Efficient and Scalable Deep Learning

by

Wei Wen

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____

Hai Li, Advisor

_____

Yiran Chen

_____

Arthur Robert Calderbank

_____

Yangqing Jia

_____

Vahid Tarokh

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2019

ABSTRACT

Efficient and Scalable Deep Learning

by

Wei Wen

Department of Electrical and Computer Engineering
Duke University

Date: _____

Approved:

_____

Hai Li, Advisor

_____

Yiran Chen

_____

Arthur Robert Calderbank

_____

Yangqing Jia

_____

Vahid Tarokh

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Electrical and Computer Engineering
in the Graduate School of
Duke University

2019

Abstract

Deep Neural Networks (DNNs) can achieve accuracy superior to traditional machine learning models, because of their large learning capacity and the availability of large amounts of labeled data. In general, larger DNNs can obtain higher accuracy. However, there are two obstacles which hinder us building larger DNNs: (1) inference of large DNNs is slow which limits their deployment to small devices; (2) training large DNNs is also slow which slows down research exploration. To remove those obstacles, this dissertation focuses on acceleration of DNN inference and training.

To accelerate DNN inference, original DNNs are compressed while keeping original accuracy. More specific, Structurally Sparse Deep Neural Networks (SSDNNs) are proposed to remove neural components. In Convolutional Neural Networks (CNNs), neurons, filters, channels and layers can be removed; in Recurrent Neural Networks (RNNs), hidden sizes can be reduced. The study shows that SSDNNs can achieve higher speedup than sparse DNNs which have non-structured sparsity. Besides SSDNNs, a Force Regularization is proposed to enforce DNNs to lower-rank space, such that DNNs can be decomposed to lower-rank architectures with fewer ranks than traditional methods. The dissertation also demonstrates that SSDNNs and Force Regularization are orthogonal and can be combined for higher speedup.

To accelerate DNN training, distributed deep learning is required. However, two problems hinder us using more compute nodes for higher training speed: Communication Bottleneck and Generalization Gap. Communication Bottleneck is that

communication time will increase and dominate when the distributed systems scale to many compute nodes. To reduce gradient communication in Stochastic Gradient Descent (SGD), SGD with low-precision gradients (*TernGrad*) is proposed. Moreover, in distributed deep learning, a large batch size is required to exploit system computing power; unfortunately, accuracy will decrease when the batch size is very large, which is referred to as the Generalization Gap. One hypothesis to explain Generalization Gap is that large-batch SGD sticks at sharp minima. The dissertation proposes a stochastic smoothing (*SmoothOut*) to escape sharp minima. The dissertation will show that *TernGrad* overcomes Communication Bottleneck and *SmoothOut* helps to close the Generalization Gap.

Acknowledgements

During my Ph.D. program, I received enormous advice, help and support from my advisors, professors, mentors, colleagues, friends and family. I first thank my advisor Hai Li and co-advisor Yiran Chen for accepting me as a Ph.D. student in their lab. Before joining the lab, I was working as a full-time software engineer and decided to apply a Ph.D. program in United States. With only half year part time to prepare Graduate Record Examinations and TOEFL English Examinations, my scores just met the line while most professors had a higher standard. My advisors overlooked my whole experiences beyond my English capability, and trustfully offered me the opportunity. Their offer was the only offer with funding support that I received during my application. Without their trust and unique insight, I cannot continue my passion in research and succeed as today. Working with my advisors is easy, they trained me to perform high-quality research and gave me full flexibility and independence to explore in a new field. They are advisors during research and friends in life.

Secondly, I thank professors in universities and mentors in companies. I thank Prof. Zhi-Hong Mao for serving as a committee member of my qualification examination at University of Pittsburgh, Prof. Guillermo Sapiro for serving during my preliminary examination at Duke University, and Prof. Robert Calderbank, Prof. Vahid Tarokh and Dr. Yangqing Jia for serving as my final Ph.D. dissertation committee. During my research internships, I was mentored and advised by excellent researchers and engineers. During my internship at Google Brain, I was mentored by Pieter-Jan

Kindermans working on Automated Machine Learning. In this project, I thank Gabriel Bender, Hanxiao Liu, Quoc Le, Esteban Real and Jonathon Shlens for giving me feedback during meetings to make the intern project more successful. During my internship at Facebook AI Infra, I was mentored by Yangqing Jia. I also received help from Jiyan Yang, Jongsoo Park and Bor-Yiing Su. They helped me to integrate my research *TernGrad* into Facebook AI Infra. During my internship at Microsoft Research, I was mentored by Yuxiong He, with acknowledgments to Fang Liu, Bin Hu, Samyam Rajbhandari, Minjia Zhang, Wenhan Wang, Jeff Rasley and Jacob Devlin. During my internship at HP Labs, I was mentored by Cong Xu. The intern project inspired my future research on distributed deep learning.

I thank co-authors of my published papers including Feng Yan, Qing Wu, Mark Barnell, Chunpeng Wu, Kent Nixon, Chi-Ruo Wu, Xiaofang Hu, Beiye Liu, Tsung-Yi Ho and Xin Li. Also, thank Sheng Li for giving me feedback on the Structured Sparsity Learning approach, and Ali Taylan Cemgil at Bogazici University for valuable suggestions on the *TernGrad* oral presentation and manuscript. My research cannot succeed without funding supports by NSF CCF-1744082, NSF CCF-1725456, DOE SC0017030, AFRL FA8750-15-2-0048, DARPA D13AP00042, ECCS-1202225, AFRL FA8750-15-1-0176, NSF 1253424, NSF XPS-1337198 and NSF CCF-1615475.

I appreciate many professors who invited me to their campuses or classes to share my research. Thank Prof. Ming Gu at UC Berkeley for inviting me to his Scientific Computing and Matrix Computations Seminar, Prof. Wei-Lun (Harry) Chao and Prof. Kilian Q. Weinberger for inviting me to their AI Seminar at Cornell University, and Prof. Yingyan Lin at Rice University for inviting me to her lecture. Without their kindness, I cannot outreach further and broadcast my research to a larger scope.

Last but not the least, I really thank my family for supporting me all the time during my student career. Thank my parents Mrs. Li and Mr. Wen for bringing me

up with full patience and selflessness. I am lucky to marry Mrs. Yandan and have a baby, who is serving as an alarm to wake me up early in the morning. They are my inner strengths to make me feel that everyday is a fresh wonderful day!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep neural networks (DNNs) made remarkable success [KSH12, GDDM14, SZ14, SLJ+15, HZRS16] by leveraging large-scale networks learning from a huge volume of data. In general, higher accuracy can be achieved by building larger DNNs. This is typically true for a specific neural architecture. However, there are two obstacles preventing us from achieving higher accuracy by building larger models. First, the ultimate goal of building large DNNs is to use them for inference. Running inference based on large DNNs is slow, which limits their usage in edge devices such as mobile phones, Virtual Reality devices, self-driving cars and autonomous drones. The reason is that those edge devices have limited compute and memory resources while requiring real-time response. Large DNN inference in high-end cloud servers can also be problematic, because the AI service requests can be in billions or even trillions per day. One research in this dissertation is to solve this efficiency problem for Efficient Deep Learning. The second obstacle blocking us building larger models is on the training side. Training large DNNs is also slow. To accelerate training, distributed systems are often used to parallelize computation, which is known as Distributed Deep Learning. However, Communication Bottleneck and Generalization Gap are problems which block us to scale up Distributed Deep Learning. Because of Communication Bottleneck, the training speed will saturate when the number of compute nodes

reaches a threshold; because of Generalization Gap, the accuracy will decrease when the number of batch size is very large in Distributed Deep Learning. Therefore, another research in this dissertation is to overcome Communication Bottleneck and Generalization Gap for Scalable Deep Learning.

Some related research in this dissertation was published at [WWW+16a, WXY+17, WXW+17, WHR+17, WWH+15, WWW+16b].

1.1 Efficient Deep Learning for Deep Neural Network Inference

As mentioned, deployment of large DNNs is computation-intensive. *Model Compression* [JVZ14, HMD15, WXW+17, LUW17] is a class of approaches of reducing the size of *Deep Neural Networks* (DNNs) to accelerate inference. To reduce computation, many studies are performed to compress the scale of DNN, including connection pruning [HPTD15, LWF+15] and low rank approximation [DSD+13, DZB+14, JVZ14, IRS+15, TXWE15]. This dissertation advances *Model Compression* by developing more effective connection pruning and low rank regularization methods.

Previous connection pruning methods [HPTD15, LWF+15] can prune connections in a non-structured way; however, the irregularity of sparse connections breaks the computation parallelizability and data locality, and can only achieve limited speedup which is far below theoretical speedup in general-purpose hardware platforms (such as GPUs and CPUs). This dissertation proposes Structured Sparsity Learning (SSL) method to learn Structurally Sparse Deep Neural Networks (SSDNNs), which preserve the regularity in sparse DNNs and therefore reach ideal speedup. More specific, we present a generic method to remove versatile neural components (such as neurons, filters, filter shapes, channels, layers and hidden states) in DNNs. Our method is simply implemented by group Lasso regularization and can learn SSDNNs much faster than

previous iterative pruning methods. Our SSL can learn sparse DNNs within the similar number of iterations of training DNNs from scratch. SSL is generic and applies to both Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) across multiple tasks including image classification, language modeling, question answering and so on. Another advantage of SSDNNs is that SSDNNs can be directly deployed to hardware and immediately receive speed gain without any hardware or software customization, which are demanded by previous research [HPTD15, LWF+15].

In recently proposed Low Rank Approximation (LRA) approaches [DSD+13, DZB+14, JVZ14, IRS+15, TXWE15], the DNN is trained first and then each trained weight tensor is decomposed and approximated by a product of smaller factors. Finally, fine-tuning is performed to restore the model accuracy. Low rank approximation is able to achieve practical speedups because it coordinates model parameters in dense matrixes and avoids the locality problem of non-structured sparsity regularization. This dissertation advances LRA by proposing a novel regularization – Force Regularization, which trains DNNs to lower-rank space such that previous LRA approaches [DSD+13, DZB+14, JVZ14, IRS+15, TXWE15] can achieve the same approximation accuracy using a smaller rank. A smaller rank means further compressed DNNs with faster inference.

This dissertation will also show that our SSL and Force Regularization can be combined for inference speed boosting.

1.2 Scalable Deep Learning for Deep Neural Network Training

As aforementioned, Communication Bottleneck and Generalization Gap are two problems that limit the scalability of distributed deep learning. In this section, I will introduce background on these problems and our motivations.

1.2.1 Communication Bottleneck in Distributed Training

The remarkable advances in deep learning is driven by data explosion and increase of model size. The training of large-scale models with huge amounts of data are often carried on distributed systems [DCM+12, AAB+16, CHW+13, RRWN11, CSAK14, XHD+15, MNSJ15, CLL+15, ZCL15], where data parallelism is adopted to exploit the compute capability empowered by multiple workers [Li17]. *Stochastic Gradient Descent* (SGD) is usually selected as the optimization method because of its high computation efficiency. In realizing the data parallelism of SGD, model copies in computing workers are trained in parallel by applying different subsets of data. A centralized parameter server performs *gradient synchronization* by collecting all gradients and averaging them to update parameters. The updated parameters will be sent back to workers, that is, *parameter synchronization*. Increasing the number of workers helps to reduce the computation time dramatically. However, as the scale of distributed systems grows up, the extensive gradient and parameter synchronizations prolong the communication time and even amortize the savings of computation time [RRWN11, LAP+14, LASY14]. A common approach to overcome such a network bottleneck is *asynchronous SGD* [DCM+12, RRWN11, MNSJ15, LASY14, HCC+13, ZWLS10], which continues computation by using stale values without waiting for the completeness of synchronization. The inconsistency of parameters across computing workers, however, can degrade training accuracy and incur occasional divergence [PCM+17, ZGLL16]. Moreover, its workload dynamics make the training nondeterministic and hard to debug.

From the perspective of inference acceleration, sparse and quantized *Deep Neural Networks* (DNNs) have been widely studied, such as [HMD15, WWW+16a, PLW+17, HCS+16, RORF16, ZWN+16, WHR+17, OLZ+16, LCMB15]. However, these methods generally aggravate the training effort. Researches such as sparse logistic regression

and Lasso optimization problems [RRWN11, LASY14, BKBG11] took advantage of the sparsity inherent in models and achieved remarkable speedup for distributed training. A more generic and important topic is how to accelerate the distributed training of dense models by utilizing sparsity and quantization techniques. For instance, Aji and Heafield [AH17] proposed to heuristically sparsify dense gradients by dropping off small values in order to reduce gradient communication. For the same purpose, quantizing gradients to low-precision values with smaller bit width has also been extensively studied [ZWN+16, SFD+14, AGL+17, GAGN15]. This dissertation proposes a method that belongs to the category of gradient quantization, which is an orthogonal approach to sparsity methods.

More specific, we propose *TernGrad* to aggressively quantize gradients to three discrete values, such that only 2 bits are required to encode each gradient which originally requires 32 bits in floating precision. In this way, we can reduce communication volume by 16× to overcome the Communication Bottleneck. In this dissertation, a performance model is also introduced to estimate the training speed in distributed deep learning. The performance model is adopted by follow-up research [LHM+18] to analyze the scalability.

1.2.2 Generalization Gap in Large-batch Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is the dominant optimization method used to train *Deep Neural Networks* (DNNs). However, the generalization of DNNs needs more understanding. Recently, one observation is that large-batch SGD has worse generalization than small-batch SGD [DSC+16][KMN+17][GDG+17]. The accuracy difference between small-batch training and large-batch training is the well known "Generalization Gap" [KMN+17]. The Generalization Gap hinders the scalability of

distributed deep learning, which often depends on a very large batch size in SGD for acceleration.

Reasons behind the Generalization Gap are still under active research. Hoffer *et al.* [HHS17] hypothesizes that the process of SGD is similar to "random walk on a random potential" [BG90]. This hypothesis attributes generalization gap to the limited number of parameter updates, and suggests to train more iterations. Learning Rate Scaling (LRS) was also proposed to match walk statistics to close the gap. Inspired by this hypothesis, practical techniques are proposed [GDG$^+$17][SKL18][YGG17][ASF17]. Another appealing hypothesis, which arouses recent research interest, is that the generalization is attributed to the flatness of minima [HS97a][CCSL17]; that is, flat minima have good generalization while sharp minima can worsen it. The hypothesis can be applied to both small-batch and large-batch SGD, but large-batch SGD tends to converge to sharper minima, ending up with the generalization gap. Sharp minima have bad generalization due to their high over-fitting to training data [HS97a][Ris78] and high sensitivity to noises [CCSL17]. Jastrzębski *et al.* [JKA$^+$18] showed the connection between these two hypotheses: LRS motivated by "random walk" leads to flatter minima and helps to improve the generalization. This dissertation is based on the second hypothesis, targeting on escaping sharp minima for better generalization in large-batch SGD to scale up distributed deep learning. However, the method can be applied to small-batch SGD to escape sharp minima. Moreover, our approach can enhance techniques inspired by the first hypothesis and *further* improve generalization.

Keskar *et al.* [KMN$^+$17] attempted to escape sharp minima through data augmentation, conservative training and adversarial training. However, all trials "do not completely remedy the problem" [KMN$^+$17], leaving how to avoid sharp minima as an open question. We propose *SmoothOut* to smooth out sharp minima and guide the convergence of SGD to flatter regions.

Chapter 2

Structurally Sparse Deep Neural Networks

In this chapter, the inefficiency of sparse neural networks under non-structured sparsity is first shown in Section 2.1. Thus, Structured Sparsity Learning (SSL) is proposed to learn structurally sparse neural networks. The effectiveness of SSL is evaluated in both Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs).

2.1 The Inefficiency of Non-structured Sparsity

Sparsity regularization and connection pruning can reduce model sizes, however, they often produce non-structured random connectivity and thus, irregular memory access that adversely impacts *practical* acceleration in hardware platforms. Figure 2.1 depicts practical layer-wise speedup of *AlexNet*, which is non-structurally sparsified by ℓ_1-norm. Compared to original model, the accuracy loss of the sparsified model is controlled within 2%. Because of the poor data locality associated with the scattered weight distribution, the achieved speedups are either very limited or negative even the actual sparsity is high, say, >95%. We define sparsity as the ratio of zeros in this dissertation.

The inefficiency problem also exists in CPU platforms and Recurrent Neural

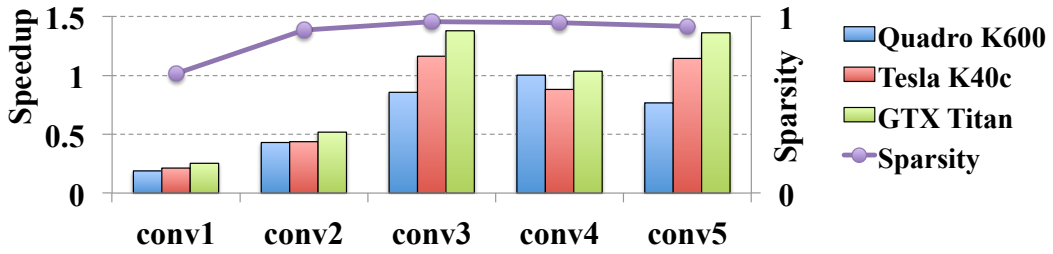


Figure 2.1: Evaluation speedups of AlexNet on GPU platforms and the sparsity. conv1 refers to convolutional layer 1, and so forth. Baseline is profiled by GEMM of cuBLAS. The sparse matrixes are stored in the format of Compressed Sparse Row (CSR) and accelerated by cuSPARSE.

Networks. A recent work by [NDSE17] proposes a pruning approach that deletes up to 90% connections in RNNs. Connection pruning methods sparsify weights of recurrent units but cannot explicitly change *basic structures*, *e.g.*, the number of *input updates, gates, hidden states, cell states and outputs*. The obtained sparse matrices have an irregular/non-structured pattern of non-zero weights, which is unfriendly for efficient computation in modern hardware systems ([LL16a]). Our study in Figure 2.1 [WWW+16a] on sparse matrix multiplication in GPUs showed that the speedup[1] was either counterproductive or ignorable. More specific, with sparsity[2] of 67.6%, 92.4%, 97.2%, 96.6% and 94.3% in weight matrices of *AlexNet*, the speedup was 0.25×, 0.52×, 1.38×, 1.04×, and 1.36×, respectively. This problem also exists in CPUs and Recurrent Neural Networks. Fig. 2.2 shows two LSTM examples running on CPUs, which show that non-structured pattern in sparsity limits the speedup. We only starts to observe speed gain when the sparsity is beyond 80%, and the speedup is about 3× to 4× even when the sparsity is 95% which is far below the theoretical 20×.

In this dissertation, we focus on learning structurally sparse CNNs and RNNs for computation efficiency. In CNNs, inspired by the facts that (1) there is redundancy across filters and channels [JVZ14]; (2) shapes of filters are usually fixed as cuboid

[1]Defined as (new speed)/(original speed).

[2]Defined as the percentage of zeros.

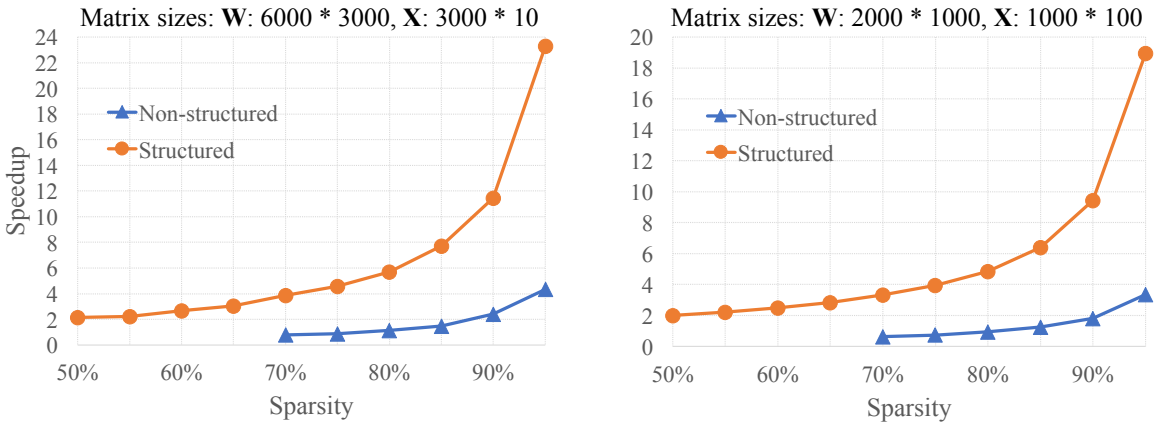


Figure 2.2: Speedups of matrix multiplication using non-structured and structured sparsity. Speeds are measured in Intel MKL implementations in Intel Xeon CPU E5-2673 v3 @ 2.40GHz. General matrix-matrix multiplication (GEMM) of $\mathbf{W} \cdot \mathbf{X}$ is implemented by `cblas_sgemm`. The matrix sizes are selected to reflect commonly used GEMMs in LSTMs. For example, (a) represents GEMM in LSTMs with hidden size 1500, input size 1500 and batch size 10. To accelerate GEMM by sparsity, $\mathbf{W}$ is sparsified. In non-structured sparsity approach, $\mathbf{W}$ is randomly sparsified and encoded as Compressed Sparse Row format for sparse computation (using `mkl_scsrmm`); in structured sparsity approach, $2k$ columns and $4k$ rows in $\mathbf{W}$ are removed to match the same level of sparsity (*i.e.,* the percentage of removed parameters) for faster GEMM under smaller sizes.

but enabling arbitrary shapes can potentially eliminate unnecessary computation imposed by this fixation; and (3) depth of the network is critical for classification but deeper layers cannot always guarantee a lower error because of the exploding gradients and degradation problem [HZRS16], we propose *Structured Sparsity Learning* (SSL) method to *directly* learn a compressed structure of deep CNNs by group Lasso regularization during the training. SSL is a generic regularization to adaptively adjust multiple structures in DNN, including structures of filters, channels, filter shapes within each layer, and structure of depth beyond the layers. SSL combines structure regularization (on DNN for classification accuracy) with locality optimization (on memory access for computation efficiency), offering not only well-regularized big models with improved accuracy but greatly accelerated computation (*e.g.*, 5.1× on CPU and 3.1× on GPU for *AlexNet*). Similarly, for RNNs, we aim to reduce the number of basic structures simultaneously during learning, such that the obtained

RNNs have the original schematic with dense connections but with smaller hidden sizes. Such compact models have structured sparsity, with columns and rows in weight matrices removed, whose computation efficiency is shown in Fig. 2.2. One advantage of structurally sparse DNNs over DNNs with non-structured sparsity is that, off-the-shelf libraries in deep learning frameworks can be directly utilized to deploy the reduced CNNs and RNNs. Details should be explained.

2.2 Structurally Sparse Convolutional Neural Networks

2.2.1 Structured Sparsity Learning by Group Lasso Regularization

We focus mainly on the *Structured Sparsity Learning* (SSL) on convolutional layers to regularize the structure of DNNs. We first propose a generic method to regularize structures of DNN, and then specify the method to structures of filters, channels, filter shapes and depth. Variants of formulations are also discussed from computational efficiency viewpoint.

Proposed structured sparsity learning for generic structures

Suppose the weights of convolutional layers in a DNN form a sequence of 4-D tensors $\boldsymbol{W}^{(l)} \in \mathbb{R}^{N_l \times C_l \times M_l \times K_l}$, where N_l, C_l, M_l and K_l are the dimensions of the l-th $(1 \leq l \leq L)$ weight tensor along the axes of filter, channel, spatial height and spatial width, respectively. L denotes the number of convolutional layers. Then the proposed generic optimization target of a DNN with structured sparsity regularization can be formulated as:

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda \cdot R(\boldsymbol{W}) + \lambda_g \cdot \sum_{l=1}^{L} R_g\left(\boldsymbol{W}^{(l)}\right). \tag{2.1}$$

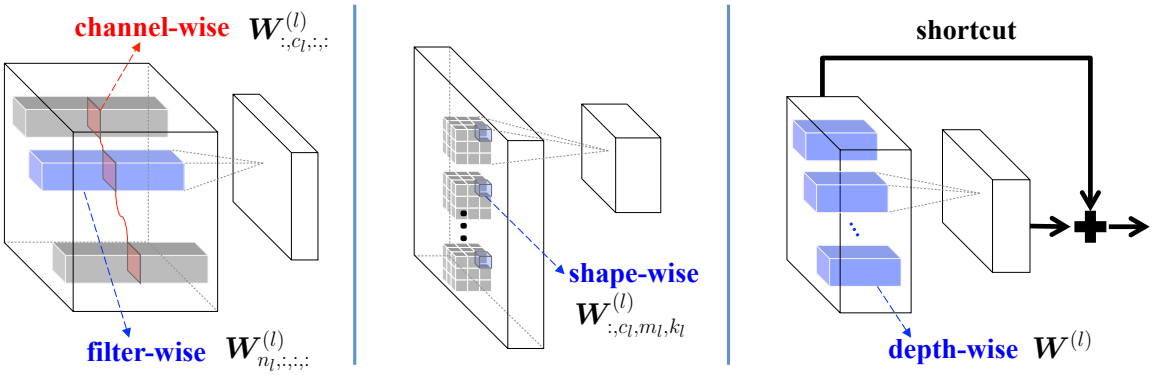


Figure 2.3: The proposed *Structured Sparsity Learning* (SSL) for DNNs. The weights in filters are split into multiple groups. Through group Lasso regularization, a more compact DNN is obtained by removing some groups. The figure illustrates the filter-wise, channel-wise, shape-wise, and depth-wise structured sparsity that are explored in the work.

Here $\boldsymbol{W}$ represents the collection of all weights in the DNN; $E_D(\boldsymbol{W})$ is the loss on data; $R(\cdot)$ is non-structured regularization applying on every weight, *e.g.*, ℓ_2-norm; and $R_g(\cdot)$ is the structured sparsity regularization on each layer. Because *group Lasso* can effectively zero out all weights in some groups [YL06][KX10], we adopt it in our SSL. The regularization of group Lasso on a set of weights $\boldsymbol{w}$ can be represented as $R_g(\boldsymbol{w}) = \sum_{g=1}^{G} ||\boldsymbol{w}^{(g)}||_g$, where $\boldsymbol{w}^{(g)}$ is a group of partial weights in $\boldsymbol{w}$ and G is the total number of groups. Different groups may overlap. Here $||\cdot||_g$ is the group Lasso, or $||\boldsymbol{w}^{(g)}||_g = \sqrt{\sum_{i=1}^{|\boldsymbol{w}^{(g)}|} \left(w_i^{(g)}\right)^2}$, where $|\boldsymbol{w}^{(g)}|$ is the number of weights in $\boldsymbol{w}^{(g)}$.

Structured sparsity learning for structures of filters, channels, filter shapes and depth

In SSL, the learned "structure" is decided by the way of splitting groups of $\boldsymbol{w}^{(g)}$. We investigate and formulate the *filer-wise*, *channel-wise*, *shape-wise*, and *depth-wise* structured sparsity in Figure 2.3. For simplicity, the $R(\cdot)$ term of Eq. (2.1) is omitted in the following formulation expressions.

Penalizing unimportant filers and channels. Suppose $\boldsymbol{W}^{(l)}_{n_l,:,:,:}$ is the n_l-*th*

filter and $\boldsymbol{W}^{(l)}_{:,c_l,:,:}$ is the c_l-*th* channel of all filters in the l-*th* layer. The optimization target of learning the filter-wise and channel-wise structured sparsity can be defined as

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_n \cdot \sum_{l=1}^{L} \left(\sum_{n_l=1}^{N_l} ||\boldsymbol{W}^{(l)}_{n_l,:,:,:}||_g \right) + \lambda_c \cdot \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||\boldsymbol{W}^{(l)}_{:,c_l,:,:}||_g \right). \quad (2.2)$$

As indicated in Eq. (2.2), our approach tends to remove less important filters and channels. Note that zeroing out a filter in the l-*th* layer results in a dummy zero output feature map, which in turn makes a corresponding channel in the $(l+1)$-*th* layer useless. Hence, we combine the filter-wise and channel-wise structured sparsity in the learning simultaneously.

Learning arbitrary shapes of filers. As illustrated in Figure 2.3, $\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}$ denotes the vector of all corresponding weights located at spatial position of (m_l, k_l) in the 2D filters across the c_l-*th* channel. Thus, we define $\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}$ as the *shape fiber* related to learning arbitrary filter shape because a homogeneous non-cubic filter shape can be learned by zeroing out some shape fibers. The optimization target of learning shapes of filers becomes:

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_s \cdot \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} \sum_{m_l=1}^{M_l} \sum_{k_l=1}^{K_l} ||\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}||_g \right). \quad (2.3)$$

Regularizing layer depth. We also explore the depth-wise sparsity to regularize the depth of DNNs in order to improve accuracy and reduce computation cost. The corresponding optimization target is $E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda_d \cdot \sum_{l=1}^{L} ||\boldsymbol{W}^{(l)}||_g$. Different from other discussed sparsification techniques, zeroing out all the filters in a layer will cut off the message propagation in the DNN so that the output neurons cannot perform any classification. Inspired by the structure of highway networks [SGS15] and deep residual networks [HZRS16], we propose to leverage the shortcuts across

layers to solve this issue. As illustrated in Figure 2.3, even when SSL removes an entire unimportant layers, feature maps will still be forwarded through the shortcut.

Structured sparsity learning for computationally efficient structures

All proposed schemes in section 2.2.1 can learn a compact DNN for computation cost reduction. Moreover, some variants of the formulations of these schemes can directly learn structures that can be efficiently computed.

2D-filter-wise sparsity for convolution. 3D convolution in DNNs essentially is a composition of 2D convolutions. To perform efficient convolution, we explored a fine-grain variant of filter-wise sparsity, namely, *2D-filter-wise* sparsity, to spatially enforce group Lasso on each 2D filter of $\boldsymbol{W}^{(l)}_{n_l,c_l,:,:}$. The saved convolution is proportional to the percentage of the removed 2D filters. The fine-grain version of filter-wise sparsity can more efficiently reduce the computation associated with convolution: Because the distance of weights (in a smaller group) from the origin is shorter, which makes group Lasso more easily to obtain a higher ratio of zero groups.

Combination of filter-wise and shape-wise sparsity for GEMM. Convolutional computation in DNNs is commonly converted to modality of *GEneral Matrix Multiplication* (GEMM) by lowering weight tensors and feature tensors to matrices [CWV+14]. For example, in Caffe [JSD+14], a 3D filter $\boldsymbol{W}^{(l)}_{n_l,:,:,:}$ is reshaped to a row in the weight matrix where each column is the collection of weights $\boldsymbol{W}^{(l)}_{:,c_l,m_l,k_l}$ related to shape-wise sparsity. Combining filter-wise and shape-wise sparsity can directly reduce the dimension of weight matrix in GEMM by removing zero rows and columns. In this context, we use *row-wise* and *column-wise* sparsity as the interchangeable terminology of *filter-wise* and *shape-wise sparsity*, respectively.

