\mathbf{X}_P^t$ by forward and backward propagation. Compute the weight gradient $\mathbf{G}^t$.
  if $t\%T_s = 0$ **then**
    Compute $\mathbf{L_{KK}}^t$, $\mathbf{L_{QQ}}^t$, $\mathbf{R_{VV}}^t$ and $\mathbf{R_{XX}}^t$ by Eq.(5.19), Eq.(5.20) and Eq.(5.21); Apply statistics momentum Eq.(5.18) on $\mathbf{L_{KK}}^t$, $\mathbf{L_{QQ}}^t$, $\mathbf{R_{VV}}^t$ and $\mathbf{R_{XX}}^t$;
  else
    $\mathbf{L_{KK}}^t = \mathbf{L_{KK}}^{t-1}$, $\mathbf{L_{QQ}}^t = \mathbf{L_{QQ}}^{t-1}$, $\mathbf{R_{VV}}^t = \mathbf{R_{VV}}^{t-1}$ and $\mathbf{R_{XX}}^t = \mathbf{R_{XX}}^{t-1}$.
  end
  if $t\%T_u = 0$ **then**
    Apply adaptive dampening Eq.(5.17) onto $\mathbf{L_{KK}}^t$, $\mathbf{L_{QQ}}^t$, $\mathbf{R_{VV}}^t$ and $\mathbf{R_{XX}}^t$; Compute the inverse $\mathbf{D_{KK}}^t$, $\mathbf{D_{QQ}}^t$, $\mathbf{D_{VV}}^t$ and $\mathbf{D_{XX}}^t$ by Eq. (5.22);
  else
    $\mathbf{D_{KK}}^t = \mathbf{D_{KK}}^{t-1}$, $\mathbf{D_{QQ}}^t = \mathbf{D_{QQ}}^{t-1}$, $\mathbf{D_{VV}}^t = \mathbf{D_{VV}}^{t-1}$ and $\mathbf{D_{XX}}^t = \mathbf{D_{XX}}^{t-1}$;
  end
  if *attention module and other linear layer* **then**
    Compute gradient $\hat{\mathbf{G}}^t$ by Eq.(5.23) and Eq.(5.24), respectively;
  else
    $\hat{\mathbf{G}}^t = \mathbf{G}^t$.
  end
  Apply gradient norm recovery by Eq.(5.16).
  $\boldsymbol{M}^t = \beta_1 \boldsymbol{M}^{t-1} + (1-\beta_1)\hat{\mathbf{G}}^t$;
  $\boldsymbol{V}^t = \beta_2 \boldsymbol{V}^{t-1} + (1-\beta_2)\hat{\mathbf{G}}^t \odot \hat{\mathbf{G}}^t$;
  $\hat{\boldsymbol{M}}^t = \frac{\boldsymbol{M}^t}{1-\beta_1^t}$, $\hat{\boldsymbol{V}}^t = \frac{\boldsymbol{V}^t}{1-\beta_2^t}$;
  $\boldsymbol{W}^{t+1} = \boldsymbol{W}^t - \tau \frac{\hat{\boldsymbol{M}}^t}{\sqrt{\hat{\boldsymbol{V}}^t}+\varepsilon}$.
end

FSGD (FAdam[1]) without further correcting the gradient of attention module. Other configurations follow the settings of the official MMDetection toolbox[2], and we cite the results of default optimizer AdamW from official results directly[3] while reproduce the result of SGDM. The experiments in this section are conducted on NVIDIA

[1]FAdam is proposed in[90]. It combines FSGD with AdamW to share the merits of adaptive stepsize methods.

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

[3]Please refer to `https://github.com/open-mmlab/mmdetection/tree/master/configs/swin`.

Table 5.1: Detection results of RetinaNet for different parmeters on COCO. Δ_{FSGD} and Δ_{AFOpt} means the gain of FSGD and AFOpt over AdamW, respectively. * indicates the default optimizer.

