Georgia State University

ScholarWorks @ Georgia State University

Computer Science Dissertations | Department of Computer Science

8-7-2024

# Towards Pruning and Parameter Efficient Fine-tuning of Deep Neural Networks

Yang Li

Follow this and additional works at: https://scholarworks.gsu.edu/cs_diss

Recommended Citation

Li, Yang, "Towards Pruning and Parameter Efficient Fine-tuning of Deep Neural Networks." Dissertation, Georgia State University, 2024.

doi: https://doi.org/10.57709/37370410

This Dissertation is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Dissertations by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.

Towards Pruning and Parameter Efficient Fine-tuning of Deep Neural Networks

by

Yang Li

Under the Direction of Shihao Ji, Ph.D.

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2024

## ABSTRACT

Deep Neural Networks (DNNs) have achieved significant success across various applications. However, the increasing number of parameters in state-of-the-art architectures presents challenges such as overfitting and high computational costs. Additionally, with the rising adoption of large language models (LLMs) and the growing demand for per-user or per-task model customization, parameter-efficient fine-tuning has become crucial. Consequently, the exploration of neural network efficiency has emerged as a vibrant and dynamic research area, focusing on optimizing model performance while minimizing resource usage.

This dissertation explores neural network efficiency in two directions: pruning and parameter-efficient fine-tuning. Three novel pruning algorithms—$L_0$-ARM, NPN, and Dep-$L_0$—are introduced. $L_0$-ARM enhances $L_0$-based pruning with the Augment-Reinforce-Merge gradient estimator, demonstrating superior performance in sparsifying networks. Building on $L_0$-ARM, the Neural Plasticity Network (NPN) enables both network pruning and expansion within the same framework. To address the inconsistencies of $L_0$-based methods on large-scale tasks, Dep-$L_0$ introduces dependency-enabled $L_0$ regularization, leveraging dependency modeling for binary gates.

In the realm of parameter-efficient fine-tuning (PEFT), this dissertation introduces VB-LoRA, which implements a novel "divide-and-share" paradigm to address the limitations of low-rank decomposition across matrix dimensions, modules, and layers by globally sharing parameters through a *vector bank*. The proposed VB-LoRA method composites *all* low-rank matrices of LoRA from a shared *vector bank* using a differentiable top-$k$ admixture module. This approach enables VB-LoRA to achieve extreme parameter efficiency while maintaining performance that is comparable to or better than state-of-the-art PEFT methods.



INDEX WORDS: Efficient neural networks, Pruning method, $L_0$ regularization, Parameter-efficient fine-tuning, Low-rank adaptation, Top-k admixture module

Copyright by
Yang Li
2024

Towards Pruning and Parameter Efficient Fine-tuning of Deep Neural Networks

by

Yang Li

Committee Chair: Shihao Ji

Committee: Rajshekhar Sunderraman

Murray Patterson

WenZhan Song

Electronic Version Approved:

Office of Graduate Services

College of Arts and Sciences

Georgia State University

August 2024

# DEDICATION

I dedicate this work to my parents, Yongtai Li and Xinrong Yang. Their unwavering support, encouragement, and unconditional love have made this possible.

## ACKNOWLEDGMENTS

I am profoundly grateful to my advisor, Dr. Jonathon Shihao Ji, whose unwavering support has been the cornerstone of my Ph.D. journey. His insightful guidance and deep expertise were instrumental in shaping this dissertation and navigating the challenges of research.

I sincerely appreciate my esteemed thesis committee members: Dr. Rajshekhar Sunderraman, Dr. Murray Patterson, and Dr. WenZhan Song. Their insightful feedback, constructive critiques, and thought-provoking questions pushed me to explore the depths of my research, greatly enhancing its quality.

My heartfelt thanks go to Dr. Shaobo Han, my mentor at NEC Labs, for his invaluable guidance and encouragement. I extend special thanks to my dedicated colleagues in the research group: Dr. Xiang Li, Dr. Xiulong Yang, Dr. Yang Ye, Qing Su, Hui Ye, and Parsa Ghazvinian. Their collaboration and support were invaluable.

I am deeply thankful to my friends Changlei Li, Dr. Kiril Kuzmin, Dr. Dongjie Wang, Wei Wang, and Mu Ge for their unwavering support and friendship.

Lastly, I extend my gratitude to everyone who contributed to the completion of this dissertation. Although I cannot individually name each person, please accept my deepest thanks. This research would not have been possible without the collaborative support and encouragement of these individuals, for which I am truly grateful.

# TABLE OF CONTENTS

**ACKNOWLEDGMENTS** . . . . . . . . . . . . . . . . . . . . . . . . . . . **v**

**LIST OF TABLES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **x**

**LIST OF FIGURES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xii**

**1 INTRODUCTION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**1.1 Network Pruning** . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

**1.2 Pruning Methods** . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**1.3 Parameter Efficient Fine-tuning** . . . . . . . . . . . . . . . . . . . 4

**1.4 Dissertation Organization** . . . . . . . . . . . . . . . . . . . . . . 5

**1.5 List of Publications** . . . . . . . . . . . . . . . . . . . . . . . . . 7

**2 $L_0$-ARM: Network Sparsification via Stochastic Binary Optimization** . . **9**

**2.1 Formulation** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

**2.2 $L_0$-ARM: Stochastic Binary Optimization** . . . . . . . . . . . . . . 13

***2.2.1 Choice of $g(\phi)$*** . . . . . . . . . . . . . . . . . . . . . . . . 15

***2.2.2 Sparsifying Network Architectures for Inference*** . . . . . . . 17

***2.2.3 Imposing Shrinkage on Model Parameters $\theta$*** . . . . . . . . . . 18

***2.2.4 Group Sparsity Under $L_0$ and $L_2$ Norms*** . . . . . . . . . . . 19

**2.3 Related Work** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

**2.4 Experimental Results** . . . . . . . . . . . . . . . . . . . . . . . . 21

***2.4.1 Implementation Details*** . . . . . . . . . . . . . . . . . . . . 22

***2.4.2 MNIST Experiments*** . . . . . . . . . . . . . . . . . . . . . . 23

***2.4.3 CIFAR Experiments*** . . . . . . . . . . . . . . . . . . . . . . 26

**2.5 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**3 Neural Plasticity Networks** . . . . . . . . . . . . . . . . . . . . . . . . . **31**

**3.1 Neural Plasticity Networks: Formulation** . . . . . . . . . . . . . . . . . 33

**3.2 Learning Stage Scheduler** . . . . . . . . . . . . . . . . . . . . . . . . 39

***3.2.1 Dropout as*** $k = 0$ . . . . . . . . . . . . . . . . . . . . . . . . 40

***3.2.2 Pre-training as*** $k = \infty$ ***at the beginning of NPN training*** . . 41

***3.2.3 Fine-tuning as*** $k = \infty$ ***at the end of NPN training*** . . . . . . 41

***3.2.4 Modulating learning stages by*** $k$ . . . . . . . . . . . . . . . . . 42

**3.3 Network Expansion** . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

**3.4 Related Work** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

***3.4.1 Network Sparsification*** . . . . . . . . . . . . . . . . . . . . . . 44

***3.4.2 Neural Architecture Search*** . . . . . . . . . . . . . . . . . . . . 45

***3.4.3 Dynamic Network Expansion*** . . . . . . . . . . . . . . . . . . . 46

**3.5 Experimental Results** . . . . . . . . . . . . . . . . . . . . . . . . . . 46

***3.5.1 Synthetic Dataset*** . . . . . . . . . . . . . . . . . . . . . . . . . 48

***3.5.2 MNIST*** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

***3.5.3 CIFAR-10/100*** . . . . . . . . . . . . . . . . . . . . . . . . . . . 51

**3.6 Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

**4 Dep-$L_0$: Improving $L_0$-based Network Sparsification via Dependency Modeling** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **57**

**4.1 Method** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

***4.1.1 Sparse Structure Learning*** . . . . . . . . . . . . . . . . . . . . 59

***4.1.2 Group Sparsity*** . . . . . . . . . . . . . . . . . . . . . . . . . . 63

***4.1.3 Gate Partition*** . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

***4.1.4 Neural Dependency Modeling*** . . . . . . . . . . . . . . . . . . . 66

**4.2 Related Work** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68

**4.3 Experiments** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

***4.3.1 Experimental Details*** . . . . . . . . . . . . . . . . . . . . . . . 70

***4.3.2 CIFAR10 Results*** . . . . . . . . . . . . . . . . . . . . . . . . . 72

*4.3.3 CIFAR100 Results* . . . . . . . . . . . . . . . . . . . . . . . 74
*4.3.4 ImageNet Results* . . . . . . . . . . . . . . . . . . . . . . . 74
*4.3.5 Study of Learned Sparse Structures* . . . . . . . . . . . . . . . . . 75
*4.3.6 Run-time Comparison* . . . . . . . . . . . . . . . . . . . . . 78
4.4 Ablation Study of the Gate Generator . . . . . . . . . . . . . . . . . 79
*4.4.1 MLP variants* . . . . . . . . . . . . . . . . . . . . . . . 79
*4.4.2 CNN* . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
*4.4.3 LSTM* . . . . . . . . . . . . . . . . . . . . . . . . . . 80
*4.4.4 Summary* . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.5 Conclusion and Future Work . . . . . . . . . . . . . . . . . . . . . 81
**5 VB-LoRA: Extreme Parameter Efficient Fine-Tuning with Vector Banks 83**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
*5.2.1 Exploit Global Redundancy for Enhanced Parameter Efficiency* . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
*5.2.2 Parameter Modeling based on Sparse Admixture Models* . . 86
5.3 Proposed Method . . . . . . . . . . . . . . . . . . . . . . . . . . 87
*5.3.1 Preliminaries: Transformer Architecture and LoRA Adapters* 87
*5.3.2 Divide-and-Share: a New Paradigm for Parameter Sharing* 88
*5.3.3 Breaking Boundaries of LoRA for Global Parameter Sharing* 91
*5.3.4 Parameter Count* . . . . . . . . . . . . . . . . . . . . . . 92
5.4 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94
*5.4.1 Natural Language Understanding* . . . . . . . . . . . . . . . . 94
*5.4.2 Natural Language Generation* . . . . . . . . . . . . . . . . . 98
*5.4.3 Instruction Tuning* . . . . . . . . . . . . . . . . . . . . . 98
*5.4.4 Ablation Study* . . . . . . . . . . . . . . . . . . . . . . . 100
5.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
**6 CONCLUSION AND FUTURE WORK . . . . . . . . . . . . . . . . . . . . 105**

**6.1 Pruning Transformer-based Models** . . . . . . . . . . . . . . . . . . . . 105

**6.2 Multi-task Parameter-efficient Fine-tuning** . . . . . . . . . . . . . . . . . 107

**6.3 Distributed Parameter-efficient Fine-tuning** . . . . . . . . . . . . . . . . 107

**REFERENCES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109**

# LIST OF TABLES

Table 2.1 Architectural details of WRN incorporated with $L_0$-ARM. . . . . . . . 23

Table 2.2 Performance comparison on MNIST. . . . . . . . . . . . . . . . . . . . 24

Table 2.3 Performance comparison of WRN on CIFAR-10. . . . . . . . . . . . . 27

Table 2.4 Performance comparison of WRN on CIFAR-100. . . . . . . . . . . . 28

Table 3.1 The network sparsification and expansion with LeNet5 on MNIST. . . 51

Table 3.2 The network sparsification and expansion with ResNet56 on CIFAR10 and CIFAR100. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Table 4.1 Comparison of pruning methods on CIFAR10. . . . . . . . . . . . . . 73

Table 4.2 Comparison of pruning methods on CIFAR100. . . . . . . . . . . . . 73

Table 4.3 Comparison of pruning methods on ImageNet. . . . . . . . . . . . . . 75

Table 4.4 Run-time comparison between Dep-$L_0$ and $L_0$-HC. . . . . . . . . . . . 78

Table 4.5 Ablation study of the gate generator architecture with VGG16 on CIFAR10. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79

Table 5.1 Hyperparameters and computing resources for natural language understanding experiments on the GLUE benchmark. . . . . . . . . . . . . . . . . 95

Table 5.2 Hyperparameters and computing resources on natural language generation experiments on the E2E dataset. . . . . . . . . . . . . . . . . . . . . 96

Table 5.3 Hyperparameters and computing resources on instruction tuning on the Cleaned Alpaca Dataset. . . . . . . . . . . . . . . . . . . . . . . . . . 96

Table 5.4 Results with $\text{RoBERTa}_{\text{base}}$ and $\text{RoBERTa}_{\text{large}}$ on the GLUE benchmark. 97

Table 5.5 Results with GPT-2 Medium and GPT-2 Large on the E2E benchmark. 98

Table 5.6 Results with Llama2 on the MT-Bench dataset. . . . . . . . . . . . . 100

Table 5.7 Ablation study of different vector selection methods. . . . . . . . . . 102

Table 5.8 Ablation study of sub-vector length. . . . . . . . . . . . . . . . . . . . 102

## LIST OF FIGURES

Figure 1.1 Visualization of the weights of a convolutional filter and different pruning granularities. . . . . 2

Figure 2.1 The plots of $g(\phi)$ with different $k$ for sigmoid and hard sigmoid functions. 16

Figure 2.2 Evolution of the histogram of $g(\phi)$ over training epochs. . . . . 18

Figure 2.3 Comparison of prune rate of sparsified network as a function of epoch for different algorithms. . . . . 26

Figure 2.4 Comparison of expected FLOPs as a function of epoch for different algorithms during training and inference. . . . . 26

Figure 2.5 Comparison of expected FLOPs as a function of iteration during training and inference. . . . . 29

Figure 2.6 Comparison of test accuracy as a function of epoch for different algorithms on CIFAR-10. . . . . 30

Figure 3.1 The plots of $g(\phi)$ with different $k$ for sigmoid and hard sigmoid functions. 39

Figure 3.2 The evolution of the decision boundaries of NPNs for network expansion (a,b,c) and network sparsification (d,e,f). . . . . 55

Figure 3.3 The evolution of network capacity and test accuracy as a function of epoch for NPN network sparsification and expansion with LetNet5 on MNIST. 56

Figure 4.1 Illustration of (a) element-wise sequential dependency modeling, and (b) partition-wise dependency modeling. . . . . 65

Figure 4.2 The computational graph of Dep-$L_0$. . . . . 67

Figure 4.3 The layer-wise prune ratios (red curves) of learned sparse structures. . 76

Figure 4.4 The layer-wise prune ratios (red curves) of learned sparse structures. . 77

Figure 5.1 Overview of VB-LoRA. . . . . 84

Figure 5.2 Comparison of the PEFT methods on RoBERTa-Large. . . . . 84

Figure 5.3 VB-LoRA's vector selection footprints during training. . . . . . . . . . 103

# CHAPTER 1
# INTRODUCTION

Deep Neural Networks (DNNs) have achieved great success in a broad range of applications in image recognition (Deng et al. 2009a), natural language processing (Devlin et al. 2018), and games (Silver et al. 2016). Latest DNN architectures, such as DenseNet (Huang et al. 2017), ResNet (He et al. 2016a) and Transformer (Vaswani et al. 2017), incorporate hundreds of millions of parameters to achieve state-of-the-art predictive performance. However, the expanding number of parameters not only increases the risk of overfitting, but also leads to high computational costs. Many practical real-time applications of DNNs, such as for smart phones, drones and the IoT (Internet of Things) devices, call for compute and memory efficient models as these devices typically have very limited computation and memory capacities.

## 1.1 Network Pruning

It has been shown that DNNs can be pruned or sparsified significantly with minor accuracy losses (Han et al. 2015, 2016), and sometimes sparsified networks can even achieve higher accuracies due to the regularization effects of the network sparsification algorithms (Neklyudov et al. 2017; Louizos et al. 2018b). Driven by the widely spread applications of DNNs in real-time systems, there has been an increasing interest in pruning or sparsifying networks recently (Han et al. 2015, 2016), (Wen et al. 2016a; Li et al. 2016; Louizos et al. 2017; Molchanov et al. 2017; Neklyudov et al. 2017; Louizos et al. 2018b). The existing network pruning algorithms can be roughly categorized into two categories according to the

pruning granularity: unstructured pruning (LeCun et al. 1990; Han et al. 2015; Ding et al. 2019b; Park et al. 2020) and structured pruning (Li et al. 2016; Wen et al. 2016b; Liu et al. 2017; Zhuang et al. 2018; Ding et al. 2019a; You et al. 2019; Lin et al. 2020). As shown in Fig. 1.1, unstructured pruning includes weight-level, vector-level and kernel-level pruning, while structured pruning normally refers to filter-level pruning. Although unstructured pruning methods usually lead to higher prune rates than structured ones, they require specialized hardware or software to fully utilize the benefits induced by the high prune rates due to the irregular network structures yielded by unstructured pruning. On the other hand, structured pruning can maintain the regularity of network structures, while pruning the networks effectively, and hence can fully utilize the parallel computing resources of general-purpose CPUs or GPUs. Because of this, in recent years structured pruning has attracted a lot of attention and achieved impressive performances (Liu et al. 2017; Zhuang et al. 2018; Ding et al. 2019a; Lin et al. 2020).

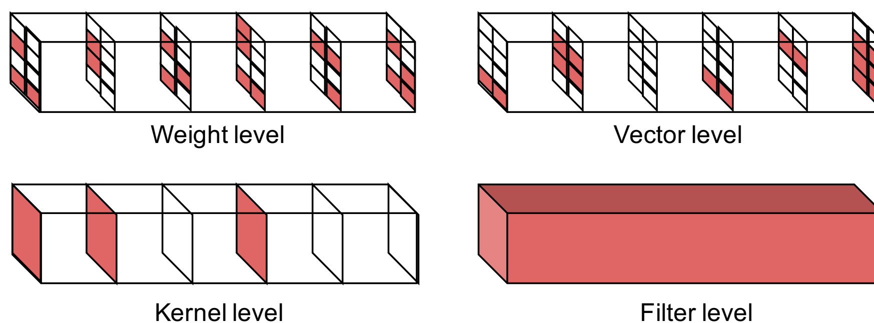



Figure 1.1: Visualization of the weights of a convolutional filter and different pruning granularities.

## 1.2 Pruning Methods

A large subset of pruning methods is heuristic-based, which assigns an importance score to each weight and prune the weights whose importance scores are below a threshold. The importance scores are usually devised according to types of networks, e.g., the magnitude of weights (LeCun et al. 1990; Han et al. 2015) (Feed-forward NNs), the $L_1$ or $L_2$ norm of filters (Li et al. 2016) (CNNs), and the average percentage of zero activations (Hu et al. 2016) (CNNs). However, Ye et al. (Ye et al. 2018) point out that the assumption that weights/filters of smaller norms are less important may not hold in general, challenging the heuristic-based approaches. These methods usually follow a three-step training procedure: training - pruning - retraining in order to achieve the best performance.

Another subset of pruning methods focuses on training networks with sparsity inducing regularizations. For example, $L_2$ and $L_1$ regularizations (Liu et al. 2015; Wen et al. 2016b) or $L_0$ regularization (Louizos et al. 2018a) can be incorporated into the objective functions to train sparse networks. Similarly, Molchanov et al. (Molchanov et al. 2017) propose variational dropout, a sparse Bayesian learning algorithm under an improper logscale uniform prior, to induce sparsity. In this framework, network pruning can be performed from scratch and gradually fulfilled during training without separated training stages.

Recently, the $L_0$-norm based regularization method (Louizos et al. 2018b) is getting attraction as this approach *explicitly* penalizes number of non-zero parameters and can drive redundant or insignificant parameters to be exact zero. However, the gradient of the $L_0$ regularized objective function is intractable. Louizos et al. (Louizos et al. 2018b) propose

to use the hard concrete distribution as a close surrogate to the Bernoulli distribution, and this leads to a differentiable objective function while still being able to zeroing out redundant or insignificant weights during training.

### 1.3 Parameter Efficient Fine-tuning

Parameter-efficient fine-tuning (PEFT) casts a new paradigm that leverages strong prior knowledge built in foundation models and adapts them to a wide range of downstream tasks by updating a small amount of trainable parameters (He et al. 2021). Compared to prefix/prompt tuning (Li & Liang 2021; Lester et al. 2021) or in-context learning (Brown et al. 2020), fine-tuning a large-scale pre-trained model yields better domain specialization dictated by high-quality datasets (Brown et al. 2020; Liu et al. 2022; Zhao et al. 2023). This process can be repeated to suit the needs of ever-changing deployment scenarios and personalizations. However, the sheer volume of parameter space across a multitude of instantiations (Sheng et al. 2023) poses challenges for storage, transmission, and computation, especially for low-resource hardware and consumer-grade networks (Borzunov et al. 2024).

To mitigate these challenges, various PEFT methods have been proposed by adding or adapting a small amount of trainable parameters per task without sacrificing performance (Houlsby et al. 2019; Karimi Mahabadi et al. 2021; Ding et al. 2023). These methods exploit the dependencies among model parameters to reduce the redundancy. For example, Hu et al. (2021) propose the low-rank adaptation (LoRA) to approximate the accumulated gradient update for self-attention modules, and induces the intra-matrix parameter

coupling. Renduchintala et al. (2023) further study the options of allowing the inter-matrix parameter sharing via weight tying across all the layers. In both cases, the number of trainable parameters is reduced significantly. These two methods stand at the two extremes of spectrum in deciding the range of model components reuse (locally or across-layers) and designating which low-rank matrices needs to be shared and updated. However, as the model size increases and the demand for user-customized models across various services rises, the expense of storing and transmitting the customizations for each combination escalates and emerges as a critical issue. Hence, investigating PEFT methods with significantly smaller number of trainable parameters has attracted a flurry of research interests (Kopiczko et al. 2024; Renduchintala et al. 2023).

### 1.4 Dissertation Organization

The overall structure of this dissertation is organized as below. We introduce the background of neural networks pruning and Parameter-efficient fine-tuning in Chapter 1. From Chapter 2 to Chapter 5, we will present four proposed algorithms.

In Chapter 2, we propose an $L_0$-norm based pruning algorithm $L_0$-ARM. $L_0$-ARM is built on top of the $L_0$ regularization framework of Louizos et al. (Louizos et al. 2018b). However, instead of using a biased hard concrete gradient estimator, we investigate the Augment-Reinforce-Merge (ARM) (Yin & Zhou 2019), a recently proposed unbiased gradient estimator for stochastic binary optimization. Extensive experiments on multiple public datasets demonstrate the superior performance of $L_0$-ARM at sparsifying networks with fully

connected layers and convolutional layers.

In Chapter 3, we propose Neural Plasticity Network (NPN) which can prune or expand a network depending on the initial capacity of network provided by the user. Neural plasticity is an important functionality of human brain, in which number of neurons and synapses can shrink or expand in response to stimuli throughout the span of life. We model this dynamic learning process as an $L_0$-norm regularized binary optimization problem. We show that both network sparsification and network expansion can yield compact models of similar architectures, while retaining competitive accuracies of the original networks.

In Chapter 4, we propose an algorithm Dep-$L_0$ as it prunes networks via a dependency-enabled $L_0$ regularization. Based on the observation (Gale et al. 2019a) that although $L_0$-based pruning method yields high compression rates on smaller datasets, it performs inconsistently on large-scale learning tasks, such as ResNet50 on ImageNet. We analyze this phenomenon through the lens of variational inference and find that it is likely due to the independent modeling of binary gates, the mean-field approximation (Blei et al. 2017), which is known in Bayesian statistics for its poor performance due to the crude approximation. To mitigate this deficiency, we propose a dependency modeling of binary gates, which can be modeled effectively as a multi-layer perceptron (MLP). Compared with the state-of-the-arts network sparsification algorithms, our dependency modeling makes the $L_0$-based sparsification once again very competitive on large-scale learning tasks.

In Chapter 5, we introduce a "divide-and-share" paradigm that breaks the barriers of low-rank decomposition across matrix dimensions, modules and layers by sharing parame-

ters globally via a *vector bank*. As an instantiation of the paradigm to LoRA, our proposed VB-LoRA composites *all* the low-rank matrices of LoRA from a shared *vector bank* with a differentiable top-$k$ admixture module. VB-LoRA achieves extreme parameter efficiency while maintaining comparable or better performance compared to state-of-the-art PEFT methods. Extensive experiments demonstrate the effectiveness of VB-LoRA on natural language understanding, natural language generation, and instruction tuning tasks. When fine-tuning the Llama2-13B model, VB-LoRA only uses 0.4% of LoRA's stored parameters, yet achieves superior results.

### 1.5 List of Publications

- Yang Li, Shihao Ji, **Dep-$L_0$: Improving $L_0$-based Network Sparsification via Dependency Modeling**, The European Conference on Machine Learning 2021. (Li & Ji 2021a)

- Yang Li, Shihao Ji, **Neural Plasticity Networks**, International Joint Conference on Neural Networks 2021. (Li & Ji 2021b)

- Yang Li, Shihao Ji, **$L_0$-ARM: Network Sparsification via Stochastic Binary Optimization**, The European Conference on Machine Learning 2019. (Li & Ji 2019)

- Yang Li, Shaobo Han, and Shihao Ji, **VB-LoRA: Extreme Parameter Efficient Fine-Tuning with Vector Banks**, arXiv preprint arXiv:2405.15179 (2024). (Li et al. 2024)

- Yang Li, Xin Ma, Raj Sunderraman, Shihao Ji, and Suprateek Kundu. **Accounting for temporal variability in functional magnetic resonance imaging improves prediction of intelligence**, Human Brain Mapping 44, no. 13 (2023): 4772-4791. (Li et al. 2023)

- K. Sarker, X. Yang, Y. Li, S. Belkasim, and Shihao Ji, **A Unified Density-Driven Framework for Effective Data Denoising and Robust Abstention**, IEEE International Conference on Image Processing 2021. (Sarker et al. 2020)

- Fatih Yaman, Yang Li, Shaobo Han, Takanori Inoue, Eduardo Mateo, and Yoshihisa Inada. **Polarization sensing using polarization rotation matrix eigenvalue method**, Optical Fiber Communication Conference 2023. (Yaman et al. 2023)

# CHAPTER 2

## $L_0$-ARM: Network Sparsification via Stochastic Binary Optimization

In this chapter, we propose $L_0$-ARM for network sparsification. $L_0$-ARM is built on top of the $L_0$ regularization framework of Louizos et al. Louizos et al. (2018b). However, instead of using a biased hard concrete gradient estimator, we investigate the Augment-Reinforce-Merge (ARM) Yin & Zhou (2019), a recently proposed unbiased gradient estimator for stochastic binary optimization. Because of the unbiasness and flexibility of the ARM estimator, $L_0$-ARM exhibits a significantly faster rate at pruning network architectures and reducing FLOPs than the hard concrete estimator. Extensive experiments on multiple public datasets demonstrate the superior performance of $L_0$-ARM at sparsifying networks with fully connected layers and convolutional layers. It achieves state-of-the-art prune rates while retaining similar accuracies compared to baseline methods. Additionally, it sparsifies the Wide-ResNet models on CIFAR-10 and CIFAR-100 while the original hard concrete estimator cannot.

This chapter is organized as follows. In Sec. 2.1 we describe the $L_0$ regularized empirical risk minimization for network sparsification and formulate it as a stochastic binary optimization problem. A new unbiased estimator to this problem $L_0$-ARM is presented in Sec. 2.2, followed by related work in Sec. 2.3. Example results on multiple public datasets are presented in Sec. 2.4, with comparisons to baseline methods and the state-of-the-art sparsification algorithms. Conclusions and future work are discussed in Sec. 2.5.

### 2.1 Formulation

Given a training set $D = \{(\boldsymbol{x}_i, y_i), i = 1, 2, \cdots, N\}$, where $\boldsymbol{x_i}$ denotes the input and $y_i$ denotes the target, a neural network is a function $h(\boldsymbol{x}; \boldsymbol{\theta})$ parametrized by $\boldsymbol{\theta}$ that fits to the training data $D$ with the goal of achieving good generalization to unseen test data. To optimize $\boldsymbol{\theta}$, typically a regularized empirical risk is minimized, which contains two terms – a data loss over training data and a regularization loss over model parameters. Empirically, the regularization term can be weight decay or Lasso, i.e., the $L_2$ or $L_1$ norm of model parameters.