Table 2.1: Results after penalizing unimportant filters and channels in *LeNet*

LeNet #	Error	Filter # [§]	Channel # [§]	FLOP [§]	Speedup [§]
1 (*baseline*)	0.9%	20—50	1—20	100%—100%	1.00×—1.00×
2	0.8%	5—19	1—4	25%—7.6%	1.64×—5.23×
3	1.0%	3—12	1—3	15%—3.6%	1.99×—7.44×

[§]In the order of *conv1*—*conv2*

2.2.2 Experiments

We evaluate the effectiveness of our SSL using published models on three databases – MNIST, CIFAR-10, and ImageNet. Without explicit explanation, SSL starts with the network whose weights are initialized by the baseline, and speedups are measured in matrix-matrix multiplication by Caffe in a single-thread Intel Xeon E5-2630 CPU. Hyper-parameters are selected by cross-validation.

LeNet and multilayer perceptron on MNIST

In the experiment of MNIST, we examine the effectiveness of SSL in two types of networks: *LeNet* [LBBH98] implemented by Caffe and a *multilayer perceptron* (*MLP*) network. Both networks were trained without data augmentation.

***LeNet*:** When applying SSL to *LeNet*, we constrain the network with filter-wise and channel-wise sparsity in convolutional layers to penalize unimportant filters and channels. Table 2.1 summarizes the remained filters and channels, *floating-point operations* (FLOP), and practical speedups. In the table, *LeNet 1* is the baseline and the others are the results after applying SSL in different strengths of structured sparsity regularization. The results show that our method achieves the similar error ($\pm 0.1\%$) with much fewer filters and channels, and saves significant FLOP and computation time.

To demonstrate the impact of SSL on the structures of filters, we present all learned *conv1* filters in Figure 2.4. It can be seen that most filters in *LeNet 2* are

Table 2.2: Results after learning filter shapes in *LeNet*

LeNet #	Error	Filter size §	Channel #	FLOP	Speedup
1 (*baseline*)	0.9%	25—500	1—20	100%—100%	1.00×—1.00×
4	0.8%	21—41	1—2	8.4%—8.2%	2.33×—6.93×
5	1.0%	7—14	1—1	1.4%—2.8%	5.19×—10.82×

§ The sizes of filters after removing zero shape fibers, in the order of *conv1*—*conv2*

entirely zeroed out except for five most important detectors of stroke patterns that are sufficient for feature extraction. The accuracy of *LeNet 3* (that further removes the weakest and redundant stroke detector) drops only 0.2% from that of *LeNet 2*. Compared to the random and blurry filter patterns in *LeNet 1* which are resulted from the high freedom of parameter space, the filters in *LeNet 2 & 3* are regularized and converge to smoother and more natural patterns. This explains why our proposed SSL obtains the same-level accuracy but has much less filters. The smoothness of the filters are also observed in the deeper layers.

The effectiveness of the shape-wise sparsity on *LeNet* is summarized in Table 2.2. The baseline *LeNet 1* has *conv1* filters with a regular 5×5 square (size = 25) while *LeNet 5* reduces the dimension that can be constrained by a 2×4 rectangle (size = 7). The 3D shape of *conv2* filters in the baseline is also regularized to the 2D shape in *LeNet 5* within only one channel, indicating that only one filter in *conv1* is needed. This fact significantly saves FLOP and computation time.

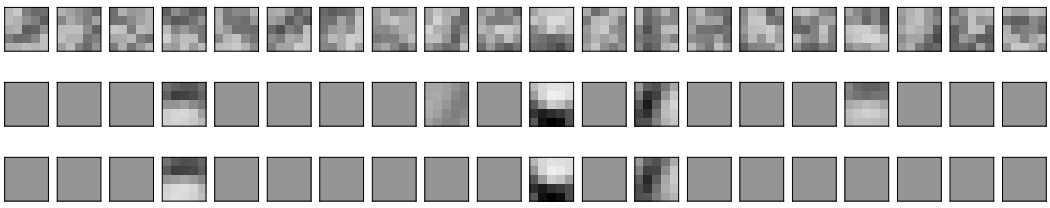


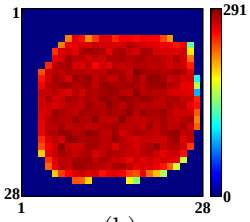
Figure 2.4: Learned *conv1* filters in *LeNet 1* (top), *LeNet 2* (middle) and *LeNet 3* (bottom)

***MLP*:** Besides convolutional layers, our proposed SSL can be extended to learn

MLP #	Error	Neuron # per layer [§]	FLOP per layer [§]
1 (*baseline*)	1.43%	784–500–300–10	100%–100%–100%
2	1.34%	469–294–166–10	35.18%–32.54%–55.33%
3	1.53%	434–174–78–10	19.26%–9.05%–26.00%

[§]In the order of *input layer–hidden layer 1–hidden layer 2–output layer*

(a)



(b)

Figure 2.5: (a) Results of learning the number of neurons in *MLP*. (b) the connection numbers of input neurons (*i.e.*, pixels) in *MLP 2* after SSL.

the structure (*i.e.*, the number of neurons) of fully-connected layers. We enforce the group Lasso regularization on all the input (or output) connections of each neuron. A neuron whose input connections are all zeroed out can degenerate to a bias neuron in the next layer; similarly, a neuron can degenerate to a removable dummy neuron if all of its output connections are zeroed out. Figure 2.5(a) summarizes the learned structure and FLOP of different *MLP* networks. The results show that SSL can not only remove hidden neurons but also discover the sparsity of images. For example, Figure 2.5(b) depicts the number of connections of each input neuron in *MLP 2*, where 40.18% of input neurons have zero connections and they concentrate at the boundary of the image. Such a distribution is consistent with our intuition: handwriting digits are usually written in the center and pixels close to the boundary contain little discriminative classification information.

ConvNet and *ResNet* on CIFAR-10

We implemented the *ConvNet* of [KSH12] and *deep residual networks* (*ResNet*) [HZRS16] on CIFAR-10. When regularizing filters, channels, and filter shapes, the results and observations of both networks are similar to that of the MNIST experiment. Moreover, we simultaneously learn the filter-wise and shape-wise sparsity to reduce the dimension of weight matrix in GEMM by *ConvNet*. We also learn the depth-wise sparsity of *ResNet* to regularize the depth of the DNNs.

Table 2.3: Learning row-wise and column-wise sparsity of *ConvNet* on CIFAR-10

ConvNet #	Error	Row sparsity §	Column sparsity §	Speedup §
1 (*baseline*)	17.9%	12.5%–0%–0%	0%–0%–0%	1.00×–1.00×–1.00×
2	17.9%	50.0%–28.1%–1.6%	0%–59.3%–35.1%	1.43×–3.05×–1.57×
3	16.9%	31.3%–0%–1.6%	0%–42.8%–9.8%	1.25×–2.01×–1.18×

§in the order of *conv1*–*conv2*–*conv3*

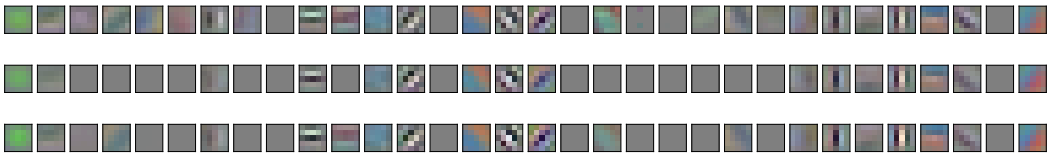


Figure 2.6: Learned *conv1* filters in *ConvNet 1* (top), *ConvNet 2* (middle) and *ConvNet 3* (bottom)

***ConvNet*:** We use the network from Alex Krizhevsky *et al.* [KSH12] as the baseline and implement it using Caffe. All the configurations remain the same as the original implementation except that we added a dropout layer with a ratio of 0.5 in the fully-connected layer to avoid over-fitting. *ConvNet* is trained without data augmentation. Table 2.3 summarizes the results of three *ConvNet* networks. Here, the row/column sparsity of a weight matrix is defined as the percentage of all-zero rows/columns. Figure 2.6 shows their learned *conv1* filters. In Table 2.3, SSL can reduce the size of weight matrix in *ConvNet 2* by 50%, 70.7% and 36.1% for each convolutional layer and achieve good speedups without accuracy drop. Surprisingly, without SSL, four *conv1* filters of the baseline are actually all-zeros as shown in Figure 2.6, demonstrating the great potential of filter sparsity. When SSL is applied, half of *conv1* filters in *ConvNet 2* can be zeroed out without accuracy drop.

On the other hand, in *ConvNet 3*, SSL lowers 1.0% (±0.16%) error with a model even smaller than the baseline. In this scenario, SSL performs as a structure regularization to dynamically learn a better network structure (including the number of filters and filer shapes) to reduce the error.

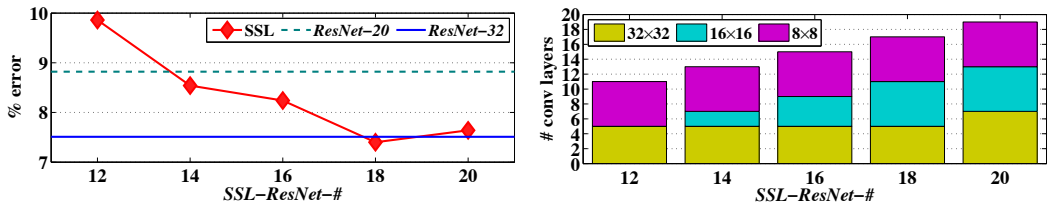


Figure 2.7: Error vs. layer number after depth regularization. # is the number of layers including the last fully-connected layer. *ResNet-#* is the *ResNet* in [HZRS16]. *SSL-ResNet-#* is the depth-regularized *ResNet* by SSL. 32×32 indicates the convolutional layers with an output map size of 32×32, *etc.*

***ResNet*:** To investigate the necessary depth of DNNs by SSL, we use a 20-layer deep residual networks (*ResNet-20*) [HZRS16] as the baseline. The network has 19 convolutional layers and 1 fully-connected layer. *Identity shortcuts* are utilized to connect the feature maps with the same dimension while 1×1 convolutional layers are chosen as shortcuts between the feature maps with different dimensions. Batch normalization [IS15] is adopted after convolution and before activation. We use the same data augmentation and training hyper-parameters as that in [HZRS16]. The final error of baseline is 8.82%. In SSL, the depth of *ResNet-20* is regularized by depth-wise sparsity. Group Lasso regularization is only enforced on the convolutional layers between each pair of shortcut endpoints, excluding the first convolutional layer and all convolutional shortcuts. After SSL converges, layers with all zero weights are removed and the net is finally fine-tuned with a base learning rate of 0.01, which is lower than that (*i.e.*, 0.1) in the baseline.

Figure 2.7 plots the trend of the error vs. the number of layers under different strengths of depth regularizations. Compared with original *ResNet* in [HZRS16], SSL learns a *ResNet* with 14 layers (*SSL-ResNet-14*) reaching a lower error than that of the baseline with 20 layers (*ResNet-20*); *SSL-ResNet-18* and *ResNet-32* achieve an error of 7.40% and 7.51%, respectively. This result implies that SSL can work as a depth regularization to improve classification accuracy. Note that SSL can efficiently learn shallower DNNs without accuracy loss to reduce computation cost; however, it

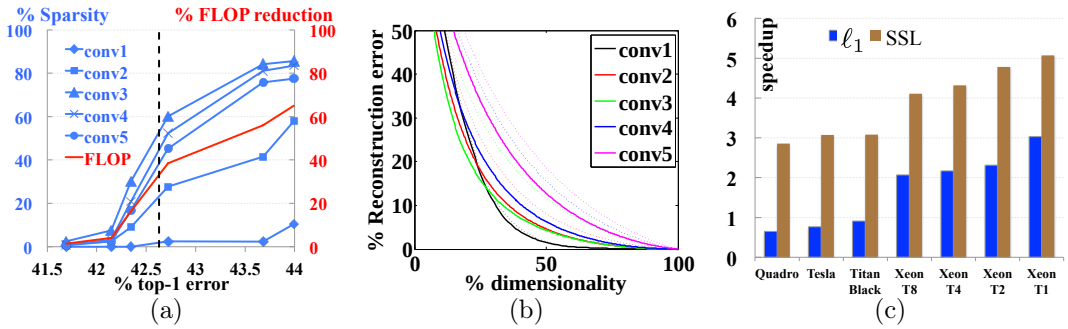


Figure 2.8: (a) 2D-filter-wise sparsity and FLOP reduction vs. top-1 error. Vertical dash line shows the error of original *AlexNet*; (b) The reconstruction error of weight tensor vs. dimensionality. *Principal Component Analysis* (PCA) is utilized to perform dimensionality reduction. The eigenvectors corresponding to the largest eigenvalues are selected as basis of lower-dimensional space. Dash lines denote the results of the baselines and solid lines indicate the ones of the *AlexNet 5* in Table 2.4; (c) Speedups of ℓ_1-norm and SSL on various CPUs and GPUs (In labels of x-axis, T# is the number of maximum physical threads in CPUs). *AlexNet 1* and *AlexNet 2* in Table 2.4 are used as testbenches.

does not mean the depth of the network is not important. The trend in Figure 2.7 shows that the test error generally declines as more layers are preserved. A slight error rise of *SSL-ResNet-20* from *SSL-ResNet-18* shows the suboptimal selection of the depth in the group of "32×32".

AlexNet on ImageNet

To show the generalization of our method to large scale DNNs, we evaluate SSL using *AlexNet* with ILSVRC 2012. *CaffeNet* [JSD+14], the replication of *AlexNet* [KSH12] with mirror changes, is used in our experiment. All training images are rescaled to the size of 256×256. A 227×227 image is randomly cropped from each scaled image and mirrored for data augmentation and only the center crop is used for validation. The final top-1 validation error is 42.63%. In SSL, *AlexNet* is first trained with structure regularization; when it converges, zero groups are removed to obtain a DNN with the new structure; finally, the network is fine-tuned without SSL to regain the accuracy.

We first study 2D-filter-wise and shape-wise sparsity by exploring the trade-offs

between computation complexity and classification accuracy. Figure 2.8(a) shows the 2D-filter sparsity (the ratio between the removed 2D filters and total 2D filters) and the saved FLOP of 2D convolutions vs. the validation error. In Figure 2.8(a), deeper layers generally have higher sparsity as the group size shrinks and the number of 2D filters grows. 2D-filter sparsity regularization can reduce the total FLOP by 30%–40% without accuracy loss or reduce the error of *AlexNet* by ∼1% down to 41.69% by retaining the original number of parameters. Shape-wise sparsity also obtains similar results. In Table 2.4, for example, *AlexNet 5* achieves on average 1.4× layer-wise speedup on both CPU and GPU without accuracy loss after shape regularization; The top-1 error can also be reduced down to 41.83% if the parameters are retained. In Figure 2.8(a), the obtained DNN with the lowest error has a very low sparsity, indicating that the number of parameters in a DNN is still important to maintain learning capacity. In this case, SSL works as a regularization to add restriction of smoothness to the model in order to avoid over-fitting. Figure 2.8(b) compares the results of dimensionality reduction of weight tensors in the baseline and our SSL-regularized *AlexNet*. The results show that the smoothness restriction enforces parameter searching in lower-dimensional space and enables *lower* rank approximation of the DNNs. Therefore, SSL can work together with low rank approximation to achieve even higher model compression.

Besides the above analyses, the computation efficiencies of structured sparsity and non-structured sparsity are compared in Caffe using standard off-the-shelf libraries, *i.e.*, Intel Math Kernel Library on CPU and CUDA cuBLAS and cuSPARSE on GPU. We use SSL to learn a *AlexNet* with high column-wise and row-wise sparsity as the representative of structured sparsity method. ℓ_1-norm is selected as the representative of non-structured sparsity method instead of connection pruning [HPTD15] because ℓ_1-norm get a higher sparsity on convolutional layers as the results of *AlexNet 3*

Table 2.4: Sparsity and speedup of *AlexNet* on ILSVRC 2012

#	Method	Top1 err.	Statistics	conv1	conv2	conv3	conv4	conv5
1	ℓ_1	44.67%	sparsity	67.6%	92.4%	97.2%	96.6%	94.3%
			CPU ×	0.80	2.91	4.84	3.83	2.76
			GPU ×	0.25	0.52	1.38	1.04	1.36
2	SSL	44.66%	column sparsity	0.0%	63.2%	76.9%	84.7%	80.7%
			row sparsity	9.4%	12.9%	40.6%	46.9%	0.0%
			CPU ×	1.05	3.37	6.27	9.73	4.93
			GPU ×	1.00	2.37	4.94	4.03	3.05
3	pruning [HPTD15]	42.80%	sparsity	16.0%	62.0%	65.0%	63.0%	63.0%
4	ℓ_1	42.51%	sparsity	14.7%	76.2%	85.3%	81.5%	76.3%
			CPU ×	0.34	0.99	1.30	1.10	0.93
			GPU ×	0.08	0.17	0.42	0.30	0.32
5	SSL	42.53%	column sparsity	0.00%	20.9%	39.7%	39.7%	24.6%
			CPU ×	1.00	1.27	1.64	1.68	1.32
			GPU ×	1.00	1.25	1.63	1.72	1.36

and *AlexNet 4* depicted in Table 2.4. Speedups achieved by SSL are measured by GEMM, where all-zero rows (and columns) in each weight matrix are removed and the remaining ones are concatenated in consecutive memory space. Note that compared to GEMM, the overhead of concatenation can be ignored. To measure the speedups of ℓ_1-norm, sparse weight matrices are stored in the format of Compressed Sparse Row (CSR) and computed by sparse-dense matrix multiplication subroutines.

Table 2.4 compares the obtained sparsity and speedups of ℓ_1-norm and SSL on CPU (Intel Xeon) and GPU (GeForce GTX TITAN Black) under approximately the same errors, *e.g.*, with acceptable or no accuracy loss. To make a fair comparison, after ℓ_1-norm regularization, the DNN is also fine-tuned by disconnecting all zero-weighted connections so that, *e.g.,* 1.39% accuracy is recovered for the *AlexNet 1*. Our experiments show that the DNNs require a very high non-structured sparsity to achieve a reasonable speedup (the speedups are even negative when the sparsity is low). SSL, however, can always achieve positive speedups. With an acceptable accuracy loss, our SSL achieves on average 5.1× and 3.1× layer-wise acceleration on

CPU and GPU, respectively. Instead, ℓ_1-norm achieves on average only 3.0× and 0.9× layer-wise acceleration on CPU and GPU, respectively. We note that, at the same accuracy, our average speedup is indeed higher than that of [LWF+15] which adopts heavy hardware customization to overcome the negative impact of non-structured sparsity. Figure 2.8(c) shows the speedups of ℓ_1-norm and SSL on various platforms, including both GPU (Quadro, Tesla and Titan) and CPU (Intel Xeon E5-2630). SSL can achieve on average $\sim 3\times$ speedup on GPU while non-structured sparsity obtain no speedup on GPU platforms. On CPU platforms, both methods can achieve good speedups and the benefit grows as the processors become weaker. Nonetheless, SSL can always achieve averagely $\sim 2\times$ speedup compared to non-structured sparsity.

2.3 Structurally Sparse Recurrent Neural Networks

Learning the compact structures in *Recurrent Neural Networks* (RNNs) is more challenging than CNNs. As a recurrent unit is shared across all the time steps in sequence, compressing the unit will aggressively affect all the steps. There is a vital challenge originated from recurrent units: as the *basic structures* [3] interweave with each other, independently removing these structures can result in mismatch of their dimensions and then inducing invalid recurrent units. The problem does not exist in CNNs, where neurons (or filters) can be independently removed without violating the usability of the final network structure. One of our key contributions is to identify the structure inside RNNs that shall be considered as a group to most effectively explore sparsity in basic structures. More specific, we propose *Intrinsic Sparse Structures* (ISS) as groups to achieve the goal. By removing weights associated with one component of ISS, the sizes/dimensions (of basic structures) are *simultaneously* reduced by one. SSL can be similarly applied to remove groups of weights in ISS, such that structurally

[3] A basic structure refers to all neural components associated with one hidden state. In LSTMs, basic structures refer to input updates, gates, hidden states, cell states and outputs.

sparse recurrent neural networks can be learned.

We evaluated our method by LSTMs and RHNs in language modeling of Penn Treebank dataset ([MMS93]) and machine Question Answering of SQuAD dataset ([RZLL16]). Our approach works both in fine-tuning and in training from scratch. In a RNN with two stacked LSTM layers with hidden sizes of 1500 (*i.e.*, 1500 components of ISS) for language modeling ([ZSV14]), our method learns that the sizes of 373 and 315 in the first and second LSTMs, respectively, are sufficient for the same perplexity. It achieves 10.59× speedup of inference time. The result is obtained by training from scratch with the same number of epochs. Directly training LSTMs with sizes of 373 and 315 cannot achieve the same perplexity, which proves the advantage of learning ISS for model compression. Encouraging results are also obtained in more compact and state-of-the-art models – the RHN models ([ZSKS17]) and BiDAF model ([SKFH17]).

2.3.1 Intrinsic Sparse Structures

The computation within LSTMs is ([HS97b])

$$\begin{aligned}
\mathbf{i}_t &= \sigma\left(\mathbf{x}_t \cdot \mathbf{W}_{xi} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hi} + \mathbf{b}_i\right) \\
\mathbf{f}_t &= \sigma\left(\mathbf{x}_t \cdot \mathbf{W}_{xf} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hf} + \mathbf{b}_f\right) \\
\mathbf{o}_t &= \sigma\left(\mathbf{x}_t \cdot \mathbf{W}_{xo} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{ho} + \mathbf{b}_o\right) \\
\mathbf{u}_t &= tanh\left(\mathbf{x}_t \cdot \mathbf{W}_{xu} + \mathbf{h}_{t-1} \cdot \mathbf{W}_{hu} + \mathbf{b}_u\right) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{u}_t \\
\mathbf{h}_t &= \mathbf{o}_t \odot tanh\left(\mathbf{c}_t\right)
\end{aligned}, \tag{2.4}$$

where $\odot$ is element-wise multiplication, $\sigma(\cdot)$ is sigmoid function, and $tanh(\cdot)$ is hyperbolic tangent function. Vectors are row vectors. $\mathbf{W}$s are weight matrices, which transform the concatenation (of hidden states $\mathbf{h}_{t-1}$ and inputs $\mathbf{x}_t$) to input updates $\mathbf{u}_t$ and gates ($\mathbf{i}_t$, $\mathbf{f}_t$ and $\mathbf{o}_t$). Fig. 2.9 is the schematic of LSTMs in the layout of

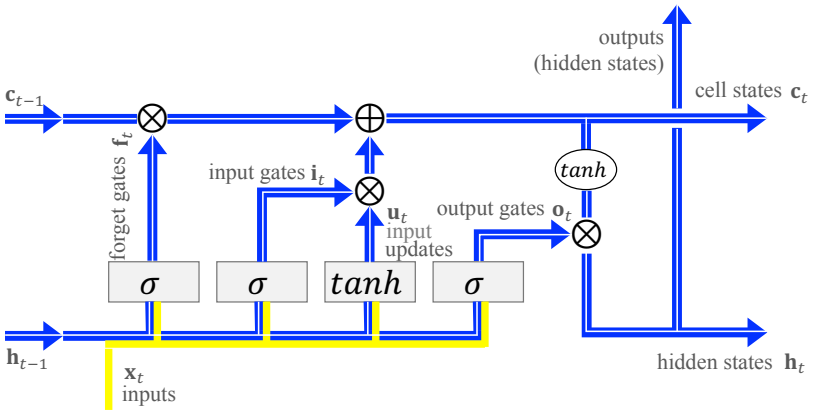


Figure 2.9: Intrinsic Sparse Structures (ISS) in LSTM units.

[Ola15]. The transformations by **W**s and the corresponding nonlinear functions are illustrated in rectangle blocks. Our goal is to reduce the size of this sophisticated structure within LSTMs, meanwhile maintaining the original schematic. Because of element-wise operators ("$\oplus$" and "$\otimes$"), all vectors along the blue band in Fig. 2.9 must have the same dimension. We call this constraint as "*dimension consistency*". The vectors required to obey the dimension consistency include input updates, all gates, hidden states, cell states, and outputs. Note that hidden states are usually outputs connected to classifier layer or stacked LSTM layers. As can be seen in Fig. 2.9, vectors (along the blue band) interweave with each other so removing an individual component from one or a few vectors *independently* can result in the violation of dimension consistency. To overcome this, we propose *Intrinsic Sparse Structures* (ISS) within LSTMs as shown by the blue band in Fig. 2.9. One component of ISS is highlighted as the white strip. By decreasing the size of ISS (*i.e.*, the width of the blue band), we are able to simultaneously reduce the dimensions of basic structures.

To learn sparse ISS, we turn to weight sparsifying. There are totally eight weight matrices in Eq. (2.4). We organize them in the form of Fig. 2.10 as basic LSTM cells in TensorFlow. We can remove one component of ISS by zeroing out all associated weights in the white rows and white columns in Fig. 2.10. Why? Suppose the k-th

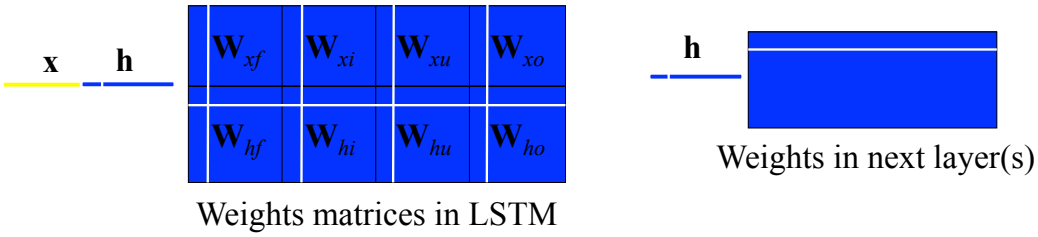


Figure 2.10: Applying Intrinsic Sparse Structures in weight matrices.

hidden state of **h** is removable, then the k-th row in the lower four weight matrices can be all zeros (as shown by the left white horizontal line in Fig. 2.10), because those weights are on connections receiving the k-th *useless* hidden state. Likewise, all connections receiving the k-th hidden state in next layer(s) can be removed as shown by the right white horizontal line. Note that next layer(s) can be an output layer, LSTM layers, fully-connected layers, or a mix of them. ISS overlay two or more layers, without explicit explanation, we refer to the first LSTM layer as the ownership of ISS. When the k-th hidden state turns useless, the k-th output gate and k-th cell state generating this hidden state are removable. As the k-th output gate is generated by the k-th column in $\mathbf{W}_{xo}$ and $\mathbf{W}_{ho}$, these weights can be zeroed out (as shown by the fourth vertical white line in Fig. 2.10). Tracing back against the computation flow in Fig. 2.9, we can reach similar conclusions for forget gates, input gates and input updates, as respectively shown by the first, second and third vertical line in Fig. 2.10. For convenience, we call the weights in white rows and columns as an "*ISS weight group*". Although we propose ISS in LSTMs, variants of ISS for vanilla RNNs, *Gated Recurrent Unit* (GRU) ([CVMG+14]), and *Recurrent Highway Networks* (RHNs) ([ZSKS17]) can also be realized based on the same philosophy.

For even a medium-scale LSTM, the number of weights in one ISS weight group can be very large. It seems to be very aggressive to simultaneously slaughter so many weights to maintain the original recognition performance. However, the proposed ISS intrinsically exists within LSTMs and can even be unveiled by *independently*

sparsifying each weight using ℓ_1-norm regularization. The experimental result is covered in Figure 2.11. It unveils that sparse ISS *intrinsically* exist in LSTMs and the learning process can easily converge to the status with a high ratio of ISS removed. In Section 2.3.2, we propose a learning method to *explicitly* remove much more ISS than the implicit ℓ_1-norm regularization.

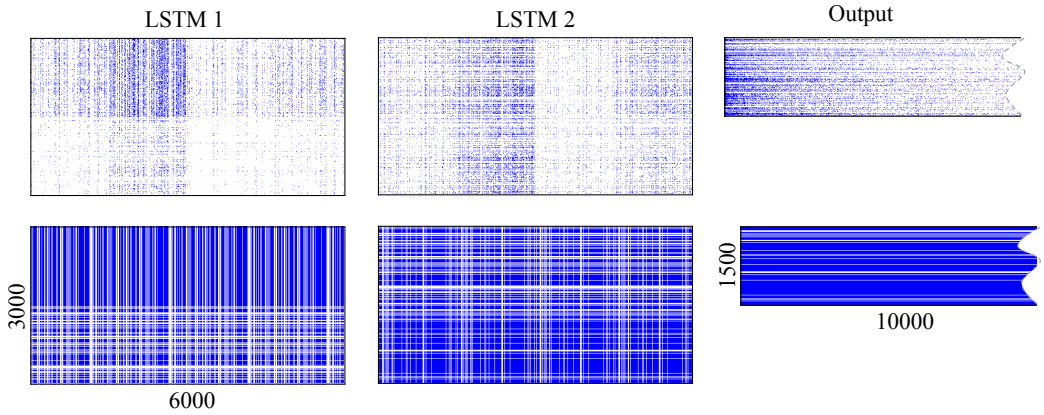


Figure 2.11: Intrinsic Sparse Structures unveiled by ℓ_1 regularization (zoom in for a better view). The top row shows the original weight matrices, where blue dots are nonzero weights and white ones refer zeros; the bottom row are the weight matrices in the format of Fig. 2.10, where white strips are ISS components whose weights are all zeros. For better visualization, the original matrices are evenly down-sampled by 10×10.

In Figure 2.11, we take the large stacked LSTMs by [ZSV14] for language modeling as the example. The network has two stacked LSTM layers whose dimensions of inputs and states are both 1500, and it has an output layer with a vocabulary of 10000 words. The sizes of "ISS weight groups" of two LSTM layers are 24000 and 28000. The perplexities of validation set and test set are respectively 82.57 and 78.57. We fine-tune this baseline LSTMs with ℓ_1-norm regularization. The same training hyper-parameters as the baseline are adopted, except a bigger dropout keep ratio of 0.6 (original 0.35). A weaker dropout is used because ℓ_1-norm is also a regularization to avoid overfitting. A too strong dropout plus ℓ_1-norm regularization can result in underfitting. The weight decay of ℓ_1-norm regularization is 0.0001. The sparsified

network has validation perplexity and test perplexity of 82.40 and 78.60, respectively, which is approximately the same with the baseline. The sparsity of weights in the first LSTM layer, the second LSTM layer and the last output layer is 91.66%, 90.32% and 90.22%, respectively. Figure 2.11 plots the learned sparse weight matrices. The sparse matrices in the top row reveal some interesting patterns: there are lots of all-zero columns and rows, and their positions are highly correlated. Those patterns are profiled in the bottom row. Much to our surprise, sparsifying individual weight *independently* can converge to sparse LSTMs with many ISS removed—504 and 220 ISS components in the first and second LSTM layer are all-zeros. This proves that the ISS structure intrinsically exists in LSTMs.

