EFFECTIVE AND EFFICIENT OPTIMIZATION METHODS FOR DEEP LEARNING

YONG HONGWEI

PhD

The Hong Kong Polytechnic University

2022

The Hong Kong Polytechnic University
Department of Computing

Effective and Efficient Optimization Methods for Deep Learning

Yong Hongwei

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
March 2022

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 acknowledgment has been made in the text.

Signature: _____

Name of Student: Yong Hongwei

Abstract

Optimization techniques play an essential role in deep learning, and a favorable optimization approach can greatly boost the final performance of the trained deep neural network (DNN). Generally speaking, there are two major goals for a good DNN optimizer: accelerating the training process and improving the model generalization capability. In this thesis, we study the effective and efficient optimization techniques for deep learning.

Batch normalization (BN) is a key technique for stable and effective DNN training. It can simultaneously improve the model training speed and the model generalization performance. However, it is well-known that the training and inference stages of BN have certain inconsistency, and the performance of BN will drop largely when the training batch size is small. In Chapter 2, we prove that BN actually introduces a certain level of noise into the sample mean and variance during the training process. We then propose a Momentum Batch Normalization (MBN) method to control the noise level and improve the training with BN. Meanwhile, in Chapter 3, we put forward an effective inference method of BN, *i.e*, Batch Statistics Regression (BSR), which uses the instance statistics to predict the batch statistics with a simple linear regression model. BSR can more accurately estimate the batch statistics, making the training and inference of BN much more consistent. We evaluate them on CIFAR100/CIFAR100, Mini-ImageNet, ImageNet, *etc*.

Gradient descent is dominantly used to update DNN models for its simplicity and

efficiency to handle large-scale data. In Chapter 4, we present a simple yet effective DNN optimization technique, namely gradient centralization (GC), which operates directly on gradients by centralizing the gradient vectors to have zero mean. GC can be viewed as a projected gradient descent method with a constrained loss function. We show that GC can regularize both the weight space and output feature space so that it can boost the generalization performance of DNNs. In Chapter 5, we present a feature stochastic gradient descent (FSGD) method to approximate the desired feature outputs with one-step gradient descent. FSGD improves the singularity of feature space and thus enhances feature learning efficacy. Finally, in Chapter 6 we propose a novel optimization approach, namely Embedded Feature Whitening (EFW), which overcomes the several drawbacks of conventional feature whitening methods while inheriting their advantages. EFW only adjusts the gradient of weight by using the whitening matrix without changing any part of the network so that it can be easily adopted to optimize pre-trained and well-defined DNN architectures. We testify them on various tasks, including image classification on CIFAR100/CIFAR100, ImageNet, fine-grained image classification datasets, and object detection and instance segmentation on COCO, and them achieve obvious performance gains.

In summary, in this thesis, we present five deep learning optimization methods. Among them, MBN and BSR improve the BN training and inference, respectively; GC adjusts the gradient of weight with a centralization operation; FSGD provides a practical approach to perform feature-driven gradient descent; and EFW embeds the existing feature whitening into the optimization algorithms for effective deep learning. Extensive experiments demonstrate their effectiveness and efficiency for DNN optimization.

Publications Arising from the Thesis

Conference Papers

1. **Hongwei Yong**, Jianqiang Huang, Xiansheng Hua, Lei Zhang. *"Gradient Centralization: A New Optimization Technique for Deep Neural Networks"* , European Conference on Computer Vision (ECCV) 2020. (Oral)

2. **Hongwei Yong**, Jianqiang Huang, Deyu Meng, Xiansheng Hua, and Lei Zhang. *"Momentum Batch Normalization for Deep Learning with Small Batch Size"*, European Conference on Computer Vision (ECCV) 2020.

3. Hongyi Zheng*, **Hongwei Yong***[1], Lei Zhang. *"Deep Convolutional Dictionary Learning for Image Denoising"*, IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2021.

4. Wangmeng Xiang, **Hongwei Yong**, Lei Zhang. *"Second-order Camera-aware Color Transformation for Cross-domain Person Re-identification"*, Asian Conference on Computer vision (ACCV), 2020

5. Jin Xiao,**Hongwei Yong**, Lei Zhang. *"Degradation Model Learning for Real-World Single Image Super-resolution"*, Asian Conference on Computer vision (ACCV), 2020

[1]* indicates co-first author, responsable for model building and methodology writing

6. Zongsheng Yue, **Hongwei Yong**, Qian Zhao, Deyu Meng, Lei Zhang. *"Variational Denoising Network: Toward Blind Noise Modeling and Removal"*, Advances in neural information processing systems (NeurIPS), 2019

7. Jianrui Cai, Hui Zeng, **Hongwei Yong**, Zisheng Cao, Lei Zhang. *"Toward Real-world Single Image Super-resolution: A new Benchmark and A New Model"*, IEEE International Conference on Computer Vision (ICCV), 2019. (Oral)

Journal Papers

1. **Hongwei Yong**, Jianqiang Huang, Wangmeng Xiang, Xiansheng Hua, Lei Zhang. *"Panoramic Background Image Generation for PTZ Cameras"*, IEEE Transactions on Image Processing (TIP), vol. 28, no. 7, pp. 3162-3176, Jan. 2019

2. Zongsheng Yue*, **Hongwei Yong***[2], Deyu Meng, Qian Zhao, Yee Leung, Lei Zhang. *"Robust Multiview Subspace Learning With Nonindependently and Non-identically Distributed Complex Noise"*, IEEE Transactions on Neural Networks and Learning Systems, 31(4), 1070-1083, June 2019

3. Hui Li, Kede Ma, **Hongwei Yong**, Lei Zhang. *"Fast Multi-Scale Structural Patch Decomposition for Multi-Exposure Image Fusion"*, IEEE Transactions on Image Processing (TIP), vol. 29, pp.5805-5816, April 2020

4. Zhihang Fu, Yaowu Chen, **Hongwei Yong**, Rongxin Jiang, Lei Zhang, Xian-Sheng Hua. *"Foreground Gating and Background Refining Network for Surveillance Object Detection"*, IEEE Transactions on Image Processing, vol. 28, issue 12, pp. 6077-6090. June 2019

Under Review

[2]* indicates co-first author, responsable for model building and experiment designing

1. **Hongwei Yong**, Lei Zhang. *"An Embedded Feature Whitening Approach to Deep Neural Network Optimization"* Submitted to European Conference on Computer Vision (ECCV) 2022

2. **Hongwei Yong**, Lei Zhang. *"Batch Statistics Regression for Effective Inference of Batch Normalization"* Submitted to European Conference on Computer Vision (ECCV) 2022

3. Hongyi Zheng, **Hongwei Yong**, Lei Zhang. *"Unfolded Deep Kernel Estimation for Blind Image Super-resolution"* Submitted to European Conference on Computer Vision (ECCV) 2022

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor Prof. Lei Zhang for his continuous support of my Ph.D. study and related research with his generosity, patience, motivation, and immense knowledge. I met Prof. Lei Zhang in 2015 when I was a visiting research student in his group, I clearly remembered the tremendous and valuable works done by his Visual Computing Lab and I was highly impressed by his enduring commitment to serving society through his research results. During my Ph.D. study, when I faced some difficulties in my research and study, Prof. Lei Zhang can always get me out of trouble with his constructive and valuable guidance. It is worth for me to spend four wonderful years with so many brilliant, positive and hard-working researchers in his research group at the Hong Kong Polytechnic University.

Besides my Ph.D. supervisor, I would like to thank my master supervisor Prof. Deyu Meng, for his insightful comments and encouragement. He gave me a great help to stimulate my interest in research and provided me with some basic training on how to conduct research. His passion for research also inspires me a lot as far as now. Meanwhile, I would like to thank my cooperators Dr. Kede Ma and Dr. Kai Zhang for their insightful comments and encouragement. Discussing with them usually makes me widen my research from various perspectives. My sincere thanks also go to DAMO Academy, Alibaba Group who provided me an opportunity to join as an intern, and who gave me access to the laboratory and research facilities. Without such precious

support, it would not be possible to conduct this research.

Last but not the least, I would like to thank my family: my parents, my wife, and my child for supporting me spiritually throughout writing this thesis and my life in general.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep learning has achieved a great success in many applications, including image classification [22], object detection [72, 20], image and video segmentation [72, 20] and image restoration [109], natural language processing [58] and computer games [62, 81], etc. The success of deep neural networks (DNNs) comes from the advances in higher computing power (*e.g*, GPUs), large scale datasets [15], and learning algorithms [37, 86, 16]. In particular, advanced network architecture [22, 25] and optimization techniques [37, 43] have been developed, making the training of very deep networks from a large amount of training data possible.

The optimization methods plays a key role in DNN learning. Traditional machine learning methods often formulate the optimization problem by using a convex objective function and constraints. Nevertheless, in deep learning, the objective function is highly non-convex, and there can be a large number of local minima, which makes it is almost impossible to find the global minimum of the loss function. The saddle point also increases the difficulties of optimization. The gradient descent methods will cost a lot of time to escape the region around the saddle points. Besides, one of the most challenging issues in DNN optimization is the gradient vanishing and exploding problem. Too large gradients make training inefficient, while too large gra-

dients make training unstable. For a long time, the gradient vanishing and exploding problem impedes the development of optimization in deep learning.

There are two major goals for an ideal DNN optimizer: accelerating the training process and improving the model generalization capability. The first goal aims to spend less time and cost to reach a good local minimum, while the second goal aims to ensure that the learned DNN model can make accurate predictions on test data. By far, gradient descent methods still dominate the optimization algorithms in deep learning because they are efficient and memory-saving. A commonly used class of DNN optimizers are the stochastic gradient descent (SGD) [6, 7] and its variants [68, 37], which iteratively update parameter in the opposite direction to the gradient obtained by the backpropagation (BP) algorithm [74]. There are three steps to optimize the parameters of a DNN: firstly, we need to compute the output of the DNN, which is called the forward propagation; secondly, we measure the loss between the output of the DNN with the target, and compute the gradient of the loss w.r.tthe parameters of weights in the DNN, which is called the backward propagation; finally, after the gradients of weights are obtained, we adopt a gradient descent based optimization algorithm to update the weights. These three steps are repeated again and again until convergence. In summary, the DNN optimization involves mainly three components: forward propagation, backward propagation, and gradient descent for weight update.

For the forward propagation, the information of input should be delivered effectively to the output layer and a good feature space is expected, while for the backward propagation, the information from the target is to be embedded into the gradient of weight. However, due to the gradient vanishing and exploding problem, it is hard to transmit such information in a very deep neural network. To address this problem, a variety of methods have been proposed, such as weight initialization strategies [17, 21], efficient active functions (e.g., ReLU [63]), skip connection [22], feature normalization layer [32], and so on. Among the above techniques, the feature normalization and

whitening methods are typical ones to make feature propagation more stable and effective in DNN optimization. The representative feature normalization methods include batch normalization (BN) [32], instance normalization (IN) [88, 30], layer normalization (LN) [44] and group normalization (GN) [93]. Among them, BN is the most widely used one, which uses the mean and variance of the intermediate features within a mini-batch to perform Z-score standardization. Though BN can speed up the training process and improve the model generalization performance, it ignores the feature correlation among different dimensions. Therefore, some feature whitening methods, *e.g*, Decorrelated Batch Normalization (DBN) [28] and Iterative Normalization (IterNorm) [29], have been proposed to perform whitening on intermediate features, which validates that making proper use of the statistics (*e.g*, covariance matrix) of intermediate features to find a more isotropic feature space can improve the learning of DNNs models. However, the feature whitening methods will cost significant computation and memory resources, and they need to redefine the forward and backward propagation by introducing the whitening module. Such limitations hinder the use of whitening methods in practice.

For the weight update, various optimizers [68, 16, 37, 16, 37] have been proposed to achieve efficient gradient descent. SGD [6, 7] and its extension SGD with momentum (SGDM) [68] are among the most commonly used ones. They update the parameters along the opposite direction of their gradients in one training step. Most of the current DNN optimization methods are based on SGD and improve it to better overcome the gradient vanishing or explosion problem. For example, Adagrad [16] adopts an adaptive learning rate strategy for different weights, *i.e*, a larger gradient step for infrequent parameters and a smaller step for frequent ones. RMSprop and Adadelta [104] follow a similar adaptive learning rate strategy. Adam [37] introduces the momentum of gradient into the the adaptive learning rate technique, which largely stabilizes the training process. Meanwhile, some works take the second-order information into consideration. Considering the large computation and memory cost

of second-order optimization, some approaches have been developed to exploit the second-order information with acceptable cost. For instance, Adahessian [97] and Apollo [59] only consider the diagonal elements of the Hessian matrix, and the Kronecker Factored Approximation Curvature (KFAC) [61] uses a block-diagonal version of the Fisher matrix to approximate the natural gradient layer-wisely. Nowadays, the the gradient vanishing and exploding problem has been eased to some extent, and one can optimize very deep DNNs.

1.1 Overview of Optimization Techniques in Deep Learning

1.1.1 Feature Normalization and Whitening

Many popular DNNs employ the feature normalization layer as a basic module, such as ResNet50 [22] and DenseNet121 [25]. Batch normalization (BN) is the most widely-used normalization technique. BN was originally introduced to solve the internal covariate shift by normalizing the activations along the sample dimension. It has several good properties, such as allowing higher learning rates [5], accelerating the training speed, and boosting the generalization accuracy [56, 77]. Despite its great success, a well-known drawback of BN is its inconsistency between training and inference. That is, BN calculates the statistics over each mini-batch for normalization during training, while it uses the exponential moving average (EMA) of these mini-batch statistics in the inference. Unfortunately, this makes the training and inference of BN inconsistent, especially when the training batch size is small.

Some variants of BN have been proposed to improve BN. For instance, Peng *et al* [67] suggested performing the synchronized computation of BN statistics across GPUs (synchronized BN) to obtain more accurate statistics estimation in training, which

can be viewed as enlarging the training batch size. Cross-Iteration BN (CBN) [98] adopts the statistics in the recent several iterations to virtually enlarge the batch size and compensate for the network weight changes. A technique based on Taylor polynomials is developed to estimate batch statistics. Batch Re-normalization [31] and Moving Average BN [96] use the moving average of batch statistics instead of batch statistics for training, but elaborated hyper-parameter settings are needed. Different from synchronized BN, Ghost-BN [23] can be adopted to simulate multiple GPUs training with one GPU for BN, which divides the mini-batch in one GPU into several small subgroups. When the overall training batch size is large, Ghost-BN can significantly improve the generalization performance with a proper number of subgroups.

Besides performing normalization on sample dimension, some methods have been proposed to perform normalization along other dimensions. For example, layer normalization (LN) [44] normalizes all activations or feature maps along feature dimension; instance normalization (IN) [88] performs normalization for each feature map of each sample individually, and group normalization (GN) [93] normalizes feature maps for each input sample in a divided group. These normalization methods demonstrate good performance for specific applications (*e.g*, LN for RNN, IN for style transfer, and GN for object detection); however, their performance is usually not as good as BN with a proper training batch size in general. The combination of BN with these normalization methods is also proposed. For example, switchable normalization [55, 78, 54] combines BN with LN and IN with learnable weights; batch group normalization [85] combines BN and GN by performing normalization along the batch dimension in a divided feature map group; batch-channel normalization (BCN) [69] integrates BN with the momentum of batch statistics and channel-based normalization. Dynamic normalization (DN) [57] combines IN, LN, GN and BN in a unified formulation. Exemplar normalization [111] learns different data-dependent normalizations for different image samples.

Beyond normalization that only considers the mean and variance of features, the feature whitening methods consider the correlation among different dimensions of features. For example, DBN [28] conducts ZCA-whitening on features across the channel dimension by eigen-decomposition and backpropagating the transformation. To improve the efficiency of feature whitening, IterNorm [29] adopts Newton's iteration on DBN. Meanwhile, Network deconvolution (ND) [99] uses deconvolution filters to decorrelate both pixel-wise and channel-wise features before the convolution layer. In general, feature whitening methods can not only speed up training processing but also boost the generalization performance of DNNs [29, 99]. However, their drawbacks, such as heavy additional computation and memory, inapplicable to pre-trained DNN models, additional parameter introduction, *etc*, make them hard to be adopted in real-world applications.

1.1.2 Weight Normalization and Weight Constraints

Weight normalization (WN) [76] re-parameterizes the weight vectors and decouples the length of a weight vector from its direction. It speeds up the convergence of SGDM algorithm to a certain degree. Weight standardization (WS) [69] adopts the Z-score standardization to re-parameterize the weight vectors. Like BN, WS can also smooth the loss landscape and improve training speed. Besides, binarized DNN [70, 13, 12] quantifies the weight into binary values, which can improve the generalization capability for certain DNNs. Meanwhile, orthogonal constraints can be introduced into the weight optimization of DNNs. Huang *et al* [26] proposed a weight reparameterization method to solve the weight orthogonal constraints. However, a shortcoming of those methods operating on weights lies in that they cannot be directly used to fine-tune pre-trained models since the pre-trained weight may not meet their constraints. As a consequence, we have to design specific pre-training methods for them in order to fine-tune the model. Besides the hard constraints on weight, there are also some soft constraints or regularization on weight, such as kernel orthogonality

regularization [94] and convolutional orthogonality regularization [92]. Such soft constraints on weight can also boost the learning of weight and improve the generalization performance of DNNs.

1.1.3 Gradient Constraints

The momentum of gradient [68] is a commonly-used operation on gradient. By using the momentum of gradient, SGDM accelerates SGD in the relevant direction and dampens oscillations. Besides, L_2 regularization based weight decay, which introduces L_2 regularization into the gradient of weight, has long been a standard trick to improve the generalization performance of DNNs [41, 106]. To make DNN training more stable and avoid gradient explosion, gradient clipping [65, 66, 1, 36] has been proposed to train a very deep DNNs. Gradient clipping operation has been widely used on training deep neural networks, and it can avoid the case that the gradient is too large. In addition, the projected gradient methods [19, 42] and Riemannian approach [11, 90] project the gradient on a subspace or a Riemannian manifold to regularize the learning of weights.

1.1.4 Weight Update Algorithm

The update algorithm of weights plays a key role in Deep learning. Popular DNN optimizers include SGDM [68], Adagrad [16], Adam [37], RAdam [49], etc. SGDM [68] uses the momentum of the gradient to accelerate gradient descent along with relevant directions and dampens oscillations. It has been widely used in the high-level vision tasks of computer vision, *e.g*, image classification, object detection, and so on. In spite of using the same learning rate to all weights, the adaptive learning rate methods are developed to allow each weight having its own learning rate. For instance, Adagrad [16] adopts a smaller step for frequent ones and a larger gradient step for infrequent parameters. RMSprop and Adadelta [104] also employ the adaptive learning

rate strategy. Adam [37] combines the adaptive learning rate technique with gradient momentum, largely stabilizing the training process. Based on Adam, RAdam [49] controls the variance of the adaptive learning rate in the early stage of training to warm up the training, and Adabelief [115] modifies the step length by the belief in different observed gradients, which achieves satisfactory performance. Such adaptive learning rate methods usually perform better than SGDM in some specific areas, such as natural language processing, image low-level vision, *etc*.

Instead of first-order information, the second-order information can also be exploited to improve the optimization of DNN. Since the dimension of parameter space in DNNs is usually very high (*e.g*, 10^7), it is hard to directly compute with the full second-order information because of overwhelming memory and computation cost. Therefore, how to utilize the second-order information practically is the key issue for applying second-order optimization algorithms in deep learning. For instance, Adahessian [97] and Apollo [59] are proposed by updating only the diagonal elements of the Hessian matrix. Particularly, Adahessian considers only the diagonal elements of the Hessian matrix by using Hessian-free techniques, while Apollo simplifies the BFGS algorithm with only diagonal elements. The Kronecker Factored Approximation Curvature (KFAC) [61] approximates the natural gradient layer-wisely by using a block-diagonal version of the Fisher matrix, which adopts the statistics of intermediate features and their gradients to adjust the original gradient of weights. Nonetheless, in many computer vision tasks the first-order optimizers, such as SGDM and Adam, are more popularly used than second-order optimizers because of their simplicity and effectiveness.

1.1.5 Learning Rate Schedule

The learning rate schedule is also crucial for training DNNs. It determines the learning rate during training, *e.g*, between epochs or iterations. Instead of using a fixed

learning rate, an alternative is to vary the learning rate over the training process. The approaches where the learning rate changes over epochs or iterations are referred to as the learning rate schedule. The learning rate used in training can be decayed to a very small value. For instance, the commonly used learning rate schedule is step-wise learning rate decay, which decays the learning rate over a fixed number of training epochs and then keeps it constant as a small value for the remaining training epochs to facilitate fine-tuning. With a small learning rate, DNNs can learn high-frequency information. Cosine annealing learning rate schedule and warm up strategy [50], as well as cyclical learning rates schedule [84] can also help to improve the generalization performance of DNNs.

1.2 Contributions and Organization of the Thesis

This thesis consists of five works on the optimization techniques and algorithms for deep learning. Fig. 1.1 shows the focuses of the five developed methods in the thesis. To be specific, we conducted researches on the improvement of Batch Normalization, gradient-based optimization techniques, and effective weight update algorithms. For the improvement of BN, we aim to explain why BN can improve the generalization performance, how can we control its generalization strength, and how to make it inconsistent between training and inference. For gradient-based optimization techniques, we aim to propose simple yet effective techniques on gradient to speed up training and improve the generalization so that they can be easily embedded into the current popular optimizers. For the effective weight update algorithms, we aim to propose effective stochastic gradient descent updating formulas, which can achieve a great performance gain over the existing optimization algorithms and can be easily adopted into various DNN models without extra hyperparameter tuning.

The thesis is organized as follows:

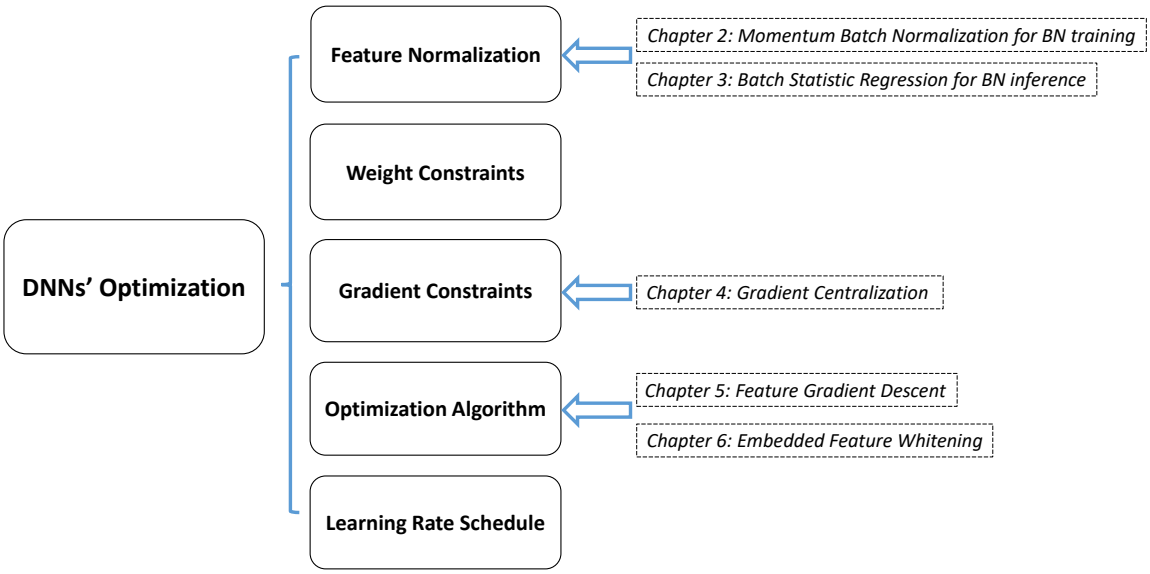


Figure 1.1: Illustration of the contributions of this thesis.

In Chapter 1, we introduce the background of DNN optimization and the major techniques.

In Chapter 2, we first prove that BN actually introduces a certain level of noise into the sample mean and variance during the training process, while the noise level depends only on the batch size. It gives a deep understanding of why BN can gain generalization performance. BN regularizes the training process by such a noise generation mechanism, and an explicit regularizer formulation of BN is also presented. Since the regularization strength of BN is determined by the batch size, a small batch size may cause the under-fitting problem, resulting in a less effective model. Meanwhile, to reduce the dependency of BN on batch size, we propose a Momentum BN (MBN) scheme by averaging the mean and variance of the current mini-batch with the historical means and variances. With a dynamic momentum parameter, we can automatically control the noise level in the training process. As a result, MBN works very well even when the batch size is very small (e.g., 2), which is hard to achieve by traditional BN.

In Chapter 3, we propose an effective inference approach of BN, *i.e*, batch statistics regression (BSR). It uses instance statistics to predict the batch statistics with a simple linear regression model. Compared with the conventional inference approach of BN, *i.e*, EMA, BSR can estimate the batch statistics more accurately, and make the training and inference of BN much more consistent. It is very easy to implement by using an online updating formulation of four statistics, whose computation and memory cost is negligible. The experimental results show that BSR can significantly improve the inference performance of BN, especially when the training batch size is small. For instance, it outperforms EMA by more than 7% in accuracy on ImageNet with ResNet50 when the training batch size is 2.

In Chapter 4, we present a new optimization technique, namely gradient centralization (GC), which operates directly on gradients by centralizing the gradient vectors to have zero mean. GC can be viewed as a projected gradient descent method with a constrained loss function. We show that GC can regularize both the weight space and output feature space so that it can boost the generalization performance of DNNs. Moreover, GC improves the Lipschitzness of the loss function and its gradient so that the training process becomes more efficient and stable. It is very simple to implement and can be easily embedded into existing gradient-based DNN optimizers with only one line of code. It can also be directly used to fine-tune the pre-trained DNNs. The experiments on various applications, including general image classification, fine-grained image classification, detection and segmentation, demonstrate that GC can consistently improve the performance of DNN learning.

In Chapter 5, we put forward a new optimizer for DNNs, namely Feature SGD (FSGD), which takes the gradient descent on intermediate features into consideration. Specifically, we use the second-order statistic matrix of intermediate features to adjust the gradient of weight. An objective function is defined to relate the gradient descent on weight to the gradient descent on feature so that FSGD can be implemented by minimizing it. FSGD can be easily adopted into different linear layers in a DNN,

including a fully-connected layer, convolutional layer, and BN layer. It improves the singularity of feature space and enhances feature learning efficacy. Meanwhile, we also show that FSGD has a close link to back-matching and feature whitening. Experimental results on CIFAR100/10, ImageNet and COCO demonstrate the superiority of FSGD to state-of-the-art DNN optimizers.

In Chapter 6, we propose a novel optimization approach, namely Embedded Feature Whitening (EFW), to DNN optimization by adjusting the gradient of weight with the ZCA transformation matrix. There are several advantages of our proposed approach. First, EFW inherits the advantages of feature whitening, *i.e*, accelerating the training process and improving the generalization performance. Second, compared with existing feature whitening methods, EFW does not introduce any module into the DNN model to be trained. As a result, it can be directly adopted to optimize most of the existing DNN models without increasing the inference time. Third, its computation and memory cost is acceptable because EFW only computes the ZCA transformation matrix once for many iterations (*e.g*, 500) and it does not store any additional intermediate features. We apply EFW to two commonly used DNN optimizers, *i.e*, SGDM and Adam (or AdamW), and name the obtained optimizers as W-SGDM and W-Adam. Extensive experimental results on various vision tasks, including image classification, object detection, segmentation and person ReID, illustrate the effectiveness of W-SGDM and W-Adam.

In Chapter 7, we conclude this thesis and present some future research directions.

1.3 Notation system

In this thesis, we denote by $\mathbf{W}$ the weight matrix, whose dimension is $C_{out} \times C_{in}$ for fully connected layers (FC layers) and $C_{out} \times C_{in} \times k_1 \times k_2$ for convolutional layers (Conv layers), where C_{in} is the number of input channels, C_{out} is the number of

output channels, and k_1, k_2 are the kernel size of convolutional layers. Denote by $\mathbf{w}_i \in \mathbb{R}^M$ $(i = 1, 2, ..., N)$ the i-th column vector of weight matrix $\mathbf{W}$. We denote by $\mathbf{A} = [\mathbf{A}_n]_{n=1}^N$ and $\mathbf{X} = [\mathbf{X}_n]_{n=1}^N$ the input and output features of the N samples in one layer. For FC layers, $\mathbf{A} \in \mathbb{R}^{C_{out} \times N}$, $\mathbf{X} \in \mathbb{R}^{C_{in} \times N}$, and $\mathbf{A} = \mathbf{W}\mathbf{X}$. For Conv layers, $\mathbf{A} \in \mathbb{R}^{C_{out} \times h \times w \times N}$, $\mathbf{X} \in \mathbb{R}^{C_{in} \times h \times w \times N}$ and $\mathbf{A} = \mathbf{W} * \mathbf{X}$, where h and w are the height and width of a feature map and "$*$" is the convolution operator. μ and σ is the mean and variance of feature $\mathbf{X}$. $p(x)$ is the distribution of x, and $E[x]$ is the expectation of x.Let $\mathcal{L}$ be the objective function, and $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$ be its gradients on activation and weight, respectively. $\mathcal{U}_1(\cdot)$ denotes the mode 1 unfold operation of a tensor. For example, for a convolution based weight matrix $\mathbf{W} \in \mathbb{R}^{C_{out} \times C_{in} \times k_1 \times k_2}$, $\mathcal{U}_1(\mathbf{W}) \in \mathbb{R}^{C_{out} \times C_{in} k_1 k_2}$. $vec(\cdot)$ denotes the vectorization function. $\mathbf{e} = \frac{1}{\sqrt{M}}\mathbf{1}$ denotes an M dimensional unit vector and $\mathbf{I}$ denotes an identity matrix.

Chapter 2

Momentum Batch Normalization for Deep Learning with Small Batch Size

Normalization layers play an important role in deep network training. As one of the most popular normalization techniques, batch normalization (BN) has shown its effectiveness in accelerating the model training speed and improving model generalization capability. The success of BN has been explained from different views, such as reducing internal covariate shift, allowing the use of large learning rate, smoothing optimization landscape, *etc*. To make a deeper understanding of BN, in this chapter, we prove that BN actually introduces a certain level of noise into the sample mean and variance during the training process, while the noise level depends only on the batch size. Such a noise generation mechanism of BN regularizes the training process, and we present an explicit regularizer formulation of BN. Since the regularization strength of BN is determined by the batch size, a small batch size may cause the under-fitting problem, resulting in a less effective model. There are two approaches to solve this problems: improving the training of BN or modifying the inference of BN. In this

chapter, we choose to improve the training of BN, while in chapter 3, we solve the problem of BN by modifying its inference. Specifically, to reduce the dependency of BN on batch size in training, we propose a momentum BN (MBN) scheme by averaging the mean and variance of the current mini-batch with the historical means and variances. With a dynamic momentum parameter, we can automatically control the noise level in the training process. As a result, MBN works very well even when the batch size is very small ($e.g$, 2), which is hard to achieve by traditional BN. We evaluate MBN on CIFAR100 and Mini-ImageNet image classification tasks, and it achieves a significant performance gain on small training batch size.

2.1 Introduction

One of the key issues in DNN training is how to normalize the training data and intermediate features. It is well-known that normalizing the input data makes training faster [43]. The widely used batch normalization (BN) technique [32] naturally extends this idea to the intermediate layers within a deep network by normalizing the samples in a mini-batch during the training process. It has been validated that BN can accelerate the training speed, enable a bigger learning rate, and improve the model generalization accuracy [22, 25]. BN has been adopted as a basic unit in most of the popular network architectures such as ResNet [22] and DenseNet [25]. Though BN has achieved a great success in DNN training, how BN works remains not very clear. Researchers have tried to explain the underlying working mechanism of BN from different perspectives. For example, it is argued in [32] that BN can reduce internal covariate shift (ICS). However, it is indicated in [77] that there is no clear link between the performance gain of BN and the reduction of ICS. Instead, it is found that BN makes the landscape of the corresponding optimization problem smoother so that it allows larger learning rates, while stochastic gradient descent (SGD) with a larger learning rate could yield faster convergence along the flat direction of the

optimization landscape so that it is less likely to get stuck in sharp minima [5].

Apart from better convergence speed, another advantage of BN is its regularization capability. Because the sample mean and variance are updated on mini-batches during training, their values are not accurate. Consequently, BN will introduce a certain amount of noise, whose function is similar to dropout. It will, however, increase the generalization capability of the trained model. This phenomenon has been empirically observed from some experimental results in [93, 105]. Teye *et al* [87, 56] tried to give a theoretical explanation of the generalization gain of BN from a Bayesian perspective; however, it needs additional assumptions and priors, and the explanation is rather complex to understand.

In this chapter, we present a simple noise generation model to clearly explain the regularization nature of BN. Our explanation only assumes that the training samples are independent and identically distributed (i.i.d.), which holds well for the randomly sampled mini-batches in the DNN training process. We prove that BN actually introduces a certain level of noise into the sample mean and variance, and the noise level only depends on the batch size. When the training batch size is small, the noise level becomes high, increasing the training difficulty. We consequently propose a momentum BN (MBN) scheme, which can automatically control the noise level in the training process. MBN can work stably for different mini-batch sizes, as validated in our experiments on benchmark datasets.

2.2 Related Work

Batch Normalization (BN). BN [32] was introduced to address the internal covariate shift (ICS) problem by performing normalization along the batch dimension. For a layer with d-dimensional input $\mathbf{x} = (x^{(1)}, x^{(2)}, ..., x^{(d)})$ in a mini-batch $\mathcal{X}_B$ with

size m, BN normalizes each dimension of the input samples as:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mu_{\mathcal{B}}^k}{\sqrt{{\sigma_{\mathcal{B}}^k}^2 + \epsilon}} \tag{2.1}$$

where $\mu_{\mathcal{B}}^k = \frac{1}{m}\sum_{i=1}^{m} x_i^{(k)}$, $\sigma_{\mathcal{B}}^k = \frac{1}{m}\sum_{i=1}^{N}(x_i^{(k)} - \mu_{\mathcal{B}}^k)^2$, and ϵ is a small positive constant. And for inference step, the mean and variance of mini-batch are replaced with that of population, often estimated by moving average. It can stabilize the latent feature distributions in deep neural network (DNN) training, and significantly improve the training speed and model generalization ability in many applications [22, 105, 24, 109, 46, 53]. Unfortunately, BN does not perform well in the case of small batch size because of the variation of batch statistics and the inconsistency between its training and inference stages [31, 93].

Some variants of BN have been proposed to improve BN. For example, Peng *et al* [67] proposed to perform the synchronized computation of BN statistics across GPUs (synchronized BN) for more accurate statistics estimation, which is equivalent to enlarging the training batch size. Cross-Iteration BN (CBN) [98] uses the statistics in the recent several iterations to enlarge the virtual batch size and compensate for the network weight changes. A technique based on Taylor polynomials was developed in CBN to estimate batch statistics. Batch Re-normalization [31], Momentum BN [101] and Moving Average BN [96] use the moving average of batch statistics for training, but they need elaborated hyper-parameter settings. Different from synchronized BN, Ghost-BN [23] divides the mini-batch in one GPU into several small subgroups, which can be used to simulate the case of small batch size training with multiple GPUs.

Inference Methods of BN. While most of the previous works [31, 101, 96, 98] focus on reducing the noise of batch statistics during training, some methods [85, 83] have been proposed to improve the inference stage of BN. To reduce the gap between training and inference of BN, Summers *et al* [85] combined the instance statistics and the exponential moving average (EMA) of mini-batch statistics. However, one hyper-

parameter is introduced and it is not convenient to tune practice. EN [83] was also developed to combine the instance statistics and batch statistics. In order to estimate the hyper-parameter, Singh *et al*defined a non-convex auxiliary objective function for each BN layer to minimize by gradient descant. However, this method is complex to implement. Compared to [85, 83], our proposed method in this chapter does not need to tune any hyper-parameters and it is very easy to implement. Meanwhile, EMA can be viewed as a special case of our method.

Other Normalization Methods. To reduce the impact of batch size in training, some methods have been proposed to perform normalization along other dimensions. For example, layer normalization (LN) [44] normalizes all activations or feature maps along feature dimension; instance normalization (IN) [88] performs normalization for each feature map of each sample individually, and group normalization (GN) [93] normalizes feature maps for each input sample in a divided group. These normalization methods demonstrate good performance for specific applications (*e.g*, LN for RNN, IN for style transfer, and GN for object detection); however, they do not outperform BN with a proper training batch size in general. Some methods have also been proposed to combine BN with other normalizations. For example, switchable normalization [55, 78, 54] combines BN with LN and IN with learnable weights; batch group normalization [85] combines BN and GN by performing normalization along the batch dimension in a divided feature map group; batch-channel normalization (BCN) [69] integrates BN with the momentum of batch statistics and channel-based normalization. Dynamic normalization (DN) [57] was proposed to combine IN, LN, GN and BN in a unified formulation. Exemplar normalization [111] was introduced to learn different data-dependent normalizations for different image samples. Though the above-mentioned methods can improve BN, they do not address the inconsistency problem of BN in training and inference.

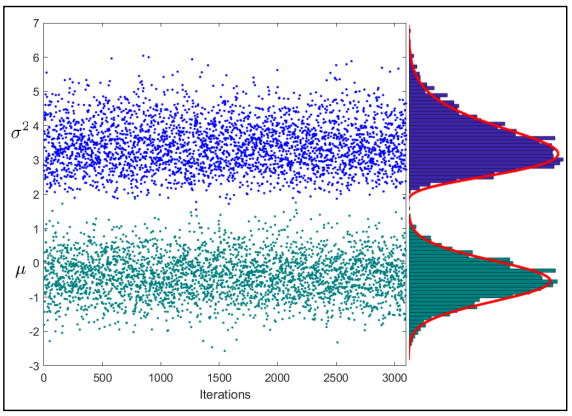


Figure 2.1: The mean and variance of mini-batches vs. iterations. The mean (green points) and variance (blue points) are from one channel of the first BN layer of ResNet18 in the last epoch when training with batch size 16 on CIFAR100. The histograms of batch mean and variance are plotted on the right, which can be well fitted by Gaussian distribution and Chi-square distribution, respectively.

2.3 The Regularization Nature of BN

2.3.1 Noise Generation of BN

Several previous works [93, 105] have indicated that the BN layer can enhance the generalization capability of DNNs experimentally; however, little work has been done on the theoretical analysis about why BN has this capability. The only work we can find is [87], where Teye *et al*tried to give a theoretical illustration for the generalization gain of BN from a Bayesian perspective with some additional priors. In the work [56], Luo *et al*presented a regularization term based on the result of [87]. Shekhovtsov *et al* [79] gave an interpretation of BN from the perspective of noise generation. However, it is assumed that the input activations follow strictly i.i.d. Gaussian distribution and there is no further theoretical analysis on how the noise affects the training process. In this section, we theoretically show that BN can be modeled as a process of noise generation.

Let's first assume that one activation input in a layer follows the Gaussian distribu-

tion $\mathcal{N}(x|\mu, \sigma^2)$, where μ and σ^2 can be simply estimated by population mean $\mu_{\mathcal{P}}$ and variance $\sigma_{\mathcal{P}}$ of training data. This assumption can be extended to more general cases other than Gaussian distribution, as we will explain later. In stochastic optimization [7, 6], randomly choosing a mini-batch of training samples can be considered as a sample drawing process, where all samples x_i in a mini-batch $\mathcal{X}_b = \{x_i\}_{i=1}^m$ are i.i.d., and follows $\mathcal{N}(x|\mu, \sigma^2)$. For the mini-batch $\mathcal{X}_b$ with mean $\mu_B = \frac{1}{m}\sum_{i=1}^m x_i$ and variance $\sigma_B^2 = \frac{1}{m}\sum_{i=1}^m (x_i - \mu_B)^2$, we can define two random variables ξ_μ and ξ_σ as follows [14]:

$$\xi_\mu = \frac{\mu - \mu_B}{\sigma} \sim \mathcal{N}(0, \frac{1}{m}), \quad \xi_\sigma = \frac{\sigma_B^2}{\sigma^2} \sim \frac{1}{m}\chi^2(m-1) \tag{2.2}$$

where χ^2 denotes the Chi-squared distribution and ξ_σ follows a Scaled-Chi-squared distribution with $E(\xi_\sigma) = \frac{m-1}{m}$ and $Var(\xi_\sigma) = \frac{2(m-1)}{m^2}$.