Backbone	Lr schedule	Algorithm	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
Swin-T	1X	SGDM*	37.3	57.4	39.6	22.2	40.7	50.6
		FSGD	37.7	58.3	39.8	23.4	41.3	50.4
		Δ_{FSGD}	↑0.4	↑0.9	↑0.2	↑1.2	↑0.6	↓0.2
		AFOpt	38.5	59.0	40.8	22.4	41.9	51.5
		Δ_{AFOpt}	↑1.2	↑1.6	↑1.2	↑0.2	↑1.2	↑0.9

RTX A6000 GPUs. For the default parameters of FSGD and AFOpt, we set all the parameters the same, specifically, we set the infrequent update interval $T_s = 50$ and $T_u = 500$, the statistic decay parameter $\alpha = 0.95$, while the adaptive dampening parameter $\eta = 0.01$. Meanwhile, in AFOpt, since we made the assumption $\mathbf{W_{VV}}$ have inverse, we add 0.0001 on the diagonal of $\mathbf{W_{VV}}$ to match the assumption.

We list the Average Precision (AP) of object detection by RetinaNet in Table 5.1, and show the AP^b of detection and AP^m of segmentation by Mask-RCNN in Table 5.2. From Table 5.1, we see that most indexes have been greatly improved, and overall the performance of AFOpt is better than FSGD, not only in the value of index improvement but also in the overall improvement stability. Figure 5.2 shows the loss curve with respect to iteration and the mAP index of detection with respect to epoch. From the loss curves, AFOpt accelerates the convergence compared with FSGD under this case, and the two optimizers mentioned both converge better than the baseline SGDM. The increase of mAP shows the same results in this group of experiment. Meanwhile, in Table 5.2, Ada_AFOpt improves all the indexes with at least 0.5%, while FAdam improves from 0.3%, which may show the improvement stability of Ada_AFOpt. Figure 5.3 shows the mAP index of detection and segmentation with respect to epoch, while Ada_AFOpt is much better than the baseline and slightly better than FAdam under most epochs. Improving Ada_AFOpt to get better generalization performance than FAdam will be our future work.

Table 5.2: Detection and segmentation results of Mask-RCNN for different parameters on COCO. Δ_{FAdam} and Δ_{Ada_AFOpt} means the gain of FSGD and AFOpt over AdamW, respectively. * indicates the default optimizer.

Backbone	Lr schedule	Algorithm	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
Swin-T	1X	AdamW*	42.7	65.2	46.8	39.3	62.2	42.2
		FAdam	43.4	66.0	47.1	39.7	62.7	42.7
		Δ_{FAdam}	↑0.7	↑0.8	↑0.3	↑0.4	↑0.5	↑0.5
		Ada_AFOpt	43.4	65.8	47.5	39.8	62.7	42.7
		Δ_{Ada_AFOpt}	↑0.7	↑0.6	↑0.7	↑0.5	↑0.5	↑0.5