Since the $L_2$ or $L_1$ norm only imposes shrinkage for large values of $\boldsymbol{\theta}$, the resulting model parameters $\boldsymbol{\theta}$ are often manifested by smaller magnitudes but none of them are exact zero. Intuitively, a more appealing alternative is the $L_0$ regularization since the $L_0$-norm measures *explicitly* the number of non-zero elements, and minimizing of it over model parameters will drive the redundant or insignificant weights to be exact zero. With the $L_0$ regularization, the empirical risk objective can be written as

$$\mathcal{R}(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(h(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i\right) + \lambda \|\boldsymbol{\theta}\|_0 \tag{2.1.1}$$

where $\mathcal{L}(\cdot)$ denotes the data loss over training data $D$, such as the cross-entropy loss for classification or the mean squared error (MSE) for regression, and $\|\boldsymbol{\theta}\|_0$ denotes the $L_0$-norm over model parameters, i.e., the number of non-zero weights, and $\lambda$ is a regularization hyper-parameter that balances between data loss and model complexity.

To represent a sparsified model, we attach a binary random variable $z$ to each element

of model parameters $\boldsymbol{\theta}$. Therefore, we can re-parameterize the model parameters $\boldsymbol{\theta}$ as an element-wise product of non-zero parameters $\widetilde{\boldsymbol{\theta}}$ and binary random variables $\boldsymbol{z}$:

$$\boldsymbol{\theta} = \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}, \tag{2.1.2}$$

where $\boldsymbol{z} \in \{0,1\}^{|\boldsymbol{\theta}|}$, and $\odot$ denotes the element-wise product. As a result, Eq. 2.1.1 can be rewritten as:

$$\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left(h\big(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}\big), y_i\right) + \lambda \sum_{j=1}^{|\widetilde{\boldsymbol{\theta}}|} \mathbf{1}_{[z_j \neq 0]}, \tag{2.1.3}$$

where $\mathbf{1}_{[c]}$ is an indicator function that is 1 if the condition $c$ is satisfied, and 0 otherwise. Note that both the first term and the second term of Eq. 2.1.3 are not differentiable w.r.t. $\boldsymbol{z}$. Therefore, further approximations need to be considered.

According to stochastic variational optimization Bird et al. (2018), given any function $\mathcal{F}(\boldsymbol{z})$ and any distribution $q(\boldsymbol{z})$, the following inequality holds

$$\min_{\boldsymbol{z}} \mathcal{F}(\boldsymbol{z}) \leq \mathbb{E}_{\boldsymbol{z} \sim q(\boldsymbol{z})}[\mathcal{F}(\boldsymbol{z})], \tag{2.1.4}$$

i.e., the minimum of a function is upper bounded by the expectation of the function. With this result, we can derive an upper bound of Eq. 2.1.3 as follows.

Since $z_j, \forall j \in \{1, \cdots, |\boldsymbol{\theta}|\}$ is a binary random variable, we assume $z_j$ is subject to a Bernoulli distribution with parameter $\pi_j \in [0,1]$, i.e. $z_j \sim \text{Ber}(z; \pi_j)$. Thus, we can upper

bound $\min_{\boldsymbol{z}} \mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z})$ by the expectation

$$\begin{aligned}\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\pi}) &= \mathbb{E}_{\boldsymbol{z}\sim \mathrm{Ber}(\boldsymbol{z};\boldsymbol{\pi})}\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z}) \\ &= \mathbb{E}_{\boldsymbol{z}\sim \mathrm{Ber}(\boldsymbol{z};\boldsymbol{\pi})}\left[\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(h(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}), y_i\right)\right] + \lambda \sum_{j=1}^{|\widetilde{\boldsymbol{\theta}}|} \pi_j. \end{aligned} \tag{2.1.5}$$

As we can see, now the second term is differentiable w.r.t. the new model parameters $\boldsymbol{\pi}$, while the first term is still problematic since the expectation over a large number of binary random variables $\boldsymbol{z}$ is intractable and so its gradient. Since $\boldsymbol{z}$ are binary random variables following a Bernoulli distribution with parameters $\boldsymbol{\pi}$, we now formulate the original $L_0$ regularized empirical risk (2.1.1) to a stochastic binary optimization problem (2.1.5).

Existing gradient estimators for this kind of discrete latent variable models include REINFORCE Williams (1992), Gumble-Softmax Jang et al. (2017); Maddison et al. (2017), REBAR Tucker et al. (2017), RELAX Grathwohl et al. (2018) and the Hard Concrete estimator Louizos et al. (2018b). However, these estimators either are biased or suffer from high variance or computationally expensive due to auxiliary modeling. Recently, the Augment-Reinforce-Merge (ARM) Yin & Zhou (2019) gradient estimator is proposed for the optimization of binary latent variable models, which is unbiased and exhibits low variance. Extending this gradient estimator to network sparsification, we find that ARM demonstrates superior performance of prunning network architectures while retaining almost the same accuracies of baseline models. More importantly, similar to the hard concrete estimator, ARM also enables conditional computation Bengio et al. (2013) that not only sparsifies model architectures for

inference but also accelerates model training.

## 2.2 $L_0$-ARM: Stochastic Binary Optimization

To minimize Eq. 2.1.5, we propose $L_0$-ARM, a stochastic binary optimization algorithm based on the Augment-Reinforce-Merge (ARM) gradient estimator Yin & Zhou (2019). We first introduce the main theorem of ARM. Refer readers to Yin & Zhou (2019) for the proof and other details.

**Theorem 2.2.1.** (ARM) Yin & Zhou (2019). For a vector of $V$ binary random variables $\boldsymbol{z} = (z_1, \cdots, z_V)$, the gradient of

$$\mathcal{E}(\boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{z} \sim \prod_{v=1}^{V} \text{Ber}(z_v; g(\phi_v))}[f(\boldsymbol{z})] \tag{2.2.1}$$

w.r.t. $\boldsymbol{\phi} = (\phi_1, \cdots, \phi_V)$, the logits of the Bernoulli distribution parameters, can be expressed as

$$\nabla_\phi \mathcal{E}(\phi) = \mathbb{E}_{\boldsymbol{u} \sim \prod_{v=1}^{V} \text{Uniform}(u_v; 0,1)} \Big[ \big( f(\mathbf{1}_{[\boldsymbol{u} > g(-\phi)]}) - f(\mathbf{1}_{[\boldsymbol{u} < g(\boldsymbol{\phi})]}) \big) (\boldsymbol{u} - 1/2) \Big], \tag{2.2.2}$$

where $\mathbf{1}_{[\boldsymbol{u} > g(-\boldsymbol{\phi})]} := \big( \mathbf{1}_{[u_1 > g(-\phi_1)]}, \cdots, \mathbf{1}_{[u_V > g(-\phi_V)]} \big)^T$ and $g(\phi) = \sigma(\phi) = 1/(1 + \exp(-\phi))$ is the sigmoid function.

Parameterizing $\pi_j \in [0, 1]$ as $g(\phi_j)$, Eq. 2.1.5 can be rewritten as

$$\begin{aligned}\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) &= \mathbb{E}_{\boldsymbol{z}\sim\text{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[f(\boldsymbol{z})\right] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} g(\boldsymbol{\phi}_j) \\ &= \mathbb{E}_{\boldsymbol{u}\sim\text{Uniform}(\boldsymbol{u};0,1)}\left[f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})\right] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} g(\boldsymbol{\phi}_j),\end{aligned} \tag{2.2.3}$$

where $f(\boldsymbol{z}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\left(h(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}), y_i\right)$.

Now according to Theorem 1, we can evaluate the gradient of Eq. 2.2.3 w.r.t. $\boldsymbol{\phi}$ by

$$\begin{aligned}\nabla_{\boldsymbol{\phi}}^{ARM}\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u}\sim\text{Uniform}(\boldsymbol{u};0,1)}&\Big[\big(f(\mathbf{1}_{[\boldsymbol{u}>g(-\boldsymbol{\phi})]}) - f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})\big)(\boldsymbol{u} - 1/2)\Big] \\ &+ \lambda \sum_{j=1}^{|\widetilde{\theta}|} \nabla_{\phi_j} g(\phi_j),\end{aligned} \tag{2.2.4}$$

which is an unbiased and low variance estimator as demonstrated in Yin & Zhou (2019).

Note from Eq. 2.2.4 that we need to evaluate $f(\cdot)$ twice to compute the gradient, the second of which is the same operation required by the data loss of Eq. 2.2.3. Therefore, one extra forward pass $f(\mathbf{1}_{[\boldsymbol{u}>g(-\phi)]})$ is required by the $L_0$-ARM gradient estimator. This additional forward pass might be computationally expensive, especially for networks with millions of parameters. To reduce the computational complexity of Eq. 2.2.4, we further

consider another gradient estimator – Augment-Reinforce (AR) Yin & Zhou (2019):

$$\nabla_{\boldsymbol{\phi}}^{AR}\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}},\boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u}\sim\text{Uniform}(\boldsymbol{u};0,1)}\Big[f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})(1-2\boldsymbol{u})\Big] + \lambda\sum_{j=1}^{|\widetilde{\theta}|}\nabla_{\phi_j}g(\phi_j), \tag{2.2.5}$$

which requires only one forward pass $f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})$ that is the same operation as in Eq. 2.2.3. This $L_0$-AR gradient estimator is still unbiased but with higher variance. Now with $L_0$-AR, we can trade off the variance of the estimator with the computational complexity. We will evaluate the impact of this trade-off in our experiments.

### 2.2.1 Choice of $g(\phi)$

Theorem 1 of ARM defines $g(\phi) = \sigma(\phi)$, where $\sigma(\cdot)$ is the sigmoid function. For the purpose of network sparsification, we find that this parametric function isn't very effective due to its slow transition between values 0 and 1. Thanks to the flexibility of ARM, we have a lot of freedom to design this parametric function $g(\phi)$. Apparently, it's straightforward to generalize Theorem 1 for any parametric functions (smooth or non-smooth) as long as $g : \mathcal{R} \to [0,1]$ and $g(-\phi) = 1 - g(\phi)$ [1]. Example parametric functions that work well in our experiments are the scaled sigmoid function

$$g_{\sigma_k}(\phi) = \sigma(k\phi) = \frac{1}{1+\exp(-k\phi)}, \tag{2.2.6}$$

[1] The second condition is not necessary. But for simplicity, we will impose this condition to select parametric function $g(\phi)$ that is antithetic. Designing $g(\phi)$ without this constraint could be a potential area that is worthy of further investigation.

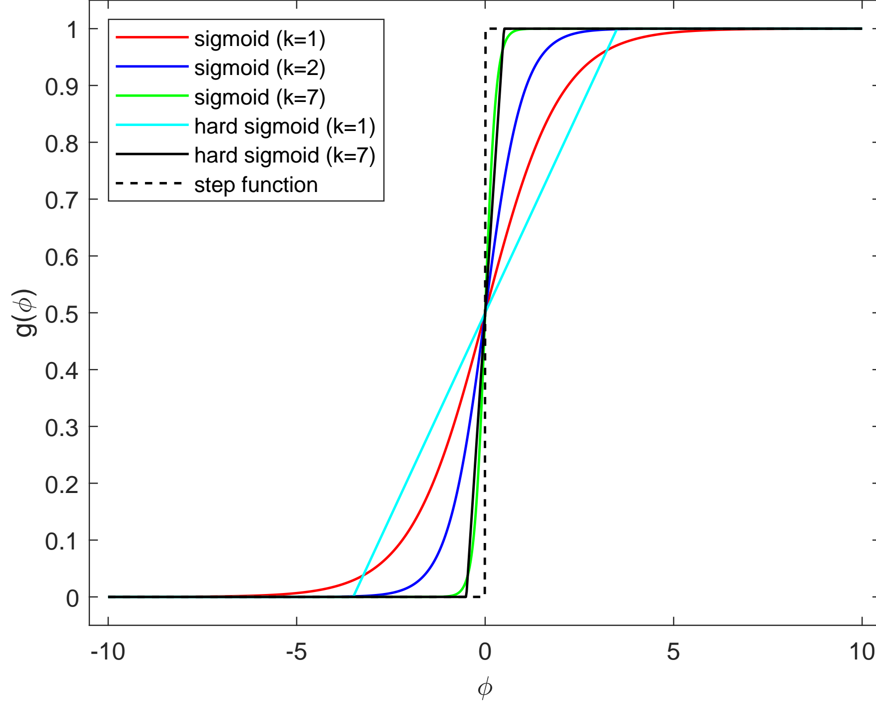



Figure 2.1: The plots of $g(\phi)$ with different $k$ for sigmoid and hard sigmoid functions.

and the centered-scaled hard sigmoid

$$g_{\bar{\sigma}_k}(\phi) = \min(1, \max(0, \frac{k}{7}\phi + 0.5)), \tag{2.2.7}$$

where 7 is introduced such that $g_{\bar{\sigma}_1}(\phi) \approx g_{\sigma_1}(\phi) = \sigma(\phi)$. See Fig. 2.1 for some example plots of $g_{\sigma_k}(\phi)$ and $g_{\bar{\sigma}_k}(\phi)$ with different $k$. Empirically, we find that $k = 7$ works well for all of our experiments.

One important difference between the hard concrete estimator from Louizos et al. Louizos et al. (2018b) and $L_0$-ARM is that the hard concrete estimator has to rely on the hard sigmoid gate to zero out some parameters during training (a.k.a. conditional computation Bengio et al. (2013)), while $L_0$-ARM achieves conditional computation naturally by sampling from

the Bernoulli distribution, parameterized by $g(\phi)$, where $g(\phi)$ can be any parametric function (smooth or non-smooth) as shown in Fig. 2.1. We validate this in our experiments.

### *2.2.2 Sparsifying Network Architectures for Inference*

After training, we get model parameters $\widetilde{\boldsymbol{\theta}}$ and $\boldsymbol{\phi}$. At test time, we can use the expectation of $\boldsymbol{z} \sim \mathrm{Ber}(\boldsymbol{z}; g(\boldsymbol{\phi}))$ as the mask $\widehat{\boldsymbol{z}}$ for the final model parameters $\widehat{\boldsymbol{\theta}}$:

$$\widehat{\boldsymbol{z}} = \mathbb{E}[\boldsymbol{z}] = g(\boldsymbol{\phi}), \qquad \widehat{\boldsymbol{\theta}} = \widetilde{\boldsymbol{\theta}} \odot \widehat{\boldsymbol{z}}. \tag{2.2.8}$$

However, this will not yield a sparsified network for inference since none of the element of $\widehat{\boldsymbol{z}} = g(\boldsymbol{\phi})$ is exact zero (unless the hard sigmoid gate $g_{\bar{\sigma}_k}(\phi)$ is used). A simple approximation is to set the elements of $\widehat{\boldsymbol{z}}$ to zero if the corresponding values in $g(\boldsymbol{\phi})$ are less than a threshold $\tau$, i.e.,

$$\bar{z}_j = \begin{cases} 0, & g(\phi_j) \leq \tau \\ g(\phi_j), & \text{otherwise} \end{cases} \qquad j = 1, 2, \cdots, |\boldsymbol{z}| \tag{2.2.9}$$

We find that this approximation is very effective in all of our experiments as the histogram of $g(\boldsymbol{\phi})$ is widely split into two spikes around values of 0 and 1 after training because of the sharp transition of the scaled sigmoid (or hard sigmoid) function. See Fig. 2.2 for a typical plot of the histograms of $g(\boldsymbol{\phi})$ evolving during training process. We notice that our algorithm isn't very sensitive to $\tau$, tuning which incurs negligible impacts to prune rates and model accuracies. Therefore, for all of our experiments we set $\tau = 0.5$ by default. Apparently, better designed $\tau$ is possible by considering the histogram of $g(\boldsymbol{\phi})$. However, we find this

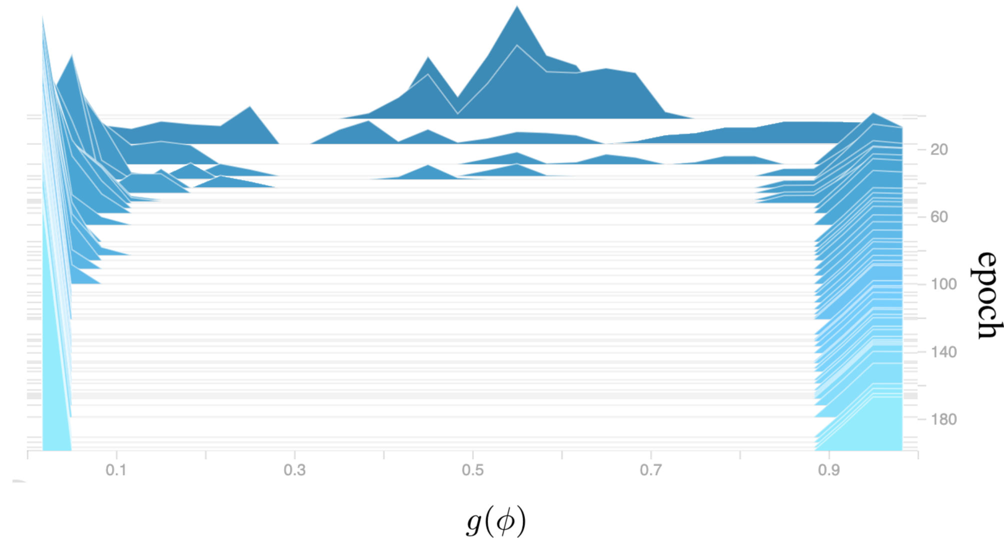



Figure 2.2: Evolution of the histogram of $g(\phi)$ over training epochs.
All $g(\phi)$ are initialized by random samples from a normal distribution $\mathcal{N}(0.5, 0.01)$, which are split into two spikes during training.

isn't very necessary for all of our experiments in the paper. Therefore, we will consider this histogram-dependent $\tau$ as our future improvement.

### *2.2.3 Imposing Shrinkage on Model Parameters* $\theta$

The $L_0$ regularized objective function (2.2.3) leads to sparse estimate of model parameters without imposing any shrinkage on the magnitude of $\boldsymbol{\theta}$. In some cases it might still be desirable to regularize the magnitude of model parameters with other norms, such as $L_1$ or $L_2$ (weight decay), to improve the robustness of model. This can be achieved conveniently by computing the expected $L_1$ or $L_2$ norm of $\boldsymbol{\theta}$ under the same Bernoulli distribution:

$\boldsymbol{z} \sim \mathrm{Ber}(\boldsymbol{z}; g(\boldsymbol{\phi}))$ as follows:

$$\mathbb{E}_{\boldsymbol{z}\sim\mathrm{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[||\boldsymbol{\theta}||_1\right] = \sum_{j=1}^{|\theta|} \mathbb{E}_{z_j\sim\mathrm{Ber}(z_j;g(\phi_j))}\left[z_j|\widetilde{\theta}_j|\right] = \sum_{j=1}^{|\theta|} g(\phi_j)|\widetilde{\theta}_j|, \tag{2.2.10}$$

$$\mathbb{E}_{\boldsymbol{z}\sim\mathrm{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[||\boldsymbol{\theta}||_2^2\right] = \sum_{j=1}^{|\theta|} \mathbb{E}_{z_j\sim\mathrm{Ber}(z_j;g(\phi_j))}\left[z_j^2\widetilde{\theta}_j^2\right] = \sum_{j=1}^{|\theta|} g(\phi_j)\widetilde{\theta}_j^2, \tag{2.2.11}$$

which can be incorporated to Eq. 2.2.3 as additional regularization terms.

### *2.2.4 Group Sparsity Under $L_0$ and $L_2$ Norms*

The formulation so far promotes a weight-level sparsity for network architectures. This sparsification strategy can compress model and reduce memory footprint of a network. However, it will usually not lead to effective speedups because weight-sparsified networks require sparse matrix multiplication and irregular memory access, which make it extremely challenging to effectively utilize the parallel computing resources of GPUs and CPUs. For the purpose of computational efficiency, it's usually preferable to perform group sparsity instead of weight-level sparsity. Similar to Wen et al. (2016a); Neklyudov et al. (2017); Louizos et al. (2018b), we can achieve this by sharing a stochastic binary gate $z$ among all the weights in a group. For example, a group can be all fan-out weights of a neuron in fully connected layers or all weights of a convolution filter. With this, the group regularized $L_0$ and $L_2$ norms can be

conveniently expressed as

$$\mathbb{E}_{\boldsymbol{z}\sim\text{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[||\boldsymbol{\theta}||_0\right] = \sum_{\text{g}=1}^{|G|} |\text{g}| g(\phi_\text{g}) \tag{2.2.12}$$

$$\mathbb{E}_{\boldsymbol{z}\sim\text{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[||\boldsymbol{\theta}||_2^2\right] = \sum_{\text{g}=1}^{|G|} \left( g(\phi_\text{g}) \sum_{j=1}^{|\text{g}|} \widetilde{\theta}_j^2 \right) \tag{2.2.13}$$

where $|G|$ denotes the number of groups and $|\text{g}|$ denotes the number of weights of group g. For the reason of computational efficiency, we perform this group sparsity in all of our experiments.

## 2.3 Related Work

It is well-known that DNNs are extremely compute and memory intensive. Recently, there has been an increasing interest to network sparsification Han et al. (2015, 2016); Wen et al. (2016a); Li et al. (2016); Louizos et al. (2017); Molchanov et al. (2017); Neklyudov et al. (2017); Louizos et al. (2018b) as the applications of DNNs to practical real-time systems, such as the IoT devices, call for compute and memory efficient networks. One of the earliest sparsification methods is to prune the redundant weights based on the magnitudes LeCun et al. (1990), which is proved to be effective in modern CNN Han et al. (2015). Although weight sparsification is able to compress networks, it can barely improve computational efficiency due to unstructured sparsity Wen et al. (2016a). Therefore, magnitude-based group sparsity is proposed Wen et al. (2016a); Li et al. (2016), which can compress networks while reducing computation cost significantly. These magnitude-based methods usually proceed in

three stages: pre-train a full network, prune the redundant weights or filters, and fine-tune the pruned model. As a comparison, our method $L_0$-ARM trains a sparsified network from scratch without pre-training and fine-tuning, and therefore is more preferable.

Another category of sparsification methods is based on Bayesian statistics and information theory Molchanov et al. (2017); Neklyudov et al. (2017); Louizos et al. (2017). For example, inspired by variational dropout Kingma et al. (2015), Molchanov et al. propose a method that unbinds the dropout rate, and also leads to sparsified networks Molchanov et al. (2017).

Recently, Louizos et al. Louizos et al. (2018b) propose to sparsify networks with $L_0$-norm. Since the $L_0$ regularization explicitly penalizes number of non-zero parameters, this method is conceptually very appealing. However, the non-differentiability of $L_0$ norm prevents an effective gradient-based optimization. Therefore, Louizos et al. Louizos et al. (2018b) propose a hard concrete gradient estimator for this optimization problem. Our work is built on top of their $L_0$ formulation. However, instead of using a hard concrete estimator, we investigate the Augment-Reinforce-Merge (ARM) Yin & Zhou (2019), a recently proposed unbiased estimator, to this binary optimization problem.

### 2.4 Experimental Results

We evaluate the performance of $L_0$-ARM and $L_0$-AR on multiple public datasets and multiple network architectures. Specifically, we evaluate MLP 500-300 LeCun et al. (1998) and LeNet 5-Caffe [2] on the MNIST dataset Lecun et al. (1998), and Wide Residual Networks Zagoruyko

[2]https://github.com/BVLC/caffe/tree/master/examples/mnist

& Komodakis (2016a) on the CIFAR-10 and CIFAR-100 datasets Krizhevsky (2009). For baselines, we refer to the following state-of-the-art sparsification algorithms: Sparse Variational Dropout (Sparse VD) Molchanov et al. (2017), Bayesian Compression with group normal-Jeffreys (BC-GNJ) and group horseshoe (BC-GHS) Louizos et al. (2017), and $L_0$-norm regularization with hard concrete estimator ($L_0$-HC) Louizos et al. (2018b). For a fair comparison, we closely follow the experimental setups of $L_0$-HC [3]. The code is public available at `https://github.com/leo-yangli/l0-arm`.

### *2.4.1 Implementation Details*

We incorporate $L_0$-ARM and $L_0$-AR into the architectures of MLP, LeNet-5 and Wide ResNet. As we described in Sec. 2.2.4, instead of sparsifying weights, we apply group sparsity on neurons in fully-connected layers or on convolution filters in convolutional layers. Once a neuron or filter is pruned, all related weights are removed from the networks.

The Multi-Layer Perceptron (MLP) LeCun et al. (1998) has two hidden layers of size 300 and 100, respectively. We initialize $g(\boldsymbol{\phi}) = \boldsymbol{\pi}$ by random samples from a normal distribution $\mathcal{N}(0.8, 0.01)$ for the input layer and $\mathcal{N}(0.5, 0.01)$ for the hidden layers, which activate around 80% of neurons in input layer and around 50% of neurons in hidden layers. LeNet-5-Caffe consists of two convolutional layers of 20 and 50 filters interspersed with max pooling layers, followed by two fully-connected layers with 500 and 10 neurons. We initialize $g(\boldsymbol{\phi}) = \boldsymbol{\pi}$ for all neurons and filters by random samples from a normal distribution $\mathcal{N}(0.5, 0.01)$. Wide-ResNets (WRNs) Zagoruyko & Komodakis (2016a) have shown state-of-the-art performance

[3] `https://github.com/AMLab-Amsterdam/L0_regularization`

on many image classification benchmarks. Following Louizos et al. (2018b), we only apply $L_0$ regularization on the first convolutional layer of each residual block, which allows us to incorporate $L_0$ regularization without further modifying residual block architecture. The architectural details of WRN are listed in Table 2.1. For initialization, we activate around 70% of convolutional filters.

Table 2.1: Architectural details of WRN incorporated with $L_0$-ARM.

| Group name | Layers |
|---|---|
| conv1 | [Original Conv (16)] |
| conv2 | [$L_0$ ARM (160); Original Conv (160)] $\times$ 4 |
| conv3 | [$L_0$ ARM (320); Original Conv (320)] $\times$ 4 |
| conv4 | [$L_0$ ARM (640); Original Conv (640)] $\times$ 4 |

The number in parenthesis is the size of activation map of each layer. For brevity, only the modified layers are included.

For MLP and LeNet-5, we train with a mini-batch of 100 data samples and use Adam Kingma & Ba (2015) as optimizer with initial learning rate of 0.001, which is halved every 100 epochs. For Wide-ResNet, we train with a mini-batch of 128 data samples and use Nesterov Momentum as optimizer with initial learning rate of 0.1, which is decayed by 0.2 at epoch 60 and 120. Each of these experiments run for 200 epochs in total. For a fair comparison, these experimental setups closely follow what were described in $L_0$-HC Louizos et al. (2018b) and their open-source implementation [3].

### *2.4.2 MNIST Experiments*

We run both MLP and LeNet-5 on the MNIST dataset. By tuning the regularization strength $\lambda$, we can control the trade off between sparsity and accuracy. We can use one $\lambda$ for all layers or a separate $\lambda$ for each layer to fine-tune the sparsity preference. In our experiments, we set

$\lambda = 0.1/N$ or $\lambda = (0.1, 0.3, 0.4)/N$ for MLP, and set $\lambda = 0.1/N$ or $\lambda = (10, 0.5, 0.1, 10)/N$ for LeNet-5, where $N$ denotes to the number of training datapoints.

We use three metrics to evaluate the performance of an algorithm: prediction accuracy, prune rate, and expected number of floating point operations (FLOPs). Prune rate is defined as the ratio of number of pruned weights to number of all weights. Prune rate manifests the memory saving of a sparsified network, while expected FLOPs demonstrates the training / inference cost of a sparsification algorithm.

Table 2.2: Performance comparison on MNIST.