2.3.2 Learning Method

Suppose $\mathbf{w}_k^{(n)}$ is a vector of all weights in the k-th component of ISS in the n-th LSTM layer ($1 \leq n \leq N$ and $1 \leq k \leq K^{(n)}$), where N is the number of LSTM layers and $K^{(n)}$ is the number of ISS components (*i.e.*, hidden size) of the n-th LSTM layer. The optimization goal is to remove as many "ISS weight groups" $\mathbf{w}_k^{(n)}$ as possible without losing accuracy. Methods to remove weight groups (such as filters, channels and layers) have been successfully studied in CNNs as summarized in Section 2.2. However, how these methods perform in RNNs is unknown. Here, we extend the group Lasso based methods ([YL06]) to RNNs for ISS sparsity learning. More specific, the group Lasso regularization is added to the minimization function in order to encourage sparsity in ISS. Formally, the ISS regularization is

$$R(\mathbf{w}) = \sum_{n=1}^{N} \sum_{k=1}^{K^{(n)}} \left|\left| \mathbf{w}_k^{(n)} \right|\right|_2, \tag{2.5}$$

where $\mathbf{w}$ is the vector of all weights and $||\cdot||_2$ is ℓ_2-norm (*i.e.*, Euclidean length). In *Stochastic Gradient Descent* (SGD) training, the step to update each ISS weight

group becomes

$$\mathbf{w}_k^{(n)} \leftarrow \mathbf{w}_k^{(n)} - \eta \cdot \left(\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_k^{(n)}} + \lambda \cdot \frac{\mathbf{w}_k^{(n)}}{\left|\left|\mathbf{w}_k^{(n)}\right|\right|_2} \right), \tag{2.6}$$

where $E(\mathbf{w})$ is data loss, η is learning rate and $\lambda > 0$ is the coefficient of group Lasso regularization to trade off recognition accuracy and ISS sparsity. The regularization gradient, *i.e.*, the last term in Eq. (2.6), is a unit vector. It constantly squeezes the Euclidean length of each $\mathbf{w}_k^{(n)}$ to zero, such that, a high portion of ISS components can be enforced to fully-zeros after learning. To avoid division by zero in the computation of regularization gradient, we can add a tiny number ϵ in $||\cdot||_2$, that is,

$$\left|\left|\mathbf{w}_k^{(n)}\right|\right|_2 \triangleq \sqrt{\epsilon + \sum_j \left(w_{kj}^{(n)}\right)^2}, \tag{2.7}$$

where $w_{kj}^{(n)}$ is the j-th element of $\mathbf{w}_k^{(n)}$. We set $\epsilon = 1.0e-8$. The learning method can effectively squeeze many groups near zeros, but it is very hard to exactly stabilize them as zeros because of the always-present fluctuating weight updates. Fortunately, the fluctuation is within a tiny ball centered at zero. To stabilize the sparsity during training, we zero out the weights whose absolute values are smaller than a pre-defined threshold τ. The process of thresholding is applied per mini-batch.

2.3.3 Experiments

Our experiments use published models as baselines. The application domains include language modeling of Penn TreeBank and machine Question Answering of SQuAD dataset. For more comprehensive evaluation, we sparsify ISS in LSTM models with both a large hidden size of 1500 and a small hidden size of 100. We also extended ISS approach to state-of-the-art *Recurrent Highway Networks* (RHNs) ([ZSKS17]) to reduce the number of units per layer. We maximize threshold τ to fully exploit the

benefit. For a specific application, we preset τ by cross validation. The maximum τ which sparsifies the dense model (baseline) without deteriorating its performance is selected. The validation of τ is performed only once and no training effort is needed. τ is $1.0e-4$ for the stacked LSTMs in Penn TreeBank, and it is $4.0e-4$ for the RHN and the BiDAF model. We used HyperDrive by [RHY+17] to explore the hyperparameter of λ. More details can be found in our source code.

To measure the inference speed, the experiments were run on a dual socket Intel Xeon CPU E5-2673 v3 @ 2.40GHz processor with a total of 24 cores (12 per socket) and 128GB of memory. Intel MKL library 2017 update 2 was used for matrix-multiplication operations. OpenMP runtime was utilized for parallelism. We used Intel C++ Compiler 17.0 to generate executables that were run on Windows Server 2016. Each of the experiments was run for 1000 iterations, and the execution time was averaged to find the execution latency.

Language Modeling

Stacked LSTMs. A RNN with two stacked LSTM layers for language modeling ([ZSV14]) is selected as the baseline. It has hidden sizes of 1500 (*i.e.*, 1500 components of ISS) in both LSTM units. The output layer has a vocabulary of 10000 words. The dimension of word embedding in the input layer is 1500. Word embedding layer is not sparsified because the computation of selecting a vector from a matrix is very efficient. The same training scheme as the baseline is adopted to learn ISS sparsity, except a larger dropout keep ratio of 0.6 versus 0.35 of the baseline because group Lasso regularization can also avoid over-fitting. All models are trained from scratch for 55 epochs. The results are shown in Table 2.5. Note that, when trained using dropout keep ratio of 0.6 without adopting group Lasso regularization, the baseline over-fits and the lowest validation perplexity is 97.73. The trade-off of perplexity and sparsity

Table 2.5: Learning ISS sparsity from scratch in stacked LSTMs.

Method	Dropout keep ratio	Perplexity (validate, test)	ISS # in (1st , 2nd) LSTM	Weight #	Total time*	Speedup	Mult-add reduction†
baseline	0.35	(82.57, 78.57)	(1500, 1500)	66.0M	157.0ms	1.00×	1.00×
ISS	0.60	(82.59, 78.65)	(373, 315)	21.8M	14.82ms	10.59×	7.48×
		(80.24, 76.03)	(381, 535)	25.2M	22.11ms	7.10×	5.01×
direct design	0.55	(90.31, 85.66)	(373, 315)	21.8M	14.82ms	10.59×	7.48×

* Measured with 10 batch size and 30 unrolled steps.
† The reduction of multiplication-add operations in matrix multiplication. Defined as (original Mult-add)/(left Mult-add)

is controlled by λ. In the second row, with tiny perplexity difference from baseline, our approach can reduce the number of ISS in the first and second LSTM unit from 1500, down to 373 and 315, respectively. It reduces the model size from 66.0M to 21.8M and achieves 10.59× speedup. Remarkably, the practical speedup (10.59×) even goes beyond theoretical mult-add reduction (7.48×) as shown in Table 2.5 —which comes from the increased computational efficiency. When applying structured sparsity, the underlying weight matrices become smaller so as to fit into the L3 cache with good locality, which improves the FLOPS (floating point operations per second). This is a key advantage of our approach over non-structurally sparse RNNs generated by connection pruning ([NDSE17]), which suffers from irregular memory access pattern and inferior-theoretical speedup. At last, when learning a compact structure, our method can perform as structure regularization to avoid overfitting. As shown in the third row in Table 2.5, lower perplexity is achieved by even a smaller (25.2M) and faster (7.10×) model. Its learned weight matrices are visualized in Fig. 2.12, where 1119 and 965 ISS components shown by white strips are removed in the first and second LSTM, respectively.

A straightforward way to reduce model complexity is to directly design a RNN with a smaller hidden size and train from scratch. Compare with direct design approach, our ISS method can automatically learn optimal structures within LSTMs. More importantly, compact models learned by ISS method have lower perplexity, comparing

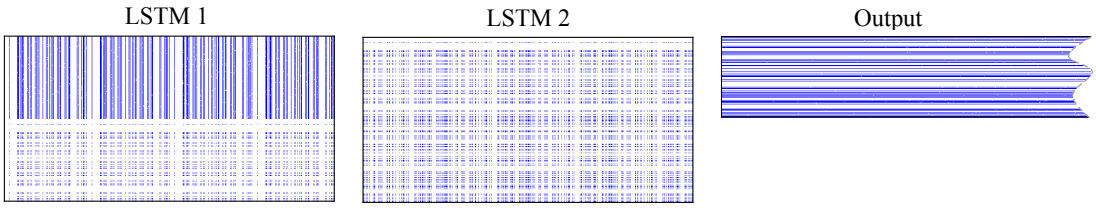


Figure 2.12: Intrinsic Sparse Structures learned by group Lasso regularization (zoom in for better view). Original weight matrices are plotted, where blue dots are nonzero weights and white ones refer zeros. For better visualization, original matrices are evenly down-sampled by 10×10.

with direct design method. To evaluate it, we directly design a RNN with exactly the same structure of the second RNN in Table 2.5 and train it from scratch instead of learning ISS from a larger RNN. The result is included in the last row of Table 2.5. We tuned dropout keep ratio to get best perplexity for the directly-designed RNN. The final test perplexity is 85.66, which is 7.01 higher that our ISS method.

Recurrent Highway Networks (RHN). *RHN* ([ZSKS17]) is a class of state-of-the-art recurrent models, which enable "step-to-step transition depths larger than one". In a RHN, we define the number of units per layer as *RHN width*. Specifically, we select the "Variational RHN + WT" model in Table 1 of [ZSKS17] as the baseline. It has depth 10 and width 830, with totally 23.5M parameters. In a nutshell, our approach can reduce the RHN width from 830 to 517 without losing perplexity.

Following the same idea of identifying the "ISS weight groups" to reduce the size of basic structures in LSTMs, we can identify the groups in RHNs to reduce the RHN width. In brief, one group include corresponding columns/rows in weight matrices of the H nonlinear transform, of the T and C gates, and of the embedding and output layers. The group size is 46520. The groups are indicated by JSON files in our source code[4]. By learning ISS in RHNs, we can simultaneously reduce the dimension of word embedding and the number of units per layer.

Table 2.6 summarizes results. All experiments are trained from scratch with the

[4]groups_hidden830.json

Table 2.6: Learning ISS sparsity from scratch in RHNs.

Method	λ	Perplexity (validate, test)	RHN width	Parameter #
baseline	0.0	(67.9, 65.4)	830	23.5M
ISS*	0.004	(67.5, 65.0)	726	18.9M
ISS*	0.005	(**68.1**, **65.4**)	**517**	**11.1**M
ISS*	0.006	(70.3, 67.7)	403	7.6M
ISS*	0.007	(74.5, 71.2)	328	5.7M

* All dropout ratios are multiplied by 0.6×.

same hyper-parameters in the baseline, except that smaller dropout ratios are used in ISS learning. Larger λ, smaller RHN width but higher perplexity. More importantly, without losing perplexity, our approach can learn a smaller model with RHN width 517 from an initial model with RHN width 830. This reduces the model size to 11.1M, which is 52.8% reduction. Moreover, ISS learning can find a smaller RHN model with width 726, meanwhile improve the state-of-the-art perplexity as shown by the second entry in Table 2.6.

Machine Reading Comprehension

We evaluate ISS method by state-of-the-art dataset (SQuAD) and model (BiDAF). SQuAD ([RZLL16]) is a recently released reading comprehension dataset, crowdsourced from $100,000+$ question-answer pairs on 500+ Wikipedia articles. ExactMatch (EM) and F1 scores are two major metrics for the task[5]. The higher those scores are, the better the model is. We adopt BiDAF ([SKFH17]) to evaluate how ISS method works in small LSTM units. BiDAF is a compact machine Question Answering model with totally 2.69M weights. The ISS sizes are only 100 in all LSTM units. The implementation of BiDAF is made available by its authors [6].

[5]Refer to [RZLL16] for the definition of EM and F1 scores.

[6]`https://github.com/allenai/bi-att-flow/tree/dev`

Table 2.7: The ISS in BiDAF.

LSTM name	Dimensions	Receivers of hidden states	Size of "ISS weight group"
`ModFwd1`	900×400	`ModFwd2` `ModBwd2`	4800
`ModBwd1`	900×400	`ModFwd2` `ModBwd2`	4800

`ModFwd2`	300×400	`OutFwd`
		`OutBwd`
		logit layer for start index
			3201

`ModBwd2`	300×400	`OutFwd`
		`OutBwd`
		logit layer for start index
			3201

`OutFwd`	1500×400	logit layer for end index	6401
`OutBwd`	1500×400	logit layer for end index	6401

BiDAF has character, word and contextual embedding layers to extract representations from input sentences, following which are bi-directional attention layer, modeling layer, and final output layer. LSTM units are used in contextual embedding layer, modeling layer, and output layer. All LSTMs are bidirectional ([SP97]). In a bidirectional LSTM, there are one forward plus one backward LSTM branch. The two branches share inputs and their outputs are concatenated for next stacked layers. We found that it is hard to remove ISS components in contextual embedding layer, because the representations are relatively dense as it is close to inputs and the original hidden size (100) is relatively small. In our experiments, we exclude LSTMs in contextual embedding layer and sparsify all other LSTM layers. Those LSTM layers are the computation bottleneck of BiDAF. We profiled the computation time on CPUs, and find those LSTM layers (excluding contextual embedding layer) consume 76.47% of total inference time. There are three bi-directional LSTM layers we will sparsify, two of which belong to the modeling layer, and one belongs to the output layer. More details of BiDAF are covered by [SKFH17]. For brevity, we mark the forward (backward) path of the 1st bi-directional LSTM in the modeling layer as

Table 2.8: Remaining ISS components in BiDAF by fine-tuning.

EM	F1	ModFwd1	ModBwd1	ModFwd2	ModBwd2	OutFwd	OutBwd	weight #	Total time*
67.98	77.85	100	100	100	100	100	100	2.69M	6.20ms
67.21	76.71	100	95	78	82	71	52	2.08M	5.79ms
66.59	76.40	84	90	38	46	34	21	1.48M	4.52ms
65.29	75.47	54	47	22	30	18	12	1.03M	3.54ms
64.81	75.22	52	50	19	26	15	12	1.01M	3.51ms

* Measured with batch size 1.

`ModFwd1` (`ModBwd1`). Similarly, `ModFwd2` and `ModBwd2` are for the 2nd bi-directional LSTM. Forward (backward) LSTM path in the output layer are marked as `OutFwd` and `OutBwd`.

As discussed in Section 2.3.1, multiple parallel layers can receive the hidden states from the same LSTM layer and all connections (weights) receive those hidden states belong to the same ISS. For instance, `ModFwd2` and `ModBwd2` both receive hidden states of `ModFwd1` as inputs, therefore the k-th "ISS weight group" includes the k-th rows of weights in both `ModFwd2` and `ModBwd2`, plus the weights in the k-th ISS component within `ModFwd1`. For simplicity, we use "ISS of `ModFwd1`" to refer to the whole group of weights. Structures of six ISS are included in Table 2.7. We learn ISS sparsity in BiDAF by both fine-tuning the baseline and training from scratch. All the training schemes keep as the same as the baseline except applying a higher dropout keep ratio. After training, we zero out weights whose absolute values are smaller than 0.02. This does not impact EM and F1 scores, but increase sparsity.

Table 2.8 shows the EM, F1, the number of remaining ISS components, model size, and inference speed. The first row is the baseline BiDAF. Other rows are obtained by fine-tuning baseline using ISS regularization. In the second row by learning ISS, with small EM and F1 loss, we can reduce ISS in all LSTMs except `ModFwd1`. For example, almost half of the ISS components are removed in `OutBwd`. By increasing the strength of group Lasso regularization (λ), we can increase the ISS sparsity by losing some EM/F1 scores. The trade-off is listed in Table 2.8. With 2.63 F1 score

Table 2.9: Remaining ISS components in BiDAF by training from scratch.

EM	F1	ModFwd1	ModBwd1	ModFwd2	ModBwd2	OutFwd	OutBwd	weight #	Total time*
67.98	77.85	100	100	100	100	100	100	2.69M	6.20ms
67.36	77.16	87	81	87	92	74	96	2.29M	5.83ms
66.32	76.22	51	33	42	58	37	26	1.17M	4.46ms
65.36	75.78	20	33	40	38	31	16	0.95M	3.59ms
64.60	74.99	23	22	35	35	25	14	0.88M	2.74ms

* Measured with batch size 1.

loss, the sizes of `OutFwd` and `OutBwd` can be reduced from original 100 to 15 and 12, respectively. At last, we find it hard to reduce ISS sizes without losing any EM/F1 score. This implies that BiDAF is compact enough and its scale is suitable for both computation and accuracy. However, our method can still significantly compress this compact model under acceptable performance loss.

At last, instead of fine-tuning baseline, we train BiDAF from scratch with ISS learning. The results are summarized in Table 2.9. Our approach also works well when training from scratch. Overall, training from scratch balances the sparsity across all layers better than fine-tuning, which results in even better compression of model size and speedup of inference time. The histogram of vector lengths of "ISS weight groups" is plotted in Figure 2.13.

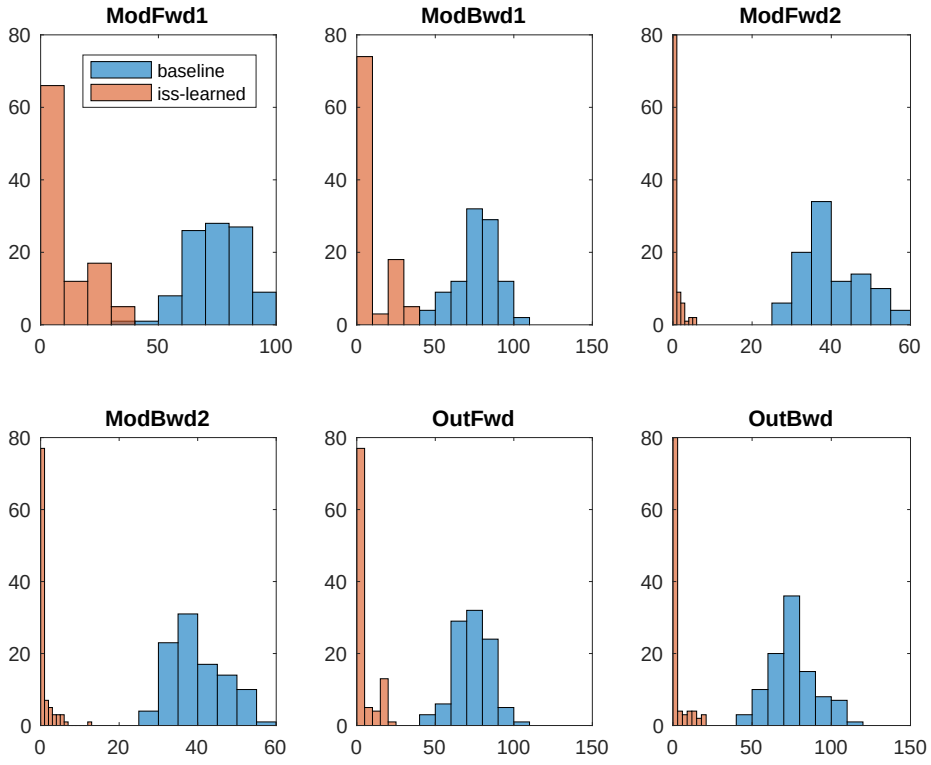


Figure 2.13: Histogram of vector lengths of "ISS weight groups" in BiDAF. The ISS-learned BiDAF is the one in the third row of Table 2.9 with EM 66.32 and F1 76.22. Using our approach, the lengths are regularized closer to zeros with a peak at the zero, resulting in high ISS sparsity.

Chapter 3

Lower Rank Deep Neural Networks

In this chapter, an alternative efficient deep learning approach will be proposed: lower-rank deep neural networks. Low-rank approximation (LRA) method decomposes a large model to a compact one with more lightweight layers by weight/tensor factorization. Denil *et al.* [DSD+13] studied different dictionaries to remove the redundancy between filters and channels in DNNs. Jaderberg *et al.* [JVZ14] explored filter and data reconstruction optimizations to attain optimal separable basis. Denton *et al.* [DZB+14] clustered filters, extended LRA (*e.g.*, *Singular Value Decomposition*, SVD) to larger-scale DNNs, and achieved 2× speedup for the first two layers with 1% accuracy loss. Many new decomposition methods were proposed [IRS+15][TXWE15][LGR+14][ZZHS16] and the effectiveness of LRA in state-of-the-art DNNs were evaluated [SZ14][SLJ+15]. Similar evaluations on mobile devices were also reported [KPY+15][WC16]. To help previous LRA to achieve more compact DNNs, we propose *Force Regularization* to coordinate DNN filters to more correlated states, in which *lower*-rank are achievable for faster computation. Lower-rank deep neural networks is orthogonal to sparsity-based methods. In the experiment part, I will show that DNNs accelerated by our method can be further sparsified by both non-structured and structured sparsity methods, potentially achieving faster computation.

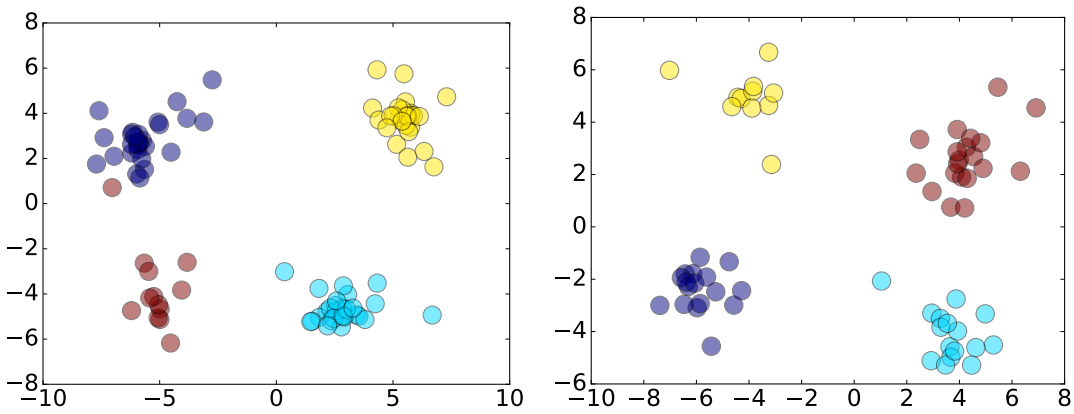


Figure 3.1: *Linear Discriminant Analysis* (LDA) of filters in the first convolutional layer of *AlexNet* (left) and *GoogLeNet* (right)

3.1 Correlated Filters and Their Low Rank Approximation

The prior knowledge is that correlation exists among trained filters in DNNs and those filters lie in a low-rank space. For example, the color-agnostic filters [KSH12] learned in the first layer of *AlexNet* lie in a hyper-plane, where RGB channels at each pixel have the same value. Fig. 3.1 presents the results of *Linear Discriminant Analysis* (LDA) of the first convolutional filters in *AlexNet* and *GoogLeNet*. The filters are normalized to unit vectors and colored to four clusters by k-means clustering, and then projected to 2D space by LDA to maximize cluster separation. The figure indicates high correlation among filters within a cluster. A naïve approach of filter approximation is to use the centroid of a cluster to approximate filters within that cluster, thus, the number of clusters is the rank of the space. Essentially, k-means clustering is a LRA [Bau15] method, although we will later show that other LRA methods can give better approximation. The motivation of this work is that if we are able to nudge filters during the training such that the filters within a cluster are coordinated closer and some adjacent clusters are even merged into one cluster, then

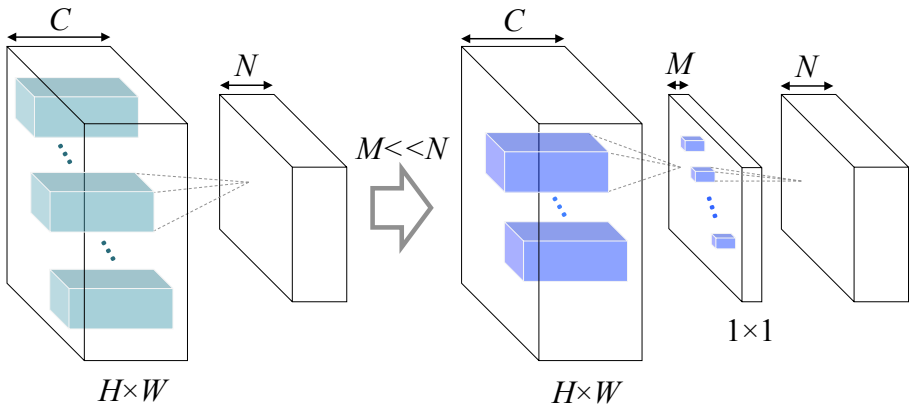


Figure 3.2: Cross-filter LRA of a convolutional layer

more accurate filter approximation using lower rank can be achieved. We propose *Force Regularization* to realize it.

Before introducing *Force Regularization*, we first mathematically formulate LRA of DNN filters. Theoretically, almost all LRA methods can gain lower-rank approximation upon our method because filters are coordinated to more correlated state. Instead of onerously replicating all of these LRA methods, we choose cross-filter approximation [DSD+13][ZZHS16] and a state-of-the-art work in [TXWE15] as our baselines.

Fig. 3.2 illustrates the cross-filter approximation of a convolutional layer. We assume all weights in a convolutional layer is a tensor $\mathcal{W} \in \mathbb{R}^{N \times C \times H \times W}$, where N and C are the numbers of filters and input channels, and H and W are the spatial height and width of the filters, respectively. With input feature map $\mathcal{I}$, the n-th output feature map $\mathcal{O}_n = \mathcal{W}_n * \mathcal{I}$, where $\mathcal{W}_n \in \mathbb{R}^{1 \times C \times H \times W}$ is the n-th filter. Because of the redundancy (or correlation) across the filters [DSD+13], tensor $\mathcal{W}_n (\forall n \in [1...N])$ can be approximated by a linear combination of the basis $\mathcal{B}_m \in \mathbb{R}^{1 \times C \times H \times W} (m \in [1...M], M \ll N)$ of a low-rank space $\mathcal{B} \in \mathbb{R}^{M \times C \times H \times W}$, such

as

$$\mathcal{O}_n \approx \left(\sum_{m=1}^{M} b_m^{(n)} \mathcal{B}_m \right) * \mathcal{I} = \sum_{m=1}^{M} \left(b_m^{(n)} \mathcal{F}_m \right). \tag{3.1}$$

Where $b_m^{(n)}$ is a scalar, and $\mathcal{F}_m = \mathcal{B}_m * \mathcal{I}$ is the feature map generated by basis filter $\mathcal{B}_m$. Therefore, the output feature map $\mathcal{O}_n$ is a linear combination of $\mathcal{F}_m (m \in [1...M])$ which can be interpreted as the feature map basis. Since the linear combination essentially is a 1×1 convolution, the convolutional layer can be decomposed to two sequential lightweight convolutional layers as shown in Fig. 3.2. The original computation complexity is $\mathcal{O}(NCHWH^{'}W^{'})$, where $H^{'}$ and $W^{'}$ is the height and width of output feature maps, respectively. After applying cross-filter LRA, the computation complexity is reduced to $\mathcal{O}(MCHWH^{'}W^{'} + NMH^{'}W^{'})$. The computation complexity decreases when the rank $M < \frac{NCHW}{CHW+N}$.

3.2 Lower-rank Deep Neural Networks by Force Regularization

3.2.1 Regularization by Attractive Forces

This section proposes *Force Regularization* from the perspective of physics. It is a gradient-based approach that adds extra gradients to data loss gradients. The data loss gradients aim to minimize classification error as traditional DNNs do. The extra gradients introduced by *Force Regularization* gently adjust the lengths and directions of data loss gradients so as to nudge filters to a more correlated state. With a good setup of hyper-parameter, our method can coordinate more useful information of filters to a lower-rank space meanwhile maintain accuracy. Inspired by Newton's Laws, we propose an intuitive, computation-efficient and effective *Force Regularization* that uses attractive forces to coordinate filters.

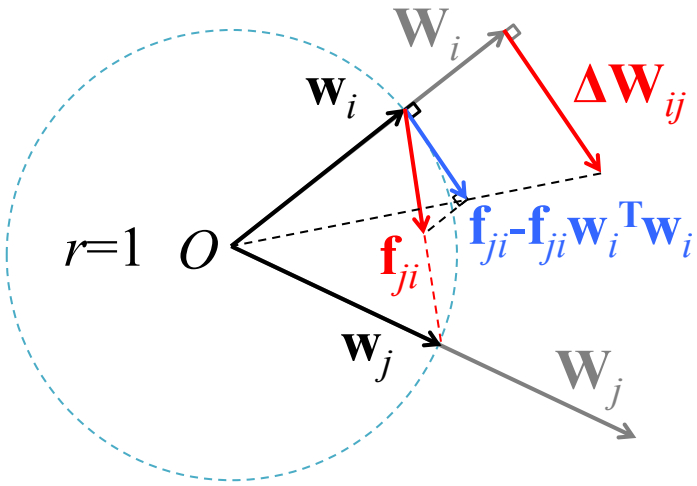


Figure 3.3: *Force Regularization* to coordinate filters

Force Regularization: As illustrated in Fig. 3.3, suppose the filter $\mathcal{W}_n \in \mathcal{W}$ is reshaped as a vector $\mathbf{W}_n \in \mathbb{R}^{1\times CHW}$ and normalized as $\mathbf{w}_n \in \mathbb{R}^{1\times CHW}$ $(\forall n \in [1...N])$, with their origin at O. We introduce the pair-wise *attractive force* $\mathbf{f}_{ji} = f(\mathbf{w}_j - \mathbf{w}_i)$ $(\forall i, j \in [1...N])$ on $\mathbf{w}_i$ generated by $\mathbf{w}_j$. The gradient of *Force Regularization* to update filter $\mathbf{W}_i$ is defined as

$$\Delta\mathbf{W}_i = \sum_{j=1}^{N} \Delta\mathbf{W}_{ij} = ||\mathbf{W}_i|| \sum_{j=1}^{N} \left(\mathbf{f}_{ji} - \mathbf{f}_{ji}\mathbf{w}_i^T\mathbf{w}_i\right), \tag{3.2}$$

where $||\cdot||$ is the Euclidean norm. The regularization gradient in Eq. (3.2) is perpendicular to filter vector and can be efficiently computed by addition and multiplication. The final updating of weights by gradient descent is

$$\mathbf{W}_i \leftarrow \mathbf{W}_i - \eta \cdot \left(\frac{\partial E(\mathcal{W})}{\partial \mathbf{W}_i} - \lambda_s \cdot \Delta\mathbf{W}_i\right), \tag{3.3}$$

where $E(\mathcal{W})$ is data loss, η is learning rate and $\lambda_s > 0$ is the coefficient of *Force Regularization* to trade off the rank and accuracy. We select λ_s by cross-validation in this work. The gradient of common weight-wise regularization (*e.g.*, ℓ_2-norm) is omitted in Eq. (3.3) for simplicity.

Fig. 3.3 intuitively explains our method. Suppose each vector $\mathbf{w}_i$ is a rigid stick and there is a particle fixed at the endpoint. The particle has unit mass, and the stick is massless and can freely spin around the origin. Given the pair-wise attractive forces (*e.g.,* universal gravitation) $\mathbf{f}_{ji}$, Eq. (3.2) is the acceleration of particle i. As the forces are attractive, neighbor particles tend to spin around the origin to assemble together. Although our regularizer seems to collapse all particles to one point which is the rank-one space for most lightweight DNNs, there exist gradients of data loss to avoid this. More specific, pre-trained filters orient to discriminative directions $\mathbf{w}_n$ ($n \in [1...N]$). In each direction $\mathbf{w}_n$, there are some correlated filters as observed in Fig. 3.1. During the subsequent retraining with our regularizer, regularization gradients coordinate a cluster of filters closer to a typical direction $\mathbf{d}_m$ ($m \in [1...M], M \ll N$), but data loss gradients avoid collapsing $\mathbf{d}_m$ together so as to maintain the filters' capability of extracting discriminative features. If all filters could be extremely collapsed toward one point meanwhile maintain classification accuracy, it implies the filters are over-redundant and we can attain a very efficient DNN by decomposing it to a rank-one space.

We derive the *Force Regularization* gradient from the *normalized* filters based on the following facts: (1) A normalized filter is on the unit hypersphere, and its orientation is the only free parameter we need to optimize; (2) The gradient of $\mathbf{W}_i$ can be easily scaled by the vector length $||\mathbf{W}_i||$ without changing the angular velocity.