In Fig. 2.1 we plot the means and variances of mini-batches computed at the first BN layer of ResNet18 in the last training epoch when training with batch size 16 on CIFAR100 dataset. One can see that these means and variances are distributed like biased random noise. Specifically, the histogram of mean values can be well modeled as a Gaussian distribution, while the histogram of variances can be well modeled as a scaled Chi-Square distribution. By neglecting the small constant ϵ in Eq.(2.1), the BN in the training process can be rewritten as

$$\widehat{x} = \frac{x - \mu_B}{\sigma_B} = \frac{x - \mu + (\mu - \mu_B)}{\sigma\frac{\sigma_B}{\sigma}} = \frac{\frac{x-\mu}{\sigma} + \xi_\mu}{\sqrt{\xi_\sigma}} = \frac{\widetilde{x} + \xi_\mu}{\sqrt{\xi_\sigma}} \tag{2.3}$$

where $\widetilde{x} = \frac{x-\mu}{\sigma}$ is the population normalized formula.

From Eq.(2.3), we can see that BN actually first adds Gaussian noise ξ_μ (**additive noise**) to the sample after population normalization, and then multiplies with a Scaled-Inverse-Chi noise $\frac{1}{\sqrt{\xi_\sigma}}$ (**multiplicative noise**). That is, training with BN is actually introducing a mixture of additive and multiplicative noise. With the introduced additive noise ξ_μ and multiplicative noise ξ_σ, the output variable $\widehat{x}$ follows $Nt(\widetilde{x}, m-1)$, which is a noncentral t-distribution [45], and its probability density

function is very complex. Fortunately, we can still get the mean and variance of $\widehat{x}$ as follows:

$$E[\widehat{x}] = \widetilde{x}\sqrt{\frac{m-1}{2}}\frac{\Gamma((m-2)/2)}{\Gamma((m-1)/2)}, \quad Var[\widehat{x}] = \frac{1}{m}(\frac{m-1}{m-3}(1+\widetilde{x}^2) - E[\widehat{x}]^2) \tag{2.4}$$

When m is very large, $E[\widehat{x}] \approx \widetilde{x}$ and $Var[\widehat{x}] \approx 0$. However, when m is small, the noise generated by BN depends on not only the statistics of entire training data $\mathcal{X}$ (*e.g*, mean μ and variance σ^2) but also the batch size m.

With the above analyses, we can partition BN into three parts: a normalizer part (*i.e*, $\widetilde{x} = \frac{x-\mu}{\sigma}$); a noise generator part (*i.e*, $\widehat{x} = \frac{\widetilde{x}+\xi_\mu}{\sqrt{\xi_\sigma}}$) ; and an affine transformation part (*i.e*, $y = \gamma\widehat{x} + \beta$). In the training stage, only the noise generator part is related to batch size m. In the inference stage, the batch mean and variance are replaced with population mean and variance, and thus BN only has the normalizer part and the affine transformation part. It should be emphasized that μ and σ are unknown in training, and they also vary during the training process. At the end of training and when statistics for activations of all samples are stable, they can be viewed as fixed.

Now we have shown that the BN process actually introduces noises ξ_μ and ξ_σ into the BN layer in the training process. When the batch size is small, the variances of both additive noise ξ_μ and multiplicative noise ξ_σ become relatively large, making the training process less stable. In our above derivation, it is assumed that the activation input follows the Gaussian distribution. However, in practical applications, the activations may not follow exactly the Gaussian distribution. Fortunately, we have the following theorem.

Theorem 1: *Suppose samples x_i for $i = 1, 2, ..., m$ are i.i.d. with $E[x] = \mu$ and $Var[x] = \sigma^2$, ξ_μ and ξ_σ are defined in Eq.(2.2), we have:*

$$\lim_{m\to\infty} p(\xi_\mu) \to \mathcal{N}(\xi_\mu|0, \frac{1}{m}), \quad \lim_{m\to\infty} p(\xi_\sigma) \to \frac{1}{m}\chi^2(\xi_\sigma|m-1).$$

Proof. From the classical central limit theorem, we have

$$\lim_{m\to\infty} p(\sum_{i=1}^{m} x_i) = \mathcal{N}(\sum_{i=1}^{m} x_i | m\mu, m\sigma^2),$$

that is

$$\lim_{m\to\infty} p(m\mu_B) = \mathcal{N}(m\mu_B | m\mu, m\sigma^2).$$

Therefore $\lim_{m\to\infty} p(\mu_B) = \mathcal{N}(\mu_B|\mu, \frac{\sigma^2}{m})$, and $\xi_\mu = \frac{\mu-\mu_B}{\sigma}$ is a linear function of μ_B, Then we can obtain that

$$\lim_{m\to\infty} p(\xi_\mu) = \mathcal{N}(\xi_\mu|0, \frac{1}{m}).$$

For χ^2 distribution, it has this property:

$$\lim_{m\to\infty} \frac{\chi^2(m-1)}{m} = \lim_{m\to\infty} \frac{\chi^2(m-1)}{m-1} = \mathcal{N}(1, \frac{2}{m}).$$

And we have $\lim_{m\to\infty} \mu_B = \mu$, and then for ξ_σ we can also use the central limit theorem to get:

$$\lim_{m\to\infty} p(\xi_\sigma) = \lim_{m\to\infty} p(\frac{1}{m\sigma^2}\sum_{i=1}^{m}(x_i-\mu_B)^2) = \lim_{m\to\infty} p(\sum_{i=1}^{m}(\frac{x_i-\mu}{\sigma})^2) = \mathcal{N}(\xi_\sigma|1, \frac{\kappa}{m}),$$

where κ is the kurtosis of x. When m is a very large number, both $\frac{\kappa}{m}$ and $\frac{2}{m}$ are close to zeros. Therefore, in this case the distribution of ξ_σ can be viewed as $\frac{1}{m}\chi^2(m-1)$.

The proof is completed. ■

In particular, when m is larger than 5, ξ_μ and ξ_σ nearly meet the distribution assumptions. As for the i.i.d. assumption on the activations of samples in a mini-batch, it generally holds because the samples are randomly drawn from the pool in training.

2.3.2 Explicit Regularization Formulation

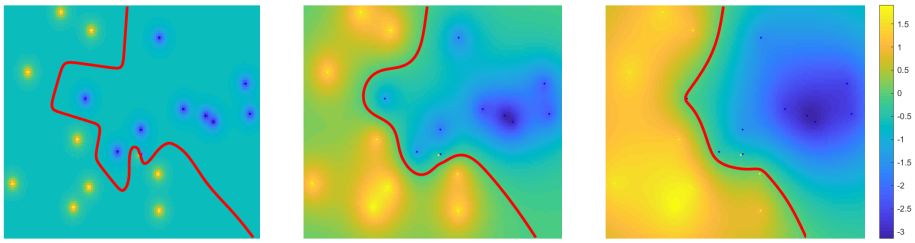
It has been verified in previous works [93, 105] that introducing a certain amount of noise into training data can increase the generalization capability of the neural network. However, there lacks a solid theoretical analysis on how this noise injection operation works. In this section, we aim to give a clear formulation.

Additive Noise: We first take additive noise ξ_μ into consideration. Let $l(t, f(x))$ (abbreviate as $l(x)$ in the following development) denote the loss w.r.t. one activation input x, where t is the target, $f(\cdot)$ represents the network and $l(\cdot)$ is the loss function. When additive noise ξ_μ is added to the activation, the loss becomes $l(x + \xi_\mu)$. By Taylor expansion [4], we have

$$E_{\xi_\mu}[l(x + \xi_\mu)] = l(x) + R^{add}(x), \quad R^{add}(x) = \sum_{n=1}^{\infty} \frac{E[\xi_\mu^n]}{n!} \frac{d^n l(x)}{dx^n}. \tag{2.5}$$

where $E(\cdot)$ is the expectation and R^{add} is the additive noise residual term, which is related to the n-th order derivative of loss function w.r.t. activation input and the n-th order moment of noise distribution.

According to [3], by considering only the major term in R^{add}, it can be shown that $R^{add}(x) \approx \frac{E[\xi_\mu^2]}{2} \left|\frac{\partial f(x)}{\partial x}\right|^2$ for mean square-error loss; and $R^{add}(x) \approx \frac{E[\xi_\mu^2]}{2} \frac{f(x)^2 - 2tf(x) + t}{f(x)^2(1-f(x))^2} \left|\frac{\partial f(x)}{\partial x}\right|^2$ for cross-entropy loss. This indicates that R^{add} regularizes the smoothness of the network function, while the strength of smoothness is mainly controlled by the second order moment of the distribution of noise ξ_μ ($i.e$, $\frac{1}{m}$), which is only related to training batch size m. In Fig. 2.2, we illustrate the influence of additive noise on learning a classification hyperplane with different noise levels. The yellow points and blue points represent samples from two classes, and the Gaussian noise is added to the samples for data augmentation. By increasing the noise level σ to a proper level ($e.g$, $\sigma = 0.5$), the learned classification hyperplane becomes smoother and thus has better generalization capability. However, a too big noise level ($e.g$, $\sigma = 1$) will over-smooth



$\sigma = 0.1$ $\sigma = 0.5$ $\sigma = 1$

Figure 2.2: The influence of noise injection on classification hyperplane (shown as red curve) learning with different noise levels. The yellow points and blue points represent samples from two classes, and the Gaussian noise with variance σ^2 is added to the samples for data augmentation. We can see that by increasing the noise level σ to a proper level (*e.g*, $\sigma = 0.5$), the learned classification hyperplane becomes smoother and thus has better generalization capability. However, a too big noise level (*e.g*, $\sigma = 1$) will over-smooth the classification boundary and decrease the discrimination ability.

the classification boundary and decrease the discrimination ability.

Multiplicative Noise: For multiplicative noise ξ_σ, we can use a simple logarithmic transformation $l(\frac{x}{\sqrt{\xi_\sigma}}) = l(e^{\log|x| - \frac{1}{2}\log\xi_\sigma}\text{sign}(x))$ to transform it into the form of additive noise. Then according to our analyses of additive noise in Eq. (2.5), we have:

$$E_{\xi_\sigma}[l(\frac{x}{\sqrt{\xi_\sigma}})] = l(x) + R^{mul}(x), \quad R^{mul}(x) = \sum_{n=1}^{\infty}\sum_{k=1}^{d} \mathrm{I}(x \neq 0)\frac{E[\log^n \xi_\sigma]}{(-2)^n n!}\frac{d^n l(x)}{(d\log|x|)^n}. \tag{2.6}$$

where $\mathrm{I}(x \neq 0)$ is an indicator function and $R^{mul}(x)$ is the residual term of Taylor expansion for multiplicative noise. Similar to the residual term of additive noise $R^{add}(x)$, the major term of $R^{mul}(x)$ can also be viewed as a regularizer to $\left|\frac{\partial f(x)}{\partial \log|x|}\right|^2$, which controls the smoothness of network on log-scale, and $E[\log^2 \xi_\sigma]$ is related to the strength of the regularizer.

Compound Noise: In Section 2.3.1 we have shown that BN will introduce both additive noise and multiplicative noise into the normalized activation input, *i.e*, $\widehat{x} = \frac{\widetilde{x}+\xi_\mu}{\sqrt{\xi_\sigma}}$. In the following theorem, we present the joint residual formulation for the compound of additive noise and multiplicative noise.

Theorem 2: *If the infinite derivative of $l(x)$ exists for any x, given two random variables ξ_μ and ξ_σ (> 0), then we have the Taylor expansion for $l(\frac{x+\xi_\mu}{\sqrt{\xi_\sigma}})$:*

$$E_{\xi_\mu,\xi_\sigma}[l(\frac{x+\xi_\mu}{\sqrt{\xi_\sigma}})] = l(x) + R^{add}(x) + R^{mul}(x) + R(x),\ R(x) = \sum_{n=1}^{\infty} \frac{E[{\xi_\mu}^n]}{n!} \frac{d^n R^{mul}(x)}{dx^n} \tag{2.7}$$

where $R^{add}(x)$ and $R^{mul}(x)$ are defined in Eq.(2.5) and (2.6), respectively.

Proof.

$$\begin{aligned}
E_{\xi_\mu,\xi_\sigma}[l(\frac{x+\xi_\mu}{\sqrt{\xi_\sigma}})] &= E_{\xi_\mu,\xi_\sigma}[l(\frac{x}{\sqrt{\xi_\sigma}} + \frac{\xi_\mu}{\sqrt{\xi_\sigma}})] \\
&= E_{\xi_\sigma}[l(\frac{x}{\sqrt{\xi_\sigma}})] + E_{\xi_\mu,\xi_\sigma}[\sum_{n=1}^{\infty} \frac{(\frac{\xi_\mu}{\sqrt{\xi_\sigma}})^n}{n!} \frac{d^n l(\frac{x}{\sqrt{\xi_\sigma}})}{d(\frac{x}{\sqrt{\xi_\sigma}})^n}] \\
&= l(x) + R^{mul}(x) + E_{\xi_\mu,\xi_\sigma}[\sum_{n=1}^{\infty} \frac{{\xi_\mu}^n}{n!} \frac{d^n l(\frac{x}{\sqrt{\xi_\sigma}})}{dx^n}] \\
&= l(x) + R^{mul}(x) + E_{\xi_\mu}[\sum_{n=1}^{\infty} \frac{{\xi_\mu}^n}{n!} \frac{d^n E_{\xi_\sigma}[l(\frac{x}{\sqrt{\xi_\sigma}})]}{dx^n}] \\
&= l(x) + R^{mul}(x) + E_{\xi_\mu}[\sum_{n=1}^{\infty} \frac{{\xi_\mu}^n}{n!} \frac{d^n (l(x) + R^{mul}(x))}{dx^n}] \\
&= l(x) + R^{mul}(x) + E_{\xi_\mu}[\sum_{n=1}^{\infty} \frac{{\xi_\mu}^n}{n!} \frac{d^n (l(x))}{dx^n}] + E_{\xi_\mu}[\sum_{n=1}^{\infty} \frac{{\xi_\mu}^n}{n!} \frac{d^n (R^{mul}(x))}{dx^n}] \\
&= l(x) + R^{mul}(x) + R^{add}(x) + R(x).
\end{aligned} \tag{2.8}$$

The proof is completed. ■

From Theorem 2, we can see the Taylor expansion residual can be divided into three parts: a residual term $R^{add}(x)$ for additive noise, a residual term $R^{mul}(x)$ for multiplicative noise and a cross residual term $R(x)$. When the noise level is small, $R(x)$ can be ignored. Particularly, the distributions of ξ_μ and ξ_σ are give in Eq.(2.2) so that the regularizer strength parameters $E[\xi_\mu^2]$ and $E[\log^2 \xi_\sigma]$ can be easily calculated, which are only determined by training batch size m. The noise is injected into the normalized data $\widetilde{x} = \frac{x-\mu}{\sigma}$. If the introduced noise by BN is strong (*e.g*, when batch size

is small), the training forward propagation through the DNN may accumulate and amplify noise, which leads to undesirable model performance. Therefore, it is crucial to choose a suitable batch size for training to make BN keep a proper noise level and ensure a favorable regularization function. However, in some situations of limited memory and computing resources, we can only use a small batch size for training. It is hence important to find an approach to control the noise level of BN with small batch size, which will be investigated in the next section.

2.4 Momentum Batch Normalization

As proved in Section 2.3, the batch size m directly controls the strength of the regularizer in BN so that BN is sensitive to batch size. In most previous literature [93, 31], the batch size m is set around 64 by experience. However, in some applications, the batch size may not be set big enough due to the limited memory and large size of the input. How to stably train a network with a small batch size in BN remains an open problem. Owe to our theoretical analyses in Section 2.3, we propose a simple solution to alleviate this problem by introducing a parameter to control the strength of the regularizer in BN. Specifically, we replace the batch means and variances in BN by their momentum or moving average:

$$\mu_M^{(n)} = \lambda \mu_M^{(n-1)} + (1-\lambda)\mu_B, \quad \left(\sigma_M^{(n)}\right)^2 = \lambda \left(\sigma_M^{(n-1)}\right)^2 + (1-\lambda)\sigma_B^2, \tag{2.9}$$

where λ is the momentum parameter to control the regularizer strength and n refers to the number of batches (or iterations). We name our new BN method as Momentum Batch Normalization (MBN), which can make the noise level generated by using a small batch size almost the same as that by using a large batch size when the training stage ends.

2.4.1 Noise Estimation

At the end of the training process, all statistics of variables tend to be converged. According to Eq. (2.9), it can be derived that

$$\mu_M^{(n)} = (1-\lambda)\sum_{i=1}^{n}\lambda^{n-i}\mu_B, \quad \left(\sigma_M^{(n)}\right)^2 = (1-\lambda)\sum_{i=1}^{n}\lambda^{n-i}\sigma_B^2 \tag{2.10}$$

When n is very large, let μ_M and σ_M denote the final momentum mean and variance, we can derive that

$$\xi_\mu = \frac{\mu - \mu_M}{\sigma} \sim \mathcal{N}(0, \frac{1-\lambda}{m}) \tag{2.11}$$

$\xi_\sigma = \frac{\sigma_M^2}{\sigma^2}$ follows Generalized-Chi-Squared distribution, whose expectation is $E[\xi_\sigma] = \frac{m-1}{m}$ and variance is $Var[\xi_\sigma] = \frac{1-\lambda}{1+\lambda}\frac{2(m-1)}{m^2}$.

We can see that the variances of ξ_μ and ξ_σ approach to zero when λ is close to 1, MBN degenerates into standard BN when λ is zero. This implies that the noise level can be controlled by momentum parameter λ. A larger value λ will weaken the regularization function of MBN, and vice versa. Even when the batch size m is very small, we are still able to reduce the noise level by adjusting λ. This is an important advantage of MBN over conventional BN. For instance, if we want to make MBN with batch size 4 have a similar noise level with batch size 16, the momentum parameter λ can be set as 3/4 to make their variances of ξ_μ similar, ($\frac{1-\frac{3}{4}}{4} = \frac{1}{16}$), and the multiplicative noise ξ_σ will also be reduced.

2.4.2 Momentum Parameter Setting

Dynamic Momentum Parameter for Training: Since the momentum parameter λ controls the final noise level, we need to set a proper momentum parameter to endow the network with a certain generalization ability. Please note that our noise analysis in Section 2.4.1 holds only when network statistics are stable at the end of training.

At the beginning of training, we cannot directly use the moving average of batch mean and variance, because the population means and variance also changes significantly. Therefore, we hope that at the beginning of training the normalization is close to BN, while at the end of it tends to be MBN. To this end, we propose a dynamic momentum parameter as follows:

$$\lambda^{(t)} = \rho^{\frac{T}{T-1}\max(T-t,0)} - \rho^T, \qquad \rho = \min(\frac{m}{m_0}, 1)^{\frac{1}{T}} \tag{2.12}$$

where t refers to the t-th iteration epoch, T is the number of the total epochs, m is the actual batch size and m_0 is a large enough batch size ($e.g$, 64).

We use the same momentum parameter within one epoch. $\lambda^{(t)}$ starts from zero. When $\frac{m}{m_0}$ is small, $\lambda^{(t)}$ tends to be a number close to 1 at the end of the training. If m is equal to or larger than m_0, $\lambda^{(t)}$ is always equal to zero, and then MBN degenerates into BN. The dynamic setting of momentum parameter ¦Ë ensures that at the beginning of the training process, the normalization is similar to standard BN, while at the end of the training the normalization approaches MBN with a noise level similar to that of BN with batch size m_0.

Momentum Parameter for Inference: For the inference step, we also need to set a momentum parameter. For the clarity of description, here we use τ to denote this momentum parameter to differentiate it from the momentum parameter λ in the training stage. One can straightforwardly set τ as a constant, $e.g\tau = 0.9$, which is independent of batch size. However, this setting is not very reasonable because it cannot reflect the final noise level when training is ended, which is related to batch size m. Therefore, we should set τ to be adaptive to batch size m. Denote by τ_0 the desired momentum value for an ideal batch size m_0, to make the inference momentum have the same influence on the last sample, we take $\tau^{\frac{N}{m}}$ as a reference to determine

Algorithm 1: Momentum Batch Normalization (MBN)

Input: Values of x over a training mini-batch $\mathcal{X}_b$; parameters γ, β; current training moving mean μ and variance σ^2; current inference moving mean μ_{inf} and variance σ^2_{inf}; momentum parameters λ for training and τ for inference.

Output: $\{y_i = \text{MBN}(x_i)\}$; updated μ and σ^2; updated μ_{inf} and σ^2_{inf}

$\mathbf{G}^t = \nabla_{\mathbf{W}^t}\mathcal{L}$

if *Training* **then**

$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$

$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_b)^2$

$\mu \leftarrow \lambda\mu + (1-\lambda)\mu_B$

$\sigma^2 \leftarrow \lambda\sigma^2 + (1-\lambda)\sigma_B^2$

$\widehat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$

$y_i = \gamma\widehat{x}_i + \beta$

$\mu_{inf} \leftarrow \tau\mu_{inf} + (1-\tau)\mu_B$

$\sigma^2_{inf} \leftarrow \tau\sigma^2_{inf} + (1-\tau)\sigma_B^2$

else

$y_i = \gamma\frac{x_i - \mu_{inf}}{\sqrt{\sigma^2_{inf} + \epsilon}} + \beta$

end

the value of τ for batch size m as follows:

$$\tau^{\frac{N}{m}} = \tau_0^{\frac{N}{m_0}} \quad \Rightarrow \quad \tau = \tau_0^{\frac{m}{m_0}} \tag{2.13}$$

where N is the number of samples, m_0 is an ideal batch size and τ_0 is its corresponding momentum parameter. In most of our experiments, we set $m_0 = 64$ and $\tau_0 = 0.9$ for the inference step. One can see that when the training batch size m is small, a larger inference momentum parameter τ will be used, and consequently, the noise in momentum mean and variance will be suppressed.

2.4.3 Algorithm

The back-propagation (BP) process of MBN is similar to that of traditional BN. During training, the gradients of loss w.r.t. to activations and model parameters are

calculated and back-propagated. The formulas of BP are listed as follows:

$$\begin{aligned}
&\frac{\partial L}{\partial \widehat{x}_i} = \frac{\partial L}{\partial \widehat{y}_i}\gamma, \quad \frac{\partial L}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial L}{\partial \widehat{y}_i}\widehat{x}_i, \quad \frac{\partial L}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial L}{\partial \widehat{y}_i} \\
&\frac{\partial L}{\partial {\sigma_B}^2} = \sum_{i=1}^{m} \frac{\partial L}{\partial \widehat{x}_i}(\widehat{x}_i - \mu_M)\frac{-1}{2}({\sigma_M}^2 + \epsilon)^{-\frac{3}{2}}(1-\lambda) \\
&\frac{\partial L}{\partial \mu_B} = (\sum_{i=1}^{m} \frac{\partial L}{\partial \widehat{x}_i}\frac{\lambda - 1}{\sqrt{{\sigma_M}^2 + \epsilon}}) + \frac{\partial L}{\partial {\sigma_B}^2}\frac{\sum_{i=1}^{m} -2(x_i - \mu_B)}{m} \\
&\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial \widehat{x}_i}\frac{1}{\sqrt{{\sigma_M}^2 + \epsilon}} + \frac{\partial L}{\partial {\sigma_B}^2}\frac{2(x_i - \mu_B)}{m} + \frac{\partial L}{\mu_B}\frac{1}{m}
\end{aligned} \tag{2.14}$$

Since the current moving averages of μ_M and ${\sigma_M}^2$ are related to the mean μ_B and variance ${\sigma_B}^2$ of the current mini-batch, they also contribute to the gradient, while the previous μ_M and ${\sigma_M}^2$ can be viewed as two constants for the current mini-batch. The training and inference of MBN are summarized in **Algorithm 1**.

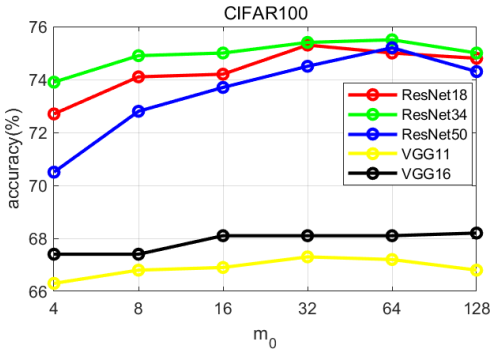
2.5 Experimental Results

2.5.1 Datasets and Experimental Setting

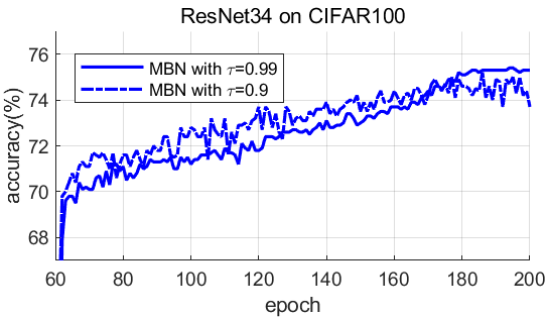
To evaluate MBN, we apply it to image classification tasks and conduct experiments on CIFAR10, CIFAR100 [39] and Mini-ImageNet100 datasets [89].

Datasets. CIFAR10 consists of 50k training images from 10 classes, while CIFAR100 consists of 50k training and 10k testing images from 100 classes. The resolution of sample images in CIFAR10/100 is 32×32. Mini-ImageNet is a subset of the well-known ImageNet dataset. It consists of 100 classes with 600 images each class, and the image resolution is 84×84. We use the first 500 images from each class as training data, and the rest 100 images for testing, *i.e*, 50k images for training and 10k images for testing.

Experimental Setting. We use SGD with momentum 0.9 and weight decay 0.0001,



(a) Recognition rates of MBN with different m_0 on CIFAR100 by using different DNNs, including ResNet18, ResNet34, ResNet50, VGG11 and VGG16.



(b) Testing accuracy curves on CIFAR100 of BN and MBN with different τ for inference and training batch size 2 per GPU.

Figure 2.3: Parameters tuning of MBN.

employ standard data augmentation and preprocessing techniques, and decreases the learning rate when learning plateaus occur. The model is trained for 200 epochs and 100 epochs for CIFAR and Mini-ImageNet-100, respectively. We start with a learning rate of $0.1 \cdot \frac{m}{64}$ both for CIFAR10 and CIFAR100 and $0.1 \cdot \frac{m}{128}$ for Mini-ImageNet-100, and divide it by 10 for every 60 epochs and 30 epochs, respectively. We mainly employ ResNet [22] as our backbone network and use similar experimental settings to the original ResNet paper. All the experiments are conducted on the Pytorch1.0 framework.

2.5.2 Parameters Setting

There are two hyper-parameters in our proposed MBN, m_0 and τ_0, which are used to determine the momentum parameters λ and τ for training and inference.

The Setting for m_0: We first fix τ_0 (*e.g*, 0.9) to find a proper m_0. We adopt ResNet18 as the backbone and train it with batch size 8 and 16 on 4 GPUs, *i.e*, batch size 2 and 4 per GPU, to test the classification accuracy with different m_0. Particularly, we let m_0 be 4, 8, 16, 32, 64, 128 in MBN. Fig. 2.3(a) shows the accuracy curves on CIFAR100. We can see that if m_0 is too small (*e.g*, 4), MBN

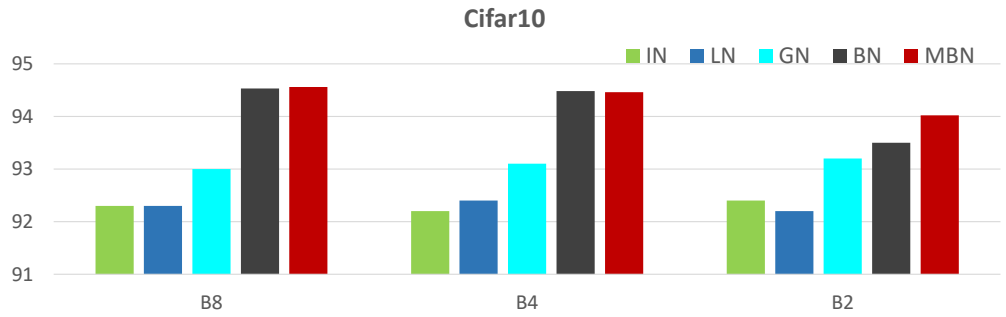


Figure 2.4: Testing accuracy on CIFAR10 of ResNet18 with training batch size (BS) 8, 4, and 2 per GPU.

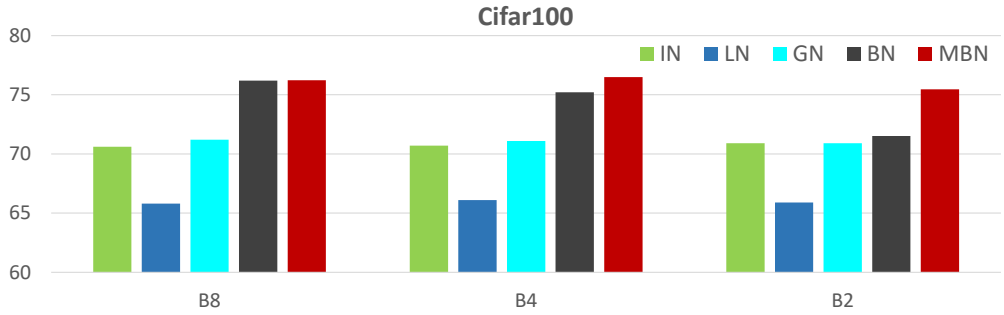


Figure 2.5: Testing accuracy on CIFAR100 of ResNet18 with training batch size (BS) 8, 4, and 2 per GPU.

will be close to BN, and the performance is not very good. The accuracies of MBN are very close for m_0 from 16 to 128, which shows that MBN is not very sensitive to parameter m_0. Considering that if m_0 is too large (*e.g*, 128), the momentum parameter λ may change too quickly so that the training may not converge, we set it to 32 in all the experiments.

The Setting for τ_0: We then fix m_0 as 32 and find a proper τ_0 based on Eq.(2.13). Fig. 2.3(b) shows the testing accuracy curves for for MBN with different values of τ. $\tau = 0.9$ is the original BN setting, and $\tau = 0.99$ is our setting based on Eq.(2.13) with $\tau_0 = 0.85$. We can see that when τ is small the testing accuracy curves of both BN and MBN have big fluctuations; while τ is large, the accuracy curves become more stable and the final accuracies can be improved. We set $\tau_0 = 0.85$ in the following experiments.

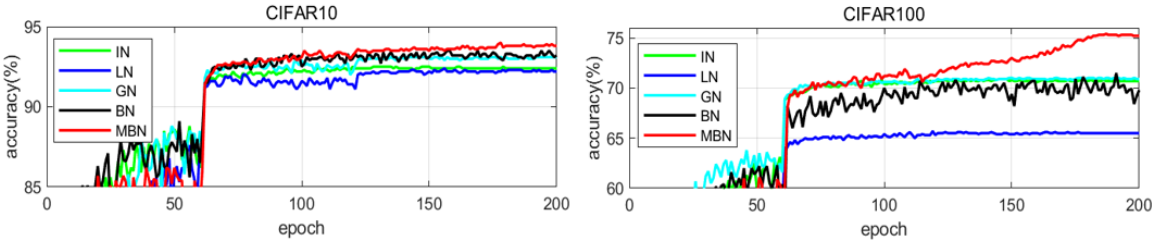


Figure 2.6: Comparison of accuracy curves for different normalization methods with a batch size of 2 per GPU. We show the test accuracies vs. the epoches on CIFAR10 (left) and CIFAR100 (right). The ResNet18 is used.

2.5.3 Results on CIFAR10/100

We first conduct experiments on the CIFAR10 and CIFAR100 datasets [39]. We first use ResNet18 as the backbone network to evaluate MBN with different batch sizes and then test the performance of MBN with more networks.

Training with Different Batch Size: To testify whether MBN is more robust than BN with a small batch size, we train Resnet18 on CIFAR10 and CIFAR100 by setting the batch size m as 8, 4, 2 per GPU, respectively. We also compare the behaviors of other normalization methods, including IN [88], LN [44] and GN [93], by replacing the BN layer with them. For GN, we use 32 groups as set in [93]. And we set $T = 180$ in Eq.(2.12) for MBN.

Fig. 2.4 and Fig. 2.5 show the results for different normalization methods on CIFAR10 and CIFAR100, respectively. We can see that on both CIFAR10 and CIFAR100 when the batch size is relatively large (*e.g*, 8), the accuracy of MBN is similar to BN. This is in accordance with our theoretical analysis in Sections 2.3 and 2.4. However, when training batch size becomes small (*e.g*, 2), the accuracy of BN drops largely, while the accuracy of MBN decreases slightly. This shows that MBN is more robust than BN for training with a small batch size. Meanwhile, MBN works much better than IN, LN, and GN.

Fig. 2.6 shows the training and testing accuracy curves vs. epoch of ResNet18 with

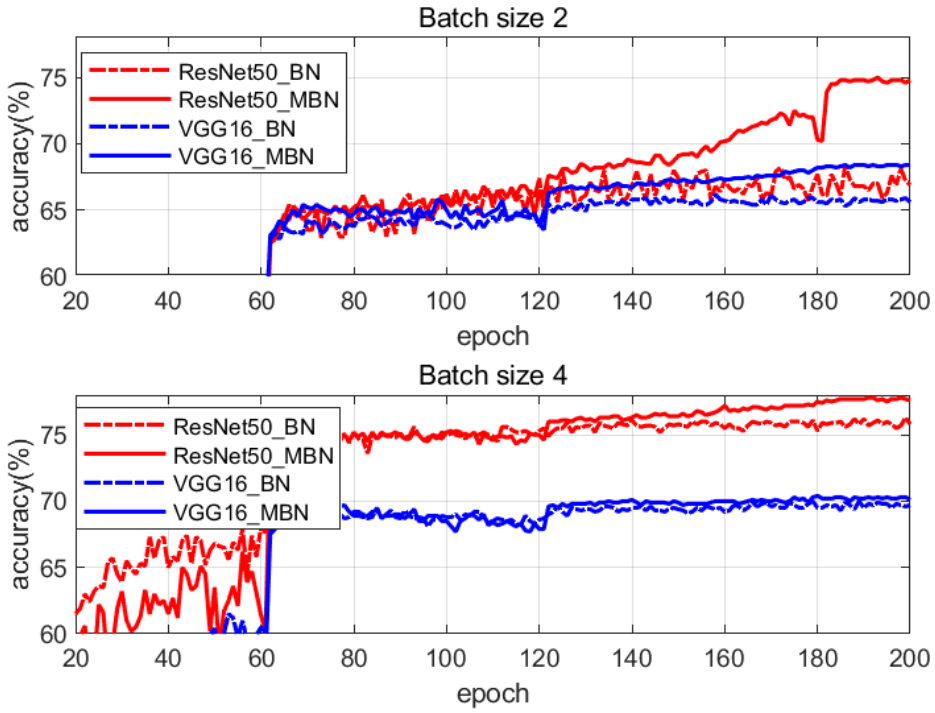


Figure 2.7: Testing accuracy curves on CIFAR100 for different network architectures with training batch size 2 (top) and 4 (bottom) per GPU.

batch size 2. We can see that at the last stage of training when all statistics become stable, MBN can still achieve a certain performance gain. This is because with MBN the momentum mean and variance approach to the population mean and variance, and hence the noise becomes small. Consequently, MBN can still keep improving though other methods are saturated.

On More Network Architectures: We further test MBN with different network architectures, including ResNet34, ResNet50, VGG11, and VGG16, by using batch size 2 per GPU on CIFAR100. Fig. 2.7 shows the training and testing accuracy curves vs. epochs, and Fig. 2.8 shows the final testing accuracies. We can have the following observations. First, on all four networks, MBN always outperforms BN. Second, under such a small batch size, the accuracy of deeper network ResNet50 can be lower than its shallower counterpart ResNet34. That is because the deeper networks have more BN layers, and each BN layer would introduce relatively large

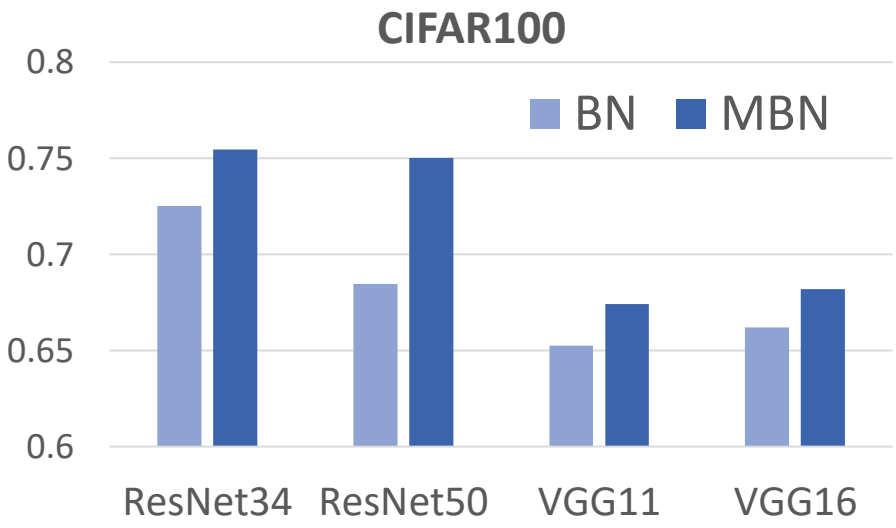


Figure 2.8: Testing accuracy on CIFAR100 for different network architectures with training batch size 2 per GPU.

noise when batch size is small. The noise is accumulated so that the benefit of more layers can be diluted by the accumulated noise. However, with MBN the performance drop from ResNet50 to ResNet34 is very minor, where the drop by BN is significant. This again validates that MBN can suppress the noise effectively in training.

2.5.4 Results on Mini-ImageNet-100

On Small Batch Size: On Mini-imageNet, we use ResNet50 as our backbone network. The input size is the same as image size 84×84. The settings for MBN is the same as Section 2.5.3. Fig. 2.9 compares the testing accuracies of IN, LN, GN, BN and MBN with batch sizes 16, 8, 4 and 2 per GPU. Fig. 2.10 shows their testing accuracy curves with training batch size 2 per GPU. We can see that BN and MBN achieve better results than other normalization methods when batch size is larger than 2, while other normalization methods, such as IN, LN and GN, usually work not very well on Mini-imageNet. But the performance of BN drops significantly when batch size is 2, even worse than IN, while MBN still works well when batch size is 2.

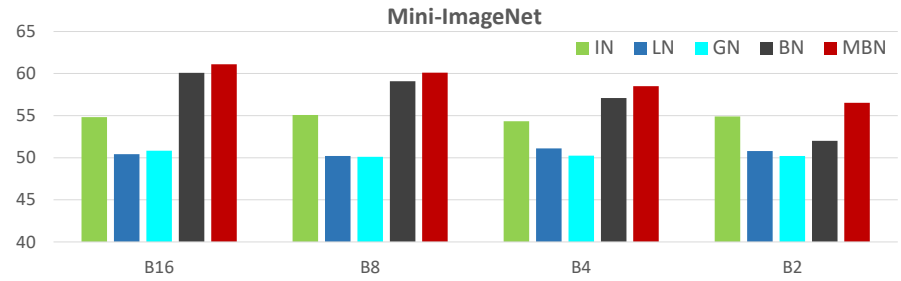


Figure 2.9: Testing accuracy on Mini-ImageNet of IN, LN, GN, BN and MBN with training batch size 16, 8, 4 and 2 per GPU.

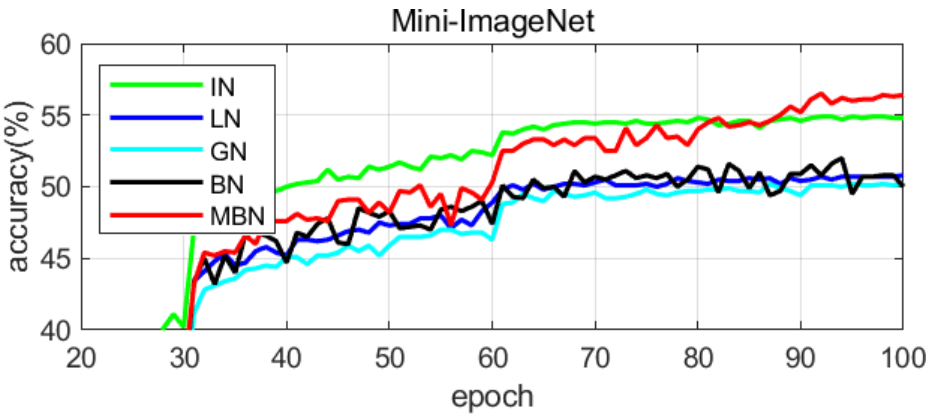


Figure 2.10: Testing accuracy curves on Mini-ImageNet of IN, LN, GN, BN and MBN with training batch size 2 per GPU.