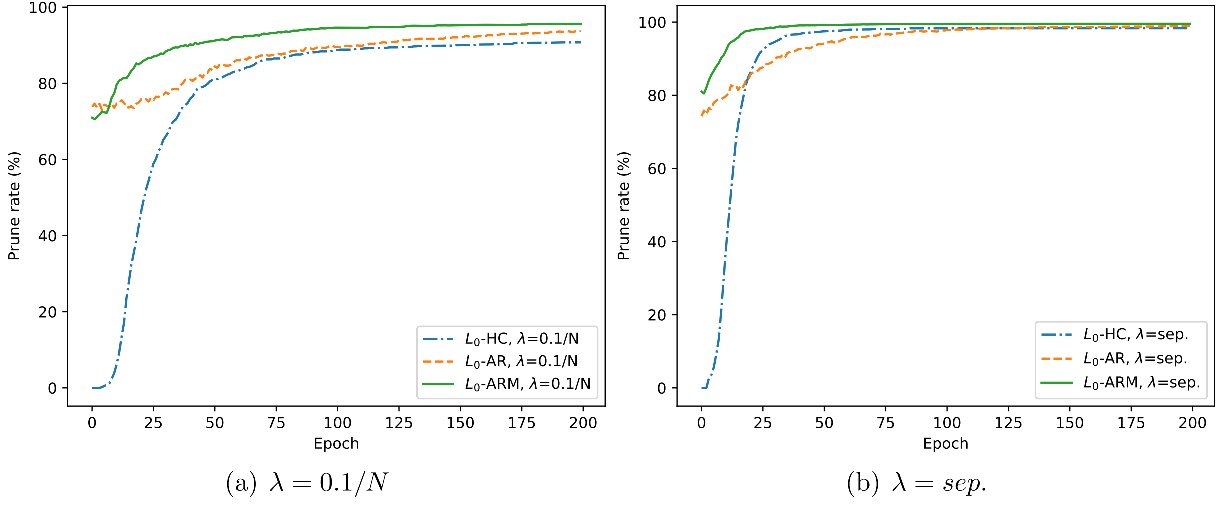
| Network | Method | Pruned Architecture | Prune rate (%) | Accuracy (%) |
|---|---|---|---|---|
| MLP 784-300-100 | Sparse VD | 219-214-100 | 74.72 | 98.2 |
| | BC-GNJ | 278-98-13 | 89.24 | 98.2 |
| | BC-GHS | 311-86-14 | 89.45 | 98.2 |
| | $L_0$-HC ($\lambda = 0.1/N$) | 219-214-100 | 73.98 | 98.6 |
| | $L_0$-HC ($\lambda$ sep.) | 266-88-33 | 89.99 | 98.2 |
| | $L_0$-AR ($\lambda = 0.1/N$) | 453-150-68 | 70.39 | 98.3 |
| | $L_0$-ARM ($\lambda = 0.1/N$) | 143-153-78 | 87.00 | 98.3 |
| | $L_0$-AR ($\lambda$ sep.) | 464-114-65 | 77.10 | 98.2 |
| | $L_0$-ARM ($\lambda$ sep.) | 159-74-73 | **92.96** | 98.1 |
| LeNet-5-Caffe 20-50-800-500 | Sparse VD | 14-19-242-131 | 90.7 | 99.0 |
| | GL | 3-12-192-500 | 76.3 | 99.0 |
| | GD | 7-13-208-16 | 98.62 | 99.0 |
| | SBP | 3-18-284-283 | 80.34 | 99.0 |
| | BC-GNJ | 8-13-88-13 | 99.05 | 99.0 |
| | BC-GHS | 5-10-76-16 | 99.36 | 99.0 |
| | $L_0$-HC ($\lambda = 0.1/N$) | 20-25-45-462 | 91.1 | 99.1 |
| | $L_0$-HC ($\lambda$ sep.) | 9-18-65-25 | 98.6 | 99.0 |
| | $L_0$-AR ($\lambda = 0.1/N$) | 18-28-46-249 | 93.73 | 98.8 |
| | $L_0$-ARM ($\lambda = 0.1/N$) | 20-16-32-257 | 95.52 | 99.1 |
| | $L_0$-AR ($\lambda$ sep.) | 5-12-131-22 | 98.90 | 98.4 |
| | $L_0$-ARM ($\lambda$ sep.) | 6-10-39-11 | **99.49** | 98.7 |

Each experiment was run five times and the median (in terms of accuracy) is reported. All the baseline results are taken from the corresponding papers.

We compare $L_0$-ARM and $L_0$-AR to five state-of-the-art sparsification algorithms on MNIST, with the results shown in Table 2.2. For the comparison between $L_0$-HC and $L_0$-AR(M) when $\lambda = 0.1/N$, we use the exact same hyper-parameters for both algorithms (the

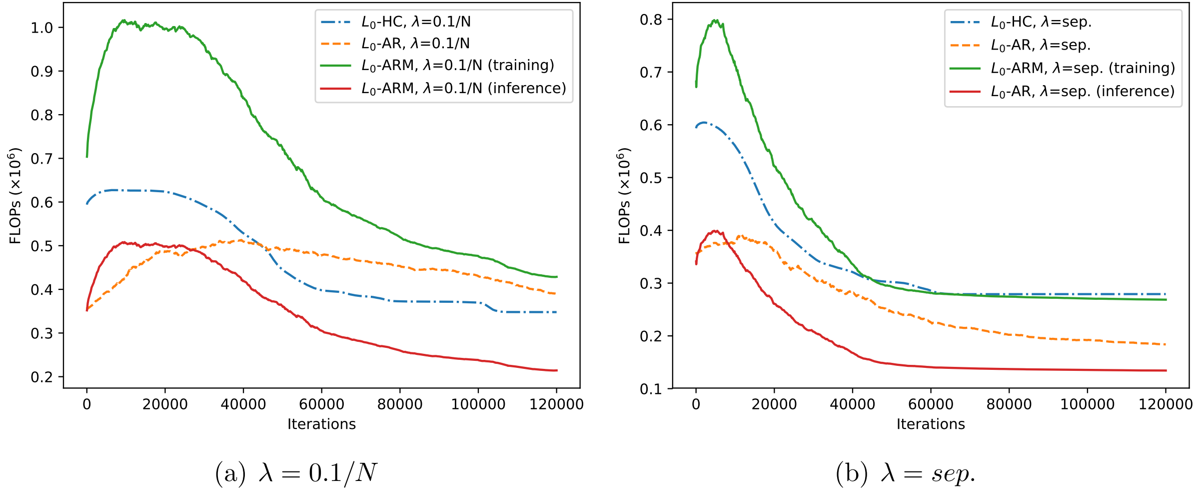
fairest comparison). In this case, $L_0$-ARM achieve the same accuracy (99.1%) on LeNet-5 with even sparser pruned architectures (95.52% vs. 91.1%). When separated $\lambda$s are considered ($\lambda$ sep.), since $L_0$-HC doesn't disclose the specific $\lambda$s for the last two fully-connected layers, we tune them by ourselves and find that $\lambda = (10, 0.5, 0.1, 10)/N$ yields the best performance. In this case, $L_0$-ARM achieves the highest prune rate (99.49% vs. 98.6%) with very similar accuracies (98.7% vs. 99.1%) on LeNet-5. Similar patterns are also observed on MLP. Regarding $L_0$-AR, although its performance is not as good as $L_0$-ARM, it's still very competitive to all the other methods. The advantage of $L_0$-AR over $L_0$-ARM is its lower computational complexity during training. As we discussed in Sec. 2.2, $L_0$-ARM needs one extra forward pass to estimate the gradient w.r.t. $\boldsymbol{\phi}$; for large DNN architectures, this extra cost can be significant.

To evaluate the training cost and network sparsity of different algorithms, we compare the prune rates of $L_0$-HC and $L_0$-AR(M) on LeNet-5 as a function of epoch in Fig. 2.4. Similarly, we compare the expected FLOPs of different algorithms as a function of epoch in Fig. 2.3. As we can see from Fig. 2.4, $L_0$-ARM yields much sparser network architectures over the whole training epochs, followed by $L_0$-AR and $L_0$-HC. The FLOPs vs. Epoch plots in Fig. 2.3 are more complicated. Because $L_0$-HC and $L_0$-AR only need one forward pass to compute gradient, they have the same expected FLOPs for training and inference. $L_0$-ARM needs two forward passes for training. Therefore, $L_0$-ARM is computationally more expensive during training (red curves), but it leads to sparser / more efficient architectures for inference (green curves), which pays off its extra cost in training.




(a) $\lambda = 0.1/N$ (b) $\lambda = sep.$

Figure 2.3: Comparison of prune rate of sparsified network as a function of epoch for different algorithms.




(a) $\lambda = 0.1/N$ (b) $\lambda = sep.$

Figure 2.4: Comparison of expected FLOPs as a function of epoch for different algorithms during training and inference.

### *2.4.3 CIFAR Experiments*

We further evaluate the performance of $L_0$-ARM and $L_0$-AR with Wide-ResNet Zagoruyko & Komodakis (2016a) on CIFAR-10 and CIFAR-100. Following Louizos et al. (2018b), we only

apply $L_0$ regularization on the first convolutional layer of each residual block, which allows us to incorporate $L_0$ regularization without further modifying residual block architecture.

Table 2.3 and 2.4 shows the performance comparison between $L_0$-AR(M) and three baseline methods. We find that $L_0$-HC cannot sparsify the Wide-ResNet architecture (prune rate 0%) [4], while $L_0$-ARM and $L_0$-AR prune around 50% of the parameters of the impacted subnet. As we activate 70% convolution filters in initialization, the around 50% prune rate is not due to initialization. We also inspect the histograms of $g(\boldsymbol{\phi})$: As expected, they are all split into two spikes around the values of 0 and 1, similar to the histograms shown in Fig. 2.2. In terms of accuracies, both $L_0$-ARM and $L_0$-AR achieve very similar accuracies as the baseline methods.

Table 2.3: Performance comparison of WRN on CIFAR-10.

| Network | Method | Pruned Architecture | Prune rate (%) | Accuracy (%) |
|---|---|---|---|---|
| WRN-28-10 CIFAR-10 | WRN | full model | 0 | 96.00 |
| | WRN-dropout | full model | 0 | 96.11 |
| | $L_0$-HC ($\lambda = 0.001/N$) | full model | 0 | 96.17 |
| | $L_0$-HC ($\lambda = 0.002/N$) | full model | 0 | 96.07 |
| | $L_0$-AR ($\lambda = 0.001/N$) | 83-77-83-88-169-167-153-165-324-323-314-329 | 49.49 | 95.58 |
| | $L_0$-ARM ($\lambda = 0.001/N$) | 74-86-83-83-164-145-167-153-333-333-310-330 | 49.46 | 95.68 |
| | $L_0$-AR ($\lambda = 0.002/N$) | 82-75-82-87-164-169-156-161-317-317-317-324 | **49.95** | 95.60 |
| | $L_0$-ARM ($\lambda = 0.002/N$) | 75-72-78-78-157-165-131-162-336-325-331-343 | 49.63 | 95.70 |

Each experiment was run five times and the median (in terms of accuracy) is reported. All the baseline results are taken from the corresponding papers. Only the architectures of pruned layers are shown. The results for WRN and WRN-dropout are taken from Zagoruyko & Komodakis (2016a).

To evaluate the training and inference costs of different algorithms, we compare the

[4]This was also reported recently in the appendix of Gale et al. (2019b), and can be easily reproduced by using the open-source implementation of $L_0$-HC [3].

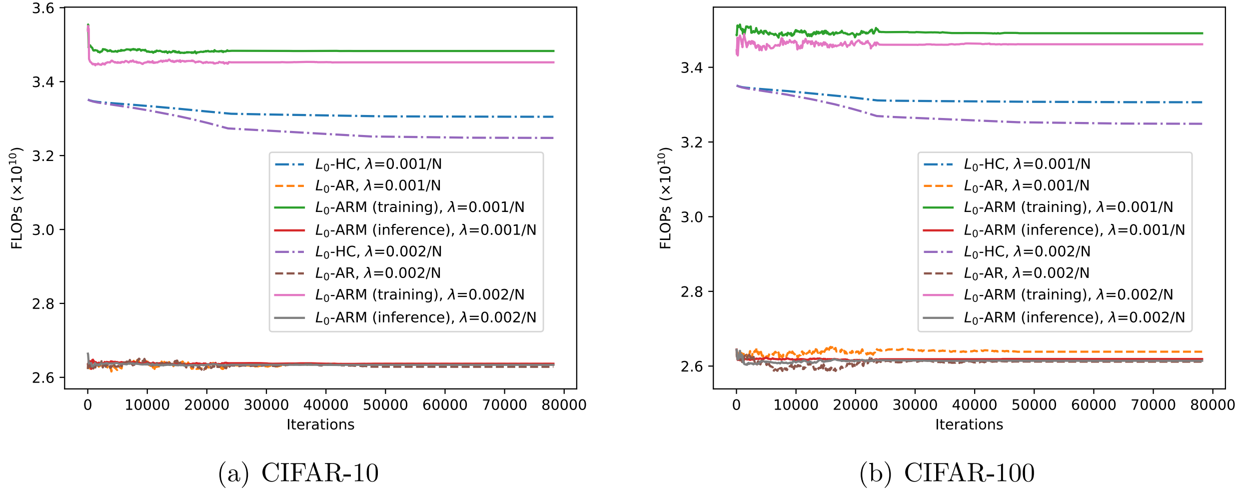
Table 2.4: Performance comparison of WRN on CIFAR-100.

| Network | Method | Pruned Architecture | Prune rate (%) | Accuracy (%) |
|---|---|---|---|---|
| WRN-28-10 CIFAR-100 | WRN | full model | 0 | 78.82 |
| | WRN-dropout | full model | 0 | 81.15 |
| | $L_0$-HC ($\lambda = 0.001/N$) | full model | 0 | 81.25 |
| | $L_0$-HC ($\lambda = 0.002/N$) | full model | 0 | 80.96 |
| | $L_0$-AR ($\lambda = 0.001/N$) | 78-78-79-85-168-168-162-164-308-326-319-330 | 49.37 | 80.50 |
| | $L_0$-ARM ($\lambda = 0.001/N$) | 75-83-80-58-172-156-160-165-324-311-313-318 | 50.51 | 80.74 |
| | $L_0$-AR ($\lambda = 0.002/N$) | 75-76-72-80-158-158-137-168-318-295-327-324 | 50.93 | 80.09 |
| | $L_0$-ARM ($\lambda = 0.002/N$) | 81-74-77-73-149-157-156-152-299-332-305-325 | **50.78** | 80.56 |

Each experiment was run five times and the median (in terms of accuracy) is reported. All the baseline results are taken from the corresponding papers. Only the architectures of pruned layers are shown. The results for WRN and WRN-dropout are taken from Zagoruyko & Komodakis (2016a).

expected FLOPs of $L_0$-HC and $L_0$-AR(M) on CIFAR-10 and CIFAR-100 as a function of iteration in Fig. 2.5. Similar to Fig. 2.4 and 2.3, $L_0$-ARM is more computationally expensive for training, but leads to sparser / more efficient architectures for inference, which pays off its extra cost in training. It's worth to emphasize that for these experiments $L_0$-AR has the lowest training FLOPs and inference FLOPs (since only one forward pass is needed for training and inference), while achieving very similar accuracies as the baseline methods (Table 2.3 and 2.4).

Finally, we compare the test accuracies of different algorithms as a function of epoch on CIFAR-10, with the results shown in Fig. 2.6. We apply the exact same hyper-parameters of $L_0$-HC to $L_0$-AR(M). As $L_0$-AR(M) prunes around 50% parameters during training (while $L_0$-HC prunes 0%), the test accuracies of the former are lower than the latter before convergence, but all the algorithms yield very similar accuracies after convergence, demonstrating




(a) CIFAR-10 (b) CIFAR-100

Figure 2.5: Comparison of expected FLOPs as a function of iteration during training and inference.

the effectiveness of $L_0$-AR(M).

## 2.5 Conclusion

We propose $L_0$-ARM, an unbiased and low-variance gradient estimator, to sparsify network architectures. Compared to $L_0$-HC Louizos et al. (2018b) and other state-of-the-art sparsification algorithms, $L_0$-ARM demonstrates superior performance of sparsifying network architectures while retaining almost the same accuracies of the baseline methods. Extensive experiments on multiple public datasets and multiple network architectures validate the effectiveness of $L_0$-ARM. Overall, $L_0$-ARM yields the sparsest architectures and the lowest inference FLOPs for all the networks considered with very similar accuracies as the baseline methods.

As for future extensions, we plan to design better (possibly non-antithetic) parametric

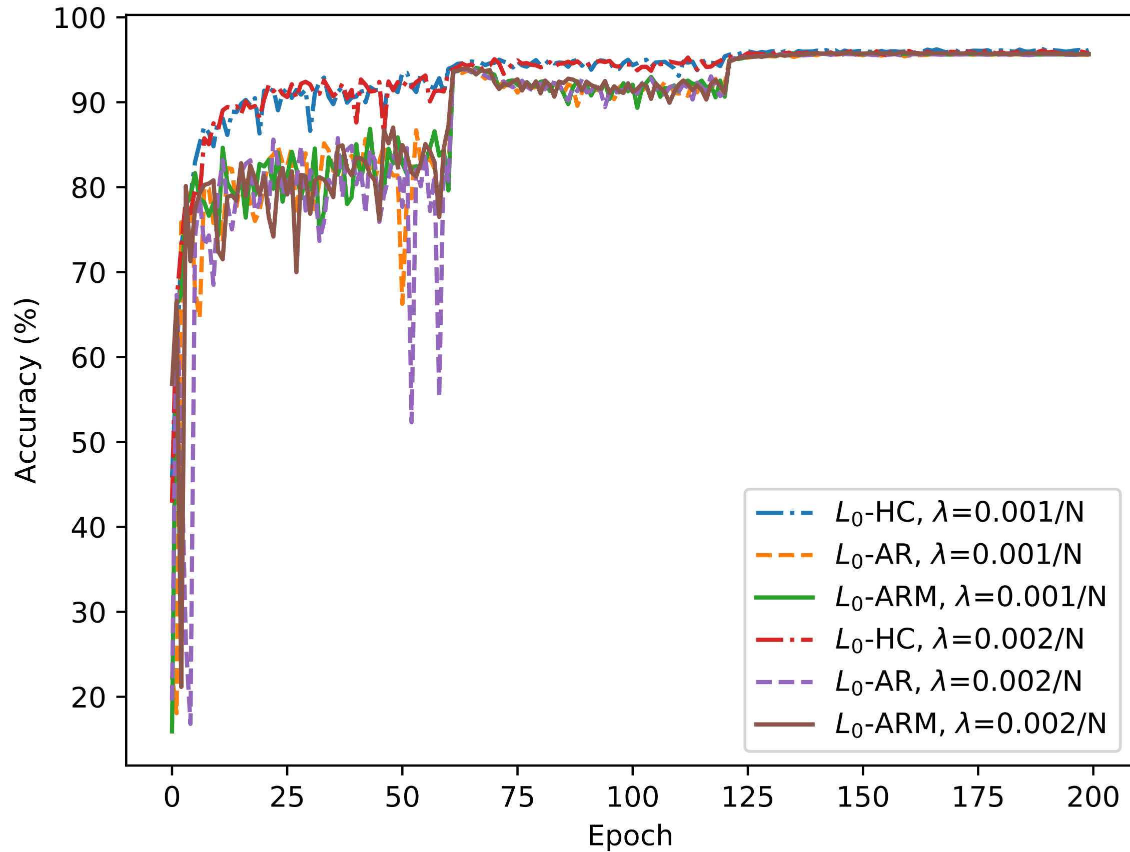



Figure 2.6: Comparison of test accuracy as a function of epoch for different algorithms on CIFAR-10.

function $g(\phi)$ to improve the sparsity of solutions. We also plan to investigate more efficient algorithm to evaluate $L_0$-ARM gradient (2.2.4) by utilizing the antithetic structure of two forward passes.

# CHAPTER 3

## Neural Plasticity Networks

© [2021] IEEE. Reprinted, with permission, from [Yang Li, Shihao Ji, Neural Plasticity Networks, 2021 International Joint Conference on Neural Networks (IJCNN), 18 July]

In this chapter, we will present Neural Plasticity Networks (NPNs). Neural plasticity is an important functionality of human brain, in which number of neurons and synapses can shrink or expand in response to stimuli throughout the span of life. We model this dynamic learning process as an $L_0$-norm regularized binary optimization problem, in which each unit of a neural network (e.g., weight, neuron or channel, etc.) is attached with a stochastic binary gate, whose parameters determine the level of activity of a unit in the whole network. Our proposed algorithm unifies network sparsification and network expansion in an end-to-end training pipeline, in which number of neurons and synapses can shrink or expand as needed to solve a given learning task.

Compared with network sparsification, network expansion is less explored. This paradigm is in the opposite to network sparsification, but might be more desirable because (1) we don't need to set an upper-bound on the network capacity (e.g., number of weights, neurons or channels, etc.) to start with, and the network can shrink or expand as needed for a given task; (2) it's computationally more efficient to train a small network and expand it to a larger one as redundant neurons are less likely to emerge during the whole training process; and (3) network expansion is more biologically plausible than network sparsification according to our current understanding to human brain development Stiles & Jernigan (2010).

NPN is built on top of our previous $L_0$-ARM algorithm Li & Ji (2019). However, the original $L_0$-ARM algorithm only explores network sparsification, in which it demonstrates state-of-the-art performance at pruning networks, while here we extend this framework to network expansion. On the algorithmic side, we further investigate the Augment-Reinforce-Merge (ARM) Yin & Zhou (2019), a recently proposed unbiased gradient estimator for binary latent variable models. We show that due to the flexibility of ARM, many smooth or non-smooth parametric functions, such as scaled sigmoid or hard sigmoid, can be used to parameterize the $L_0$-norm regularized binary optimization problem and the unbiasness of the ARM estimator is retained, while a closly related hard concrete estimator Louizos et al. (2017) has to rely on the hard sigmoid function for binary optimization. It is this difference that entails NPN the capability of shrinking or expanding network capacity as needed for a given task. We also introduce a learning stage scheduler for NPN and demonstrate that many training stages of network sparsification and expansion, such as pre-training, sparsification/expansion and fine-tuning, can be modulated by a single parameter $k$ seamlessly; along the way, we also give a new interpretation of dropout Srivastava et al. (2014). Extensive experiments on synthetic dataset and multiple public datasets demonstrate the superior performance of NPNs for network sparsification and network expansion with fully connected layers, convolutional layers and skip connections. Our experiments show that both network sparsification and network expansion can converge to similar network capacities with similar accuracies even though they are initialized with networks of different sizes. To the best of our knowledge, this is the first learning framework that unifies network sparsification and

network expansion in an end-to-end training pipeline modulated by a single parameter.

The remainder of this chapter is organized as follows. In Sec. 3.1 we describe the $L_0$-norm regularized empirical risk minimization for NPN and its solver $L_0$-ARM Li & Ji (2019) for network sparsification. A new learning stage scheduler for NPN is introduced in Sec. 3.2. We then extend NPN to network expansion in Sec. 3.3, followed by the related work in Sec. 3.4. Experimental results are presented in Sec. 3.5. Conclusions and future work are discussed in Sec. 3.6.

## 3.1 Neural Plasticity Networks: Formulation

Our Neural Plasticity Network (NPN) is built on the basic framework of $L_0$-ARM Li & Ji (2019), which was proposed in our previous work for network sparsification. We extend $L_0$-ARM to network expansion, and unify network sparsification and expansion in an end-to-end training pipeline. For the sake of clarity, we first introduce NPN in the context of network sparsification, and later extend it to network expansion. The formulation below largely follows that of $L_0$-ARM Li & Ji (2019).

Given a training set $D = \{(\boldsymbol{x}_i, y_i), i = 1, 2, \cdots, N\}$, where $\boldsymbol{x_i}$ denotes the input and $y_i$ denotes the target, a neural network is a function $h(\boldsymbol{x}; \boldsymbol{\theta})$ parametrized by $\boldsymbol{\theta}$ that fits to the training data $D$ with the goal of achieving good generalization to unseen test data. To optimize $\boldsymbol{\theta}$, typically a regularized empirical risk is minimized, which contains two terms – a data loss over training data and a regularization loss over model parameters. Empirically, the regularization term can be weight decay or Lasso, i.e., the $L_2$ or $L_1$ norm of model

parameters.

Intuitively, network sparsification or expansion is a model selection problem, in which a suitable model capacity is selected for a given learning task. In this problem, how to measure model complexity is a core issue. The Akaike Information Criterion (AIC) Akaike (1998) and the Bayesian Information Criterion (BIC) Schwarz (1978), well-known model selection criteria, measure model complexity by counting number of non-zero parameters. Since the $L_2$ or $L_1$ norm only imposes shrinkage on large values of $\boldsymbol{\theta}$, the resulting model parameters $\boldsymbol{\theta}$ are often manifested by smaller magnitudes but none of them are exact zero. Therefore, the $L_2$ or $L_1$ norm is not suitable for measuring model complexity. A more appealing alternative is the $L_0$ norm of model parameters as it measures *explicitly* the number of non-zero parameters, which is the *exact* model complexity measured by AIC and BIC. With the $L_0$ regularization, the empirical risk objective can be written as

$$\mathcal{R}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(h(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i\right) + \lambda\|\boldsymbol{\theta}\|_0 \tag{3.1.1}$$

where $\mathcal{L}(\cdot)$ denotes the data loss over training data $D$, such as the cross-entropy loss for classification or the mean squared error (MSE) for regression, and $\|\boldsymbol{\theta}\|_0$ denotes the $L_0$-norm over model parameters, i.e., the number of non-zero weights, and $\lambda$ is a regularization hyperparameter that balances between data loss and model complexity. For network sparsification, minimizing of Eq. 3.1.1 will drive the redundant or insignificant weights to be exact zero and thus pruned away. For network expansion, adding additional neurons will increase model complexity (the second term) but potentially can reduce data loss (the first term) and therefore the total loss. Thus, we will use Eq. 3.1.1 as our guiding principle for sparsifying

or expanding a network.

To represent a sparsified network, we attach a binary random variable $z$ to each element of model parameters $\boldsymbol{\theta}$. Therefore, we can reparameterize the model parameters $\boldsymbol{\theta}$ as an element-wise product of non-zero parameters $\widetilde{\boldsymbol{\theta}}$ and binary random variables $\boldsymbol{z}$:

$$\boldsymbol{\theta} = \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}, \tag{3.1.2}$$

where $\boldsymbol{z} \in \{0, 1\}^{|\boldsymbol{\theta}|}$, and $\odot$ denotes the element-wise product. As a result, Eq. 3.1.1 can be rewritten as:

$$\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left( h\big(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}\big), y_i \right) + \lambda \sum_{j=1}^{|\widetilde{\boldsymbol{\theta}}|} \mathbf{1}_{[z_j \neq 0]}, \tag{3.1.3}$$

where $\mathbf{1}_{[c]}$ is an indicator function that is 1 if the condition $c$ is satisfied, and 0 otherwise. Note that both the first term and the second term of Eq. 3.1.3 are not differentiable w.r.t. $\boldsymbol{z}$. Therefore, further approximations need to be considered.

Fortunately, we can approximate Eq. 3.1.3 through an inequality from stochastic variational optimization Bird et al. (2018). Specifically, given any function $\mathcal{F}(\boldsymbol{z})$ and any distribution $q(\boldsymbol{z})$, the following inequality holds

$$\min_{\boldsymbol{z}} \mathcal{F}(\boldsymbol{z}) \leq \mathbb{E}_{\boldsymbol{z} \sim q(\boldsymbol{z})}[\mathcal{F}(\boldsymbol{z})], \tag{3.1.4}$$

i.e., the minimum of a function is upper bounded by the expectation of the function. With this result, we can derive an upper bound of Eq. 3.1.3 as follows.