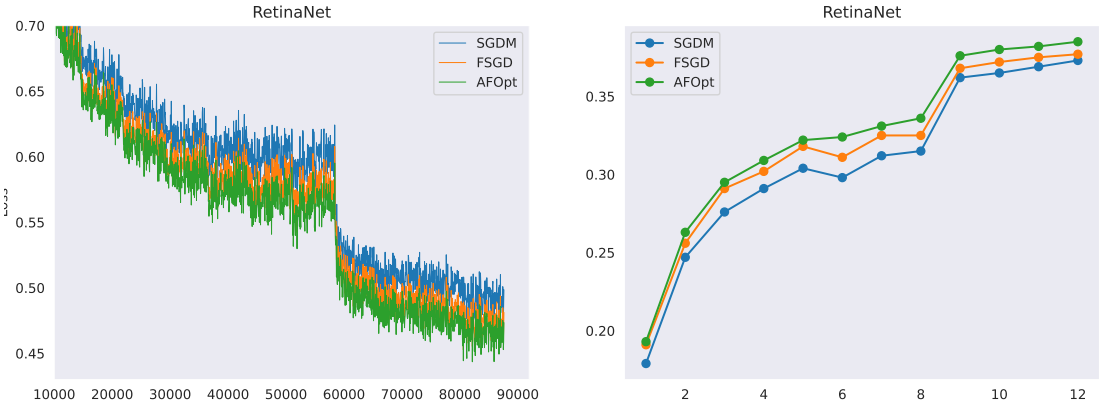


Figure 5.2: The loss curve (left) and the mAP index (right) of detection for different optimizers in training RetinaNet with Swin-T backbone, 1× schedule.

5.4.2 Ablation Study

Adaptive dampening. Since adaptive dampening is strongly related to numerical stability, the dampening parameter is important and deserves further ablation study. Here we do the ablation study on RetinaNet with Swin backbone. Except for the dampening parameter η, the other settings remain the same as the experiments in the above section. In Table 5.3, we report the experiment result of different dampening parameters $0.1, 0.01, 0.001, 0.001$, where 0.01 gets the best performance. Although the result of 0.001 is similar to the baseline, we still got an extensive range of dampening parameters, which performs better than FSGD and SGDM.

Infrequent Updating. Here, we test different update intervals T_s and T_u to see

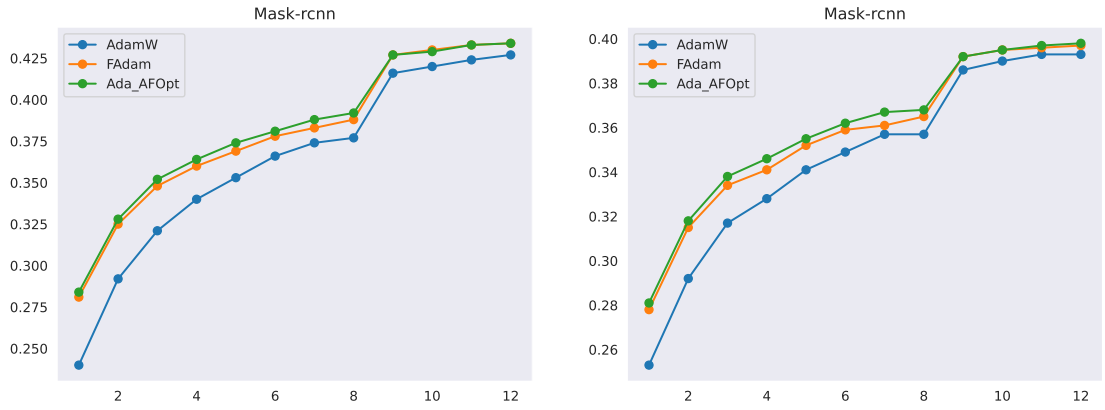


Figure 5.3: The mAP index for detection (left) and for segmentation (right) of different optimizers in training Mask-Rcnn with Swin-T backbone, 1× schedule.

Table 5.3: Detection results of RetinaNet, Swin-T backbone for different dampening parameter on COCO.

Backbone	Lr schedule	Dampening	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
Swin-T	1X	0.1	38.3	58.7	40.4	22.5	**41.9**	50.9
		0.01	**38.5**	**59.0**	**40.8**	22.4	**41.9**	**51.5**
		0.001	38.0	58.5	40.5	**23.2**	41.5	50.5
		0.0001	37.4	57.9	39.9	21.6	40.7	50.2

the influence of the update frequency. We test $T_s = 20, 50, 100, 200$ with corresponding $T_u = 10T_s$, and report the corresponding results in Table 5.4. The experiment results show that the optimizer is stable to the interval in the given range, and the results do not vary a lot. Although we choose a small interval $T_s = 50$ and $T_u = 500$ since intuitively, a small interval may be better because it can gather more information, larger interval like $T_s = 200$ and $T_u = 2000$ can also be adapted to balance computational cost.