This clearly demonstrates the effectiveness of MBN. Furthermore, we also compare MBN with BN on full ImageNet using ResNet50 with 64 GPUs and 4 batch size per GPU. It is found that find MBN outperforms BN by 2.5% in accuracy on the validation set.

Comparison with BreN: BreN [31] was also proposed to make BN work for training with a small batch size. It adopts a heuristic clipping strategy to control the influence of the current moving average on the normalizer. Though BreN and our proposed MBN have similar goals, they are very different in theoretical development and methodology design. First, the dynamic momentum setting in MBN makes it easy to analyze the noise level in the final training stage, while in BreN it is hard to know the noise level with the heuristic clipping strategy. Second, the hyper-parameters m_0

and τ_0 are very easy to be tuned and fixed in MBN (we fixed them in all our experiments on all datasets), while the hyper-parameters (clipping bounds) in BreN are very difficult to set. Although a strategy to set the clipping bound was given in [31], we found that this setting usually leads to unsatisfactory performance when the dataset or training batch size changes. We have tried various parameter settings for BreN on Mini-ImageNet when the training batch size is 2 but found that in most cases the results are even worse. So we report the best result of BreN on Mini-ImageNet with $r_{max} = 1.5$ and $d_{max} = 0.5$, which is 55.47%, lower than the performance of MBN (56.50%)

2.6 Conclusion

Batch normalization (BN) is a milestone technique in deep learning and it largely improves the effectiveness and efficiency in optimizing various deep networks. However, the working mechanism of BN is not fully revealed yet, while the performance of BN drops much when the training batch size is small because of the inaccurate batch statistics estimation. In this chapter, we first revealed that the generalization capability of BN comes from its noise generation mechanism in training, and then presented the explicit regularization formulation of BN. We consequently presented an improved version of BN, namely momentum batch normalization (MBN), which uses the moving average of sample mean and variance in a mini-batch for training. By adjusting a dynamic momentum parameter, the noise level in the estimated mean and variance can be well controlled in MBN. The experimental results demonstrated that MBN can work stably for different batch sizes. In particular, it works much better than BN and other popular normalization methods when the batch size is small.

Chapter 3

Batch Statistics Regression for Effective Inference of Batch Normalization

It is well-known that the training and inference stages of BN have certain inconsistencies. In training, BN computes the batch statistics to perform Z-score standardization, while it adopts the exponential moving average (EMA) of batch statistics in inference. Unfortunately, EMA is not accurate to approximate the batch statistics, especially when the batch size is small, because it ignores the relationship between instance statistics and batch statistics. In this chapter, we propose a new inference approach of BN, namely batch statistics regression (BSR), by using the instance statistics to predict the batch statistics with a simple linear regression model. Compared with EMA, BSR can more accurately estimate the batch statistics, making the training and inference of BN much more consistent. BSR is very easy to implement by using an online updating formulation. It only needs to store four statistics during training with the negligible cost of computation and memory. Our experiments show that it can significantly improve the inference performance of BN. In particular, it outper-

forms EMA by more than 7% in accuracy on ImageNet (backbone: ResNet50) when the training batch size is 2.

3.1 Introduction

A variety of optimization techniques, including efficient activation function [63], skip connection [22, 25], normalization [32, 88, 93], adaptive learning rate [16, 37], warm up strategy [18], gradient centralization [100], *etc*, have been developed to effectively train deep neural networks (DNNs). Among them, normalization methods are of great importance. They normalize the network activations or weights by utilizing the statistics of samples, such as mean and variance, to make the training process more stable and efficient. The most representative method that performs normalization on activations is batch normalization (BN) [32], which has been widely adopted in advanced DNN architecture design [22, 25, 24, 112]. The success of BN has been explained from several aspects, such as reducing internal covariate shift [32], smoothing optimization landscape [77] and the regularization capability [87, 105, 101].

Despite the great success, a well-known problem of BN is its inconsistency between training and inference. Specifically, BN calculates the statistics over each mini-batch for normalization during training, while it uses the exponential moving average (EMA) of these mini-batch statistics (which can be viewed as the expectation of mini-batch statistics) in the inference. However, this makes the training and inference of BN inconsistent, especially when the training batch size is small. Actually, the EMA of mini-batch statistics aims to approximate the sample statistics observed during training. Due to the random sampling in the training process, unfortunately, the mini-batch statistics vary a lot [87, 101]. When the training batch size is large, the variation is small so that EMA provides a good estimation of sample statistics. When the batch size is small, however, the variation is big, and EMA becomes inaccurate for statistics estimation, resulting in a significant performance drop [31, 93].

From another perspective, EMA actually assumes that the batch statistics are uncorrelated with the instance statistics within the batch. When the training batch size is large, this assumption nearly holds; but when the batch size is small, this assumption is not true. For example, if the mini-batch only has one sample during training, the statistics of the mini-batch are the statistics of the instance. In this case, BN reduces to instance normalization (IN) [88] in training, but the inference schemes of them are totally different. IN uses the instance statistics for inference while BN adopts the EMA of instance statistics. This problem also exists when the training batch size is larger than one. To solve this issue, some methods [85, 83] have been proposed to combine instance statistics and EMA of batch statistics for BN inference, but they introduce some hyper-parameters to tune. Summers *et al* [85] suggested to use an additional validation set to tune the hyper-parameter and Singh *et al* [83] constructed an auxiliary objective function for each BN layer. These methods are not convenient to implement.

To address the inconsistency issue of BN, we propose a new inference approach, namely batch statistics regression (BSR), by using the instance statistics to predict the batch statistics with a simple linear regression model. BSR is very easy to implement. Similar to EMA, BSR only needs to compute and store some statistics with negligible computational and memory cost in training, while it can estimate the batch statistics more accurately and largely reduce the gap between training and inference. BSR outperforms EMA significantly when training with a small batch size, as demonstrated in our experiments.

3.2 Statistics of Batch Normalization

3.2.1 Batch Normalization

BN [32] performs normalization along the batch dimension of a mini-batch. For each activation or channel of input features in a mini-batch $\mathbf{X}_{\mathcal{B}} = (x_{k_1}, x_{k_2}, ..., x_{k_m})$ of size m, where $\mathcal{B} = \{k_1, k_2, .., k_m\}$ is the set of sampled index, BN normalizes each input feature as:

$$\widehat{x} = \frac{x - \mu_{\mathcal{B}}}{\sqrt{{\sigma_{\mathcal{B}}}^2 + \epsilon}}, \tag{3.1}$$

where $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{k\in\mathcal{B}} x_k$, $\sigma^2_{\mathcal{B}} = \frac{1}{m}\sum_{k\in\mathcal{B}}(x_k - \mu_{\mathcal{B}})^2$, and ϵ is a small positive constant. After normalization, BN also employs a learned affine transform to keep the representative capacity of neural networks.

In convolutional neural networks (CNNs), the most commonly used BN is two dimensional BN (2dBN). In this case, the input activation is a feature map $\mathbf{x}_k \in \mathbb{R}^{h\times w}$, where h and w are the height and width of the feature map. The statistics of a mini-batch is computed along batch, height and width directions, *i.e*, $\mu_{\mathcal{B}} = \frac{1}{hwm}\sum_{k\in\mathcal{B},i,j}\mathbf{x}_k(i,j)$, and $\sigma^2_{\mathcal{B}} = \frac{1}{hwm}\sum_{k\in\mathcal{B},i,j}(\mathbf{x}_k(i,j) - \mu_{\mathcal{B}})^2$. In the following development, we focus on the case of 2dBN. The cases of 1dBN and 3dBN are similar to 2dBN.

3.2.2 Problem of EMA for BN Inference

From Eq. (3.1), one can see that with BN the output activation of one sample depends on the mean and variance computed from all samples in a mini-batch, which are randomly sampled from the training dataset. However, in the inference stage, the output activation of the given test sample is deterministic, and we need to find an estimation of the batch statistics $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ in Eq. (3.1). In the original paper of BN [32], Ioffe and Szegedy suggested to replace $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ with the statistics of the population, *i.e*, $E_{\mathcal{B}}(\mu_{\mathcal{B}})$ and $E_{\mathcal{B}}(\sigma^2_{\mathcal{B}})$, which are often approximated by the EMA of

batch statistics as follows:

$$\mu_n = \lambda\mu_{n-1} + (1-\lambda)\mu_{\mathcal{B}_n}, \quad \sigma_n^2 = \lambda\sigma_{n-1}^2 + (1-\lambda)\sigma_{\mathcal{B}_n}^2, \tag{3.2}$$

where $n = 1, 2, ..., N$ and N is the total number of iterations. μ_N and σ_N^2 are the statistics estimated by EMA in the N-th iteration during training, and $E_{\mathcal{B}}(\mu_{\mathcal{B}}) \approx \mu_N$ and $E_{\mathcal{B}}(\sigma_{\mathcal{B}}^2) \approx \sigma_N^2$.

The EMA of batch mean and variance can be viewed as the expectation of population statistics, and they are constant for any given instance in the inference stage. However, the batch statistics $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ are related to the instances in each mini-batch during training, and they are different for different batches. As a result, there exists a certain statistics disparity between the training and inference stages of BN. Such a disparity cannot be ignored when the training batch size is small. Let's give a more detailed analysis in the next section.

3.2.3 Stochasticity in Batch Statistics

The batch statistics $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}$ can be viewed as a function of input activation $\mathbf{X}_{\mathcal{B}}$. They actually depend on the instance statistics of each sample in the batch. For a specific sample $\mathbf{x}_s$, we denote by

$$\mu_s = \frac{1}{hw}\sum\nolimits_{i,j}\mathbf{x}_s(i,j), \quad \sigma_s^2 = \frac{1}{hw}\sum\nolimits_{i,j}(\mathbf{x}_s(i,j) - \mu_s)^2, \tag{3.3}$$

the instance statistics of $\mathbf{x}_s$ and denote by

$$\begin{aligned} \mu_{\mathcal{B}\backslash\{s\}} &= \frac{1}{hw(m-1)}\sum\nolimits_{k\in\mathcal{B}\backslash\{s\},i,j,}\mathbf{x}_k(i,j), \\ \sigma_{\mathcal{B}\backslash\{s\}}^2 &= \frac{1}{hw(m-1)}\sum\nolimits_{k\in\mathcal{B}\backslash\{s\},i,j,}(\mathbf{x}_k(i,j) - \mu_{\mathcal{B}\backslash\{s\}})^2, \end{aligned} \tag{3.4}$$

the statistics of mini-batch $\mathbf{X}_{\mathcal{B}}$ excluding sample $\mathbf{x}_s$ (*i.e*, $\mathbf{X}_{\mathcal{B}\setminus\{s\}}$). As in [83], we can separate the instance statistics from the batch statistics as follows:

$$\mu_{\mathcal{B}} = \alpha\mu_s + (1-\alpha)\mu_{\mathcal{B}\setminus\{s\}}, \sigma^2_{\mathcal{B}} = \beta\sigma^2_s + (1-\beta)\sigma^2_{\mathcal{B}\setminus\{s\}} + \beta(1-\beta)(\mu_s - \mu_{\mathcal{B}\setminus\{s\}})^2, \tag{3.5}$$

where the hyper-parameters $\alpha = \beta = \frac{1}{m}$ control the impact of instance statistics on batch statistics. Eq. (3.5) indicates that batch statistics are related to instance statistics. When the training batch size m is small, α and β become big, and the dependency of batch statistics on instance statistics is strong. Fig. 3.1 illustrates the stochasticity in BN. In training, for a specific sample, its statistics (*i.e*, μ_s and σ^2_s) are deterministic, because we can obtain them only from the give sample, but the statistics of other samples within the minibatch (*i.e*, $\mu_{\mathcal{B}\setminus\{s\}}$ and $\sigma^2_{\mathcal{B}\setminus\{s\}}$) are stochastic in different epochs.

In the inference step, for a given test sample, we can view it as the sample $\mathbf{x}_s$ in the training step and its instance statistics μ_s and σ^2_s can be computed by Eq. (3.3). To estimate $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$, from Eq. (3.5) we can see that we need to estimate $\mu_{\mathcal{B}\setminus\{s\}}$ and $\sigma^2_{\mathcal{B}\setminus\{s\}}$. Since in inference only the given test sample is available, we approximate $\mu_{\mathcal{B}\setminus\{s\}}$ and $\sigma^2_{\mathcal{B}\setminus\{s\}}$ by their expectations, *i.e*, $E_{\mathcal{B}\setminus\{s\}}(\mu_{\mathcal{B}})$ and $E_{\mathcal{B}\setminus\{s\}}(\sigma^2_{\mathcal{B}})$. However, EMA (refer to Eq. (3.2)) directly takes the expectations $E_{\mathcal{B}}(\mu_{\mathcal{B}})$ and $E_{\mathcal{B}}(\sigma^2_{\mathcal{B}})$ by using all training batches as the estimations of $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$, ignoring the specific instance statistics of sample $\mathbf{x}_s$. Consequently, the statistics inconsistency between training and inference stages of BN is introduced.

3.2.4 Expectation of Batch Statistics

As discussed in section 3.2.3, to estimate the statistics $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ for a test sample $\mathbf{x}_s$, we should first compute μ_s and σ^2_s from itself by Eq. (3.3), and then compute

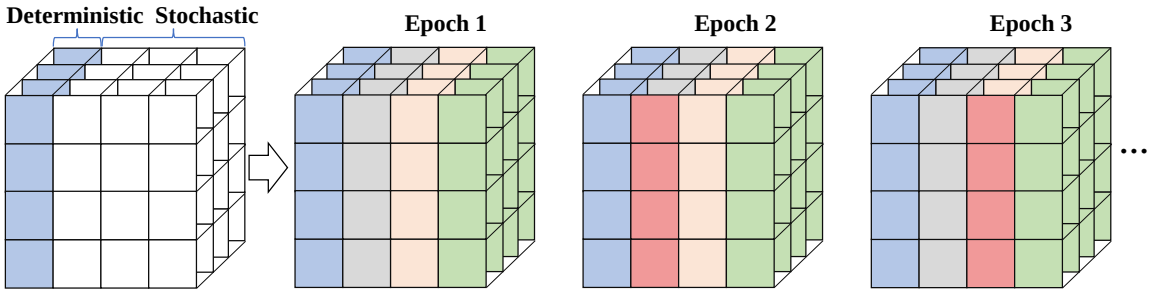


Figure 3.1: Illustration of the stochasticity in BN. For a specific sample (blue sample), in the training process, its statistics are deterministic but other samples within the minibatch are stochastic.

$E_{\mathcal{B}\backslash\{s\}}(\mu_{\mathcal{B}})$ and $E_{\mathcal{B}\backslash\{s\}}(\sigma^2_{\mathcal{B}})$. Therefore, from Eq. (3.5), we have:

$$\begin{aligned} E_{\mathcal{B}\backslash\{s\}}(\mu_{\mathcal{B}}) &= \alpha\mu_s + (1-\alpha)E_{\mathcal{B}\backslash\{s\}}(\mu_{\mathcal{B}\backslash\{s\}}), \\ E_{\mathcal{B}\backslash\{s\}}(\sigma^2_{\mathcal{B}}) &= \beta\sigma^2_s + (1-\beta)E_{\mathcal{B}\backslash\{s\}}(\sigma^2_{\mathcal{B}\backslash\{s\}}) \\ &\quad + \beta(1-\beta)(\mu^2_s - 2\mu_s E_{\mathcal{B}\backslash\{s\}}(\mu_{\mathcal{B}\backslash\{s\}}) + E_{\mathcal{B}\backslash\{s\}}(\mu^2_{\mathcal{B}\backslash\{s\}})). \end{aligned} \tag{3.6}$$

If we assume that the elements of input activations are independently and identically distributed (i.i.d.) and follow Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, it can be derived that

$$E_{\mathcal{B}\backslash\{s\}}(\mu_{\mathcal{B}\backslash\{s\}}) = \mu, E_{\mathcal{B}\backslash\{s\}}(\sigma^2_{\mathcal{B}\backslash\{s\}}) = \frac{hw(m-1)}{hw(m-1)-1}\sigma^2, E_{\mathcal{B}\backslash\{s\}}(\mu^2_{\mathcal{B}\backslash\{s\}}) = \frac{1}{hw(m-1)}\sigma^2 + \mu^2, \tag{3.7}$$

where μ and σ^2 can be estimated by the expectation of batch statistics, *i.e*, $\mu = E_{\mathcal{B}}(\mu_{\mathcal{B}})$ and $\sigma^2 = \frac{hwm-1}{hwm}E_{\mathcal{B}}(\sigma^2_{\mathcal{B}})$. By introducing Eq. (3.7) into Eq. (3.6) and replacing the instance statistics of sample $\mathbf{x}_s$ with those of a testing sample, we can obtain an inference formulation which actually combines the instance statistics and EMA.

The above analysis is based on the assumption that the input activations of different samples are i.i.d. Unfortunately, there are usually many BN layers in a DNN, while only the first BN layer satisfies the i.i.d. assumption. In other BN layers, the features of all samples in previous BN layers have been used in the normalization step. As a result, the input activations are actually correlated. Therefore, the inference formulation in Eqs. (3.6) and (3.7) are not accurate and the settings for hyper-parameters

$\{\alpha, \beta\}$ should not be $1/m$ for most BN layers.

To reduce the gap between normalized activations in training and inference, Singh and Shrivastava [83] proposed to tune the hyper-parameters $\{\alpha, \beta\}$ for each BN layer by optimizing an auxiliary nonconvex objective function, which is however difficult to implement. Different from [83], in the next section, we propose to regress the batch mean and variance with instance statistics. We show that this can be formulated as a simple linear regression problem, which can be easily solved with a closed-form solution. Meanwhile, the online updating formulation makes it very easy and efficient to implement in real applications.

3.3 Batch Statistics Regression

3.3.1 Batch Statistics Regression Model

Suppose for any simple $\mathbf{x}_s$, $f_\mu(\mathbf{x}_s)$ and $f_\sigma(\mathbf{x}_s)$ are the mean and variance that we will use in the inference step, we hope that when $\mathbf{x}_s$ in a training batch, its statistics used in training are close as the statistics it used in inference. We propose the following objective function for each channel of a BN layer:

$$\min_{f_\mu} E_{\mathcal{B}}(\sum\nolimits_{s\in\mathcal{B}} ||\mu_{\mathcal{B}} - f_\mu(\mathbf{x}_s)||_2^2), \quad \min_{f_\sigma} E_{\mathcal{B}}(\sum\nolimits_{s\in\mathcal{B}} ||\sigma_{\mathcal{B}}^2 - f_\sigma(\mathbf{x}_s)||_2^2), \tag{3.8}$$

where f_μ and f_σ are two functions of instance $\mathbf{x}_s$ to predict the batch statistics. Once learned, in the inference step $f_\mu(\mathbf{x}_s)$ and $f_\sigma(\mathbf{x}_s)$ can be used to estimate the batch statistics for any given testing sample. We call our model *batch statistics regression* (BSR), which uses instance statistics to regress batch statistics.

One key problem of BSR is the choice of f_μ and f_σ. From Eq. (3.6), it can be seen that the instance statistics are related to batch statistics. More specifically, μ_s is related to $\mu_{\mathcal{B}}$, while σ_s^2, μ_s and μ_s^2 are related to $\sigma_{\mathcal{B}}^2$, and their relationship is linear.

Therefore, we think a linear regression is preferred for its simplicity. Based on Eq. (3.6), we propose the following formulation:

$$f_\mu(\mathbf{x}_s) = \phi(\mathbf{x}_s)^T\boldsymbol{\alpha}, f_\sigma(\mathbf{x}_s) = \varphi(\mathbf{x}_s)^T\boldsymbol{\beta}, \tag{3.9}$$

where $\phi(\mathbf{x}_s) = (\mu_s, 1)^T$, $\varphi(\mathbf{x}_s) = (\sigma_s^2, \mu_s^2, \mu_s, 1)^T$ are the basis functions, and $\boldsymbol{\alpha} \in \mathbb{R}^2$ and $\boldsymbol{\beta} \in \mathbb{R}^4$ are the parameters to be learned. It should be noted that the predicted variance $f_\sigma(\mathbf{x}_s)$ should be nonnegative. If we directly use the linear function $\varphi(\mathbf{x}_s)^T\boldsymbol{\beta}$ to regress $\sigma_{\mathcal{B}}^2$, however, the non-negativity of predicted variance cannot be always ensured. To solve this problem, we set $\varphi(\mathbf{x}_s) = (\sigma_s^2, (\mu_s - \mu)^2, 1)^T$, where μ is the population mean that can be estimated by EMA (refer to Eq. (3.2)). And after estimating $\boldsymbol{\beta} \in \mathbb{R}^3$, we let $\boldsymbol{\beta} = max(\boldsymbol{\beta}, 0)$ so that $\varphi(\mathbf{x}_s)^T\boldsymbol{\beta}$ can be guaranteed to be nonnegative for any instance $\mathbf{x}_s$.

We have the following objective functions to estimate $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$:

$$\begin{aligned} &\min_{\boldsymbol{\alpha}} \sum\nolimits_{n=1}^{N} (\lambda^{N-n} \sum\nolimits_{s\in\mathcal{B}_n} ||\mu_{\mathcal{B}_n} - \phi(\mathbf{x}_s)^T\boldsymbol{\alpha})||_2^2), \\ &\min_{\boldsymbol{\beta}} \sum\nolimits_{n=1}^{N} (\lambda^{N-n} \sum\nolimits_{s\in\mathcal{B}_n} ||\sigma_{\mathcal{B}_n}^2 - \varphi(\mathbf{x}_s)^T\boldsymbol{\beta}||_2^2), \end{aligned} \tag{3.10}$$

where $\mathcal{B}_n$ denotes the batch in the n-th iteration, N is the total number of batches, and λ is the momentum parameter (*e.g*, $\lambda = 0.9$) that balances the importance of the current batch and past batches. Eq. (3.10) is a least square regression problem, which has a closed-form solution:

$$\boldsymbol{\alpha}_N = (\mathbf{A}_N^\mu)^{-1}\mathbf{B}_N^\mu, \quad \boldsymbol{\beta}_N = (\mathbf{A}_N^\sigma)^{-1}\mathbf{B}_N^\sigma, \tag{3.11}$$

where

$$\begin{aligned}
\mathbf{A}_N^{\mu} &= (1-\lambda)\sum_{n=1}^{N}\lambda^{N-n}\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T, \mathbf{B}_N^{\mu} = (1-\lambda)\sum_{n=1}^{N}\lambda^{N-n}\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_n}, \\
\mathbf{A}_N^{\sigma} &= (1-\lambda)\sum_{n=1}^{N}\lambda^{N-n}\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T, \mathbf{B}_N^{\sigma} = (1-\lambda)\sum_{n=1}^{N}\lambda^{N-n}\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_n}^2.
\end{aligned} \tag{3.12}$$

3.3.2 Online Updating Formula

The closed-form solution in Eq. (3.12), however, requires us to store the statistics of past historical batches, which is not practical to implement. Fortunately, with the online learning strategy [103], we can adopt the following online updating formula to compute the statistics:

$$\begin{aligned}
\mathbf{A}_N^{\mu} &= \lambda\mathbf{A}_{N-1}^{\mu} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T, \mathbf{B}_N^{\mu} = \lambda\mathbf{B}_{N-1}^{\mu} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_N}, \\
\mathbf{A}_N^{\sigma} &= \lambda\mathbf{A}_{N-1}^{\sigma} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T, \mathbf{B}_N^{\sigma} = \lambda\mathbf{B}_{N-1}^{\sigma} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_N}^2.
\end{aligned} \tag{3.13}$$

It can be shown that the updating formula in Eq. (3.13) is equivalent to Eq. (3.12). The proof is shown as follows:

Proof. According to Eq. (3.12), let the index N be $N-1$, we have

$$\begin{aligned}
\mathbf{A}_{N-1}^{\mu} &= (1-\lambda)\textstyle\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T, \\
\mathbf{B}_{N-1}^{\mu} &= (1-\lambda)\textstyle\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_n}, \\
\mathbf{A}_{N-1}^{\sigma} &= (1-\lambda)\textstyle\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T, \\
\mathbf{B}_{N-1}^{\sigma} &= (1-\lambda)\textstyle\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_n}^2.
\end{aligned} \tag{3.14}$$

then

$$
\begin{aligned}
\mathbf{A}_N^{\mu} &= \lambda(1-\lambda)\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T \\
&= \lambda\mathbf{A}_{N-1}^{\mu} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T, \\
\mathbf{B}_N^{\mu} &= \lambda(1-\lambda)\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_n} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_N} \\
&= \lambda\mathbf{B}_{N-1}^{\mu} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_N}, \\
\mathbf{A}_N^{\sigma} &= \lambda(1-\lambda)\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T \\
&= \lambda\mathbf{A}_{N-1}^{\sigma} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T, \\
\mathbf{B}_N^{\sigma} &= \lambda(1-\lambda)\sum_{n=1}^{N-1}\lambda^{N-1-n}\sum_{s\in\mathcal{B}_n}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_n}^2 + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_N}^2 \\
&= \lambda\mathbf{B}_{N-1}^{\sigma} + (1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma_{\mathcal{B}_N}^2.
\end{aligned}
\tag{3.15}
$$

■

Therefore, we only need to store four statistics $\mathbf{A}_N^{\mu} \in \mathbb{R}^{2\times 2}$, $\mathbf{B}_N^{\mu} \in \mathbb{R}^2$, $\mathbf{A}_N^{\sigma} \in \mathbb{R}^{3\times 3}$ and $\mathbf{B}_N^{\sigma} \in \mathbb{R}^3$ for each channel during training, and update them with Eq. (3.13). The number of parameters that needs to be stored for one BN layer is $18C$, where C is the number of channels. This number is negligible compared with the parameters in a CNN layer. The computational complexity of BSR is the same as BN, *i.e*, $O(NChw)$. So extra computational cost is very small compared with the normalization operation on input activation. Moreover, in case these statistics are not stored in the training process (*e.g*, the original BN), we can simply run several more training epochs (*e.g*, one epoch) to obtain these statistics.

To make a better understanding of BSR, from Eq. (3.11) and (3.13), it can be shown that the objective functions in Eq. (3.10) are equivalent to the following objective functions:

$$
\begin{aligned}
&\min_{\boldsymbol{\alpha}}(1-\lambda)\sum_{s\in\mathcal{B}_N}||\mu_{\mathcal{B}_N} - \phi(\mathbf{x}_s)^T\boldsymbol{\alpha})||_2^2 + \lambda||\boldsymbol{\alpha} - \boldsymbol{\alpha}_{N-1}||_{\mathbf{A}_{N-1}^{\mu}}, \\
&\min_{\boldsymbol{\beta}}(1-\lambda)\sum_{s\in\mathcal{B}_N}||\sigma_{\mathcal{B}_N}^2 - \varphi(\mathbf{x}_s)^T\boldsymbol{\beta}||_2^2 + \lambda||\boldsymbol{\beta} - \boldsymbol{\beta}_{N-1}||_{\mathbf{A}_{N-1}^{\sigma}},
\end{aligned}
\tag{3.16}
$$

where $\boldsymbol{\alpha}_{N-1}$ and $\boldsymbol{\beta}_{N-1}$ are the regression parameters obtained in the $(N-1)$-th iteration, and $||\mathbf{x}||_{\mathbf{A}} = \mathbf{x}^T\mathbf{A}\mathbf{x}$ is the Mahalanobis distance. The proof is shown as

follows:

Proof. we can first optimize Eq. (3.10) by letting its derivative be zero. There is

$$\begin{aligned}(1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)(\mu_{\mathcal{B}_N}-\phi(\mathbf{x}_s)^T\boldsymbol{\alpha}))+\lambda\mathbf{A}^{\mu}_{N-1}(\boldsymbol{\alpha}-\boldsymbol{\alpha}_{N-1})=0,\\(1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)(\sigma^2_{\mathcal{B}_N}-\varphi(\mathbf{x}_s)^T\boldsymbol{\beta})+\lambda\mathbf{A}^{\sigma}_{N-1}(\boldsymbol{\beta}-\boldsymbol{\beta}_{N-1})=0,\end{aligned}\tag{3.17}$$

From Eq. (3.17), we have

$$\begin{aligned}\boldsymbol{\alpha}_{N-1}&=(\lambda\mathbf{A}^{\mu}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T)^{-1}(\lambda\mathbf{A}^{\mu}_{N-1}\boldsymbol{\alpha}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_N})\\\boldsymbol{\beta}_{N-1}&=(\lambda\mathbf{A}^{\sigma}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T)^{-1}(\lambda\mathbf{A}^{\sigma}_{N-1}\boldsymbol{\beta}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma^2_{\mathcal{B}_N})\end{aligned}\tag{3.18}$$

By letting N be $N-1$ in Eq. (3.11), we have $\boldsymbol{\alpha}_{N-1}=(\mathbf{A}^{\mu}_{N-1})^{-1}\mathbf{B}^{\mu}_{N-1},\boldsymbol{\beta}_{N-1}=(\mathbf{A}^{\sigma}_{N-1})^{-1}\mathbf{B}^{\sigma}_{N-1}$, and consequently

$$\mathbf{A}^{\mu}_{N-1}\boldsymbol{\alpha}_{N-1}=\mathbf{B}^{\mu}_{N-1},\mathbf{A}^{\sigma}_{N-1}\boldsymbol{\beta}_{N-1}=\mathbf{B}^{\sigma}_{N-1}\tag{3.19}$$

According to Eq (3.13) and Eq (3.19), Eq (3.20) can be rewritten as

$$\begin{aligned}\boldsymbol{\alpha}_{N-1}&=(\lambda\mathbf{A}^{\mu}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\phi(\mathbf{x}_s)^T)^{-1}(\lambda\mathbf{B}^{\mu}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\phi(\mathbf{x}_s)\mu_{\mathcal{B}_N})\\&=(\mathbf{A}^{\mu}_{N})^{-1}\mathbf{B}^{\mu}_{N}\\\boldsymbol{\beta}_{N-1}&=(\lambda\mathbf{A}^{\sigma}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\varphi(\mathbf{x}_s)^T)^{-1}(\lambda\mathbf{B}^{\sigma}_{N-1}+(1-\lambda)\sum_{s\in\mathcal{B}_N}\varphi(\mathbf{x}_s)\sigma^2_{\mathcal{B}_N})\\&=(\mathbf{A}^{\sigma}_{N})^{-1}\mathbf{B}^{\sigma}_{N}\end{aligned}\tag{3.20}$$

which is the solution of the objective functions in Eq. (3.10). The proof is completed. ■

From Eq. (3.16), we can see that the first term accounts for the observation term from the current data and the second term accounts for the regularization from the past historical data.

In the inference stage, we first use Eq. (3.11) to calculate the regression parameters $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, and let $\boldsymbol{\beta}=max(\boldsymbol{\beta},0)$ to ensure that the variance is nonnegative. Then

the final inference of the testing sample is applied to

$$\widehat{\mathbf{x}} = \frac{\mathbf{x} - \phi(\mathbf{x})^T \boldsymbol{\alpha}}{\sqrt{\varphi(\mathbf{x})^T \boldsymbol{\beta} + \epsilon}}. \tag{3.21}$$

Compared with Eqs. (3.6) and (3.7), which employ fixed hyper-parameters, BSR applies optimized parameters for different BN layers. The proposed BSR algorithm for BN inference is summarized in **Algorithm 2**.

3.3.3 Relationship with EMA

It can be shown that the classical EMA in Eq. (3.2) is a special case of the proposed BSR. Actually, if we set the basis functions of linear regression in BSR to constants, *e.g*, $\phi(\mathbf{x}_s) = 1$, $\varphi(\mathbf{x}_s) = 1$, and let $\alpha = \mu$ and $\beta = \sigma^2$, then the objective function of BSR in Eq. (3.16) degrades to

$$\begin{aligned} \mu_N &= \arg\min_{\mu} \ (1-\lambda)||\mu_{\mathcal{B}_n} - \mu||_2^2 + \lambda||\mu - \mu_{N-1}||_2^2, \\ \sigma_N^2 &= \arg\min_{\sigma^2} \ (1-\lambda)||\sigma_{\mathcal{B}_n}^2 - \sigma^2||_2^2 + \lambda||\sigma^2 - \sigma_{N-1}^2||_2^2. \end{aligned} \tag{3.22}$$

The solution of Eq. (3.22) is exactly the EMA updating rules for mini-batch statistics in Eq. (3.2). Therefore, the original BN inference method of EMA is a special case of BSR with constant basis functions, while our proposed BSR adopts more informative basis functions $\phi(\mathbf{x}_s) = (\mu_s, 1)^T$ and $\varphi(\mathbf{x}_s) = (\sigma_s^2, (\mu_s - \mu)^2, 1)^T$ of instance statistics. BSR is a natural generalization of EMA and it achieves a more accurate approximation to batch statistics than EMA.

3.3.4 Measure of Disparity

The regression problems in Eq. (3.10) or Eq. (3.16) are actually stochastic regression problems, where both the targets $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ and the input $\mathbf{x}_s$ have certain stochasticity. In order to validate whether the proposed BSR can reduce the disparity between

Algorithm 2: Batch Normalization with Batch Statistics Regression (BSR)

Input: Sample $\mathbf{x}$ over a training mini-batch $\mathbf{X}_{\mathcal{B}}$; statistics μ, $\mathbf{A}^{\mu}$, $\mathbf{B}^{\mu}$, $\mathbf{A}^{\sigma}$ and $\mathbf{B}^{\sigma}$, momentum parameters λ.

Output: $\{\widehat{\mathbf{x}} = \mathrm{BN}(\mathbf{x})\}$; $\mathbf{A}^{\mu}$, $\mathbf{B}^{\mu}$, $\mathbf{A}^{\sigma}$ and $\mathbf{B}^{\sigma}$

if *Training* **then**
- $\mu_{\mathcal{B}} = \frac{1}{hwm}\sum_{k\in\mathcal{B},i,j}\mathbf{x}_k(i,j)$
- $\sigma^2_{\mathcal{B}} = \frac{1}{hwm}\sum_{k\in\mathcal{B},i,j}(\mathbf{x}_k(i,j)-\mu_{\mathcal{B}})^2$
- $\widehat{\mathbf{x}}_k = \frac{\mathbf{x}_k-\mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}}+\epsilon}}, \quad k\in\mathcal{B}$
- $\mu \leftarrow \lambda\mu + (1-\lambda)\mu_{\mathcal{B}}$
- update $\mathbf{A}^{\mu}$, $\mathbf{B}^{\mu}$, $\mathbf{A}^{\sigma}$ and $\mathbf{B}^{\sigma}$ according to Eq. (3.13)

else
- Solve $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ according to Eq. (3.11) and $\boldsymbol{\beta} = max(\boldsymbol{\beta}, 0)$
- $\widehat{\mathbf{x}} = \frac{\mathbf{x}-\phi(\mathbf{x})^T\boldsymbol{\alpha}}{\sqrt{\varphi(\mathbf{x})^T\boldsymbol{\beta}+\epsilon}}$.

end

the training and inference stages of BN, we need a measure to quantify the difference. Here, we adopt the following measure:

$$\Delta_{\mathbf{x}} = E_{\mathcal{B}}\Big(\frac{1}{m}\sum_{s\in\mathcal{B}}\Big|\Big|\frac{\mathbf{x}_s-\mu_{\mathcal{B}}}{\sqrt{{\sigma_{\mathcal{B}}}^2+\epsilon}} - \frac{\mathbf{x}_s - f_{\mu}(\mathbf{x}_s)}{\sqrt{f_{\sigma}(\mathbf{x}_s)+\epsilon}}\Big|\Big|_2\Big), \tag{3.23}$$

where $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ are the statistics of a training batch $\mathcal{B}$, while $f_{\mu}(\mathbf{x}_s)$ and $f_{\sigma}(\mathbf{x}_s)$ are the predicted batch statistics by sample $\mathbf{x}_s$, suppose it is used for inference. For EMA inference, $f_{\mu}(\mathbf{x}_s) = \mu$ and $f_{\sigma}(\mathbf{x}_s) = \sigma$, which can be obtained by Eq. (3.2), while for BSR inference, we have $f_{\mu}(\mathbf{x}_s) = \phi(\mathbf{x})^T\boldsymbol{\alpha}$ and $f_{\sigma}(\mathbf{x}_s) = \varphi(\mathbf{x})^T\boldsymbol{\beta}$.

The $\Delta_{\mathbf{x}}$ in Eq. (3.23) measures the difference of output activations between training and inference, which is also adopted in [29] to measure the stochasticity in BN for different training schemes. In our work, we use the same training scheme of BN and compare different inference approaches by their values of $\Delta_{\mathbf{x}}$.

3.4 Experiments

A series of experiments are conducted in this section to evaluate the effectiveness of BSR. We first conduct experiments on two commonly used small datasets, CIFAR100 and CIFAR10 [39], to analyze the performance of BSR with different batch sizes. We then perform the experiment on ImageNet [75] to demonstrate that BSR can work well on large-scale datasets. Moreover, to testify the performance of BSR on transfer learning, we also evaluate it on four fine-grained image classification (FGVC) datasets (Aircraft [60], Stanford Cars [38], Stanford Dogs [35] and CUB-200-2011 [91]) with the pre-trained DNN models on ImageNet. Finally, we employ the COCO dataset to show that BSR can boost BN on other tasks beyond image classification, such as object detection.

All experiments are conducted under the Pytorch 1.5 framework with several NVIDIA Tesla P100 machines. Ghost-BN [23] can be adopted to simulate multiple GPUs training with one GPU for BN. Therefore, we can fix the total batch size and simulate the training process with different numbers of GPUs with Ghost-BN so that the GPUs can be fully taken use of. The statistics of the BN layer are computed within one simulated GPU. We compare BSR with the widely used EMA and the recently proposed EvalNorm (EN) [83]. Since the source code of EN is not publically available, we implement it following the configurations in [83]. The implementation of BSR is much simpler than EN since we have a closed-form solution, while EN needs to first compute the gradient of the auxiliary objective function and then update the parameters with gradient descent.

3.4.1 CIFAR100/CIFAR10

Experimental Settings: CIFAR100/CIFAR10 [39] include 60,000 images from 100 and 10 classes, respectively. $50K$ images are used for training and 10K images for

Table 3.1: Testing accuracies (%) on CIFAR100/CIFAR10.

	CIFAR100 / ResNet50					
Samples/GPU	2	4	8	16	32	64
EMA	70.47	77.23	78.93	79.02	78.98	78.63
EN	74.30	78.03	79.35	79.15	79.00	78.63
BSR	75.88	78.30	79.53	79.50	79.28	78.80
$\Delta_{\text{BSR}-\text{EMA}}$	↑5.41	↑1.07	↑0.60	↑0.48	↑0.22	↑0.17
	CIFAR100 / ResNet101					
EMA	70.67	77.37	79.47	79.53	79.78	79.13
EN	74.85	78.18	79.83	79.63	79.78	79.15
BSR	76.30	78.23	80.03	79.97	79.90	79.35
$\Delta_{\text{BSR}-\text{EMA}}$	↑5.63	↑0.86	↑0.56	↑0.44	↑0.12	↑0.22
	CIFAR10 / ResNet50					
Samples/GPU	2	4	8	16	32	64
EMA	93.33	95.02	95.27	95.27	95.20	95.05
EN	94.25	95.22	95.37	95.27	95.22	95.05
BSR	94.60	95.27	95.40	95.37	95.25	95.10
$\Delta_{\text{BSR}-\text{EMA}}$	↑1.27	↑0.25	↑0.13	↑0.1	↑0.05	↑0.05
	CIFAR10 / ResNet101					
EMA	93.80	95.22	94.97	95.40	95.37	95.45
EN	94.65	95.40	95.13	95.40	95.37	95.45
BSR	95.00	95.42	95.17	95.45	95.40	95.47
$\Delta_{\text{BSR}-\text{EMA}}$	↑1.2	↑0.2	↑0.2	↑0.05	↑0.03	↑0.02

testing. The image resolution is 32×32. All DNN models are trained for 200 epochs. We set the overall batch size as 128 and change the number of GPUs so that in training the number of samples per GPU varies from 2 to 64. We repeat the experiments 5 times and report the average accuracy. We employ ResNet50 and ResNet101 [22] as DNN models, which are optimized by SGDM with a momentum of 0.9. The weight decay is set to 0.0005. The initial learning rate is 0.1 and it decays by 0.1 for every 60 epochs. For both EMA and BSR, the momentum parameter is set to 0.1.