Since $z_j, \forall j \in \{1, \cdots, |\boldsymbol{\theta}|\}$ is a binary random variable, we assume $z_j$ is subject to a

Bernoulli distribution with parameter $\pi_j \in [0,1]$, i.e. $z_j \sim \text{Ber}(z;\pi_j)$. Thus, we can upper bound $\min_{\boldsymbol{z}} \mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z})$ by the expectation

$$\begin{aligned}\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\pi}) &= \mathbb{E}_{\boldsymbol{z}\sim\text{Ber}(\boldsymbol{z};\boldsymbol{\pi})}\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{z}) \\ &= \mathbb{E}_{\boldsymbol{z}\sim\text{Ber}(\boldsymbol{z};\boldsymbol{\pi})}\left[\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(h(\boldsymbol{x}_i;\widetilde{\boldsymbol{\theta}}\odot\boldsymbol{z}), y_i\right)\right] + \lambda\sum_{j=1}^{|\widetilde{\boldsymbol{\theta}}|}\pi_j.\end{aligned} \tag{3.1.5}$$

As we can see, now the second term is differentiable w.r.t. the new model parameters $\boldsymbol{\pi}$, while the first term is still problematic since the expectation over a large number of binary random variables $\boldsymbol{z} \in \{0,1\}^{|\theta|}$ is intractable, so is its gradient.

To minimize Eq. 3.1.5, $L_0$-ARM utitlizes the Augment-Reinforce-Merge (ARM) Yin & Zhou (2019), an unbiased gradient estimator, to this stochastic binary optimization problem. Specifically,

**Theorem 1** (ARM) Yin & Zhou (2019). For a vector of $V$ binary random variables $\boldsymbol{z} = (z_1, \cdots, z_V)$, the gradient of

$$\mathcal{E}(\boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{z}\sim\prod_{v=1}^{V}\text{Ber}(z_v;g(\phi_v))}[f(\boldsymbol{z})] \tag{3.1.6}$$

w.r.t. $\boldsymbol{\phi} = (\phi_1, \cdots, \phi_V)$, the logits of the Bernoulli distribution parameters, can be expressed as

$$\begin{aligned}\nabla_{\phi}\mathcal{E}(\phi) =& \mathbb{E}_{\boldsymbol{u}\sim\prod_{v=1}^{V}\text{Uniform}(u_v;0,1)}\Big[\big(f(\mathbf{1}_{[\boldsymbol{u}>g(-\phi)]})- \\ & f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})\big)(\boldsymbol{u}-1/2)\Big],\end{aligned} \tag{3.1.7}$$

where $\mathbf{1}_{[\boldsymbol{u}>g(-\boldsymbol{\phi})]} := \mathbf{1}_{[u_1>g(-\phi_1)]}, \cdots, \mathbf{1}_{[u_V>g(-\phi_V)]}$ and $g(\phi) = \sigma(\phi) = 1/(1+\exp(-\phi))$ is the sigmoid function.

Parameterizing $\pi_j \in [0,1]$ as $g(\phi_j)$, we can rewrite Eq. 3.1.5 as

$$\begin{aligned} \widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) &= \mathbb{E}_{\boldsymbol{z}\sim \text{Ber}(\boldsymbol{z};g(\boldsymbol{\phi}))}\left[f(\boldsymbol{z})\right] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} g(\boldsymbol{\phi}_j) \\ &= \mathbb{E}_{\boldsymbol{u}\sim \text{Uniform}(\boldsymbol{u};0,1)}\left[f(\mathbf{1}_{[\boldsymbol{u}<\boldsymbol{g}(\boldsymbol{\phi})]})\right] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} g(\boldsymbol{\phi}_j), \end{aligned} \tag{3.1.8}$$

where $f(\boldsymbol{z}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\left(h(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}), y_i\right)$. From Theorem 1, we can evaluate the gradient of Eq. 3.1.8 w.r.t. $\boldsymbol{\phi}$ by

$$\begin{aligned} \nabla_{\boldsymbol{\phi}} \widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u}\sim \text{Uniform}(\boldsymbol{u};0,1)}\Big[\big(f(\mathbf{1}_{[\boldsymbol{u}>g(-\boldsymbol{\phi})]}) - \\ f(\mathbf{1}_{[\boldsymbol{u}<g(\boldsymbol{\phi})]})\big)(\boldsymbol{u} - 1/2)\Big] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} \nabla_{\phi_j} g(\phi_j), \end{aligned} \tag{3.1.9}$$

which is an unbiased and low variance estimator as demonstrated in Yin & Zhou (2019).

**Choice of $g(\phi)$**

Theorem 1 of ARM defines $g(\phi) = \sigma(\phi)$, where $\sigma(\cdot)$ is the sigmoid function. For the purpose of network sparsification and expansion, we find that this parametric function isn't very effective due to its fixed rate of transition between values 0 and 1. Thanks to the flexibility of ARM, we have a large freedom to design this parametric function $g(\phi)$. Apparently, it's straightforward to generalize Theorem 1 for any parametric functions (smooth or

non-smooth) as long as $g : \mathcal{R} \rightarrow [0,1]$ and $g(-\phi) = 1 - g(\phi)$[1]. Example parametric functions that work well in our experiments are the scaled sigmoid function

$$g_{\sigma_k}(\phi) = \sigma(k\phi) = \frac{1}{1 + \exp(-k\phi)}, \tag{3.1.10}$$

and the centered-scaled hard sigmoid

$$g_{\bar{\sigma}_k}(\phi) = \min(1, \max(0, \frac{k}{7}\phi + 0.5)), \tag{3.1.11}$$

where 7 is introduced such that $g_{\bar{\sigma}_1}(\phi) \approx g_{\sigma_1}(\phi) = \sigma(\phi)$. See Fig. 3.1 for some example plots of $g_{\sigma_k}(\phi)$ and $g_{\bar{\sigma}_k}(\phi)$ with different $k$s. Empirically, we find that $k = 7$ works well for network sparsification, and $k = 0.5$ for network expansion. More on this will be discussed when we present results.

One important difference between the hard concrete estimator from Louizos et al. Louizos et al. (2018b) and $L_0$-ARM is that the hard concrete estimator has to rely on the hard sigmoid gate to zero out some parameters during training (a.k.a. conditional computation Bengio et al. (2013)), while $L_0$-ARM achieves conditional computation naturally by sampling from the Bernoulli distribution, parameterized by $g(\phi)$, where $g(\phi)$ can be any parametric function (smooth or non-smooth) as shown in Fig. 3.1. The consequence of using the hard sigmoid gate is that once a unit is pruned, the corresponding gradient will be always zero due to the

[1]The second condition is not necessary. But for simplicity, we will impose this condition to select parametric function $g(\phi)$ that is antithetic. Designing $g(\phi)$ without this constraint could be a potential area that is worthy of further investigation.

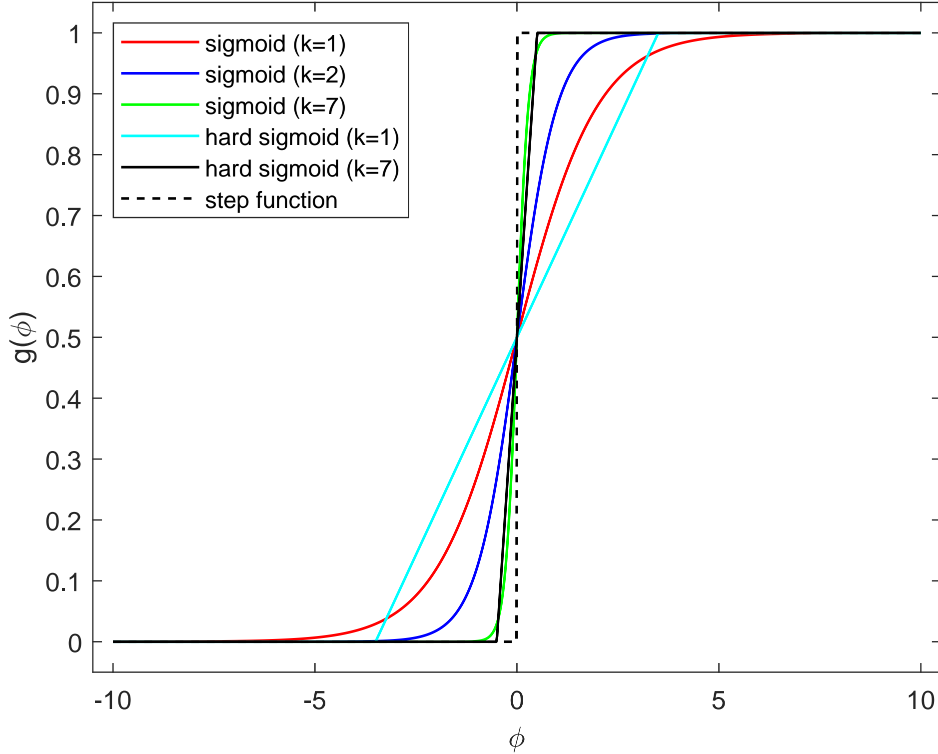



Figure 3.1: The plots of $g(\phi)$ with different $k$ for sigmoid and hard sigmoid functions.

exact 0 gradient at the left tail of the hard sigmoid gate (see Fig. 3.1) and therefore it can never be reactivated in the future. To mitigate this issue of the hard concrete estimator, $L_0$-ARM can utilize the scaled sigmoid gate (3.1.10), which has non-zero gradient everywhere $(-\infty, \infty)$, and therefore a unit can be activated or deactivated freely and thus be plastic.

## 3.2 Learning Stage Scheduler

As far as we know, all network sparsification algorithms either operate in a three-stage of pre-training, sparsification, and fine-tuning Han et al. (2016, 2015); Wen et al. (2016a) or only have one sparsification stage from scratch Louizos et al. (2018b). It has been shown that the three-stage sparsification leads to better predictive accuracies than the one-stage

alternatives Lee et al. (2018). To support this three-stage learning process, previous methods Han et al. (2016, 2015); Wen et al. (2016a) however manage this tedious process manually. Thanks to the flexibility of NPN, we can modulate these learning stages by simply adjusting $k$ of the $g_k(\phi)$ function at different training stages. Along the way, we also discover a new interpretation of dropout Srivastava et al. (2014).

### *3.2.1 Dropout as* $k = 0$

When $k = 0$, it is readily to verify that $g_{\sigma_k}(\phi) = g_{\bar{\sigma}_k}(\phi) = 0.5$, and the objective function (3.1.8) is degenerated to

$$\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u} \sim \text{Uniform}(\boldsymbol{u};0,1)} \left[ f(\mathbf{1}_{[\boldsymbol{u}<\mathbf{0.5}]}) \right] + \lambda |\widetilde{\theta}|/2, \tag{3.2.1}$$

which is in fact the standard dropout with a dropout probability of 0.5 Srivastava et al. (2014). Note that the value of 0.5 is due to the artifact of the antithetic constraint on the parametric function $g(\phi)$. As we discussed in Sec. 3.1, this constraint isn't necessary and we have freedom of designing $g(0) = c$ with $c \in [0, 1]$, which corresponds to any dropout probability of the standard dropout. From this point of view, dropout is just a special case of NPN when $k = 0$, and this is a new interpretation of dropout.

### *3.2.2 Pre-training as $k = \infty$ at the beginning of NPN training*

At the beginning of NPN training, we initialize all $\phi$'s to some positive values, e.g., $\phi > 0.1$. If we set $k = \infty$, then $g_{\sigma_\infty}(\phi) = g_{\bar{\sigma}_\infty}(\phi) = 1$ and the objective function (3.1.8) becomes

$$\widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u} \sim \text{Uniform}(\boldsymbol{u};0,1)} \left[ f(\mathbf{1}_{[\boldsymbol{u}<\mathbf{1}]}) \right] + \lambda |\widetilde{\theta}|, \tag{3.2.2}$$

which corresponds to the standard training of DNNs with all neurons activated. Moreover, the gradient w.r.t. $\phi$ is degenerated to

$$\begin{aligned} \nabla_{\boldsymbol{\phi}} \widehat{\mathcal{R}}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\phi}) = \mathbb{E}_{\boldsymbol{u} \sim \text{Uniform}(\boldsymbol{u};0,1)} \Big[ \big( f(\mathbf{1}_{[\boldsymbol{u}>0]}) \\ - f(\mathbf{1}_{[\boldsymbol{u}<1]}) \big) (\boldsymbol{u} - 1/2) \Big] + \lambda \sum_{j=1}^{|\widetilde{\theta}|} \nabla_{\phi_j} 1 = 0, \end{aligned} \tag{3.2.3}$$

such that $\phi$ will not be updated during the training and the architecture is fixed. This corresponds to the pre-training of a network from scratch.

### *3.2.3 Fine-tuning as $k = \infty$ at the end of NPN training*

At the end of NPN training, the histogram of $g(\phi)$ is typically split to two spikes of values around 0 and 1 as demonstrated in $L_0$-ARM Li & Ji (2019). If we set $k = \infty$, then the values of $g(\phi)$ will be exactly 0 or 1. In this case, the gradient w.r.t. $\phi$ is zero, the neurons with $g(\phi) = 1$ are activated and the neurons with $g(\phi) = 0$ are deactivated. This corresponds to the case of fine-tuning a fixed architecture without the $L_0$ regularization.

#### *3.2.4 Modulating learning stages by $k$*

As discussed above, we can now integrate the three-stage of pre-training, sparsification and fine-tuning into one end-to-end pipeline, modulated by a single parameter $k$. At the beginning of the training, we set $k = \infty$ to pre-train a network from scratch. Upon convergence, we can set $k$ to some small values (e.g., $k = 7$) to enable the $L_0$ regularized optimization for network sparsification. After the convergence, we can set $k = \infty$ again to fine-tune the final learned architecture without the $L_0$ regularization. To the best of our knowledge, there is no other network sparsification algorithm that supports this three-stage training in a native end-to-end pipeline. As an analogy to the common learning rate scheduler, we call $k$ as a learning stage scheduler. We will demonstrate this when we present results.

### 3.3 Network Expansion

So far we have described NPN in the context of network sparsification. Thanks to the flexibility of $L_0$-ARM, it is straightforward to extend it to network expansion. Instead of starting from a large network for pruning, we can expand a small network by adding neurons during training, an analogy of growing a brain from small to large. Specifically, given the level of activity of a neuron is determined by its $\phi$, a new neuron can be added to the network with a large $\phi$ such that it will be activated in future training epochs. If this neuron is useful at reducing the $L_0$-regularized loss function (3.1.8), its $\phi$ value will be increased such that it will be activated more often in the future; otherwise, it will be gradually deactivated and pruned away in the future. At each training iteration, we add a new neuron to a layer if (a)

the validation loss is lower than its previous value when a neuron was added last time, and (b) all neurons in the layer are activated (i.e., no redundant neurons in the layer). The network expansion will terminate when the $L_0$-regularized loss plateaus or some added neurons are deactivated due to the $L_0$-norm regularization. This network expansion procedure is detailed in Algorithm 1.

**Algorithm 1** Network Expansion

**Require:** number of iterations $T$, deep network $f$, number of layers $L$, $loss_{old} = +\infty$, $loss_{val} = 0$

```
for t = 1 to T do
    Train f on a mini-batch of training examples
    loss_val = Validate f on validation dataset
    if loss_val < loss_old then
        for l = 1 to L do
            if all neurons in layer l are activated then
                Add a new neuron to layer l
            end if
        end for
        loss_old = loss_val
    end if
end for
```

Thanks to the learning stage scheduler discussed in Sec. 3.2, we can simulate network expansion easily by manipulating $k$. Specifically, we can initialize a very large network to represent an upper-bound on network capacity. To start with a small network, we randomly select a small number of neurons and initialize the corresponding $\phi$s to large positive values and set all the remaining $\phi$'s to large negative values, such that only a small portion of the neurons will be activated while the remaining neurons are in a hibernation mode. Since they will not be activated, these hibernating neurons consume no computation resources. To pretrain the initial small network, we can train NPN with $k = \infty$. Upon convergence, we

can randomly activate a few hibernating neurons (under the conditions discussed above) and switch $k$ to a small value (e.g., $k = 0.5$). Along with the original neurons, we can optimize the expanded network by reducing the $L_0$-regularized loss. In such a way, we can readily simulate network expansion. Upon the network expansion terminates, we can set $k = \infty$ to fine-tune the final network architecture. In the experiments, we will resort to this approach to simulate network expansion.

## 3.4 Related Work

Our NPN has a built-in support to network sparsification, network expansion, and can automatically determine an appropriate network capacity for a given learning task. In this section, we review related works in these areas.

### *3.4.1 Network Sparsification*

Driven by the widespread applications of DNNs in resource-limited embedded systems, recently there has been an increasing interest in network sparsification Han et al. (2015, 2016); Wen et al. (2016a); Li et al. (2016); Louizos et al. (2017); Molchanov et al. (2017); Neklyudov et al. (2017); Louizos et al. (2018b); Li & Ji (2019). One of the earliest sparsification methods is to prune the redundant weights based on the magnitudes LeCun et al. (1990), which is proved to be effective in modern CNNs Han et al. (2015). Although weight sparsification is able to compress networks, it can barely improve computational efficiency due to unstructured sparsity. Therefore, magnitude-based group sparsity is proposed Wen et al. (2016a); Li et al. (2016), which can prune networks while reducing computation cost significantly.

These works are mainly based on the $L_2$ or $L_1$ regularization to penalize the magnitude of weights. A more appealing approach is based on the $L_0$ regularization Louizos et al. (2018b); Li & Ji (2019) as this corresponds to the well-known model selection criteria such as AIC Akaike (1998) and BIC Schwarz (1978). Our NPN is built on the basic framework of $L_0$-ARM Li & Ji (2019) and extend it for network expansion. In addition, as far as we know almost all the network sparsification algorithms Han et al. (2015, 2016); Wen et al. (2016a); Li et al. (2016) usually proceed in three stages manually: pretrain a full network, prune the redundant weights or filters, and fine-tune the pruned network. In contrast, our NPN can support this three-stage training by simply adjusting the learning stage scheduler $k$ at different stages in an end-to-end fashion.

### *3.4.2 Neural Architecture Search*

Another closely related area is neural architecture search Zoph & Le (2017); Zoph et al. (2018); Real et al. (2019) that searches for an optimal network architecture for a given learning task. It attempts to determine number of layers, types of layers, layer configurations, different activation functions, etc. Given the extremely large search space, typically reinforcement learning algorithms are utilized for efficient implementations. Our NPN can be categorized as a subset of neural architecture search in the sense that we start with a fixed architecture and aim to determine an optimal capacity (e.g., number of weights, neurons or channels) of a network.

### *3.4.3 Dynamic Network Expansion*

Compared to network sparsification, network expansion is a relatively less explored area. There are few existing works that can dynamically increase the capacity of network during training. For example, DNC Ash (1989) sequentially adds neurons one at a time to the hidden layers of network until the desired approximation accuracy is achieved. Zhou et al. (2012) proposes to train a denoising autoencoder (DAE) by adding in new neurons and later merging them with other neurons to prevent redundancy. For convolutional networks, Wang et al. (2017) proposes to widen or deepen a pretrained network for better knowledge transfer. Recently, a boosting-style method named AdaNet Cortes et al. (2017) is used to adaptively grow the structure while learning the weights. However, all these approaches either only add neurons or add/remove neurons manually. In contrast, our NPN can add or remove (deactivate) neurons during training as needed without human intervention, and is an end-to-end unified framework for network sparsification and expansion.

## 3.5 Experimental Results

We evaluate the performance of NPNs on multiple public datasets with different network architectures for network sparsification and network expansion. Specifically, we illustrate how NPN evolves on a synthetic "moons" dataset Miyato et al. (2018) with a 2-hidden-layer MLP. We also demonstrate LeNet5-Caffe[2] on the MNIST dataset Lecun et al. (1998), and ResNet56 He et al. (2016a) on the CIFAR10 and CIFAR100 datasets Krizhevsky & Hinton (2009). Similar to $L_0$-ARM Li & Ji (2019) and $L_0$-HC Louizos et al. (2017), to achieve

[2] `https://github.com/BVLC/caffe/tree/master/examples/mnist`

computational efficiency, only neuron-level (instead of weight-level) sparsification/expansion is considered, i.e., all weights of a neuron or filter are either pruned from or added to a network altogether. For the comparison to the state-of-the-art network sparsification algorithms Louizos et al. (2018b); Molchanov et al. (2017); Louizos et al. (2017), we refer the readers to $L_0$-ARM Li & Ji (2019) for more details since NPN is an extension of $L_0$-ARM for network sparsification and expansion.

As discussed in Sec. 3.1, each neuron in an NPN is attached with a Bernoulli random variable parameterized by $g(\phi)$. Therefore, the level of activity of a neuron is determined by the value of $\phi$. To initialize an NPN, in our experiments we activate a neuron by setting $\phi = 3/k$. Since $g_{\sigma_k}(\phi) = \sigma(3) \approx 0.95$, this means that the corresponding neuron has a 95% probability of being activated. Similarly, we set $\phi = -3/k$ to deactivate a neuron with a 95% probability.

As discussed in Sec. 3.2, all of our experiments are performed in three stages: (1) pre-training, (2) sparsification/expansion, and (3) fine-tuning, in an end-to-end training pipeline modulated by parameter $k$. In pre-training and fine-tuning stages, we set $k = 5000$ (as a close approximation to $k = \infty$) to train an NPN with a fixed architecture. In sparsification/-expansion stage, we set $k$ to a small value to allow NPNs to search for a suitable network capacity freely.

The final architecture of a network is influenced significantly by two hyperparameters: (1) the regularization strength $\lambda$, and (2) $k$ of the $g_{\sigma_k}(.)$ function, which determine how aggressively to sparsify or expand a network. Typically, a positive $\lambda$ is used both for spar-

sification and expansion. However, in some expansion experiments, we notice that $\lambda = 0$ is beneficial because $\lambda = 0$ essentially encourages more neurons to be activated, which is important for network expansion to achieve competitive accuracies. For the hyperparameter $k$ used in stage 2, in all of our experiments we set $k = 7$ for sparsification and $k = 0.5$ for expansion. The reason that different $k$s are used is because for expansion we need to encourage the network to grow and a small $k$ is more amenable to keep neurons activated. The rest of hyperparameters of NPNs are determined via cross validation on the validation datasets.

Unless specifically noted, we use the Adam optimizer Kingma & Ba (2015) with an initial learning rate of 0.001. Our experiments are performed on NVIDIA Titan-Xp GPUs. Our source code is available at `https://github.com/leo-yangli/npns`.

### *3.5.1 Synthetic Dataset*

To demonstrate that NPNs can adapt their capacities for a learning task, we visualize the learning process of NPNs on a synthetic "moons" dataset Miyato et al. (2018) for network sparsification and network expansion. The "moons" dataset contains 1000 data points distributed in two moon-shaped clusters for binary classification. We randomly pick 500 data points for training and use the rest 500 data points for test. Two different MLP architectures are used for network sparsification and network expansion, respectively. For network sparsification, we train an MLP with 2 hidden layers of 100 and 80 neurons, respectively. The input layer has 2 neurons corresponding to the 2-dim coordinates of each data point, which is transformed to a 100-dim vector by a fixed matrix. The output layer has two

neurons for binary classification. The overall architecture of the MLP is 2-100 (fixed)-80-2 with the first weight matrix fixed, and the overall number of trainable model parameters is 8,160 (excluding biases for clarity). The binary gates are attached to the outputs of the two hidden layers for sparsification. For the expansion experiment, we start an MLP with a very small architecture of 2-100 (fixed)-3-2. Initially, only three neurons at each hidden layer are activated, and therefore the total number of trainable model parameters is 15 (excluding biases for clarity). Apparently, the first MLP is overparameterized for this synthetic binary classification task, while the second MLP is too small and doesn't have enough capacity to solve the classification task with a high accuracy.

To visualize the learning process of NPNs, in Fig. 3.2 we plot the decision boundaries and confidence contours of the NPN-sparsified MLP and NPN-expanded MLP on the test set of "moons". We pick three snapshots from each experiment. The evolution of the decision boundaries of the NPN-expanded MLP is shown in Fig. 3.2 (a, b, c). At the end of pre-training (a. epoch 99), the decision boundary is mostly linear and the capacity of the network is obviously not enough. During the stage 2 expansion (b. epoch 199), more neurons are added to the network and the decision boundary becomes more expressive as manifested by a piece-wise linear function, and at the same time the accuracy is significantly improved to about 96%. At the end of stage 3 fine-tuning (c. epoch 1999), the accuracy reaches 99.2% with more neurons being added. Similarly, Fig. 3.2 (d, e, f) demonstrates the evolution of the decision boundaries of NPN-sparsified MLP on the test set. The model achieves an accuracy of 99.2% at the end of stage 1 pre-training (d. epoch 499). Then 47.6% of weights are

pruned without any accuracy loss during stage 2 sparsification (e. epoch 999). The model finally prunes 60.4% of neurons at the end of stage 3 fine-tuning (f. epoch 1999). In this sparsification experiment, across different training stages, the shapes of decision boundaries are appropriately the same even though a large amount of neurons are pruned.

Interestingly, the final architectures achieved by network expansion and network sparsification are very similar (3300 vs. 3234), so are their accuracies (99.6% vs. 99.00%) even though the initial network capacities are quite different (15 vs. 8160). This experiment demonstrates that given an initial network architecture either large or small, NPNs can adapt their capacities to solve a learning task with high accuracies.

### 3.5.2 MNIST

In the second part of experiments, we run NPNs with LeNet5-Caffe on the MNIST dataset for network sparsification and expansion. LeNet5-Caffe consists of two convolutional layers of 20 and 50 neurons, respectively, interspersed with max pooling layers, followed by two fully-connected layers with 800 and 500 neurons. We start network sparsification from the full LeNet5 architecture (20-50-800-500, in short), while in the expansion experiment we start from a very small network with only 3 neurons in the first two convolutional layers, 48 and 3 neurons in the fully-connected layers (3-3-48-3, in short). We pre-train the NPNs for 100 epochs, followed by sparsification/expansion for 250 epochs and fine-tuning for 150 epochs. For both experiments, we use $\lambda = (10, 0.5, 0.1, 10)/N$ where $N$ is the number of training images.

The results are shown in Table 3.1 and Fig. 3.3. For the sparsification experiment, the

Table 3.1: The network sparsification and expansion with LeNet5 on MNIST.

| | Stage | Arch. (# of Parameters) | Accuracy (%) |
|---|---|---|---|
| Baseline | - | 20-50-800-500 (4.23e5) | **99.40** |
| Sparsification | stage 1 | 20-50-800-500 (4.23e5) | 99.36 |
| | stage 2 | 7-9-109-30 (5320) | 98.90 |
| | stage 3 | 7-9-109-30 (5320) | 98.91 |
| Expansion | stage 1 | 3-3-48-3 (474) | 93.85 |
| | stage 2 | 8-8-53-8 (2304) | 96.71 |
| | stage 3 | 8-8-53-8 (**2304**) | 98.31 |

NPN achieves 99.36% accuracy at the end of pre-training, and yields a sparse architecture with a minor accuracy drop at the end of sparsification. With the fine-tuning at stage 3, the accuracy reaches 98.91% in the end. For the expansion experiment, the NPN achieves a low accuracy of 93.85% after pre-training due to insufficient capacity of the initial network, then it expands to a larger network and improves the accuracy to 96.71%. Finally, the accuracy reaches 98.31% after fine-tuning. Fig. 3.3 demonstrate the learning processes of NPNs on MNIST for network sparsification and network expansion. It is interesting to note that both network sparsification and expansion reach the similar network capacity with similar classification accuracies at the end of the training even though they are started from two significantly different network architectures. It's worth emphasizing that both sparsification and expansion reach similar accuracies to the baseline model, while over 99% weights are pruned.

### *3.5.3 CIFAR-10/100*

In the final part of experiments, we evaluate NPNs with ResNet56 on CIFAR-10 and CIFAR-100 for network sparsification and network expansion. Due to the existence of skip connections, we do not sparsify the last convolutional layer of each residual block to keep a valid

addition operation. To train NPNs with this modern CNN architecture, two optimizers are used: (1) SGD with momentum for ResNet56 parameters with an initial learning rate of 0.1, and (2) Adam for the binary gate parameters $\boldsymbol{\phi}$ with an initial learning rate of 0.001. The batch size, weight decay and momentum of SGD are set to 128, 5e-4 and 0.9, respectively. The learning rate is multiplied by 0.1 every 60 epochs for SGD optimizer, while for Adam optimizer we multiplied learning rate by 0.1 at epoch 120 and 180.