5.4.3 Analysis

By the experimental results, our method still has some areas that need to be analyzed and improved. First, we have several approximations in our methods, which may cause some performance deduction. First, for matrix $\mathbf{W_Q}$ and $\mathbf{W_K}$,

Table 5.4: Detection results of RetinaNet, Swin-T backbone for different updating interval on COCO.

Backbone	Lr schedule	T_s, T_u	AP	$AP_{.5}$	$AP_{.75}$	AP_s	AP_m	AP_l
Swin-T	1X	20,200	38.3	58.8	40.6	22.5	41.7	50.7
		50,500	**38.5**	**59.0**	40.8	22.4	41.9	**51.5**
		100,1000	38.3	58.6	40.5	22.8	41.7	50.8
		200,2000	**38.5**	58.9	**40.9**	**22.9**	**42.0**	51.2

we approximate the softmax function by first-order Taylor expansion to deal with the difficulties caused by the nonlinear function. Second, because some positional embedding information is added onto the attention matrix before passing to the softmax function and it is difficult for us to acquire the information, the attention matrix we compute and adopt in the optimizer is not precisely the one passed to the softmax function when training the model, which may encounter some inaccuracy. Finally, for matrix $\mathbf{W_V}$, since we convert its computation by utilizing the matrix $\mathbf{X}_P$, we assume $\mathbf{W_{VV}}$ to be inversible and assume all the samples share a same $\mathbf{W_V}$, which may influence the performance of the optimizer.

There are still some future works. Firstly, given the approximations above, some better approximations may be considered for a better performance. Secondly, AdamW is more popular in training transformers, but the results of Ada_AFOpt need to be more satisfactory. As we show in Figure 5.3, Ada_AFOpt does not significantly improve the results during the training process compared to FAdam. Some more efficient ways to combine AFOpt with adaptive stepsize methods need further research. Finally, we also hope to validate the results on more tasks beyond computer vision.

5.5 Conclusion

In this chapter, we propose an attention-feature-based optimizer named AFOpt. By considering optimizing the attention module as a whole, we make the parameters

within attention interact with each other to assist in training and better shaping the output. Specifically, we first apply gradient descent on the output of the attention module, and then solve the problem of making the descent of parameters within attention modules close to the result of attention feature descent. On the one hand, optimization in feature space may be more accurate; on the other hand, update interactions occur because of the specific structure of attention, which may help the optimization. Experiments on detection and segmentation tasks show the effect of our proposed method.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Preconditioned methods have widely applications in DNNs optimization. In this thesis, we mainly design four preconditioned optimizers from different perspectives, and conduct experiments on computer vision tasks to prove their advantages and effectiveness.

Chapter 2 proposes SGD-PH, a combined optimizer that integrates first-order and second-order information. For some particular layers that contain mainly channel-wise 1D parameters but greatly enhance the performance (e.g., BN layers), we compute the partially accurate Hessian of the channel-wise 1D parameters and utilize this second-order derivative as the preconditioner to update them. Besides normalization layers, we also reformulated the convolutional layer by WN, from which the length vector is a channel-wise 1D parameter and can be optimized by SGD-PH. Sufficient experiments are accomplished to verify the robustness and adaptability of our proposed SGD-PH.

In Chapter 3, we mainly consider the unavoidable inversion process in preconditioned method. Here for the well known KFAC method, we combined Newton's iteration into computing the inversion, resulted in the optimizer NKFAC. The key point of this method is that, the fisher information matrix changes slow during

training, which means the inversion of the last iteration is a good initial point of the current iteration. Combining with the fast local convergence of Newton's iteration, our inversion only needs several steps, which will be computational friendly. Meanwhile, we implement NKFAC with some useful techniques. It is proved by experiments that NKFAC is much stable to inversion and considerably enhance the generalization performance.