Results: Since during training the normalization step in BN is the same for EMA, EN and BSR, we only compare their results in inference. Table 3.1 reports the classification accuracies of ResNet50 and ResNet101 with a different number of samples per GPU. The results show that BSR consistently outperforms EMA and EN. When the training batch size is small, BSR can largely boost the performance over EMA. In particular, with 2 samples per GPU, BSR achieves 5.54% and 5.63% performance gain over EMA on CIFAR100 with ResNet50 and ResNet101 backbone, respectively, and the gains are 1.27% and 1.2% on CIFAR10. When the batch size increases, the

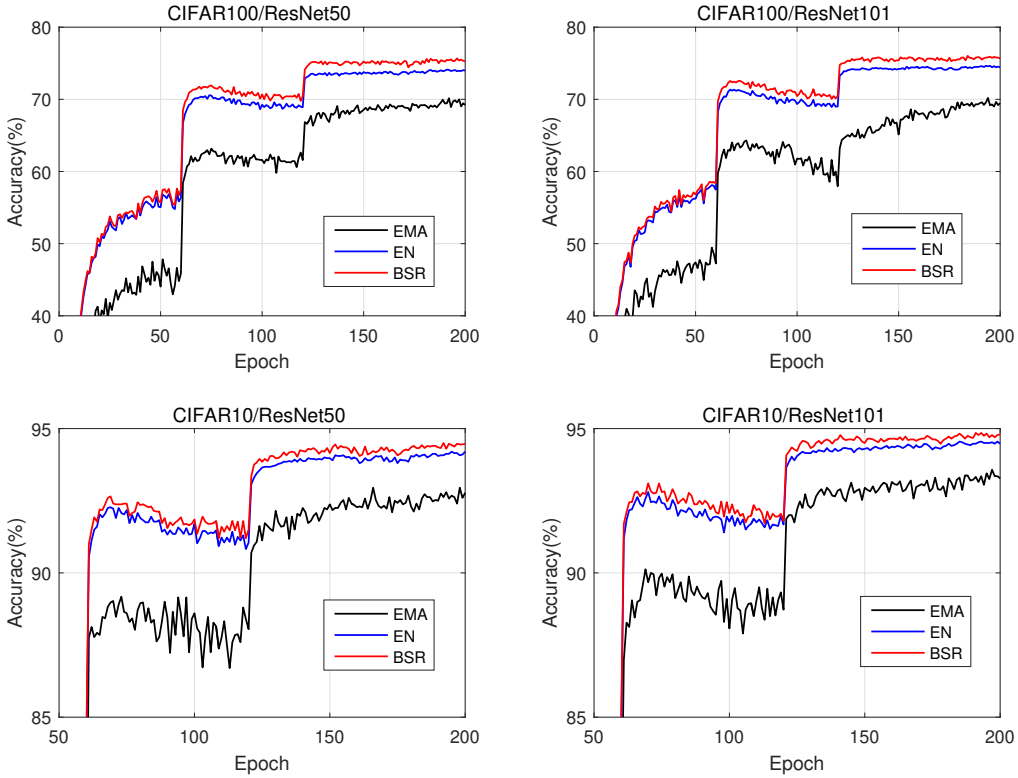


Figure 3.2: The testing accuracy curves of ResNet50 and ResNet101 on CIFAR100 (top row) and CIFAR10 (bottom row) with 2 samples per GPU during training.

improvement by BSR decreases. This is because when the batch size is large, BSR approaches EMA, and the inconsistency between BN training and inference stages will be diluted.

Fig. 3.2 plots the testing accuracy curves of ResNet50 and ResNet101 for different training epochs with 2 samples per GPU. It can be seen that BSR performs the best during the whole training process. It is also observed that higher performance gain is achieved on CIFAR100 than CIFAR10. This implies that BSR can bring more benefit for more difficult problems (100 classes vs. 10 classes), where an accurate estimation of BN statistics is more important.

Meanwhile, it is found that on both CIFAR100 and CIFAR10, the best results are achieved when the number of samples is 8 or 16 within one GPU. This is because BN

Table 3.2: Validation accuracies (%) of ResNet50 on ImageNet.

Samples/GPU	2	4	8	16	32
GN [93]	75.51	75.48	75.53	75.50	75.47
BRN [31]	70.70	73.30	74.24	75.70	76.23
BN+EMA [32]	65.70	72.92	75.24	75.88	76.35
BN+EN [83]	69.74	73.92	75.62	76.01	76.41
BN+BSR	72.24	74.45	75.66	76.20	76.58
$\Delta_{\text{BSR}-\text{EMA}}$	↑6.54	↑1.53	↑0.42	↑0.32	↑0.23

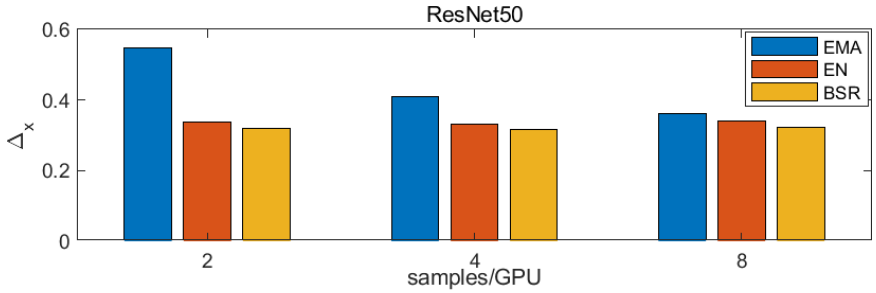


Figure 3.3: The $\Delta_{\mathbf{x}}$ of EMA, EN and BSR ResNet50 on ImageNet.

also acts as a regularization function to regularize the stochasticity of batch statistics during training [87, 105, 101]. Too-big batch size will weaken the regularization property of BN. Therefore, a proper training batch size for BN can boost the generalization performance of trained DNN models. In such a case, BSR can effectively improve the inference performance of BN. For example, on CIFAR100 the best performance of EMA is 79.02% when the batch size is 16, and BSR can further improve the accuracy to 79.50%, which is much higher than the results with a commonly used large batch size, *e.g*, 78.63% with batch size 64. Therefore, This means that BSR with GhostBN [23] can significantly gain the performance with common training batch size for some tasks.

3.4.2 ImageNet

Experimental Settings: We then evaluate BSR on ImageNet [75], which consists of 1.28M images from 1,000 categories for training and $50K$ images for validation.

Table 3.3: Validation accuracies (%) of more models on ImageNet.

Model\ Samples/GPU		2	4	8	16	32
ResNet101	EMA	68.63	75.33	77.12	77.80	78.06
	EN	72.70	76.30	77.48	77.93	78.08
	BSR	75.01	76.43	77.60	78.24	78.22
	$\Delta_{\text{BSR}-\text{EMA}}$	↑6.38	↑1.10	↑0.48	↑0.44	↑0.16
VGG16	EMA	65.10	70.43	72.91	73.70	74.24
	EN	70.71	72.25	73.27	73.77	74.25
	BSR	71.41	72.54	73.53	73.90	74.37
	$\Delta_{\text{BSR}-\text{EMA}}$	↑6.31	↑2.11	↑0.62	↑0.20	↑0.13
ResNeXt50	EMA	68.40	74.61	77.00	77.68	77.71
	EN	71.37	76.33	77.55	77.85	77.74
	BSR	74.27	75.96	77.59	77.90	78.04
	$\Delta_{\text{BSR}-\text{EMA}}$	↑5.87	↑1.35	↑0.59	↑0.22	↑0.33

The resolution of input images is 224. All DNN models are trained for 100 epochs with an overall batch size of 256. We use 128, 64, 32, 16, and 8 GPUs to train the models, and the number of samples per GPU is accordingly 2, 4, 8, 16, and 32. All networks are trained using SGDM with a momentum 0.9. The weight decay is set to 0.0001. The initial learning rate is 0.1 and it decays by 0.1 for every 30 epochs. The momentum of EMA and BSR is set to be 0.1.

Results: We first compare BSR with GN [93] and BRN [31], as well as EMA and EN for inference. GN and BRN are two well-known normalization methods to address the problem of the small training batch size of BN. For GN, the group number is 32, and for BRN, we set $r_{max} = 2$ and $d_{max} = 1$. Table 3.2 lists the validation accuracies of ResNet50 on ImageNet with 2, 4, 8, 16 and 32 samples per GPU. From Table 3.2, we can see that BSR performs consistently better than BRN, EMA and EN across all batch sizes. Especially, when training batch size is 2, it achieves a significant performance gain over EMA (*i.e*, 6.54%). Compared with GN, BSR outperforms it when the batch size is larger than 4. It should be stressed that BSR does not change the training scheme of BN, while GN uses a different normalization strategy from BN.

We then show in Fig. 3.3 the average value of $\Delta_{\mathbf{x}}$ (refer to Eq. (3.23)) over all BN layers in ResNet50 for EMA, EN and BSR with training batch sizes 2, 4 and 8. As explained in Section 3.3.4, $\Delta_{\mathbf{x}}$ measures the disparity of batch statistics between BN training and inference. A smaller value of $\Delta_{\mathbf{x}}$ means that the batch statistics in training and inference are more consistent, implying better prediction results in general. We see that the $\Delta_{\mathbf{x}}$ of EMA is significantly larger than EN and BSR, especially when the batch size is 2. BSR has the smallest $\Delta_{\mathbf{x}}$, which means that it can better reduce the gap of statistics between BN training and inference.

We also apply BSR on more DNN models, including ResNet101, VGG16 and ResNetX50. Table 3.3 shows the validation accuracies on ImageNet with 2, 4, 8, 16 and 32 samples per GPU. It can be seen that BSR consistently outperforms EMA under all batch sizes for all DNN models. It works better than EN under most settings as well. Especially, when the training batch size is 2, BSR achieves 6.38%, 6.31% and 5.87% performance gains over EMA, and 2.31%, 0.70% and 2.90% performance gains over EN by using the three DNN models, respectively. Such improvements clearly validate that BSR provides a much more accurate solution to estimate the batch statistics when the batch size is small.

3.4.3 Fine-grained Image Classification

Experimental Settings: In our above experiments, the DNN models are trained from scratch. To show that BSR can also be adopted in pre-trained models, we conduct experiments on four popular fine-grained visual classification (FGVC) datasets, including Aircraft [60], Stanford Cars [38], Stanford Dogs [35] and CUB-200-2011 [91]. We use the pre-trained ResNet50 by Pytorch as the baseline on all four datasets. The original images are resized into 256×256, and then cropped to 224×224 as inputs for training and testing. The SGDM with a momentum of 0.9 is used to fine-tune the pre-trained ResNet50. The initial learning rate is 0.1 and it decays by 0.1 at the

Table 3.4: Testing accuracies (%) of ResNet50 on four fine-grained image classification.

Dataset	Aircraft					
Samples/GPU	2	4	8	16	32	64
EMA	80.63	84.72	84.30	83.60	82.77	82.35
EN	83.13	85.42	84.52	83.75	82.85	82.37
BSR	83.22	85.48	84.77	84.07	82.87	82.53
$\Delta_{\text{BSR}-\text{EMA}}$	↑2.59	↑0.76	↑0.47	↑0.47	↑0.10	↑0.18
Dataset	Stanford Cars					
Samples/GPU	2	4	8	16	32	64
EMA	83.90	89.45	89.78	89.08	88.67	88.03
EN	86.15	90.00	89.98	89.12	88.70	88.03
BSR	86.30	90.05	90.10	89.27	88.78	88.08
$\Delta_{\text{BSR}-\text{EMA}}$	↑2.40	↑0.60	↑0.32	↑0.19	↑0.11	↑0.05
Dataset	Stanford Dogs					
Samples/GPU	2	4	8	16	32	64
EMA	71.83	75.05	75.60	75.30	74.90	74.30
EN	72.72	75.83	76.05	75.48	74.98	74.32
BSR	73.98	76.20	76.35	75.90	75.22	74.50
$\Delta_{\text{BSR}-\text{EMA}}$	↑2.15	↑1.15	↑0.75	↑0.60	↑0.32	↑0.20
Dataset	CUB-200-2011					
Samples/GPU	2	4	8	16	32	64
EMA	72.60	77.37	77.60	77.10	75.95	75.58
EN	76.07	78.43	78.18	77.35	76.05	75.60
BSR	76.58	78.50	78.48	77.80	76.37	75.88
$\Delta_{\text{BSR}-\text{EMA}}$	↑3.98	↑1.13	↑0.88	↑0.70	↑0.42	↑0.30

50-th and 80-th epochs. We repeat the experiments 5 times and report the average accuracy.

Results: Table 3.4 reports the accuracies of ResNet50 on the four FGVC datasets with a different number of samples per GPU. We can have two findings. First, with pre-trained DNN models (on ImageNet), the best classification results are often obtained when the batch size is 8 or 4, instead of 32 or 64. This is because the batch statistics computed from the ImageNet dataset will be transferred to the current dataset via the pre-trained model so that a larger batch size may not bring many benefits. On the other hand, relatively smaller batch size is good for BN to perform the regularization function [87, 105, 101], as we explained in the experiments on

Table 3.5: Average Precision (AP) on COCO by using Faster-RCNN with ResNet50 backbone and FPN.

samples/GPU	2	4	8	16
EMA	34.2	35.9	36.9	37.4
EN	35.1	36.1	37.1	37.5
BSR	35.6	36.4	37.2	37.6
$\Delta_{\text{BSR}-\text{EMA}}$	$\uparrow$1.4	$\uparrow$0.5	$\uparrow$0.3	$\uparrow$0.2

CIFAR100/10. Therefore, for this task, Ghost-BN [23] can simulate multiple GPUs training (*e.g*, 8 or 16 GPUs) with a large training batch size, *e.g*, 128, to initially improve the generalization performance. And BSR can further boost the performance over Ghost-BN. This also demonstrates the practical value of BSR for the general optimization of a DNN.

3.4.4 Object Detection

Experimental Settings: Finally, we evaluate BSR on tasks other than classification, *e.g*, object detection. The training batch size for object detection is usually very small (*e.g*, 2 or 4) because of the high resolution of the input image and the limitation of memory. The Feature Pyramid Network (FPN) [47] and the Faster R-CNN [72] with ResNet50 backbone pre-trained on ImageNet are used as the detectors. The MMDetection [10] toolbox is used as the detection framework. The models have trained on COCO train2017 dataset (118K images) and evaluated on the COCO val2017 dataset (40K images) [48]. The learning rate schedule is 1X for all models. The BN layers are not frozen to compare the performance of BN inference methods. The overall training batch size is 16 and the number of samples per GPU is set to 2, 4, 8, 16, respectively.

Results: Table 3.5 gives the Average Precision (AP) results of Faster-RCNN on COCO. We can see that BSR outperforms EMA on all training batch sizes from 2 to 16. When the training batch size is 2, BSR can gain 1.4% AP over EMA. This

demonstrates that BSR can also boost the performance of BN on other tasks beyond classification.

3.5 Conclusion

To reduce the statistics disparity of BN between training and inference, we proposed a new inference approach of BN, namely batch statistics regression (BSR), which uses instance statistics to predict the batch statistics with a simple linear regression model. Without changing the training scheme of BN, the proposed BSR only needs to store four statistics in training with the negligible cost of computation and memory, and it updates the statistics with a simple online formulation. We also revealed that EMA is actually a special case of BSR. Experiments on image classification and object detection were conducted to evaluate the effectiveness of BSR. Compared with EMA, BSR is more accurate to approximate the batch statistics and it significantly improves the performance of BN when the training batch size is small. BSR is a good choice to replace the commonly used EMA for BN inference.

Chapter 4

Gradient Centralization: A Simple and Effective Optimization Technique for Deep Learning

Different from these existing methods that mostly operate on features or weights, in this chapter, we present a simple but effective optimization technique, namely gradient centralization (GC), which operates directly on gradients by centralizing the gradient vectors to have zero mean. GC can be viewed as a projected gradient descent method with a constrained loss function. We show that GC can regularize both the weight space and output feature space so that it can boost the generalization performance of DNNs. Moreover, GC improves the Lipschitzness of the loss function and its gradient so that the training process becomes more efficient and stable. GC is very simple to implement and can be easily embedded into existing gradient-based DNN optimizers with only one line of code. It can also be directly used to fine-tune the pre-trained DNNs. Our experiments on various applications, including general image classification, fine-grained image classification, detection, and segmentation, demonstrate that GC can consistently improve the performance of DNN learning.

4.1 Introduction

The broad success of deep learning largely owes to the recent advances on large-scale datasets [75], powerful computing resources (*e.g*, GPUs and TPUs), sophisticated network architectures [22, 25] and optimization algorithms [6, 37]. Among these factors, the efficient optimization techniques, such as stochastic gradient descent (SGD) with momentum [68], Adagrad [16] and Adam [37], make it possible to train very deep neural networks (DNNs) with a large-scale dataset and consequently deliver more powerful and robust DNN models in practice. The generalization performance of the trained DNN models as well as the efficiency of the training process depends essentially on the employed optimization techniques.

There are two major goals for a good DNN optimizer: accelerating the training process and improving the model generalization capability. The first goal aims to spend less time and cost to reach a good local minima, while the second goal aims to ensure that the learned DNN model can make accurate predictions on test data. A variety of optimization algorithms [68, 16, 37, 16, 37] have been proposed to achieve these goals. SGD [6, 7] and its extension SGD with momentum (SGDM) [68] are among the most commonly used ones. They update the parameters along the opposite direction of their gradients in one training step. Most of the current DNN optimization methods are based on SGD and improve SGD to better overcome the gradient vanishing or explosion problems. A few successful techniques have been proposed, such as weight initialization strategies [17, 21], efficient active functions (*e.g*, ReLU [63]), gradient clipping [65, 66], adaptive learning rate optimization algorithms [16, 37], and so on.

In addition to the above techniques, the sample/feature statistics such as mean and variance can also be used to normalize the network activations or weights to make the training process more stable. The representative methods operating on activations include batch normalization (BN) [32], instance normalization (IN) [88, 30], layer normalization (LN) [44] and group normalization (GN) [93]. Among them, BN is

the most widely used optimization technique which normalizes the features along the sample dimension in a mini-batch for training. BN smooths the optimization landscape [77] and it can speed up the training process and boost model generalization performance when proper batch size is used [105, 22]. However, BN works not very well when the training batch size is small, which limits its applications to memory-consuming tasks, such as object detection [20, 72], video classification [34, 2], *etc*.

Another line of statistics-based methods operates on weights. The representative ones include weight normalization (WN) [76, 27] and weight standardization (WS) [69]. These methods re-parameterize weights to restrict weight vectors during training. For example, WN decouples the length of weight vectors from their direction to accelerate the training of DNNs. WS uses the weight vectors' mean and variance to standardize them to have zero mean and unit variance. Similar to BN, WS can also smooth the loss landscape and speed up training. Nevertheless, such methods operating on weight vectors cannot directly adopt the pre-trained models (*e.g*, on ImageNet) because their weights may not meet the condition of zero mean and unit variance.

Different from the above techniques which operate on activations or weight vectors, we propose a very simple yet effective DNN optimization technique, namely gradient centralization (GC), which operates on the gradients of weight vectors. As illustrated in Fig. 4.1, GC simply centralizes the gradient vectors to have zero mean. It can be easily embedded into the current gradient-based optimization algorithms (*e.g*, SGDM [68], Adam [37]) using only one line of code. Though simple, GC demonstrates various desired properties, such as accelerating the training process, improving the generalization performance, and the compatibility for fine-tuning pre-trained models. The main contributions of this chapter are highlighted as follows:

- We propose a new general network optimization technique, namely gradient centralization (GC), which can not only smooth and accelerate the training process of DNN but also improve the model generalization performance.

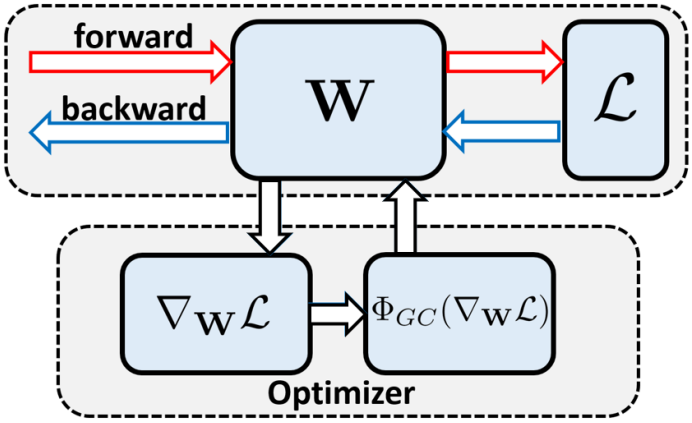


Figure 4.1: Sketch map for using gradient centralization (GC). $\mathbf{W}$ is the weight, $\mathcal{L}$ is the loss function, $\nabla_{\mathbf{W}}\mathcal{L}$ is the gradient of weight, and $\Phi_{GC}(\nabla_{\mathbf{W}}\mathcal{L})$ is the centralized gradient. It is very simple to embed GC into existing network optimizers by replacing $\nabla_{\mathbf{W}}\mathcal{L}$ with $\Phi_{GC}(\nabla_{\mathbf{W}}\mathcal{L})$.

- We analyze the theoretical properties of GC, and show that GC constrains the loss function by introducing a new constraint on the weight vector, which regularizes both the weight space and output feature space so that it can boost model generalization performance. Besides, the constrained loss function has better Lipschitzness than the original one, which makes the training process more stable and efficient.

Finally, we perform comprehensive experiments on various applications, including general image classification, fine-grained image classification, object detection and instance segmentation. The results demonstrate that GC can consistently improve the performance of learned DNN models in different applications. It is a simple, general and effective network optimization method.

4.2 Related Work

In order to accelerate the training and boost the generalization performance of DNNs, a variety of optimization techniques [32, 93, 76, 69, 68, 65] have been proposed to

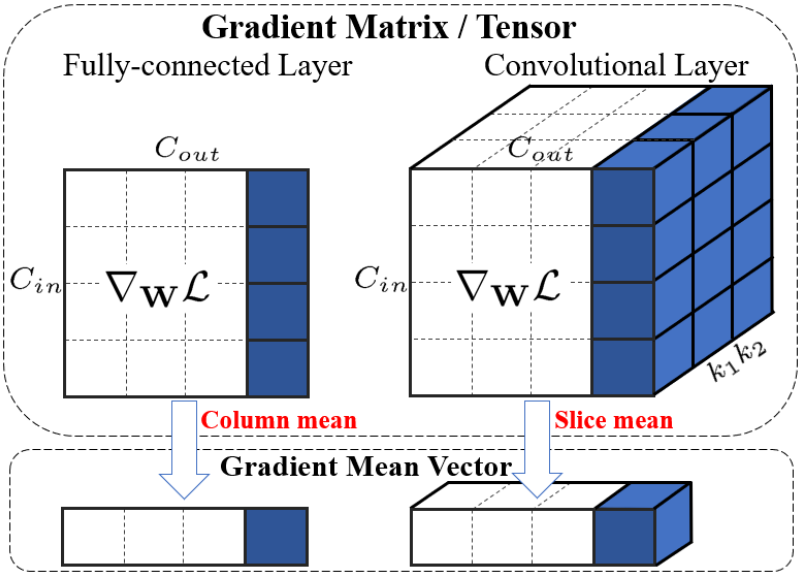


Figure 4.2: Illustration of the GC operation on gradient matrix/tensor of weights in the fully-connected layer (left) and convolutional layer (right). GC computes the column/slice mean of gradient matrix/tensor and centralizes each column/slice to have zero mean.

operate on activation, weight and gradient. In this section, we briefly review the related work from these three aspects.

Activation: The activation normalization layer has become a common setting in DNN, such as batch normalization (BN) [32] and group normalization (GN) [93]. BN was originally introduced to solve the internal covariate shift by normalizing the activations along the sample dimension. It allows higher learning rates [5], accelerates the training speed and improves the generalization accuracy [56, 77]. However, BN does not perform well when the training batch size is small, and GN is proposed to address this problem by normalizing the activations or feature maps in a divided group for each input sample. In addition, layer normalization (LN) [44] and instance normalization (IN) [88, 30] have been proposed for RNN and style transfer learning, respectively.

Weight: Weight normalization (WN) [76] re-parameterizes the weight vectors and decouples the length of a weight vector from its direction. It speeds up the convergence

of SGDM algorithm to a certain degree. Weight standardization (WS) [69] adopts the Z-score standardization to re-parameterize the weight vectors. Like BN, WS can also smooth the loss landscape and improve training speed. Besides, binarized DNN [70, 13, 12] quantifies the weight into binary values, which can improve the generalization capability for certain DNNs. However, a shortcoming of those methods operating on weights is that they cannot be directly used to fine-tune pre-trained models since the pre-trained weight may not meet their constraints. As a consequence, we have to design specific pre-training methods for them in order to fine-tune the model.

Gradient: A commonly used operation on gradient is to compute the momentum of gradient [68]. By using the momentum of gradient, SGDM accelerates SGD in the relevant direction and dampens oscillations. Besides, L_2 regularization based weight decay, which introduces L_2 regularization into the gradient of weight, has long been a standard trick to improve the generalization performance of DNNs [41, 106]. To make DNN training more stable and avoid gradient explosion, gradient clipping [65, 66, 1, 36] has been proposed to train a very deep DNNs. In addition, the projected gradient methods [19, 42] and Riemannian approach [11, 90] project the gradient on a subspace or a Riemannian manifold to regularize the learning of weights.

4.3 Gradient Centralization

4.3.1 Motivation

BN [32] is a powerful DNN optimization technique, which uses the first and second-order statistics to perform Z-score standardization on activations. It has been shown in [77] that BN reduces the Lipschitz constant of the loss function and makes the gradients more Lipschitz smooth so that the optimization landscape becomes smoother. WS [69] can also reduce the Lipschitzness of the loss function and smooth the opti-

mization landscape through Z-score standardization on weight vectors. BN and WS operate on activations and weight vectors, respectively, and they implicitly constrict the gradient of weights, which improves the Lipschitz property of loss for optimization.

Apart from operating on activation and weight, can we directly operate on the gradient to make the training process more effective and stable? One intuitive idea is that we use Z-score standardization to normalize gradients, like what has been done by BN and WS on activation and weight. Unfortunately, we found that normalizing gradient cannot improve the stability of training. Instead, we propose to compute the mean of gradient vectors and centralize the gradients to have zero mean. As we will see in the following development, the so-called gradient centralization (GC) method can have good Lipschitz property, smooth the DNN training and improve the model generalization performance.

4.3.2 Formulation of GC

For a FC layer or a Conv layer, suppose that we have obtained the gradient through backward propagation, then for a weight vector $\mathbf{w}_i$ whose gradient is $\nabla_{\mathbf{w}_i}\mathcal{L}$ ($i = 1, 2, ..., N$), the GC operator, denoted by Φ_{GC}, is defined as follows:

$$\Phi_{GC}(\nabla_{\mathbf{w}_i}\mathcal{L}) = \nabla_{\mathbf{w}_i}\mathcal{L} - \mu_{\nabla_{\mathbf{w}_i}\mathcal{L}} \tag{4.1}$$

where $\mu_{\nabla_{\mathbf{w}_i}\mathcal{L}} = \frac{1}{M}\sum_{j=1}^{M}\nabla_{\mathbf{W}_{i,j}}\mathcal{L}$. The formulation of GC is very simple. As shown in Fig. 4.2, we only need to compute the mean of the column vectors of the weight matrix, and then remove the mean from each column vector. We can also have a matrix formulation of Eq. (4.1):

$$\Phi_{GC}(\nabla_{\mathbf{W}}\mathcal{L}) = \mathbf{P}\nabla_{\mathbf{W}}\mathcal{L}, \qquad \mathbf{P} = \mathbf{I} - \mathbf{e}\mathbf{e}^T \tag{4.2}$$

The physical meaning of $\mathbf{P}$ will be explained later in Section 4.4.1. In practical

Algorithm 3: SGDM with Gradient Centralization

Input: Weight vector $\mathbf{w}^0$, step size η, momentum factor β, $\mathbf{m}^0$
Output: Weight vector $\mathbf{w}^{(T)}$

for *t=1:T* **do**
  $\mathbf{g}^t = \nabla_{\mathbf{w}^t}\mathcal{L}$
  $\hat{\mathbf{g}}^t = \Phi_{GC}(\mathbf{g}^t)$
  $\mathbf{m}^t = \beta\mathbf{m}^{t-1} + (1-\beta)\hat{\mathbf{g}}^t$
  $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\mathbf{m}_t$
end

implementation, we can directly remove the mean value from each weight vector to accomplish the GC operation. The computation is very simple and efficient.

4.3.3 Embedding of GC to SGDM/Adam

GC can be easily embedded into the current DNN optimization algorithms such as SGDM [68, 7] and Adam [37]. After obtaining the centralized gradient $\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})$, we can directly use it to update the weight matrix. **Algorithm 3** and **Algorithm 4** show how to embed GC into the two most popular optimization algorithms, SGDM and Adam, respectively. Moreover, if we want to use weight decay, we can set $\hat{\mathbf{g}}^t = \mathbf{P}(\mathbf{g}^t + \lambda\mathbf{w})$, where λ is the weight decay factor. It only needs to add one line of code into most existing DNN optimization algorithms to execute GC with negligible additional computational cost. For example, it costs only 0.6 sec extra training time in one epoch on CIFAR100 with ResNet50 model in our experiments (71 sec for one epoch).

4.4 Properties of GC

As we will see in the section of the experimental result, GC can accelerate the training process and improve the generalization performance of DNNs. In this section, we perform some theoretical analysis to explain why GC works.

Algorithm 4: Adam with Gradient Centralization

Input: Weight vector $\mathbf{w}^0$, step size η, β_1, β_2, ϵ, $\mathbf{m}^0$,$\mathbf{v}^0$

Output: Weight vector $\mathbf{w}^{(T)}$

for *t=1:T* **do**

- $\mathbf{g}^t = \nabla_{\mathbf{w}^t}\mathcal{L}$
- $\hat{\mathbf{g}}^t = \Phi_{GC}(\mathbf{g}^t)$
- $\mathbf{m}^t = \beta_1\mathbf{m}^{t-1} + (1-\beta_1)\hat{\mathbf{g}}^t$
- $\mathbf{v}^t = \beta_2\mathbf{v}^{t-1} + (1-\beta_2)\hat{\mathbf{g}}^t \odot \hat{\mathbf{g}}^t$
- $\hat{\mathbf{m}}^t = \mathbf{m}^t/(1-(\beta_1)^t)$
- $\hat{\mathbf{v}}^t = \mathbf{v}^t/(1-(\beta_2)^t)$
- $\mathbf{w}^{t+1} = \mathbf{w}^t - \eta\frac{\hat{\mathbf{m}}^t}{\sqrt{\hat{\mathbf{v}}^t+\epsilon}}$

end

4.4.1 Improving Generalization Performance

One important advantage of GC is that it can improve the generalization performance of DNNs. We explain this advantage from two aspects: weight space regularization and output feature space regularization.

Weight Space Regularization: Let's first explain the physical meaning of $\mathbf{P}$ in Eq.(4.2). Actually, it is easy to prove that:

$$\mathbf{P}^2 = \mathbf{P} = \mathbf{P}^T, \qquad \mathbf{e}^T\mathbf{P}\nabla_{\mathbf{W}}\mathcal{L} = 0. \tag{4.3}$$

The above equations show that $\mathbf{P}$ is the projection matrix for the hyperplane with normal vector $\mathbf{e}$ in weight space, and $\mathbf{P}\nabla_{\mathbf{W}}\mathcal{L}$ is the projected gradient.

The property of projected gradient has been investigated in some previous works [19, 42, 11, 90], which indicate that projecting the gradient of weight will constrict the weight space in a hyperplane or a Riemannian manifold. Similarly, the role of GC can also be viewed from the perspective of projected gradient descent.. We give a geometric illustration of SGD with GC in Fig. 4.3. As shown in Fig. 4.3, in the t-th step of SGD with GC, the gradient is first projected on the hyperplane determined by $\mathbf{e}^T(\mathbf{w}-\mathbf{w}^t) = 0$, where $\mathbf{w}^t$ is the weight vector in the t-th iteration, and then

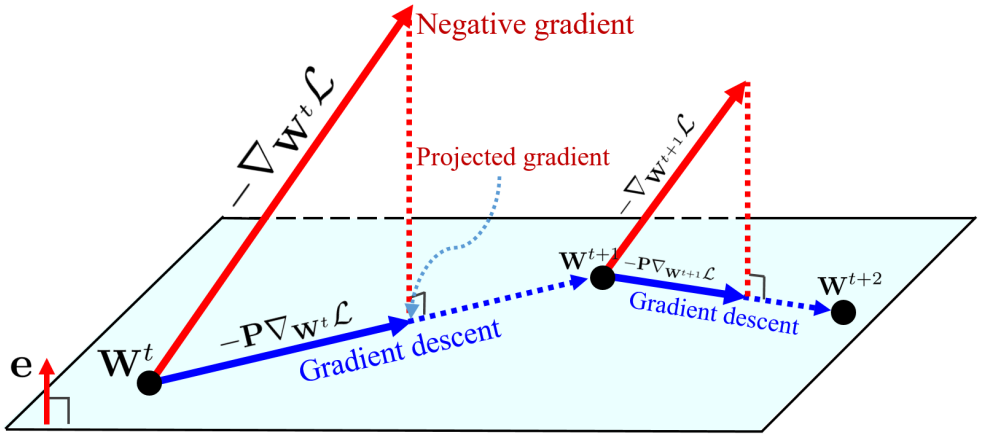


Figure 4.3: The geometrical interpretation of GC. The gradient is projected on a hyperplane $\mathbf{e}^T(\mathbf{w} - \mathbf{w}^t) = 0$, where the projected gradient is used to update the weight.

the weight is updated along the direction of projected gradient $-\mathbf{P}\nabla_{\mathbf{w}^t}\mathcal{L}$. From $\mathbf{e}^T(\mathbf{w} - \mathbf{w}^t) = 0$, we have $\mathbf{e}^T\mathbf{w}^{t+1} = \mathbf{e}^T\mathbf{w}^t = ... = \mathbf{e}^T\mathbf{w}^0$, *i.e*, $\mathbf{e}^T\mathbf{w}$ is a constant during training. Mathematically, the latent objective function w.r.t. one weight vector $\mathbf{w}$ can be written as follows:

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}), \quad s.t. \quad \mathbf{e}^T(\mathbf{w} - \mathbf{w}^0) = 0 \tag{4.4}$$

Clearly, this is a constrained optimization problem on weight vector $\mathbf{w}$. It regularizes the solution space of $\mathbf{w}$, reducing the possibility of overfitting on training data. As a result, GC can improve the generalization capability of trained DNN models, especially when the number of training samples is limited.

It is noted that WS [69] uses a constraint $\mathbf{e}^T\mathbf{w} = 0$ for weight optimization. It reparameterizes weights to meet this constraint. However, this constraint largely limits its practical applications because the initialized weight may not satisfy this constraint. For example, a pre-trained DNN on ImageNet usually cannot meet $\mathbf{e}^T\mathbf{w}^0 = 0$ for its initialized weight vectors. If we use WS to fine-tune this DNN, the advantages of pre-trained models will disappear. Therefore, we have to retrain the DNN on Ima-

geNet with WS before we fine-tune it. This is very cumbersome. Fortunately, the weight constraint of GC in Eq. (4.4) fits any initialization of weight, *e.g*, ImageNet pre-trained initialization, because it involves the initialized weight $\mathbf{w}^0$ into the constraint so that $\mathbf{e}^T(\mathbf{w}^0 - \mathbf{w}^0) = 0$ is always true. This greatly extends the applications of GC.

Output Feature Space Regularization: For SGD based algorithms, we have $\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha^t \mathbf{P}\nabla_{\mathbf{w}^t}\mathcal{L}$. It can be derived that $\mathbf{w}^t = \mathbf{w}^0 - \mathbf{P}\sum_{i=0}^{t-1}\alpha^{(i)}\nabla_{\mathbf{w}^{(i)}}\mathcal{L}$. For any input feature vector $\mathbf{x}$, we have the following theorem:

Theorem 4.1: *Suppose that SGD (or SGDM) with GC is used to update the weight vector* $\mathbf{w}$*, for any input feature vectors* $\mathbf{x}$ *and* $\mathbf{x} + \gamma\mathbf{1}$*, we have*

$$(\mathbf{w}^t)^T\mathbf{x} - (\mathbf{w}^t)^T(\mathbf{x} + \gamma\mathbf{1}) = \gamma\mathbf{1}^T\mathbf{w}^0 \tag{4.5}$$

where $\mathbf{w}^0$ *is the initial weight vector and* γ *is a scalar.*

Proof. First we show below a simple property of $\mathbf{P}$:

$$\mathbf{1}^T\mathbf{P} = \mathbf{1}^T(\mathbf{I} - \mathbf{e}\mathbf{e}^T) = \mathbf{1}^T - \frac{1}{M}\mathbf{1}^T\mathbf{1}\mathbf{1}^T = \mathbf{0}^T,$$

where M is the dimension of $\mathbf{e}$.

For each SGD step with GC, we have:

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \alpha^t\mathbf{P}\nabla_{\mathbf{w}^t}\mathcal{L}.$$

It can be easily derived that:

$$\mathbf{w}^t = \mathbf{w}^0 - \mathbf{P}\sum_{i=0}^{t-1}\alpha^{(i)}\nabla_{\mathbf{w}^{(i)}}\mathcal{L},$$

where t is the number of iterations. Then for the output activations of $\mathbf{x}$ and $\mathbf{x} + \gamma\mathbf{1}$, there is

$$
\begin{aligned}
(\mathbf{w}^t)^T\mathbf{x} - (\mathbf{w}^t)^T(\mathbf{x} + \gamma\mathbf{1}) &= \gamma\mathbf{1}^T\mathbf{w}^t \\
&= \gamma\mathbf{1}^T(\mathbf{w}^0 - \mathbf{P}\sum_{i=0}^{t-1}\alpha^{(i)}\nabla_{\mathbf{w}^{(i)}}\mathcal{L}) \\
&= \gamma\mathbf{1}^T\mathbf{w}^0 - \gamma\mathbf{1}^T\mathbf{P}\sum_{i=0}^{t-1}\alpha^{(i)}\nabla_{\mathbf{w}^{(i)}}\mathcal{L} \\
&= \gamma\mathbf{1}^T\mathbf{w}^0.
\end{aligned}
\tag{4.6}
$$

Therefore,

$$
(\mathbf{w}^t)^T\mathbf{x} - (\mathbf{w}^t)^T(\mathbf{x} + \gamma\mathbf{1}) = \gamma\mathbf{1}^T\mathbf{w}^0. \tag{4.7}
$$

For SGD with momentum, the conclusion is the same, because we can obtain a term $\gamma\mathbf{1}^T\mathbf{P}\sum_{i=0}^{t-1}\alpha^{(i)}\mathbf{m}^i$ in the third row of Eq.(4.6), where $\mathbf{m}^i$ is the momentum in the ith iteration, and this term is also equal to zero.

The proof is completed. ■

Theorem 4.4.1 indicates that a constant intensity change (*i.e*, $\gamma\mathbf{1}$) of an input feature causes a change of output activation; interestingly, this change is only related to γ and $\mathbf{1}^T\mathbf{w}^0$ but not the current weight vector $\mathbf{w}^t$. $\mathbf{1}^T\mathbf{w}^0$ is the scaled mean of the initial weight vector $\mathbf{w}^0$. In particular, if the mean of $\mathbf{w}^0$ is close to zero, then the output activation is not sensitive to the intensity change of input features, and the output feature space becomes more robust to the training sample variations.

Indeed, the mean of $\mathbf{w}^0$ is very close to zero by the commonly used weight initialization strategies, such as Xavier initialization [17], Kaiming initialization [21] and even ImageNet pre-trained weight initialization. Fig. 4.4 shows the absolute value (log scale) of the mean of weight vectors for Conv layers in ResNet50 with Kaiming normal initialization and ImageNet pre-trained weight initialization. We can see that the mean values of most weight vectors are very small and close to zero (less than e^{-7}). This ensures that if we train the DNN model with GC, the output features will not be sensitive to the variation of the intensity of input features. This property regularizes the output feature space and boosts the generalization of DNN training.