As before, the networks are training in three stages by leveraging $k$. We pre-train the network for the first 20 epochs. In the expansion experiments, we pre-train a small network which only has 20% of neurons, while in the sparsification experiments, we pre-triain the full architecture. The network is then trained in stage 2 for 180 epochs, and finally is fine-tuned in stage 3 for 20 epochs. For the sparsification experiments, we use $\lambda = 1e-5$ for all layers, while for the expansion experiments we use $\lambda = 0$ to encourage more neurons to be activated.

Table 3.2: The network sparsification and expansion with ResNet56 on CIFAR10 and CIFAR100.

| Model | Method | Acc. (%) | $\Delta_{Acc}$ | FLOPs (P.R. %) | Params. (P.R. %) |
|---|---|---|---|---|---|
| CIFAR10 | SFP (He et al. 2018a) | 93.6→93.4 | -0.2 | 59.4M (53.1) | - |
| | AMC (He et al. 2018b) | 92.8→91.9 | -0.9 | 62.5M (50.0) | - |
| | FPGM (He et al. 2019) | 93.6→93.5 | -0.1 | 59.4M (52.6) | - |
| | TAS (Dong & Yang 2019) | 94.5→93.7 | -0.8 | 59.5M (52.7) | - |
| | HRank (Lin et al. 2020) | 93.3→93.5 | **+0.2** | 88.7M (29.3) | 0.71M (16.8) |
| | NPN Sparsification | 93.2→93.0 | -0.2 | 76.1M (40.1) | **0.33M (61.2)** |
| | NPN Expansion | 93.2→92.7 | -0.5 | 100.5M (20.9) | 0.55M (35.3) |
| CIFAR100 | SFP (He et al. 2018a) | 71.4→68.8 | -2.6 | 59.4M (52.6) | - |
| | FPGM (He et al. 2019) | 71.4→69.7 | -1.7 | 59.4M (52.6) | - |
| | TAS (Dong & Yang 2019) | 73.2→72.3 | -0.9 | 61.2M (51.3) | - |
| | NPN Sparsification | 71.1→70.9 | **-0.2** | 101.8M (19.8) | 0.61M (28.2) |
| | NPN Expansion | 71.1→69.9 | -0.5 | 102.8M (19.1) | 0.60M (29.4) |

"Δ": '+' denotes accuracy gain; '-' denotes accuracy loss. "Params. (P.R. %)": prune ratio in parameters.

We compare the performance of NPNs with the state-of-the-art pruning algorithms, in-

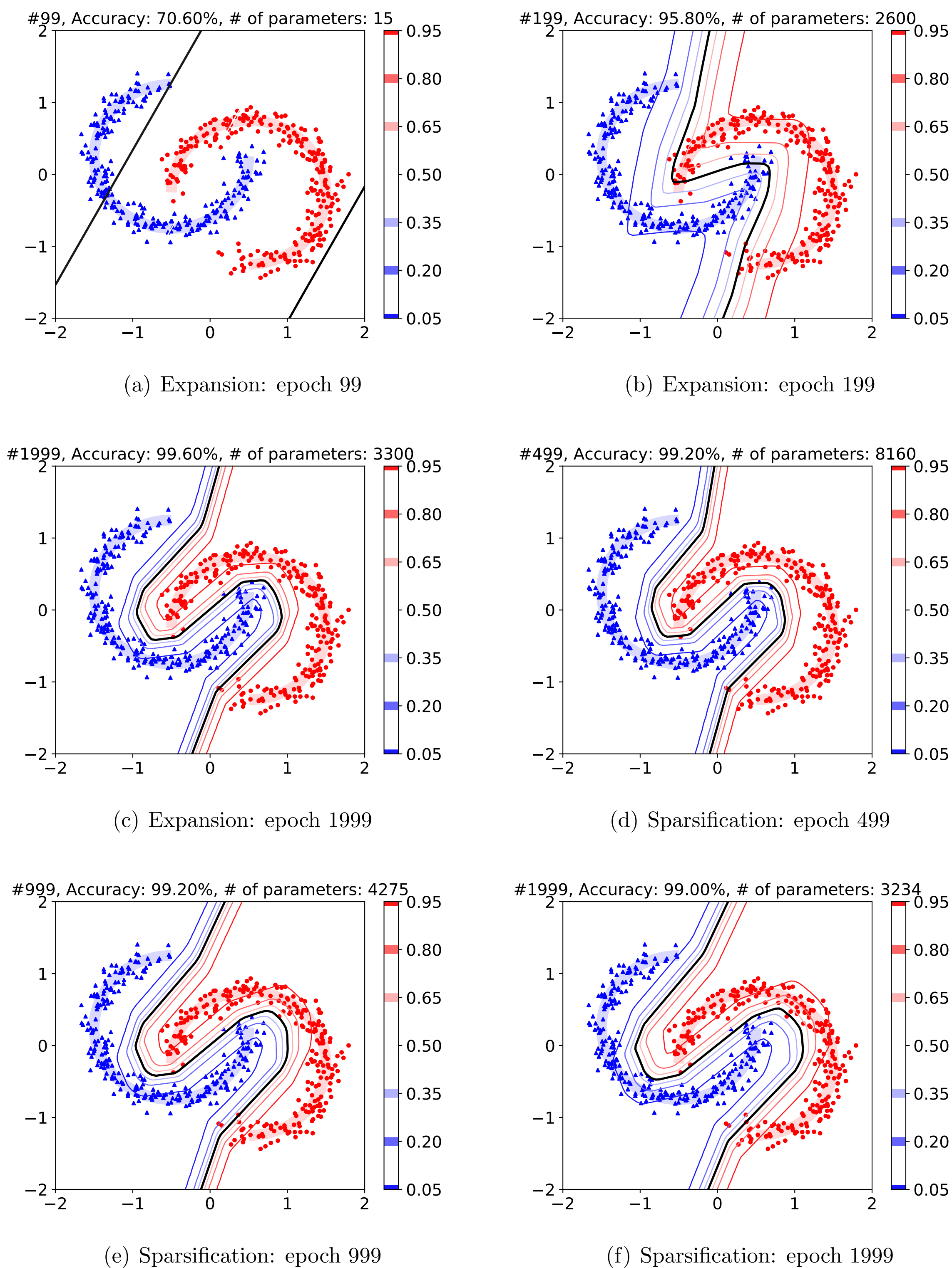
cluding SFP He et al. (2018a), AMC He et al. (2018b), FPGM He et al. (2019), TAS Dong & Yang (2019) and HRank Lin et al. (2020), with the results shown in Table 3.2. Since the baseline accuracies in all the reference papers are different, we follow the common practice and compare the performances of all competing methods by their accuracy gains $\Delta_{Acc}$ and their pruning rates in terms of FLOPs and network parameters. It can be observed that NPN sparsification and NPN expansion achieve very competitive performances to the state-of-the-arts in terms of classification accuracies and prune rates. More interestingly, similar to the results on MNIST, both network sparsification and expansion reach the similar network capacities with similar classification accuracies at the end of the training even though they are started from two significantly different network architectures, demonstrating the plasticity of NPN for network sparsification and expansion.

### 3.6 Conclusion

We propose neural plasticity networks (NPNs) for network sparsification and expansion by attaching each unit of a network with a stochastic binary gate, whose parameters are jointly optimized with original network parameters. The activation or deactivation of a unit is completely data-driven and determined by an $L_0$-regularized objective. Our NPN unifies dropout (when $k=0$), traditional training of DNNs (when $k=\infty$) and interpolate between these two. To the best of our knowledge, it is the first learning framework that unifies network sparsification and network expansion in an end-to-end training pipeline that supports pre-training, sparsification/expansion, and fine-tuning seamlessly. Along the way, we also give

a new interpretation of dropout. Extensive experiments on multiple public datasets and multiple network architectures validate the effectiveness of NPNs for network sparsification and expansion in terms of model compactness and predictive accuracies.

As for future extensions, we plan to design better (possibly non-antithetic) parametric function $g(\phi)$ to improve the compactness of learned networks. We also plan to extend the framework to prune or expand network layers to further improve model compactness and accuracy altogether.




(a) Expansion: epoch 99

(b) Expansion: epoch 199

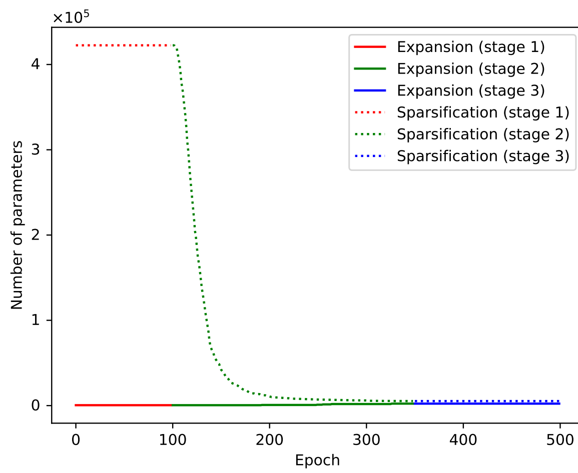
(c) Expansion: epoch 1999

(d) Sparsification: epoch 499

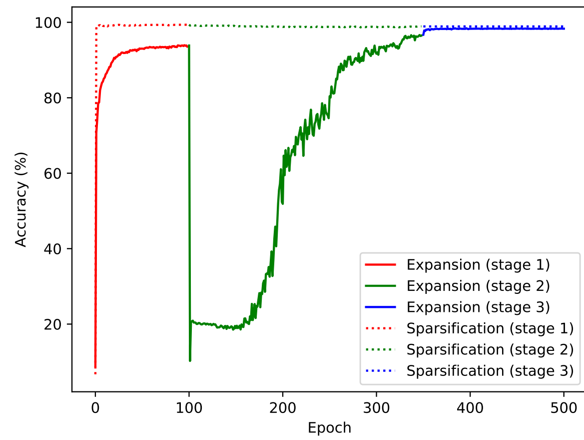
(e) Sparsification: epoch 999

(f) Sparsification: epoch 1999

Figure 3.2: The evolution of the decision boundaries of NPNs for network expansion (a,b,c) and network sparsification (d,e,f).




(a) LeNet5: # of parameters




(b) LetNet5: Test accuracy

Figure 3.3: The evolution of network capacity and test accuracy as a function of epoch for NPN network sparsification and expansion with LetNet5 on MNIST.

# CHAPTER 4

## Dep-$L_0$: Improving $L_0$-based Network Sparsification via Dependency Modeling

Training deep neural networks with an $L_0$ regularization is one of the prominent approaches for network pruning or sparsification. This method attaches a binary gate to each weight of a neural network, and penalizes the complexity of the network, measured by the $L_0$ norm of the weight matrix. However, recent work of Gale et al. Gale et al. (2019a) reveals that although $L_0$-HC works well on smaller datasets, it fails to prune very deep networks on large-scale datasets, such as ResNet50 on ImageNet. The original $L_0$-HC algorithm was proposed and evaluated on filter-level pruning, while Gale et al. Gale et al. (2019a) focus on the weight-level pruning. Therefore, it is unclear if the observation of Gale et al. (2019a) is due to pruning granularity or the deficiency of the $L_0$ regularization based method. To understand this, we evaluate the original $L_0$-HC to sparsify ResNet50 at filter level on ImageNet, and find that it indeed cannot prune ResNet50 without a significant damage of model quality, confirming the observation made by Gale et al. (2019a). This indicates that the failure of $L_0$-HC is likely due to the deficiency of the $L_0$-norm based approach. We further analyze $L_0$-HC in the lens of variational inference Blei et al. (2017), and find that **the failure is likely due to an over-simplified assumption that models the variational posterior of binary gates to be element-wise independent**. To verify this hypothesis, we propose to incorporate the dependency into the binary gates, and model the gate dependency across CNN layers with a multi-layer perceptron (MLP). Extensive experiments show that our dependency-enabled $L_0$ sparsification, termed Dep-$L_0$, once again is able to prune very deep networks

on large-scale datasets, while achieving competitive or sometimes even better performances than the state-of-the-art pruning methods.

Our main contributions can be summarized as follows:

- From a variational inference perspective, we show that the effectiveness of $L_0$-HC Louizos et al. (2018a) might be hindered by the implicit assumption that all binary gates attached to a neural network are independent to each other. To mitigate this issue, we propose Dep-$L_0$ that incorporates the dependency into the binary gates to improve the original $L_0$-based sparsification method.

- A series of experiments on multiple datasets and multiple modern CNN architectures demonstrate that Dep-$L_0$ improves $L_0$-HC consistently, and is very competitive or sometimes even outperforms the state-of-the-art pruning algorithms.

- Moreover, Dep-$L_0$ converges faster than $L_0$-HC in terms of network structure search, and reduces the time to solution by 20%-40% compared to $L_0$-HC in our experiments.

### 4.1 Method

Our algorithm is motivated by $L_0$-HC Louizos et al. (2018a), which prunes neural networks by optimizing an $L_0$ regularized loss function and relaxing the non-differentiable Bernoulli distribution with the Hard Concrete (HC) distribution. Since $L_0$-HC can be viewed as a special case of variational inference under the spike-and-slab prior Louizos et al. (2018a), in this section we first formulate the sparse structure learning from this perspective, then discuss the deficiency of $L_0$-HC and propose dependency modeling, and finally present Dep-$L_0$.

### *4.1.1 Sparse Structure Learning*

Consider a dataset $\mathcal{D} = \{\boldsymbol{x}_i, y_i\}_{i=1}^N$ that consists of $N$ pairs of instances, where $\boldsymbol{x}_i$ is the $i$th observed data and $y_i$ is the associated class label. We aim to learn a model $p(\mathcal{D}|\boldsymbol{\theta})$, parameterized by $\boldsymbol{\theta}$, which fits $\mathcal{D}$ well with the goal of achieving good generalization to unseen test data. In order to sparsify the model, we introduce a set of binary gates $\boldsymbol{z} = \{z_1, \cdots, z_{|\boldsymbol{\theta}|}\}$, one gate for each parameter, to indicate whether the corresponding parameter being kept ($z = 1$) or not ($z = 0$).

This formulation is closely related to the spike-and-slab distribution Mitchell & Beauchamp (1988), which is widely used as a prior in Bayesian inference to impose sparsity. Specifically, the spike-and-slab distribution defines a mixture of a delta spike at zero and a standard Gaussian distribution:

$$
\begin{aligned}
p(z) &= \mathrm{Bern}(z|\pi) \\
p(\theta|z=0) = \delta(\theta), &\quad p(\theta|z=1) = \mathcal{N}(\theta|0,1),
\end{aligned} \tag{4.1.1}
$$

where $\mathrm{Bern}(\cdot|\pi)$ is the Bernoulli distribution with parameter $\pi$, $\delta(\cdot)$ is the Dirac delta function, i.e., a point probability mass centered at the origin, and $\mathcal{N}(\theta|0,1)$ is the Gaussian distribution with zero mean and unit variance. Since both $\boldsymbol{\theta}$ and $\boldsymbol{z}$ are vectors, we assume the prior $p(\boldsymbol{\theta}, \boldsymbol{z})$ factorizes over the dimensionality of $\boldsymbol{z}$.

In Bayesian statistics, we would like to estimate the posterior of $(\boldsymbol{\theta}, \boldsymbol{z})$, which can be

calculated by Bayes' rule:

$$p(\boldsymbol{\theta}, \boldsymbol{z}|\mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{\theta}, \boldsymbol{z})p(\boldsymbol{\theta}, \boldsymbol{z})}{p(\mathcal{D})}. \tag{4.1.2}$$

Practically, the true posterior distribution $p(\boldsymbol{\theta}, \boldsymbol{z}|\mathcal{D})$ is intractable due to the non-conjugacy of the model likelihood and the prior. Therefore, here we approximate the posterior distribution via variational inference Blei et al. (2017). Specially, we can approximate the true posterior with a parametric variational posterior $q(\boldsymbol{\theta}, \boldsymbol{z})$, the quality of which can be measured by the Kullback-Leibler (KL) divergence:

$$KL[q(\boldsymbol{\theta}, \boldsymbol{z})||p(\boldsymbol{\theta}, \boldsymbol{z}|\mathcal{D})], \tag{4.1.3}$$

which is again intractable, but can be optimized by maximizing the variational lower bound of $\log p(\mathcal{D})$, defined as

$$L = \mathbb{E}_{q(\boldsymbol{\theta}, \boldsymbol{z})}[\log p(\mathcal{D}|\boldsymbol{\theta}, \boldsymbol{z})] - KL[q(\boldsymbol{\theta}, \boldsymbol{z})\|p(\boldsymbol{\theta}, \boldsymbol{z})], \tag{4.1.4}$$

where the second term can be further expanded as:

$$\begin{aligned} KL[q(\boldsymbol{\theta}, \boldsymbol{z})\|p(\boldsymbol{\theta}, \boldsymbol{z})] =& \mathbb{E}_{q(\boldsymbol{\theta}, \boldsymbol{z})}[\log q(\boldsymbol{\theta}, \boldsymbol{z}) - \log p(\boldsymbol{\theta}, \boldsymbol{z})] \\ =& \mathbb{E}_{q(\boldsymbol{\theta}, \boldsymbol{z})}[\log q(\boldsymbol{\theta}|\boldsymbol{z}) - \log p(\boldsymbol{\theta}|\boldsymbol{z}) + \log q(\boldsymbol{z}) - \log p(\boldsymbol{z})] \\ =& KL[q(\boldsymbol{\theta}|\boldsymbol{z})||p(\boldsymbol{\theta}|\boldsymbol{z})] + KL[q(\boldsymbol{z})||p(\boldsymbol{z})]. \end{aligned} \tag{4.1.5}$$

In $L_0$-HC Louizos et al. (2018a), the variational posterior $q(\boldsymbol{z})$ is factorized over the dimensionality of $\boldsymbol{z}$, i.e., $q(\boldsymbol{z}) = \prod_{j=1}^{|\boldsymbol{\theta}|} q(z_j) = \prod_{j=1}^{|\boldsymbol{\theta}|} \text{Bern}(z_j|\pi_j)$. By the law of total probability,

we can further expand Eq. 4.1.5 as

$$\begin{aligned}
&KL[q(\boldsymbol{\theta}, \boldsymbol{z})\|p(\boldsymbol{\theta}, \boldsymbol{z})] \\
&= \sum_{j=1}^{|\boldsymbol{\theta}|} \Big( q(z_j = 0) KL[q(\theta_j|z_j = 0)||p(\theta_j|z_j = 0)] + \\
&\quad + q(z_j = 1) KL[q(\theta_j|z_j = 1)||p(\theta_j|z_j = 1)] \Big) + \sum_{j=1}^{|\boldsymbol{\theta}|} KL[q(z_j)||p(z_j)] \\
&= \sum_{j=1}^{|\boldsymbol{\theta}|} q(z_j = 1) KL[q(\theta_j|z_j = 1)||p(\theta_j|z_j = 1)] + \sum_{j=1}^{|\boldsymbol{\theta}|} KL[(q(z_j)||p(z_j)]. \qquad (4.1.6)
\end{aligned}$$

The last step holds because $KL[q(\theta_j|z_j = 0)||p(\theta_j|z_j = 0)] = KL[q(\theta_j|z_j = 0)||\delta(\theta_j)] = 0$.

Furthermore, letting $\boldsymbol{\theta} = \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}$ and assuming $\lambda = KL[q(\theta_j|z_j = 1)||p(\theta_j|z_j = 1)]$, the lower bound $L$ (4.1.4) can be simplified as

$$\begin{aligned}
L &= \mathbb{E}_{q(\boldsymbol{z})}[\log p(\mathcal{D}|\widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z})] - \sum_{j=1}^{|\boldsymbol{\theta}|} KL\left(q\left(z_j\right)||p\left(z_j\right)\right) - \lambda \sum_{j=1}^{|\boldsymbol{\theta}|} q\left(z_j = 1\right) \\
&\leq \mathbb{E}_{q(\boldsymbol{z})}[\log p(\mathcal{D}|\widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z})] - \lambda \sum_{j=1}^{|\boldsymbol{\theta}|} \pi_j, \qquad (4.1.7)
\end{aligned}$$

where the inequality holds due to the non-negativity of KL-divergence.

Given that our model is a neural network $h(\boldsymbol{x}; \widetilde{\boldsymbol{\theta}}, \boldsymbol{z})$, parameterized by $\widetilde{\boldsymbol{\theta}}$ and $\boldsymbol{z}$, Eq. 4.1.7 turns out to be an $L_0$-regularized loss function Louizos et al. (2018a):

$$\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\pi}) = \mathbb{E}_{q(\boldsymbol{z})}\left[\frac{1}{N}\sum_{i=1}^{N} \mathcal{L}\left(h(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}), y_i\right)\right] + \lambda \sum_{j=1}^{|\boldsymbol{\theta}|} \pi_j, \qquad (4.1.8)$$

where $\mathcal{L}(\cdot)$ is the cross entropy loss for classification.

In the derivations above, the variational posterior $q(\boldsymbol{z})$ is assumed to factorize over the

dimensionality of $\boldsymbol{z}$, i.e., $q(\boldsymbol{z}) = \prod_{j=1}^{|\boldsymbol{\theta}|} q(z_j)$. This means all the binary gates $\boldsymbol{z}$ are assumed to be *independent* to each other – the mean-field approximation Blei et al. (2017). In variational inference, it is common to assume the prior $p(\boldsymbol{z})$ to be element-wise independent; the true posterior $p(\boldsymbol{z}|\mathcal{D})$, however, is unlikely to be element-wise independent. Therefore, approximating the true posterior by an element-wise independent $q(\boldsymbol{z})$ is a very restrict constraint that limits the search space of admissible $q(\boldsymbol{z})$ and is known in Bayesian statistics for its poor performance Bishop (2007); Blei et al. (2017). We thus hypothesize that this mean-field approximation may be the cause of the failure reported by Gale et al. Gale et al. (2019a), and the independent assumption hinders the effectiveness of the $L_0$-based pruning method. Therefore, we can potentially improve $L_0$-HC by relaxing this over-simplified assumption and modeling the dependency among binary gates $\boldsymbol{z}$ explicitly.

Specifically, instead of using a fully factorized variational posterior $q(\boldsymbol{z})$, we can model $q(\boldsymbol{z})$ as a conditional distribution by using the chain rule of probability

$$q(\boldsymbol{z}) = q(z_1)q(z_2|z_1)q(z_3|z_1, z_2) \cdots q(z_{|\theta|}|z_1, \cdots, z_{|\boldsymbol{\theta}|-1}),$$

where given an order of binary gates $\boldsymbol{z} = \{z_1, z_2, \cdots, z_{|\theta|}\}$, $z_i$ is dependent on all previous gates $z_{<i}$. With this, Eq. 4.1.8 can be rewritten as

$$\mathcal{R}(\widetilde{\boldsymbol{\theta}}, \boldsymbol{\pi}) = \lambda \sum_{j=1}^{|\boldsymbol{\theta}|} \pi_j + \mathbb{E}_{q(z_1)\cdots q(z_{|\boldsymbol{\theta}|}|z_1,\cdots,z_{|\boldsymbol{\theta}|-1})} \left[ \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}\left( h(\boldsymbol{x}_i; \widetilde{\boldsymbol{\theta}} \odot \boldsymbol{z}), y_i \right) \right], \tag{4.1.9}$$

which is a dependency-enabled $L_0$ regularized loss function for network pruning. Detailed design of the dependency modeling is to be discussed in later sections.

#### *4.1.2 Group Sparsity*

So far we have modeled a sparse network by attaching a set of binary gates $\boldsymbol{z}$ to the network at the weight level. As we discussed in the introduction, we prefer to prune the network at the filter level to fully utilize general purpose CPUs or GPUs. To this end, we consider group sparsity that shares a gate within a group of weights. Let $G = \{g_1, g_2, \cdots, g_{|G|}\}$ be a set of groups, where each element corresponds to a group of weights, and $|G|$ is the number of groups. With the group sparsity, the expected $L_0$-norm of model parameters (the first term of Eq. 4.1.9) can be calculated as

$$\mathbb{E}_{q(\boldsymbol{z})}\|\boldsymbol{\theta}\|_0 = \sum_{j=1}^{|\boldsymbol{\theta}|} q(z_j = 1|z_{<j}) = \sum_{k=1}^{|G|} |g_k|\pi_k, \tag{4.1.10}$$

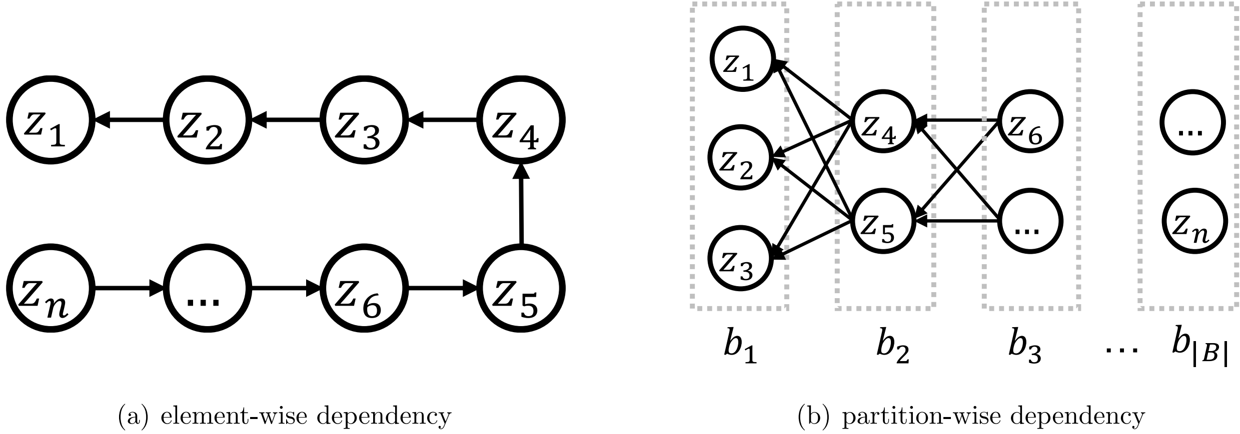
where $|g_k|$ denotes the number of weights in group $k$.

In all our experiments, we perform filter-level pruning by attaching a binary gate to all the weights of a filter (i.e., a group). Since modern CNN architectures often contain batch normalization layers Ioffe & Szegedy (2015), in our implementation we make a slight modification that instead of attaching the gates to filters directly, we attach the gates to the feature maps after batch normalization. This is because batch normalization accumulates a moving average of feature statistics for normalization during the training process. Simply attaching a binary gate to the weights of a filter cannot remove the impact of a filter completely when $z = 0$ due to the memorized statistics from batch normalization. By attaching the gates to the feature maps after batch normalization, the impact of the corresponding filter can be completely removed when $z = 0$.

### *4.1.3 Gate Partition*

Modern CNN architectures, such as VGGNet Simonyan & Zisserman (2014), ResNet He et al. (2016b) and WideResNet Zagoruyko & Komodakis (2016b), often come with a large number of weights and filters. For example, VGG16 Simonyan & Zisserman (2014) contains 138M parameters and 4,224 filters. Since we attach a binary gate to each filter, the number of gates would be large and modeling the dependencies among them would lead to a huge computational overhead and optimization issues. To make our dependency modeling more practical, we propose gate partition to simplify the dependency modeling among gates. Specifically, the gates are divided into blocks, and the gates within each block are considered independent to each other, whereas the gates cross blocks are considered dependent. Fig. 4.1 illustrates the difference between an element-wise sequential dependency modeling and a partition-wise dependency modeling. Let's consider $z_1$, $z_2$, $z_3$ and $z_4$ in these two cases. In the element-wise sequential dependency modeling, as shown in Fig. 4.1(a), $z_2$ is dependent on $z_1$, $z_3$ is dependent on $z_1$ and $z_2$, and so on. As number of gates could be very large, the element-wise sequential modeling would lead to a very long sequence, whose calculation would incur huge computational overhead. Instead, we can partition the gates, as in Fig. 4.1(b), where $z_1$, $z_2$ and $z_3$ are in block $\boldsymbol{b}_1$ so they are considered independent to each other, while $z_4$ in block $\boldsymbol{b}_2$ is dependent on all $z_1$, $z_2$ and $z_3$.