Chapter 4 proposes AdamR, an adaptive learning rate optimizer with a rotation transformation. This optimizer, similarly as AdaGrad, is a preconditioned optimizer that designed to have a lower online regret bound. This optimizer consists of three key steps: gradient computation with rotation, execution of the standard Adam step, and reorientation of the gradient back to its original space. To reduce computational cost, we also apply layer-wise block diagonal constraint and the Kronecker factorized constraint on the rotation transform, converting the preconditioner to be a left and a right one. AdamR can inherit the hyperparameters and advantages of adaptive stepsize methods, and experiments on computer vision tasks demonstrate its effectiveness.

In Chapter 5, we focus on the popular attention module and propose AFOpt. AFOpt contains both left and right preconditioners. By regarding the attention module as a whole and approximating attention feature gradient descent, the attention parameters interact in the update, which may better utilize the feature information and similarities of different patches to assist the optimization. We conduct experiments on detection and segmentation tasks to show the effect of our proposed method.

6.2 Future Work

Following the optimizers proposed in this thesis, some further research directions exist. In future work, we will conduct research on the following possible topics.

- Efficient and effective second-order optimizers. Second-order optimizers usually need more computational time and memory compared with first-order optimizers. In SGD-PH, we also need the second-time back-propagation, which requires more time. Therefore, how to compute or approximate second-order Hessian efficiently is still an important topic.

- Inversion approximation in preconditioned optimizers. Inverse computation is unavoidable in preconditioned methods. In NKFAC, we approximate the exact inversion by Newton's method, which is a numerical approximation that can reduce the computational cost. Better approximating the inverse by numerical or theoretical means are both topics that deserve to be studied.

- More economic optimizer while lowering the online regret bound. AdamR successfully lowers the online regret bound but requires more computational cost. Therefore, designing low regret bound optimizers with less computational cost is still a topic that can be studied.

- Combination of the preconditioned gradient with adaptive stepsize methods. In AFOpt, we directly embed the preconditioned gradient into the adaptive stepsize method. However, the result is not satisfactory enough. Since Adam and AdamW have been widely used in training transformers, combining the adaptive stepsize method effectively is a topic that is worth researching.

References

[1] Saurabh Adya, Vinay Palakkode, and Oncel Tuzel. Nonlinear conjugate gradients for scaling synchronous distributed dnn training. *arXiv preprint arXiv:1812.02886*, 2018.

[2] 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.

[3] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International conference on machine learning*, pages 173–182, 2016.

[4] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[5] Achraf Bahamou, Donald Goldfarb, and Yi Ren. A mini-block natural gradient method for deep neural networks. *arXiv preprint arXiv:2202.04124*, 2022.

[6] Adi Ben-Israel. An iterative method for computing the generalized inverse of an arbitrary matrix. *Mathematics of Computation*, pages 452–455, 1965.

[7] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *International Conference on Machine Learning*, pages 557–565. PMLR, 2017.

[8] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European conference on computer vision*, pages 213–229. Springer, 2020.

[9] Siddhartha Chandra and Iasonas Kokkinos. Fast, exact and multi-scale inference for semantic image segmentation with deep gaussian crfs. In *European conference on computer vision*, pages 402–418. Springer, 2016.

[10] 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.

[11] 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.

[12] Kai Chen, Jiaqi Wang, Jiangmiao Pang, Yuhang Cao, Yu Xiong, Xiaoxiao Li, Shuyang Sun, Wansen Feng, Ziwei Liu, Jiarui Xu, Zheng Zhang, Dazhi Cheng, Chenchen Zhu, Tianheng Cheng, Qijie Zhao, Buyu Li, Xin Lu, Rui Zhu, Yue Wu, Jifeng Dai, Jingdong Wang, Jianping Shi, Wanli Ouyang, Chen Change Loy, and Dahua Lin. MMDetection: Open mmlab detection toolbox and benchmark. *arXiv preprint arXiv:1906.07155*, 2019.