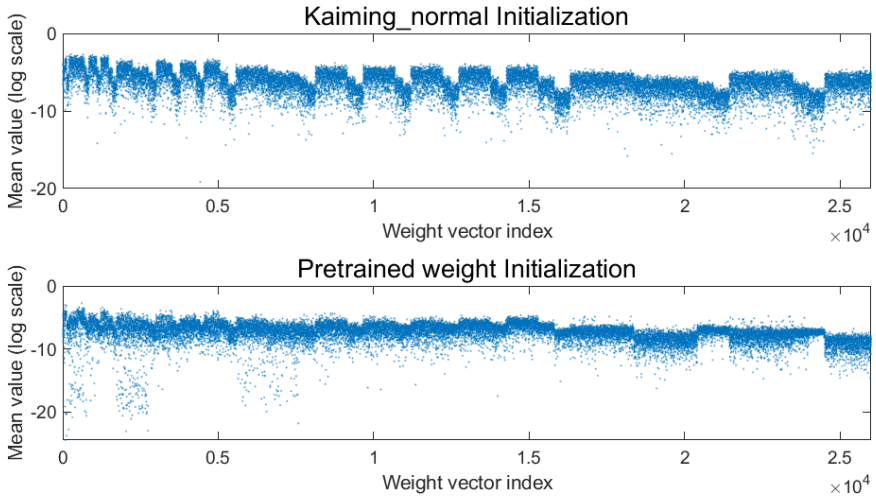


Figure 4.4: The absolute value (log scale) of the mean of weight vectors for convolution layers in ResNet50. The x-axis is the weight vector index. We plot the mean value of different convolution layers from left to right with the order from sallow to deep layers. Kaiming normal initialization [21] (top) and ImageNet pre-trained weight initialization (bottom) are employed here. We can see that the mean values are usually very small (less than e^{-7}) for most of the weight vectors.

4.4.2 Accelerating Training Process

Optimization Landscape Smoothing: It has been shown in [77, 69] that both BN and WS smooth the optimization landscape. Although BN and WS operate on activations and weights, they implicitly constrict the gradient of weights, making the gradient of weight more predictive and stable for fast training. Specifically, BN and WS use the gradient magnitude $||\nabla f(\mathbf{x})||_2$ to capture the Lipschitzness of function $f(\mathbf{x})$. For the loss and its gradients, $f(\mathbf{x})$ will be $\mathcal{L}$ and $\nabla_{\mathbf{w}}\mathcal{L}$, respectively, and $\mathbf{x}$ will be $\mathbf{w}$. The upper bounds of $||\nabla_{\mathbf{w}}\mathcal{L}||_2$ and $||\nabla^2_{\mathbf{w}}\mathcal{L}||_2$ ($\nabla^2_{\mathbf{w}}\mathcal{L}$ is the Hessian matrix of $\mathbf{w}$) have been given in [77, 69] to illustrate the optimization landscape smoothing property of BN and WS. Similar conclusion can be made for our proposed GC by comparing the Lipschitzness of original loss function $\mathcal{L}(\mathbf{w})$ with the constrained loss function in Eq. (4.4) and the Lipschitzness of their gradients. We have the following theorem:

Theorem 4.2: *Suppose $\nabla_{\mathbf{w}}\mathcal{L}$ is the gradient of loss function $\mathcal{L}$ w.r.t. weight vector*

$\mathbf{w}$. *With the* $\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})$ *defined in Eq.(4.2), we have the following conclusion for the loss function and its gradient, respectively:*

$$\begin{cases} ||\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})||_2 \leqslant ||\nabla_{\mathbf{w}}\mathcal{L}||_2, \\ ||\nabla_{\mathbf{w}}\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})||_2 \leqslant ||\nabla^2_{\mathbf{w}}\mathcal{L}||_2. \end{cases} \tag{4.8}$$

Proof. Because $\mathbf{e}$ is a unit vector, there is $\mathbf{e}^T\mathbf{e} = 1$. We can easily prove that:

$$\mathbf{P}^T\mathbf{P} = \mathbf{P}.$$

Then for $\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})$, we have:

$$\begin{aligned} ||\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})||_2^2 &= \Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})^T\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L}) \\ &= (\mathbf{P}\nabla_{\mathbf{w}}\mathcal{L})^T(\mathbf{P}\nabla_{\mathbf{w}}\mathcal{L}) \\ &= \nabla_{\mathbf{w}}\mathcal{L}^T\mathbf{P}^T\mathbf{P}\nabla_{\mathbf{w}}\mathcal{L} \\ &= \nabla_{\mathbf{w}}\mathcal{L}^T\mathbf{P}\nabla_{\mathbf{w}}\mathcal{L} \\ &= \nabla_{\mathbf{w}}\mathcal{L}^T(\mathbf{I} - \mathbf{e}\mathbf{e}^T)\nabla_{\mathbf{w}}\mathcal{L} \\ &= \nabla_{\mathbf{w}}\mathcal{L}^T\nabla_{\mathbf{w}}\mathcal{L} - \nabla_{\mathbf{w}}\mathcal{L}^T\mathbf{e}\mathbf{e}^T\nabla_{\mathbf{w}}\mathcal{L} \\ &= ||\nabla_{\mathbf{w}}\mathcal{L}||_2^2 - ||\mathbf{e}^T\nabla_{\mathbf{w}}\mathcal{L}||_2^2 \\ &\leqslant ||\nabla_{\mathbf{w}}\mathcal{L}||_2^2. \end{aligned} \tag{4.9}$$

For $\nabla_{\mathbf{w}}\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})$, we also have

$$\begin{aligned} ||\nabla\Phi_{GC}(\nabla_{\mathbf{w}}\mathcal{L})||_2^2 &= ||\mathbf{P}\nabla^2_{\mathbf{w}}\mathcal{L}||_2^2 \\ &= \nabla^2_{\mathbf{w}}\mathcal{L}^T\mathbf{P}^T\mathbf{P}\nabla^2_{\mathbf{w}}\mathcal{L} \\ &= \nabla^2_{\mathbf{w}}\mathcal{L}^T\mathbf{P}\nabla^2_{\mathbf{w}}\mathcal{L} \\ &= ||\nabla^2_{\mathbf{w}}\mathcal{L}||_2^2 - ||\mathbf{e}^T\nabla^2_{\mathbf{w}}\mathcal{L}||_2^2 \\ &\leqslant ||\nabla^2_{\mathbf{w}}\mathcal{L}||_2^2. \end{aligned} \tag{4.10}$$

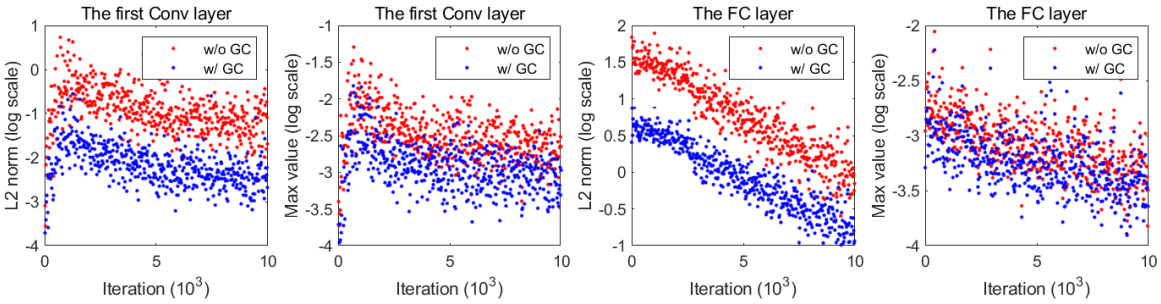


Figure 4.5: The L_2 norm (log scale) and max value (log scale) of gradient matrix or tensor vs. iterations. ResNet50 trained on CIFAR100 is used as the DNN model here. The left two sub-figures show the results on the first Conv layer and the right two show the FC layer. The red points represent the results of training without GC and the blue points represent the results with GC. We can see that GC largely reduces the L_2 norm and max value of gradient.

The proof is completed. ■

Theorem 4.4.2 shows that for the loss function $\mathcal{L}$ and its gradient $\nabla_{\mathbf{w}}\mathcal{L}$, the constrained loss function in Eq. (4.4) by GC leads to a better Lipschitzness than the original loss function so that the optimization landscape becomes smoother. This means that GC has similar advantages to BN and WS on accelerating training. A good Lipschitzness on gradient implies that the gradients used in training are more predictive and well-behaved so that the optimization landscape can be smoother for faster and more effective training.

Gradient Explosion Suppression: Another benefit of GC for DNN training is that GC can avoid gradient explosion and make training more stable. This property is similar to gradient clipping [65, 66, 36, 1]. Too large gradients will make the weights change abruptly during training so that the loss may severely oscillate and be hard to converge. It has been shown that gradient clipping can suppress large gradients so that the training can be more stable and faster [65, 66]. There are two popular gradient clipping approaches: element-wise value clipping [65, 36] and norm clipping [66, 1], which apply thresholding to element-wise value and gradient norm to gradient matrix, respectively. In order to investigate the influence of GC on clipping

gradient, in Fig. 4.5 we plot the max value and L_2 norm of gradient matrix of the first convolutional layer and the fully-connected layer in ResNet50 (trained on CIFAR100) with and without GC. It can be seen that both the max value and the L_2 norm of the gradient matrix become smaller by using GC in training. This is in accordance with our conclusion in Theorem 4.4.2 that GC can make the training process smoother and faster.

4.5 Experimental Results

4.5.1 Setup of Experiments

Extensive experiments are performed to validate the effectiveness of GC. To make the results as comprehensive and clear as possible, we arrange the experiments as follows:

- We start from experiments on the Mini-ImageNet dataset [89] to demonstrate that GC can accelerate the DNN training process and improve the model generalization performance. We also evaluate the combinations of GC with BN and WS to show that GC can improve them for DNN optimization.

- We then use the CIFAR100 dataset [39] to evaluate GC with various DNN optimizers (*e.g*, SGDM, Adam, Adagrad), various DNN architectures (*e.g*, ResNet, DenseNet, VGG), and and different hyper-parameters.

- We then perform experiments on ImageNet [75] to demonstrate that GC also works well on large-scale image classification, and show that GC can also work well with normalization methods other than BN, such as GN.

- We consequently perform experiments on four fine-grained image classification datasets (FGVC Aircraft [60], Stanford Cars [38], Stanford Dogs [35] and CUB-200-2011 [91]) to show that GC can be directly adopted to fine-tune the pre-

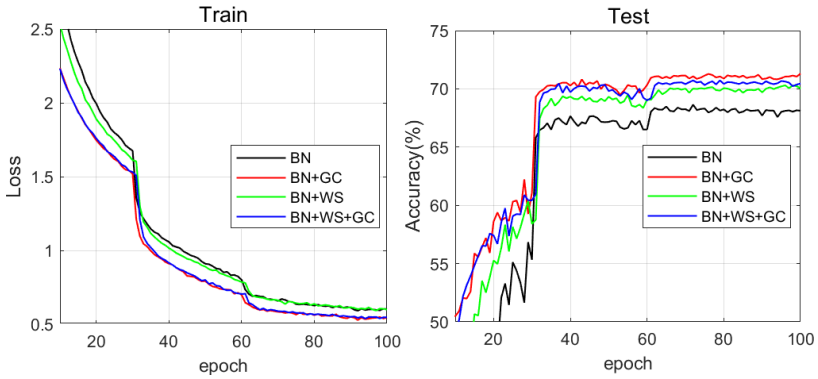


Figure 4.6: Training loss (left) and testing accuracy (right) curves vs. training epoch on the Mini-ImageNet. The ResNet50 is used as the DNN model. The compared optimization techniques include BN, BN+GC, BN+WS and BN+WS+GC.

trained DNN models and improve them.

- At last, we perform experiments on the COCO dataset [48] to show that GC also works well on other tasks such as object detection and segmentation.

GC can be applied to either Conv layer or FC layer, or both of them. In all of our following experiments, if not specified, we always apply GC to both Conv and FC layers. Except for Section 4.5.3 where we embed GC into different DNN optimizers for the test, in all other sections, we embed GC into SGDM for experiments, and the momentum is set to 0.9. All experiments are conducted under the Pytorch 1.3 framework and the GPUs are NVIDIA Tesla P100.

We would like to stress that no additional hyper-parameter is introduced in our GC method. Only one line of code is needed to embed GC into the existing optimizers while keeping all other settings remain unchanged. We compare the performances of DNN models trained with and without GC to validate the effectiveness of GC.

4.5.2 Results on Mini-Imagenet

Mini-ImageNet [89] is a subset of the ImageNet dataset [75], which was originally proposed for few-shot learning. We use the train/test splits provided by [71, 33]. It

Table 4.1: Testing accuracies of different DNN models on CIFAR100

Model	R18	R101	X29	V11	D121
w/o GC	76.87±0.26	78.82± 0.42	79.70±0.30	70.94± 0.34	79.31±0.33
w/ GC	**78.82±0.31**	**80.21±0.31**	**80.53±0.33**	**71.69±0.37**	**79.68±0.40**
Δ	↑1.95	↑1.39	↑0.83	↑0.75	↑0.37

Table 4.2: Testing accuracies of different optimizers on CIFAR100

Algorithm	SGDM	Adam	Adagrad	SGDW	AdamW
w/o GC	78.23±0.42	71.64±0.56	70.34 ±0.31	74.02±0.27	74.12±0.42
w/ GC	**79.14±0.33**	**72.80±0.62**	**71.58±0.37**	**76.82±0.29**	**75.07±0.37**
Δ	↑0.91	↑1.16	↑1.24	↑2.80	↑0.95

consists of 100 classes and each class has 500 images for training and 100 images for testing. The image resolution is 84×84. We resize the images into 224×224, which is the standard ImageNet training input size. The DNN we used here is ResNet50, which is trained on 4 GPUs with batch size 128. Other settings are the same as training ImageNet. We repeat the experiments 10 times and report the average results over the 10 runs.

BN, WS and GC operate on activations, weights and gradients, respectively, and they can be used together to train DNNs. Actually, it is necessary to normalize the feature space by methods such as BN; otherwise, the model is hard to be well trained. Therefore, we evaluate four combinations here: BN, BN+GC, BN+WS and BN+WS+GC. The optimizer is SGDM with a momentum 0.9. Fig. 4.6 presents the training loss and testing accuracy curves of these four combinations. Compared with BN, the training loss of BN+GC decreases much faster and the testing accuracy increases more rapidly. For both BN and BN+WS, GC can further speed up their training speed. Moreover, we can see that BN+GC achieves the highest testing accuracy, validating that GC can accelerate training and enhance the generalization performance simultaneously.

4.5.3 Experiments on CIFAR100

CIFAR100 [39] consists of 50K training images and 10K testing images from 100 classes. The size of the input image is 32×32. Since the image resolution is small, we found that applying GC to the Conv layer is good enough on this dataset. All DNN models are trained for 200 epochs using one GPU with batch size 128. The experiments are repeated 10 times and the results are reported in mean $\pm$ std format.

Different Networks: We testify GC on different DNN architectures, including ResNet18 (R18), ResNet101 (R101) [22], ResNeXt29 4x64d (X29) [95], VGG11 (V11) [82] and DenseNet121 (D121) [25]. SGDM is used as the network optimizer. The weight decay is set to 0.0005. The initial learning rate is 0.1 and it is multiplied by 0.1 for every 60 epochs. Table 4.1 shows the testing accuracies of these DNNs. It can be seen that the performance of all DNNs is improved by GC, which verifies that GC is a general optimization technique for different DNN architectures.

Different Optimizers: We embed GC into different DNN optimizers, including SGDM [68], Adagrad [16], Adam [37], SGDW and AdamW [51], to test their performance. SGDW and AdamW optimizers directly apply weight decay on weight, instead of using L_2 weight decay regularization. Weight decay is set to 0.001 for SGDW and AdamW, and 0.0005 for other optimizers. The initial learning rate is set to 0.1, 0.01, and 0.001 for SGDM/SGDW, Adagrad, Adam/AdamW, respectively, and the learning rate is multiplied by 0.1 for every 60 epochs. The other hyper-parameters are set by their default settings on Pytorch. The DNN model used here is ResNet50. Table 4.2 shows the testing accuracies. It can be seen that GC boosts the generalization performance of all five optimizers. It is also found that adaptive learning rate based algorithms Adagrad and Adam have poor generalization performance on CIFAR100, while GC can improve their performance by $> 0.9\%$.

Different Hyper-parameter Settings: In order to illustrate that GC can achieve consistent improvement with different hyper-parameters, we present the results of GC

Table 4.3: Testing accuracies of different weight decay on CIFAR100 with ResNet50.

Weight decay	0	$1e^{-4}$	$2e^{-4}$	$5e^{-4}$	$1e^{-3}$
w/o GC	71.62±0.31	73.91±0.35	75.57±0.33	78.23±0.42	77.43±0.30
w/ GC	**72.83±0.29**	**76.56±0.31**	**77.62±0.37**	**79.14±0.33**	**78.10±0.36**

Table 4.4: Testing accuracies of different learning rates on CIFAR100 with ResNet50 for SGDM and Adam.

Algorithm	SGDM	SGDM	SGDM	Adam	Adam	Adam
Learning rate	0.05	0.1	0.2	0.0005	0.001	0.0015
w/o GC	76.81±0.27	78.23±0.42	76.53±0.32	73.88±0.46	71.64±0.56	**70.63±0.44**
w/ GC	**78.12±0.33**	**79.14±0.33**	**77.71±0.35**	**74.32±0.55**	**72.80±0.62**	**71.22±0.49**

with different settings of weight decay and learning rates on the CIFAR100 dataset. ResNet50 is used as the backbone. Table 4.3 shows the testing accuracies with different settings of weight decay, including 0, $1e^{-4}$, $2e^{-4}$, $5e^{-4}$ and $1e^{-3}$. The optimizer is SGDM with learning rate 0.1. It can be seen that the performance of weight decay is consistently improved by GC. Table 4.4 shows the testing accuracies with different learning rates for SGDM and Adam. For SGDM, the learning rates are 0.05, 0.1 and 0.2, and for Adam, the learning rates are 0.0005, 0.001 and 0.0015. The weight decay is set to $5e^{-4}$. Other settings are the same as those in the manuscript. We can see that GC consistently improves performance.

4.5.4 Results on ImageNet

We then evaluate GC on the large-scale ImageNet dataset [75] which consists of 1.28 million images for training and 50K images for validation from 1000 categories. We use the common training settings and embed GC to SGDM on the Conv layer. The ResNet50 and ResNet101 are used as the backbone networks. For the former, we use 4 GPUs with batch size 64 per GPU, and for the latter, we use 8 GPUs with batch size 32 per GPU.

We evaluate four models here: ResNet50 with BN (R50BN), ResNet50 with GN (R50GN), ResNet101 with BN (R101BN) and ResNet101 with GN (R101GN). Table

Table 4.5: Top-1 error rates on ImageNet w/o GC and w/ GC.

Datesets	R50BN	R50GN	R101BN	R101GN
w/o GC	23.71	24.50	22.37	23.34
w/ GC	**23.21**	**23.53**	**21.82**	**22.14**
Δ	$\uparrow$0.50	$\uparrow$0.97	$\uparrow$0.55	$\uparrow$1.20

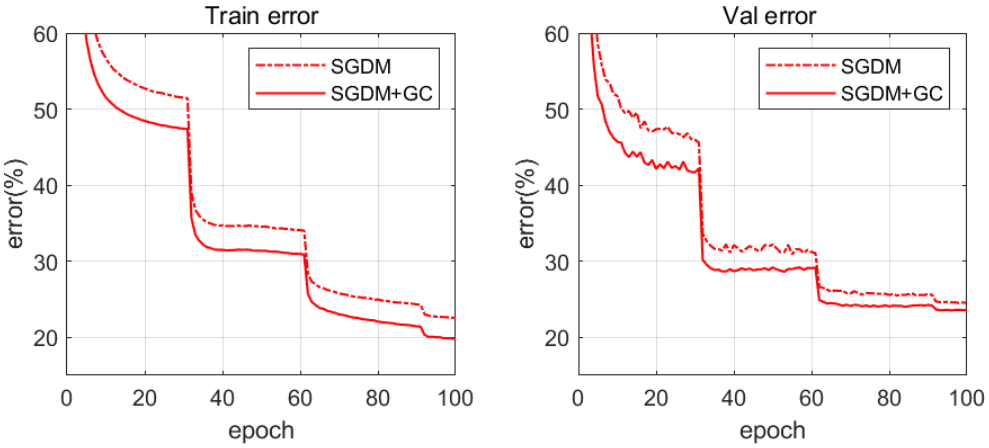


Figure 4.7: Training error (left) and validation error (right) curves vs. training epoch on ImageNet. The DNN model is ResNet50 with GN.

4.5 shows the final Top-1 errors of these four DNN models trained with and without GC. We can see that GC can improve the performance by 0.5% $\sim$ 1.2% on ImageNet. Fig. 4.7 plots the training and validation error curves of ResNet50 (GN is used for feature normalization). We can see that GC can largely speed up the training with GN.

4.5.5 Results on Fine-grained Image Classification

In order to show that GC can also work with the pre-trained models, we conduct experiments on four challenging fine-grained image classification datasets, including FGVC Aircraft [60], Stanford Cars [38], Stanford Dogs [35] and CUB-200-2011 [91]. The detailed statistics of these four datasets are summarized in Table 4.6. We use the official pre-trained ResNet50 provided by Pytorch as the baseline DNN for all these four datasets. The original images are resized into 512 $\times$ 512 and we crop the

Table 4.6: The statistics of fine-grained datasets used in this chapter.

Datasets	#Category	#Training	#Testing
FGVC Aircraft	100	6,667	3,333
Stanford Cars	196	8,144	8,041
Stanford Dogs	120	12,000	8,580
CUB-200-2011	200	5,994	5,794

Table 4.7: Testing accuracies on the four fine-grained image classification datasets.

Datesets	FGVC Aircraft	Stanford Cars	Stanford Dogs	CUB-200-2011
w/o GC	86.62±0.31	88.66±0.22	76.16±0.25	82.07±0.26
w/ GC	**87.77±0.27**	**90.03±0.26**	**78.23±0.24**	**83.40±0.30**
Δ	↑1.15	↑1.37	↑2.07	↑1.33

center region with 448 × 448 as input for both training and testing. The models are pre-trained on ImageNet. We use SGDM with a momentum of 0.9 to fine-tune ResNet50 for 100 epochs on 4 GPUs with batch size 256. The initial learning rate is 0.1 for the last FC layer and 0.01 for all pre-trained Conv layers. The learning rate is multiplied by 0.1 at the 50th and 80th epochs. Please note that our goal is to validate the effectiveness of GC but not to push state-of-the-art results, so we only use simple training tricks. We repeat the experiments 10 times and report the result in mean ± std format.

Fig. 4.8 shows the training and testing accuracies of SGDM and SGDM+GC for the first 40 epochs on the four fine-grained image classification datasets. Table 4.7 shows the final testing accuracies. We can see that both the training and testing accuracies of SGDM are improved by GC. For the final classification accuracy, GC improves SGDM by 1.1% ∼ 2.1% on these four datasets. This sufficiently demonstrates the effectiveness of GC on fine-tuning pre-trained models.

4.5.6 Object Detection and Segmentation

Finally, we evaluate GC on object detection and segmentation tasks to show that GC can also be applied to more tasks beyond image classification. The models are pre-

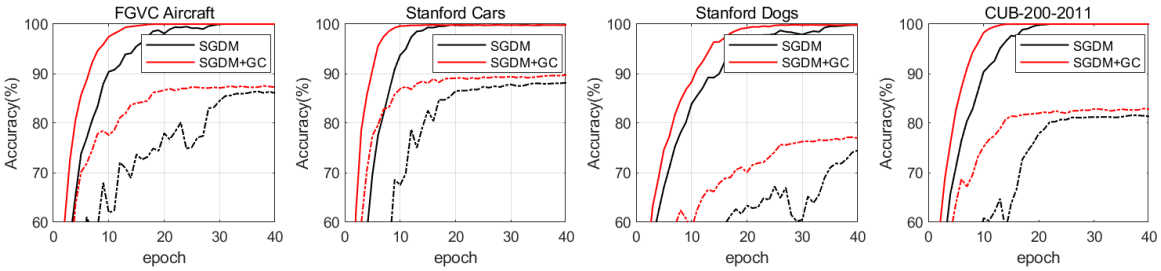


Figure 4.8: Training accuracy (solid line) and testing accuracy (dotted line) curves vs. training epoch on four fine-grained image classification datasets.

Table 4.8: Detection results on COCO by using Faster-RCNN and FPN with various backbone models.

Backbone	Method	AP	$AP_{.5}$	$AP_{.75}$	Backbone	AP	$AP_{.5}$	$AP_{.75}$
	w/o GC	36.4	58.4	39.1		40.1	62.0	43.8
R50	w/ GC	37.0	59.0	40.2	X101-32x4d	40.7	62.7	43.9
	Δ	↑0.6	↑0.6	↑ 1.1		↑0.6	↑0.7	↑0.1
	w/o GC	38.5	60.3	41.6		41.3	63.3	45.2
R101	w/ GC	38.9	60.8	42.2	X101-64x4d	41.6	63.8	45.4
	Δ	↑0.4	↑0.5	↑ 0.6		↑0.3	↑0.5	↑0.2

trained on ImageNet. The training batch size for object detection and segmentation is usually very small (*e.g*, 1 or 2) because of the high resolution of the input image. Therefore, the BN layer is usually frozen [22] and the benefits from BN cannot be enjoyed during training. One alternative is to use GN instead. The models are trained on COCO *train*2017 dataset (118K images) and evaluated on COCO *val*2017 dataset (40K images) [48]. COCO dataset can be used for multiple tasks, including image classification, object detection, semantic segmentation, and instance segmentation.

We use the MMDetection [10] toolbox, which contains comprehensive models on object detection and segmentation tasks, as the detection framework. The official implementations and settings are used for all experiments. All the pre-trained models are provided from their official websites, and we fine-tune them on COCO *train*2017 set with 8 GPUs and 2 images per GPU. The optimizers are SGDM and SGDM+GC. The backbone networks include ResNet50 (R50), ResNet101 (R101), ResNeXt101-32x4d (X101-32x4d), ResNeXt101-64x4d (X101-32x4d). The Feature Pyramid Net-

Table 4.9: Detection and segmentation results on COCO by using Mask-RCNN and FPN with various backbone models.

Backbone	Method	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
R50	w/o GC	37.4	59.0	40.6	34.1	55.5	36.1
	w/ GC	37.9	59.6	41.2	34.7	56.1	37.0
	Δ	↑0.5	↑0.6	↑ 0.6	↑0.6	↑0.6	↑0.9
R101	w/o GC	39.4	60.9	43.3	35.9	57.7	38.4
	w/ GC	40.0	61.5	43.7	36.2	58.1	38.7
	Δ	↑0.6	↑0.6	↑ 0.4	↑0.3	↑0.4	↑0.3
X101-32x4d	w/o GC	41.1	62.8	45.0	37.1	59.4	39.8
	w/ GC	41.6	63.1	45.5	37.4	59.8	39.9
	Δ	↑0.5	↑0.3	↑ 0.5	↑ 0.3	↑0.4	↑0.1
X101-64x4d	w/o GC	42.1	63.8	46.3	38.0	60.6	40.9
	w/ GC	42.8	64.5	46.8	38.4	61.0	41.1
	Δ	↑0.7	↑0.7	↑ 0.5	↑0.4	↑0.4	↑0.2
R50 (4c1f)	w/o GC	37.5	58.2	41.0	33.9	55.0	36.1
	w/ GC	38.4	59.5	41.8	34.6	55.9	36.7
	Δ	↑0.9	↑1.3	↑ 0.8	↑0.7	↑0.9	↑0.6
R101GN	w/o GC	41.1	61.7	44.9	36.9	58.7	39.3
	w/ GC	41.7	62.3	45.3	37.4	59.3	40.3
	Δ	↑0.6	↑0.6	↑ 0.4	↑0.5	↑0.6	↑1.0
R50GN+WS	w/o GC	40.0	60.7	43.6	36.1	57.8	38.6
	w/ GC	40.6	61.3	43.9	36.6	58.2	39.1
	Δ	↑0.6	↑0.6	↑ 0.3	↑0.5	↑0.4	↑0.5

work (FPN) [47] is also used. The learning rate schedule is $1X$ for both Faster R-CNN [72] and Mask R-CNN [20], except R50 with GN and R101 with GN, which use $2X$ learning rate schedule.

Tabel 4.8 shows the Average Precision (AP) results of Faster R-CNN. We can see that all the backbone networks trained with GC can achieve a performance gain about 0.3% $\sim$ 0.6% on object detection. Tabel 4.9 presents the Average Precision for bounding box (AP^b) and instance segmentation (AP^m). It can be seen that the AP^b increases by 0.5% $\sim$ 0.9% for object detection task and the AP^m increases by 0.3% $\sim$ 0.7% for instance segmentation task. Moreover, we find that if a 4conv1fc bounding box head, like R50 (4c1f), is used, the performance can increase more by GC. And GC can also boost the performance of GN (see R101GN) and improve the performance of WS (see R50GN+WS). Overall, we can see that GC boosts the generalization performance of all evaluated models. This demonstrates that it is a simple yet effective optimization technique, which is general to many tasks beyond

image classification.

4.6 Conclusions

How to efficiently and effectively optimize a DNN is one of the key issues in deep learning research. Previous methods such as batch normalization (BN) and weight standardization (WS) mostly operate on network activations or weights to improve DNN training. We proposed a different approach which operates directly on gradients. Specifically, we removed the mean from the gradient vectors and centralized them to have zero mean. The so-called Gradient Centralization (GC) method demonstrated several desired properties of deep network optimization. We showed that GC actually improves the loss function with a constraint on weight vectors, which regularizes both weight space and output feature space. We also showed that this constrained loss function has better Lipschitzness than the original one so that it has a smoother optimization landscape. Comprehensive experiments were performed and the results demonstrated that GC can be well applied to different tasks with different optimizers and network architectures, improving their training efficiency and generalization performance.

Chapter 5

Training Deep Neural Networks with Feature-based Gradient Descent

Most existing deep neural network (DNN) optimizers perform gradient descent on weight to minimize the loss. However, we find that performing weight gradient descent tends to update the features into a low dimensional space, which reduces the feature learning efficacy. To address this problem, in this chapter, we propose a new DNN optimizer, named Feature Stochastic Gradient Descent (FSGD), to approximate the desired feature outputs with one-step gradient descent for linear layers. FSGD only needs to store an additional second-order statistic matrix of the input features and use its inverse to adjust the gradient descent of weight. FSGD improves the singularity of feature space and enhances feature learning efficacy. We also show that FSGD has a close link to back-matching and feature whitening. Experimental results on CIFAR100/10, ImageNet and COCO demonstrate the superior performance of FSGD to state-of-the-art DNN optimizers.

5.1 Introduction

Deep neural networks (DNNs) have recently achieved a great success in many computer vision applications, including image classification [22], object detection [72, 20], image and video segmentation [72, 20] and image restoration [109], etc. The optimization of DNNs, however, is highly non-convex, making it difficult to find a favorable local minimum. The development of optimization techniques is thus crucial to train a good DNN model from a large amount of data. One class of commonly used DNN optimizers are the stochastic gradient descent (SGD) method [6, 7] and its variants [68, 37], which iteratively update the parameters along the opposite direction of their gradients obtained by the back-propagation (BP) algorithm [74].

Most existing DNN optimizers are designed and investigated in the weight space, including SGD with momentum (SGDM) [68], Adam [37], Radam [49] and Adabelief [115]. However, such weight gradient descent methods cannot directly ensure that the final image features will lie in the desired space. Actually, we find that when the input features of a linear layer are correlated, the weight gradient descent tends to update the features in a low dimensional space, which reduces the efficacy of information prorogation and feature update, especially in the early stages of DNN training. Denote by $\mathbf{A}$ the output feature of a linear layer, in Figure 5.1 we plot the eigenvalue distribution of feature covariance $\mathbf{A}\mathbf{A}^T$ in the first Conv layer and the FC layer of ResNet18 trained by SGDM on CIFAR100 after one epoch. We can see that the eigenvalues of covariance $\mathbf{A}\mathbf{A}^T$ decay rapidly, which means that most of the energy of features trained by SGDM concentrates in a low-dimensional subspace spanned by the principal eigenvectors of $\mathbf{A}\mathbf{A}^T$. This is however not favorable to effectively update the features and improve their data representation power.

Some feature normalization and whitening methods have been proposed to make feature propagation more stable and effective in DNN optimization. Among them, Batch Normalization (BN) [32] is the most widely used one, which uses the mean

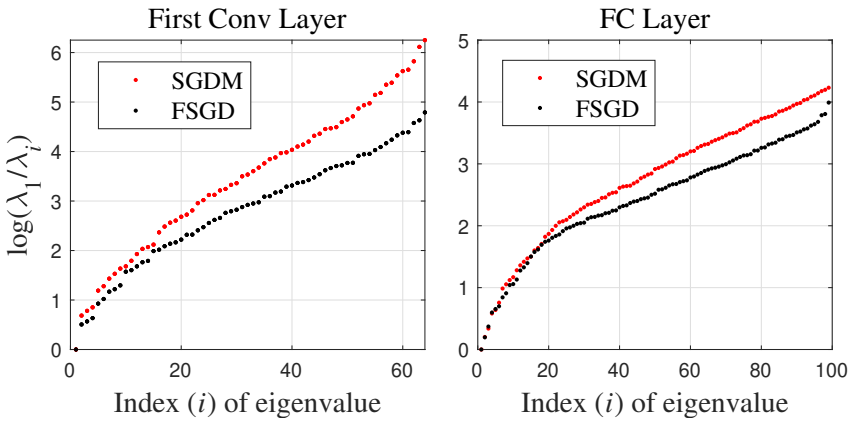


Figure 5.1: Illustration of the eigenvalue distribution of $\mathbf{A}\mathbf{A}^T$ in the first Conv layer and the FC layer of ResNet18 trained by SGDM and FSGD on CIFAR100 after one epoch, where $\mathbf{A}$ is the output feature. The Y-axis is $\log(\lambda_1/\lambda_i)$, where λ_i is the i^{th} eigenvalue of $\mathbf{A}\mathbf{A}^T$ in a descending order. The condition number of $\mathbf{A}\mathbf{A}^T$ is $1.7e^6$ and $6.1e^4$ for SGDM and FSGD, respectively, on the first Conv layer, and $1.7e^4$ and $9.7e^3$ on the FC layer. One can see that the $\mathbf{A}\mathbf{A}^T$ obtained by SGDM is much more singular than FSGD.

and variance of the intermediate features within a mini-batch to perform Z-score standardization. Though BN can speed up the training process and improve the model generalization performance, it ignores the feature correlation among different dimensions. Therefore, Decorrelated Batch Normalization (DBN) [28] and Iterative Normalization (IterNorm) [29] have been proposed to perform whitening on intermediate features, which validates that making proper use of statistics (*e.g*, covariance matrix) of intermediate features to find a more isotropic feature space can improve the learning of DNNs models. However, the feature whitening methods will cost significant computation and memory resources, and they need to redefine the forward and backward propagations by introducing the whitening module. Such limitations hinder the use of whitening methods in practice.

In this chapter, we put forward a new optimizer for DNNs, namely Feature SGD (FSGD), which takes the gradient descent on intermediate features into consideration. However, directly performing gradient descent on features is difficult to implement because it requires huge memory to store the intermediate features in training, and

the feature gradients of query samples are not available in inference either. Fortunately, we find that feature gradient descent can be achieved by updating weights. Specifically, we use the second-order statistic matrix of intermediate features to adjust the gradient of weight. An objective function is defined to relate the gradient descent on weight to the gradient descent on feature so that FSGD can be implemented by minimizing it. FSGD can be easily adopted into different linear layers in a DNN, including a fully-connected layer, convolutional layer and BN layer. As shown in Figure 5.1, the eigenvalues of feature covariance matrix $\mathbf{A}\mathbf{A}^T$ obtained by FSGD decay much more slowly than SGDM. The condition number of $\mathbf{A}\mathbf{A}^T$ is $1.7e^6$ and $6.1e^4$ for SGDM and FSGD, respectively, on the first Conv layer, and $1.7e^4$ and $9.7e^3$ on the FC layer. Clearly, FSGD improves much the singularity of matrix $\mathbf{A}\mathbf{A}^T$ over SGDM, which generally implies a better feature space for data representation. Our experimental results also show that FSGD performs significantly better than SGDM in terms of accuracy and generalization capability while costing only $20\% \sim 30\%$ additional training time. The main contributions of this chapter are highlighted as follows:

- We propose a practical feature gradient descent method (FSGD) to boost the feature space for the optimization of DNN, which only modifies the gradient of weight with the second-order statistics of the feature. The formulations of FSGD on various linear layers, such as FC layer, Conv layer, and Norm layer, are derived.

- A series of tricks of techniques and tricks, including the associated momentum and weight decay, statistics matrix computation, damping and gradient norm recovery, are proposed to improve the effectiveness and efficiency of FSGD.

- We investigate the relationship of FSGD between the existing optimization method for deep learning, *e.g*, the back-matching propagation and feature whitening to explain it from multiple views.

- We extend FSGD algorithm to other advanced optimizers, *i.e*, Adam to show that FSGD has favorable compatibility with existing deep learning optimizers.
- We have conducted extensive benchmark experiments to show the superiority of FSGD, including CIFAR100/CIFAR10, ImageNet, COCO and so on to testify the performance of FSGD.

This chapter is organized as follows: Section 5.2 reviews some related works, including optimization algorithm for deep learning and the normalization and whitening method. Section **??** gives the formulation and algorithm of our proposed method FSGD. Section 5.3 shows the relationship between FSGD and some existing optimization techniques. Section 5.4 reports the experimental results on a variety of tasks and benchmarks to substantiate the superiority of the proposed method. Discussions and concluding remarks are finally given.

5.2 Related Work

5.2.1 First-order Optimizers

Popular DNN optimizers include SGDM [68], Adagrad [16], Adam [37], RAdam [49], etc. SGDM [68] uses the momentum of gradient to accelerate gradient descent along relevant directions and dampens oscillations. It has been widely used in the high-level vision tasks of computer vision, *e.g*, image classification, object detection and so on. However, SGDM applies the same learning rate to all weights, which slows down its convergence speed. The adaptive learning rate methods are developed to allow each weight to have its own learning rate. For instance, Adagrad [16] adopts a larger gradient step for infrequent parameters and a smaller step for frequent ones. Similar works include RMSprop and Adadelta [104]. Adam [37] combines the adaptive learning rate technique with gradient momentum, largely stabilizing the training process.

Following Adam, RAdam [49] controls the variance of the adaptive learning rate in the early stage of training, and Adabelief [115] adjusts the step size by the belief in observed gradients, which achieves favorable performance. The adaptive learning rate methods can perform better than SGDM in some specific areas, such as natural language processing, image low-level vision and so on.

5.2.2 Second-order Optimizers

Besides first-order DNN optimizers, the second-order information has also been investigated to help the optimization of DNN recently. Computing with the full second-order information will need extensive memory and computation cost because the dimension of the parameter space in DNNs is usually very high (*e.g*, 10^7). Therefore, how to approach the second-order information practically is the key problem in deep learning. For instance, Adahessian [97] and Apollo [59] were proposed by updating only the diagonal elements of the Hessian matrix. Particularly, Adahessian considers only the diagonal elements of the Hessian matrix by using Hessian-free techniques, while Apollo simplifies the BFGS algorithm with only diagonal elements. Instead of the diagonal approach, the Kronecker Factored Approximation Curvature (KFAC) [61] approximates the natural gradient layer-wisely by using a block-diagonal version of the Fisher matrix, which adopts the statistics of intermediate features and their gradients to adjust the original gradient of weights. Nonetheless, in many computer vision tasks the first-order optimizers, such as SGDM and Adam, are more popularly used than second-order optimizers because of their simplicity and effectiveness.

5.2.3 Normalization and Whitening

Batch normalization (BN) [32] was originally introduced to address the internal covariate shift of DNNs by normalizing the activations along the sample dimension. It

has become a common layer in many DNNs, which allows higher learning rates [5], accelerates the training speed, and improves the generalization accuracy [56, 77]. Later on, layer normalization (LN) [44], instance normalization (IN) [88, 30] and group normalization (GN) [93] were proposed to perform normalization along other dimensions. Beyond standardization, the feature whitening methods take the feature correlation among different dimensions of features into consideration. For example, DBN [28] was proposed to conduct ZCA-whitening on feature across the channel dimension by eigen-decomposition and backpropagating the transformation. IterNorm [29] adopts Newton's iteration on DBN to approach the ZCA-whitening matrix more efficiently. Meanwhile, Network deconvolution (ND) [99] uses deconvolution filters to eliminate both pixel-wise and channel-wise correlations before the convolution layer. Generally speaking, the feature whitening methods can not only speed up training processing but also boost the generalization performance of DNNs [29, 99]. However, their drawbacks, including heavy extra computation and memory, inapplicable to pre-trained DNN models, additional parameter introduction, *etc*, make them can not be widely employed in real-world applications.