We formally describe the gate partition as following. Given a set of gates $G = \{g_1, g_2, \cdots, g_{|G|}\}$, let $B = \{b_1, b_2, \cdots, b_{|B|}\}$ be a partition of $G$, where $b_i$ denotes block $i$, and $|B|$ is the total number of blocks. Then we can approximate the variational posterior of $\boldsymbol{z}$ by modeling the




(a) element-wise dependency

(b) partition-wise dependency

Figure 4.1: Illustration of (a) element-wise sequential dependency modeling, and (b) partition-wise dependency modeling.

distribution over blocks as

$$q(\boldsymbol{z}) \approx q(b_1)q(b_2|b_1)q(b_3|b_1, b_2) \cdots q(b_{|B|}|b_1, \cdots, b_{|B|-1}).$$

To reduce the complexity, we can further simplify it as

$$q(\boldsymbol{z}) \approx q(b_1)q(b_2|b_1)q(b_3|b_2) \cdots q(b_{|B|}|b_{|B|-1}), \tag{4.1.11}$$

where block $i$ only depends on previous block $i-1$, ignoring all the other previous blocks, i.e., $q(b_i|b_{i-1})$, – the first-order Markov assumption.

In our experiments, we define a layer-wise partition, i.e., a block containing all the filters in one layer. For example, in VGG16 after performing a layer-wise gate partition, we only need to model the dependency within 16 blocks instead of 4,224 gates, and therefore the computational overhead can be reduced significantly.

### *4.1.4 Neural Dependency Modeling*

Until now we have discussed the dependency modeling in a mathematical form. To incorporate the dependency modeling into the original deep network, we adopt neural networks to model the dependencies among gates. Specifically, we choose to use an MLP network as the *gate generator*. With the proposed *layer-wise* gate partition (i.e., attaching an MLP layer to a convolutional layer and a gate to a filter; see Fig. 4.1(b)), the MLP architecture can model the dependency of gates, as expressed in Eq. 4.1.11, effectively.

Formally, we represent the gate generator as an MLP, with $gen_l$ denoting the operation of the $l$th layer. The binary gate $z_{lk}$ (i.e. the $k$th gate in block $l$) can be generated by

$$\begin{aligned} &\log \alpha_0 = \mathbf{1} \\ &\log \alpha_l = gen_l(\log \alpha_{l-1}) \quad \text{with } gen_l(\cdot) = c \cdot \tanh(W_l \cdot), \\ &z_{lk} \sim \mathrm{HC}(\log \alpha_{lk}, \beta), \end{aligned} \tag{4.1.12}$$

where $W_l$ is the weight matrix of MLP at the $l$th layer, $c$ is a hyperparameter that bounds the value of $\log \alpha$ in the range of $(-c, c)$, and $\mathrm{HC}(\log \alpha, \beta)$ is the Hard Concrete distribution with the location parameter $\log \alpha$ and the temperature parameter $\beta$ Louizos et al. (2018a)[1], which makes the sample $z_{lk}$ differentiable w.r.t. $\log \alpha_{lk}$. We set $c = 10$ as default, which works well in all our experiments.

Fig. 5.1 illustrates the overall architecture of Dep-$L_0$. The original network is shown on

[1]Following $L_0$-HC Louizos et al. (2018a), $\beta$ is fixed to 2/3 in our experiments.

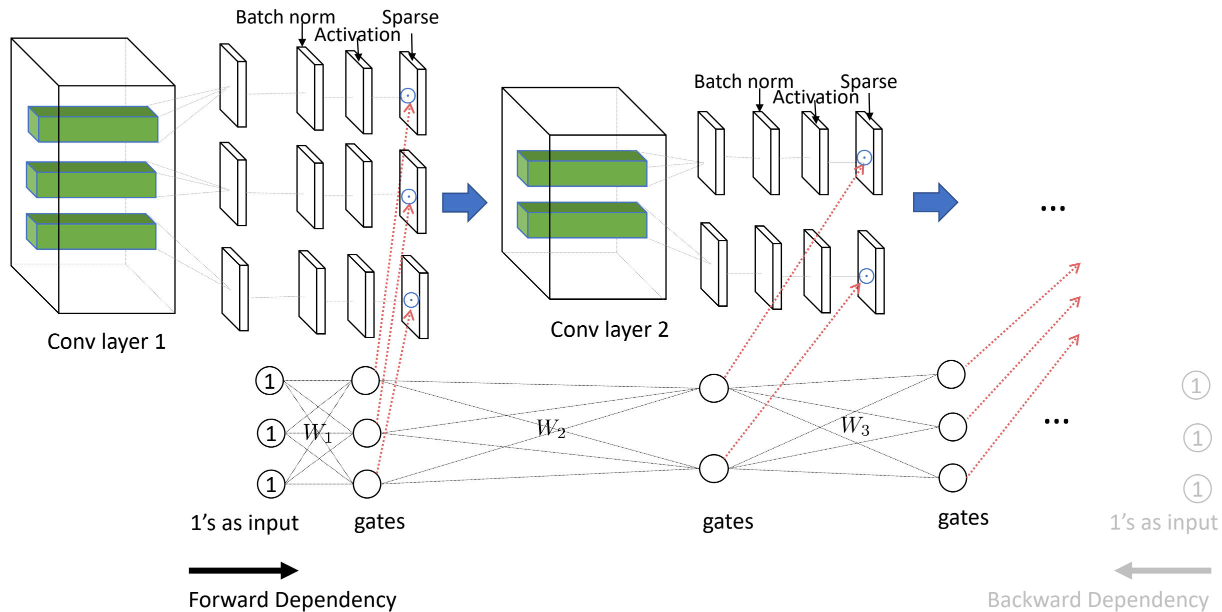



Figure 4.2: The computational graph of Dep-$L_0$.
The original CNN network is shown on the top, and the gate generator network (MLP) is shown at the bottom. Instead of attaching gates directly to filters, we attach gates to the feature maps after batch normalization (as shown by the red dotted lines). The gate can be generated by propagating the generator networks in either forward or backward direction. Both the original network and the gate generator are trained together as the whole pipeline is fully differentiable. $\{W_1, W_2, W_3, \cdots\}$ are the parameters of the MLP gate generator.

the top, and the gate generator (MLP) is shown at the bottom. Here we have a layer-wise partition of the gates, so the gate generator has the same number of layers as the original network. As we discussed in group sparsity, each gate is attached to a filter's output feature map after batch normalization. The input of the gate generator is initialized with a vector of 1's (i.e., $\log \alpha_0 = \mathbf{1}$, such that all the input neurons are activated at the beginning). The values of gates $\boldsymbol{z}$ are generated as we forward propagate the generator. The generated $\boldsymbol{z}$s are then attached to original networks, and the gate dependencies can be learned from the data directly. The whole pipeline (the original network and the gate generator) is fully differentiable as the Hard Concrete distribution (instead of Bernoulli) is used to sample $\boldsymbol{z}$,

so that we can use backpropagation to optimize the whole pipeline.

Furthermore, as shown in Fig. 5.1, the dependencies can be modeled in a backward direction as well, i.e., we generate the gates from the last layer $L$ of MLP first, and then generate the gates from layer $L-1$, and so on. In our experiments, we will evaluate the performance impacts of both forward and backward modeling.

In addition to MLPs, other network architectures such as LSTMs and CNNs can be used to model the gate generator as well. However, neither LSTMs nor CNNs achieves a competitive performance to MLPs in our experiments. Detailed ablation study is provided in the appendix.

## 4.2 Related Work

Model compression Deng et al. (2020) aims to reduce the size of a model and speed up its inference at the same time. Recently, there has been a flurry of interest in model compression, ranging from network pruning LeCun et al. (1990); Han et al. (2015); Louizos et al. (2018a); **?**); Lin et al. (2019a), quantization and binarization Courbariaux et al. (2016); Gupta et al. (2015), tensor decomposition Jaderberg et al. (2014); Denton et al. (2014), and knowledge distillation Hinton et al. (2015). Since our algorithm belongs to the category of network pruning, we mainly focus on reviewing related work in pruning.

### Dependency Modelling

Even though there are many network pruning algorithms today, most of them (if not all) *implicitly* assume all the neurons of a network are independent to each other when selecting

neurons for pruning. There are quite few works exploring the dependency inside neural networks for pruning. The closest one is LookAhead Park et al. (2020), which reinterprets the magnitude-based pruning as an optimization of the Frobenius distortion of a single layer, and improves the magnitude-based pruning by optimizing the Frobenius distortion of multiple layers, considering previous layer and next layer. Although the interaction of different layers is considered, the authors do not model the dependency of them explicitly. To the best of our knowledge, our Dep-$L_0$ is the first to model the dependency of neurons explicitly for network pruning.

### 4.3 Experiments

In this section we compare Dep-$L_0$ with the state-of-the-art pruning algorithms for CNN architecture pruning. In order to demonstrate the generality of Dep-$L_0$, we consider multiple image classification benchmarks (CIFAR10, CIFAR100 Krizhevsky & Hinton (2009) and ImageNet Deng et al. (2009b)) and multiple modern CNN architectures (VGG16 Simonyan & Zisserman (2014), ResNet50, and ResNet56 He et al. (2016b)). As majority of the computations of modern CNNs are in the convolutional layers, following the competing pruning methods, we only prune the convolutional filters and leave the fully connected layers intact (even though our method can be used to prune any layers of a network). For a fair comparison, our experiments closely follow the benchmark settings provided in the literature. All our experiments are performed with PyTorch on Nvidia V100 GPUs.

### *4.3.1 Experimental Details*

The experimental details are provided for the purpose of reproducibility. For a fair comparison, our experiments closely follow the benchmark settings provided in the literature.

#### *4.3.1.1 VGG16 on CIFAR10/100*

We adopt a tailored VGG16 network Liu & Deng (2015) to the CIFAR10/100 datasets. The network contains 13 convolutional layers and 1 fully connected layer. To sparsify the convolutional layers, the binary gates are attached to the output feature maps of all convolutional filters and the dependencies of the gates are modeled by an MLP in either *forward* or *backward* direction, as described in Sec. 3.4. The models are trained on CIFAR10/100 for 300 epochs with a batch size of 128. We apply two different optimizers for two different groups of parameters: SGD with momentum for VGG16 parameters, and Adam Kingma & Ba (2015) for the parameters of the gate generator. The initial learning rate is set to 0.05 for SGD and 0.001 for Adam. The momentum and weight decay is set to 0.9 and 5e-4, respectively. The learning rate is multiplied by 0.2 every 60 epochs. For the gate generator, we initialize the bias terms of each MLP layer with samples from a Gaussian distribution $\mathcal{N}(3, 0.01)$ to encourage gates being activated at the beginning of training.

#### *4.3.1.2 ResNet56 on CIFAR10/100*

To evaluate the effectiveness of our Dep-$L_0$ on a more compact and modern CNN architecture, we further conduct experiments on ResNet56 He et al. (2016b). Due to the existence of skip connections, we do not sparsify the last convolutional layer of each residual block to keep

a valid addition operation. The dependencies of gates across layers are then built as an MLP. The training has been done in 300 epochs. Two optimizers are used: SGD with momentum for ResNet56 parameters with an initial learning rate of 0.1 and Adam for the gate generator parameters with an initial learning rate of 0.001. The batch size, weight decay and momentum of SGD are set to 128, 5e-4 and 0.9, respectively. The learning rate is multiplied by 0.1 and 0.2 every 60 epochs for SGD and Adam optimizer, respectively.

#### *4.3.1.3 ResNet50 on ImageNet*

Following Gale et al. (2019a), we perform experiments on ImageNet Deng et al. (2009b) with ResNet50 to evaluate our algorithm on a large-scale benchmark. This is one of our main experiments since Gale et al. Gale et al. (2019a) claim that $L_0$-HC fails to sparsify ResNet50 on ImageNet. Similar to the settings of ResNet56, we do not prune the last layer of each residual block. We train the model for 90 epochs and fine-tune for another 10 epochs with the gate generator frozen. We again use two optimizers: SGD with momentum as the optimizer for the parameters of ResNet50 with initial learning rate of 0.1 and Adam for the gate generator with initial learning rate of 0.001. The batch size, weight decay and momentum are set to 256, 1e-4 and 0.9, respectively. The learning rate is multiplied by 0.1 and 0.2 every 30 epochs for SGD and Adam optimizer, respectively.

#### *4.3.1.4 $L_0$-HC implementations*

From our experiments, we found that the original $L_0$-HC implementation[2] has a couple issues. First, the binary gates are not properly attached after batch normalization, which results in pruned neurons still having impact after being removed. Second, it only uses one optimizer – Adam for the original network parameters and the hard concrete parameters. We noted that using two optimizers: SGD with momentum for the original network and Adam for the hard concrete parameters works better. Therefore, we fixed these issues of $L_0$-HC for all the experiments and observed improved performance. For a fair comparison, we follow the same experiment settings as in Dep-$L_0$, and tune $L_0$-HC for the best performance.

### *4.3.2 CIFAR10 Results*

We compare Dep-$L_0$ with ten state-of-the-art filter pruning algorithms, including our main baseline $L_0$-HC, in this experiment. Since the baseline accuracies in all the reference papers are different, we compare the performances of all competing methods by their accuracy gains $\Delta_{Acc}$ and their pruning rates in terms of FLOPs and network parameters. For Dep-$L_0$, we evaluate our algorithm with *forward* and *backward* dependency modeling. Table 4.1 provides the results on CIFAR10. As can be seen, for VGG16, our algorithm (with *backward* dependency modeling) achieves the highest FLOPs reduction of 65.9% on CIFAR10 with only 0.1% of accuracy loss. For ResNet56, our *forward* dependency modeling achieves the highest accuracy gain of 0.2% with a very competitive FLOPs reduction of 45.5%.

Since $L_0$-HC is our main baseline, we highlight the comparison between Dep-$L_0$ and

[2] `https://github.com/AMLab-Amsterdam/L0_regularization`

Table 4.1: Comparison of pruning methods on CIFAR10.

| Model | Method | Acc. (%) | $\Delta_{Acc}$ | FLOPs (P.R. %) | Params. (P.R. %) |
|---|---|---|---|---|---|
| VGG16 | Slimming (Liu et al. 2017) | 93.7→93.8 | +0.1 | 195M (51.0) | 2.30M (88.5) |
| | DCP (Zhuang et al. 2018) | 94.0→94.6 | **+0.6** | 109.8M (65.0) | **0.94M (93.6)** |
| | AOFP (Ding et al. 2019a) | 93.4→93.8 | +0.4 | 215M (31.3) | - |
| | HRank (Lin et al. 2020) | 94.0→93.4 | -0.6 | 145M (53.5) | 2.51M (82.9) |
| | $L_0$-HC (Our implementation) | 93.5→93.1 | -0.4 | 135.6M (39.8) | 2.8M (80.9) |
| | Dep-$L_0$ (forward) | 93.5→93.5 | 0 | 111.9M (64.4) | 2.1M (85.7) |
| | Dep-$L_0$ (backward) | 93.5→93.4 | -0.1 | **107.0M (65.9)** | 1.8M (87.8) |
| ResNet56 | SFP (He et al. 2018a) | 93.6→93.4 | -0.2 | 59.4M (53.1) | - |
| | AMC (He et al. 2018b) | 92.8→91.9 | -0.9 | 62.5M (50.0) | - |
| | DCP (Zhuang et al. 2018) | 93.8→93.8 | 0 | 67.1M (47.1) | **0.25M (70.3)** |
| | FPGM (He et al. 2019) | 93.6→93.5 | -0.1 | 59.4M (52.6) | - |
| | TAS (Dong & Yang 2019) | 94.5→93.7 | -0.8 | **59.5M (52.7)** | - |
| | HRank (Lin et al. 2020) | 93.3→93.5 | **+0.2** | 88.7M (29.3) | 0.71M (16.8) |
| | $L_0$-HC (Our implementation) | 93.3→92.8 | -0.5 | 71.0M (44.1) | 0.46M (45.9) |
| | Dep-$L_0$ (forward) | 93.3→93.5 | **+0.2** | 69.1M (45.5) | 0.48M (43.5) |
| | Dep-$L_0$ (backward) | 93.3→93.0 | -0.3 | 66.7M (47.4) | 0.49M (42.4) |

"$\Delta_{Acc}$": '+' denotes accuracy gain; '-' denotes accuracy loss; the worst result is in red. "FLOPs (P.R. %)": pruning ratio in FLOPs. "Params. (P.R. %)": prune ratio in parameters. "-": results not reported in original paper.

Table 4.2: Comparison of pruning methods on CIFAR100.

| Model | Method | Acc. (P.R. %) | $\Delta_{Acc}$ | FLOPs (P.R. %) | Params. (%) |
|---|---|---|---|---|---|
| VGG16 | Slimming (Liu et al. 2017) | 73.3→73.5 | +0.2 | 250M (37.1) | 5.0M (75.1) |
| | $L_0$-HC (Our implementation) | 72.2→70.0 | -1.2 | 138M (56.2) | 4.1M (72.5) |
| | Dep-$L_0$ (forward) | 72.2→71.6 | -0.6 | **98M (68.8)** | **2.1M (85.7)** |
| | Dep-$L_0$ (backward) | 72.2→72.5 | **+0.3** | 105M (66.6) | 2.2M (85.0) |
| ResNet56 | SFP (He et al. 2018a) | 71.4→68.8 | -2.6 | **59.4M (52.6)** | - |
| | FPGM (He et al. 2019) | 71.4→69.7 | -1.7 | **59.4M (52.6)** | - |
| | TAS (Dong & Yang 2019) | 73.2→72.3 | -0.9 | 61.2M (51.3) | - |
| | $L_0$-HC (Our implementation) | 71.8→70.4 | -1.4 | 82.2M (35.2) | 0.73M (15.2) |
| | Dep-$L_0$ (forward) | 71.8→71.7 | **-0.1** | 87.6M (30.9) | 0.56M (34.9) |
| | Dep-$L_0$ (backward) | 71.8→71.2 | -0.6 | 93.4M (26.3) | **0.52M (39.5)** |

"$\Delta_{Acc}$": '+' denotes accuracy gain; '-' denotes accuracy loss; the worst result is in red. "FLOPs (P.R. %)": pruning ratio in FLOPs. "Params. (P.R. %)": prune ratio in parameters. "-": results not reported in original paper.

$L_0$-HC in the table. As we can see, Dep-$L_0$ outperforms $L_0$-HC consistently in all the experiments. For VGG16, $L_0$-HC prunes only 39.8% of FLOPs but suffers from a 0.4% of accuracy drop, while our algorithm prunes more (65.9%) and almost keeps the same accuracy (-0.1%). For ResNet56, our algorithm prunes more (45.5% v.s. 44.1%) while achieves a higher accuracy (0.2% vs. -0.5%) than that of $L_0$-HC.

### *4.3.3 CIFAR100 Results*

Experimental results on CIFAR100 are reported in Table 4.2, where Dep-$L_0$ is compared with four state-of-the-arts pruning algorithms: Slimming Liu et al. (2017), SFP He et al. (2018a), FPGM He et al. (2019) and TAS Dong & Yang (2019). Similar to the results on CIFAR10, on this benchmark Dep-$L_0$ achieves the best accuracy gains and very competitive or sometimes even higher prune rates compared to the state-of-the-arts. More importantly, Dep-$L_0$ outperforms $L_0$-HC in terms of classification accuracies and pruning rates consistently, demonstrating the effectiveness of dependency modeling.

### *4.3.4 ImageNet Results*

The main goal of the paper is to make $L_0$-HC once again competitive on the large-scale benchmark of ImageNet. In this section, we conduct a comprehensive experiment on ImageNet, where the original $L_0$-HC fails to prune without a significant damage of model quality Gale et al. (2019a). Table 4.3 reports the results on ImageNet, where eight state-of-the-art filter pruning methods are included, such as SSS-32 Huang & Wang (2018), DCP Zhuang et al. (2018), Taylor Molchanov et al. (2019), FPGM He et al. (2019), HRank Lin et al. (2020) and others. As can be seen, Dep-$L_0$ (forward) prunes 36.9% of FLOPs and 37.2% of parameters with a 1.38% of accuracy loss, which is comparable with other state-of-the-art algorithms as shown in the table.

Again, since $L_0$-HC is our main baseline, we highlight the comparison between Dep-$L_0$ and $L_0$-HC in the table. We tune the performance of $L_0$-HC extensively by searching for

Table 4.3: Comparison of pruning methods on ImageNet.

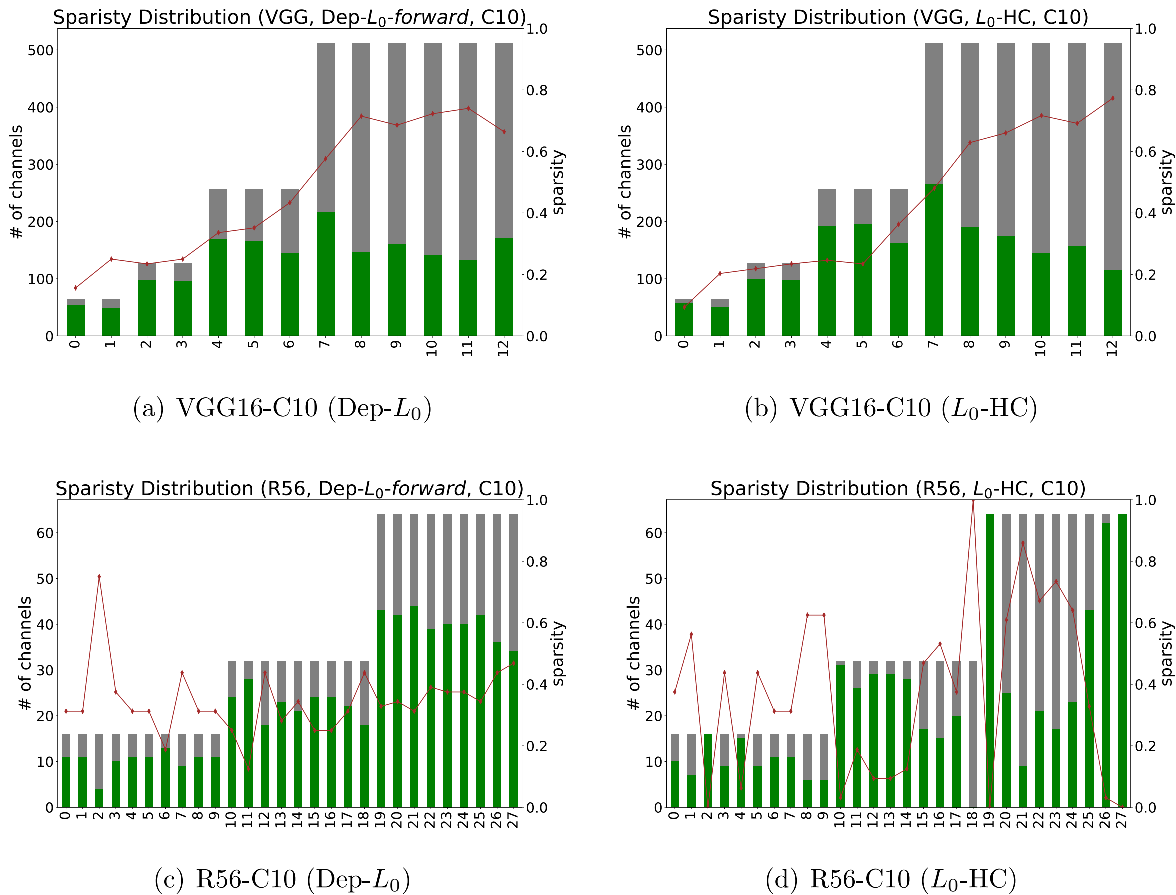
| Model | Method | Acc. (%) | $\Delta_{Acc}$ | FLOPs (P.R.%) | Params. (P.R.%) |
|---|---|---|---|---|---|
| ResNet50 | SSS-32 (Huang & Wang 2018) | 76.12 → 74.18 | -1.94 | 2.82B (31.1) | 18.60M (27.3) |
| | DCP (Zhuang et al. 2018) | 76.01 → 74.95 | -1.06 | **1.82B (55.6)** | **12.40M (51.5)** |
| | GAL-0.5 (Lin et al. 2019b) | 76.15 → 71.95 | -4.2 | 2.33B (43.1) | 21.20M (17.2) |
| | Taylor-72 (Molchanov et al. 2019) | 76.18 → 74.50 | -1.68 | 2.25B (45.0) | 14.20M (44.5) |
| | Taylor-81 (Molchanov et al. 2019) | 76.18 → 75.48 | -0.70 | 2.66B (34.9) | 17.90M (30.1) |
| | FPGM-30 (He et al. 2019) | 76.15 → 75.59 | -0.56 | 2.36B (42.2) | - |
| | FPGM-40 (He et al. 2019) | 76.15 → 74.83 | -1.32 | 1.90B (53.5) | - |
| | LeGR (Chin et al. 2019) | 76.10 → 75.70 | -0.40 | 2.37B (42.0) | - |
| | TAS (Dong & Yang 2019) | 77.46 → 76.20 | -1.26 | 2.31B (43.5) | - |
| | HRank (Lin et al. 2020) | 76.15 → 74.98 | -1.17 | 2.30B (43.8) | 16.15M (36.9) |
| | $L_0$-HC (Our implementation) | 76.15 → 76.15 | 0 | 4.09B (0.00) | 25.58M (0.00) |
| | Dep-$L_0$ (forward) | 76.15 → 74.77 | -1.38 | 2.58B (36.9) | 16.04M (37.2) |
| | Dep-$L_0$ (backward) | 76.15 → 74.70 | -1.45 | 2.53B (38.1) | 14.34M (43.9) |

"$\Delta_{Acc}$": '+' denotes accuracy gain; '-' denotes accuracy loss; the worst result is in red. "FLOPs (P.R. %)": pruning ratio in FLOPs. "Params. (P.R. %)": prune ratio in parameters. "-": results not reported in original paper.

the best hyperparameters in a large space. However, even with extensive efforts, $L_0$-HC still fails to prune the network without a significant damage of model quality, confirming the observation made by Gale et al. (2019a). On the other hand, our Dep-$L_0$ successfully prunes ResNet50 with a very competitive pruning rate and high accuracy compared to the state-of-the-arts, indicating that our dependency modeling indeed makes the original $L_0$-HC very competitive on the large-scale benchmark of ImageNet – the main goal of the paper.