In Eq. (3.2), $\mathbf{f}_{ji} = f(\mathbf{w}_j - \mathbf{w}_i)$ is the force function related to distance. We study *ℓ_2-norm Force*

$$f_{\ell_2}(\mathbf{w}_j - \mathbf{w}_i) = \mathbf{w}_j - \mathbf{w}_i \tag{3.4}$$

and *ℓ_1-norm Force*

$$f_{\ell_1}(\mathbf{w}_j - \mathbf{w}_i) = \frac{\mathbf{w}_j - \mathbf{w}_i}{||\mathbf{w}_j - \mathbf{w}_i||} \tag{3.5}$$

in this work. We define the force of Eq. (3.4) as *ℓ_2-norm Force* because the strength

Table 3.1: Ranks *vs.* scalers of step sizes of regularization gradients.

Scaler	Error	conv1*	conv2	conv3
0 (baseline)	18.0%	17/32	27/32	55/64
$\|\|\mathbf{W}_i\|\|$	17.9%	15/32	22/32	30/64
$1/\|\|\mathbf{W}_i\|\|$	18.0%	16/32	27/32	32/64

* The first convolutional layer.

Table 3.2: The *rank M* in each convolutional layer after *Force Regularization*.

Net	Force	Top-1 error	conv1	conv2	conv3	conv4	conv5	Average rank ratio ‡
ConvNet	None (baseline)†	18.0%	17/32‡	27/32	55/64	–	–	74.48%
ConvNet	ℓ_2-norm	17.9%	15/32	22/32	30/64	–	–	54.17%
ConvNet	ℓ_1-norm	18.0%	17/32	25/32	20/64	–	–	54.17%
AlexNet	None (baseline)	42.63%	47/96	164/256	306/384	318/384	220/256	72.29%
AlexNet	ℓ_2-norm	42.70%	49/96	143/256	128/384	122/384	161/256	46.98%
AlexNet	ℓ_1-norm	42.45%	49/96	155/256	157/384	108/384	178/256	50.03%

†The baseline without *Force Regularization*. ‡M/N: Low *rank M* over full rank N, which is defined as rank ratio.

linearly decreases with the distance $||\mathbf{w}_j - \mathbf{w}_i||$, just as the gradient of regularization ℓ_2-norm does. We name the force of Eq. (3.5) as ℓ_1*-norm Force* because the gradient is a constant unit vector regardless of the distance, just as the gradient of sparsity regularization ℓ_1-norm is.

3.2.2 Mathematical Implications

This section explains the mathematical implications behind: *Force Regularization* is related to but *different* from minimizing the sum of pair-wise distances between normalized filters.

Theorem 1. *Suppose filter* $\mathcal{W}_n \in \mathcal{W}$ *is reshaped as a vector* $\mathbf{W}_n \in \mathbb{R}^{1\times CHW}$ *and normalized as* $\mathbf{w}_n \in \mathbb{R}^{1\times CHW} (\forall n \in [1...N])$. *For each filter, Force Regularization under* ℓ_2*-norm force has the same gradient direction of regularization* $R(\mathcal{W})$*, but differs by adapting the step size to the filer's length, where*

$$R(\mathcal{W}) = \frac{1}{2}\sum_{j=1}^{N}\sum_{i=1}^{N}\left|\left|\frac{\mathbf{W}_j}{||\mathbf{W}_j||} - \frac{\mathbf{W}_i}{||\mathbf{W}_i||}\right|\right|^2. \tag{3.6}$$

Proof: Because $\mathbf{w}_j = \frac{\mathbf{W}_j}{||\mathbf{W}_j||}$,

$$\begin{aligned}\frac{\partial R(\mathcal{W})}{\partial \mathbf{W}_i} &= \frac{1}{2}\sum_{j=1}^{N}\frac{\partial\left(\mathbf{w}_j-\mathbf{w}_i\right)\left(\mathbf{w}_j-\mathbf{w}_i\right)^T}{\partial \mathbf{W}_i} \\ &= \frac{1}{2}\sum_{j=1}^{N}\frac{\partial\left(1-2\mathbf{w}_j\mathbf{w}_i^T+1\right)}{\partial \mathbf{W}_i} \\ &= -\sum_{j=1}^{N}\frac{\partial\left(\mathbf{w}_j\mathbf{w}_i^T\right)}{\partial \mathbf{W}_i} = -\sum_{j=1}^{N}\mathbf{w}_j\frac{\partial \mathbf{w}_i^T}{\partial \mathbf{W}_i},\end{aligned} \tag{3.7}$$

where $\frac{\partial \mathbf{w}_i^T}{\partial \mathbf{W}_i} := \mathbf{G}_i$ is a derivative matrix with element

$$\begin{aligned}G_i^{(pq)} &= \frac{\partial w_i^{(p)}}{\partial W_i^{(q)}} = \frac{\partial \frac{W_i^{(p)}}{||\mathbf{W}_i||}}{\partial W_i^{(q)}} \\ &= \frac{1}{||\mathbf{W}_i||}\left(\delta(p,q)-\frac{W_i^{(p)}W_i^{(q)}}{||\mathbf{W}_i||^2}\right).\end{aligned} \tag{3.8}$$

Superscripts $p, q \in [1 \dots CHW]$ index the elements in vectors $\mathbf{w}_i$ and $\mathbf{W}_i$. $\delta(p,q)$ is the *unit impulse* function:

$$\delta(p,q) = \begin{cases}1 & p = q \\ 0 & p \neq q\end{cases}. \tag{3.9}$$

Therefore,

$$\mathbf{G}_i = \frac{1}{||\mathbf{W}_i||}\left(\mathbf{I}-\mathbf{w}_i^T\mathbf{w}_i\right). \tag{3.10}$$

Replacing Eq. (3.10) to Eq. (3.7), we have

$$\begin{aligned}-\frac{\partial R(\mathcal{W})}{\partial \mathbf{W}_i} &= \frac{1}{||\mathbf{W}_i||}\sum_{j=1}^{N}\left(\left(\mathbf{w}_j-\mathbf{w}_i\right)-\left(\mathbf{w}_j-\mathbf{w}_i\right)\mathbf{w}_i^T\mathbf{w}_i\right) \\ &= \frac{1}{||\mathbf{W}_i||}\left(\left(\sum_{j=1}^{N}\mathbf{f}_{ji}\right)-\left(\sum_{j=1}^{N}\mathbf{f}_{ji}\right)\mathbf{w}_i^T\mathbf{w}_i\right),\end{aligned} \tag{3.11}$$

where $\mathbf{f}_{ji} = f_{\ell_2}(\mathbf{w}_j - \mathbf{w}_i) = \mathbf{w}_j - \mathbf{w}_i$. Therefore, Eq. (3.11) and Eq. (3.2) have the same direction.

Theorem 1 states that our proposed *Force Regularization* in Eq. (3.2) is related to Eq. (3.11). However, the step size of the gradient in Eq. (3.2) is scaled by the length $||\mathbf{W}_i||$ of the filter instead of its reciprocal in Eq. (3.11). This ensures that the filter spins the same angle regardless of its length and avoids the issue of being divided by zero. Table 3.1 summarizes the ranks *vs.* step sizes for the *ConvNet* [KSH12], which is trained by CIFAR-10 database without data augmentation. The original *ConvNet* has 32, 32, and 64 filters in each convolutional layer, respectively. The rank is the smallest number of basis filters (in Fig. 3.2) obtained by PCA with $\leq 5\%$ reconstruction error. Therefore, $||\mathbf{W}_i||$ works better than its reciprocal when coordinating filters to a lower-rank space.

Following the same proof procedure, we can easily find that *Force Regularization* under ℓ_1-*norm Force* has the same conclusion when

$$R(\mathcal{W}) = \sum_{j=1}^{N} \sum_{i=1}^{N} \left|\left| \frac{\mathbf{W}_j}{||\mathbf{W}_j||} - \frac{\mathbf{W}_i}{||\mathbf{W}_i||} \right|\right|. \tag{3.12}$$

3.3 Experiments

3.3.1 Implementation

Our experiments are performed in Caffe [JSD+14] using CIFAR-10 [KH09] and ILSVRC-2012 ImageNet [DDS+09]. Published models are adopted as the baselines: In CIFAR-10, we choose *ConvNet* without data augmentation [KSH12] and *ResNets-20* with data augmentation [HZRS16]. We adopt the same shortcut connections in [WWW+16a] for *ResNets-20*. For ImageNet, we use *AlexNet* and *GoogLeNet* models trained by Caffe, and report accuracy using only center crop of images.

Our experiments of *Force Regularization* show that, with the same maximum iterations, the training from the baseline can achieve a better tradeoff between accuracy and speedup comparing with the training from scratch, because the baseline offers a good initial point for both accuracy and filter correlation. During the training with *Force Regularization* on CIFAR-10, we use the same base learning rate as the baseline; while in ImageNet, $0.1\times$ base learning rate of the baseline is adopted.

3.3.2 Rank Analysis of Coordinated DNNs

In light of various low-rank approximation methods, without losing the generalization, we first adopt *Principal Component Analysis* (PCA) [ZZHS16][LWF+15] to evaluate the effectiveness of *Force Regularization*. Specifically, the filter tensor $\mathcal{W}$ can be reshaped to a matrix $\mathbf{W} \in \mathbb{R}^{N \times CHW}$, the rows of which are the reshaped filters $\mathbf{W}_n$ ($\forall n \in [1...N]$). PCA minimizes the *least square reconstruction error* when projecting a column ($\mathbb{R}^N$) of $\mathbf{W}$ to a low-rank space $\mathbb{R}^M (M \ll N)$. The reconstruction error is $e_M = \sum_{i=M+1}^{N} \lambda_i$, where λ_i is the i-th largest eigenvalue of covariance matrix $\frac{\mathbf{W}\mathbf{W}^T}{CHW-1}$. Under the constraint of *error percentage* $\frac{e_M}{e_0}$ (*e.g.*, $\frac{e_M}{e_0} \leq 5\%$), *lower*-rank approximation can be obtained if the minimal rank M can be *smaller*. In this section, without explicit explanation, we define *rank* M of a convolutional layer as the minimal M which has $\leq 5\%$ reconstruction error by PCA.

Table 3.2 summarizes the *rank* M in each layer of *ConvNet* and *AlexNet* without accuracy loss after *Force Regularization*. In the baselines, the learned filters in the front layers are intrinsically in a very low-rank space but the *rank* M in deeper layers is high. This could explain why only speedups of the first two convolutional layers were reported in [DZB+14]. Fortunately, by using either ℓ_2-norm or ℓ_1-norm force, our method can efficiently maintain the low *rank* M in the first two layers (*e.g.*, conv1-conv2 in *AlexNet*), meanwhile significantly reduce the *rank* M of deeper layers

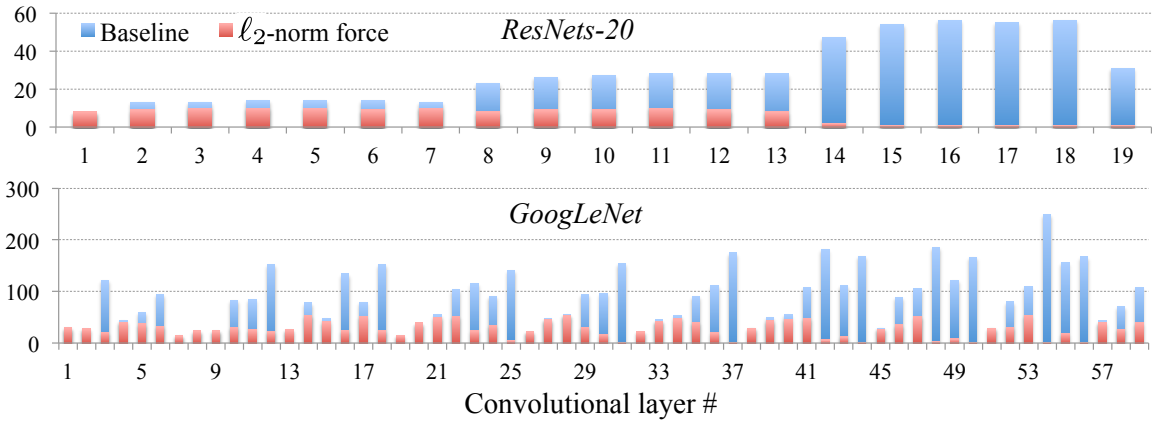


Figure 3.4: The *rank M* in each convolutional layer of *ResNets-20* and *GoogLeNet*. Red bar overlaps blue bar. The accuracy loss is 0.75% for *ResNets-20* and 2.46% (top-5) for *GoogLeNet*.

(*e.g.*, conv3-conv5 in *AlexNet*). On average, our method can reduce the layer-wise rank ratio by $\sim 50\%$. The effectiveness of our method on deep layers is very important as the depth of modern DNNs grows dramatically [SLJ+15][HZRS16]. Fig. 3.4 shows the *rank M* of *ResNets-20* [HZRS16] and *GoogLeNet* [SLJ+15] after *Force Regularization*, representing the scalability of our method on deeper DNNs. With an acceptable accuracy loss, 5 layers in *ResNets-20* and 6 layers in *GoogLeNet* are even coordinated to *rank* $M = 1$, which indicates those Inception blocks in *GoogLeNet* or Residual blocks in *ResNets* have been over-parameterized and can be greatly simplified.

To study the trade-off between rank, accuracy, and the pros and cons of ℓ_2-norm and ℓ_1-norm force, we conducted comprehensive experiments on *AlexNet*. As shown in Fig. 3.5, with mere 1.71% (1.80%) accuracy loss, the average rank ratio can be reduced to 28.59% (28.72%) using ℓ_2-norm (ℓ_1-norm) force. Very impressively, the *rank M* of each group in conv4 can be reduced to one by ℓ_1-norm force. The results also show that ℓ_2-norm force is more effective than ℓ_1-norm force when the rank ratio is high (*e.g.*, conv2 and conv5), while ℓ_1-norm force works better for layers whose potential rank ratios are low (*e.g.*, conv3 and conv4). In general, ℓ_2-norm force can better balance the ranks across all the layers.

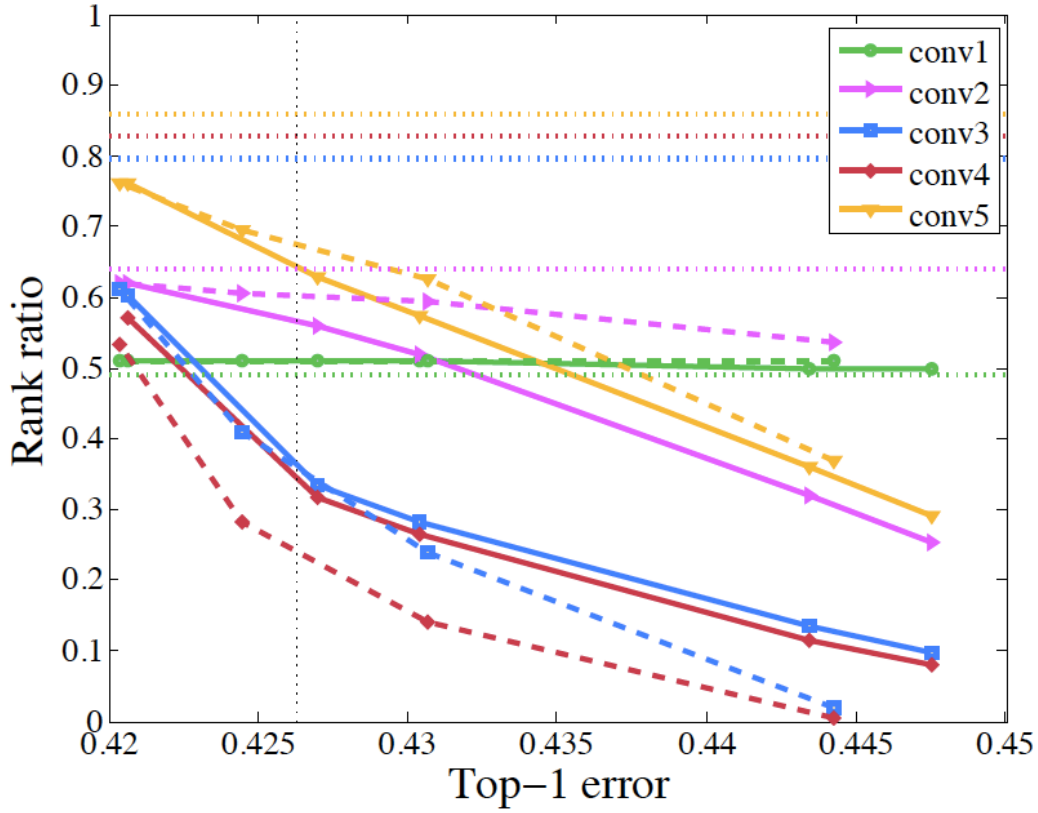


Figure 3.5: Th rank ratio (having $\leq 5\%$ PCA reconstruction error) in each layer *vs.* top-1 error for *AlexNet*. Horizontal dotted lines represent the rank ratios of the baseline, and vertical dotted line is the error of baseline. Solid (dashed) curves depict rank ratios of the *AlexNet* after *Force Regularization* by ℓ_2-norm (ℓ_1-norm) force. Each layer is denoted by a typical color. The sensitivity of hyper-parameter λ_s: along the direction from left to right, λ_s of ℓ_2-norm force changes from 1.2e-5, to 1.8e-5, 2.0e-5, 3.0e-5, and 3.5e-5; and for ℓ_1-norm force, it changes from 1.5e-5, to 1.8e-5, 2.0e-5, and 2.5e-5.

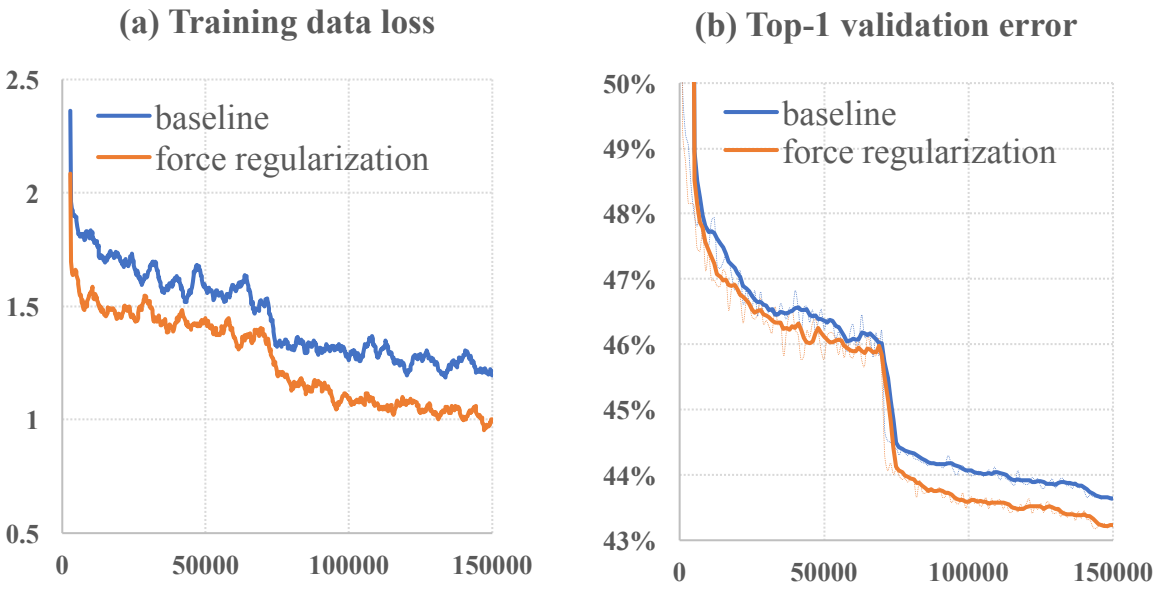


Figure 3.6: Training data loss and top-1 validation error vs. iteration when fine-tuning *AlexNet* which is decomposed to the same ranks

Because *Force Regularization* coordinates more useful weight information in a low-rank space, it essentially can provide a better training initialization for the DNNs that are decomposed by LRA. Fig. 3.6 plots the training data loss and top-1 validation error of *AlexNet*, which is decomposed to the same ranks by PCA. The baseline is the original *AlexNet* and the other *AlexNet* is coordinated by *Force Regularization*. The figure shows that the error sharply converges to a low level after a few iterations, indicating LRA provides a very good initialization for the low-rank DNNs. Training it from scratch has significant accuracy loss. More importantly, DNNs coordinated by *Force Regularization* can converge faster to a lower error.

Besides PCA [LWF+15][ZZHS16], we also evaluated the effectiveness of *Force Regularization* when integrating it with SVD [DZB+14][TXWE15] or k-means clustering [DZB+14][Bau15]. Table 3.3 compares the accuracies of *AlexNet* decomposed by different LRA methods. All LRAs preserve the same ranks in all layers, which means the decomposed *AlexNet* have the same network structure. In summary, PCA and SVD obtain similar accuracy and surpass k-means clustering. Due to the limited

Table 3.3: The accuracy of different LRA under the same ranks.

Force	LRA	Top-1 error
None	PCA	43.21%
	SVD†	43.27%
	k-means†	44.34%
ℓ_2-norm	PCA	43.25%
	SVD†	43.20%
	k-means†	44.80%

† SVD and k-means preserve the same ranks with PCA

pages, we adopt PCA as the representative in our study.

3.3.3 Acceleration of DNN Testing

In our experiments, we first train DNNs with *Force Regularization*, then decompose DNNs using LRA methods and fine-tune them to recover accuracy. In evaluation of speed, we omit small CIFAR-10 database and focus on large-scale DNNs on ImageNet, whose speed is a real concern. To prove the effective acceleration of *Force Regularization*, we adopt the speedup of state-of-the-art LRAs [ZZHS16][DSD+13][TXWE15] as our baseline. Our speedup is achieved in the case that the DNN filters are first coordinated by *Force Regularization* and then decomposed using the same LRAs. The practical GPU speed is profiled by the advanced hardware (NVIDIA GTX 1080) and software (cuDNN 5.0). The CPU speed is measured in Intel Xeon E5-2630 and ATLAS library. The batch size is 256.

Cross-filter LRA: We first evaluate the speedup of cross-filter LRA shown in Fig. 3.2. In previous works [DZB+14][TXWE15], the optimal rank in each layer can be selected layer-by-layer using cross validation. However, the number of hyper-parameters increases linearly with the depth of DNNs. To save development time, we utilize an identical *error percentage* $\frac{e_M}{e_0}$ across all layers as the single hyper-parameter although layer-wise rank selection may give better tradeoff. The rank in a layer is the

Table 3.4: The higher speedups of *AlexNet* by *Force Regularization*.

Force	Top-1 error		conv3	conv4	conv5
None	43.21%	rank	184	201	146
ℓ_2-norm	43.25%	rank	124	106	129
None	43.21%	GPU	1.58×	1.21×	1.15×
ℓ_2-norm	43.25%	GPU	2.16×	2.03×	1.33×
None	43.21%	CPU	1.78×	1.60×	1.47×
ℓ_2-norm	43.25%	CPU	2.45×	2.76×	1.64×
None	43.21%	theoretical	1.79×	1.72×	1.63×
ℓ_2-norm	43.25%	theoretical	2.65×	3.26×	1.85×

minimal M which has error $\leq \frac{e_M}{e_0}$.

As aforementioned in Section 3.3.2 and Table 3.2, the learned conv1 and conv2 of *AlexNet* are already in a very low-rank space and achieve good speedups using LRAs [DZB+14]. Thus we mainly focus on conv3-conv5 here. Table 3.4 summarizes the speedups of PCA approximation of *AlexNet* with and without ℓ_2-norm *Force Regularization*. With ignoble accuracy difference, *Force Regularization* successfully coordinates filters to a lower-rank space and accelerates the testing by a higher factor, comparing with the state-of-the-art LRA. Similar results are observed when applying ℓ_1-norm force.

Results in Table 3.4 also show that practical speedup is different from theoretical speedup. Generally, the difference is smaller in lower-performance processors. In CPU mode of Table 3.4, *Force Regularization* achieves 2× speedup of total convolutional time.

Speeding up state-of-the-art LRA: We also duplicate the state-of-the-art work [TXWE15] as the baseline[1] (lra_1). After LRA, *AlexNet* is fine-tuned with learning rate starting from 0.001 and divided by 10 at iteration 70,000 and 140,000. Fine-tuning terminates after 150,000 iterations.

[1]Code is provided by the authors in `https://github.com/chengtaipu/lowrankcnn/`

Table 3.5: The higher speedup factors by force regularization.

LRA	Force	Top-5 err.		conv3	conv4	conv5
lra_1 [TXWE15]	None	20.65%	GPU	0.86×	0.57×	0.40×
lra_2	None	19.93%	GPU	1.89×	1.57×	1.57×
lra_2	ℓ_2-norm	20.14%	GPU	2.25×	2.03×	1.60×
lra_2	ℓ_2-norm	21.68%	GPU	3.56×	3.01×	2.40×
			CPU	4.81×	4.00×	2.92×

The first row in Table 3.5 contains the results of the baseline [TXWE15], which don't scale well to the advanced "TITAN 1080 + cuDNN 5.0" in conv3–5. This is because 3×3 convolution is highly optimized in cuDNN 5.0, *e.g.*, using Winograd's minimal filtering algorithms [Lav15]. However, the baseline decomposes the 3×3 convolution to a pair of 3×1 and 1×3 convolution so that the optimized cuDNN is not fully exploited. This will be a common issue in the baseline, considering Winograd's algorithm is universally used and 3×3 convolution is one of the most common structures. We find that LRA in Fig. 3.2 can be utilized for conv 3–5 to solve this issue, because it can maintain the 3×3 shape. We name this LRA as lra_2, which decomposes conv1–conv2 using LRA in [TXWE15] and conv 3–5 using LRA of Fig. 3.2. The second row in Table 3.5 shows that our lra_2 can scale well to the hardware and software advances of "TITAN 1080 + cuDNN 5.0". More importantly, *Force Regularization* on conv3–5 can enforce them to more lightweight layers and attain higher speedup factors than lra_2 without using it. The result is shown in the third row, which in total achieves 2.03× speedup for the whole convolution in GPU. With small accuracy loss in row 4 of Table 3.5, *Force Regularization* achieves 2.50× speedup of total convolution on GPU and 4.05× on CPU.

Table 3.6 compares our method with state-of-the-art DNN acceleration methods, in CPU mode. When the speedup of total time was not reported by the authors, we estimate it by the weighted average speedups over all layers, where the weighting

Table 3.6: Comparison of speedup factor on *AlexNet* by state-of-the-art DNN acceleration methods.

Method	Top-5 err.	conv1	conv2	conv3	conv4	conv5	total
AlexNet in Caffe	19.97%	1.00×	1.00×	1.00×	1.00×	1.00×	**1.00**×
cp-decomposition [LGR+14]	20.97% (+1.00%)	–	4.00×	–	–	–	**1.27**×
one-shot [KPY+15]	21.67% (+1.70%)	1.48×	2.30×	3.84×	3.53×	3.13×	**2.52**×
SSL [WWW+16a]	19.58% (-0.39%)	1.00×	1.27×	1.64×	1.68×	1.32×	**1.35**×
	21.63% (+1.66%)	1.05×	3.37×	6.27×	9.73×	4.93×	**3.13**×
our lra_2	20.14% (+0.17%)	2.61×	6.06×	2.48×	2.20×	1.58×	**2.69**×
	21.68% (+1.71%)	2.65×	6.22×	4.81×	4.00×	2.92×	**4.05**×

coefficients are derived from the percentage of running time of each layer. In our hardware platform, conv1–conv5 respectively consume 15.89%, 28.25%, 24.32%, 18.70% and 12.84% testing time. The estimation is accurate, for example, we estimate 2.58× of total time in *one-shot* [KPY+15], which is very close to 2.52× reported by the authors. Comparing with both *cp-decomposition* and *one-shot* methods, our method can achieve higher accuracy and higher speedup. Comparing with *SSL*, with almost the same top-5 error (21.68% vs. 21.63%), we can attain higher speedup of 4.05× *vs.* 3.13×.

deep-compression [HMD15] reported 3× to 4× speedups in fully-connected layers when batch size was 1. However, convolution is the bottleneck of DNNs, *e.g.*, the convolution time in *AlexNet* is 5× of the time in fully-connected layers when profiled in our CPU platform. Moreover, no speedup was observed in the batching scenario as reported by the authors [HMD15]. More importantly, as we will show in Section 3.3.4, our work can work together with sparsity-based methods (*e.g.*, *SSL* or *deep-compression*) to obtain *lower-rank and sparse DNNs* and potentially further accelerate the testing of DNNs.

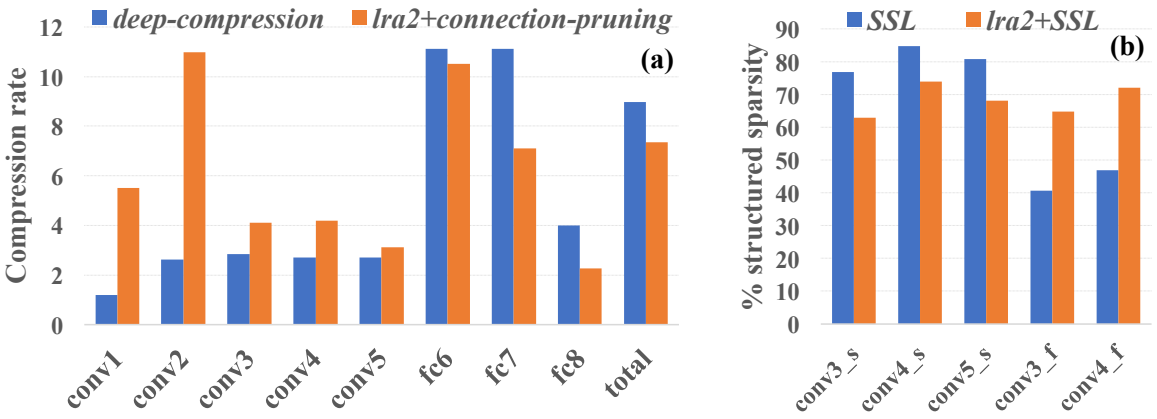


Figure 3.7: The results of sparsifying lightweight DNNs whose filters are coordinated to a lower-rank space by *Force Regularization*. In terms of *deep-compression* in (a), we only count the compression rate obtained from connection pruning for a fair comparison, but quantization and Huffman coding can also be utilized to improve the compression rate for our model. Based on *SSL* in (b), we enforce shape-wise sparsity on conv3_s, conv4_s and conv5_s to learn the shapes of basis filters meanwhile enforce filter-wise sparsity on conv3_f and conv4_f to learn the number of filters [WWW+16a]. As each convolutional layer in the lra_2 is decomposed to two small layers, we respectively denote the first and second small layer by suffixing "_s" and "_f". The baseline and our model have the same accuracy.

3.3.4 Lower-rank and Sparse DNNs

We sparsify the lightweight deep neural network (*i.e.*, the first one of lra_2 in Table 3.6), using Structured Sparsity Learning *SSL* [WWW+16a] or non-structured *connection-pruning* [PLW+17]. Note that Guided Sparsity Learning (*GSL*) is not adopted in our *connection-pruning* though better sparsity is achievable when applying it. Figure 3.7 summarizes the results.