5.2.4 Motivation

Almost all existing DNN optimization methods focus on how to effectively perform weight gradient descent, *i.e*,

$$\mathbf{W}^{t+1} = \mathbf{W}^{t} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{t}}, \tag{5.1}$$

where η is the learning rate and t denotes the iteration. Though many works [68, 16, 37] on weight gradient descent have been reported, few of them have discussed how the update on weights affects the changing direction of intermediate features. Here we attempt to make some analysis on this problem to better understand the update of DNN features.

In order to simplify the problem, we only take one layer into consideration and fix the parameters of the other layers in a DNN. Considering one FC layer with input $\mathbf{X}$ and output $\mathbf{A}$, when we fix the parameters of the previous layers we have $\mathbf{X}^{t+1} = \mathbf{X}^t$, and with the fact that $\mathbf{A} = \mathbf{W}\mathbf{X}$, by right multiplying $\mathbf{X}^t$ on both sides of Eq. (5.1) and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t} = \frac{\partial \mathcal{L}}{\partial \mathbf{A}^t}\mathbf{X}^{t^T}$, we can easily derive that

$$\mathbf{A}^{t+1} = \mathbf{A}^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{A}^t}\mathbf{X}^{t^T}\mathbf{X}^t. \tag{5.2}$$

It can be clearly seen that the update of intermediate feature $\mathbf{A}$ is not exactly along its gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{A}}$ but along the direction determined by $\frac{\partial \mathcal{L}}{\partial \mathbf{A}^t}\mathbf{X}^{t^T}\mathbf{X}^t$, which is related to $\mathbf{X}^{t^T}\mathbf{X}^t$. One problem of this feature updating strategy is that when the input features $\mathbf{X}^t$ are highly correlated among different dimensions, the feature updating directions will be dominated by the principal eigenvectors of $\mathbf{X}^{t^T}\mathbf{X}^t$. As a result, the output features in $\mathbf{A}^{t+1}$ tend to fall into a lower dimensional subspace, reducing the efficacy of information propagation and feature learning. More discussions on the statistics of $\mathbf{A}^T\mathbf{A}$ have been made in Figure 5.1 and Section 5.1.

In order to obtain a more favorable feature space, a natural idea is that we can perform gradient descent directly on the intermediate features, *i.e*,

$$\mathbf{A}^{t+1} = \mathbf{A}^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{A}^t}. \tag{5.3}$$

However, there are some difficulties in directly implementing Eq. (5.3). First, it requires large memory to store the intermediate features of all samples in the training dataset at the same time for the next gradient descent iteration during training, which is usually impractical for large-scale datasets. Second, in the inference step, we cannot get the intermediate features of the query sample without the label and its gradients. Fortunately, in the following sections, we present a novel solution to address these difficulties and make the gradient descent on intermediate features in Eq. (5.3) practical.

Table 5.1: The updating formulas of FC, Conv and Norm layers in FSGD.

Layer type	Actual feature $\mathbf{A}^t(\mathbf{W})$	Target feature $\mathbf{A}^{t+1}$	Updating formula
FC layer	$\mathbf{W}\mathbf{X}^t$	$\mathbf{W}^t\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}$	$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{W}^t}\left(\mathbf{X}^t\mathbf{X}^{tT}\right)^{-1}$
Conv layer	$\mathbf{W} * \mathbf{X}^t$	$\mathbf{W}^t * \mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}$	$vec(\mathbf{W}^{t+1}) = vec(\mathbf{W}^t) - \eta\mathcal{U}_1(\frac{\partial\mathcal{L}}{\partial\mathbf{W}^t})\left(\mathfrak{X}\mathfrak{X}^T\right)^{-1}$
Norm layer	$\gamma\mathbf{X}^t + \beta$	$\gamma^t\mathbf{X}^t + \beta^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}$	$\begin{bmatrix}\gamma^{t+1}\\ \beta^{t+1}\end{bmatrix} = \begin{bmatrix}\gamma^t\\ \beta^t\end{bmatrix} - \eta\left(\begin{bmatrix}vec(\mathbf{X}^t)^T\\ \mathbf{1}^T\end{bmatrix}[vec(\mathbf{X}^t), \mathbf{1}]\right)^{-1}\begin{bmatrix}\frac{\partial\mathcal{L}}{\partial\gamma^t}\\ \frac{\partial\mathcal{L}}{\partial\beta^t}\end{bmatrix}$

5.2.5 Feature Gradient Descent

To achieve the feature gradient descent, we still update the weights of DNNs with the stochastic gradient descent. However, the difference with SGD is that we need to find a specific updating direction of the weight that can make the output features change along their gradient direction. To be specific, supposing $\mathbf{A}^{t+1}$ is the desired output feature of a linear layer after one-step gradient descent (see Eq. (5.3)), and $\mathbf{A}^t(\mathbf{W})$ is the actual output feature obtained by using weight $\mathbf{W}$, we hope $\mathbf{A}^t(\mathbf{W})$ is close to $\mathbf{A}^{t+1}$ under some measurement after updating the weight. If using L_2 Norm as the measurement, we can have the following objective function w.r.t$\mathbf{W}$ in the t-th step:

$$\mathbf{W}^{t+1} = \arg\min_{\mathbf{W}} ||\mathbf{A}^{t+1} - \mathbf{A}^t(\mathbf{W})||_2^2. \tag{5.4}$$

By solving Eq. (5.4), we can find an optimal weight $\mathbf{W}^{t+1}$ to approximate the desired feature in iteration t after one-step gradient descent. Eq. (5.4) can be viewed as the local objective function, which only works on one linear layer of a DNNs. For different layers, $\mathbf{A}^t(\mathbf{W})$ and $\mathbf{A}^{t+1}$ usually have different forms. As long as the layer has a linear operation w.r.tthe parameters, we can construct such a local objective function to update its parameters. The conventional linear layers in DNNs include the FC layer and Conv layer. Meanwhile, since the affine transformation in the Norm layer is also a linear operation w.r.tthe parameters, it can also adopt such a local objective function. We present the solutions of $\mathbf{W}^{t+1}$ for FC, Conv and Normalization layers as follows.

Fully-connected Layer

We first give the derivation of FSGD on the Fully-connected layer (FC layer). First, we need to define the actual output activation $\mathbf{A}^t(\mathbf{W})$ and $\mathbf{A}^{t+1}$ the target output activation. For the FC layer, they are obtained by $\mathbf{A}^t(\mathbf{W}) = \mathbf{W}\mathbf{X}^t$ and $\mathbf{A}^{t+1} = \mathbf{W}^t\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}$. We expect them to be as similar as possible. Therefore, by introducing them into Eq. (5.4), we have the following objective function:

$$\mathbf{W}^{t+1} = \arg\min_{\mathbf{W}} ||\mathbf{W}^t\mathbf{X}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t} - \mathbf{W}\mathbf{X}^t||_2^2. \tag{5.5}$$

We can see that it is a simple linear least square problem. Taking the derivative of Eq. (5.5) w.r.t. $\mathbf{W}$ and letting it be zero, we have

$$\mathbf{W}^t\mathbf{X}^t\mathbf{X}^{tT} - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}\mathbf{X}^{tT} - \mathbf{W}\mathbf{X}^t\mathbf{X}^{tT} = 0. \tag{5.6}$$

Note that $\frac{\partial\mathcal{L}}{\partial\mathbf{W}^t} = \frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}\mathbf{X}^{tT}$. Then by solving Eq. (5.6), we can get a closed-form solution for the weight updating rule of FC layer, which is

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{W}^t}(\mathbf{X}^t\mathbf{X}^{tT})^{-1}. \tag{5.7}$$

Compared with the updating formula of $\mathbf{W}$ in SGD (*i.e*, Eq. (5.1)), the updating formula in Eq. (5.7) needs to calculate the second-order statistic of input activation, *i.e*, $\mathbf{X}^t\mathbf{X}^{tT}$, and its inverse. The second-order statistic matrix can be viewed as a local Hessian [107] of the local objective function. According to Eq. (5.7), instead of storing all the intermediate features during training, what we need to store additionally is only the second-order statistic matrix of input activation. Therefore, in this case, It can be shown that we can update the weight along a modified gradient direction to achieve the feature gradient descent step. Although the actual output features do not go along the exact feature gradient direction, because its information has been embedded into the modified gradient direction of weight, its actual changing direction

after updating weight will be very close to the feature gradient direction.

Conv Layer

For the Conv layer, the actual output feature $\mathbf{A}^t(\mathbf{W})$ and the target output feature $\mathbf{A}^{t+1}$ are obtained by $\mathbf{A}^t(\mathbf{W}) = \mathbf{W} * \mathbf{X}^t$ and $\mathbf{A}^{t+1} = \mathbf{W}^t * \mathbf{X}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{A}^t}$. The derivation process is similar to the FC layer. The difference is that we need to unfold the convolution operation to matrix multiplication first. The convolution operation can be formulated as matrix multiplication with the *im2col* operation [99, 108], and then the Conv layer can be viewed as a 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 the mode 1 unfold operation of a tensor. The objective function Eq. (5.4) to be minimized becomes:

$$\min_{\mathbf{W}} ||\mathcal{U}_1(\mathbf{W}^t)\mathfrak{X}^t - \eta \cdot (\frac{\partial \mathcal{L}}{\partial \mathfrak{A}^t}) - \mathcal{U}_1(\mathbf{W})\mathfrak{X}^t||_2^2, \tag{5.8}$$

Taking the derivative of Eq. (5.8) w.r.t. $\mathcal{U}_1(\mathbf{W})$ and letting it be zero, we have

$$\mathcal{U}_1(\mathbf{W}^t)\mathfrak{X}^{tT}\mathfrak{X}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathfrak{A}^t}\mathfrak{X}^{tT} - \mathcal{U}_1(\mathbf{W})\mathfrak{X}^{tT}\mathfrak{X}^t = 0. \tag{5.9}$$

Note that $\mathcal{U}_1(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}) = \frac{\partial \mathcal{L}}{\partial \mathfrak{A}^t}\mathfrak{X}^{tT}$, and then we have the following weight updating formula:

$$\mathcal{U}_1(\mathbf{W}^{t+1}) = \mathcal{U}_1(\mathbf{W}^t) - \eta\mathcal{U}_1\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}\right)\left(\mathfrak{X}^{tT}\mathfrak{X}^t\right)^{-1}. \tag{5.10}$$

Therefore, similar to the FC layer, the updating formula of the Conv layer still needs to compute an additional second-order statistic of input feature, *i.e*, $\mathfrak{X}^{tT}\mathfrak{X}^t$, and its inverse. The original weight gradient is also modified with this second-order statistic. Instead of *im2col* operation [99, 108], this statistic can also be obtained efficiently from a convolution operation according to work in [113], which is a memory-saving method for the large kernel size.

Normalization Layer

The normalization layers usually have a channel-wise affine transformation, which is also a linear function. Suppose that the normalized feature is $\mathbf{X}^t$ and the parameters of affine transformation for one channel are γ and β. The actual output feature $\mathbf{A}^t(\gamma, \beta)$ and the target feature $\mathbf{A}^{t+1}$ are obtained by $\mathbf{A}^t(\gamma, \beta) = \gamma\mathbf{X}^t + \beta$ and $\mathbf{A}^{t+1} = \gamma^t\mathbf{X}^t + \beta^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}$. By introducing them into Eq. (5.4), we have the following objective function to be solved:

$$\min_{\gamma,\beta} ||[vec(\mathbf{X}^t), \mathbf{1}] \begin{bmatrix} \gamma^t \\ \beta^t \end{bmatrix} - \eta \cdot vec(\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}) - [vec(\mathbf{X}^t), \mathbf{1}] \begin{bmatrix} \gamma \\ \beta \end{bmatrix} ||_2^2. \tag{5.11}$$

where $vec(\cdot)$ is the Taking the derivative of Eq. (5.11) w.r.t. $\begin{bmatrix} \gamma \\ \beta \end{bmatrix}$ letting it be zero, we have

$$\begin{bmatrix} vec(\mathbf{X}^t)^T \\ \mathbf{1}^T \end{bmatrix} \left([vec(\mathbf{X}^t), \mathbf{1}] \begin{bmatrix} \gamma^t \\ \beta^t \end{bmatrix} - \eta \cdot vec(\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t}) - [vec(\mathbf{X}^t), \mathbf{1}] \begin{bmatrix} \gamma \\ \beta \end{bmatrix} \right) = 0. \tag{5.12}$$

Note that $\begin{bmatrix} \frac{\partial\mathcal{L}}{\partial\gamma^t} \\ \frac{\partial\mathcal{L}}{\partial\beta^t} \end{bmatrix} = \begin{bmatrix} vec(\mathbf{X}^t)^T \\ \mathbf{1}^T \end{bmatrix} vec(\frac{\partial\mathcal{L}}{\partial\mathbf{A}^t})$, and we can obtain the the updating rules for γ, β as follows:

$$\begin{bmatrix} \gamma^{t+1} \\ \beta^{t+1} \end{bmatrix} = \begin{bmatrix} \gamma^t \\ \beta^t \end{bmatrix} - \eta \left(\begin{bmatrix} vec(\mathbf{X}^t)^T \\ \mathbf{1}^T \end{bmatrix} [vec(\mathbf{X}^t), \mathbf{1}] \right)^{-1} \begin{bmatrix} \frac{\partial\mathcal{L}}{\partial\gamma^t} \\ \frac{\partial\mathcal{L}}{\partial\beta^t} \end{bmatrix}. \tag{5.13}$$

If the mean and the variance of $\mathbf{X}^t$ are zero and one, $\begin{bmatrix} vec(\mathbf{X}^t)^T \\ \mathbf{1}^T \end{bmatrix} [vec(\mathbf{X}^t), \mathbf{1}]$ will be an isotropic diagonal 2×2 matrix. If the mean and variance of $\mathbf{X}^t$ are zero and one, the second-order statistics will be an isotropic diagonal 2×2 matrix. For example, when BN and IN are used, the update rules in Eq. (5.13) for (γ, β) will degrade to

Table 5.2: The three ways to add momentum.

	Momentum	Final updates
I	$\mathbf{M}_{xx}^t = \alpha\mathbf{M}_{xx}^{t-1} + (1-\alpha)\mathbf{X}^t\mathbf{X}^{tT}$ $\mathbf{M}_G^t = \beta\mathbf{M}_G^{t-1} + (1-\beta)\mathbf{G}^t$	$\mathbf{M}_G^t(\mathbf{M}_{xx}^t + \epsilon\mathbf{I})^{-1}$
II	$\hat{\mathbf{G}}^t = \mathbf{G}^t(\mathbf{M}_{xx}^t + \epsilon\mathbf{I})^{-1}$ $\mathbf{M}_G^t = \beta\mathbf{M}_G^{t-1} + (1-\beta)\hat{\mathbf{G}}^t$	$\mathbf{M}_G^t$
III	$\mathbf{M}_{xx}^t = \alpha\mathbf{M}_{xx}^{t-1} + (1-\alpha)\mathbf{X}^t\mathbf{X}^{tT}$ $\hat{\mathbf{G}}^t = \mathbf{G}^t(\mathbf{M}_{xx}^t + \epsilon\mathbf{I})^{-1}$ $\mathbf{M}_G^t = \beta\mathbf{M}_G^{t-1} + (1-\beta)\hat{\mathbf{G}}^t$	$\mathbf{M}_G^t$

the case of SGD. However, for other normalization methods such as GN and LN, the mean and variance of each channel may not be zero and one, and Eq. (5.13) should be used to update (γ, β).

In Table 5.1, we conclude the solutions of $\mathbf{W}^{t+1}$ for the FC layer, Conv layer and Normalization layer.

5.2.6 Detailed Implementation

Damping

In practice, when the dimension of $\mathbf{X}^t\mathbf{X}^{tT}$ is high or the training batch size is small, it tends to be a singular matrix. In such a case, its condition number is too large, it would be unstable to calculate its inverse. To avoid this case, we should add an extra term $\epsilon\mathbf{I}$ to the statistic matrix, where ϵ is the damping parameter and $\mathbf{I}$ is an identity matrix. ϵ parameter is very essential to the final performance, and we find that a proper ϵ can significantly boost the training. Too large damping may lose the information of the statistics, while too small damping would not help improve its condition number which leads to unstable training. We will tune it in the experiment part.

Momentum

By employing the momentum of gradient, SGDM accelerates SGD in the relevant direction and dampens oscillations. Momentum has become a commonly used operation in SGD optimizers [86]. There are three ways to add momentum to our proposed FSGD algorithm, as shown in Table 5.2, where $\mathbf{G}$, $\mathbf{M}_{xx}^t$ and $\mathbf{M}_G^t$ refer to the gradient of weight, the momentum of second-order statistics and the momentum of gradient, respectively. It can be seen that the places where the momentum of statistics and gradient is located are different for the three ways. We empirically find that the third way usually achieves the best generalization performance. (please refer to Section 5.4.5 for details). The momentum of second-order statistics and the momentum of the gradient are both crucial to the final performance. For the statistics, the momentum can help to reduce the noise so that a more accurate estimation of feature statistics can be obtained.

Weight Decay

Weight decay is an effective technique to improve the generalization performance and accuracy of DNNs [41, 40, 106]. The most commonly used weight decay is the L_2 norm weight regularization on loss $\mathcal{L}$, which can be implemented by introducing the gradient of the regularizer into the gradient of loss. Loshchilov *et al* [51] found that when using Adam as the optimizer, a decoupled weight decay usually keeps better generalization than the L_2 regularizer, which scales the weights by a factor less than 1 in each iteration. Therefore, finding a proper way to add weight decay is very important to the generalization performance.

Since our goal is to perform gradient descent on features but not on weights, directly adopting the standard L_2 norm regularization weight decay cannot yield satisfactory results. We then investigate a new weight decay formula, which actually imposes an

L_2 norm feature regularization term on Eq. (5.4), *i.e*,

$$\min_{\mathbf{W}} ||\mathbf{A}^{t+1} - \mathbf{A}^t(\mathbf{W})||_2^2 + \lambda||\mathbf{A}^t(\mathbf{W})||_2^2. \tag{5.14}$$

The solution can be easily obtained as follows:

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \frac{\eta}{1+\lambda}\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}(\mathbf{X}^t\mathbf{X}^{t^T})^{-1} + \frac{\lambda}{\eta}\mathbf{W}^t\right). \tag{5.15}$$

One can see that we only need to add a term $\frac{\lambda}{\eta}\mathbf{W}^t$ into the modified gradient, which is very easy to implement. If not considering the momentum of the modified gradient, this weight decay method is equivalent to the decoupled weight decay. But if there is a momentum operation, the weight decay term also is involved in it.

Efficient Statistics Computation

Compared with SGDM, the additional computational cost in our algorithm mainly comes from two parts: computing the second-order statistic matrix and its matrix inverse. It will be a large computational burden if we compute them in each iteration. Actually, we find the statistic changes slowly in the training, so it is unnecessary to compute them in each iteration. Instead, to balance the efficiency and performance, we only need to compute them once for several steps. Specifically, we introduce two hyperparameters, *i.e*, T_{xx} and T_{Inv}, to control the interval for updating the statistics matrix and its inverse, respectively. Meanwhile, we use a matrix $\mathbf{D} = (\mathbf{M}_{xx} + \epsilon\mathbf{I})^{-1}$ to store the inverse of the statistics matrix. The updating interval T_{Inv} should be large because its computation involves matrix inverse, which is a computationally expensive operation. In our experiments, we found that setting $T_{xx} = 50$ and $T_{Inv} = 500$ could work effectively and efficiently. It would not cost much more computation than SGDM while keeping a significant performance gain.

What is more, for multiple GPUs, the conventional implementation of computing

statistics is only within one GPU. As a consequence, in this case, the computed second-order statistic would have large noise, which leads to the inaccuracy result. Therefore, we also develop a cross-GPU synchronization approach to expedite the calculation of the feature statistics with multiple GPUs. the statistics are first computed within each GPU and then synchronized across all GPUs. Synchronized BN (SyncBN) [67, 10] also adopt a similar approach to synchronize the statistics across the GPUs. The difference is that SyncBN synchronizes the statistics layer by layer during the forward propagation. The latter layers need to wait for the results of the former layers, which will cost a certain amount of time. While the synchronization operation in the proposed method is independent of the forward propagation so that the statistics of all layers can be synchronized at the same time. it only needs a slight cost. Both the momentum operation and synchronization operation make the obtained feature statistics more stable and more reliable.

Gradient Norm Recovery.

From the above discussion, we can know the proposed FSGD is very similar to the SGDM. The difference is that FSGD adopts a modified gradient to replace the original gradient. The modified gradient in FSGD may have a different scale from the vanilla gradient in SGDM. As a consequence, the optimal hyperparameters of FSGD, such as learning rate and weight decay, may also differ from SGDM. As the most commonly used optimizers, SGDM has been adopted into many tasks with well-tuned hyperparameters. For instance, in objection detection, SGDM usually applies a learning rate of 0.02 and weight decay of 0.0001 as its default settings. A natural question is, can we keep these well-tuned hyperparameters in FSGD to alleviate the tedious work of hyperparameter tuning? If this can be achieved, it would not need further tuning of hyperparameters for FSGD to be used in various tasks that SGD has been well-tuned. It could directly inherit the hyperparameters of SGDM to achieve a performance gain.

Algorithm 5: Feature Stochastic Gradient Descent (FSGD)

Input: $\mathbf{W}^0$, $\mathbf{M}^0$, $\mathbf{M}^0_{xx}$, η, α, β, λ, T_{xx}, T_{Inv}, $\mathbf{D}^0$, ϵ
Output: $\mathbf{W}^{(T)}$

for *t=1:T* **do**
- $\mathbf{G}^t = \nabla_{\mathbf{W}^t}\mathcal{L}$;
- **if** $t\%T_{xx} = 0$ **then**
  - $\mathbf{M}^t_{xx} = \alpha\mathbf{M}^{t-1}_{xx} + (1-\alpha)\mathbf{X}^t\mathbf{X}^{tT}$ %*Statistic momentum*
- **else**
  - $\mathbf{M}^t_{xx} = \mathbf{M}^{t-1}_{xx}$
- **end**
- **if** $t\%T_{Inv} = 0$ **then**
  - $\mathbf{D}^t = (\mathbf{M}^t_{xx} + \epsilon\mathbf{I})^{-1}$ %*Damping and matrix inverse*
- **else**
  - $\mathbf{D}^t = \mathbf{D}^{t-1}$
- **end**
- $\hat{\mathbf{G}}^t = \mathbf{G}^t\mathbf{D}^t$ %*Modified Gradient*
- $\tilde{\mathbf{G}}^t = \hat{\mathbf{G}}^t\frac{||\mathbf{G}^t||_2}{||\hat{\mathbf{G}}^t||_2} + \lambda\mathbf{W}^t$; % *Gradient norm recovery and weight decay*
- $\mathbf{M}^t_G = \beta\mathbf{M}^{t-1}_G + (1-\beta)\tilde{\mathbf{G}}^t$ %*Gradient momentum*
- $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\mathbf{M}^t_G$

end

To achieve this goal, we propose a gradient norm recovery trick to make the norm of the modified gradient the same as the norm of the original gradient. To be specific, we need to find a scale variable to make the L_2 norm of adjusted gradient $\hat{\mathbf{G}}^t = \mathbf{G}^t\mathbf{D}^t$ change to $||\mathbf{G}^t||_2$. Fortunately, It is easy to see that we can recover the gradient norm as follows:

$$\tilde{\mathbf{G}}^t = \hat{\mathbf{G}}^t\frac{||\mathbf{G}^t||_2}{||\hat{\mathbf{G}}^t||_2}, \tag{5.16}$$

We can see that $\tilde{\mathbf{G}}^t$ and $\mathbf{G}^t$ keep the same L_2 norm. By using the gradient norm recovery operation, the gradient $\tilde{\mathbf{G}}^t$ can be readily adopted in the FSGD so that it can boost the performance with no extra hyperparameter tuning based on SGD. Definitely, the performance of FSGD can be further improved by tuning fine-grained hyperparameters around the default settings of SGD.

Overall Algorithm of FSGD

The overall algorithm of FSGD is summarized in **Algorithm 5**. The complexity of FSGD for a FC layer is $T(O(\frac{C_{in}^3}{T_{svd}}) + O(\frac{C_{in}^2 N}{T_{xx}}) + O(C_{in}^2 C_{out}))$, and for a Conv layer it is $T(O(\frac{C_{in}^3 k_1^3 k_2^3}{T_{svd}}) + O(\frac{C_{in}^2 k_1^2 k_2^2 N}{T_{xx}}) + O(C_{in}^2 k_1^2 k_2^2 C_{out}))$, where T is the total number of iterations. As T_{xx} and T_{svd} can be set large in the implementation of FSGD (*e.g*, 50 and 500, respectively), the total complexity is acceptable.

5.2.7 Extension to Other Optimizers

From **Algorithm 5**, it is easy to see that FSGD can be divided into two parts: a pre-conditioner and a main optimizer. The pre-conditioner part decomposes the original gradient **G** into **GD**, while the main optimizer part remains the standard SGD optimizer. Based on this fact, we can easily extend FSGD to other optimizers by introducing the pre-conditioner of FSGD into the main part of the given optimizer. For example, we can introduce the pre-conditioner of FSGD to the widely-used adaptive learning rate optimizer, *i.e*, Adam, resulting in a new Feature Adam (FAdam) optimizer. The detailed algorithm of FAdam is concluded in **Algorithm 6**. Although this extension is straightforward, the experimental results show that FAdam does speed up training processing and improve much the generalization performance over Adam.

5.3 Discussions

5.3.1 Relationship with Back-matching Propagation

FSGD can be considered as a special back-matching propagation method [107, 108, 8], which minimizes a sequence of local back-matching losses following the backward order. Suppose $Q(\{\mathbf{W}\}_{i=1}^K) = \mathcal{L}(\mathbf{Y}, f(\{\mathbf{W}\}_{i=1}^K, \mathbf{X}_0))$ is the loss function of one sample,

Algorithm 6: Feature Adam (FAdam)

Input: $\mathbf{W}^0$, $\mathbf{M}^0$, $\mathbf{M}^0_{xx}$, η, α, β_1, β_2, λ, T_{xx}, T_{Inv}, $\mathbf{D}^0$, ϵ

Output: $\mathbf{W}^{(T)}$

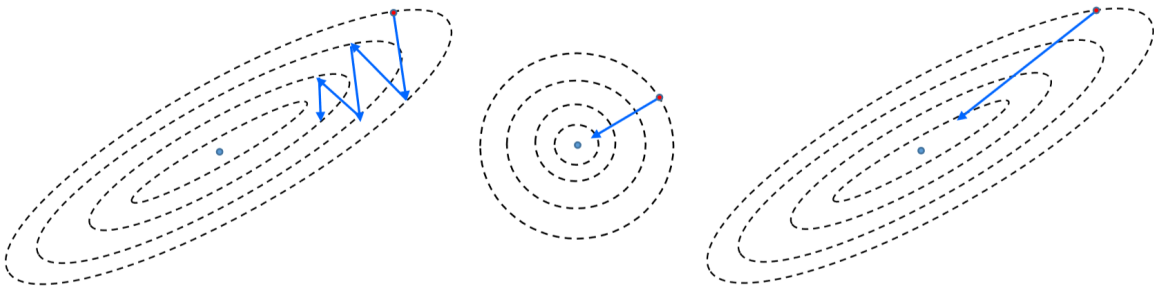
for *t=1:T* **do**
- $\mathbf{G}^t = \nabla_{\mathbf{W}^t}\mathcal{L}$;
- **if** $t\%T_{xx} = 0$ **then**
  - $\mathbf{M}^t_{xx} = \alpha\mathbf{M}^{t-1}_{xx} + (1-\alpha)\mathbf{X}^t\mathbf{X}^{tT}$ %*Statistic momentum*
- **else**
  - $\mathbf{M}^t_{xx} = \mathbf{M}^{t-1}_{xx}$
- **end**
- **if** $t\%T_{Inv} = 0$ **then**
  - $\mathbf{D}^t = (\mathbf{M}^t_{xx} + \epsilon\mathbf{I})^{-1}$ %*Damping and matrix inverse*
- **else**
  - $\mathbf{D}^t = \mathbf{D}^{t-1}$
- **end**
- $\hat{\mathbf{G}}^t = \mathbf{G}^t\mathbf{D}^t$ %*Modified Gradient*
- $\tilde{\mathbf{G}}^t = \hat{\mathbf{G}}^t\frac{||\mathbf{G}^t||_2}{||\hat{\mathbf{G}}^t||_2}$; % *Gradient norm recovery*
- $\mathbf{M}^t_G = \beta_1\mathbf{M}^{t-1}_G + (1-\beta_1)\tilde{\mathbf{G}}^t$
- $\mathbf{V}^t_G = \beta_2\mathbf{V}^{t-1}_G + (1-\beta_2)\tilde{\mathbf{G}}^t \odot \tilde{\mathbf{G}}^t$
- $\hat{\mathbf{M}}^t_G = \frac{\mathbf{M}^t_G}{1-\beta_2^t}$, $\hat{\mathbf{V}}^t_G = \frac{\mathbf{V}^t_G}{1-\beta_3^t}$
- $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\hat{\mathbf{M}}^t_G}{\sqrt{\hat{\mathbf{V}}^t_G}+\epsilon_2}$

end

where $\mathbf{X}_0$ is the input feature, $\mathbf{Y}$ is the target and $f(\cdot,\cdot)$ is the network mapping (*e.g*, ReLU($\mathbf{WX}$) or $\mathbf{WX}$). According to [8], we can take the intermediate output of the network as auxiliary variables and introduce a quadratic penalty:

$$Q(\{\mathbf{W},\mathbf{X}\}_{i=1}^K) = \mathcal{L}(\mathbf{Y}, f_K(\mathbf{W}_K,\mathbf{X}_{K-1})) + \sum_{k=1}^{K-1}\frac{\gamma}{2}||\mathbf{X}_k - f_k(\mathbf{W}_k,\mathbf{X}_{k-1})||_2^2, \quad (5.17)$$

where $\{\mathbf{W}\}_{i=1}^K$ and $\{\mathbf{X}\}_{i=1}^{K-1}$ are to be learned. It has been shown in [64] that when γ is large enough, the solutions of Eq. (5.17) will be very close to the original problem $Q(\{\mathbf{W}\}_{i=1}^K)$. Carreira *et al* [8] suggested minimizing Eq. (5.17) by a block coordinate descent algorithm with $\mathbf{X}$-step and $\mathbf{W}$-step. However, it will cost a large amount of computation and memory cost compared with SGD, because in each $\mathbf{X}$-step or



(a) SGD (b) SGD with whitening (c) FSGD

Figure 5.2: Illustration of the optimization paths of (a) SGD; (b) SGD with feature whitening; and (c) FSGD.

W-step, there are also several iterations to optimize the local objective function.

Instead of solving $\mathbf{X}$ explicitly, we adopt a one-step SGD to update $\mathbf{X}$ and solve $\mathbf{W}$ explicitly. Specifically, for a linear layer, the objective function becomes $\min_{\mathbf{W}_k} ||\mathbf{X}_k^{t+1} - \mathbf{W}_k\mathbf{X}_{(k-1)}^t||_2^2$, where $\mathbf{X}_k^{t+1} = \mathbf{X}_k^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{X}_k^t} = \mathbf{W}_k^t\mathbf{X}_{(k-1)}^t - \eta\frac{\partial\mathcal{L}}{\partial\mathbf{X}_k^t}$ is the intermediate feature after one-step feature gradient descent. The solution is actually the updating formula of FSGD in Eq. (5.7). Therefore, FSGD can be considered as a specific back-matching propagation method but with much higher efficiency.

5.3.2 Relationship with Feature Whitening

We also find that SGD with feature ZCA-whitening in the reparameterized weight space is closely related to FSGD in the original weight space. Let's use a simple linear regression problem to illustrate this relationship. The objective function is $\min_{\mathbf{W}}||\mathbf{Y} - \mathbf{W}\mathbf{X}||_2^2$, where $\mathbf{Y}$ is the ground truth label to be regressed, $\mathbf{X}$ is the feature and $\mathbf{W}$ is the weight to be optimized. If using gradient descent to optimize this problem, there are three choices:

a) Original SGD :$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta(\mathbf{W}^t\mathbf{X}\mathbf{X}^T - \mathbf{Y}\mathbf{X})$

b) SGD with whitening: $\mathbf{W}'^{t+1} = \mathbf{W}'^t - (\mathbf{W}'^t - \mathbf{Y}\mathbf{X}') = (1-\eta)\mathbf{W}'^t + \eta\mathbf{W}'_*$, where $\mathbf{X}' = \mathbf{D}\mathbf{X}$, $\mathbf{W} = \mathbf{W}'\mathbf{D}$, $\mathbf{D} = \mathbf{U}_x\mathbf{\Sigma}_x^{-0.5}\mathbf{U}_x^T$ and $\mathbf{X}\mathbf{X}^T = \mathbf{U}_x\mathbf{\Sigma}_x\mathbf{U}_x^T$. $\mathbf{W}'_* = \mathbf{Y}\mathbf{X}'$ is

the optimal solution in weight space $\mathbf{W}'$.

c) FSGD: $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta(\mathbf{W}^t\mathbf{X}\mathbf{X}^T - \mathbf{Y}\mathbf{X})(\mathbf{X}\mathbf{X}^T)^{-1} = (1-\eta)\mathbf{W}^t + \eta\mathbf{W}_*$, where $\mathbf{W}_* = (\mathbf{Y}\mathbf{X})(\mathbf{X}\mathbf{X}^T)^{-1}$ is the optimal solution in weight space $\mathbf{W}$.

Figure 5.2 illustrates the optimization paths of the above three solutions when training with highly correlated data. The optimization path of the original SGD is jagged and it needs a number of steps to reach the optimal point, while both SGD with feature whitening and FSGD can go directly toward the optimal point. Meanwhile, in this case, $\mathbf{X}\mathbf{X}^T$ is exactly the Hessian matrix and FSGD can also be viewed as a Newton method.

For general DNNs, FSGD still has a close link to SGD with whitening. For a linear layer with feature ZCA-whitening in a DNN, if only considering this one layer and fixing the parameters of other layers, it is easy to obtained that $\mathbf{X}' = \mathbf{D}\mathbf{X}$, $\mathbf{W} = \mathbf{W}'\mathbf{D}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{W}'} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}}\mathbf{D}^T$. The SGD updating formula for $\mathbf{W}'$ is $\mathbf{W}'^{t+1} = \mathbf{W}'^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{W}'^t}$. Multiply both sides of this equation by $\mathbf{D}$, we can get that $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}\mathbf{D}^T\mathbf{D} = \mathbf{W}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}(\mathbf{X}\mathbf{X}^T)^{-1}$, which is actually the update formula of FSGD. Therefore, we can know that SGD with feature ZCA-whitening in the reparameterized weight space is nearly the same as FSGD in the original weight space.

In practical applications, however, feature whitening methods need to redefine forward and backward propagation. They cannot be directly adopted to optimize already-defined DNNs, *e.g*, ResNet50 pre-trained on ImageNet. Moreover, in each iteration of SGD, we have to compute the second-order statistic of the input feature and its SVD decomposition for each whitening layer, which is computationally very expensive [28, 29, 99]. In addition, in most cases, the feature whitening methods can only use the statistics of the current batch. When the batch size is small, the statistics are not accurate and the performance will drop. Compared with the feature whitening methods, FSGD only needs to update the second-order statistic matrix of features and its inverse once for several iterations, which is much more efficient, and it is easy

for FSGD to introduce the moving average of the second-order statistic matrix in batches to reduce the noise in statistics.

5.4 Experiment Results

5.4.1 Experiment Setup

We perform experiments on image classification (on CIFAR100/CIFAR10 [39] and ImageNet [75]) as well as detection and segmentation (on COCO [48]) to illustrate the effectiveness of our proposed FSGD and FAdam method. We first test FSGD and FAdam with different DNN models on CIFAR100/CIFAR10, including VGG11 [82], ResNet18, ResNet50 [22] and DenseNet121 [25], in comparison with advanced optimization algorithms, such as SGDM, Adam [37], AdamW [51], RAdam[1] [49], Ranger[2] [49, 110, 100] and Adabelief[3] [115], AdaHessian[4] [97] and Apollo[5] [59], whose source codes are officially provided. To validate whether FSGD and FAdam performs well on large-scale data set, we further conduct experiments on ImageNet. Finally, we test FSGD on COCO [48] to demonstrate its effectiveness on other vision tasks. And finally we conduct ablation studies (see Section 5.4.5) to choose the momentum and weight decay strategy and tune hyperparameters T_{xx} and T_{Inv} on CIFAR100.

All experiments are conducted under the Pytorch 1.7 framework with eight NVIDIA Tesla P100 GPUs. In FSGD, parameters α control the momentum for second-order statistic matrix, and we set $\alpha = 0.95$ in all the experiments. In addition, we set the small positive number $\epsilon = 0.1$ to make sure that the second-order statistic matrix can inverse. For T_{xx} and T_{Inv}, we set them to 50 and 500, respectively.

[1] `https://github.com/LiyuanLucasLiu/RAdam`
[2] `https://github.com/lessw2020/Ranger-Deep-Learning-Optimizer`
[3] `https://github.com/juntang-zhuang/Adabelief-Optimizer`
[4] `https://github.com/amirgholami/adahessian`
[5] `https://github.com/XuezheMax/apollo`

Table 5.3: Testing accuracies (%) on CIFAR100/CIFAR10. The best and second best results are highlighted in bold and italic fonts, respectively. The improvement of FSGD and FAdam over SGDM and AdamW are is given in red color. "-" means the result is not available.

CIFAR100									
Model	SGDM	AdamW	RAdam	Ranger	Adabelief	AdaHessian	Apollo	FSGD	FAdam
R18	77.20±.30	77.23±.10	77.05±.15	76.75±.11	77.43±.36	76.73±.23	76.63±.27	**79.10**±.15(↑1.90)	*78.18*±.25(↑0.95)
R50	77.78±.43	78.10±.17	78.20±.15	78.13±.12	79.08±.23	78.48±.22	78.68±.11	**81.17**±.17(↑3.39)	*79.90*±.07(↑1.80)
V11	70.80±.29	71.20±.29	71.08±.24	70.58±.14	72.43±.16	67.78±.34	70.05±.11	**73.48**±.16(↑2.68)	*72.48*±.13(↑1.28)
D121	79.53±.19	78.05±.26	78.65±.05	78.28±.08	79.88±.08	-	79.10±.21	**81.00**±.33 (↑1.47)	*80.03*±.08(↑1.98)
CIFAR10									
R18	95.10±.07	94.80±.10	94.70±.18	94.75±.18	95.12±.14	94.70±.15	95.03±.12	**95.50**±.07(↑0.40)	*95.17*±.08(↑0.37)
R50	94.75±.30	94.72±.10	94.72±.10	95.27±.12	95.35±.05	95.35±.11	95.27±.11	**95.90**±.10(↑1.15)	*95.57*±.08(↑0.85)
V11	92.17±.19	92.02±.08	92.00±.18	92.10±.07	92.45±.18	91.85±.16	92.38±.19	**93.28**±.08(↑1.11)	*92.73*±.09(↑0.71)
D121	95.37±.17	94.80±.07	95.02±.08	95.45±.11	95.37±.04	-	95.47±.04	**95.87**±.08 (↑0.50)	*95.50*±.10(↑0.70)

Table 5.4: Top 1 accuracy (%) on the validation set of ImageNet with ResNet18 and ResNet50. The best and second best results are highlighted in bold and italic fonts, respectively. The improvement of FSGD and FAdam over SGDM and AdamW are given in red color.

Model	SGDM	AdamW	RAdam	Ranger	Adabelief	AdaHessian	Apollo	FSGD	FAdam
R18	70.47	70.01	69.92	69.35	70.08	70.08	70.39	**71.04**(↑0.57)	70.69(↑0.68)
R50	76.31	76.02	76.12	75.95	76.22	-	76.32	**77.06**(↑0.75)	77.01(↑0.99)

5.4.2 Results on CIFAR100 and CIFAR10

CIFAR100 and CIFAR10 [39] are two commonly used datasets to testify DNN optimizers. They consist of 50K training images and 10K testing images from 100 categories and 10 categories, respectively, and the size of the input image is 32×32. We train the DNN models for 200 epochs with batch size 128 on one GPU. The learning rate is multiplied by 0.1 for every 60 epochs. We test the proposed method with four representative DNN models[6], including VGG11 (V11) [82] , ResNet18 (R18), ResNet50 (R50) [22] and DenseNet121 (D121) [25]. The compared methods are representative and advanced optimizers, including SGDM, AdamW [51], RAdam [49], Ranger [49, 110, 100] and Adabelief [115], AdaHessian[7] [97] and Apollo [59].