### *4.3.5 Study of Learned Sparse Structures*

To understand of the behavior of Dep-$L_0$, we further investigate the sparse structures learned by Dep-$L_0$ and $L_0$-HC, with the results reported in Fig. 4.4. For VGG16 on CIFAR10, Figs. 4.4(a-b) demonstrate that both algorithms learn a similar sparsity pattern: the deeper a layer is, the higher prune ratio is, indicating that the shallow layers of VGG16 are more important for its predictive performance. However, for deeper networks such as ResNet56




(a) VGG16-C10 (Dep-$L_0$)

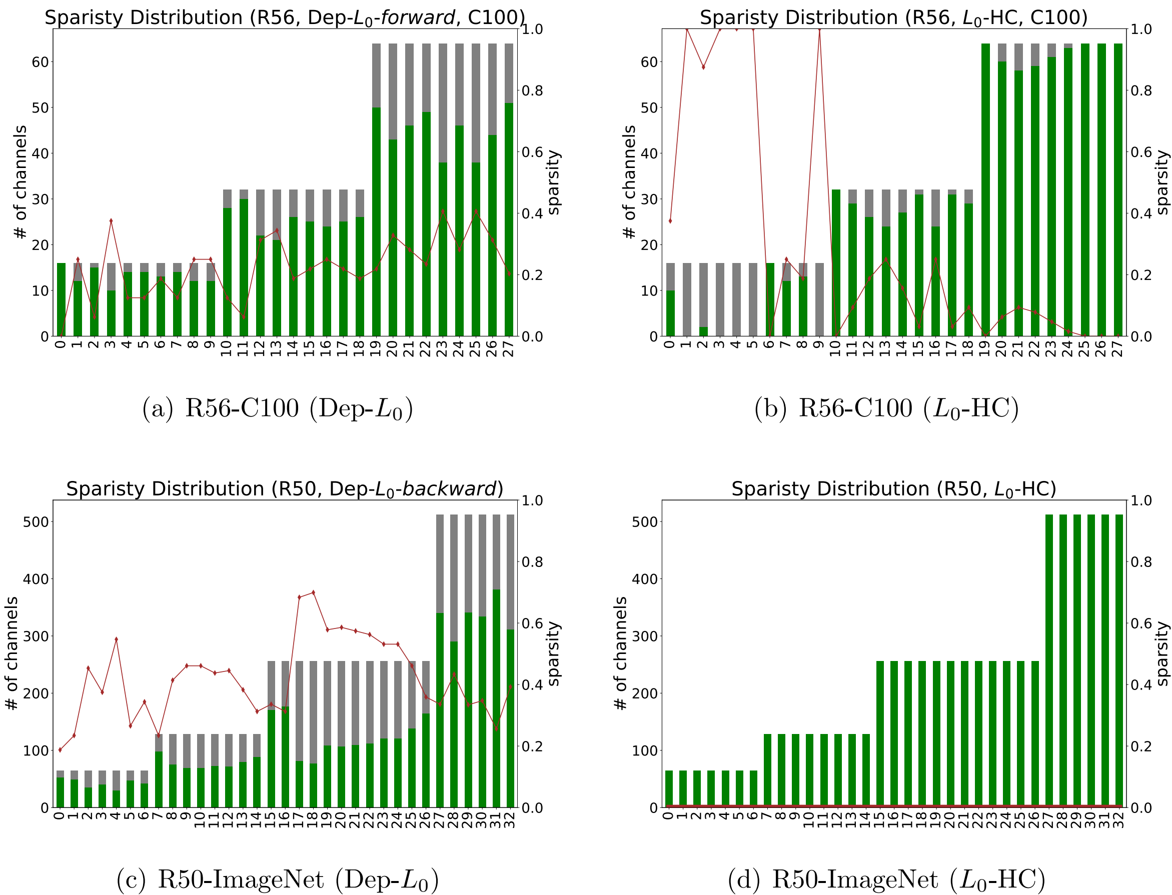
(b) VGG16-C10 ($L_0$-HC)

(c) R56-C10 (Dep-$L_0$)

(d) R56-C10 ($L_0$-HC)

Figure 4.3: The layer-wise prune ratios (red curves) of learned sparse structures. The height of a bar denotes the number of filters of a convolutional layer and gray (green) bars correspond to the original (pruned) architecture, respectively. "R50/R56": ResNet 50/56; "C10/C100": CIFAR10/100.

and ResNet50, the two algorithms perform very differently. For ResNet56 on CIFAR100, Figs. 4.4(c-d) show that Dep-$L_0$ sparsifies each layer by a roughly similar prune rate: it prunes around 20% of the filters in first 12 layers, and around 30% of the filters in the rest of layers. However, on the same benchmark $L_0$-HC tends to prune *all or nothing*: it completely prunes 5 out of 28 layers, but does not prune any filters in other six layers; for the rest of layers, the sparsity produced by $L_0$-HC is either extremely high or low. As of ResNet50




(a) R56-C100 (Dep-$L_0$)

(b) R56-C100 ($L_0$-HC)

(c) R50-ImageNet (Dep-$L_0$)

(d) R50-ImageNet ($L_0$-HC)

Figure 4.4: The layer-wise prune ratios (red curves) of learned sparse structures. The height of a bar denotes the number of filters of a convolutional layer and gray (green) bars correspond to the original (pruned) architecture, respectively. "R50/R56": ResNet 50/56; "C10/C100": CIFAR10/100.

on ImageNet, Figs. 4.4(e-f) show that the difference between Dep-$L_0$ and $L_0$-HC is more significant: Dep-$L_0$ successfully prunes the model with a roughly similar prune rate across all convolutional layers, while $L_0$-HC fails to prune any filters.

Table 4.4: Run-time comparison between Dep-$L_0$ and $L_0$-HC.

| Benchmark | Method | # Epochs | # Epochs BC | Per-epoch Time BC | TTS |
|---|---|---|---|---|---|
| R56-C10 | $L_0$-HC | 300 | 218 | 37.9 sec | 160 min |
| | Dep-$L_0$ (forward) | 300 | **106** | 42.2 sec | **124 min** |
| | Dep-$L_0$ (backward) | 300 | 140 | 42.6 sec | 141 min |
| R56-C100 | $L_0$-HC | 300 | 117 | 38.7 sec | 167 min |
| | Dep-$L_0$ (forward) | 300 | **58** | 43.9 sec | **127 min** |
| | Dep-$L_0$ (backward) | 300 | 61 | 43.6 sec | 133 min |
| R50-ImageNet | $L_0$-HC | 90 | fail to prune | 4185 sec | 104.6 h |
| | Dep-$L_0$ (forward) | 90 | **30** | 4342 sec | **59.5 h** |
| | Dep-$L_0$ (backward) | 90 | 32 | 4350 sec | 60.1 h |

"R50/R56": ResNet50/56; "C10/C100": CIFAR10/100. "BC": Before Convergence; "TTS": Time to Solution.

### *4.3.6 Run-time Comparison*

The main architectural difference between Dep-$L_0$ and $L_0$-HC is the gate generator. Even though the gate generator (MLP) is relatively small compared to the original deep network to be pruned, its existence increases the computational complexity of Dep-$L_0$. Thus, it is worth comparing the run-times of Dep-$L_0$ and $L_0$-HC as well as their convergence rates in terms of sparse structure search. Once a sparse structure is learned by a pruning algorithm, we can extract the sparse network from the original network and continue the training on the smaller structure such that we can reduce the total time to solution (TTS). To this end, we compare Dep-$L_0$ with $L_0$-HC in terms of (1) structure search convergence rate, i.e., how many training epochs are needed for a pruning algorithm to converge to a sparse structure? (2) Per-epoch training time before convergence, and (3) the total time to solution (TTS). The results are reported in Table 4.4. As can be seen, Dep-$L_0$ (both *forward* and *backward*) converges to a sparse structure in roughly half of the epochs that $L_0$-HC needs (column 4). Even though the per-epoch training time of Dep-$L_0$ is 12% (4%) larger than that of $L_0$-HC on CIFAR10/100 (ImageNet) due to the extra computation of the gate generator (column

5), the total time to solution reduces by 22.5% (43.1%) on the CIFAR10/100 (ImageNet) benchmarks thanks to the faster convergence rates and sparser models induced by Dep-$L_0$ as compared to $L_0$-HC (column 6).

## 4.4 Ablation Study of the Gate Generator

So far we have modelled the dependency among the binary gates with an MLP network. We note that other network architectures, such as MLP variants, CNN and LSTM Gers et al. (1999), can also be used to model the dependency. We therefore provide an ablation study to analyze their performances on VGG16-CIFAR10, with the results reported in Table 4.5.

Table 4.5: Ablation study of the gate generator architecture with VGG16 on CIFAR10.

| Dependency Modeling | Acc. (%) | FLOPs (P.R. %) | Params. (P.R. %) |
|---|---|---|---|
| FC($c_1$, $c_2$) $\rightarrow$ selected model | 93.5 | 111.9M (64.4) | 2.1M (85.7) |
| FC($c_1$, $c_2 * 2$) ReLU FC($c_2 * 2$, $c_2$) | 92.8 | 97.1M (69.1) | 2.4M (83.7) |
| FC($c_1$, $c_2 * 2$) Tanh FC($c_2 * 2$, $c_2$) | 92.9 | 85.3M (72.8) | 1.3M (91.1) |
| Conv1d ReLU FC | 93.4 | 162.3M (48.3) | 5.6M (61.9) |
| Conv1d ReLU Conv1d ReLU Conv1d ReLU FC | 93.4 | 155.8M (50.4) | 5.4M (63.3) |
| LSTM | not converged | | |

"FLOPs": pruning ratio in FLOPs. "Params.": prune ratio in parameters.

### *4.4.1 MLP variants*

In Sec. 3.4, we propose to use a fully connected (FC) layer, parameterized by $W_l$, to connect the gates between two consecutive layers. Other variants of MLP can also be used to model the dependency. Suppose the two consecutive layers have $c_1$ and $c_2$ gates, respectively. The single-FC-layer model used in the main text can be denoted as FC($c_1, c_2$) (the 1st row of Table 4.5). Alternatively, we can also use two FC layers interspersed by ReLU or Tanh to represent the dependency among gates (the 2nd and 3rd rows of Table 4.5). As can be seen,

these two MLP variants suffer from non-trivial accuracy drops, even though they achieve higher prune rates.

### *4.4.2 CNN*

We can also use a 1-D ConvNet as the gate generator. Specifically, we can model the dependency among gates in two consecutive layers by a 1-D convolution with the kernel size of $3 \times 1$, followed by ReLU and a FC layer (the 4th row in Table 4.5). Moreover, deep 1-D ConvNet, such as the one listed in the 5th row of Table 4.5 can be used. Even though they achieve similar accuracies as that of FC($c_1, c_2$), their prune rates are not competitive.

### *4.4.3 LSTM*

Finally, we exploit an LSTM as the gate generator to model the element-wise dependency autoregressively. We first experiment this LSTM gate generator with a small-scale CNN architecture – LeNet5 (LeCun et al. 1998), which only contains 70 filters. We achieve a competitive pruning rate with almost no accuracy loss. However, when it comes to VGG16, which contains 4,224 filters, the LSTM gate generator has to predict 4,224 binary gates autoregressively. After exhaustive hyperparameter tuning, we still cannot get the LSTM gate generator converged and thus fail to sparsify VGG16. In addition, training of the LSTM is very time consuming due to the autoregressive modeling of 4,224 binary gates.

### *4.4.4 Summary*

The ablation study presented in Table 4.5 show that FC($c_1, c_2$), the MLP network used in the main text, achieves the best balance between classification accuracy and the prune rate. Moreover, this MLP network is also computational efficient as it is relatively small compared to the original networks to be pruned. Therefore, we select FC($c_1, c_2$) as our MLP network in all the experiments.

## 4.5 Conclusion and Future Work

We propose Dep-$L_0$, an improved $L_0$ regularized network sparsification algorithm via dependency modeling. The algorithm is inspired by a recent observation of Gale et al. Gale et al. (2019a) that $L_0$-HC performs inconsistently in large-scale learning tasks. Through the lens of variational inference, we found that this is likely due to the mean-field assumption in variational inference that ignores the dependency among all the neurons for network pruning. We further propose a dependency modeling of binary gates to alleviate the deficiency of the original $L_0$-HC. A series of experiments are performed to evaluate the generality of our Dep-$L_0$. The results show that our Dep-$L_0$ outperforms the original $L_0$-HC in all the experiments consistently, and the dependency modeling makes the $L_0$-based sparsification once again very competitive and sometimes even outperforms the state-of-the-art pruning algorithms. Further analysis shows that Dep-$L_0$ also learns a better structure in fewer epochs, and reduces the total time to solution by 20%-40%.

As for future work, we plan to explore whether dependency modeling can be used to

improve other pruning methods. To the best of our knowledge, there are very few prior works considering dependency for network pruning (e.g., Park et al. (2020)). Our results show that this may be a promising direction to further improve many existing pruning algorithms. Moreover, the way we implement dependency modeling is still very preliminary, which can be improved further in the future.

# CHAPTER 5

# VB-LoRA: Extreme Parameter Efficient Fine-Tuning with Vector Banks

## 5.1 Introduction

In this chapter, we will introduce VB-LoRA, extreme parameter-efficient fine-tuning with *vector banks* based on a simple yet effective "divide-and-share" paradigm. We push the limits of LoRA (Hu et al. 2021) parameter efficiency by breaking the two barriers of low-rank decomposition: (1) locally within each module and each layer, and (2) only across the two original matrix dimensions (without division; see Sec. 5.3.2 for details). We argue that the parameters across different modules and layers can be shared, and thus the redundancy in parameters can be further reduced. In addition, by partitioning rank-one component vectors into sub-vectors, we introduce "virtrual" dimensions such that deep structure in the parameter space can be represented by a highly compressed matrix factorization.

VB-LoRA draws inspirations from previous line of work on quantized tensor networks (Oseledets 2010; Cichocki 2014) in breaking the constraint of physical dimension for extreme parameter compression. Specifically, VB-LoRA reparameterizes LoRA's low-rank adaptation by a rank-one decomposition and then divides the resulting vectors into sub-vectors of the same size. A *global* sharing mechanism is then learnt based on a sparse top-$k$ admixture module. The same sized sub-vectors allows parameters to be shared across modules and layers at the sub-vector level. Moreover, compared to the post-hoc matrix compression methods (Oseledets 2010; Khoromskij 2011), VB-LoRA is end-to-end differentiable, and therefore the fine-tuning process is aware of the compressed form, enabling task-oriented compression.

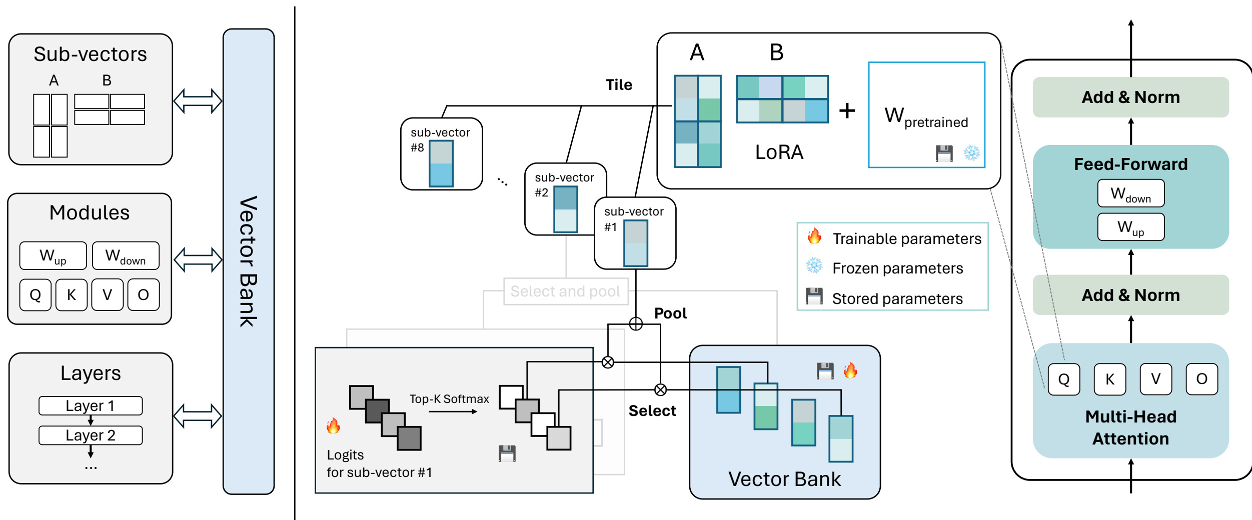



Figure 5.1: Overview of VB-LoRA.
**Left**: The model parameters can be represented as a composition of vectors from a *vector bank*, which is shared across sub-vectors, modules and layers. **Right**: Architecture of VB-LoRA. We use a top-$k$ softmax function to select $k$ vectors from the vector bank. The selected vectors are then pooled into a sub-vector, which is arranged at a desired position, forming the parameters of LoRA.

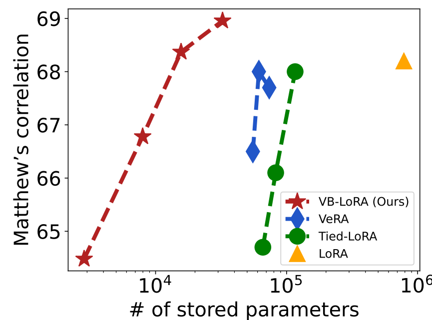



Figure 5.2: Comparison of the PEFT methods on RoBERTa-Large.
Our VB-LoRA achieves higher scores with significantly smaller number of stored parameters.

Figure 5.2 illustrates the parameter efficiency of VB-LoRA as compared with state-of-the-art PEFT methods. Our contributions are summarized as follows:

1. We introduce a "divide-and-share" paradigm that breaks the barriers of low-rank decomposition across matrix dimensions, modules, and layers by sharing parameters *globally* via a vector bank.

2. We reparameterize LoRA's low-rank decomposition by a rank-one decomposition, and divide the resulting vectors further into sub-vectors of the same size, enabling extreme parameter efficiency at the sub-vector level.

3. We propose a sparse top-$k$ module based on the admixture model to learn a global sharing mechanism, making our framework end-to-end differentiable and compression-aware.

4. Our method achieves extreme parameter efficiency while maintaining comparable or better empirical performance compared to the state-of-the-art PEFT methods on natural language understanding, natural language generation, and instruction tuning tasks.

## 5.2 Related Work

### *5.2.1 Exploit Global Redundancy for Enhanced Parameter Efficiency*

The parameters of deep neural networks (DNNs) can be naturally divided by layers, heads, or types (MHA or FFN). While LoRA (Hu et al. 2021) only exploits the *intra-matrix* dependency, Tied-LoRA (Renduchintala et al. 2023) employs a simple weight tying scheme on the low-rank matrices $A$ and $B$ across layers to reduce the *inter-matrix* redundancy. When $A$ and $B$ are randomly initialized, frozen, and shared across all layers, Tied-LoRA degenerates to VeRA (Kopiczko et al. 2024), which only requires two scaling vectors to be updated, leading to impressive parameter efficiency. Our VB-LoRA pushes the limits of LoRA parameter efficiency by sharing parameters globally across modules and layers at the sub-vector level.

On the low-dimensional reparameterization, Aghajanyan et al. (2020) empirically show that there exists a low-dimensional reparameterization that is as effective for fine-tuning as the full parameter space. The actualization of the random projection is achieved through the Fastfood transform (Le et al. 2013) for large-scale pre-trained language models. To make it structure-aware, a set of layer-wise scaling parameters are included as part of the training parameters. Following this intuition, we study the lightweight fine-tuning performance within LoRA based on the customized reparameterization that arises from the rank-one matrix decomposition.

Moreover, tensor decomposition has been leveraged for PEFT in ViT models (Jie & Deng 2023) based on classical formats, such as tensor-train or Tucker (Kolda & Bader 2009). We find that forcing multilinear decomposition across multiple modes results in a higher rank number, which is detrimental to the objective of parameter compression. An indirect comparison of VB-LoRA to Jie & Deng (2023) can be conducted by referring the compression rate to LoRA. From this perspective, our VB-LoRA can be viewed as a customized tensor format endowed by a convex geometry structure, which is enabled by a sparse top-$k$ admixture model we proposed.

### *5.2.2 Parameter Modeling based on Sparse Admixture Models*

Admixture models have been widely used in population genetics (Pritchard et al. 2000), topic modeling (Reisinger et al. 2010; Inouye et al. 2014), and hyperspectral unmixing (Li & Bioucas-Dias 2008; Fu et al. 2015) to extract archetypal (or endmember) components from observed data. The archetypal components can be relaxed to have mixed sign (Ding

et al. 2008) with identifiability guarantees (Lin et al. 2015). Conventionally, parameters estimation are conducted based on linear programming (Chan et al. 2009) or combinatorial algorithms (Arora et al. 2013). However, an involved integer programming problem arises when incorporating an extra top-$k$ constraint into the mixing weights that is especially challenging for the large-scale language models. In this work, we propose learning archetypal vector banks not from observed data but from model parameters of LLMs. By modifying the sparse top-$k$ module (Shazeer et al. 2016) commonly used in Mixture-of-Expert models (Jiang et al. 2024), the mixing weights and vector banks are optimized by backpropgatation under the objective of downstream fine-tuning tasks. The proposed top-$k$ admixture model is model-agnostic in the sense that it can be readily integrated into any neural network parameters or accumulated gradient updates.

## 5.3 Proposed Method

### *5.3.1 Preliminaries: Transformer Architecture and LoRA Adapters*

The transformer architecture (Vaswani et al. 2017) consists of $L$ layers, each containing two types of blocks: Multi-Head Attention (MHA) and Feed-Forward Network (FFN). We denote the query, key, value, and output matrices of MHA at layer $\ell$ as $\boldsymbol{\mathcal{W}}_t^\ell = \{\boldsymbol{W}_t^i\}_{i=1}^{N_h}$, $t \in \{q, k, v, o\}$, where $\boldsymbol{W}_t^i \in \mathbb{R}^{d \times d}$, and $N_h$ is the number of heads. Given $\text{FFN}(\boldsymbol{x}) = \boldsymbol{W}_{\text{down}}\text{ReLU}(\boldsymbol{W}_{\text{up}}\boldsymbol{x})$ with $\boldsymbol{x} \in \mathbb{R}^d$, viewing FFN as a multi-head operation, we further divide $\boldsymbol{W}_{\text{up}} \in \mathbb{R}^{cd \times d}$ and $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d \times cd}$ into $c$ matrices of size $d \times d$, denoted by $\boldsymbol{\mathcal{W}}_{\text{up}}^\ell = \{\boldsymbol{W}_{\text{up}}^{\ell,i}\}_{i=1}^c$ and $\boldsymbol{\mathcal{W}}_{\text{down}}^\ell = \{\boldsymbol{W}_{\text{down}}^{\ell,i}\}_{i=1}^c$, $c = 4$.

Given a pre-trained matrix $\boldsymbol{W}_0 \in \mathbb{R}^{m\times n}$, LoRA (Hu et al. 2021) constrains the weight increments $\Delta\boldsymbol{W}$ as a low-rank decomposition $\Delta\boldsymbol{W} = \boldsymbol{B}\boldsymbol{A}$, where $\boldsymbol{B} \in \mathbb{R}^{m\times r}$, $\boldsymbol{A} \in \mathbb{R}^{r\times n}$ are trainable parameters, with $r \ll \min(m, n)$. VeRA (Kopiczko et al. 2024) further limits the trainable parameters to two scaling vectors $b$ and $d$, which form the diagonal elements of two diagonal matrices $\Lambda_b$ and $\Lambda_d$. Hence, VeRA can be expressed as $\Delta\boldsymbol{W} = \Lambda_b\boldsymbol{B}\Lambda_d\boldsymbol{A}$, where $\boldsymbol{B}$ and $\boldsymbol{A}$ are randomly initialized, frozen and shared across layers.

Collectively, we denote the model parameters of transformer as $\boldsymbol{\Omega} = \{\{\boldsymbol{\mathcal{W}}_q^\ell, \boldsymbol{\mathcal{W}}_k^\ell, \boldsymbol{\mathcal{W}}_v^\ell, \boldsymbol{\mathcal{W}}_o^\ell\} \cup \{\boldsymbol{\mathcal{W}}_{\text{up}}^\ell, \boldsymbol{\mathcal{W}}_{\text{down}}^\ell\}\}_{\ell=1}^L \in \mathbb{R}^{12L\times d\times d}$. In the sequel, we propose a *global* reparameterization on the weight increments of $\boldsymbol{W} \in \boldsymbol{\Omega}$ based on the LoRA decomposition $\Delta\boldsymbol{W} = \boldsymbol{B}\boldsymbol{A}$. we will show how extreme parameter efficiency can be achieved by (1) parameter sharing across matrix dimensions of $\boldsymbol{A}$ and $\boldsymbol{B}$ based on a rank-one decomposition and sub-vector partitions (Sec. 5.3.2), and (2) across modules and layers regardless of the index or matrix type (Sec. 5.3.3).

### *5.3.2 Divide-and-Share: a New Paradigm for Parameter Sharing*

The low rank decomposition of LoRA can be *equivalently* expressed in a rank-one form as follows:

$$\Delta\boldsymbol{W} = \boldsymbol{B}\boldsymbol{A} = \sum_{k=1}^{r} \boldsymbol{b}_k \otimes \boldsymbol{a}_k = \sum_{k=1}^{r} \otimes_{i=1}^{2} \boldsymbol{v}_k^{(i)}, \quad \boldsymbol{v}_k^{(1)} = \boldsymbol{b}_k, \quad \boldsymbol{v}_k^{(2)} = \boldsymbol{a}_k, \tag{5.3.1}$$

where $\otimes$ denotes the outer product operator and $\boldsymbol{v}_k^{(i)}$ is a vector of size $d_i$.

*5.3.2.1 Divide*

Based on the rank-one decomposition above, we further represent each component vector $\boldsymbol{v}_k^{(i)}$ as a concatenation of a set of sub-vectors,

$$\boldsymbol{v}_k^{(i)} = \text{concat}(\mathbf{u}_{k,1}^{(i)}, \mathbf{u}_{k,2}^{(i)}, \ldots, \mathbf{u}_{k,d_i'}^{(i)}), \quad \boldsymbol{u}_{k,j}^{(i)} \in \mathbb{R}^b, \quad j \in \{1, \ldots, d_i'\}, \tag{5.3.2}$$

where $\{d_i\}_{i=1,2}$ represents the size of the matrix dimension of $\Delta\boldsymbol{W}$. In general, $\{d_i\}_{i=1,2}$ are not equal across $\boldsymbol{A}$ and $\boldsymbol{B}$, and we choose $b$ as a common factor of $d_i$ such that $d_i' = d_i/b$ and $d_i' \in \mathbb{Z}$.

Remarks The divide operator was first introduced in Quantized Tensor Train (QTT) for super compression of large-scale matrices (Oseledets 2010; Cichocki 2014). For example, dyadic division reshapes a vector of length $L = 2^p$ into a $p$-dimensional array which facilitates the efficient Tensor Train decomposition to be used. Our divide operator instead applies to the rank-one component vectors $\boldsymbol{v}_k^{(i)}$, and the resulting hierarchical tensorial representation of $\Delta\boldsymbol{W}$ can be viewed as a Canonical Polyadic Decomposition (CPD) (Kolda & Bader 2009) with component vectors $\boldsymbol{v}_k^{(i)}$ folded into 2- dimensional arrays with sub-vectors $\boldsymbol{u}_{k,j}^{(i)}$ as columns.

*5.3.2.2 Share*

To facilitate parameter sharing across model dimensions, we assume each sub-vector $\boldsymbol{u}_{k,j}^{(i)}$ as a top-$k$ admixture of basic elements from vector bank $\mathcal{B} = \{\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_h\}$, where $\boldsymbol{\alpha}_i \in \mathbb{R}^b$ for

$i \in \{1, \ldots, h\}$, and is defined as follows (with the subscripts omitted for clarity):

$$\boldsymbol{u} = \sum\nolimits_{s=1}^{h} w_s(\boldsymbol{\sigma})\boldsymbol{\alpha}_s, \quad \mathbf{w}(\boldsymbol{\sigma}) = \mathrm{Softmax}(\mathrm{TopK}(\boldsymbol{\sigma}, k)), \tag{5.3.3}$$

where $\mathrm{TopK}\left(\boldsymbol{\sigma}, k\right)_i = \sigma_i$ if $\sigma_i$ is among the top-$k$ of $\boldsymbol{\sigma}$ and $\mathrm{TopK}\left(\boldsymbol{\sigma}, k\right)_i = -\infty$ otherwise. For each sub-vector $\boldsymbol{u}$, we introduce logits $\boldsymbol{\sigma} \in \mathbb{R}^h$ as its learnable parameters. We call the model expressed in Eq. 5.3.3 as the *top-k admixture module* (TKAM), which is differentiable. This design enables the joint learning of vector bank $\mathcal{B}$ and logits $\boldsymbol{\sigma}$ in an end-to-end manner, which is amenable for model fine-tuning to the downstream tasks.

Remarks The TKAM module promotes sparsity by selecting $k$ vectors of the largest logits from the vector bank. By setting $k \ll h$, we restrict the sub-vector $\boldsymbol{u}$ to be sparse. That is, in each iteration, the updates to the vector bank remain locally dominated – with at most $k$ basis vectors $\boldsymbol{\alpha} \in \mathcal{B}$ affected by the backpropagation through $\boldsymbol{u}$ – in the hope that the learnt vectors can be more specialized and the knowledge encapsulated in the vector bank can be activated and updated sparsely.