[13] MMClassification Contributors. Openmmlab's image classification toolbox and benchmark. `https://github.com/open-mmlab/mmclassification`, 2020.

[14] Aaron Defazio and Konstantin Mishchenko. Learning-rate-free learning by d-adaptation. In *International Conference on Machine Learning*, pages 7449–7479. PMLR, 2023.

[15] Xingping Dong, Jianbing Shen, Wenguan Wang, Yu Liu, Ling Shao, and Fatih Porikli. Hyperparameter optimization for tracking with continuous deep q-learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 518–527, 2018.

[16] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

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

[18] Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-fac: Efficient matrix-free approximations of second-order information. *Advances in Neural Information Processing Systems*, 34:14873–14886, 2021.

[19] Matteo Gadaleta, Federico Chiariotti, Michele Rossi, and Andrea Zanella. D-dash: A deep q-learning framework for dash video streaming. *IEEE Transactions on Cognitive Communications and Networking*, 3(4):703–718, 2017.

[20] Kai-Xin Gao, Xiao-Lei Liu, Zheng-Hai Huang, Min Wang, Shuangling Wang, Zidong Wang, Dachuan Xu, and Fan Yu. Eigenvalue-corrected natural gradient based on a new approximation. *arXiv preprint arXiv:2011.13609*, 2020.

[21] Kaixin Gao, Xiaolei Liu, Zhenghai Huang, Min Wang, Zidong Wang, Dachuan Xu, and Fan Yu. A trace-restricted kronecker-factored approximation to natural gradient. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7519–7527, 2021.

[22] 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.

[23] Donald Goldfarb, Yi Ren, and Achraf Bahamou. Practical quasi-newton methods for training deep neural networks. *Advances in Neural Information Processing Systems*, 33:2386–2396, 2020.

[24] 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.

[25] Harshit Gupta, Kyong Hwan Jin, Ha Q Nguyen, Michael T McCann, and Michael Unser. Cnn-based projected gradient descent for consistent ct image reconstruction. *IEEE transactions on medical imaging*, 37(6):1440–1453, 2018.

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

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

[28] 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.

[29] 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.

[30] Magnus R Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving. *Journal of research of the National Bureau of Standards*, 49(6):409, 1952.

[31] 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.

[32] 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.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

[35] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2, 2019.

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

[37] Abdoulaye Koroko, Ani Anciaux-Sedastrian, Ibtihel Gharbia, Valérie Garès, Mounir Haddou, and Quang Huy Tran. Efficient approximations of the fisher matrix in neural networks using kronecker product singular value decomposition. *arXiv preprint arXiv:2201.10285*, 2022.

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

[39] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[40] Susanna Lange, Kyle Helfrich, and Qiang Ye. Batch normalization preconditioning for neural network training. *Journal of Machine Learning Research*, 23(72):1–41, 2022.

[41] Weiguo Li and Zhi Li. A family of iterative methods for computing the approximate inverse of a square matrix and inner inverse of a non-square matrix. *Applied Mathematics and Computation*, 215(9):3433–3442, 2010.

[42] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017.

[43] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[44] 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.

[45] 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.

[46] 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.

[47] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[48] Hao Luo, Youzhi Gu, Xingyu Liao, Shenqi Lai, and Wei Jiang. Bag of tricks and a strong baseline for deep person re-identification. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2019.

[49] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[50] Xuezhe Ma. Apollo: An adaptive parameter-wise diagonal quasi-newton method for nonconvex stochastic optimization. *arXiv preprint arXiv:2009.13586*, 2020.

[51] James Martens. Deep learning via hessian-free optimization. In *ICML*, volume 27, pages 735–742, 2010.

[52] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[53] Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479(480):104, 1969.

[54] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[55] H Saberi Najafi and M Shams Solary. Computational algorithms for computing the inverse of a square matrix, quasi-inverse of a non-square matrix and block matrices. *Applied mathematics and computation*, 183(1):539–550, 2006.