We tune the hyperparameters, including learning rate and weight decay, for all these methods to achieve their best results. Specifically, the learning rate is 0.1 and weight decay is 5e-4 for SGDM. For Adam, the learning rate is 0.001 and weight decay is 5e-4. For AdamW, Radam, Ranger and Adabelief, they all use the decoupled weight decay method [51], and we find that a large weight decay usually leads to better results. Hence, for these optimizers, the learning rate is set to 0.001 and weight decay is set to 0.5. For Adahessian and Apollo, the learning rate is 0.15 and 1, the weight decay is 0.0005 and 0.05, respectively. For FSGD, the learning rate is 0.05 and weight decay is 0.001, and for FAdam the learning rate is 0.0005 and weight decay is 1. Other hyperparameters follow their default settings.

[6]The models for CIFAR100/10 can be found at the repository `https://github.com/weiaicunzai/pytorch-cifar100`

[7]AdaHessian needs huge memory, so we only give partial results.

The experiments are repeated for 5 times and the results are reported in Table 5.3 in mean $\pm$ std format. $\Delta_{\text{FSGD}-\text{SGDM}}$ is the improvement of FSGD over SGDM. It can be seen from the results that FSGD achieves the best testing accuracy for all the used DNN models. More specifically, FSGD improves SGDM from 1.47% to 3.39% on CIFAR100 dataset, and from 0.4% to 1.15% on CIFAR10 dataset. The performance of AdamW, RAdam, Ranger, AdaHessian and Apollo is often worse than SGDM. Adabelief outperforms SGDM but it is still much worse than FSGD. Meanwhile FAdam also achieve remarkable improvement over AdamW.

We can see that the final generalization performance of FSGD and FAdam surpasses other optimizers by a large margin. The better generalization performance of FSGD and FAdam comes from the fact that it takes the feature gradient direction into consideration to adjust the weight gradient. By employing the feature gradient direction to update the weights, they can learn a more favorable feature space, making it easier to reach a flat local minimum.

5.4.3 Results on ImageNet

We then evaluate the proposed FSGD and FAdam methods on the large-scale dataset ImageNet [75], which is a benchmark for image classification. It consists of 1.28 million images for training and 50K images for validation from 1000 categories. We employ ResNet18 and ResNet50 as the backbone networks and train them with different optimizers, including SGDM, AdamW, RAdam, Ranger, Adabelief, AdaHessian, Apollo and our proposed FSGD and FAdam. The training batch size is 256 and four GPUs are used to train the model. We use the standard settings in [9] for training. The models are trained for 100 epochs, and the learning rate is multiplied by 0.1 for every 30 epochs. We refer to the settings in [115, 9] and tune the learning rate and weight decay around their default settings on ImageNet. The learning rate is set to 0.1 for SGDM, 0.001 for AdamW, RAdam, Ranger, Adabelief and FAdam, 0.15 for

AdaHessian, 1 for Apollo, respectively. The weight decay is set to 1e-4 for SGDM and Apollo, 0.1 for AdamW, Ranger and Adabelief, 0.005 for AdaHessian. FSGD and AdamW share the same settings with SGDM and AdamW, respectively. RAdam adopts a weight decay of 0.1 for ResNet‘8 and 0.05 for ResNet50, respectively.

The top 1 accuracies of different methods on the validation set are shown in Table 5.4. The training and validation accuracy curves of SGDM and FSGD are plotted in Fig. 5.3. It can be seen that the proposed FSGD achieves the best result. It outperforms SGDM by 0.57% and 0.75% for ResNet18 and ResNet50, respectively. Meanwhile, the final training accuracy of FSGD surpasses SGDM by 1.56% and 4.14% for ResNet18 and ResNet50, respectively. This means that FSGD can not only boost the generalization but also speed up the optimization process. FAdam also gains a certain performance over Adam. The results of other compared methods are all worse than FSGD and FAdam. This is mainly because that the distributions of both input data and intermediate features on large-scale datasets are more complex, and hence it is more difficult to train the DNNs. The other compared optimizers do not fully exploit the information of intermediate features to compute the weight gradient direction so that their performance is mediocre. Whitening methods de-correlate the features to make their distribution isotropic to avoid being stuck in a bad local minimum [28, 29, 99]. FSGD can be viewed as a special whitening method, which uses the distribution information of features to update the gradient weight. This is why FSGD still keeps high performance on large-scale datasets.

5.4.4 Object Detection and Segmentation

Finally, we evaluate FSGD on COCO [48], which is widely used for object detection and segmentation tasks, to show that FSGD is also effective for more tasks beyond image classification. The models are trained on the COCO *train*2017 dataset (118K images) and evaluated on the COCO *val*2017 dataset (40K images). We use the latest

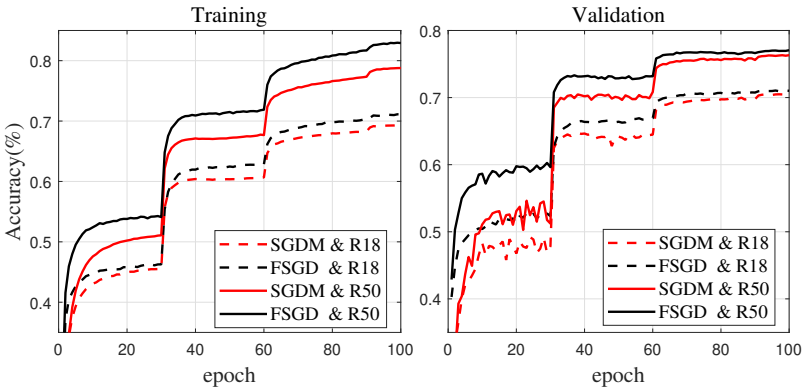


Figure 5.3: Training and validation accuracy curves of SGDM and FSGD on ImageNet with ResNet18 and ResNet50.

Table 5.5: Detection results on COCO by using Faster-RCNN and FPN with ResNet50 and ResNet101 backbone models. Δ means the improvement of FSGD over SGDM.

Backbone	Method	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
	FSGD	38.5	59.3	41.6	21.8	41.8	50.4
	Δ	+1.1	+1.2	+1.2	+0.6	+0.8	+2.3
R101	SGDM	39.4	60.1	43.1	22.4	43.7	51.1
	FSGD	40.8	61.5	44.7	24.6	45.0	53.7
	Δ	+1.4	+1.4	+1.6	+2.2	+1.3	+2.6

version of MMDetection [10] toolbox, which contains comprehensive models on object detection and segmentation, as the detection framework. The official implementations and settings are used for all experiments here. All the pre-trained models are downloaded from their official websites, and we fine-tune them on COCO *train*2017 set with 4 GPUs and 4 images per GPU. The backbone networks include ResNet50 (R50) and ResNet101 (R101). The Feature Pyramid Network (FPN) [47] is also used. The learning rate schedule is 1X for both Faster-RCNN [72] and Mask-RCNN [20].

Since SGDM is dominantly used on COCO for model optimization, we compare it with FSGD. Table 5.5 shows the Average Precision (AP) results of Faster-RCNN. We can see that the models trained with FSGD achieve a clear performance gain by 1.1% for ResNet50 and 1.4% for ResNet101 on object detection. Table 5.6 presents

Table 5.6: Detection and segmentation results on COCO by using Mask-RCNN and FPN with ResNet50 and ResNet101 backbone models. Δ means the improvement of FSGD over SGDM.

Backbone	Method	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
	SGDM	38.2	58.8	41.4	34.7	55.7	37.2
R50	FSGD	39.3	60.6	42.7	36.1	57.5	38.4
	Δ	+1.1	+1.8	+1.3	+1.4	+1.8	+1.2
	SGDM	40.0	60.5	44.0	36.1	57.5	38.6
R101	FSGD	41.0	61.8	44.5	37.3	58.8	39.7
	Δ	+1.0	+1.3	+0.5	+1.2	+1.3	+1.1

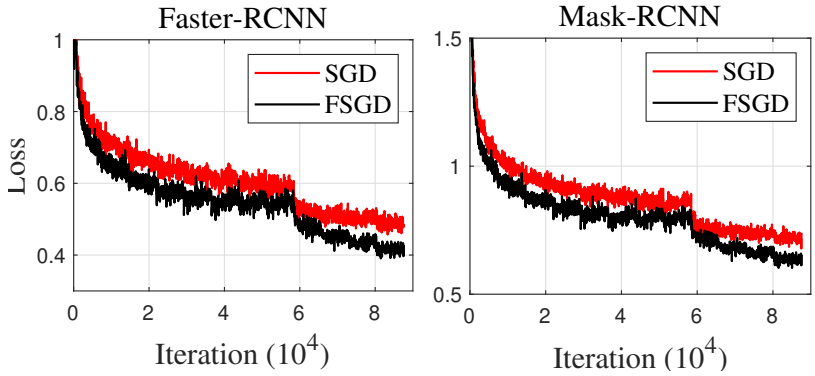


Figure 5.4: Training loss curves of ResNet50 backbone trained by SGDM and FSGD on COCO.

the Average Precision for bounding box (AP^b) and instance segmentation (AP^m) of Mask-RCNN. It can be seen that by FSGD, the AP^b is increased by 1.1% and 1.0% on object detection and 1.4% and 1.2% on segmentation for ResNet50 and ResNet101, respectively. Fig. 5.4 shows the training loss curves of Faster-RCNN and Mask-RCNN with ResNet50 backbone. It can be seen that FSGD can speed up training process and achieve lower final training loss than SGDM. This demonstrates that FSGD is an effective optimization algorithm for various tasks beyond image classification.

5.4.5 Ablation Study

Momentum and Weight Decay. As discussed in Section 5.2.6, there are three ways to introduce the momentum into our proposed method FSGD. We perform a simple experiment on CIFAR100 with ResNet18 to evaluate the three momentum methods. All other hyperparameters are the same as in Section 5.4.2. The testing accuracies of the three momentum methods are 78.72%, 78.36% and 79.10%, respectively. The third momentum method shows obvious improvements over the other two, and hence we choose it for use in FSGD.

In Section 5.2.6, we also presented two weight decay methods, including the original L_2 norm weight decay on gradient and the weight decay in Eq. (5.15). We test these two weight decay strategies on CIFAR100 with ResNet18, and the testing accuracies are 73.36% and 79.10%, respectively. Clearly, our weight decay strategy proposed in Eq. (5.15) is much more effective and it is adopted in FSGD.

Hyperparameter Tuning We first tune T_{xx} and T_{Inv}. A too small interval for updating the input statistics will consume a large amount of computational cost, while a too large interval may lead to unsatisfactory performance. A proper setting of T_{xx} and T_{Inv} can ensure good generalization results with little additional cost. Due to the high computational cost of matrix inverse, T_{Inv} should be set larger than T_{xx}. As shown in Table 5.7, we test six groups of T_{xx} and T_{Inv} and report their testing accuracies and training time per epoch. The combination of $T_{xx} = 50$ and $T_{Inv} = 500$ delivers the best result. Compared with those smaller settings of T_{xx}, the accuracy of $T_{xx} = 50$ does not drop much while it costs less training time. Compared with those larger settings of T_{xx}, the training time of $T_{xx} = 50$ only increases slightly but better performance is kept. With this setting, FSGD only costs about $20\% \sim 30\%$ additional training time to SGDM (23.45 sec/epoch) while achieving much better generalization performance. Meanwhile, we also compare with the whitening method ND [99], which achieves 79.07% (184 sec/epoch) and 78.65% (65 sec/epoch) with 1

Table 5.7: Testing accuracies (%) of ResNet18 by FSGD on CIFAR100 for different T_{xx} and T_{Inv}. The best combination is highlighted in bold font.

T_{xx}	5	10	20	**50**	100	200
T_{Inv}	50	100	200	**500**	1000	2000
Acc	79.07	79.15	79.23	**79.10**	78.85	78.75
Sec/epoch	86.05	58.51	39.78	**30.30**	26.21	24.57

Table 5.8: Testing accuracies (%) of ResNet18 by FSGD with different ϵ on CIFAR100. The best result is highlighted in bold font.

ϵ	0.001	0.01	**0.1**	1
Acc	73.47	77.72	**79.10**	78.55

and 3 (default) sampling strides, respectively. It consumes much more training time. This illustrates that FSGD is more practical for real applications.

We then tune the damping parameter ϵ. The results with different ϵ ($i.e$, 0.001, 0.01, 0.1, 1) of ResNet18 trained by FSGD on CIFAR100 are shown in Table 5.8. It can be clearly observed that $\epsilon = 0.1$ is the best choice. A too small ϵ makes the training unstable because of the large condition number of the statistic matrix. A too large ϵ will drop too much useful information from the second order statistics, which also leads to unsatisfactory performance.

5.5 Conclusion

In this chapter, we proposed a new DNN optimizer, namely feature stochastic gradient descent (FSGD), which takes the gradient of the intermediate features into consideration to update weights. Specifically, we computed the second-order statistic matrix of the input features in a liner layer and used its inverse to update the gradient of weight. Compared with previous DNN optimizers that perform gradient descent on weight, FSGD can find more informative directions to update the feature space. Experimentally, FSGD demonstrated much better generalization performance than SGDM, Adam, AdamW, RAdam and Adabelief on CIFAR100/CIFAR10, ImageNet

and COCO datasets for image classification, object detection and segmentation tasks, with acceptable additional cost on training time and memory over SGDM.

Chapter 6

An Embedded Feature Whitening Approach to Optimize a Deep Neural Network

Compared with the feature normalization methods that are widely used in deep neural network (DNN) training, feature whitening methods take the correlation of features into consideration, which can help to learn more effective features. However, existing feature whitening methods have several limitations, such as the large computation and memory cost, inapplicable to pre-trained DNN models, the introduction of additional parameters, etc., making them impractical to use in optimizing DNNs. To overcome these drawbacks, in this chapter, we propose a novel Embedded Feature Whitening (EFW) approach to DNN optimization. EFW only adjusts the gradient of weight by using the whitening matrix without changing any part of the network so that it can be easily adopted to optimize pre-trained and well-defined DNN architectures. The momentum, adaptive damping and gradient norm recovery techniques associated with EFW are consequently developed, which can be implemented efficiently with acceptable extra computation and memory cost. We apply EFW to two commonly

used DNN optimizers, *i.e*, SGDM and Adam (or AdamW), and name the obtained optimizers as W-SGDM and W-Adam. Extensive experimental results on various vision tasks, including image classification, object detection, segmentation and person ReID, demonstrate the superiority of W-SGDM and W-Adam to state-of-the-art DNN optimizers.

6.1 Introduction

The remarkable success of Deep Neural Networks (DNNs) on various vision tasks, including image classification [22], object detection [72, 20], segmentation [20], image retrieval [114, 52], etc., largely owes to the development of DNN optimization techniques. The main goal of DNN optimization is to find a favorable local minimum of the objective function by using the given training data and ensure good generalization performance of the trained model to testing data. Meanwhile, it is anticipated that we can accelerate the converge speed and reduce the training cost. To achieve these goals, a variety of DNN optimization techniques have been proposed, such as weight initialization strategies [17, 21], efficient active functions (*e.g*, ReLU [63]), batch normalization (BN) [32], gradient clipping [65, 66], adaptive learning rate optimizers [16, 37, 115], and so on. All these techniques facilitate the training of very deep and effective DNN models.

Among the above techniques, normalization methods have been widely used as a basic module to train a variety of DNN architectures [22, 25]. The most representative method is BN [32]. Similar to BN, instance normalization (IN) [88, 30], layer normalization (LN) [44] and group normalization (GN) [93] have also been proposed to perform Z-score standardization on other dimensions. It has been shown that normalization methods can both speed up the training speed and improve the generalization performance [77, 87, 105, 102]. However, normalization methods do not take the correlation of features into consideration. Therefore, feature whitening or feature

decorrelation methods have been developed to solve this problem. For instance, decorrelated batch normalization (DBN) [28] was proposed to perform ZCA-whitening on each mini-batch with a ZCA transformation matrix obtained by eigen-decomposition. IterNorm [29] aims at a more efficient approximation of the ZCA transformation matrix with Newton's iteration. Network deconvolution (ND) [99] extends the ZCA-whitening transformation on a patch of features. The DNN models trained with whitening methods can achieve certain performance gains over normalization methods.

Nevertheless, the existing feature whitening methods have several obvious weaknesses, which make them hard to be widely used in practical applications. The major disadvantage of feature whitening lies in its large computational cost. In each iteration, the ZCA transformation matrix has to be computed by eigen-decomposition, which is computationally expensive when the dimension of features is high. Although some works [29, 99] adopt Newton's iteration to speed up the computation of ZCA transformation, the training cost is still unacceptable compared with BN. Meanwhile, the inference time of the network will increase largely when feature whitening is used. Moreover, feature whitening methods are very memory-consuming in training because more intermediate features need to be stored, especially for the iterative whitening methods. Last but not the least, the existing feature whitening methods cannot be directly applied to optimize pre-trained and well-defined DNN models. One needs to add a feature whitening module into the proper layer and redefine the forward propagation. For instance, if we want to adopt the ResNet50 [22] model pre-trained on ImageNet to downstream tasks, we must redefine the ResNet50 with these whitening methods and train it again on ImageNet. All these drawbacks largely limit the practical usage of feature whitening methods in DNN training.

To address these problems, we propose a novel approach, namely Embedded Feature Whitening (EFW), to DNN optimization by adjusting the gradient of weight with the ZCA transformation matrix. There are several advantages of our proposed approach.

First, EFW inherits the advantages of feature whitening, *i.e*, accelerating the training process and improving the generalization performance. Second, compared with existing feature whitening methods, EFW does not introduce any module into the DNN model to be trained. As a result, it can be directly adopted to optimize most of the existing DNN models without increasing the inference time. Third, its computation and memory cost is acceptable because EFW only computes the ZCA transformation matrix once for many iterations (*e.g*, 500) and it does not store any additional intermediate features. In this chapter, we adopt EFW into two widely used DNN optimizers: SGD with momentum (SGDM) [68, 37] and Adam (or AdanW) [37, 51], and name the obtained optimizers as W-SGDM and W-Adam. Extensive experiments are conducted to validate the effectiveness of EFW on various vision tasks.

6.2 Related Work

DNN Optimizers. The first-order optimization algorithms have been widely adopted in training a DNN. For example, SGD with Momentum (SGDM) [68] makes use of the momentum of the gradient to avoid oscillations and strengthen the relevant gradient direction. Adagrad [16] adapts adaptive learning rates to different parameters, performing larger/smaller gradient steps for infrequent/frequent ones. RMSprop and Adadelta [104] use a similar mechanism to Adagrad, and Adam [37] further introduces the momentum of gradient into adaptive learning rate methods. Based on Adam, Adabelief [115] considers the belief of observed gradient to adjust the adaptive learning rates.

For the second-order optimizers, AdaHessian [97] simplifies the Hessian matrix with the diagonal elements through Hessian-free techniques. Similar to AdaHessian, Apollo [59] simplifies the BFGS algorithm with only diagonal elements. Meanwhile, Kronecker Factored Approximation Curvature (KFAC) [61] uses the Kronecker Factor decomposition to approximate the natural gradient layer-wisely. However, in many computer

Algorithm 7: Overview of Batch Feature Whitening

Input: Mini-batch input $\mathbf{X} \in \mathbb{R}^{C_{out} \times N}$
Output: Output $\mathbf{Y} \in \mathbb{R}^{C_{out} \times N}$

if *Training* **then**
  Centralization: $\hat{\mathbf{X}} = \Phi_1(\mathbf{X}|\boldsymbol{\mu}_{\mathcal{B}})$, $\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{N}\mathbf{X}\mathbf{1}$;
  Standardization or decorrelation: $\mathbf{Y} = \Phi_2(\hat{\mathbf{X}}|\boldsymbol{\Sigma}_{\mathcal{B}})$, $\boldsymbol{\Sigma}_{\mathcal{B}} = \frac{1}{N}\hat{\mathbf{X}}\hat{\mathbf{X}}^T + \epsilon\mathbf{I}$;
  Update the population statistics $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$;
else
  Calculate output $\mathbf{Y} = \Phi_3(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma})$;
end
Recovery Operation $\hat{\mathbf{Y}} = \Phi_4(\mathbf{Y})$

vision tasks, the generalization performance of these second-order methods does not outperform SGDM.

Feature Whitening. Feature whitening methods remove the linear correlation among different channel features to perform gradient descent more efficiently. Beyond standardization, DBN [28] was proposed to perform ZCA-whitening by eigen-decomposition and backpropagating the transformation. IterNorm [29] aims at a more efficient approximation of the ZCA-whitening matrix in DBN with Newton's iteration. Network deconvolution (ND) [99] adopts deconvolution filters to remove pixel-wise and channel-wise correlations. It has been shown that feature whitening methods can boost both the optimization and the generalization of DNNs [29, 99]. However, they usually need a lot of extra computation and memory, making them impractical in real-world applications.

6.3 Embedded Feature Whitening

6.3.1 Overview of Batch Feature Whitening

We briefly summarize the batch whitening process in **Algorithm 7**. In training, batch feature whitening [29, 99] usually involves two main steps, *i.e*, centralization

$\Phi_1(\mathbf{X})$ and decorrelation $\Phi_2(\mathbf{X})$, which are defined as follows:

$$\begin{aligned}\Phi_1(\mathbf{X}|\boldsymbol{\mu}) &= \mathbf{X} - \boldsymbol{\mu}\mathbf{1}^T,\ \mu = \frac{1}{N}\mathbf{X}\mathbf{1},\\ \Phi_2(\mathbf{X}|\boldsymbol{\Sigma}) &= \mathbf{T}\mathbf{X},\ \boldsymbol{\Sigma} = \frac{1}{N}\mathbf{X}\mathbf{X}^T + \epsilon\mathbf{I},\end{aligned} \tag{6.1}$$

where $\mathbf{T}$ is the whitening matrix, which is related to $\boldsymbol{\Sigma}$. For different whitening methods, $\mathbf{T}$ has different formulations [28, 29, 99, 80]. All the whitening matrices should meet that $\frac{1}{N}\Phi_2(\mathbf{X})\Phi_2(\mathbf{X})^T = \mathbf{I}$. Among those whitening transformations, PCA and ZCA whitening are widely used, whose whitening matrices are $\mathbf{T} = \mathbf{D}^{-\frac{1}{2}}\mathbf{U}^T$ and $\mathbf{T} = \mathbf{U}\mathbf{D}^{-\frac{1}{2}}\mathbf{U}^T$, respectively, where $\boldsymbol{\Sigma} = \mathbf{U}\mathbf{D}\mathbf{U}^T$ is the eigen-decomposition of $\boldsymbol{\Sigma} = \mathbf{X}\mathbf{X}^T/N + \epsilon\mathbf{I}$.

In the training step, the batch statistics $\boldsymbol{\mu}_{\mathcal{B}}$ and $\boldsymbol{\Sigma}_{\mathcal{B}}$ are used to perform whitening. Meanwhile, the population statistics $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are updated by exponential moving average [29, 99]. In the inference step, the population statistics are used to replace batch statistics, *i.e*, $\Phi_3(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \Phi_2(\Phi_1(\mathbf{X}|\boldsymbol{\mu}), \boldsymbol{\Sigma})$. After the whitening operation, an additional recovery operation $\Phi_4(\cdot)$ is used to keep the representation capability of the network. The recovery operation is usually a linear operation, such as affine transformation [29] and coloring operation [80], which introduce extra parameters in training.

During the DNN training process, due to the variation of input feature statistics, the whitening matrix also changes. As a consequence, whitening may change the intermediate features acutely, making the following layers hard to learn. It has been shown that ZCA whitening can avoid such a Stochastic Axis Swapping (SAS) problem, leading to better feature learning performance [28]. Actually, we can show that the solution of the following objective function

$$\min_{\mathbf{T}} ||\mathbf{X} - \Phi(\mathbf{X})||_2^2, \quad s.t. \quad \Phi(\mathbf{X}) = \mathbf{T}\mathbf{X}, \quad \frac{1}{N}\Phi(\mathbf{X})\Phi(\mathbf{X})^T = \mathbf{I} \tag{6.2}$$

is $\mathbf{T} = (\mathbf{X}\mathbf{X}^T/N)^{-\frac{1}{2}}$, which is just the ZCA whitening formulation. The proof is as

follows:

Proof. Suppose that $\mathbf{U}\mathbf{D}\mathbf{U}^T$ is the SVD decomposition of $\mathbf{\Sigma} = \mathbf{X}\mathbf{X}^T/N$. From the constraint that $\frac{1}{N}\Phi(\mathbf{X})\Phi(\mathbf{X})^T = \mathbf{I}$, we know $\mathbf{T}\mathbf{\Sigma}\mathbf{T}^T = \mathbf{I}$, and consequently we have $\mathbf{T} = \mathbf{M}\mathbf{D}^{-\frac{1}{2}}\mathbf{U}^T$, where $\mathbf{M}$ is an arbitrary orthogonal matrix with $\mathbf{M}\mathbf{M}^T = \mathbf{M}^T\mathbf{M} = \mathbf{I}$.

The objective to be minimized in Eq. (6.2) is

$$\begin{aligned}
&||\mathbf{X} - \mathbf{T}\mathbf{X}||_2^2 \\
=&tr((\mathbf{X} - \mathbf{T}\mathbf{X})(\mathbf{X} - \mathbf{T}\mathbf{X})^T) \\
=&tr(\mathbf{X}\mathbf{X}^T - \mathbf{X}\mathbf{X}^T\mathbf{T}^T - \mathbf{T}\mathbf{X}\mathbf{X}^T + \mathbf{T}\mathbf{X}\mathbf{X}^T\mathbf{T}^T) \\
=&tr(\mathbf{X}\mathbf{X}^T) + tr(\mathbf{T}\mathbf{X}\mathbf{X}^T\mathbf{T}^T) - 2tr(\mathbf{T}\mathbf{X}\mathbf{X}^T) \\
=&N \cdot tr(\mathbf{\Sigma}) + N \cdot tr(\mathbf{I}) - 2N \cdot tr(\mathbf{T}\mathbf{\Sigma}).
\end{aligned} \tag{6.3}$$

The first two terms of Eq. (6.3) are independent of $\mathbf{T}$. Therefore, minimizing the objective in Eq. (6.2) w.r.t. $\mathbf{T}$ is to maximize the last term in Eq. (6.3):

$$\begin{aligned}
&\max_{\mathbf{T}} tr(\mathbf{T}\mathbf{\Sigma}) \\
=&\max_{\mathbf{M}\mathbf{M}^T=\mathbf{I}} tr(\mathbf{M}\mathbf{D}^{-\frac{1}{2}}\mathbf{U}^T\mathbf{U}\mathbf{D}\mathbf{U}^T) \\
=&\max_{\mathbf{M}\mathbf{M}^T=\mathbf{I}} tr(\mathbf{M}\mathbf{D}^{\frac{1}{2}}\mathbf{U}^T) \\
=&\max_{\mathbf{M}\mathbf{M}^T=\mathbf{I}} tr(\mathbf{D}^{\frac{1}{2}}\mathbf{U}^T\mathbf{M}) \\
=&\max_{\mathbf{Q}=\mathbf{U}^T\mathbf{M},\mathbf{Q}\mathbf{Q}^T=\mathbf{I}} tr(\mathbf{D}^{\frac{1}{2}}\mathbf{Q}) \\
=&\max_{\mathbf{Q}=\mathbf{U}^T\mathbf{M},\mathbf{Q}\mathbf{Q}^T=\mathbf{I}} \sum_i^C \mathbf{D}_{ii}^{\frac{1}{2}}\mathbf{Q}_{ii},
\end{aligned} \tag{6.4}$$

where $\mathbf{D}_{ii}$ and $\mathbf{Q}_{ii}$ are the i^{th} diagonal elements of $\mathbf{D}$ and $\mathbf{Q}$, respectively. Please note that $\mathbf{D}_{ii}^{\frac{1}{2}}$ is positive definite. Since $\mathbf{Q}$ is an orthogonal matrix, its diagonal elements $\mathbf{Q}_{ii} \leqslant 1$. Therefore, $\sum_i^C \mathbf{D}_{ii}^{\frac{1}{2}}\mathbf{Q}_{ii} \leqslant \sum_i^C \mathbf{D}_{ii}^{\frac{1}{2}}$. When $\mathbf{Q} = \mathbf{I}$, the equality holds, and the maximum value $\sum_i^C \mathbf{D}_{ii}^{\frac{1}{2}}$ is reached. Meanwhile, according to $\mathbf{Q} = \mathbf{U}^T\mathbf{M} = \mathbf{I}$, we have $\mathbf{M} = \mathbf{U}$. Therefore, the optimal whitening matrix for Eq. (6.2) is $\mathbf{T} = \mathbf{U}\mathbf{D}^{-\frac{1}{2}}\mathbf{U}^T$. ■

This ensures that the ZCA whitened feature $\Phi(\mathbf{X})$ is close to the original data $\mathbf{X}$ and hence dilutes the SAS issue.

6.3.2 Drawbacks of Feature Whitening

Although many works have shown that feature whitening can both speed up training and gain generalization performance, it has some obvious drawbacks that largely limit its applications to DNN training. First, it needs to perform eigen-decomposition or use Newton's iteration to compute the whitening matrix, both of which will significantly increase the computation and memory cost. Second, existing feature whitening methods cannot be directly adopted to optimize pre-trained DNN models (*e.g*, ImageNet pre-trained models). We have to redefine the forward propagation of DNNs by introducing a whitening module and retrain the models. Third, the batch feature whitening methods are very sensitive to the training batch size. When batch size is small, the statistics will become inaccurate, leading to a large performance drop. Fourth, most of the current whitening methods will introduce additional parameters into the recovery operation step to keep the representation capability of the DNNs, which increases the number of parameters to be optimized.

Due to the above limitations, though having many attractive properties, feature whitening methods have not been widely used to optimize DNNs yet. To overcome the above drawbacks of feature whitening while inheriting its advantages, we should not change the forward propagation of DNN or introduce new modules (*e.g*, whitening layer) in DNN, and should reduce its extra computation cost. To achieve these goals, we propose a novel approach to embed the feature whitening operation into the optimization algorithms.

6.3.3 Removal of Recovery and Centralization Operations

Most batch whitening methods employ a recovery operation to keep the representation capability of DNNs. Actually, the recovery operation may not be necessary. According to their locations in DNN layers, whitening methods can be divided into pre-whitening and post-whitening ones. When the whitening layer is placed before the convolutional layer, it is a pre-whitening layer, otherwise, it is a post-whitening layer. Traditional normalization layers and whitening layers usually introduce an additional recovery transformation, such as affine transformation [29] or coloring operation [80], to keep the feature representation performance. When post-whitening is adopted, the recovery transformation must be introduced after the whitening operation to keep the performance.

When pre-whitening is adopted, however, the recovery transformation can be removed without harming the representation power of DNNs, because it can be assimilated by the following Conv layer. For instance, supposing that $\mathbf{W_r} * \mathbf{X}$ is the recovery transformation (affine and coloring transformation can be viewed as a sparse convolutional operation), where $\mathbf{W_r}$ is the extra parameters to be learned, $\mathbf{W} * \mathbf{W_r} * \mathbf{X}$ will be the output feature of Conv layer. We can let $\mathbf{W}' = \mathbf{W} * \mathbf{W_r}$ and hence only optimize the Conv layer with parameter $\mathbf{W}'$. This property of pre-whitening inspires us to embed the whitening layer into the optimization algorithm without changing any module of the DNN.

Meanwhile, in the traditional whitening methods, there are two main operations: centralization and decorrelation. In forward propagation, we need to introduce these two operations into the whitening layer before optimization. However, for a well-defined DNN, the mean of input feature to a Conv or FC layer is usually not zero since there is no centralization operation before them. A practical way to achieve feature centralization is to introduce an extra bias that is related to the mean of input activation. However, since the normalization layers are usually located after

the Conv layer in many popular DNNs (*e.g*, ResNet), the bias in the Conv layer will have no function. Moreover, since our goal is to optimize a well-defined DNN without changing its forward propagation and introducing any extra parameters, we omit the centralization operation and only take the decorrelation into consideration.

6.3.4 Formulation of Embedded Feature Whitening

For a FC layer $\mathbf{Y} = \mathbf{W}\mathbf{X}$, where $\mathbf{W}$ denotes the parameters to learn, suppose there is a virtual whitening layer before FC layer, which is $\hat{\mathbf{X}} = \mathbf{T}\mathbf{X}$, where $\mathbf{T}$ is defined in Eq. (6.1). We can reformulate this FC layer with a whitening transformation as $\mathbf{Y} = \mathbf{W}'\mathbf{T}\mathbf{X}$, where $\mathbf{W}'$ is the new parameters to be optimized. In this way, we can optimize the loss function w.r.t$\mathbf{W}'$, and let $\mathbf{W} = \mathbf{W}'\mathbf{T}$ once the training is finished. According to the backpropagation algorithm, the gradient of $\mathbf{W}'$ can be easily obtained by $\frac{\partial \mathcal{L}}{\partial \mathbf{W}'} = \frac{\partial \mathcal{L}}{\partial \mathbf{W}}\mathbf{T}$. However, the above approach has several serious problems. First, the whitening matrix $\mathbf{T}$ will change during the training process because of the update of weights in the previous layers. As a consequence, the relationship between $\mathbf{W}$ and $\mathbf{W}'$ is not fixed. Second, in the training process, $\mathbf{T}$ will contain a certain amount of noise due to the random batch sampling, so it is hard to get an accurate $\mathbf{T}$. Therefore, it is difficult to obtain an accurate $\mathbf{W}$ from the optimization of $\mathbf{W}'$.

To maintain the benefits of batch feature whitening on optimization, we propose to use a modified gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}\mathbf{T}$ to replace the original weight gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{W}}$, and name the method Embedded Feature Whitening (EFW), which embeds the information of feature whitening into the weight gradient. EFW can be introduced into the FC layer, convolutional layer, and Norm layer. Compared with the weight updating formula of SGD $\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}$, the updating formula of SGD with EFW is

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}\mathbf{T}^t. \tag{6.5}$$

The detailed updating formulas are summarized in TABLE 6.1. We ignore the factor

Table 6.1: The updating formulas and whitening matrices of FC, Conv and Norm layers in SGD with the proposed EFW.

Layer	Updating formula	Whitening matrix
FC layer	$\mathbf{W}^{t+1} = \mathbf{W}^t - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^t} \mathbf{T}^t$	$\mathbf{T}^t = \left(\mathbf{X}^t \mathbf{X}^{tT}\right)^{-\frac{1}{2}}$
Conv layer	$\mathcal{U}_1(\mathbf{W}^{t+1}) = \mathcal{U}_1(\mathbf{W}^t) - \eta \mathcal{U}_1(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^t}) \mathbf{T}^t$	$\mathbf{T}^t = \left(\boldsymbol{\mathfrak{X}}^t \boldsymbol{\mathfrak{X}}^{tT}\right)^{-\frac{1}{2}}$
Norm layer	$\begin{bmatrix} \gamma^{t+1} \\ \beta^{t+1} \end{bmatrix} = \begin{bmatrix} \gamma^t \\ \beta^t \end{bmatrix} - \eta \mathbf{T}^t \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \gamma^t} \\ \frac{\partial \mathcal{L}}{\partial \beta^t} \end{bmatrix}$	$\mathbf{T}^t = \left(\begin{bmatrix} vec(\mathbf{X}^t)^T \\ \mathbf{1}^T \end{bmatrix} [vec(\mathbf{X}^t), \mathbf{1}] \right)^{-\frac{1}{2}}$

$1/N$ in the second-order statistic because of the gradient norm recovery operation, which will be explained in Section 6.3.5.

For the FC layer, we need to calculate the second-order statistic of input activation, *i.e*, $\mathbf{X}^t \mathbf{X}^{tT}$, and the whitening matrix $\mathbf{T}^t$, which can be obtained by SVD decomposition of $\mathbf{X}^t \mathbf{X}^{tT}$. For the Conv layer, the difference from the FC layer lies in that we need to unfold the convolution operation to matrix multiplication first. The convolution operation can be formulated as a matrix multiplication with the *im2col* operation [99, 108], and then the Conv layer can be viewed as an FC layer. The updating formula of weights for the Conv layer is list in TABLE 6.1, where $\mathcal{U}_1(\cdot)$ is the mode 1 unfold operation of a tensor and $\boldsymbol{\mathfrak{X}}$ is the matrix of $\mathbf{X}^t$ after *im2col* operation. The normalization layers usually have a channel-wise affine transformation, which is also a linear operation. Suppose that the normalized features are $\mathbf{X}^t$ and the parameters of affine transformation are γ and β for one channel, we can obtain the updating rules for γ, β as shown in the bottom row of TABLE 6.1. If the mean and variance of $\mathbf{X}^t$ are zero and one, the second-order statistics will be a diagonal 2×2 matrix. For example, when BN [32] and IN [88, 30] are used, the update rules for (γ, β) degrade to the case of SGD. However, for other normalization methods such as GN [93] and LN [44], the mean and variance of each channel may not be zero and one.

In practice, to avoid that the condition number of the statistic matrix $\mathbf{X}^t \mathbf{X}^{tT}$ is too

large, we need to add an additional term $\epsilon\mathbf{I}$ to the statistic matrix, where $\mathbf{I}$ is an identity matrix and ϵ is the damping parameter. We will discuss how to choose a proper ϵ in the next section.

6.3.5 Implementation of EFW

Momentum. The estimation of the second-order statistics of $\mathbf{X}$ is very important for the whitening methods. The original batch whitening method can only use the current batch statistics for computation, and hence they are very sensitive to the training batch size. When the training batch size is small, the batch statistics will have large noise so that the training will be unstable. In contrast, our proposed EFW method works directly on the final weight updating stage, and it does not change the forward propagation and backward propagation during training. Therefore, EFW can adopt the statistics from more batches to achieve a more accurate estimation of feature statistics. Specifically, we compute the momentum of the batch statistics as follows:

$$\mathbf{M}_{xx}^{t} = \alpha\mathbf{M}_{xx}^{t-1} + (1-\alpha)\mathbf{X}^{t}\mathbf{X}^{t^{T}}, \tag{6.6}$$

where $\mathbf{M}_{xx}^{t}$ is the momentum of statistics $\mathbf{X}\mathbf{X}^{T}$ in iteration t and α is the momentum parameter. As an approximation to the population of feature statistics, momentum can significantly reduce the noise caused by random batch sampling.

Statistics Computation. Feature whitening methods need to compute the second-order statistics and then compute the whitening matrix for feature learning. The previous batch whitening methods need to perform these computations in each iteration for each batch because the batch statistics and whitening matrix are involved in forward and backward propagations. This however introduces a large amount of computational burden.

Different from the previous batch whitening methods, in our proposed EFW there

is no need to compute the second-order statistics and the whitening matrix in each iteration. We only need to compute them once for many iterations. Two hyperparameters, T_{xx} and T_{svd}, are introduced to control the interval for updating the statistics matrix and the whitening matrix, respectively. For the whitening matrix, the updating interval should be set larger because its computation involves SVD decomposition, which is more computationally expensive. In our experiments, we set $T_{xx} = 50$ and $T_{svd} = 500$ and we find that they work effectively to improve the DNN optimization performance without introducing much additional computation cost.

We also implement a cross-GPU synchronization method to facilitate the computation of more reliable feature statistics when using multiple GPUs. For each linear layer, the statistics are first computed on each GPU and then synchronized across all GPUs. This method shares a similar spirit with Synchronized BN (SyncBN) [67, 10]. However, the synchronization operation in SyncBN needs to be performed during both the forward propagation and backpropagation processes for each BN layer, which will cost a certain amount of time. Fortunately, the synchronization operation in our algorithm can be implemented independently after forward propagation and backpropagation, so that its cost is negligible. With this synchronization operation, the computation of feature statistics is more stable even when the per-GPU batch size is small.

Adaptive Damping. The dimension of $\mathbf{M}_{xx}^t$ is very high and it is usually a very singular matrix. When the condition number of $\mathbf{M}_{xx}^t$ is too large, it will be unstable to compute the inverse square root of it. To avoid such a case, in practice we need to add an additional term $\epsilon\mathbf{I}$ to the statistic matrix, where $\mathbf{I}$ is an identity matrix and ϵ is a damping parameter.