Experiments prove that our method can work together with both structured and non-structured sparsity methods to further compress and accelerate models. Comparing with *deep-compression* in Figure 3.7(a), our model has comparable compression rates but 2.69× faster testing time. Typically, our model has higher compression rates in convolutional layers, which provides more space for computation reduction and generalizes better to modern DNNs (*ResNets-152* [HZRS16], for example, whose parameters in fc layers are only 4%). In Figure 3.7(b), our accelerated model can be

Table 3.7: Improved accuracy with *Discrimination Regularization*.

Net	Regularization	Top-1 error
AlexNet	None (baseline)	42.63%
AlexNet	ℓ_2-norm force	41.71%
AlexNet	ℓ_1-norm force	41.53%
ResNets-20	None (baseline)	8.82%
ResNets-20	ℓ_2-norm force	7.97%
ResNets-20	ℓ_1-norm force	8.02%

further accelerated using *SSL*. The shape-wise sparsity in conv3–5 of our model is slightly lower because our model is already aggressively compressed by LRA. The higher filter-wise sparsity, however, implies the orthogonality of our approach to *SSL*.

3.3.5 Generalization of Force Regularization

In convolutional layers, each filter basically extracts a discriminative feature, *e.g.*, an orientation-selective pattern or a color blob in the first layer [KSH12] or a high-level feature (*e.g.*, textures, faces, *etc.*) in deeper layers [ZF14]. The discrimination among filters is important for classification performance. Our method can coordinate filters for more lightweight DNNs meanwhile maintain the discrimination. It can also be generalized to learn more discriminative filters to improve the accuracy. The extension to *Discrimination Regularization* is straightforward but effective: the opposite gradient of *Force Regularization* (*i.e.*, $\lambda_s < 0$) is utilized to update the filter. In this scenario, it works as the *repulsive force* to repel surrounding filters and enhance the discrimination. Table 3.7 summarizes the improved accuracy of state-of-the-art DNNs.

Chapter 4

Scalable Deep Learning with Stochastic Low-precision Gradients

As explained in Section 1.2, communication is the bottleneck in distributed deep learning. In this dissertation, a new gradient quantization method will be proposed. The method is an orthogonal approach to gradient sparsification methods. More specifically, we propose *TernGrad* that quantizes gradients to ternary levels $\{-1, 0, 1\}$ to reduce the overhead of *gradient synchronization*. Furthermore, we propose *scaler sharing* and *parameter localization*, which can replace *parameter synchronization* with a low-precision gradient pulling. Comparing with previous works, our major contributions include: (1) we use ternary values for gradients to reduce communication; (2) we mathematically prove the convergence of *TernGrad* in general by proposing a statistical bound on gradients; (3) we propose *layer-wise ternarizing* and *gradient clipping* to move this bound closer toward the bound of standard SGD. These simple techniques successfully improve the convergence; (4) we build a performance model to evaluate the speed of training methods with compressed gradients, like *TernGrad*.

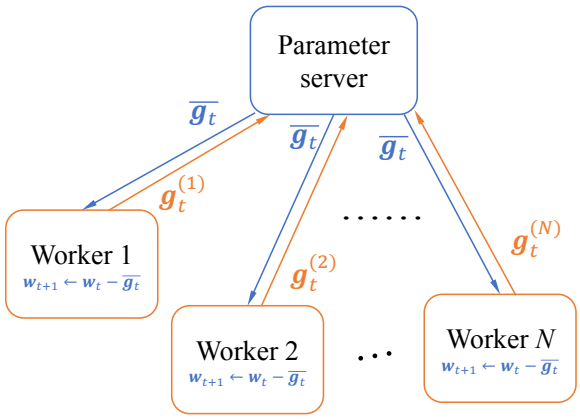


Figure 4.1: Distributed SGD with data parallelism

4.1 Problem Formulation and Our Approach

4.1.1 Problem Formulation and *TernGrad*

Figure 4.1 formulates the distributed training problem of synchronous SGD using data parallelism. At iteration t, a mini-batch of training samples are split and fed into multiple workers ($i \in \{1, ..., N\}$). Worker i computes the gradients $\boldsymbol{g}_t^{(i)}$ of parameters *w.r.t.* its input samples $\boldsymbol{z}_t^{(i)}$. All gradients are first synchronized and averaged at *parameter server*, and then sent back to update workers. Note that parameter server in most implementations [DCM+12][LASY14] are used to preserve shared *parameters*, while here we utilize it in a slightly different way of maintaining shared *gradients*. In Figure 4.1, each worker keeps a copy of parameters locally. We name this technique as *parameter localization*. The parameter consistency among workers can be maintained by random initialization with an identical seed. *Parameter localization* changes the communication of parameters in floating-point form to the transfer of quantized gradients that require much lighter traffic. Note that our proposed *TernGrad* can be integrated with many settings like *Asynchronous SGD* [DCM+12][RRWN11], even though the scope of this paper only focuses on the distributed SGD in Figure 4.1.

Algorithm 1 formulates the t-th iteration of *TernGrad* algorithm according to Figure 4.1. Most steps of *TernGrad* remain the same as traditional distributed training, except that gradients shall be quantized into ternary precision before sending to parameter server. More specific, $ternarize(\cdot)$ aims to reduce the communication volume of gradients. It randomly quantizes gradient $\boldsymbol{g}_t$ [1] to a ternary vector with values $\in \{-1, 0, +1\}$. Formally, with a random binary vector $\boldsymbol{b}_t$, $\boldsymbol{g}_t$ is ternarized as

$$\tilde{\boldsymbol{g}}_t = ternarize(\boldsymbol{g}_t) = s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t, \tag{4.1}$$

where

$$s_t \triangleq max\left(abs\left(\boldsymbol{g}_t\right)\right) \triangleq ||\boldsymbol{g}_t||_\infty \tag{4.2}$$

is a *scaler*, *e.g. maximum norm*, that can shrink ± 1 to a much smaller amplitude. $\circ$ is the Hadamard product. $sign(\cdot)$ and $abs(\cdot)$ respectively returns the sign and absolute value of each element. Giving a $\boldsymbol{g}_t$, each element of $\boldsymbol{b}_t$ independently follows the Bernoulli distribution

$$\begin{cases} P(b_{tk} = 1 \mid \boldsymbol{g}_t) = |g_{tk}|/s_t \\ P(b_{tk} = 0 \mid \boldsymbol{g}_t) = 1 - |g_{tk}|/s_t \end{cases}, \tag{4.3}$$

where b_{tk} and g_{tk} is the k-th element of $\boldsymbol{b}_t$ and $\boldsymbol{g}_t$, respectively. This *stochastic rounding*, instead of deterministic one, is chosen by both our study and QSGD [AGL+17], as stochastic rounding has an unbiased expectation and has been successfully studied for low-precision processing [HCS+16][GAGN15].

Theoretically, ternary gradients can at least reduce the *worker-to-server* traffic by a factor of $32/log_2(3) = 20.18\times$. Even using 2 bits to encode a ternary gradient, the reduction factor is still $16\times$. In this work, we compare *TernGrad* with 32-bit gradients, considering 32-bit is the default precision in modern deep learning frameworks. Although a lower-precision (*e.g.* 16-bit) may be enough in some scenarios,

[1] Here, the superscript of $\boldsymbol{g}_t$ is omitted for simplicity.

it will not undervalue *TernGrad*. As aforementioned, *parameter localization* reduces *server-to-worker* traffic by pulling quantized gradients from servers. However, summing up ternary values in $\sum_i \tilde{\boldsymbol{g}}_t^{(i)}$ will produce more possible levels and thereby the final averaged gradient $\overline{\boldsymbol{g}_t}$ is no longer ternary as shown in Figure 4.2(d). It emerges as a critical issue when workers use different scalers $s_t^{(i)}$. To minimize the number of levels, we propose a shared scaler

$$s_t = max(\{s_t^{(i)}\} : i = 1...N) \tag{4.4}$$

across all the workers. We name this technique as *scaler sharing*. The sharing process has a small overhead of transferring $2N$ floating scalars. By integrating *parameter localization* and *scaler sharing*, the maximum number of levels in $\overline{\boldsymbol{g}_t}$ decreases to $2N + 1$. As a result, the *server-to-worker* communication reduces by a factor of $32/log_2(1 + 2N)$, unless $N \geq 2^{30}$.

Algorithm 1 *TernGrad*: distributed SGD training using ternary gradients.

Worker : $i = 1, ..., N$

1 Input $\boldsymbol{z}_t^{(i)}$, a part of a mini-batch of training samples $\boldsymbol{z}_t$

2 Compute gradients $\boldsymbol{g}_t^{(i)}$ under $\boldsymbol{z}_t^{(i)}$

3 Ternarize gradients to $\tilde{\boldsymbol{g}}_t^{(i)} = ternarize(\boldsymbol{g}_t^{(i)})$

4 Push ternary $\tilde{\boldsymbol{g}}_t^{(i)}$ to the server

5 Pull averaged gradients $\overline{\boldsymbol{g}_t}$ from the server

6 Update parameters $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \eta \cdot \overline{\boldsymbol{g}_t}$

Server :

7 Average ternary gradients $\overline{\boldsymbol{g}_t} = \sum_i \tilde{\boldsymbol{g}}_t^{(i)}/N$

4.1.2 Convergence Analysis and Gradient Bound

We analyze the convergence of *TernGrad* in the framework of online learning systems. An online learning system adapts its parameter $\boldsymbol{w}$ to a sequence of observations to maximize performance. Each observation $\boldsymbol{z}$ is drawn from an unknown distribution, and a loss function $Q(\boldsymbol{z}, \boldsymbol{w})$ is used to measure the performance of current system

with parameter $\boldsymbol{w}$ and input $\boldsymbol{z}$. The minimization target then is the loss expectation

$$C(\boldsymbol{w}) \triangleq \mathbf{E}\left\{Q(\boldsymbol{z}, \boldsymbol{w})\right\}. \tag{4.5}$$

In *General Online Gradient Algorithm* (GOGA) [Bot98], parameter is updated at learning rate γ_t as

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \boldsymbol{g}_t = \boldsymbol{w}_t - \gamma_t \cdot \nabla_{\boldsymbol{w}} Q(\boldsymbol{z}_t, \boldsymbol{w}_t), \tag{4.6}$$

where

$$\boldsymbol{g} \triangleq \nabla_{\boldsymbol{w}} Q(\boldsymbol{z}, \boldsymbol{w}) \tag{4.7}$$

and the subscript t denotes observing step t. In GOGA, $\mathbf{E}\left\{\boldsymbol{g}\right\}$ is the gradient of the minimization target in Eq. (4.5).

According to Eq. (4.1), the parameter in *TernGrad* is updated, such as

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \left(s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right), \tag{4.8}$$

where $s_t \triangleq ||\boldsymbol{g}_t||_\infty$ is a *random variable* depending on $\boldsymbol{z}_t$ and $\boldsymbol{w}_t$. As $\boldsymbol{g}_t$ is known for given $\boldsymbol{z}_t$ and $\boldsymbol{w}_t$, Eq. (4.3) is equivalent to

$$\begin{cases} P(b_{tk} = 1 \mid \boldsymbol{z}_t, \boldsymbol{w}_t) = |g_{tk}|/s_t \\ P(b_{tk} = 0 \mid \boldsymbol{z}_t, \boldsymbol{w}_t) = 1 - |g_{tk}|/s_t \end{cases}. \tag{4.9}$$

At any given $\boldsymbol{w}_t$, the expectation of ternary gradient satisfies

$$\mathbf{E}\left\{s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right\} = \mathbf{E}\left\{s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \mathbf{E}\left\{\boldsymbol{b}_t | \boldsymbol{z}_t\right\}\right\} = \mathbf{E}\left\{\boldsymbol{g}_t\right\} = \nabla_{\boldsymbol{w}} C(\boldsymbol{w}_t), \tag{4.10}$$

which is an unbiased gradient of minimization target in Eq. (4.5).

The convergence analysis of *TernGrad* is adapted from the convergence proof of GOGA presented in [Bot98]. We adopt two assumptions, which were used in analysis of the convergence of standard GOGA in [Bot98]. Without explicit mention, vectors indicate column vectors here.

Assumption 1. *$C(\boldsymbol{w})$ has a single minimum $\boldsymbol{w}^*$ and gradient $-\nabla_{\boldsymbol{w}} C(\boldsymbol{w})$ always points to $\boldsymbol{w}^*$, i.e.,*

$$\forall \epsilon > 0, \inf_{||\boldsymbol{w}-\boldsymbol{w}^*||^2>\epsilon} (\boldsymbol{w}-\boldsymbol{w}^*)^T \nabla_{\boldsymbol{w}} C(\boldsymbol{w}) > 0. \tag{4.11}$$

Convexity is a subset of Assumption 1, and we can easily find non-convex functions satisfying it.

Assumption 2. *Learning rate γ_t is positive and constrained as*

$$\begin{cases} \sum_{t=0}^{+\infty} \gamma_t^2 < +\infty \\ \sum_{t=0}^{+\infty} \gamma_t = +\infty \end{cases}, \tag{4.12}$$

which ensures γ_t decreases neither very fast nor very slow respectively.

We define the square of distance between current parameter $\boldsymbol{w}_t$ and the minimum $\boldsymbol{w}^*$ as

$$h_t \triangleq ||\boldsymbol{w}_t - \boldsymbol{w}^*||^2, \tag{4.13}$$

where $||\cdot||$ is ℓ_2 norm. We also define the set of all random variables before step t as

$$\boldsymbol{X}_t \triangleq (\boldsymbol{z}_{1\ldots t-1}, \boldsymbol{b}_{1\ldots t-1}). \tag{4.14}$$

Under Assumption 1 and Assumption 2, using Lyapunov process and Quasi-Martingales convergence theorem, L. Bottou [Bot98] proved

Lemma 1. *If $\exists A, B > 0$ s.t.*

$$\mathbf{E}\left\{\left(h_{t+1} - \left(1+\gamma_t^2 B\right) h_t\right) | \boldsymbol{X}_t\right\} \leq -2\gamma_t (\boldsymbol{w}_t - \boldsymbol{w}^*)^T \nabla_{\boldsymbol{w}} C(\boldsymbol{w}_t) + \gamma_t^2 A, \tag{4.15}$$

*then $C(\boldsymbol{z}, \boldsymbol{w})$ converges **almost surely** toward minimum $\boldsymbol{w}^*$, i.e., $P\left(\lim_{t\to+\infty} \boldsymbol{w}_t = \boldsymbol{w}^*\right) = 1$.*

We further make an assumption on the gradient as

Assumption 3 (Gradient Bound)**.** *The gradient* $\boldsymbol{g}$ *is bounded as*

$$\mathbf{E}\left\{||\boldsymbol{g}||_\infty \cdot ||\boldsymbol{g}||_1\right\} \leq A + B\,||\boldsymbol{w} - \boldsymbol{w}^*||^2\,, \tag{4.16}$$

where $A, B > 0$ *and* $||\cdot||_1$ *is* ℓ_1 *norm.*

With Assumption 3 and Lemma 1, we prove Theorem 2 (in Appendix A):

Theorem 2. *When online learning systems update as*

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \gamma_t \left(s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right) \tag{4.17}$$

using stochastic ternary gradients, they converge ***almost surely*** *toward minimum* $\boldsymbol{w}^*$*, i.e.,* $P\left(\lim_{t\to+\infty} \boldsymbol{w}_t = \boldsymbol{w}^*\right) = 1$.

Comparing with the gradient bound of standard GOGA [Bot98]

$$\mathbf{E}\left\{||\boldsymbol{g}||^2\right\} \leq A + B\,||\boldsymbol{w} - \boldsymbol{w}^*||^2\,, \tag{4.18}$$

the bound in Assumption 3 is stronger because

$$||\boldsymbol{g}||_\infty \cdot ||\boldsymbol{g}||_1 \geq ||\boldsymbol{g}||^2. \tag{4.19}$$

We propose *layer-wise ternarizing* and *gradient clipping* to make two bounds closer, which shall be explained in Section 4.1.3. A side benefit of our work is that, by following the similar proof procedure, we can prove the convergence of GOGA when Gaussian noise $\mathcal{N}(0, \sigma^2)$ is added to gradients [NVL$^+$15], under the gradient bound of

$$\mathbf{E}\left\{||\boldsymbol{g}||^2\right\} \leq A + B\,||\boldsymbol{w} - \boldsymbol{w}^*||^2 - \sigma^2. \tag{4.20}$$

Although the bound is also stronger, Gaussian noise encourages active exploration of parameter space and improves accuracy as was empirically studied in [NVL$^+$15]. Similarly, the randomness of ternary gradients also encourages space exploration and improves accuracy for some models, as shall be presented in Section 4.2.

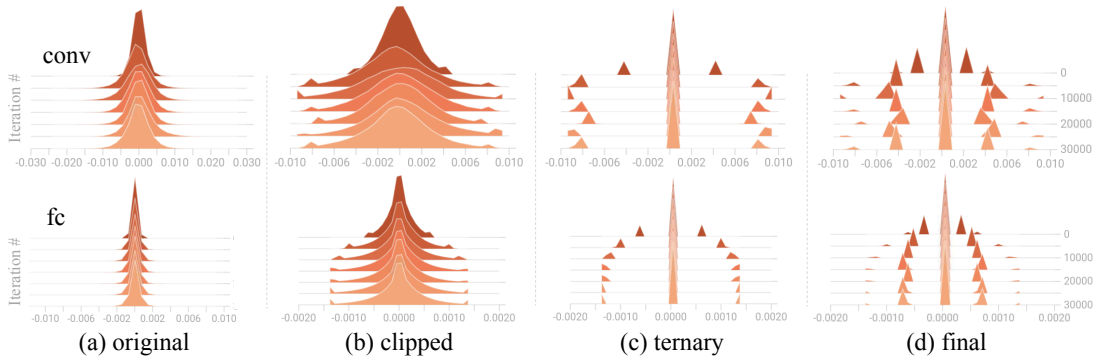


Figure 4.2: Histograms of (a) original floating gradients, (b) clipped gradients, (c) ternary gradients and (d) final averaged gradients. Visualization by TensorBoard. The DNN is *AlexNet* distributed on two workers, and vertical axis is the training iteration. As examples, top row visualizes the third convolutional layer and bottom one visualizes the first fully-connected layer.

4.1.3 Feasibility Considerations

The gradient bound of *TernGrad* in Assumption 3 is stronger than the bound in standard GOGA. Pushing the two bounds closer can improve the convergence of *TernGrad*. In Assumption 3, $||\boldsymbol{g}||_\infty$ is the maximum absolute value of *all* the gradients in the DNN. So, in a large DNN, $||\boldsymbol{g}||_\infty$ could be relatively much larger than most gradients, implying that the bound in *TernGrad* becomes much stronger. Considering the situation, we propose *layer-wise ternarizing* and *gradient clipping* to reduce $||\boldsymbol{g}||_\infty$ and therefore shrink the gap between these two bounds.

Layer-wise ternarizing is proposed based on the observation that the range of gradients in each layer changes as gradients are back propagated. Instead of adopting a large global maximum scaler, we independently ternarize gradients in each layer using the layer-wise scalers. More specific, we separately ternarize the gradients of biases and weights by using Eq. (4.1), where $\boldsymbol{g}_t$ could be the gradients of biases or weights in each layer. To approach the standard bound more closely, we can split gradients to more buckets and ternarize each bucket independently as D. Alistarh *et al.* [AGL+17] does. However, this will introduce more floating scalers and increase communication.

When the size of bucket is one, it degenerates to floating gradients.

Layer-wise ternarizing can shrink the bound gap resulted from the dynamic ranges of the gradients across layers. However, the dynamic range within a layer still remains as a problem. We propose *gradient clipping*, which limits the magnitude of each gradient g_i in $\boldsymbol{g}$ as

$$f(g_i) = \begin{cases} g_i & |g_i| \leq c\sigma \\ sign(g_i) \cdot c\sigma & |g_i| > c\sigma \end{cases}, \tag{4.21}$$

where σ is the standard derivation of gradients in $\boldsymbol{g}$. In distributed training, gradient clipping is applied to every worker before ternarizing. c is a hyper-parameter to select, but we cross validate it only once and use the constant in all our experiments. Specifically, we used a CNN [KSH12] trained on CIFAR-10 by momentum SGD with staircase learning rate and obtained the optimal $c = 2.5$. Suppose the distribution of gradients is close to Gaussian distribution as shown in Figure 4.2(a), very few gradients can drop out of $[-2.5\sigma, 2.5\sigma]$. Clipping these gradients in Figure 4.2(b) can significantly reduce the scaler but slightly changes the length and direction of original $\boldsymbol{g}$. Numerical analysis shows that *gradient clipping* with $c = 2.5$ only changes the length of $\boldsymbol{g}$ by $1.0\% - 1.5\%$ and its direction by $2^\circ - 3^\circ$. In our experiments, $c = 2.5$ remains valid across multiple databases (MNIST, CIFAR-10 and ImageNet), various network structures (*LeNet*, *CifarNet*, *AlexNet*, *GoogLeNet*, *etc*) and training schemes (momentum, vanilla SGD, adam, *etc*).

The effectiveness of *layer-wise ternarizing* and *gradient clipping* can also be explained as follows. When the scalar s_t in Eq. (4.1) and Eq. (4.3) is very large, most gradients have a high possibility to be ternarized to zeros, leaving only a few gradients to large-magnitude values. The scenario raises a severe parameter update pattern: most parameters keep unchanged while others likely overshoot. This will introduce large training variance. Our experiments on *AlexNet* show that by applying both

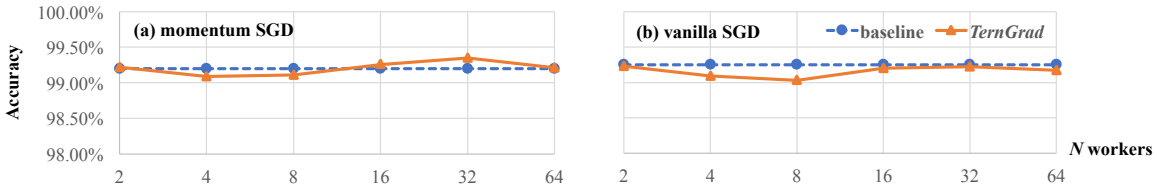


Figure 4.3: Accuracy vs. worker number for baseline and *TernGrad*, trained with (a) momentum SGD or (b) vanilla SGD. In all experiments, total mini-batch size is 64 and maximum iteration is $10K$.

layer-wise ternarizing and *gradient clipping* techniques, *TernGrad* can converge to the same accuracy as standard SGD. Removing any of the two techniques can result in accuracy degradation, *e.g.*, 3% top-1 accuracy loss without applying *gradient clipping* as we shall show in Table 4.2.

4.2 Experiments

We first investigate the convergence of *TernGrad* under various training schemes on relatively small databases and show the results in Section 4.2.1. Then the scalability of *TernGrad* to large-scale distributed deep learning is explored and discussed in Section 4.2.2. The experiments are performed by TensorFlow[AAB+16]. We maintain the exponential moving average of parameters by employing an exponential decay of 0.9999 [PCM+17]. The accuracy is evaluated by the final averaged parameters. This gives slightly better accuracy in our experiments. For fair comparison, in each pair of comparative experiments using either floating or ternary gradients, all the other training hyper-parameters are the same unless differences are explicitly pointed out. In experiments, when SGD with momentum is adopted, momentum value of 0.9 is used. When polynomial decay is applied to decay the *learning rate* (LR), the power of 0.5 is used to decay LR from the base LR to zero.

Table 4.1: Results of *TernGrad* on *CifarNet*.

SGD	base LR	total mini-batch size	iterations	gradients	workers	accuracy
Adam	0.0002	128	300K	floating	2	86.56%
				TernGrad	2	85.64% (-0.92%)
Adam	0.0002	2048	18.75K	floating	16	83.19%
				TernGrad	16	82.80% (-0.39%)

4.2.1 Integrating with Various Training Schemes

We study the convergence of *TernGrad* using *LeNet* on MNIST and a ConvNet [KSH12] (named as *CifarNet*) on CIFAR-10. *LeNet* is trained without data augmentation. While training *CifarNet*, images are randomly cropped to 24×24 images and mirrored. Brightness and contrast are also randomly adjusted. During the testing of *CifarNet*, only center crop is used. Our experiments cover the scope of SGD optimizers over vanilla SGD, SGD with momentum [Qia99] and Adam [KB14].

Figure 4.3 shows the results of *LeNet*. All are trained using polynomial LR decay with weight decay of 0.0005. The base learning rates of momentum SGD and vanilla SGD are 0.01 and 0.1, respectively. Given the total mini-batch size M and the worker number N, the mini-batch size per worker is M/N. Without explicit mention, mini-batch size refers to the total mini-batch size in this work. Figure 4.3 shows that *TernGrad* can converge to the similar accuracy within the same iterations, using momentum SGD or vanilla SGD. The maximum accuracy gain is 0.15% and the maximum accuracy loss is 0.22%. Very importantly, the communication time per iteration can be reduced. The figure also shows that *TernGrad* generalizes well to distributed training with large N. No degradation is observed even for $N = 64$, which indicates one training sample per iteration per worker.

Table 4.1 summarizes the results of *CifarNet*, where all trainings terminate after the same epochs. Adam SGD is used for training. Instead of keeping total mini-batch size unchanged, we maintain the mini-batch size per worker. Therefore, the total mini-

batch size linearly increases as the number of workers grows. Though the base learning rate of 0.0002 seems small, it can achieve better accuracy than larger ones like 0.001 for baseline. In each pair of experiments, *TernGrad* can converge to the accuracy level with less than 1% degradation. The accuracy degrades under a large mini-batch size in both baseline and *TernGrad*. This is because parameters are updated less frequently and large-batch training tends to converge to poorer sharp minima [KMN+17]. However, the noise inherent in *TernGrad* can help converge to better flat minimizers [KMN+17], which could explain the smaller accuracy gap between the baseline and *TernGrad* when the mini-batch size is 2048. In our experiments of *AlexNet* in Section 4.2.2, *TernGrad* even improves the accuracy in the large-batch scenario. This attribute is beneficial for distributed training as a large mini-batch size is usually required.

4.2.2 Scaling to Large-scale Deep Learning

We also evaluate *TernGrad* by *AlexNet* and *GoogLeNet* trained on ImageNet. It is more challenging to apply *TernGrad* to large-scale DNNs. It may result in some accuracy loss when simply replacing the floating gradients with ternary gradients while keeping other hyper-parameters unchanged. However, we are able to train large-scale DNNs by *TernGrad* successfully after making some or all of the following changes: (1) decreasing dropout ratio to keep more neurons; (2) using smaller weight decay; and (3) disabling ternarizing in the last classification layer. Dropout can regularize DNNs by adding randomness, while *TernGrad* also introduces randomness. Thus, dropping fewer neurons helps avoid over-randomness. Similarly, as the randomness of *TernGrad* introduces regularization, smaller weight decay may be adopted. We suggest not to apply ternarizing to the last layer, considering that the one-hot encoding of labels generates a skew distribution of gradients and the symmetric ternary encoding $\{-1, 0, 1\}$ is not optimal for such a skew distribution. Though asymmetric ternary levels could be an option, we decide to stick to floating gradients in the last layer for

Table 4.2: Accuracy comparison for *AlexNet*.

base LR	mini-batch size	workers	iterations	gradients	weight decay	DR†	top-1	top-5
0.01	256	2	370K	floating	0.0005	0.5	57.33%	80.56%
				TernGrad	0.0005	0.2	57.61%	80.47%
				TernGrad-noclip ‡	0.0005	0.2	54.63%	78.16%
0.02	512	4	185K	floating	0.0005	0.5	57.32%	80.73%
				TernGrad	0.0005	0.2	57.28%	80.23%
0.04	1024	8	92.5K	floating	0.0005	0.5	**56.62%**	80.28%
				TernGrad	0.0005	0.2	**57.54%**	80.25%

† DR: dropout ratio, the ratio of dropped neurons. ‡ *TernGrad* without gradient clipping.

simplicity. The overhead of communicating these floating gradients is small, as the last layer occupies only a small percentage of total parameters, like 6.7% in *AlexNet* and 3.99% in *ResNet-152* [HZRS16].

All DNNs are trained by momentum SGD with Batch Normalization [IS15] on convolutional layers. *AlexNet* is trained by the hyper-parameters and data augmentation depicted in Caffe. *GoogLeNet* is trained by polynomial LR decay and data augmentation in [SVI+16]. Our implementation of *GoogLeNet* does not utilize any auxiliary classifiers, that is, the loss from the last softmax layer is the total loss. More training hyper-parameters are reported in corresponding tables and published source code. Validation accuracy is evaluated using only the central crops of images.

The results of *AlexNet* are shown in Table 4.2. Mini-batch size per worker is fixed to 128. For fast development, all DNNs are trained through the same epochs of images. In this setting, when there are more workers, the number of iterations becomes smaller and parameters are less frequently updated. To overcome this problem, we increase the learning rate for large-batch scenario [Li17]. Using this scheme, SGD with floating gradients successfully trains *AlexNet* to similar accuracy, for mini-batch size of 256 and 512. However, when mini-batch size is 1024, the top-1 accuracy drops 0.71% for the same reason as we point out in Section 4.2.1.

TernGrad converges to approximate accuracy levels regardless of mini-batch

Table 4.3: Accuracy comparison for *GoogLeNet*.

base LR	mini-batch size	workers	iterations	gradients	weight decay	DR	top-5
0.04	128	2	600K	floating	4e-5	0.2	88.30%
				TernGrad	1e-5	0.08	86.77%
0.08	256	4	300K	floating	4e-5	0.2	87.82%
				TernGrad	1e-5	0.08	85.96%
0.10	512	8	300K	floating	4e-5	0.2	89.00%
				TernGrad	2e-5	0.08	86.47%

size. Notably, it improves the top-1 accuracy by 0.92% when mini-batch size is 1024, because its inherent randomness encourages to escape from poorer sharp minima [NVL+15][KMN+17]. Figure 4.4 plots training details vs. iteration when mini-batch size is 512. Figure 4.4(a) shows that the convergence curve of *TernGrad* matches well with the baseline's, demonstrating the effectiveness of *TernGrad*. The training efficiency can be further improved by reducing communication time as shall be discussed in Section 4.3. The training data loss in Figure 4.4(b) shows that *TernGrad* converges to a slightly lower level, which further proves the capability of *TernGrad* to minimize the target function even with ternary gradients. A smaller dropout ratio in *TernGrad* can be another reason of the lower loss. Figure 4.4(c) simply illustrate that on average 71.32% gradients of a fully-connected layer (fc6) are ternarized to zeros.

Finally, we summarize the results of *GoogLeNet* in Table 4.3. On average, the accuracy loss is less than 2%. In *TernGrad*, we adopted all that hyper-parameters

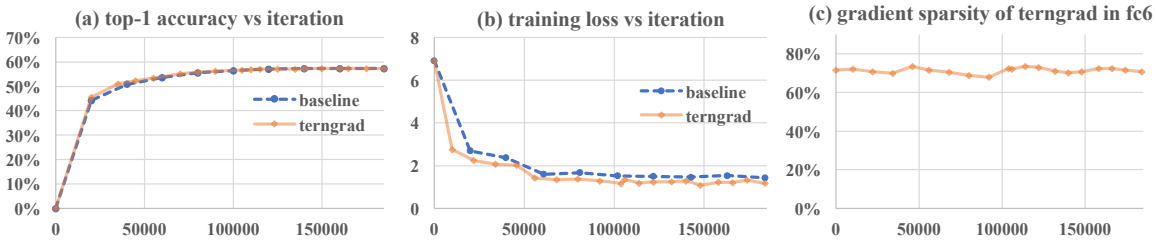


Figure 4.4: *AlexNet* trained on 4 workers with mini-batch size 512: (a) top-1 validation accuracy, (b) training data loss and (c) sparsity of gradients in first fully-connected layer (fc6) vs. iteration.