[56] Lam Nguyen, Phuong Ha Nguyen, Marten Dijk, Peter Richtárik, Katya Scheinberg, and Martin Takác. Sgd and hogwild! convergence without the bounded gradients assumption. In *International Conference on Machine Learning*, pages 3750–3758. PMLR, 2018.

[57] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-scale distributed second-order optimization using kronecker-factored approximate curvature for deep convolutional neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[58] Christopher C Paige and Michael A Saunders. Solution of sparse indefinite systems of linear equations. *SIAM journal on numerical analysis*, 12(4):617–629, 1975.

[59] Chao Peng, Tete Xiao, Zeming Li, Yuning Jiang, Xiangyu Zhang, Kai Jia, Gang Yu, and Jian Sun. Megdet: A large mini-batch object detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 6181–6189, 2018.

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

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

[62] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. In *International conference on learning representations*, 2016.

[63] 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.

[64] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Jun 2017.

[65] Ergys Ristani, Francesco Solera, Roger Zou, Rita Cucchiara, and Carlo Tomasi. Performance measures and a data set for multi-target, multi-camera tracking. In *European conference on computer vision*, pages 17–35. Springer, 2016.

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

[67] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[68] 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.

[69] Tara N Sainath, Lior Horesh, Brian Kingsbury, Aleksandr Y Aravkin, and Bhuvana Ramabhadran. Accelerating hessian-free optimization for deep neural networks by implicit preconditioning and sampling. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 303–308. IEEE, 2013.

[70] Tim Salimans and Diederik P. Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. 2016.

[71] G. Schulz. Iterative berechnung der reziproken matrix. *Z. Angew. Math. Mech.*, 13:57–59, 1933.

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

[73] Walter A Shewhart and Samuel S Wilks. *Wiley series in probability and mathematical statistics*. Wiley, 1975.

[74] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

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

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

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[78] Fazlollah Soleymani. On a fast iterative method for approximate inverse of matrices. *Communications of the Korean Mathematical Society*, 28(2):407–418, 2013.

[79] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[80] Zedong Tang, Fenlong Jiang, Maoguo Gong, Hao Li, Yue Wu, Fan Yu, Zidong Wang, and Min Wang. Skfac: Training neural networks with faster kronecker-factored approximate curvature. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13479–13487, 2021.

[81] 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.

[82] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance normalization: The missing ingredient for fast stylization. *arXiv preprint arXiv:1607.08022*, 2016.

[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[84] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080*, 2016.

[85] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *PhD thesis, Committee on Applied Mathematics, Harvard University, Cambridge, MA*, 1974.

[86] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pages 3–19, 2018.

[87] Xingyu Xie, Pan Zhou, Huan Li, Zhouchen Lin, and Shuicheng Yan. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2024.

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

[89] 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.

[90] Hongwei Yong. Effective and efficient optimization methods for deep learning. 2022.

[91] 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.

[92] Hongwei Yong, Ying Sun, and Lei Zhang. A general regret bound of preconditioned gradient method for dnn training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7866–7875, 2023.

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

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

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

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

[97] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising. *IEEE transactions on image processing*, 26(7):3142–3155, 2017.

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

[99] Liang Zheng, Liyue Shen, Lu Tian, Shengjin Wang, Jingdong Wang, and Qi Tian. Scalable person re-identification: A benchmark. In *Proceedings of the IEEE international conference on computer vision*, pages 1116–1124, 2015.

[100] Zhedong Zheng, Liang Zheng, and Yi Yang. Unlabeled samples generated by gan improve the person re-identification baseline in vitro. In *Proceedings of the IEEE international conference on computer vision*, pages 3754–3762, 2017.

[101] Juntang Zhuang, Tommy Tang, Yifan Ding, Sekhar Tatikonda, Nicha Dvornek, Xenophon Papademetris, and James S Duncan. Adabelief optimizer: Adapting stepsizes by the belief in observed gradients. *arXiv preprint arXiv:2010.07468*, 2020.