Too-small damping may not improve the condition number of $\mathbf{M}_{xx}^t$, while too strong damping may reduce the accuracy of statistics. Therefore, it is important to choose a proper damping parameter ϵ. For different layers in a DNN, the statistics $\mathbf{M}_{xx}^t$ may have different magnitude. Thus, it is improper to use a uniform damping scheme for

Algorithm 8: Embedded Feature Whitening (EFW)

Input: T_{xx}, T_{svd}, α, $\mathbf{M}_{xx}^{t-1}$, $\mathbf{T}^{t-1}$, ϵ, input activation $\mathbf{X}^t$, gradient $\nabla_{\mathbf{W}^t}\mathcal{L}$
Output: $\tilde{\mathbf{G}}^t$

$\mathbf{G}^t = \nabla_{\mathbf{W}^t}\mathcal{L}$
if $t\%T_{xx} = 0$ **then**
  $\mathbf{M}_{xx}^t = \alpha\mathbf{M}_{xx}^{t-1} + (1-\alpha)\mathbf{X}^t\mathbf{X}^{tT}$ % *Momentum step*
else
  $\mathbf{M}_{xx}^t = \mathbf{M}_{xx}^{t-1}$
end
if $t\%T_{svd} = 0$ **then**
  $\mathbf{U}\mathbf{D}\mathbf{U}^T = \mathbf{M}_{xx}^t$ % *SVD decomposition*
  $\mathbf{T}^t = \mathbf{U}(\mathbf{D} + \epsilon d_{max}\mathbf{I})^{-1/2}\mathbf{U}^T$ % *Whitening matrix with adaptive damping*
else
  $\mathbf{T}^t = \mathbf{T}^{t-1}$
end
$\hat{\mathbf{G}}^t = \mathbf{G}^t\mathbf{T}^t$ % *Adjust gradient with whitening matrix*
$\tilde{\mathbf{G}}^t = \hat{\mathbf{G}}^t\frac{||\mathbf{G}^t||_2}{||\hat{\mathbf{G}}^t||_2}$; % *Gradient norm recovery*

all layers. By taking the magnitude of different features into consideration, we choose an adaptive damping parameter ϵd_{max}, where d_{max} is the max singular value of $\mathbf{M}_{xx}^t$. It is easy to show that the condition number of $\mathbf{M}_{xx}^t + \epsilon d_{max}\mathbf{I}$ is $\frac{d_{max}+\epsilon d_{max}}{d_{min}+\epsilon d_{max}} < \frac{1+\epsilon}{\epsilon}$. In practice, we can first compute the SVD decomposition of $\mathbf{M}_{xx}^t$, *i.e*, $\mathbf{U}\mathbf{D}\mathbf{U}^T = \mathbf{M}_{xx}^t$, and then obtain the whitening matrix by $\mathbf{T}^t = \mathbf{U}(\mathbf{D} + \epsilon d_{max}\mathbf{I})^{-1/2}\mathbf{U}^T$. The computation cost of adaptive damping is the same as fixed damping.

Gradient Norm Recovery. SGDM and Adam are among the most commonly used optimizers in training DNNs. Their hyperparameters, including learning rate and weight decay, have been well-tuned by researchers on many specific tasks. For example, in objection detection, SGDM with a learning rate 0.02 and weight decay 0.0001 is widely adopted. A natural question is can we hold these well-tuned hyperparameters in the proposed method to ease the tedious work of hyperparameter tuning? If this can be done, EFW can be easily used for solving various vision tasks without further hyperparameter tuning.

In the proposed EFW, the scale of adjusted gradient $\hat{\mathbf{G}}^t = \mathbf{G}^t\mathbf{T}^t$ might be changed.

This implies that the optimal setting of hyperparameters should be changed for the adopted optimizer, limiting the application of the proposed method. Fortunately, this problem of gradient scale changing can be easily addressed by recovering the gradient norm, which is

$$\tilde{\mathbf{G}}^t = \hat{\mathbf{G}}^t \frac{||\mathbf{G}^t||_2}{||\hat{\mathbf{G}}^t||_2}, \tag{6.7}$$

It is easy to see that $\tilde{\mathbf{G}}^t$ and $\mathbf{G}^t$ have the same L_2 norm. With the gradient norm recovery operation, $\tilde{\mathbf{G}}^t$ can be readily used in the employed optimizers (*e.g*, SGDM and Adam) to achieve favorable performance without additional hyperparameter tuning. Of course, one may further improve the performance by tuning fine-grained hyperparameters around their default settings.

Algorithm of EFW. The complexity of EFW is $T(O(\frac{C_{in}^3}{T_{svd}}) + O(\frac{C_{in}^2 N}{T_{xx}}) + O(C_{in}^2 C_{out}))$ for a FC layer, and $T(O(\frac{C_{in}^3 k_1^3 k_2^3}{T_{svd}}) + O(\frac{C_{in}^2 k_1^2 k_2^2 N}{T_{xx}}) + O(C_{in}^2 k_1^2 k_2^2 C_{out}))$ for a Conv layer, where T is the total number of iterations. Since T_{xx} and T_{svd} can be set as large numbers in our implementation (50 and 500, respectively), the complexity is acceptable. The algorithm of EFW is summarized in **Algorithm 8**. In the experiments, we apply EFW to the two commonly used DNN optimizers, *i.e*, SGDM and Adam (or AdamW), and name the obtained new optimizers as W-SGDM and W-Adam accordingly.

6.4 Experiment Results

6.4.1 Experiment Setup

We evaluate the proposed W-SGDM and W-Adam on various vision tasks, including image classification (on CIFAR100/CIFAR10 [39] and ImageNet [75]), object detection and segmentation (on COCO [48]), and Person Re-identification (Person ReID, on Market1501 [114] and DukeMTMC-ReID [73]). The compared methods

Table 6.2: The learning rate (LR), weight decay (WD) and weight decay methods for different optimizers on CIFAR100 and CIFAR10. The weight decay methods include L_2 regularization weight decay (WD1) and weight decouple (WD2).

Optimizer	SGDM	AdamW	RAdam	Ranger	Adabelief	AdaHessian	Apollo	W-SGDM	W-Adam
LR	0.1	0.001	0.001	0.001	0.001	0.15	0.01	0.05	0.0005
WD	0.0005	0.5	0.5	0.5	0.5	0.0005	0.05	0.001	1
WD method	WD1	WD2	WD2	WD2	WD2	WD2	WD2	WD1	WD2

include the representative and state-of-the-art DNN optimizers, including SGDM, AdamW [51], RAdam [49], Ranger [49, 110, 100] and Adabelief [115], AdaHessian[1] [97] and Apollo [59]. For the competing methods, we use the default settings for most of their hyper-parameters, and tune their learning rates and weight decays to report their best results.

We first testify W-SGDM and W-Adam with different DNN models on CIFAR100/CIFAR10, including VGG11 [82], ResNet18, ResNet50 [22] and DenseNet-121 [25]. Then we perform experiments on ImageNet to validate their performance on the large-scale datasets. After that, we test W-SGDM on COCO for detection and segmentation, and test W-Adam on Market1501 [114] and DukeMTMC-ReID for Person ReID to demonstrate that EFW can be easily adopted to finetune pre-trained models. All experiments are conducted under the Pytorch 1.7 framework with eight NVIDIA Tesla P100 GPUs. For the hyper-parameters of EFW, we set $\alpha = 0.95$, $T_{xx} = 50$ and $T_{svd} = 500$, $\epsilon = 0.001$ throughout the experiments if not specified. Ablation studies on hyperparameter selection are also provided.

6.4.2 Image Classification

Results on CIFAR100 and CIFAR10:

CIFAR100 and CIFAR10 [39] are two popular datasets to testify DNN optimizers. They include 50K training images and 10K testing images from 100 categories

[1]Since AdaHessian is very memory expensive, we can only give partial results in the following experiments.

Table 6.3: Testing accuracies (%) on CIFAR100/CIFAR10. The best and second best results are highlighted in bold and italic fonts, respectively. The numbers in red color indicate the improvement of W-SGDM/W-Adam over SGDM/AdamW, respectively. "-" means that the result is not available due to the problem of "out of memory".

CIFAR100									
Model	SGDM	AdamW	RAdam	Ranger	Adabelief	AdaHessian	Apollo	W-SGDM	W-Adam
R18	77.20±.30	77.23±.10	77.05±.15	76.75±.11	77.43±.36	76.73±.23	76.63±.27	**79.28**±.27 (↑2.08)	*78.75*±.16(↑1.52)
R50	77.78±.43	78.10±.17	78.20±.15	78.13±.12	79.08±.23	78.48±.22	78.68±.11	**80.90**±.23 (↑3.12)	*80.15*±.22(↑2.05)
V11	70.80±.29	71.20±.29	71.08±.24	70.58±.14	72.43±.16	67.78±.34	70.05±.11	**73.42**±.28 (↑2.62)	*72.92*±.14(↑1.72)
D121	79.53±.19	78.05±.26	78.65±.05	78.28±.08	79.88±.08	-	79.10±.21	**81.23**±.10 (↑1.70)	*80.10*±.25(↑2.05)
CIFAR10									
R18	95.10±.07	94.80±.10	94.70±.18	94.75±.18	95.12±.14	94.70±.15	95.03±.12	**95.43**±.08 (↑0.33)	*95.20*±.10(↑0.40)
R50	94.75±.30	94.72±.10	94.72±.10	95.27±.12	95.35±.05	95.35±.11	95.27±.11	**95.80**±.15 (↑1.05)	*95.70*±.07(↑0.98)
V11	92.17±.19	92.02±.08	92.00±.18	92.10±.07	92.45±.18	91.85±.16	92.38±.19	**92.95**±.20 (↑0.78)	*92.88*±.19(↑0.86)
D121	95.37±.17	94.80±.07	95.02±.08	95.45±.11	95.37±.04	-	95.47±.04	**95.72**±.14 (↑0.35)	*95.47*±.12(↑0.67)

Table 6.4: The learning rate (LR), weight decay (WD) and weight decay methods and for different optimizers on ImageNet.The weight decay methods include L_2 regularization weight decay (WD1) and weight decouple (WD2).

	Optimizer	SGDM	AdamW	RAdam	Ranger	Adabelief	AdaHessian	Apollo	W-SGDM	W-Adam
R18	LR	0.1	0.001	0.001	0.001	0.001	0.15	1	0.1	0.001
	WD	0.0001	0.1	0.1	0.1	0.05	0.0005	0.0001	0.0001	0.1
R50	LR	0.1	0.001	0.001	0.001	0.001	-	1	0.1	0.001
	WD	0.0001	0.1	0.05	0.1	0.1	-	0.0001	0.002	0.2
	WD method	WD1	WD2	WD2	WD2	WD2	WD2	WD1	WD1	WD2

and 10 categories, respectively, and the resolution of the input image is 32×32. We conduct experiments on these two relatively small-scale datasets to illustrate the effectiveness of W-SGDM and W-Adam with different DNN backbone models, including VGG11 (V11), ResNet18 (R18), ResNet50 (R50) and DenseNet121 (D121)[2]. All the DNN models are trained for 200 epochs with batch size 128 on one GPU. The learning rate is multiplied by 0.1 for every 60 epochs. We tune the learning rate in $\{1e^{-4}, 5e^{-4}, 1e^{-3}, 5e^{-3}, 1e^{-2}, 5e^{-2}, 0.1, 0.15\}$ and weight decay in $\{1e^{-4}, 5e^{-4}, 1e^{-3}, 5e^{-3}, 1e^{-2}, 5e^{-2}, 0.1, 0.5, 1\}$, and choose the best combination of them for all methods. The detailed settings can be found in Table 6.2. We use the default settings for other hyperparameters.

The experiments are repeated 4 times and the results are reported in Table 6.3 in mean $\pm$ std format. We can see that W-SGDM and W-Adam achieve the best and second-best testing accuracies for all the used DNN models. More specifically, W-SGDM improves SGDM from 1.7% to 3.12% on CIFAR100, and from 0.33% to 1.05% on CIFAR10, while W-Adam improves AdamW from 1.52% to 2.05% on CIFAR100, and from 0.4% to 0.98% on CIFAR10. Among the adaptive learning rate methods, Adam, AdamW, RAdam and Ranger perform worse than SGDM, while only Adabelief outperforms SGDM but it is still much worse than W-SGDM and W-Adam. It can be seen that the generalization performance of W-SGDM and W-Adam significantly surpass other optimizers, validating the effectiveness of our proposed EFW scheme.

[2]These models for CIFAR100/10 can be downloaded at the repository `https://github.com/weiaicunzai/pytorch-cifar100`.

Table 6.5: Top 1 accuracy (%) on the validation set of ImageNet. The numbers in red color indicate the improvement of W-SGDM/W-Adam over SGDM/AdamW, respectively. "-" means that the result is not available due to the problem of "out of memory".

Model	SGDM	AdamW [51]	RAdam [49]	Ranger	Adabelief [115]	AdaHessian [97]	Apollo [59]	W-SGDM	W-Adam
R18	70.47	70.01	69.92	69.35	70.08	70.08	70.39	*71.43*(↑0.96)	**71.59**(↑1.58)
R50	76.31	76.02	76.12	75.95	76.22	-	76.32	**77.48**(↑1.17)	*76.83*(↑0.81)

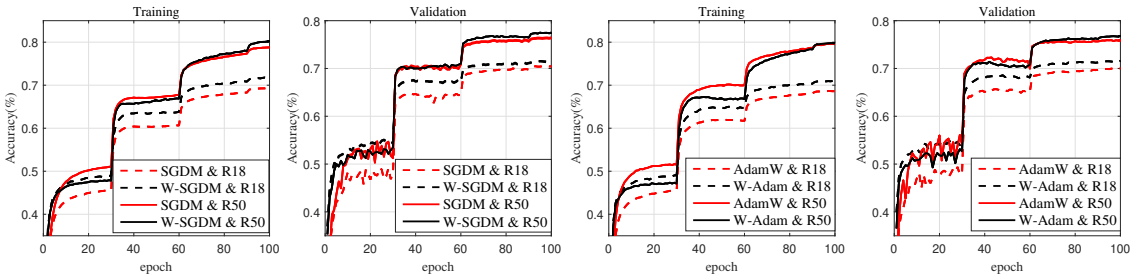


Figure 6.1: Training and validation accuracy curves of SGDM, W-SGDM, AdamW and W-Adam on ImageNet with ResNet18 and ResNet50.

Results on ImageNet:

We then evaluate W-SGDM and W-Adam on the large-scale image classification dataset ImageNet [75], which consists of 1.28 million training images and 50K validation images from 1000 categories. ResNet18 and ResNet50 are employed as the backbone models with training batch size 256 on 4 GPUs. The standard settings in [9] are used, where the models are trained for 100 epochs. We refer to the strategies in [115] to set the learning rate and weight decay. The detailed settings for different optimizers can be found in the Table 6.4. The top 1 accuracies of competing optimizers on the validation set are reported in Table 6.5. We can see that W-SGDM and W-Adam are the top 2 performers. Specifically, W-SGDM outperforms SGDM by 0.96% and 1.17%, and W-Adam outperforms AdamW by 1.58% and 0.81% for ResNet18 and ResNet50, respectively. The training and validation accuracy curves of SGDM vs. W-SGDM and AdamW vs. W-Adam are plotted in Fig. 6.1. For ResNet18, the learning rate and weight decays of W-SGDM and W-Adam are the same as SGDM and AdamW, respectively. While for ResNet50, the weight decays of W-SGDM and W-Adam are set larger than SGDM and AdamW. It can be seen

Table 6.6: Detection and segmentation results of Faster-RCNN on COCO. Δ means the gain of W-SGDM over SGDM.

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
	W-SGDM	39.4	60.6	43.1	23.1	42.9	50.7
	Δ	$\uparrow$2.0	$\uparrow$2.5	$\uparrow$2.7	$\uparrow$1.9	$\uparrow$1.9	$\uparrow$2.6
R101	SGDM	39.4	60.1	43.1	22.4	43.7	51.1
	W-SGDM	41.1	61.6	45.1	24.0	45.2	54.3
	Δ	$\uparrow$1.7	$\uparrow$1.5	$\uparrow$2.0	$\uparrow$1.6	$\uparrow$1.5	$\uparrow$3.2

that W-SGDM and W-Adam achieve both higher training accuracy and validation accuracy than SGDM and AdamW. This indicates that EFW can not only boost the generalization performance but also speed up the training process of DNN models on large-scale datasets.

6.4.3 Object Detection and Segmentation

We then test W-SGDM on COCO [48] detection and segmentation tasks to show that it can be adopted for finetuning pre-trained models without changing the hyperparameters of SGDM, which is the default optimizer with well-tuned hyperparameters for these tasks. The pre-trained models are downloaded from the PyTorch official websites. They are fine-tuned on COCO *train*2017 (118K images) with 4 GPUs and 4 images per GPU, and then evaluated on COCO *val*2017 (40K images). The latest version of MMDetection [10] toolbox is used as the framework. We adopt the official implementations and settings for all experiments here. The backbone networks include ResNet50 (R50) and ResNet101 (R101). The Feature Pyramid Network (FPN) [47] is also used. The learning rate schedule is 1X for both Faster-RCNN [72] and Mask-RCNN [20].

As we discussed in Section 6.3.5, with the gradient norm recovery operation in EFW, we can directly adopt the hyperparameters of SGDM into W-SGDM. Table 6.6 lists the Average Precision (AP) of object detection by Faster-RCNN. One can see that the

Table 6.7: Detection results of Mask-RCNN on COCO. Δ means the gain of W-SGDM and W-Adan over SGDM and AdamW, respectively.

Backbone	Algorithm	AP^b	$AP^b_{.5}$	$AP^b_{.75}$	AP^m	$AP^m_{.5}$	$AP^m_{.75}$
	SGDM	38.2	58.8	41.4	34.7	55.7	37.2
R50	W-SGDM	39.8	60.8	43.4	36.4	57.6	38.9
	Δ	↑1.6	↑2.0	↑2.0	↑1.7	↑1.9	↑1.7
	SGDM	40.0	60.5	44.0	36.1	57.5	38.6
R101	W-SGDM	41.7	62.5	45.5	37.9	59.4	40.8
	Δ	↑1.7	↑2.0	↑1.5	↑1.8	↑1.9	↑2.2
	AdamW	42.7	65.2	46.8	39.3	62.2	42.2
Swin-T	W-Adam	43.4	65.7	47.5	40.1	63.0	43.2
	Δ	↑0.7	↑0.5	↑0.7	↑0.8	↑0.8	↑1.0

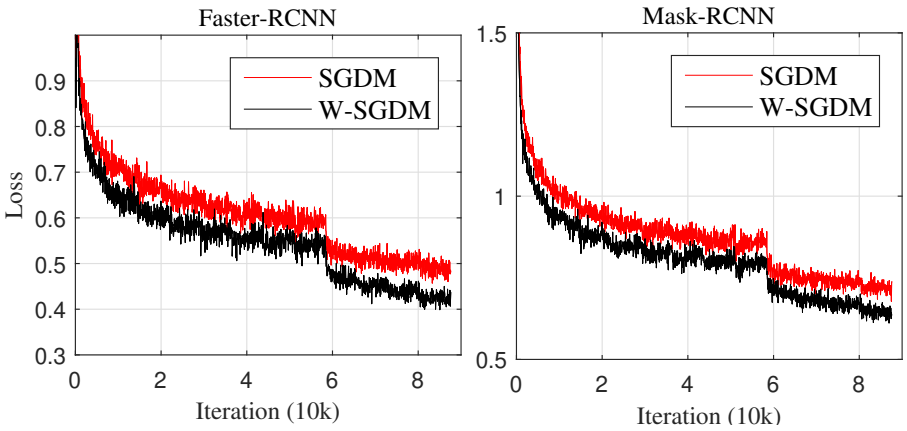


Figure 6.2: Training loss curves on COCO by ResNet50.

models trained by W-SGDM achieve a clear performance boost of 2.0% for ResNet50 and 1.7% for ResNet101. Table 6.7 reports the AP^b of detection and AP^m of segmentation by Mask-RCNN. W-SGDM gains AP^b by 1.6% and 1.7% on object detection and 1.7% and 1.8% on segmentation for ResNet50 and ResNet101, respectively. Fig. 6.2 shows the training loss curves of Faster-RCNN and Mask-RCNN with ResNet50 backbone. One can see that W-SGDM accelerates the training process and achieves a more favorable local minimum than SGDM. This experiment clearly validates that EFW can be readily embedded into existing optimizers without extra hyper-parameter tuning.

Table 6.8: Rank1(%) and mAP(%) on Market1501 and DukeMTMC-reID. Δ means the gain of W-Adam over Adam.

Dataset		Market1501		DukeMTMC-reID	
Backbone	Algorithm	Rank1	mAP	Rank1	mAP
R18	Adam	91.7	77.8	82.5	68.8
	W-Adam	91.8	79.2	83.5	70.4
	Δ	$\uparrow$0.1	$\uparrow$1.4	$\uparrow$1.0	$\uparrow$1.6
R50	Adam	94.5	85.9	86.4	76.4
	W-Adam	94.5	86.5	87.5	77.2
	Δ	$\uparrow$0.0	$\uparrow$0.6	$\uparrow$1.1	$\uparrow$0.8
R101	Adam	94.5	87.1	87.6	77.6
	W-Adam	95.0	87.9	88.2	78.3
	Δ	$\uparrow$0.5	$\uparrow$0.8	$\uparrow$0.6	$\uparrow$0.7

Table 6.9: Testing accuracy (%) of ResNet18 by W-SGDM on CIFAR100 w.r.tϵ.

ϵ	$1e^{-5}$	$1e^{-4}$	$\mathbf{1e^{-3}}$	$1e^{-2}$	$1e^{-1}$
Acc	78.97	79.05	**79.28**	78.88	78.50

6.4.4 Person Re-identification

In this section, we use two widely used Person ReID benchmarks, Market1501 [114] and DukeMTMC-ReID [73], to show that W-Adam can also be easily adopted into pre-trained models without extra hyperparameter tuning. In this task, the Adam with L_2 regularization weight decay usually outperforms other optimizers and its hyperparameters have been well-tuned. The person ReID baselines in [52] are used[3]. The default hyperparameters of Adam, such as learning rate and weight decay, are directly applied to W-Adam. The experiments are repeated 4 times, and the average results are reported. Table 6.8 shows the Rank1 and mAP on Market1501 and DukeMTMC-ReID with ResNet18, ResNet50, ResNet101 backbones. It is clear that W-Adam outperforms much Adam, especially in mAP. This experiment again demonstrates the advantages of EFW as a general DNN optimization technique.

[3]`https://github.com/michuanhaohao/reid-strong-baseline`

6.4.5 Ablation study

Hyperparameter Tuning:

We first tune the damping parameter ϵ. A too small ϵ cannot improve the condition number of the statistic matrix, while a too large ϵ will suppress the useful information in the second order statistics. The results of ResNet18 trained by W-SGDM on CIFAR100 with different ϵ in $\{1e^{-5}, 1e^{-4}, 1e^{-3}, 1e^{-2}, 1e^{-1}\}$ are shown in Table 6.9. We can clearly see that $\epsilon = 1e^{-3}$ is the best choice and we adopt it in our experiments.

We then tune T_{xx} and T_{svd} to balance the performance and efficiency. Because of the high computational cost of SVD decomposition, T_{svd} should be set larger than T_{xx}. We test six combinations of T_{xx} and T_{svd}, and report their testing accuracies and training time per epoch (sec/epoch) in Table 6.10. We see that the combination of $T_{xx} = 50$ and $T_{svd} = 500$ can balance the performance and efficiency well, and it is chosen as our default setting in the proposed EFW.

Training Efficiency:

We conduct experiments on CIFAR100 with ResNet18 to study the training efficiency of EFW. To better evaluate the efficiency of EFW, we compare it with a representative whitening method ND [99] by applying them to SGD and AdamW. For EFW, we test two settings: $T_{xx} = 50$ and $T_{svd} = 500$ (setting 1), and $T_{xx} = 200$ and $T_{svd} = 2000$ (setting 2). Table 6.11 shows the testing accuracies and training time of the original SGDM/AdamW, ND and our EFW.

ND balances the accuracy and efficiency by adjusting the sampling stride, while EFW trades off the accuracy and training time by adjusting T_{xx} and T_{svd}. From Table 6.11, we can see that EFW costs less than 30% additional training time over the original SGDM/AdamW but achieves convincing performance gain over them. Setting 2 is faster but its accuracy is lower than setting 1 (default). In contrast, the whitening

Table 6.10: Testing accuracy (%) and training efficiency of ResNet18 by W-SGDM and W-Adam on CIFAR100 w.r.tT_{xx} and T_{svd}.

Algorithm	T_{xx} / T_{svd}	5 / 50	10 / 100	20 / 200	**50** / **500**	100 / 1000	200 / 2000
W-SGDM	Acc	79.40	79.33	79.23	**79.28**	79.11	79.02
	Sec/epoch	85.50	58.32	38.93	**29.78**	26.03	24.25
W-Adam	Acc	78.84	78.79	78.76	**78.75**	78.67	78.40
	Sec/epoch	90.17	60.1	40.9	**30.18**	27.14	25.55

Table 6.11: Testing accuracy (%) and training efficiency (sec/epoch) of ResNet18 by SGDM/AdamW, ND [99] and EFW on CIFAR100.

Algorithm		Original	ND stride=1	ND stride=3	EFW setting 1	EFW setting 2
SGDM	Acc	77.20	79.07	78.65	79.28	79.02
	Sec/epoch	23.45	184.23	65.37	29.78	24.25
AdamW	Acc	77.23	78.02	77.82	78.75	78.40
	Sec/epoch	24.21	188.97	66.25	30.18	25.55

method ND with stride 1 or stride 3 (default) consumes more than twice the training time than original SGDM/AdamW, while its performance gain is not as solid as EFW. Clearly, EFW is much more efficient to perform feature whitening and achieves a more favorable performance boost than ND. It overcomes the major drawbacks of whitening methods and inherits their advantages, providing a practical solution to perform feature whitening under the deep learning framework.

6.5 Conclusion

In this chapter, we proposed a novel DNN optimization technique, namely Embedded Feature Whitening (EFW), to address the drawbacks of conventional feature whitening methods, such as inapplicable to pre-trained DNN models, large computation cost, extra parameter introduction, and so on. Different from the existing feature whitening methods, which usually perform a whitening operation on features during forward propagation, EFW only adjusts the gradient of weight with the whitening

matrix without changing the forward and backward propagation processes of DNN model training. Meanwhile, we developed the associated momentum, statistics matrix computation, adaptive damping and gradient norm recovery techniques to make EFW effective and efficient to use. By adopting EFW to the popular SGDM and Adam optimizers, the resulting W-SGDM and W-Adam methods demonstrated superior performance to the leading DNN optimizers in various vision tasks with acceptable extra computation, including image classification, detection, segmentation and person ReID.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Optimization methods play an essential role in deep learning, while it is very challenging to train a very deep neural network (DNN) because of the non-linearity of network architecture, non-convexity of objectives function, too many local minima and saddle points, gradient vanishing and exploding, *etc*. A desired optimization technique or algorithm should speed up the training process and improve the final generalization performance. Most existing optimization algorithms are mainly based on the gradient descent technique due to its efficiency and effectiveness. It updates the parameters along the opposite direction of gradient in each iteration, which contains three steps: forward propagation, backward propagation, and updating parameters. A lot of optimization methods have been developed from different perspectives, including feature normalization, weight constraints, gradient constraints, optimization algorithm, learning rate schedule, and so on. In this thesis, we made some attempts to design reliable optimization methods on feature normalization, gradient constraints, optimization algorithms.

Batch Normalization (BN) can largely improve the effectiveness and efficiency in

training various types of DNNs. Nevertheless, the working mechanism of BN is not fully revealed yet. In Chapter 2, we first revealed that the generalization capability of BN comes from its noise generation mechanism in training, and then presented an explicit regularization formulation of BN to explain how it works. Consequently, we presented a modified version of BN, namely Momentum Batch Normalization (MBN). By using a dynamic momentum parameter, the noise level in the estimated mean and variance can be well controlled in MBN. What's more, the conventional inference method of BN, *i.e*, EMA, makes the training and inference stages of BN have a certain difference. To reduce the statistics disparity of BN between training and inference, in Chapter 3, we proposed an effective inference approach of BN, namely batch statistics regression (BSR). Without changing the training of BN, the proposed BSR does not need to change the training of BN, and it only needs to store four statistics in training with negligible cost of computation and memory. Both MBN and BSR improve BN largely, especially when training with a small batch size.

For gradient constraints, in Chapter 4, we proposed a simple but effective approach that operates directly on gradients. To be specific, it removes the mean from the gradient vectors and centralizes them to have zero mean. We called it Gradient Centralization (GC), which can not only smooth and accelerate the training process of DNN but also improve the model generalization performance. Meanwhile, we showed that GC actually constrains the loss function by introducing a new constraint on the weight vector, which regularizes both weight space and output feature space. We also showed that this constrained loss function has better Lipschitzness than the original one so that it makes the training process more stable and efficient. We performed comprehensive experiments to demonstrate that GC can consistently improve the performance of learned DNN models in different applications. It is a simple, general and effective optimization technique for deep learning.

For the weight update algorithm, based on the observation that performing weight gradient descent tends to update the features into a low dimensional space so that

the feature learning efficacy is reduced, in Chapter 5 we proposed a new DNN optimizer, namely feature stochastic gradient descent (FSGD), which takes the gradient of the intermediate features into consideration to update weights. We calculated the second-order statistic matrix of the input features in a liner layer and adopted its inverse to modify the gradient of weight. Compared with existing DNN optimizers that perform gradient descent on weight, FSGD can follow more informative directions to reach a favorable feature space. Moreover, to address the drawbacks of conventional feature whitening methods, in Chapter 6 we proposed a novel DNN optimization technique, namely Embedded Feature Whitening (EFW). Different from the existing feature whitening methods, which usually use a whitening operation on features during forward propagation, EFW only adjusts the gradient of weight with the whitening matrix without changing the forward and backward propagation during training. EFW works effectively and efficiently. We applied it to two popular optimizers, *i.e*, SGDM and Adam, and the resulting W-SGDM and W-Adam optimizers achieve superior performance in various vision tasks.

7.2 Future Work

In future work, we will further conduct our study in the following possible directions:

- The first two works in this thesis, *i.e*, MBN and BSR, only aim to improve BN. There are many other feature normalization methods besides BN. Investigating more effective feature normalization for DNN optimization is still a hot research topic, and we will investigate more powerful feature normalization methods.

- Our current implementation of GC only considers the mean of the gradients. How to extend it with more constraints needs more investigation, *e.g*, constraints on the high-order statistics of the gradient. In the future, we will investigate high-order statistics constraints on the gradient of weights.

- The proposed FSGD and EFW consider the three conventional types of layer, *i.e*, fully-connected layer, common convolutional layer, and normalization layer. In the future, we will extend them on more types of layers, *i.e*, self-attention layer and DW Conv layer.

- We will apply our proposed methods on more tasks, such as image restoration, action recognition, speech recognition, visual object tracking, industrial defect detection, person ReID, crowd counting, *etc*.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM, 2016.

[2] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. Youtube-8m: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675*, 2016.

[3] Guozhong An. The effects of adding noise during backpropagation training on a generalization performance. *Neural computation*, 8(3):643–674, 1996.

[4] Christopher M. Bishop. Training with noise is equivalent to tikhonov regularization. *Neural Computation*, 7(1):108–116, 1995.

[5] Johan Bjorck, Carla Gomes, Bart Selman, and Kilian Q. Weinberger. Understanding batch normalization. pages 7694–7705, 2018.

[6] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[7] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[8] Miguel Carreira-Perpinan and Weiran Wang. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, pages 10–19. PMLR, 2014.

[9] 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.

[10] 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.

[11] Minhyung Cho and Jaehyung Lee. Riemannian approach to batch normalization. In *Advances in Neural Information Processing Systems*, pages 5225–5235, 2017.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[14] Linda Crocker and James Algina. *Introduction to classical and modern test theory.* ERIC, 1986.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

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

[17] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[18] Akhilesh Gotmare, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. A closer look at deep learning heuristics: Learning rate restarts, warmup and distillation. *arXiv preprint arXiv:1810.13243*, 2018.

[19] 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.

[20] 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.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[22] 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.

[23] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. In *Advances in Neural Information Processing Systems*, pages 1731–1741, 2017.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[25] 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.

[26] Lei Huang, Xianglong Liu, Bo Lang, Adams Wei Yu, Yongliang Wang, and Bo Li. Orthogonal weight normalization: Solution to optimization over multiple dependent stiefel manifolds in deep neural networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] Lei Huang, Xianglong Liu, Yang Liu, Bo Lang, and Dacheng Tao. Centered weight normalization in accelerating training of deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2803–2811, 2017.

[28] Lei Huang, Dawei Yang, Bo Lang, and Jia Deng. Decorrelated batch normalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 791–800, 2018.

[29] Lei Huang, Yi Zhou, Fan Zhu, Li Liu, and Ling Shao. Iterative normalization: Beyond standardization towards efficient whitening. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4874–4883, 2019.

[30] Xun Huang and Serge Belongie. Arbitrary style transfer in real-time with adaptive instance normalization. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1501–1510, 2017.

[31] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In *Advances in neural information processing systems*, pages 1945–1953, 2017.

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

[33] 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.

[34] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[35] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Fei-Fei Li. Novel dataset for fgvc: Stanford dogs. In *San Diego: CVPR Workshop on FGVC*, volume 1, 2011.

[36] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1646–1654, 2016.

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

[38] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3d object representations for fine-grained categorization. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 554–561, 2013.

[39] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[40] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

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

[42] Måns Larsson, Anurag Arnab, Fredrik Kahl, Shuai Zheng, and Philip Torr. A projected gradient descent method for crf inference allowing end-to-end training of arbitrary pairwise potentials. In *International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 564–579. Springer, 2017.

[43] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

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

[45] R V Lenth. cumulative distribution function of the non-central t distribution. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 38(1):185–189, 1989.

[46] Yanghao Li, Naiyan Wang, Jianping Shi, Xiaodi Hou, and Jiaying Liu. Adaptive batch normalization for practical domain adaptation. *Pattern Recognition*, 80:109–117, 2018.

[47] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[48] 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.

[49] 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.

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

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

[52] Hao Luo, Youzhi Gu, Xingyu Liao, Shenqi Lai, and Wei Jiang. Bag of tricks and a strong baseline for deep person re-identification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[53] Hao Luo, Wei Jiang, Youzhi Gu, Fuxu Liu, Xingyu Liao, Shenqi Lai, and Jianyang Gu. A strong baseline and batch normalization neck for deep person re-identification. *IEEE Transactions on Multimedia*, 2019.

[54] Ping Luo, Zhanglin Peng, Jiamin Ren, and Ruimao Zhang. Do normalization layers in a deep convnet really need to be distinct? *arXiv preprint arXiv:1811.07727*, 2018.

[55] Ping Luo, Jiamin Ren, and Zhanglin Peng. Differentiable learning-to-normalize via switchable normalization. *arXiv preprint arXiv:1806.10779*, 2018.

[56] Ping Luo, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. Towards understanding regularization in batch normalization. 2018.

[57] Ping Luo, Peng Zhanglin, Shao Wenqi, Zhang Ruimao, Ren Jiamin, and Wu Lingyun. Differentiable dynamic normalization for learning deep representation. In *International Conference on Machine Learning*, pages 4203–4211, 2019.

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

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

[60] Subhransu Maji, Esa Rahtu, Juho Kannala, Matthew Blaschko, and Andrea Vedaldi. Fine-grained visual classification of aircraft. *arXiv preprint arXiv:1306.5151*, 2013.

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

[62] 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.

[63] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[64] Jorge Nocedal and Stephen J Wright. Sequential quadratic programming. *Numerical optimization*, pages 529–562, 2006.

[65] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. Understanding the exploding gradient problem. *CoRR, abs/1211.5063*, 2, 2012.

[66] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318, 2013.

[67] 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.

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

[69] Siyuan Qiao, Huiyu Wang, Chenxi Liu, Wei Shen, and Alan Yuille. Weight standardization. *arXiv preprint arXiv:1903.10520*, 2019.

[70] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[71] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

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

[73] 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.

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

[75] 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.

[76] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

[77] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? (no, it is not about internal covariate shift). pages 2483–2493, 2018.

[78] Wenqi Shao, Tianjian Meng, Jingyu Li, Ruimao Zhang, Yudian Li, Xiaogang Wang, and Ping Luo. Ssn: Learning sparse switchable normalization via sparsestmax. *arXiv preprint arXiv:1903.03793*, 2019.

[79] Alexander Shekhovtsov and Boris Flach. Stochastic normalizations as bayesian learning. In *Asian Conference on Computer Vision*, pages 463–479. Springer, 2018.

[80] Aliaksandr Siarohin, Enver Sangineto, and Nicu Sebe. Whitening and coloring batch transform for gans. *arXiv preprint arXiv:1806.00420*, 2018.

[81] 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.

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

[83] Saurabh Singh and Abhinav Shrivastava. Evalnorm: Estimating batch normalization statistics for evaluation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3633–3641, 2019.

[84] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[85] Cecilia Summers and Michael J Dinneen. Four things everyone should know to improve batch normalization. *arXiv preprint arXiv:1906.03548*, 2019.

[86] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[87] M Teye, H Azizpour, and K Smith. Bayesian uncertainty estimation for batch normalized deep networks. *arXiv preprint arXiv:1802.06455*, 2018.

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

[89] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Daan Wierstra, et al. Matching networks for one shot learning. In *Advances in neural information processing systems*, pages 3630–3638, 2016.

[90] Eugene Vorontsov, Chiheb Trabelsi, Samuel Kadoury, and Chris Pal. On orthogonality and learning recurrent networks with long term dependencies. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3570–3578. JMLR. org, 2017.

[91] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The caltech-ucsd birds-200-2011 dataset. 2011.

[92] Jiayun Wang, Yubei Chen, Rudrasis Chakraborty, and Stella X Yu. Orthogonal convolutional neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11505–11515, 2020.

[93] Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, 2018.

[94] Di Xie, Jiang Xiong, and Shiliang Pu. All you need is beyond a good init: Exploring better solution for training extremely deep convolutional neural networks with orthonormality and modulation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6176–6185, 2017.

[95] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[96] Junjie Yan, Ruosi Wan, Xiangyu Zhang, Wei Zhang, Yichen Wei, and Jian Sun. Towards stabilizing batch statistics in backward propagation of batch normalization. *arXiv preprint arXiv:2001.06838*, 2020.

[97] Zhewei Yao, Amir Gholami, Sheng Shen, Mustafa Mustafa, Kurt Keutzer, and Michael W Mahoney. Adahessian: An adaptive second order optimizer for machine learning. *arXiv preprint arXiv:2006.00719*, 2020.

[98] Zhuliang Yao, Yue Cao, Shuxin Zheng, Gao Huang, and Stephen Lin. Cross-iteration batch normalization. *arXiv preprint arXiv:2002.05712*, 2020.

[99] 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.

[100] Hongwei Yong, Jianqiang Huang, Xiansheng Hua, and Lei Zhang. Gradient centralization: A new optimization technique for deep neural networks. *arXiv preprint arXiv:2004.01461*, 2020.

[101] Hongwei Yong, Jianqiang Huang, Deyu Meng, Xiansheng Hua, and Lei Zhang. Momentum batch normalization for deep learning with small batch size. In *Proceedings of the European Conference on Computer Vision*, 2020.

[102] Hongwei Yong, Jianqiang Huang, Deyu Meng, Xiansheng Hua, and Lei Zhang. Momentum batch normalization for deep learning with small batch size. In *European Conference on Computer Vision*, pages 224–240. Springer, 2020.

[103] Hongwei Yong, Deyu Meng, Wangmeng Zuo, and Lei Zhang. Robust online matrix factorization for dynamic background subtraction. *IEEE transactions on pattern analysis and machine intelligence*, 40(7):1726–1740, 2017.

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

[105] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[106] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*, 2018.

[107] Huishuai Zhang, Wei Chen, and Tie-Yan Liu. On the local hessian in back-propagation. *Advances in Neural Information Processing Systems*, 31:6520–6530, 2018.

[108] 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.

[109] 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.

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

[111] Ruimao Zhang, Zhanglin Peng, Lingyun Wu, Zhen Li, and Ping Luo. Exemplar normalization for learning deep representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12726–12735, 2020.

[112] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[113] Hongyi Zheng, Hongwei Yong, and Lei Zhang. Deep convolutional dictionary learning for image denoising. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 630–641, 2021.

[114] 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.

[115] 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.