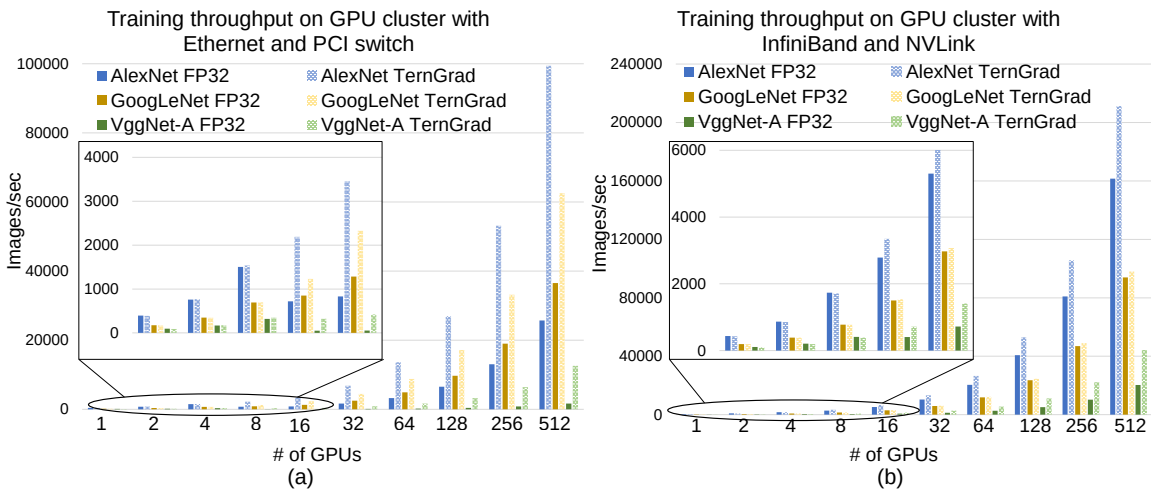


Figure 4.5: Training throughput on two different GPUs clusters: (a) 128-node GPU cluster with 1Gbps Ethernet, each node has 4 NVIDIA GTX 1080 GPUs and one PCI switch; (b) 128-node GPU cluster with 100 Gbps InfiniBand network connections, each node has 4 NVIDIA Tesla P100 GPUs connected via NVLink. Mini-batch size per GPU of *AlexNet*, *GoogLeNet* and *VggNet-A* is 128, 64 and 32, respectively

(except dropout ratio and weight decay) that are well tuned for the baseline [SLJ+15]. Tuning these hyper-parameters specifically for *TernGrad* could further optimize *TernGrad* and obtain higher accuracy.

4.3 Performance Model and Speedup

Our proposed *TernGrad* requires only three numerical levels $\{-1, 0, 1\}$, which can aggressively reduce the communication time. Moreover, our experiments in Section 4.2 demonstrate that within *the same iterations*, *TernGrad* can converge to *approximately the same accuracy* as its corresponding baseline. Consequently, a dramatical throughput improvement on the distributed DNN training is expected. Due to the resource and time constraint, unfortunately, we aren't able to perform the training of more DNN models like *VggNet-A* [SZ14] and distributed training beyond 8 workers. We plan to continue the experiments in our future work. We opt for using a performance model to conduct the scalability analysis of DNN models when utilizing up to 512

GPUs, with and without applying *TernGrad*. Three neural network models—*AlexNet*, *GoogLeNet* and *VggNet-A*—are investigated. In discussions of performance model, *performance* refers to training speed. Here, we extend the performance model that was initially developed for CPU-based deep learning systems [YRHC15] to estimate the performance of distributed GPUs/machines. The key idea is combining the lightweight profiling on single machine with analytical modeling for accurate performance estimation. In the interest of space, please refer to Appendix B for details of the performance model.

Figure 4.5 presents the training throughput on two different GPUs clusters. Our results show that *TernGrad* effectively increases the training throughput for the three DNNs. The speedup depends on the communication-to-computation ratio of the DNN, the number of GPUs, and the communication bandwidth. DNNs with larger communication-to-computation ratios (*e.g. AlexNet* and *VggNet-A*) can benefit more from *TernGrad* than those with smaller ratios (*e.g.*, *GoogLeNet*). Even on a very high-end HPC system with InfiniBand and NVLink, *TernGrad* is still able to double the training speed of *VggNet-A* on 128 nodes as shown in Figure 4.5(b). Moreover, the *TernGrad* becomes more efficient when the bandwidth becomes smaller, such as 1Gbps Ethernet and PCI switch in Figure 4.5(a) where *TernGrad* can have 3.04× training speedup for *AlexNet* on 8 GPUs.

Chapter 5

Escaping Sharp Minima in Large-Batch Deep Learning

As explained in Section 1.2, sharp minima are problems in large-batch SGD, which limits the scalability of distributed deep learning. In this chapter, we propose *SmoothOut* to escape sharp minima for better generalization. *SmoothOut* slightly perturbs DNN *function* by noise injecting or function reshaping, then averages all perturbed DNNs. Because sharp minima are sensitive to perturbation, slight perturbation can result in significant function increase at each sharp minimum, which means the averaged value will be high. In this way, sharp minima can be eliminated. Conversely, small perturbation only influences the margin of each flat region and the "flat bottom" still aligns well with the original "bottom". Averaging aligned "bottoms" can maintain the original minimum. Beyond this intuition, we prove that *SmoothOut* under uniform noises can eliminate sharp minima while maintaining flat minima. Note that we majorly use uniform noise for study as it is well motivated, but other noise types like Gaussian noise can fit into our new *SmoothOut* framework. Moreover, training over many perturbed DNNs for averaging is computation intensive. We propose *Stochastic SmoothOut*, which injects noise per iteration during SGD. We prove that *Stochastic SmoothOut* is equivalent to the original *SmoothOut* in expectation. Adaptive *SmoothOut* – *AdaSmoothOut*, is also proposed to further improve generalization

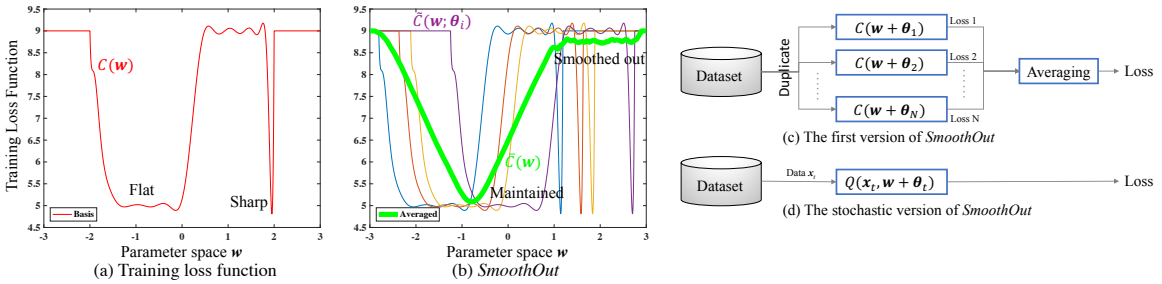


Figure 5.1: Illustration and framework of *SmoothOut*. (a) The training loss function of the basis model w.r.t. parameter $\boldsymbol{w}$. (b) Each thin curve represents a perturbed model; there are totally 1024 perturbations, but only four are plotted for cleaner visualization; the perturbation is done by slightly shifting the basis model in (a); the shift distance is randomly drawn from a uniform distribution; the green curve is a new model by averaging over all perturbations. (c) The first version of proposed *SmoothOut* framework in Eq. (5.1). (d) The *Stochastic SmoothOut* which randomly perturbs parameter $\boldsymbol{w}$ at each batch.

by adapting noise strength to filter norm. Our experiments show that *SmoothOut* and *AdaSmoothOut* can help to escape sharp minima and improve the generalization. *SmoothOut* and *AdaSmoothOut* are easy to be implemented and our code is at `https://github.com/wenwei202/smoothout`.

5.1 *SmoothOut*: Principles, Theory and Implementation

We first introduce our *SmoothOut* method and its principles in Section 5.1.1. To reduce computation complexity, *Stochastic SmoothOut* is proposed in Section 5.1.2; we prove that *Stochastic SmoothOut* is an unbiased approximation of deterministic *SmoothOut*. Section 5.1.3 implements *Stochastic SmoothOut* in back-propagation of DNNs. At last, an adaptive variant – *AdaSmoothOut*, is introduced.

5.1.1 Principles: Averaging Perturbed Models Smooths Out Sharp Minima

As [KMN+17] studied, sharp minima have large generalization gaps, because small distortion/shift of testing function from training function can significantly increase testing loss even though current parameter is a minimum of the training function[1]. Our optimization goal is to encourage convergence to flat minima for more robust models. Our solution is derived from the sensitivity nature of sharp minima. We intentionally inject noises into the *model* to smooth out sharp minima. The concept is illustrated in Figure 5.1(a)(b). We define $\boldsymbol{w}$ as a point in the parameter space, $C(\boldsymbol{w})$ as the training loss function and $\tilde{C}(\boldsymbol{w};\boldsymbol{\Theta})$ as a perturbation of $C(\boldsymbol{w})$. $\tilde{C}(\boldsymbol{w};\boldsymbol{\Theta})$ is parameterized by both $\boldsymbol{w}$ and $\boldsymbol{\Theta}$, where $\boldsymbol{\Theta}$ is a random vector to generate the perturbation. Instead of minimizing $C(\boldsymbol{w})$, we propose to minimize

$$\bar{C}(\boldsymbol{w}) = \mathbf{E}\left\{\tilde{C}(\boldsymbol{w};\boldsymbol{\Theta})\right\} \approx \frac{1}{N}\sum_{i=1}^{N}\tilde{C}(\boldsymbol{w};\boldsymbol{\theta}_i) \tag{5.1}$$

to find a optimal $\boldsymbol{w}^*$ for $C(\boldsymbol{w})$, where $\boldsymbol{\theta}_i$ is a sample of $\boldsymbol{\Theta}$ and N is the number of samples. For simplicity, we assume $C(\boldsymbol{w})$ has one flat minimum $\boldsymbol{w}_f$ and one sharp minimum $\boldsymbol{w}_s$, but the discussion can be generalized to $C(\boldsymbol{w})$ with multiple flat and sharp minima. Our goal is to design an auxiliary function $\bar{C}(\boldsymbol{w})$ such that its minimum within the original flat region can approximate $\boldsymbol{w}_f$, by satisfying the *Flat Constraint*

$$\left|\underset{\boldsymbol{w}\in\mathcal{D}(\boldsymbol{w}_f,\tau)}{\arg\min}\left(\bar{C}(\boldsymbol{w})\right) - \boldsymbol{w}_f\right| \leq \varphi, \tag{5.2}$$

meanwhile the sharp $\boldsymbol{w}_s$ is smoothed out, by satisfying the *Sharp Constraint*

$$\begin{aligned}\min_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)}\left(\bar{C}(\boldsymbol{w})\right) &\geq \max_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)}\left(C(\boldsymbol{w})\right)\\ &> \min_{\mathcal{D}(\boldsymbol{w}_f,\tau)}\left(\bar{C}(\boldsymbol{w})\right),\end{aligned} \tag{5.3}$$

[1]Figure 1 in their paper [KMN+17] illustrates this conception.

where $\mathcal{D}(\boldsymbol{w}, \varsigma)$ represents a region around $\boldsymbol{w}$, being constrained as

$$\mathcal{D}(\boldsymbol{w}, \varsigma) = \{\boldsymbol{w}' \in \mathbb{R}^m : |(\boldsymbol{w}' - \boldsymbol{w})_i| \leq \varsigma, \forall i \in \{1...m\}\}. \tag{5.4}$$

When φ is small and τ is large, Inequality (5.2) ensures that the auxiliary function $\bar{C}(\boldsymbol{w})$ maintains the minimality of $C(\boldsymbol{w})$ in the flat region; in the extreme case of $\varphi = 0$ and $\tau \to \infty$, the minimum of $\bar{C}(\boldsymbol{w})$ is exactly $\boldsymbol{w}_f$. Conversely, near the original sharp region, Inequality (5.3) ensures that minimality of $\bar{C}(\boldsymbol{w})$ is eliminated when ε is relatively large, because $\max_{\mathcal{D}(\boldsymbol{w}_s, \varepsilon)}(C(\boldsymbol{w}))$, the lower bound of $\bar{C}(\boldsymbol{w})$, increases rapidly by slightly increasing ε around the sharp minimum; in the extreme case of $\varepsilon \to \infty$, the lower bound is the maximum of $C(\boldsymbol{w})$. In a nutshell, a good design of $\bar{C}(\boldsymbol{w})$ allow a small φ, a large τ and a large ε. In this way, minimization process of $\bar{C}(\boldsymbol{w})$ will skip $\boldsymbol{w}_s$ and converge to $\boldsymbol{w}_f$. It is infeasible to find an optimal $\bar{C}(\boldsymbol{w})$ which minimizes φ and maximizes τ and ε, especially when $C(\boldsymbol{w})$ is a deep neural network. However, we find that, under the *Uniform Perturbation*

$$\begin{aligned} &\tilde{C}(\boldsymbol{w}; \boldsymbol{\Theta}) = C(\boldsymbol{w} + \boldsymbol{\Theta}) \\ &\text{where } \Theta_i \overset{\text{i.i.d.}}{\sim} U(-a, a) \text{ and } \forall i \in \{1, 2, ..., m\}, \end{aligned} \tag{5.5}$$

$\bar{C}(\boldsymbol{w})$ can well perform the purpose. $U(-a, a)$ is a uniform distribution within a range of $[-a, a]$. In this case, we have abused $\bar{C}(\boldsymbol{w}; a)$ as $\bar{C}(\boldsymbol{w})$ in the notation for simplicity. In the Appendix C, we prove that, under *Uniform Perturbation*, appropriate φ, τ and ε can be found to satisfy *Flat Constraint* and *Sharp Constraint*:

Theorem 3. *When $C(\boldsymbol{w})$ is symmetric in $\mathcal{D}(\boldsymbol{w}_f, \tau)|_{\tau > a}$, the minimum of φ is 0 to satisfy the Flat Constraint when $\bar{C}(\boldsymbol{w})$ is generated under the Uniform Perturbation.*

Theorem 4. *Suppose $C(\boldsymbol{w})$ is high dimensional ($\boldsymbol{w} \in \mathbb{R}^m, m \to \infty$) and is symmetric and strictly monotonic in $\mathcal{D}(\boldsymbol{w}_s, b)|_{b > a}$, then $\exists a$ such that Sharp Constraint is satisfied with $\varepsilon \to a^-$ when $\bar{C}(\boldsymbol{w})$ is generated under the Uniform Perturbation.*

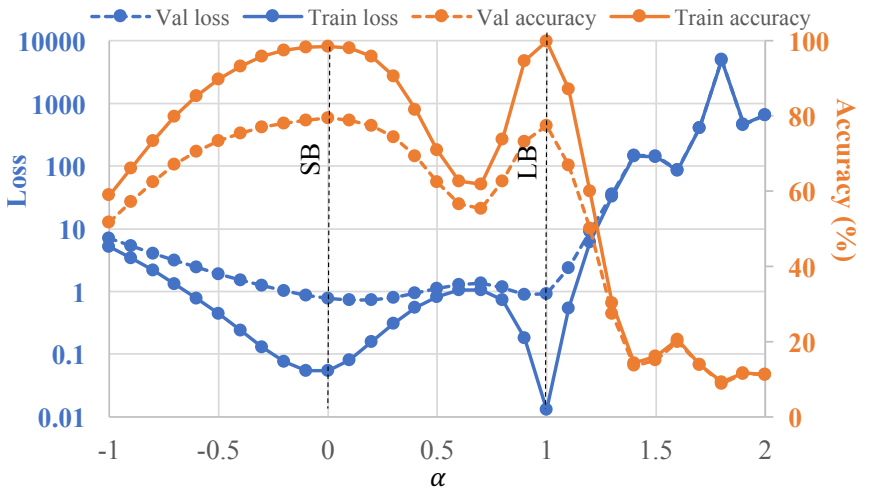
In theorems, the symmetry is assumed only near minima, and the loss surface does not have to be symmetric in the whole space. By referring to the visualization of loss landscapes of neural nets in [LXTG17], it is reasonable to make this assumption near minima.

Besides the rigor proof in the Appendix C, *SmoothOut* can be explained from the perspective of signal processing: imagining the parameter space as a time domain and the function as signals, then averaging is a low-pass filter which eliminates high-frequency signals (sharp regions) while maintains low-frequency signals (flat regions).

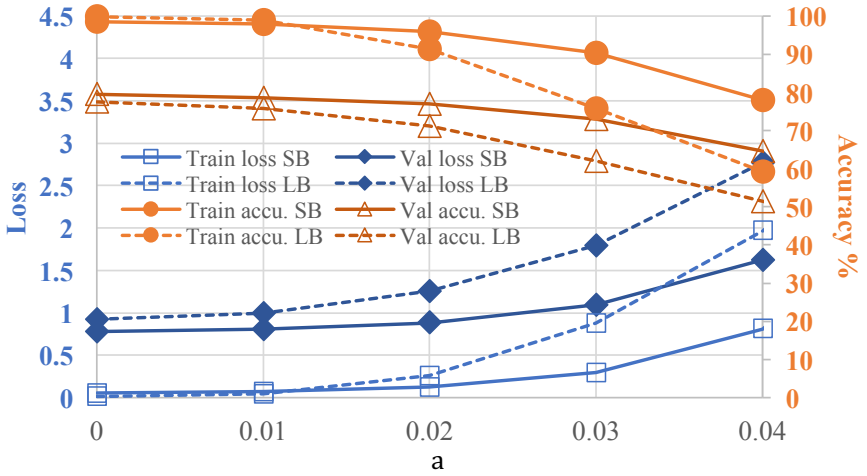
Figure 5.1(c) illustrates the framework of the proposed *SmoothOut* in SGD. All models share the same parameter $\boldsymbol{w}$. Before training starts, the i-th model is independently perturbed by $\boldsymbol{\theta}_i$; during training, all $\boldsymbol{\theta}_i$ are fixed and an identical batch of data is sent to all models for training. Because a large N is required for approximation in Eq. (5.1), the computation complexity and memory usage will be very high, especially when $C(\boldsymbol{w})$ is a deep neural network. In the next section, the *Stochastic SmoothOut* will be proposed to solve this issue.

5.1.2 Theory: *Stochastic SmoothOut* is Unbiased

To reduce the computation complexity and memory usage of *SmoothOut* in Figure 5.1(c), *Stochastic SmoothOut* is proposed in this section as shown in Figure 5.1(d). Instead of using multiple perturbed models to learn from identical data, only one model is trained. At the t-th batch of training data $\boldsymbol{x}_t$, the parameter $\boldsymbol{w}_t$ is first perturbed to $\boldsymbol{w}_t + \boldsymbol{\theta}_t$ and then $\boldsymbol{x}_t$ is fed into the model to calculate the loss function. We can prove that, in both frameworks, the outputs can approximate $\bar{C}(\boldsymbol{w})$ without bias.



(a) Sharpness visualization



(b) Sensitivity to noise

Figure 5.2: Notation:"SB": Small Batch (256); "LB": Large Batch (5000); "accu.": accuracy. (a) loss and accuracy vs. α, which controls $\boldsymbol{w}$ along the direction from SB minimum ($\boldsymbol{w}_f$) to LB minimum ($\boldsymbol{w}_s$); (b) loss and accuracy under influence of different strengths of noise. Dataset: CIFAR-10. Network: C_1 in [KMN+17] implemented by [HHS17]. Optimizer: Adam with 0.001 initial learning rate.

Formally, in Figure 5.1(c), the expectation of the output is

$$\begin{aligned}&\mathbf{E}_{\boldsymbol{\theta}_{1\ldots N}}\left\{\frac{1}{N}\sum_{i=1}^{N}C(\boldsymbol{w}+\boldsymbol{\theta}_i)\right\}\\&=\mathbf{E}_{\boldsymbol{\Theta}}\left\{C(\boldsymbol{w}+\boldsymbol{\Theta})\right\}=\bar{C}(\boldsymbol{w}).\end{aligned}\tag{5.6}$$

In online learning systems [Bot98] like Figure 5.1(d), the data $\boldsymbol{x}_t$ is independently generated from a random distribution and its online loss is obtained by model $Q(\boldsymbol{x}_t, \boldsymbol{w})$; the final loss function to minimize is the expectation of online loss under data distribution, *i.e.*,

$$C(\boldsymbol{w}) \triangleq \mathbf{E}_{\boldsymbol{X}}\left\{Q(\boldsymbol{x}, \boldsymbol{w})\right\}. \tag{5.7}$$

Therefore, in Figure 5.1(d), the expectation of the output is

$$\begin{aligned}\mathbf{E}\left\{Q(\boldsymbol{x}_t, \boldsymbol{w}+\boldsymbol{\theta}_t)\right\}&=\mathbf{E}_{\boldsymbol{\Theta}}\left\{\mathbf{E}_{\boldsymbol{X}}\left\{Q(\boldsymbol{x}, \boldsymbol{w}+\boldsymbol{\Theta})\right\}\right\}\\&=\mathbf{E}_{\boldsymbol{\Theta}}\left\{C(\boldsymbol{w}+\boldsymbol{\Theta})\right\}\\&=\bar{C}(\boldsymbol{w}).\end{aligned}\tag{5.8}$$

Consequently, both frameworks in Figure 5.1 can approximate $\bar{C}(\boldsymbol{w}) = \mathbf{E}\{\tilde{C}(\boldsymbol{w}; \boldsymbol{\Theta})\}$ in Eq. (5.1), but *Stochastic SmoothOut* is much more computation efficient. The only overhead of *Stochastic SmoothOut* is noise injection and denoising as will be shown. In the following sections, without explicit clarification, *SmoothOut* will refer to the stochastic version in Figure 5.1(d).

The reason why *SmoothOut* can eliminate sharp minima is that $C(\boldsymbol{w}_s)$ is more sensitive to noise than $C(\boldsymbol{w}_f)$, and we expect $\bar{C}(\boldsymbol{w}_s)$ increases faster than $\bar{C}(\boldsymbol{w}_f)$ as the noise strength a increases from 0. To verify this, we first train a DNN under a small batch size to get a flat minimum $\boldsymbol{w}_f$; second, $\boldsymbol{w} = \boldsymbol{w}_f$ is deployed into the framework in Figure 5.1(d); third, the whole train/validation dataset is fed to the framework in batch size of 100, and at each batch, the parameter is perturbed to $\boldsymbol{w} = \boldsymbol{w}_f + \boldsymbol{\theta}_t$; finally, the losses are averaged over all batches to estimate $\bar{C}(\boldsymbol{w}_f)$. The

same process is done using a large batch size for the same DNN to estimate $\bar{C}(\boldsymbol{w}_s)$. We scan a in a range to test the sensitivity of $\bar{C}(\boldsymbol{w}_f)$ and $\bar{C}(\boldsymbol{w}_s)$ to perturbation. Figure 5.2(a) visualizes the sharpness of $C(\boldsymbol{w})$ around $\boldsymbol{w}_f$ and $\boldsymbol{w}_s$, using the technique adopted in [KMN+17] which was originally proposed in [GVS14]. In Figure 5.2(a), each point on the loss curve is $(\boldsymbol{w}^+, C(\boldsymbol{w}^+))$ where $\boldsymbol{w}^+ = \alpha \cdot \boldsymbol{w}_s + (1-\alpha) \cdot \boldsymbol{w}_f$. The visualization is consistent with [KMN+17], which concluded that large-batch training converges to sharp minima. Figure 5.2(b) analyzes the sensitivity. For both training and validation datasets, $\bar{C}(\boldsymbol{w}_s)$ indeed increases faster than $\bar{C}(\boldsymbol{w}_f)$ as a increases. The accuracy curves have a similar trend. Sensitivity analyses of more DNNs and more datasets are included in the Appendix D. Therefore, a side outcome of this work is that we can use

$$s = \frac{\Delta\left(\bar{C}(\boldsymbol{w}^*; a)\right)}{\Delta a} \tag{5.9}$$

as a metric to measure the sharpness of $C(\boldsymbol{w})$ at minimum $\boldsymbol{w}^*$. A larger s means a sharper minimum.

At last, under our framework, we can view Dropout as an noise under Bernoulli distribution adapting its noise strength to the corresponding weight. Concretely, in Figure 5.1(d), $\theta_{ti} = -w_i$ with probability p and $\theta_{ti} = 0$ with probability $1-p$, where p is the dropout ratio. Under this view, Dropout can fit into our framework, but it cannot guide the convergence to sharp minima, because the strength of noise $\theta_{ti} = -w_i$ is too large.

5.1.3 Implementation: Back-propagation with Perturbation and Denoising

Algorithm 2 *SmoothOut* in Back Propagation

Input : Training dataset $\boldsymbol{X}$, total iterations T, model $Q(\boldsymbol{x}, \boldsymbol{w})$ with initial parameter $\boldsymbol{w} = \boldsymbol{w}_0$

1: **for** $t \in \{0, \ldots, T-1\}$ **do**
2: Randomly sample a batch data $\boldsymbol{x}_t$ from $\boldsymbol{X}$
3: Perturbation: $\boldsymbol{w}_t = \boldsymbol{w}_t + \boldsymbol{\theta}_t$ where $\theta_{ti} \overset{\text{i.i.d.}}{\sim} U(-a, a)$
4: Back-propagation: $\boldsymbol{g}_t = \frac{\partial Q(\boldsymbol{x}_t, \boldsymbol{w}_t)}{\partial \boldsymbol{w}_t}$
5: Denoising: $\boldsymbol{w}_t = \boldsymbol{w}_t - \boldsymbol{\theta}_t$
6: Updating: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \cdot \boldsymbol{g}_t$
7: **end for**

Output :

8 The model $Q(\boldsymbol{x}, \boldsymbol{w})$ with final parameter $\boldsymbol{w} = \boldsymbol{w}_T$

In Figure 5.1(d), the gradient to update parameter at iteration t is

$$\begin{aligned} \boldsymbol{g_t} &= \left. \frac{\partial Q(\boldsymbol{x}_t, \boldsymbol{w} + \boldsymbol{\theta}_t)}{\partial \boldsymbol{w}} \right|_{\boldsymbol{w}=\boldsymbol{w}_t} \\ &= \left. \nabla_{\boldsymbol{w}} Q(\boldsymbol{x}_t, \boldsymbol{w}) \right|_{\boldsymbol{w}=\boldsymbol{w}_t+\boldsymbol{\theta}_t} \cdot \frac{\partial(\boldsymbol{w} + \boldsymbol{\theta}_t)}{\partial \boldsymbol{w}}. \end{aligned} \tag{5.10}$$

Therefore, the parameter is updated as

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta_t \cdot \left. \nabla_{\boldsymbol{w}} Q(\boldsymbol{x}_t, \boldsymbol{w}) \right|_{\boldsymbol{w}=\boldsymbol{w}_t+\boldsymbol{\theta}_t}, \tag{5.11}$$

where η_t is the learning rate and the gradient is obtained by back propagation when the parameter value is $\boldsymbol{w}_t + \boldsymbol{\theta}_t$. Thus, *SmoothOut* can be implemented as Algorithm 2 as illustrated in Figure 5.3. This reveals a pitfall in implementation that the noise $\boldsymbol{\theta}_t$ added to $\boldsymbol{w}_t$ must be denoised before applying the gradient, which is also a key difference from existing noise injection approaches [NVL+15][Mob16][FAP+17][PHD+17].

As shown in Figure 5.3, the only overhead of *SmoothOut* is adding and subtracting noises, which is much more efficient than training multiple DNNs in Figure 5.1(c). Note that, although Algorithm 2 is proposed in the context of vanilla SGD, it can be extended to SGD variants by simply utilizing the gradient $\boldsymbol{g}_t$ for momentum accumulation, learning rate adaptation, and so on.

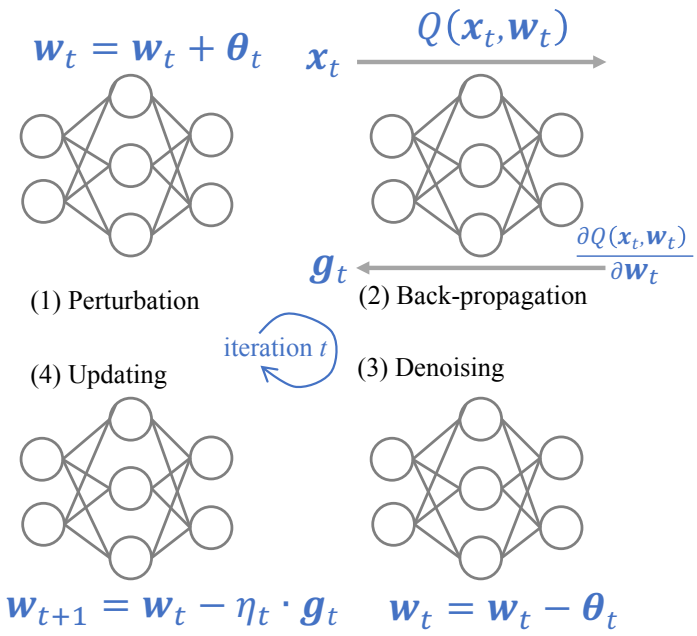


Figure 5.3: *SmoothOut* in back propagation.

5.1.4 Adaptive *SmoothOut* – *AdaSmoothOut*

Due to the fact that the weight distributions across all layers vary a lot, adding noise with a constant strength to all weights may over-perturb the layers with small weights while under-perturb others. The varying distribution is also the source of problem in visualizing the sharpness as pointed out in [LXTG17]. To overcome this, [LXTG17] proposed "filter normalization" and achieved more accurate visualization. Inspired by "filter normalization", in *SmoothOut*, the noises added to a filter are linearly scaled by ℓ_2 norm of the filter. In fully-connected layers, the noises are scaled per neuron, *i.e.*, all input connections of each neuron form a vector and noises are divided by ℓ_2 norm of the vector. We call it *Adaptive SmoothOut* (*AdaSmoothOut*) because it adapts the strength of noises to the filters instead of fixing the strength. Mathematically, suppose $\boldsymbol{w}^{(i)}$ is a vector of parameters in filter i and $\boldsymbol{\theta}^{(i)}$ is a noise vector, then adapted noise $\hat{\boldsymbol{\theta}}^{(i)}$ will be

$$\hat{\boldsymbol{\theta}}^{(i)} = a \cdot \frac{||\boldsymbol{w}^{(i)}||_2}{||\boldsymbol{\theta}^{(i)}||_2} \cdot \boldsymbol{\theta}^{(i)}, \tag{5.12}$$

Table 5.1: Sharpness reduction and generalization improvement for DNNs in [KMN+17].

DNN	Dataset	Batch size	Baseline	***SmoothOut***	Improvement	$(\mathcal{C}_\epsilon, A)$-sharpness change baseline → ***SmoothOut***
F_1	MNIST	256	98.49%	**98.61**%	0.12%	0.0000 → **0.0000**
		6000	98.01%	**98.42**%	0.41%	57.8246 → **5.4128**
C_1	CIFAR-10	256	79.67%	**81.72**%	2.05%	27.7448 → **0.0003**
		5000	77.30%	**80.34**%	3.04%	117.8266 → **0.0004**
C_3	CIFAR-100	256	47.99%	**51.17**%	3.18%	23.5103 → **0.0001**
		5000	44.37%	**48.43**%	4.06%	62.0682 → **10.7303**

where a controls the strength of noises. Adaptive noise is another key difference from noise injection in previous work. Our ablation study will show adaptive noise is more effective in improving generalization.

5.2 Experiments

We evaluate *SmoothOut* in MNIST [LBBH98], CIFAR-10 [KH09], CIFAR-100 [KH09] and ImageNet [DDS+09] dataset. *SmoothOut* and *AdaSmoothOut* are evaluated in small-batch ("SB") SGD and large-batch ("LB") SGD. $(\mathcal{C}_\epsilon, A)$-sharpness [KMN+17] is utilized to measure the sharpness of a minimum, which is solved using L-BFGS-B algorithm [BLNZ95]. In solving $(\mathcal{C}_\epsilon, A)$-sharpness, the full-space (*i.e.*, $A = I_n$) in the bounding box $\mathcal{C}_\epsilon$ (with $\epsilon = 5 \cdot 10^{-4}$) is explored to find the maximum for measurement. As L-BFGS-B is an estimation algorithm and may fail to find the exact maximum value, variance in measurements is observed. We run 5 experiments for each measurement, and use the *maximum* as the final sharpness metric. Unlike [KMN+17] which *averaged* over 5 runs, ours is more reasonable because $(\mathcal{C}_\epsilon, A)$-sharpness is based on measuring the *maximum* value around the box. In training with *SmoothOut*, a is the only additional hyper-parameter to tune, which controls the strength of noise. a is very robust because of the width of flat minima. More concretely, a is 0.0375 in all experiments of *SmoothOut* in Table 5.1 and Table 5.2. We believe the value of a is

network architecture dependent (*i.e.*, loss function dependent). We cross-validate it in small-batch SGD and directly use it in large-batch SGD without further tuning, and it generalizes well and improves accuracy in both small-batch SGD and large-batch SGD.

5.2.1 Convergence to Flatter Minima

We first adopt benchmarks by [KMN+17] to verify that *SmoothOut* can effectively guide both SB and LB SGD to flatter minima and thus improve the generalization (accuracy). The comparison is in Table 5.1. Figure 5.4 visualizes and compares the sharpness of baseline (C_3) and *SmoothOut*. Similar visualization results for F_1 and C_1 can be found in the Appendix D. Note that Keskar *et al.* [KMN+17] did not target on achieving state-of-the-art accuracy but studying the characteristics of minima, and we simply follow this purpose. Comparison in state-of-the-art models will be covered in Section 5.2.2.

In Table 5.1 and Figure 5.4, we observed consistency among sharpness, visualization, and generalization, that is, a smaller $(\mathcal{C}_\epsilon, A)$-sharpness, then a flatter region in the visualization and a higher accuracy. More importantly, the results indicate that (1) comparing with SB training, LB training converges to sharper minima with worse generalization, but *SmoothOut* can guide it to converge to flatter minima and closes the gap or even improves the accuracy; (2) the sharp minima problem also exist in SB training as shown in Figure 5.4(a), but *SmoothOut* can reduce the sharpness and improve the accuracy; (3) sharp minima problem is severer in LB training such that *SmoothOut* can improve more.

At last, we argue that the convergence of our method is stable although noises are injected; that is, different runs converge to similar accuracy under the same strength of injected noises. More specific, for C_1 in Table 5.1, accuracy standard

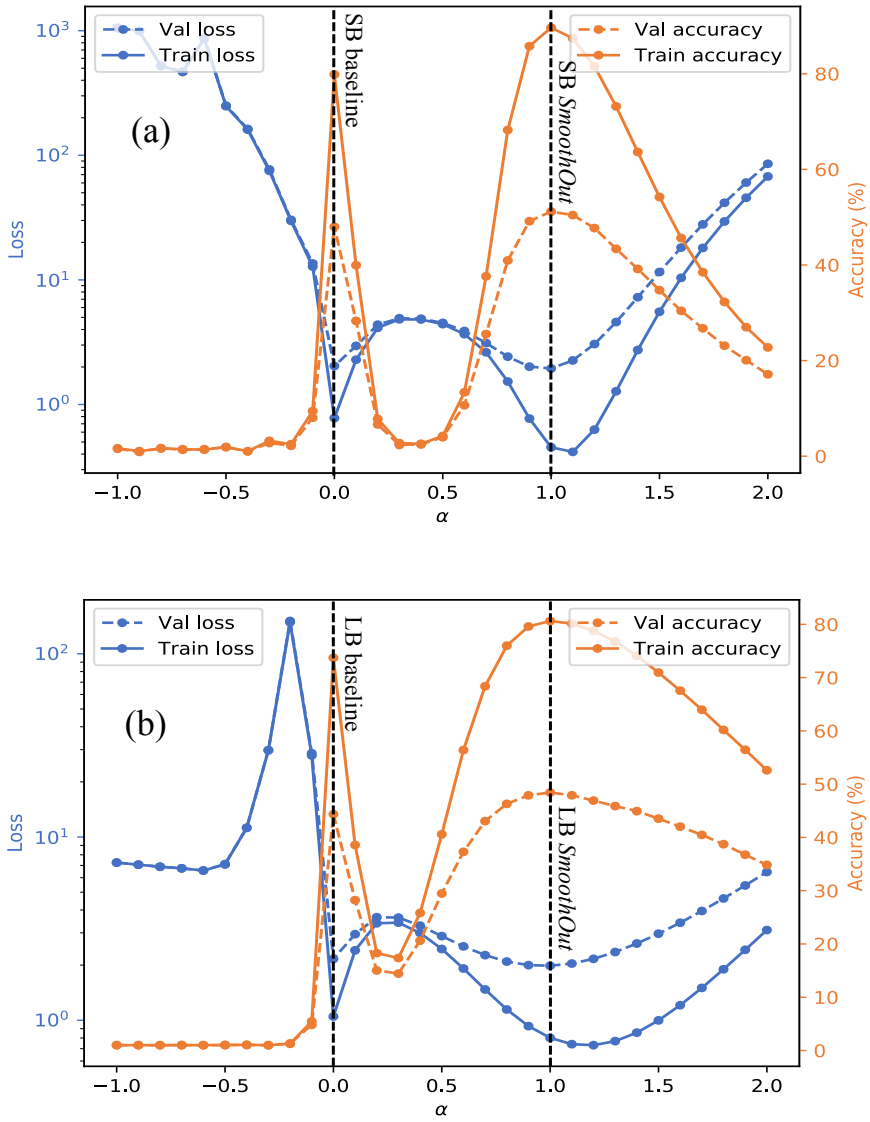


Figure 5.4: Sharpness of baseline and *SmoothOut* in (a) "SB" training and (b) "LB" training of C_3.

Table 5.2: *SmoothOut* and *AdaSmoothOut* improve the state-of-the-art baselines.

DNN	Dataset	Batch size	Epochs	LRS	Method	Accuracy
ResNet44	CIFAR-10	2048	400	square root	Baseline	91.02%
					SmoothOut	91.95%
					AdaSmoothOut	92.63%
ResNet44	CIFAR-100	1024	200	square root	Baseline	67.23%
					SmoothOut	68.68%
					AdaSmoothOut	70.09%
			400	square root	Baseline	68.62%
					SmoothOut	70.01%
					AdaSmoothOut	72.39%
			400	linear	Baseline	71.21%
					SmoothOut	71.67%
					AdaSmoothOut	72.85%
AlexNet	ImageNet	16384	60	square root	Baseline	47.64%
					AdaSmoothOut	52.53%
			120	square root	Baseline	54.24%
					AdaSmoothOut	55.51%
ResNet18	ImageNet	16384	90	square root	Baseline	66.75%
					AdaSmoothOut	67.11%

deviation is $\pm 0.33\%$, $\pm 0.12\%$, $\pm 0.24\%$ and $\pm 0.31\%$ in small-batch baseline, small-batch *SmoothOut*, large-batch baseline and large-batch *SmoothOut*, respectively.

5.2.2 Improving Generalization on the Top of State-of-the-art Solutions

In this section, we evaluate our method by state-of-the-art DNNs, including *ResNet44* on CIFAR-10 and CIFAR-100, *AlexNet* [KSH12] and *ResNet18* [HZRS16] on ImageNet. Table 5.2 summarizes all the results. As generalization issue is severer in LB training, we focus on LB training in this section. There are lots of proposed techniques to relieve the generalization issue in LB training [HHS17, GDG+17, JSH+18, ASF17, YGG17, SKL18], however, our method is orthogonal and we simply apply *SmoothOut* on the top of them to verify if *SmoothOut* can be combined with those state-of-the-art

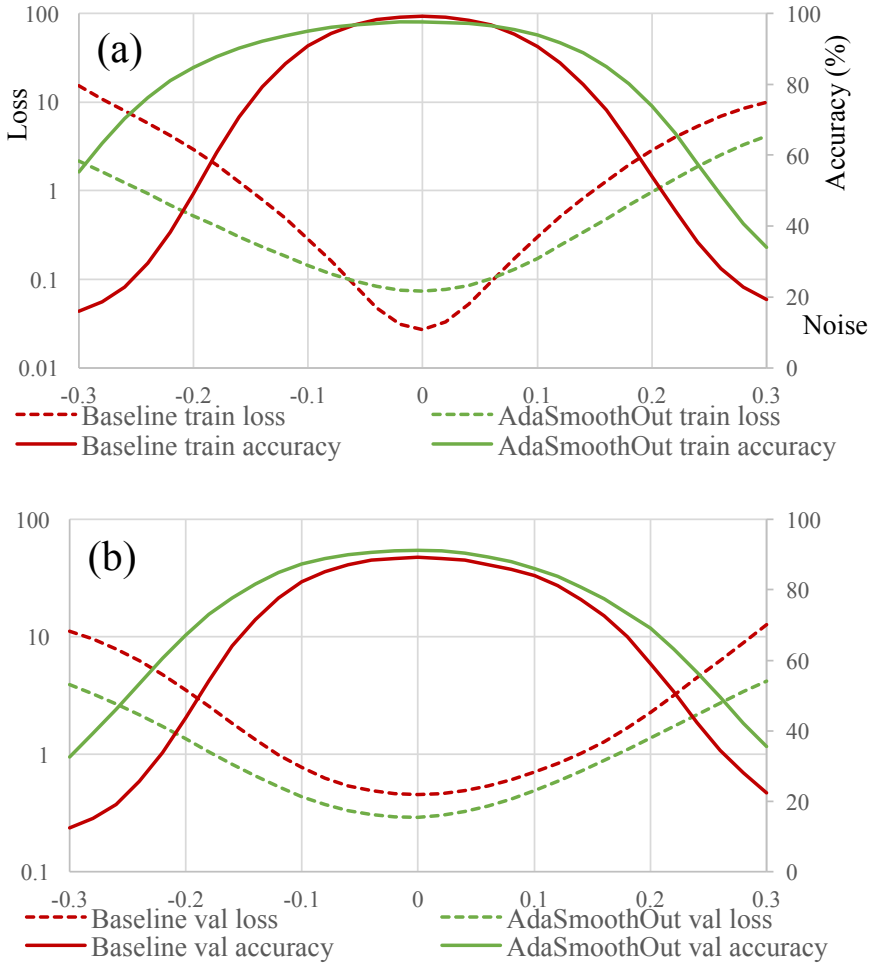


Figure 5.5: Sharpness visualization by "filter normalization" [LXTG17] using (a) training dataset and (b) validation dataset. The DNN is *ResNet44* trained by CIFAR-10 using baseline [HHS17] and *AdaSmoothOut*.

solutions. We are not able to duplicate all of those techniques, but we select Learning Rate Scaling (LRS), Ghost Batch Normalization (GBN) and Training Longer (TL) techniques [HHS17, GDG+17] as the representatives.

For LRS, [GDG+17] used linear LRS (*i.e.* learning rate is scaled linearly w.r.t. the batch size), while [HHS17] used square root LRS. The preferable LRS rule is dependent on the dataset and DNN [HHS17]. In our experiments, linear LRS[2] is preferable for *ResNet44* on CIFAR-100 and square root LRS is preferable for the others. For TL, we simply double the training epochs for each learning rate.

In the experiments of *ResNet44* on CIFAR-10 and CIFAR-100, we applied GBN, TL (400 epochs), linear LRS or square root LRS in the baselines, so that we can diversify the setups to evaluate our method. In all setups, *SmoothOut* improves generalization on the top of the GBN, TL and LRS, verifying that our method is orthogonal to state-of-the-art solutions. More importantly, the *AdaSmoothOut* variant has the best generalization in all experiments on CIFAR-10 and CIFAR-100, showing the necessity of adaptive noises. Therefore, we choose *AdaSmoothOut* as the representative in ImageNet for faster development.

As *AdaSmoothOut* is one type of regularizations by stochastic model averaging, the regularizations by weight decay and dropout are not adopted in training ImageNet, such that we can reduce the number of hyperparameters. The top-1 accuracy of *AlexNet* in SB training is 56.15% with the batch size of 256, however, in LB training with the batch size of 16384, the accuracy drops to 47.64% if trained by the same epochs. TL indeed can improve the generalization to 54.24%. More importantly, *AdaSmoothOut* improves the accuracy in both cases, *i.e.*, improving 4.89% when TL is not applied and improving 1.27% on the top of TL. Last but not the least, our method also achieve improvement on *ResNet18* on the ImageNet.

[2]Warm up pre-training is not adopted in our experiments for neat evaluation.

Table 5.3: *SmoothOut* without de-noising tested on CIFAR-10.

DNN	Batch size	*SmoothOut*	*SmoothOut* w/o de-noising
C_1	256	81.72%	36.52%
	5000	80.34%	46.05%
$ResNet44$	2048	91.95%	27.57%

Noise strength a is 0.0375 in all experiments.

Table 5.4: Accuracy with and without de-noising tested by $ResNet44$ on CIFAR-10 with the same setting in Table 5.2.

Method	a	Accuracy
Baseline	0	91.02%
SmoothOut	0.0375	91.95%
SmoothOut w/o de-noising	0.0001	91.15%
AdaSmoothOut	0.15	92.63%
AdaSmoothOut w/o de-noising	0.00075	91.54%

At the end, we visualize the sharpness of minima by "filter normalization" visualization [LXTG17], *AdaSmoothOut* indeed converges to a flatter region as shown in Figure 5.5.

5.2.3 Ablation Study

The necessity of de-noising

One of our contributions is the de-noising process. We perform an ablation study by removing the de-noising process to test its necessity. We rerun all CIFAR-10 *SmoothOut* experiments in Table 5.1 and Table 5.2, but without de-noising. We use the same noise strength for comparison. The results are summarized in Table 5.3. Without de-noising, the accuracy significantly drops. The reason is straightforward: strong noises make original parameters and gradients less accurate and deteriorate convergence, but our gradients are exactly the gradients of auxiliary function $\bar{C}(\boldsymbol{w})$ and perturbed parameters are recovered before applying gradients.

For a fair comparison, we further carefully tune the noise strength a in *SmoothOut* and *AdaSmoothOut* "w/o de-noising" to get a near optimal accuracy. More specific, we have to decrease a as SGD is more sensitive to noises when de-noising is not applied. The results are summarized in Table 5.4. Without de-noising, accuracy is lower. Note that the de-noising process is naturally generated by our framework and theory in Section 5.1; without de-noising, it will not fit into our framework and the optimization target will not be the auxiliary function $\bar{C}(\boldsymbol{w})$.

Gaussian noise vs. uniform noise

Another contribution is the generic *SmoothOut* and *AdaSmoothOut* framework, which is agnostic to the type of noises. We majorly used uniform noise for study as it is well motivated, but any type of noises can fit into our framework. As Gaussian noise is broadly used in the literature [NVL+15][Mob16][FAP+17][PHD+17][LL16b][HLS08][KW13], we perform an ablation study here by replacing uniform noise with Gaussian noise, for the purpose of verifying our framework is noise agnostic and answering how performance changes when the noise type alters. The results are in Table 5.5, where, in injecting Gaussian noises, a is the standard derivation. Table 5.5 indicates that

- both uniform and Gaussian noises improve generalization in *SmoothOut* and *AdaSmoothOut*, verifying they are agnostic to noise types;
- uniform noise is superior to Gaussian noise. A intuitive explanation is that Gaussian distribution gives a high probability to average over values near the minimum, and thus has a smaller probability to smooth out sharp minima. However, uniform distribution evenly treats values around the minimum, and can eliminate the minimum when it is sharp. We do not aggressively conclude that uniform noise will always be superior in all settings, but leaving noise selection as an building block when using our framework.

Table 5.5: Comparison between uniform and Gaussian noises tested on CIFAR-10 with the same setting in Table 5.2.

Method	a	Accuracy
Baseline	0	91.02%
SmoothOut (uniform)	0.0375	91.95%
SmoothOut (Gaussian)	0.025	91.53%
AdaSmoothOut (uniform)	0.15	92.63%
AdaSmoothOut (Gaussian)	0.20	92.44%
Uniform noise only	0.0001	91.15%
Gaussian noise only	0.00015	91.38%

- a smaller generalization improvement is observed if only injecting noises into parameters without using our framework (as shown by the "noise only" experiments).

Chapter 6

Conclusion

In this dissertation, I study the efficiency problems in large-scale Deep Neural Networks (DNNs). With efficiency problems alleviated, the community can build larger DNNs and achieve higher accuracy. With my contributions to enable faster training, I wish future researchers can evaluate DNN models faster and thus benefit from a faster model design exploration. With my contributions to large DNN compression, I wish the community can focus on building larger DNNs to pursue accuracy, after which they can use my methods (such as Structured Sparsity Learning and Lower-rank DNNs) as a post-processing to compress their models and deploy to ubiquitous devices.

In this dissertation, I advocate structurally sparse DNNs, which is superior to randomly sparse DNNs. I propose Structured Sparsity Learning and Lower-rank DNNs to learn small but dense DNNs for faster inference. For Structured Sparsity Learning, I foresee many future research directions beyond DNN inference acceleration: (1) DNN training acceleration by gradually training smaller and smaller sparse DNNs [LCZ+19]; (2) Winning tickets discovery motivated by Lottery Ticket Hypothesis [FC18]; (3) Neural Architecture Search by pruning branches and operations in one-shot models [BKZ+18]; (4) Building larger but sparse DNNs for higher accuracy [GRK17]. On the training side, *TernGrad* can reduce communication volume. One future extension is to apply variance reduction methods to *TernGrad* to improve its accuracy.

SmoothOut works well when sharp minima exist in DNNs. However, sharp minima might not exist in some settings. It is still an open research problem when sharp minima are not the source of the Generalization Gap in large-batch training.

Appendices

Appendix A

Convergence Analysis of *TernGrad*

Proof of Theorem 2:

Proof.

$$h_{t+1} - h_t = -2\gamma_t(\boldsymbol{w}_t - \boldsymbol{w}^*)^T \left(s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right) + \gamma_t^2 \left|\left|s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right|\right|^2. \quad \text{(A.1)}$$

We have

$$\begin{aligned}\mathbf{E}\left\{\left(h_{t+1} - h_t\right)|\boldsymbol{X}_t\right\} = &-2\gamma_t(\boldsymbol{w}_t - \boldsymbol{w}^*)^T \mathbf{E}\left\{\left(s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right)|\boldsymbol{X}_t\right\} \\ &+ \gamma_t^2 \mathbf{E}\left\{\left|\left|s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right|\right|^2 \,\middle|\, \boldsymbol{X}_t\right\}.\end{aligned} \quad \text{(A.2)}$$

Eq. (A.2) satisfies based on the fact that γ_t is deterministic, and $\boldsymbol{w}_t$ is also deterministic given $\boldsymbol{X}_t$. According to $\mathbf{E}\left\{s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right\} = \nabla_{\boldsymbol{w}} C(\boldsymbol{w}_t)$,

$$\begin{aligned}&\mathbf{E}\left\{\left(h_{t+1} - h_t\right)|\boldsymbol{X}_t\right\} + 2\gamma_t \cdot (\boldsymbol{w}_t - \boldsymbol{w}^*)^T \cdot \nabla_{\boldsymbol{w}} C(\boldsymbol{w}_t) \\ &= \gamma_t^2 \cdot \mathbf{E}\left\{\left|\left|s_t \cdot sign\left(\boldsymbol{g}_t\right) \circ \boldsymbol{b}_t\right|\right|^2 \,\middle|\, \boldsymbol{X}_t\right\} \\ &= \gamma_t^2 \cdot \mathbf{E}\left\{s_t^2 \left|\left|\boldsymbol{b}_t\right|\right|^2 \,\middle|\, \boldsymbol{w}_t\right\} = \gamma_t^2 \cdot \mathbf{E}\left\{s_t^2 \cdot \mathbf{E}\left\{\left|\left|\boldsymbol{b}_t\right|\right|^2 \middle| \boldsymbol{z}_t, \boldsymbol{w}_t\right\} \,\middle|\, \boldsymbol{w}_t\right\} \\ &= \gamma_t^2 \cdot \mathbf{E}\left\{s_t^2 \cdot \sum_k \mathbf{E}\left\{b_{tk}^2 \middle| \boldsymbol{z}_t, \boldsymbol{w}_t\right\} \,\middle|\, \boldsymbol{w}_t\right\}\end{aligned} \quad \text{(A.3)}$$

Based on the Bernoulli distribution of b_{tk} and Assumption 3, we further have

$$\begin{aligned}&\mathbf{E}\left\{\left(h_{t+1} - h_t\right)|\boldsymbol{X}_t\right\} + 2\gamma_t \cdot (\boldsymbol{w}_t - \boldsymbol{w}^*)^T \cdot \nabla_{\boldsymbol{w}} C(\boldsymbol{w}_t) \\ &= \gamma_t^2 \cdot \mathbf{E}\left\{s_t \left|\left|\boldsymbol{g}_t\right|\right|_1\right\} = \gamma_t^2 \cdot \mathbf{E}\left\{max\left(abs\left(\boldsymbol{g}_t\right)\right) \cdot \left|\left|\boldsymbol{g}_t\right|\right|_1\right\} \\ &\leq A\gamma_t^2 + B\gamma_t^2 \left|\left|\boldsymbol{w}_t - \boldsymbol{w}^*\right|\right|^2 = A\gamma_t^2 + B\gamma_t^2 h_t.\end{aligned} \quad \text{(A.4)}$$

That is

$$\mathbf{E}\left\{\left(h_{t+1}-\left(1+\gamma_t^2 B\right)h_t\right)|\boldsymbol{X}_t\right\} \leq -2\gamma_t(\boldsymbol{w}_t-\boldsymbol{w}^*)^T\nabla_{\boldsymbol{w}}C(\boldsymbol{w}_t)+\gamma_t^2 A, \tag{A.5}$$

which satisfies the condition of Lemma 1 and proves Theorem 2. The proof can be extended to mini-batch SGD by treating $\boldsymbol{z}$ as a mini-batch of observations instead of one observation. □

Appendix B

Performance Model

As mentioned in the main context of our paper, the performance model was developed based on the one initially proposed for CPU-based deep learning systems [YRHC15]. We extended it to model GPU-based deep learning systems in this work. Lightweight profiling is used in the model. We ran all performance tests with distributed TensorFlow on a cluster of 4 machines, each of which has 4 GTX 1080 GPUs and one Mellanox MT27520 InfiniBand network card. Our performance model was successfully validated against the measured results by the server cluster we have.

There are two scaling schemes for distributed training with data parallelism: a) *strong scaling* that spreads the same size problem across multiple workers, and b) *weak scaling* that keeps the size per worker constant when the number of workers increases [SCW+02]. Our performance model supports both scaling models.

We start with strong scaling to illustrate our performance model. According to the definition of strong scaling, here the same size problem is corresponding to the same mini-batch size. In other words, the more workers, the less training samples per worker. Intuitively, more workers bring more computing resources, meanwhile inducing higher communication overhead. The goal is to estimate the throughput of a system that uses j machines with i GPUs per machine and mini-batch size of

K[1]. Note the total number of workers equals to the total number of GPUs on all machines, i.e., $N = i * j$. We need to distinguish workers within a machine and across machines due to their different communication patterns. Next, we illustrate how to accurately model the impacts in communication and computation to capture both the benefits and overheads.

Communication. For GPUs within a machine, first, the gradient $\boldsymbol{g}$ computed at each GPU needs to be accumulated together. Here we assume all-reduce communication model, that is, each GPU communicates with its neighbor until all gradient $\boldsymbol{g}$ is accumulated into a single GPU. The communication complexity for i GPUs is $log_2 i$. The GPU with accumulated gradient then sends the accumulated gradient to CPU for further processing. Note for each communication (either GPU-to-GPU or GPU-to-CPU), the communication data size is the same, i.e., $|\boldsymbol{g}|$. Assume that within a machine, the communication bandwidth between GPUs is C_{gwd}[2] and the communication bandwidth between CPU and GPU is C_{cwd}, then the communication overhead within a machine can be computed as $\frac{|\boldsymbol{g}|}{C_{gwd}} * log_2 i + \frac{|\boldsymbol{g}|}{C_{cwd}}$. We successfully used NCCL benchmark to validate our model. For communication between machines, we also assume all-reduce communication model, so the communication time between machines are: $(C_{ncost} + \frac{|\boldsymbol{g}|}{C_{nwd}}) * log_2 j$, where C_{ncost} is the network latency and C_{nwd} is the network bandwidth. So the total communication time is $T_{comm}(i, j, K, |\boldsymbol{g}|) = \frac{|\boldsymbol{g}|}{C_{gwd}} * log_2 i + \frac{|\boldsymbol{g}|}{C_{cwd}} + (C_{ncost} + \frac{|\boldsymbol{g}|}{C_{nwd}}) * log_2 j$. We successfully used OSU Allreduce benchmark to validate this model.

Computation. To estimate computation time, we rely on profiling the time for training a mini-batch of totally K images on a machine with a single CPU and a single GPU. We define this profiled time as $T(1, 1, K, |\boldsymbol{g}|)$. In strong scaling, each

[1]For ease of the discussion, we assume symmetric system architecture. The performance model can be easily extended to support heterogeneous system architecture.

[2]For ease of the discussion, we assume GPU-to-GPU communication has *Dedicated Bandwidth*.

work only trains $\frac{K}{N}$ samples, so the total computation time is $T_{comp}(i,j,K,|\boldsymbol{g}|) = (T(1,1,K,|\boldsymbol{g}|) - \frac{|\boldsymbol{g}|}{C_{cwd}}) * \frac{1}{N}$, where $\frac{|\boldsymbol{g}|}{C_{cwd}}$ is the communication time (between GPU and CPU) included in when we profile $T(1,1,K,|\boldsymbol{g}|)$.

Therefore, the time to train a mini-batch of K samples is:

$$\begin{aligned} T_{strong}(i,j,K,|\boldsymbol{g}|) &= T_{comp}(i,j,K,|\boldsymbol{g}|) + T_{comm}(i,j,K,|\boldsymbol{g}|) \\ &= (T(1,1,K,|\boldsymbol{g}|) - \frac{|\boldsymbol{g}|}{C_{cwd}}) * \frac{1}{N} \\ &\quad + \frac{|\boldsymbol{g}|}{C_{gwd}} * log_2 i + \frac{|\boldsymbol{g}|}{C_{cwd}} + (C_{ncost} + \frac{|\boldsymbol{g}|}{C_{nwd}}) * log_2 j. \end{aligned} \tag{B.1}$$

The throughput of strong scaling is:

$$Tput_{strong}(i,j,K,|\boldsymbol{g}|) = \frac{K}{T_{strong}(i,j,K,|\boldsymbol{g}|)}. \tag{B.2}$$

For weak scaling, the difference is that each worker always trains K samples. So the mini-batch size becomes $N * K$. In the interest of space, we do not present the detailed reasoning here. Basically, it follows the same logic for developing the performance model of strong scaling. We can compute the time to train a mini-batch of $N * K$ samples as follows:

$$\begin{aligned} T_{weak}(i,j,K,|\boldsymbol{g}|) &= T_{comp}(i,j,K,|\boldsymbol{g}|) + T_{comm}(i,j,K,|\boldsymbol{g}|) \\ &= T(1,1,K,|\boldsymbol{g}|) - \frac{|\boldsymbol{g}|}{C_{cwd}} + \frac{|\boldsymbol{g}|}{C_{gwd}} * log_2 i + \frac{|\boldsymbol{g}|}{C_{cwd}} + (C_{ncost} + \frac{|\boldsymbol{g}|}{C_{nwd}}) * log_2 j \\ &= T(1,1,K,|\boldsymbol{g}|) + \frac{|\boldsymbol{g}|}{C_{gwd}} * log_2 i + (C_{ncost} + \frac{|\boldsymbol{g}|}{C_{nwd}}) * log_2 j. \end{aligned} \tag{B.3}$$

So the throughput of weak scaling is:

$$Tput_{weak}(i,j,K,|\boldsymbol{g}|) = \frac{N * K}{T_{weak}(i,j,K,|\boldsymbol{g}|)}. \tag{B.4}$$

Appendix C

Proof of Theorem 3 and Theorem 4

Proof of Theorem 3:

Proof. As $\mathcal{D}(\boldsymbol{w}, a)$ is defined as a box centering at $\boldsymbol{w}$ with size $2a$, *i.e.*,

$$\mathcal{D}(\boldsymbol{w}, a) = \{\boldsymbol{w}' \in \mathbb{R}^m : |(\boldsymbol{w}' - \boldsymbol{w})_i| \leq a, \forall i \in \{1...m\}\}, \tag{C.1}$$

then, under *Uniform Perturbation*,

$$\begin{aligned} \bar{C}(\boldsymbol{w}) &= \mathbf{E}\left\{\tilde{C}(\boldsymbol{w}; \boldsymbol{\Theta})\right\} = \mathbf{E}\left\{C(\boldsymbol{w} + \boldsymbol{\Theta})\right\} \\ &= \frac{1}{(2a)^m} \int_{\mathcal{D}(\boldsymbol{w},a)} \cdots \int C(\boldsymbol{w}') dw'_1 \dots dw'_m \end{aligned} \tag{C.2}$$

$$\begin{aligned} &\frac{\partial \bar{C}(\boldsymbol{w})}{\partial w_i} = \frac{1}{(2a)^m} \cdot \\ &\int_{\mathcal{D}(\boldsymbol{w}_{\setminus i},a)} \cdots \int \left(C(\boldsymbol{w}')|_{w'_i = w_i + a} - C(\boldsymbol{w}')|_{w'_i = w_i - a}\right) d\boldsymbol{w}'_{\setminus i}, \end{aligned} \tag{C.3}$$

where

$$\boldsymbol{w}_{\setminus i} \triangleq [w_1, \cdots, w_{i-1}, w_{i+1}, \cdots, w_m]^T \in \mathbb{R}^{m-1} \tag{C.4}$$

and

$$d\boldsymbol{w}'_{\setminus i} \triangleq dw'_1 \dots dw'_{i-1} dw'_{i+1} \dots dw'_m. \tag{C.5}$$

When $C(\boldsymbol{w})$ is symmetric about $\boldsymbol{w}_f$ in $\mathcal{D}(\boldsymbol{w}_f, \tau)$ such that, $\forall i$, a cut along $w_i = (\boldsymbol{w}_f)_i + a$ and a cut along $w_i = (\boldsymbol{w}_f)_i - a$ get the same function in the subspace $\boldsymbol{w}_{\setminus i}$, then $\nabla \bar{C}(\boldsymbol{w}_f) = \mathbf{0}$; that is, the *Flat Constraint* satisfies with $\varphi = 0$. □

The optimal φ and τ are determined by the symmetry of the flat region. φ may be relaxed to a larger value when the symmetry is broken; however, within a flat region, a larger φ may only slightly increase $C(\boldsymbol{w}^*)$.

Proof of Theorem 4:

Proof. Suppose $C^{(s)}_{\varepsilon'}$ is the maximum value near the sharp minimum, *i.e.*,

$$C^{(s)}_{\varepsilon'} = \max_{\mathcal{D}(\boldsymbol{w}_s, \varepsilon')} \left(C(\boldsymbol{w}) \right), \tag{C.6}$$

as $C(\boldsymbol{w})$ is strictly monotonic in $\mathcal{D}(\boldsymbol{w}_s, b)$, we have, $\forall \varepsilon' < a < b$,

$$\min_{\mathcal{D}(\boldsymbol{w}_s, a) \setminus \mathcal{D}(\boldsymbol{w}_s, \varepsilon')} \left(C(\boldsymbol{w}) \right) > C^{(s)}_{\varepsilon'}, \tag{C.7}$$

where $\mathcal{D}(\boldsymbol{w}_s, a) \setminus \mathcal{D}(\boldsymbol{w}_s, \varepsilon')$ is a *Set Difference*, notating a domain within $\mathcal{D}(\boldsymbol{w}_s, a)$ but outside of $\mathcal{D}(\boldsymbol{w}_s, \varepsilon')$.

Then, follow the proof of Theorem 3, we have

$$\begin{aligned}
\min_{\mathcal{D}(\boldsymbol{w}_s, \varepsilon)|_{\varepsilon < b}} \left(\bar{C}(\boldsymbol{w}) \right) &= \frac{1}{(2a)^m} \int \cdots \int_{\mathcal{D}(\boldsymbol{w}_s, a)} C(\boldsymbol{w}') dw'_1 \dots dw'_m \\
&\geq \frac{1}{(2a)^m} \cdot \left((2a)^m C^{(s)}_{\varepsilon'} - (2\varepsilon')^m \left(C^{(s)}_{\varepsilon'} - C(\boldsymbol{w}_s) \right) \right) \\
&= \left(1 - \left(\frac{\varepsilon'}{a} \right)^m \right) C^{(s)}_{\varepsilon'} + \left(\frac{\varepsilon'}{a} \right)^m \cdot C(\boldsymbol{w}_s).
\end{aligned} \tag{C.8}$$

Because of

$$\lim_{m \to \infty} \left(\left(1 - \left(\frac{\varepsilon'}{a} \right)^m \right) C^{(s)}_{\varepsilon'} + \left(\frac{\varepsilon'}{a} \right)^m \cdot C(\boldsymbol{w}_s) \right) = C^{(s)}_{\varepsilon'}, \tag{C.9}$$

in high dimensional models (like deep neural networks), we can find $\varepsilon \to \varepsilon'^{-} \to a^{-}$ (left limit) to satisfy

$$\min_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)} \left(\bar{C}(\boldsymbol{w})\right) > C_{\varepsilon}^{(s)} \triangleq \max_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)} \left(C(\boldsymbol{w})\right). \tag{C.10}$$

In the flat region,

$$\begin{aligned} \min_{\mathcal{D}(\boldsymbol{w}_f,\tau)} \left(\bar{C}(\boldsymbol{w})\right) &= \frac{1}{(2a)^m} \int \cdots \int_{\mathcal{D}(\boldsymbol{w}_f,a)} C(\boldsymbol{w}') dw'_1 \dots dw'_m \\ &< \frac{1}{(2a)^m} \cdot \left((2a)^m \cdot \max_{\mathcal{D}(\boldsymbol{w}_f,a)} \left(C(\boldsymbol{w})\right) \right) \\ &= \max_{\mathcal{D}(\boldsymbol{w}_f,a)} \left(C(\boldsymbol{w})\right) \triangleq C_a^{(f)} \end{aligned} \tag{C.11}$$

Assuming $C(\boldsymbol{w}_s) \approx C(\boldsymbol{w}_f)$, as a grows, $C_{\varepsilon}^{(s)}|_{\varepsilon \to a^-}$ increases fast in the sharp region while $C_a^{(f)}$ increases slowly in the flat region; therefore, $\exists a$ such that

$$C_{\varepsilon}^{(s)} > C_a^{(f)}. \tag{C.12}$$

According to Inequality (C.10)(C.11)(C.12),

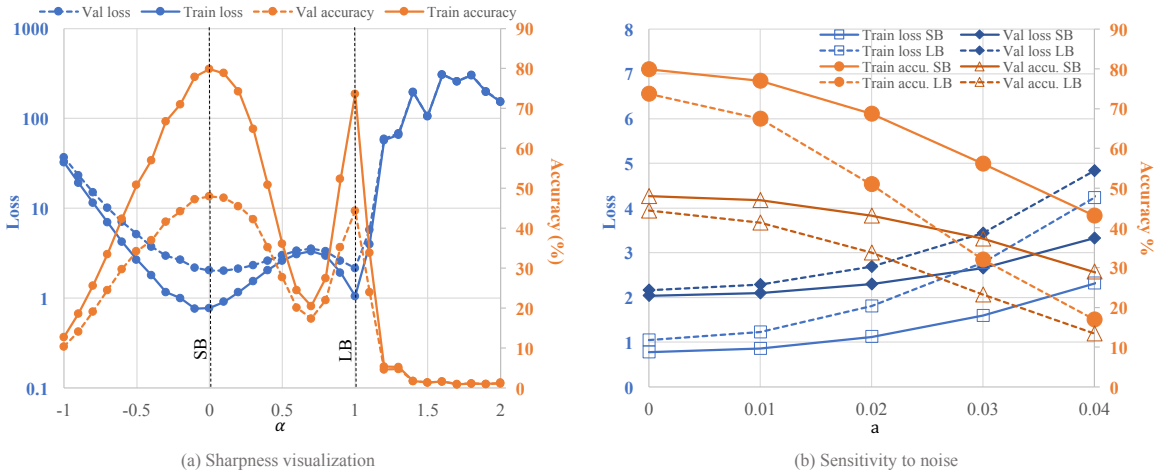
$$\begin{aligned} \min_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)} \left(\bar{C}(\boldsymbol{w})\right) &> \max_{\mathcal{D}(\boldsymbol{w}_s,\varepsilon)} \left(C(\boldsymbol{w})\right) \\ &> \max_{\mathcal{D}(\boldsymbol{w}_f,a)} \left(C(\boldsymbol{w})\right) \\ &> \min_{\mathcal{D}(\boldsymbol{w}_f,\tau)} \left(\bar{C}(\boldsymbol{w})\right) \end{aligned} \tag{C.13}$$

which satisfies the *Sharp Constraint*. □

Appendix D

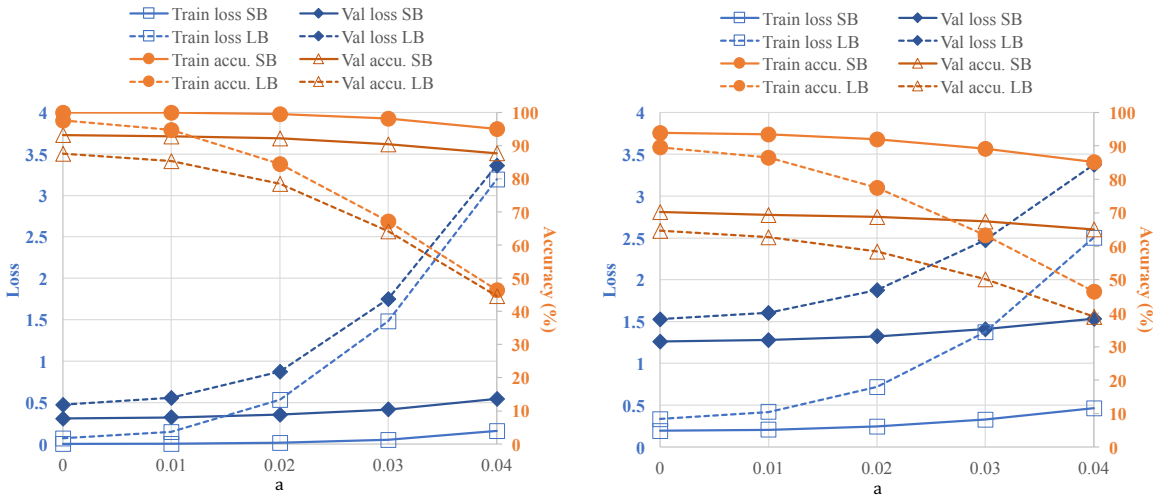
Sensitivity Analyses and Sharpness Visualization

We provide more sensitivity analyses in Figure D.1 and Figure D.2 as tested on different DNNs and datasets. More sharpness comparison between baseline and *SmoothOut* is visualized in Figure D.3 and Figure D.4.



(a) Sharpness visualization (b) Sensitivity to noise

Figure D.1: Notation:"SB": Small Batch (256); "LB": Large Batch (5000); "accu.": accuracy. (a) loss and accuracy vs. α, which controls $\boldsymbol{w}$ along the direction from SB minimum ($\boldsymbol{w}_f$) to LB minimum ($\boldsymbol{w}_s$); (b) loss and accuracy under influence of different strengths of noise. Dataset: CIFAR-100. Network: C_3. The optimizer is Adam with 0.001 initial learning rate.



(a) Sensitivity to noise (ResNet-44 on Cifar-10) (b) Sensitivity to noise (ResNet-44 on Cifar-100)

Figure D.2: Loss and accuracy of ResNet-44 under influence of different strengths of noise on (a) CIFAR-10 and (b) CIFAR-100. The optimizer is SGD with momentum 0.9. Notation:"SB": Small Batch (128); "LB": Large Batch (2048 for CIFAR-10 and 1024 for CIFAR-100); "accu.": accuracy.

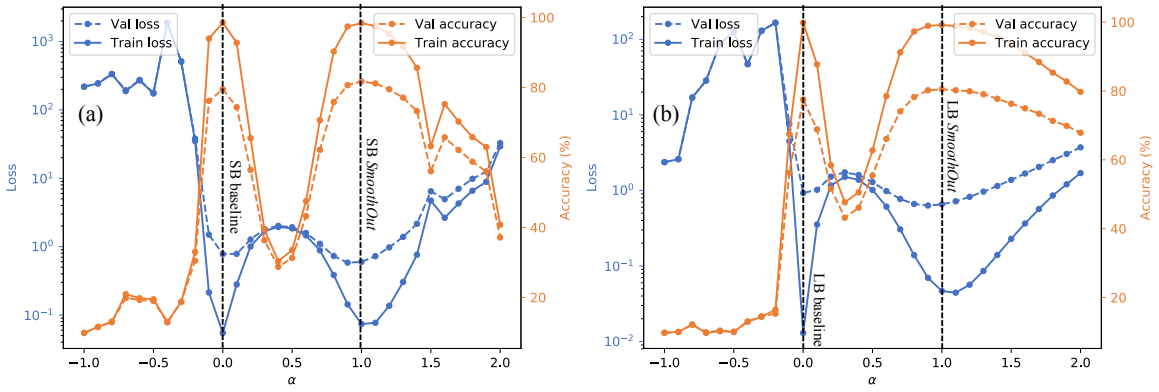


Figure D.3: Sharpness of baseline and *SmoothOut* in (a) "SB" training and (b) "LB" training of C_1.

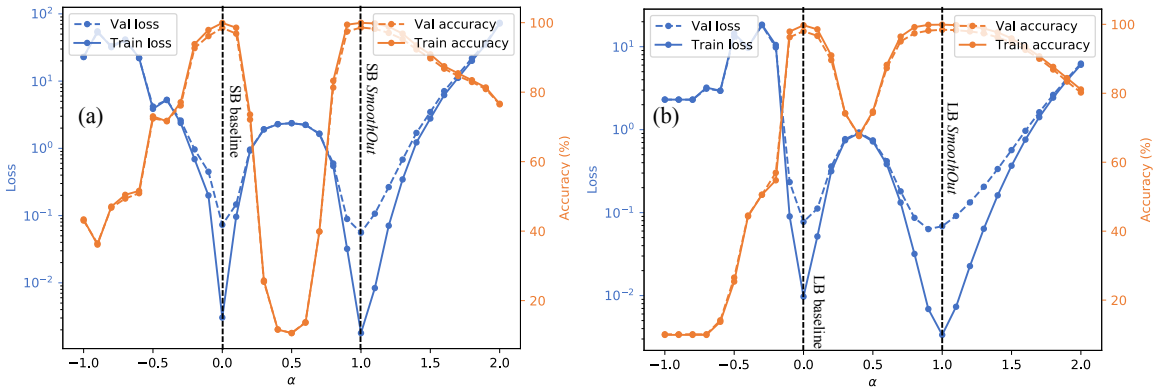


Figure D.4: Sharpness of baseline and *SmoothOut* in (a) "SB" training and (b) "LB" training of F_1.

Bibliography

[AAA+16] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, et al. Deep speech 2: End-to-end speech recognition in english and mandarin. In *International Conference on Machine Learning*, pages 173–182, 2016.

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint:1603.04467*, 2016.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1707–1718, 2017.

[AH17] Alham Fikri Aji and Kenneth Heafield. Sparse communication for distributed gradient descent. *arXiv preprint:1704.05021*, 2017.

[AS16] Jose M Alvarez and Mathieu Salzmann. Learning the number of neurons in deep networks. In *Advances in Neural Information Processing Systems*, 2016.

[ASF17] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[Bau15] Christian Bauckhage. k-means clustering is matrix factorization. *arXiv:1512.07548*, 2015.

[BG90] Jean-Philippe Bouchaud and Antoine Georges. Anomalous diffusion in disordered media: statistical mechanisms, models and physical applications. *Physics reports*, 195(4-5):127–293, 1990.

[BJMO12] Francis Bach, Rodolphe Jenatton, Julien Mairal, and Guillaume Obozinski. Structured sparsity through convex optimization. *Statistical Science*, 27(4):450–468, 2012.

[BKBG11] Joseph K Bradley, Aapo Kyrola, Danny Bickson, and Carlos Guestrin. Parallel coordinate descent for l1-regularized loss minimization. *arXiv preprint arXiv:1105.5379*, 2011.

[BKZ+18] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 549–558, 2018.

[BLNZ95] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.

[BMXS16] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv:1611.01576*, 2016.

[Bot98] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[CCSL17] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, and Yann LeCun. Entropy-sgd: Biasing gradient descent into wide valleys. In *International Conference on Learning Representations*, 2017.

[CGK+17] Corinna Cortes, Xavi Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 874–883, 2017.

[CHW+13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning*, pages 1337–1345, 2013.

[CLL+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint:1512.01274*, 2015.

[CSAK14] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582, 2014.

[CVMG+14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv:1406.1078*, 2014.

[CWV+14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[CWW+17] Hsin-Pai Cheng, Wei Wen, Chunpeng Wu, Sicheng Li, Hai Helen Li, and Yiran Chen. Understanding the design of ibm neurosynaptic system and its tradeoffs: a user perspective. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 139–144. European Design and Automation Association, 2017.

[DCM+12] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231. 2012.

[DDS+09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.

[DPBB17] Laurent Dinh, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. Sharp minima can generalize for deep nets. In *International Conference on Machine Learning*, pages 1019–1028, 2017.

[DSC+16] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*, pages 2024–2033, 2016.

[DSD+13] Misha Denil, Babak Shakibi, Laurent Dinh, Marc' Aurelio Ranzato, and Nando de Freitas. Predicting parameters in deep learning. In *Advances in Neural Information Processing Systems*, pages 2148–2156. 2013.

[DZB+14] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems*, pages 1269–1277. 2014.

[FAP+17] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, et al. Noisy networks for exploration. *arXiv:1706.10295*, 2017.

[FC18] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.

[FD15] Jiashi Feng and Trevor Darrell. Learning the structure of deep convolutional networks. In *The IEEE International Conference on Computer Vision (ICCV)*, 2015.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *ICML*, pages 1737–1746, 2015.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[GDG+17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[GG16] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[Gib17] Andrew Gibiansky. Bringing hpc techniques to deep learning, 2017.

[GK09] Rahul Garg and Rohit Khandekar. Gradient descent with sparsification: an iterative algorithm for sparse recovery with restricted isometry property. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 337–344. ACM, 2009.

[GRK17] Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv preprint arXiv:1711.09224*, 2017.

[GVS14] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *arXiv:1412.6544*, 2014.

[GYC16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, 2016.

[HCC+13] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing.

More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[HCS+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016.

[HHS17] 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 1729–1739, 2017.

[HLS08] Kevin Ho, Chi-sing Leung, and John Sum. On weight-noise-injection training. In *International Conference on Neural Information Processing*, pages 919–926. Springer, 2008.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149*, 2015.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems (NIPS)*. 2015.

[HS97a] Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1–42, 1997.

[HS97b] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[HvC] GE Hinton and Drew van Camp. Keeping neural networks simple by minimising the description length of weights. 1993. In *Proceedings of COLT-93*, pages 5–13.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[HWW+16] Miao Hu, Yandan Wang, Wei Wen, Yu Wang, and Hai Li. Leveraging stochastic memristor devices in neuromorphic hardware systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(2):235–246, 2016.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *International Conference on Computer Vision (ICCV)*, 2015.

[HZRS16] 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.

[IPG+18] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv:1803.05407*, 2018.

[IRS+15] Yani Ioannou, Duncan P. Robertson, Jamie Shotton, Roberto Cipolla, and Antonio Criminisi. Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*, 2015.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[IWLC19] Nathan Inkawhich, Wei Wen, Hai Helen Li, and Yiran Chen. Feature space perturbations yield more transferable adversarial examples. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7066–7074, 2019.

[JKA+18] Stanisław Jastrzębski, Zachary Kenton, Devansh Arpit, Nicolas Ballas, Asja Fischer, Yoshua Bengio, and Amos Storkey. Finding flatter minima with sgd. 2018.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[JSH+18] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv:1807.11205*, 2018.

[JST+14] Martin Jaggi, Virginia Smith, Martin Takác, Jonathan Terhorst, Sanjay Krishnan, Thomas Hofmann, and Michael I Jordan. Communication-efficient distributed dual coordinate ascent. In *Advances in Neural Information Processing Systems*, pages 3068–3076, 2014.

[JVS+16] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[JVZ14] M. Jaderberg, A. Vedaldi, and A. Zisserman. Speeding up convolutional neural networks with low rank expansions. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2014.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint:1412.6980*, 2014.

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[KMN+17] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2017.

[KPY+15] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv:1511.06530*, 2015.

[Kri14] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[KSH12] 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.

[KW13] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv:1312.6114*, 2013.

[KX10] Seyoung Kim and Eric P Xing. Tree-guided group lasso for multi-task regression with structured sparsity. In *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[KXS18] Shankar Krishnan, Ying Xiao, and Rif A Saurous. Neumann optimizer: A practical optimization algorithm for deep neural networks. In *International Conference on Learning Representations*, 2018.

[LAP+14] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pages 583–598, 2014.

[LASY14] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[Lav15] Andrew Lavin. Fast algorithms for convolutional neural networks. *arXiv:1509.09308*, 2015.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[LCMB15] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv:1510.03009*, 2015.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[LCZ+19] Sangkug Lym, Esha Choukse, Siavash Zangeneh, Wei Wen, Sujay Sanghavi, and Mattan Erez. Prunetrain: Fast neural network training by dynamic sparse model reconfiguration. 2019.

[LDS+89] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pages 598–605, 1989.

[LGR+14] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv:1412.6553*, 2014.

[LHM+18] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *International Conference on Learning Representations*, 2018.

[Li17] Mu Li. *Scaling Distributed Machine Learning with System and Algorithm Co-design*. PhD thesis, Carnegie Mellon University, 2017.

[LJ17] Lihua Lei and Michael Jordan. Less than a single pass: Stochastically controlled stochastic gradient. In *Artificial Intelligence and Statistics*, pages 148–156, 2017.

[LKD+17] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. In *International Conference on Learning Representations (ICLR)*, 2017.

[LL16a] Vadim Lebedev and Victor Lempitsky. Fast convnets using group-wise brain damage. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[LL16b] Yinan Li and Fang Liu. Whiteout: Gaussian adaptive noise regularization in feedforward neural networks. *arXiv:1612.01490*, 2016.

[LSS16] Zhiyun Lu, Vikas Sindhwani, and Tara N Sainath. Learning compact recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 5960–5964. IEEE, 2016.

[LUW17] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. *arXiv:1705.08665*, 2017.

[LWF+15] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[LWM+18] Bing Li, Wei Wen, Jiachen Mao, Sicheng Li, Yiran Chen, and Hai Helen Li. Running sparse and low-precision neural network: When algorithm meets hardware. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 534–539. IEEE, 2018.

[LWQ+18] Xiaoxiao Liu, Wei Wen, Xuehai Qian, Hai Li, and Yiran Chen. Neu-noc: A high-efficient interconnection network for accelerated neuromorphic systems. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 141–146. IEEE, 2018.

[LWW+17] Sicheng Li, Wei Wen, Yu Wang, Song Han, Yiran Chen, and Hai Li. An fpga design framework for cnn sparsification and acceleration. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 28–28. IEEE, 2017.

[LXTG17] Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets. *arXiv preprint arXiv:1712.09913*, 2017.

[Mac92] David JC MacKay. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472, 1992.

[MHB17] Stephan Mandt, Matthew D Hoffman, and David M Blei. Stochastic gradient descent as approximate bayesian inference. *The Journal of Machine Learning Research*, 18(1):4873–4907, 2017.

[MMS93] Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

[MNSJ15] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. Sparknet: Training deep networks in spark. *arXiv preprint:1511.06051*, 2015.

[Mob16] Hossein Mobahi. Training recurrent neural networks by diffusion. *arXiv:1601.04114*, 2016.

[MTK+17] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. In *International Conference on Learning Representations (ICLR)*, 2017.

[MYW+17] Jiachen Mao, Zhongda Yang, Wei Wen, Chunpeng Wu, Linghao Song, Kent W Nixon, Xiang Chen, Hai Li, and Yiran Chen. Mednn: A distributed mobile system with enhanced partition and deployment for large-scale dnns. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 751–756. IEEE Press, 2017.

[NDSE17] Sharan Narang, Gregory Diamos, Shubho Sengupta, and Erich Elsen. Exploring sparsity in recurrent neural networks. *arXiv:1704.05119*, 2017.

[Nea12] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[NH10] 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.

[NVL+15] Arvind Neelakantan, Luke Vilnis, Quoc V Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding gradient noise improves learning for very deep networks. *arXiv preprint:1511.06807*, 2015.

[Ola15] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[OLZ+16] Joachim Ott, Zhouhan Lin, Ying Zhang, Shih-Chii Liu, and Yoshua Bengio. Recurrent neural networks with limited numerical precision. *arXiv:1608.06902*, 2016.

[PABM16] Rohit Prabhavalkar, Ouais Alsharif, Antoine Bruguier, and Lan McGraw. On the compression of recurrent neural networks with an application to lvcsr acoustic modeling for embedded speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*, pages 5970–5974. IEEE, 2016.

[PC17] George Philipp and Jaime G Carbonell. Nonparametric neural networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[PCM+17] Xinghao Pan, Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint:1702.05800*, 2017.

[PHD+17] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *arXiv:1706.01905*, 2017.

[PLW+16] Jongsoo Park, Sheng Li, Wei Wen, Hai Li, Yiran Chen, and Pradeep Dubey. Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *arXiv:1608.01409*, 2016.

[PLW+17] J Park, S Li, W Wen, PTP Tang, H Li, Y Chen, and P Dubey. Faster cnns with direct sparse convolutions and guided pruning. In *International Conference on Learning Representations (ICLR)*, 2017.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[RHY+17] Jeff Rasley, Yuxiong He, Feng Yan, Olatunji Ruwase, and Rodrigo Fonseca. HyperDrive: Exploring Hyperparameters with POP Scheduling. In *Proceedings of the 18th International Middleware Conference*, Middleware '17. ACM, 2017.

[Ris78] Jorma Rissanen. Modeling by shortest data description. *Automatica*, 14(5):465–471, 1978.

[RORF16] 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.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[RZLL16] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. *arXiv:1606.05250*, 2016.

[SCW+02] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 21–21. IEEE, 2002.

[SFD+14] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Interspeech*, pages 1058–1062, 2014.

[SGS15] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[SHK+14] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[SKFH17] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *International Conference on Learning Representations (ICLR)*, 2017.

[SKL18] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[SL17] Samuel L Smith and Quoc V Le. A bayesian perspective on generalization and stochastic gradient descent. In *Proceedings of Second workshop on Bayesian Deep Learning (NIPS 2017)*, 2017.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2015.

[SP97] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[SYMK16] Ananda Theertha Suresh, Felix X Yu, H Brendan McMahan, and Sanjiv Kumar. Distributed mean estimation with limited communication. *arXiv:1611.00429*, 2016.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TXWE15] Cheng Tai, Tong Xiao, Xiaogang Wang, and Weinan E. Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*, 2015.

[WB68] C. S. Wallace and D. M. Boulton. An information measure for classification. *The Computer Journal*, 11(2):185–194, 1968.

[WC16] Peisong Wang and Jian Cheng. Accelerating convolutional neural networks for mobile applications. In *Proceedings of the 2016 ACM on Multimedia Conference*, 2016.

[WHR+17] Wei Wen, Yuxiong He, Samyam Rajbhandari, Wenhan Wang, Fang Liu, Bin Hu, Yiran Chen, and Hai Li. Learning intrinsic sparse structures within long short-term memory. *arXiv:1709.05027*, 2017.

[WT11] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 681–688, 2011.

[WWA+17] Chunpeng Wu, Wei Wen, Tariq Afzal, Yongmei Zhang, Yiran Chen, and Hai Li. A compact dnn: Approaching googlenet-level accuracy of classification and domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017.

[WWH+15] Wei Wen, Chi-Ruo Wu, Xiaofang Hu, Beiye Liu, Tsung-Yi Ho, Xin Li, and Yiran Chen. An eda framework for large scale hybrid neuromorphic computing systems. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[WWHC16] Chi-Ruo Wu, Wei Wen, Tsung-Yi Ho, and Yiran Chen. Thermal optimization for memristor-based hybrid neuromorphic computing systems. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 274–279. IEEE, 2016.

[WWL+17] Yandan Wang, Wei Wen, Beiye Liu, Donald Chiarulli, and Hai Li. Group scissor: Scaling neuromorphic computing design to large neural networks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.

[WWSL17] Yandan Wang, Wei Wen, Linghao Song, and Hai Helen Li. Classification accuracy improvement for neuromorphic computing systems with one-level precision synapses. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 776–781. IEEE, 2017.

[WWW+16a] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, 2016.

[WWW+16b] Wei Wen, Chunpeng Wu, Yandan Wang, Kent Nixon, Qing Wu, Mark Barnell, Hai Li, and Yiran Chen. A new learning method for inference accuracy, core occupation, and performance co-optimization on truenorth chip. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[WXW+17] Wei Wen, Cong Xu, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Coordinating filters for faster deep neural networks. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2017.

[WXY+17] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in Neural Information Processing Systems*, pages 1508–1518, 2017.

[XHD+15] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[YGG17] Yang You, Igor Gitman, and Boris Ginsburg. Scaling sgd batch size to 32k for imagenet training. *arXiv preprint arXiv:1708.03888*, 2017.

[YH17] Jaehong Yoon and Sung Ju Hwang. Combined group and exclusive sparsity for deep neural networks. In *International Conference on Machine Learning*, pages 3958–3966, 2017.

[YL06] Ming Yuan and Yi Lin. Model selection and estimation in regression with grouped variables. *Journal of the Royal Statistical Society. Series B (Statistical Methodology)*, 68(1):49–67, 2006.

[YRG+13] Xiao Yu, Xiang Ren, Quanquan Gu, Yizhou Sun, and Jiawei Han. Collaborative filtering with entity similarity regularization in heterogeneous information networks. *Proceeding of IJCAI HINA workshop*, 2013.

[YRHC15] Feng Yan, Olatunji Ruwase, Yuxiong He, and Trishul M. Chilimbi. Performance modeling and scalability optimization of distributed deep learning systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pages 1355–1364, 2015.

[YZW+13] Shuyuan Yang, Linfang Zhao, Min Wang, Yueyuan Zhang, and Licheng Jiao. Dictionary learning and similarity regularization based image noise reduction. *Journal of Visual Communication and Image Representation*, 24(2):181–186, 2013.

[ZBH+16] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[ZCL15] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[ZF14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision (ECCV)*, 2014.

[ZGLL16] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 2350–2356. AAAI Press, 2016.

[ZL17] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

[ZSKS17] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 4189–4198, 2017.

[ZSV14] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv:1409.2329*, 2014.

[ZVSL17] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. *arXiv:1707.07012*, 2017.

[ZWD+19] Jingchi Zhang, Wei Wen, Michael Deisher, Hsin-Pai Cheng, Hai Li, and Yiran Chen. Learning efficient sparse structures in speech recognition. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2717–2721. IEEE, 2019.

[ZWLS10] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603, 2010.

[ZWN+16] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[ZWO+14] Xiaoming Zheng, Yan Wang, Mehmet Orgun, Youliang Zhong, and Guanfeng Liu. Trust prediction with propagation and similarity regularization. In *AAAI Conference on Artificial Intelligence*, 2014.

[ZZHS16] X. Zhang, J. Zou, K. He, and J. Sun. Accelerating very deep convolutional networks for classification and detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(10):1943–1955, Oct 2016.

Biography

Wei Wen is a Ph.D. candidate at Electrical and Computer Engineering in Duke University. His research is Machine Learning with recent focuses on efficient deep learning, scalable deep learning, and automated machine learning. He spent the first three years of Ph.D. study at University of Pittsburgh, and then moved to Duke University with his advisors Hai Li and Yiran Chen. He received Master's degree and Bachelor's degree from the Electronic Information Engineering School in Beihang University, Beijing, China. During his Ph.D. study, he worked as Research Intern at Google Brain, Facebook AI, Microsoft Research and HP Labs. Some of his methods have been deployed into AI productions, such as Facebook AI Infra and PyTorch/Caffe2.

He has co-authored one Best Paper Award in ASP-DAC 2017 [WWSL17], and authored two Best Paper Nominations in DAC 2015 [WWH+15]and DAC 2016 [WWW+16b]. He has authored or co-authored thirty papers in top-tier conferences and journals, including NeurIPS [WWW+16a, WXY+17], ICLR [WHR+17, PLW+17], ICCV [WXW+17], CVPR [WWA+17, IWLC19], ICASSP [ZWD+19], DAC [WWH+15, WWW+16b, WWL+17], FCCM [LWW+17], ICCAD [MYW+17], ASP-DAC [WWSL17, LWQ+18, LWM+18, WWHC16], DATE [CWW+17], IEEE Journal on Emerging and Selected Topics in Circuits and Systems [HWW+16], *etc*. He is serving as a paper reviewer of NeurIPS, ICML, ICLR, CVPR, ICCV, TPAMI, IJCV, TNNLS, TCAD, Neurocomputing, TCBB, ICME, *etc*.