The TKAM module can also be viewed as a factor model with simplex constraints on the mixing weight (e.g., $k = 2$, the sub-vector $\boldsymbol{u}$ lies on the edges of the simplex) and common factors stored in $\mathcal{B}$. Let $\boldsymbol{u} \in \mathbb{R}^b$ and $\boldsymbol{u} = \sum_{s=1}^{h} \boldsymbol{\alpha}_s w_s$, where $\boldsymbol{\alpha}_s$ is the $s$-th factor, and $\boldsymbol{w}$ is the factor score for the sub-vector $\boldsymbol{u}$. We consider the following options for $\boldsymbol{w}$: (1) Admixture (convex combination): $\boldsymbol{w} \in [0, 1]^h$ and $\sum_{s=1}^{h} w_s = 1$, which is commonly used in various communities. (2) Sparse Admixture (TKAM): $\boldsymbol{w} \in [0, 1]^h$ and $\sum_{s=1}^{h} w_s = 1$ with only $k \ll h$ non-zero elements allowed.

In addition, the Noisy Top-$k$ Gating module (Shazeer et al. 2016) has been widely used to replace the fully connected layers with the Mixture of Experts (MoE) layers in large language models (Jiang et al. 2024). In contrast, we use Eq. 5.3.3 to learn the selective sharing scheme across the rank-one component vectors without changing the original model. We find that the decomposed cumulative gradient parameter are more sensitive than the original model parameters during training process. Therefore, keeping zero noise in the gating function can help make the learning more efficient and stable. An ablation study of different vector selection methods is provided in Sec. 5.4.4.

### *5.3.3 Breaking Boundaries of LoRA for Global Parameter Sharing*

While LoRA only applies the low rank decomposition to each individual weight increment, the boundary can be broken by the *divide-and-share* scheme we proposed in Sec. 5.3.2. In our global parameter sharing scheme, each sub-vector $\boldsymbol{u}_{\mathbf{i}}$ is composed from a *globally* shared vector bank $\mathcal{B}$ via TKAM, where $\mathbf{i} = [\mathbf{j}, \mathbf{v}]$ is a multi-index including physical indices $\mathbf{j}$, such as module, layer, head, and left/right decomposed matrix, and virtual indices $\mathbf{v}$ (created from vector partition).

Specifically, LoRA provides a *local* low-rank factorization for each $d_1 \times d_2$ matrix $\Delta \boldsymbol{W}$ independently. In contrast, our VB-LoRA proposes a *global* low-rank factorization on a $b \times |\{\mathbf{i}\}|$ matrix, where $|\{\mathbf{i}\}|$ represents the cardinality of the index set including both physical and virtual indices. As we will see below, this differentiation can better leverage the redundancy in the cumulative gradients, leading to extreme parameter efficiency. It's worth mentioning that adding the multi-index information to the vector selection mechanism can

make the TKAM model structure-aware, potentially yielding additional benefits. But we will leave this as future work.

Figure 5.1 provides an overview of our method. The left section demonstrates the high-level idea of VB-LoRA: the vector bank is shared across sub-vectors, modules, and layers. The right section details the architecture of VB-LoRA. To form each sub-vector, we use a top-$k$ softmax function to select $k$ vectors from the vector bank, which are then pooled into a sub-vector. These sub-vectors are arranged in the desired positions, forming the parameters for LoRA with negligible computational overhead. Our method can be seamlessly integrated into the PyTorch framework and the HuggingFace PEFT library[1]. Algorithm 2 provides the PyTorch-like pseudocode for VB-LoRA.

### *5.3.4 Parameter Count*

In full fine-tuning, the number of trainable parameters is equal to the model size, i.e., $LMd^2$, where $L$ is the numebr of layers, $M$ is the number of fine-tuned modules, and $d$ is hidden dimension. LoRA reduces this number to $2LMdr$, while VeRA further reduces it to $LM(d+r)$. The trainable parameters of LoRA and VeRA are the same as the parameters they need to store.

In VB-LoRA, the trainable parameters consist of two parts: the parameters of the vector bank $\mathcal{B}$ and the parameters of logits $\boldsymbol{\sigma}$. However, at the end of training, the logit parameters can be discarded and only the $k$ selected indices and the top-$k$ admixture weights need to be stored. Therefore, the stored parameters can be represented by a triplet $\Theta = \{\mathcal{B}, \mathcal{I}, \mathcal{V}\}$,

[1] `https://github.com/huggingface/peft`

**Algorithm 2** Pseudocode of VB-LoRA in a PyTorch-like style

```
# d: hidden dimension; b: length of sub-vectors; r: rank; h: size of vector bank
# k: number of selected vectors used in the top-k admixture module
# logits: Each linear layer has two trainable parameters: logits_A and logits_B.
#         Both parameters have a shape of (d/b)*r*h.
# vector_bank: The shared vector bank with a shape of h*b.

def get_low_rank_matrix(logits, vector_bank, k):
    topk_logits, topk_indices = logits.topk(k, dim=-1)
    topk_weights = torch.softmax(topk_logits, dim=-1)
    matrix = (topk_weights * vector_bank[topk_indices]).sum(-2)
    return matrix

def VBLoRA_forward(vector_bank, logits_A, logits_B, k):
    r = logits_A.shape[1]
    A = get_low_rank_matrix(logits_A, vector_bank, k).transpose(0, 1).reshape(r,
    -1)
    B = get_low_rank_matrix(logits_B, vector_bank, k).transpose(1, 2).reshape(-1,
     r)
    delta_W = B @ A
    return delta_W
```

where $\mathcal{B} \in \mathbb{R}^{h\times b}$ is a vector bank containing $h$ vectors of $b$-dimensional, $\mathcal{I} \in \mathbb{R}^{L\times M\times r\times (d/b)\times k}$ is the top-$k$ indices of the vectors in $\mathcal{B}$ for all sub-vectors, and $\mathcal{V} \in \mathbb{R}^{L\times M\times r\times (d/b)\times (k-1)}$ is the top-$k$ admixture weights used to composite the sub-vectors from the bank. It is worth noting that the top-$k$ admixture weights have only $k-1$ degrees of freedom since they must be summed to 1. Additionally, depending on the size of the vector bank $h$, the indices $\mathcal{I}$ can be efficiently stored as unsigned integers (e.g., uint8 when $h \leq 256$), and hence, we count the number of parameters as the float32-equivalent size for a fair comparison. When we use $k = 2$ and uint8 for indices, the number of stored parameters of VB-LoRA is $hb + 1.5LMr(d/b)$.

Unlike LoRA and VeRA, the number of parameters in VB-LoRA does not increase linearly with the model size (determined by $L$ and $d$) or the number of fine-tuned modules, i.e., $M$.

While the second term of VB-LoRA's parameters is a linear function of $LMd$, the coefficient is $1.5r/b$, which is typically very small. For example, in our experiments, the typical values are $r = 4$ and $b = 256$, leading to a coefficient of 0.02, whereas the coefficient is $2r$ for LoRA and 1 for VeRA. Most of the parameters in VB-LoRA reside within the shared vector bank, whose size does not increase linearly with the model size or number of fine-tuned modules.

## 5.4 Experiments

In this section, we conduct a comprehensive evaluation of our method through a series of experiments. We begin by comparing VB-LoRA to the state-of-the-art PEFT methods: LoRA, VeRA, and Tied-LoRA on the GLUE benchmark. Next, we extend our analysis to natural language generation tasks using GPT-2, as well as instruction tuning tasks on the Llama2 models with 7B and 13B parameters. All our experiments were conducted on a server equipped with 8 A100 GPUs. The hyperparameters used for the natural language understanding, natural language generation and instruction tuning are provided in Table 5.1, 5.2 and 5.3. All experiments were conducted on a server equipped with 8 A100 80GB GPUs.

### *5.4.1 Natural Language Understanding*

We adopt the General Language Understanding Evaluation (GLUE) benchmark[2] (Wang et al. 2018) to assess the performance of VB-LoRA across various natural language understanding tasks, including similarity, paraphrase, and inference tasks. Following Kopiczko

[2] `https://gluebenchmark.com/`

Table 5.1: Hyperparameters and computing resources for natural language understanding experiments on the GLUE benchmark.

| Model | Hyperparameter | SST-2 | MRPC | CoLA | QNLI | RTE | STS-B |
|---|---|---|---|---|---|---|---|
| | Optimizer | AdamW | | | | | |
| | Warmup Ratio | 0.06 | | | | | |
| | LR Schedule | Linear | | | | | |
| | Init. of the Vector Bank | $\mathcal{U}(-0.02, 0.02)$ | | | | | |
| | Init. of the Logits | $\mathcal{N}(0, 0.01)$ | | | | | |
| BASE | # GPUs | 1 | | | | | |
| | Epochs | 60 | 30 | 80 | 25 | 160 | 80 |
| | Learning Rate (Head) | 4E-3 | 4E-3 | 2E-2 | 1E-2 | 2E-2 | 2E-2 |
| | Learning Rate (Logits) | 1E-2 | | | | | |
| | Learning Rate (Vector Bank) | 1E-3 | | | | | |
| | Vector Bank Size | 90 | | | | | |
| | Vector Length | 256 | | | | | |
| | Rank | 4 | | | | | |
| | Max Seq. Len. | 512 | | | | | |
| | Batch Size Per GPU | 32 | | | | | |
| | Training Time | 8h / 10h | 27m / 40m | 80m / 100m | 5h / 6.5h | 50m / 1h | 1h / 80m |
| | GPU Memory | 24,552 MiB / 28,120 MiB | | | | | |
| LARGE | # GPUs | 1 | | | | | |
| | Epochs | 20 | 40 | 40 | 20 | 40 | 40 |
| | Learning Rate (Head) | 3E-3 | 3E-3 | 3E-3 | 2E-3 | 2E-3 | 6E-3 |
| | Learning Rate (Logits) | 1E-2 | | | | | |
| | Learning Rate (Vector Bank) | 1E-3 | | | | | |
| | Vector Bank Size | 90 | | | | | |
| | Vector Length | 256 | | | | | |
| | Rank | 4 | | | | | |
| | Max Seq. Len. | 128 | | | | | |
| | Batch Size Per GPU | 32 | | | | | |
| | Training Time | 2h / 3h | 12m / 20m | 30m / 45m | 3h / 4.5h | 10m / 15m | 20m / 30m |
| | GPU Memory | 9,804 MiB / 12,170 MiB | | | | | |

Training time and GPU memory are reported as "query and value only" / "all linear modules". h: hour, m: minute.

et al. (2024), we focus on six tasks from GLUE: CoLA (Warstadt et al. 2019) (linguistic acceptability), SST-2 (Socher et al. 2013) (sentiment analysis), MRPC (Dolan & Brockett 2005) (paraphrase detection), STS-B (Cer et al. 2017) (semantic textual similarity), QNLI (Rajpurkar et al. 2018) (inference), and RTE (inference).

Our experiments are performed with RoBERTa$_{\text{base}}$ and RoBERTa$_{\text{large}}$ (Liu et al. 2019). While LoRA and VeRA only finetune the query and value modules, we explore two finetuning strategies: query and value only (VB-LoRA$_{\text{qv}}$), and all linear modules (VB-LoRA$_{\text{all}}$), including $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v, \boldsymbol{W}_o$, $\boldsymbol{W}_{\text{up}}$, and $\boldsymbol{W}_{\text{down}}$. We create a vector bank of 90 vectors of a length of 256, initialized with a uniform distribution $\mathcal{U}(-0.02, 0.02)$. The logits are initialized

Table 5.2: Hyperparameters and computing resources on natural language generation experiments on the E2E dataset.

| Hyperparameter | Medium | Large |
|---|---|---|
| # GPUs | 1 | |
| Optimizer | AdamW | |
| Learning Rate Schedule | Linear | |
| Weight Decay | 0.01 | |
| Batch Size | 8 | |
| Epochs | 5 | |
| Warmup Steps | 500 | |
| Label Smooth | 0.1 | |
| Rank | 4 | |
| Vector Length | 256 | |
| Vector Bank Size | 256 | 350 |
| Learning Rate (Vector Bank) | 1E-3 | 1E-3 |
| Learning Rate (Logits) | 1E-2 | 1E-2 |
| Training Time | 3h | 3h |
| GPU Memory | 29,061 MiB | 29,282 MiB |

Training time and GPU memory are reported as "query and value only" / "all linear modules". h: hour, m: minute.

Table 5.3: Hyperparameters and computing resources on instruction tuning on the Cleaned Alpaca Dataset.

| Hyperparameter | LoRA, 7B | LoRA, 13B | VB-LoRA, 7B | VB-LoRA, 13B |
|---|---|---|---|---|
| # GPUs | 1 | | | |
| Optimizer | AdamW | | | |
| Warmup Ratio | 0.1 | | | |
| Batch Size | 4 | | | |
| Accumulation Steps | 4 | | | |
| Epochs | 1 | | | |
| LR Schedule | Linear | | | |
| Vector Length | N/A | N/A | 256 | 256 |
| Rank | 64 | 64 | 4 | 6 |
| Vector Bank Size | N/A | N/A | 2048 | 2048 |
| Learning Rate (Vector bank) | N/A | N/A | 1E-3 | 1E-3 |
| Learning Rate (Logits) | N/A | N/A | 1E-2 | 1E-2 |
| Learning Rate (LoRA) | 4e-4 | 4e-4 | N/A | N/A |
| Training Time | 2h | 2.6h | 2.5h | 3h |
| GPU Memory | 8,467 MiB | 11,624 MiB | 6,872 MiB | 11,486 MiB |

h: hour. 7B: llama2 7B, 13B: llama2 13B.

with a normal distribution $\mathcal{N}(0, 0.01)$. The learning rates for the vector bank and logit parameters are set to 0.001 and 0.01, respectively. We set the rank to 4 and $k = 2$ for all our experiments.

Table 5.4 reveals that VB-LoRA achieves competitive or superior performance compared to VeRA and Tied-LoRA, while being more parameter efficient. For example, when fine-

Table 5.4: Results with RoBERTa$_{\text{base}}$ and RoBERTa$_{\text{large}}$ on the GLUE benchmark.

| | Method | # Params | SST-2 | MRPC | CoLA | QNLI | RTE | STS-B | Avg. |
|---|---|---|---|---|---|---|---|---|---|
| Base | FT | 125M | 94.8 | 90.2 | 63.6 | 92.8 | 78.7 | 91.2 | 85.2 |
| | LoRA$_{\text{qv}}$ | 0.295M | 95.1$_{\pm 0.2}$ | 89.7$_{\pm 0.7}$ | 63.4$_{\pm 1.2}$ | 93.3$_{\pm 0.3}$ | 86.6$_{\pm 0.7}$ | 91.5$_{\pm 0.2}$ | 86.6 |
| | VeRA$_{\text{qv}}$ | 0.043M | **94.6**$_{\pm 0.1}$ | **89.5**$_{\pm 0.5}$ | **65.6**$_{\pm 0.8}$ | 91.8$_{\pm 0.2}$ | 78.7$_{\pm 0.7}$ | 90.7$_{\pm 0.2}$ | 85.2 |
| | Tied-LoRA$_{\text{qv}}$ | 0.043M | 94.4$_{\pm 0.5}$ | 88.5$_{\pm 1.0}$ | 61.9$_{\pm 1.6}$ | 92.0$_{\pm 0.1}$ | 76.2$_{\pm 1.0}$ | 89.8$_{\pm 0.3}$ | 83.8 |
| | VB-LoRA$_{\text{qv}}$ ***(Ours)*** | **0.023M** | 94.4$_{\pm 0.2}$ | **89.5**$_{\pm 0.5}$ | 63.3$_{\pm 0.7}$ | **92.2**$_{\pm 0.2}$ | **82.3**$_{\pm 1.3}$ | **90.8**$_{\pm 0.1}$ | **85.4** |
| | VeRA$_{\text{all}}$ | 0.157M | **95.1**$_{\pm 0.4}$ | 88.7$_{\pm 0.5}$ | 64.5$_{\pm 1.0}$ | 92.3$_{\pm 0.2}$ | 81.9$_{\pm 1.4}$ | 90.2$_{\pm 0.3}$ | 85.5 |
| | Tied-LoRA$_{\text{all}}$ | 0.109M | 94.7$_{\pm 0.2}$ | 88.5$_{\pm 0.8}$ | **64.7**$_{\pm 0.8}$ | **92.4**$_{\pm 0.1}$ | 76.5$_{\pm 1.3}$ | 90.3$_{\pm 0.1}$ | 84.5 |
| | VB-LoRA$_{\text{all}}$ ***(Ours)*** | **0.027M** | 95.0$_{\pm 0.2}$ | **89.7**$_{\pm 0.2}$ | 64.3$_{\pm 1.4}$ | 92.3$_{\pm 0.2}$ | **82.3**$_{\pm 0.9}$ | **90.7**$_{\pm 0.2}$ | **85.7** |
| Large | LoRA$_{\text{qv}}$ | 0.786M | 96.2$_{\pm 0.5}$ | 90.2$_{\pm 1.0}$ | 68.2$_{\pm 1.9}$ | 94.8$_{\pm 0.3}$ | 85.2$_{\pm 1.1}$ | 92.3$_{\pm 0.5}$ | 87.8 |
| | VeRA$_{\text{qv}}$ | 0.061M | **96.1**$_{\pm 0.1}$ | 90.9$_{\pm 0.7}$ | 68.0$_{\pm 0.8}$ | 94.4$_{\pm 0.2}$ | 85.9$_{\pm 0.7}$ | 91.7$_{\pm 0.8}$ | 87.8 |
| | Tied-LoRA$_{\text{qv}}$ | 0.066M | 94.8$_{\pm 0.6}$ | 89.7$_{\pm 1.0}$ | 64.7$_{\pm 1.2}$ | 94.1$_{\pm 0.1}$ | 81.2 $_{\pm 0.1}$ | 90.8 $_{\pm 0.3}$ | 85.9 |
| | VB-LoRA$_{\text{qv}}$ ***(Ours)*** | **0.024M** | **96.1**$_{\pm 0.2}$ | **91.4**$_{\pm 0.6}$ | **68.3**$_{\pm 0.7}$ | **94.7**$_{\pm 0.5}$ | **86.6**$_{\pm 1.3}$ | **91.8**$_{\pm 0.1}$ | **88.2** |
| | VeRA$_{\text{all}}$ | 0.258M | **96.6**$_{\pm 0.5}$ | 90.9$_{\pm 0.8}$ | 68.5$_{\pm 1.4}$ | **94.4**$_{\pm 0.4}$ | 85.9$_{\pm 1.2}$ | **92.2**$_{\pm 0.2}$ | 88.1 |
| | Tied-LoRA$_{\text{all}}$ | 0.239M | 94.8$_{\pm 0.3}$ | 90.0$_{\pm 0.4}$ | 66.8$_{\pm 0.1}$ | 94.1$_{\pm 0.1}$ | 82.3$_{\pm 2.0}$ | 91.6$_{\pm 0.2}$ | 86.6 |
| | VB-LoRA$_{\text{all}}$ ***(Ours)*** | **0.033M** | 96.3$_{\pm 0.2}$ | **91.9**$_{\pm 0.9}$ | **69.3**$_{\pm 1.5}$ | **94.4**$_{\pm 0.2}$ | **87.4**$_{\pm 0.7}$ | 91.8$_{\pm 0.2}$ | **88.5** |

The best results in each group are shown in **bold**. We report Matthew's correlation for CoLA, Pearson correlation for STS-B, and accuracy for all other datasets. Results for LoRA$_{\text{qv}}$ and VeRA$_{\text{qv}}$ are sourced from their respective original papers, while the other results are based on our implementations. We report the median performance from 5 runs using different random seeds.

tuning the query and value modules on the RoBERTa$_{\text{large}}$ model, our method reduces the stored parameters to less than 40% of those required by VeRA or Tied-LoRA, while outperforming them across all tasks.

Moreover, the results consistently indicate that fine-tuning all modules, beyond just the query and value modules, enhances performance for all the methods. However, LoRA, VeRA and Tied-LoRA requires 2–4 times of the parameters in this case because their parameter counts increase linearly with the number of fine-tuned modules. In contrast, our method uses only 37.5% additional parameters as we maintain the same vector bank size but add additional parameters for indices and top-$k$ weights. Thus, with only 12.8% of the parameters compared to VeRA$_{\text{all}}$ (4% compared to LoRA$_{\text{qv}}$), our method achieves the best average performance.

Table 5.5: Results with GPT-2 Medium and GPT-2 Large on the E2E benchmark.

| | Method | # Params | BLEU | NIST | METEOR | ROUGE-L | CIDEr |
|---|---|---|---|---|---|---|---|
| Medium | FT | 354.92M | 68.2 | 8.62 | 46.2 | 71.0 | 2.47 |
| | LoRA | 0.35M | 68.9 | 8.69 | 46.4 | 71.3 | 2.51 |
| | VeRA | 0.098M | **70.1** | **8.81** | **46.6** | **71.5** | 2.50 |
| | VB-LoRA ***(Ours)*** | **0.076M** | 70.0 | **8.81** | **46.6** | **71.5** | **2.52** |
| Large | FT | 774.03M | 68.5 | 8.78 | 46.0 | 69.9 | 2.45 |
| | LoRA | 0.77M | 70.1 | 8.80 | 46.7 | 71.9 | 2.52 |
| | VeRA | 0.17M | **70.3** | 8.85 | **46.9** | 71.6 | **2.54** |
| | VB-LoRA ***(Ours)*** | **0.13M** | **70.3** | **8.86** | 46.7 | **72.2** | **2.54** |

The results for FT and LoRA are taken from Hu et al. (2021), and the results for VeRA are taken from Kopiczko et al. (2024). We report the mean of 3 runs using different random seeds.

### *5.4.2 Natural Language Generation*

For natural language generation experiments, we fine-tune the GPT-2 Medium and Large models Radford et al. (2019) on the E2E dataset[3] (Novikova et al. 2017), which contains approximately 42,000 training examples, 4,600 validation examples, and 4,600 test examples from the restaurant domain. We use a vector bank of size 256 for GPT-2 Medium and 350 for GPT-2 Large. The vector length is set to 256 and the rank is set to 4 for both models. To achieve the best performance, we fine-tune all attention layers and FFN layers. As shown in Table 5.5, our approach achieves competitive performance compared to VeRA, while requiring about 20% less stored parameters for both models.

### *5.4.3 Instruction Tuning*

Instruction tuning is a process of fine-tuning model with a set of instructions or prompts to enhance its performance on specific instructions (Ouyang et al. 2022). Following Kopiczko et al. (2024), we fine-tune the Llama2 model (Touvron et al. 2023) within the QLoRA (Dettmers et al. 2023) framework[4], which aims to reduce memory usage when fine-tuning large language

[3]Licensed under CC BY-SA 4.0. URL: `https://github.com/tuetschek/e2e-dataset`
[4]`https://github.com/artidoro/qlora`

models on a single GPU. We utilize the quantization strategy provided by QLoRA, including 4-bit NormalFloat for storage data, BFloat16 for computation parameters, double quantization and paged optimizers to train it on a single GPU. We use the Cleaned Alpaca Dataset [5], which improves the data quality of the original Alpaca dataset (Taori et al. 2023). We evaluate the fine-tuned models on the MT-Bench[6] (Zheng et al. 2024), which contains 80 multi-turn questions. Our fine-tuned models generate responses to these questions, and subsequently, GPT-4 is employed to review and evaluate the generated answers, assigning a quantitative score on a scale of 10. Note that aligning with VeRA, we report the score of the first turn of the conversation. We apply VB-LoRA to all linear layers except the top one, following Kopiczko et al. (2024). For Llama2 7B, we use a vector bank of 2,048 vectors, each with a length of 256, and the rank is set to 4, resulting in a total of 0.8M stored parameters. For Llama2 13B, we use the same-sized vector bank but increase the rank to 6, leading to 1.1M stored parameters. For all the experiments, we train for one epoch.

The results are reported in Table 5.6. Notably, we report two sets of LoRA results for each experiment: one from our implementation and the other from Kopiczko et al. (2024), due to a noticeable discrepancy between the scores. Since we closely follow the experimental settings of Kopiczko et al. (2024), we speculate that the difference is due to changes in the GPT-4 model over time. However, comparing the relative improvements of VeRA and VB-LoRA with their respective implementations of LoRA remains fair. VB-LoRA achieves

[5]The original and cleaned Alpaca datasets are licensed under CC BY-NC 4.0. URLs: `https://huggingface.co/datasets/tatsu-lab/alpaca`, `https://huggingface.co/datasets/yahma/alpaca-cleaned`

[6]Licensed under CC BY 4.0. URL: `https://huggingface.co/datasets/lmsys/mt_bench_human_judgments`

higher scores than LoRA while using only 0.5% (Llama2 7B) and 0.4% (Llama2 13B) of the stored parameters. While VeRA can reach similar scores with their implementation of LoRA, it requires more than twice of parameters compared to VB-LoRA.

Table 5.6: Results with Llama2 on the MT-Bench dataset.

| Model | Method | # Parameters | Score |
|---|---|---|---|
| Llama2 7B | w/o FT | - | 4.79 |
| | LoRA† | 159.9M | 5.19 |
| | VeRA | 1.6M | 5.08 |
| | LoRA‡ | 159.9M | 5.63 |
| | VB-LoRA ***(Ours)*** | **0.8M** | **5.71** |
| Llama2 13B | w/o FT | - | 5.38 |
| | LoRA† | 250.3M | 5.77 |
| | VeRA | 2.4M | 5.93 |
| | LoRA‡ | 250.3M | 6.13 |
| | VB-LoRA ***(Ours)*** | **1.1M** | **6.31** |

The scores are assigned by GPT-4 on a scale of 10. LoRA† and VeRA are sourced from Kopiczko et al. (2024). LoRA‡ and VB-LoRA are from our implementations. The discrepancy between LoRA† and LoRA‡ may be due to changes in the GPT-4 model over time.

### *5.4.4 Ablation Study*

We conduct an ablation study to examine the impact of each individual component of VB-LoRA. The experiments are performed on RoBERTa-large, fine-tuning only the query and value modules.

#### *5.4.4.1 Vector Selection Methods*

Besides the top-$k$ admixture module (abbreviated as Top-$k$ below), there exist several commonly used discrete optimization methods for vector selection, including Noisy Top-$k$ (Shazeer et al. 2016), Gumbel-Softmax (GS), and Straight-Through Gumbel-Softmax (Jang et al. 2016; Maddison et al. 2016). For Top-$k$ and Noisy Top-$k$, we evaluate the impact of

different $k$ to the performances on the CoLA dataset. For GS and Straight-Through GS, we set the temperature $\tau = 1/3$ during training and use Top-1 and Top-2 Softmax for inference. Additionally, we explore "Select All", a special case of Top-$k$ with $k$ equals to the vector bank size $h$. As shown in Table 5.7, Noisy Top-$k$, GS, and Straight-Through GS significantly underperform Top-$k$ and "Select All". We hypothesize that random noise injected by these methods likely disrupts the parameters of vector bank, leading to instability in the learning process.

We further investigate the impact of $k$ to the training dynamics and performance of VB-LoRA. As discussed in Sec. 5.3.4, the choice of $k$ affects not only the model's performance but also the number of parameters to be stored. Hence, a smaller $k$ is generally preferred for improved parameter efficiency. Table 5.7 shows that $k = 2$ yields the best result on CoLA, whereas $k = 1$ performs significantly worse. To explain this, we delve into the training dynamics of VB-LoRA. As shown in Figure 5.3 (a), when $k = 1$, the selected vectors remain largely unchanged during training. In contrast, when $k > 1$, the model actively explore the vector bank as illustrated in Figure 5.3 (b) and (c), i.e., different vectors are selected and updated actively during the training process.

#### *5.4.4.2 Sub-vector Length $b$*

VB-LoRA introduces a new virtual dimension $b$ that divides the original dimensions of LoRA matrices into sub-vectors. As discussed in Sec. 5.3.2, using finer granularity in sub-vectors promotes sharing across modules and layers, leading to extreme parameter efficiency. Furthermore, such a division is also necessary when the weight matrices have different shapes.

Table 5.7: Ablation study of different vector selection methods.

| Method | Training | Inference | CoLA |
|---|---|---|---|
| Select All | S. | S. | $67.5_{\pm 1.2}$ |
| Top-$k$ | Top 1 S. | Top 1 S. | $66.9_{\pm 0.5}$ |
| | Top 2 S. | Top 2 S. | $\mathbf{68.3}_{\pm 0.7}$ |
| | Top 3 S. | Top 3 S. | $68.1_{\pm 1.3}$ |
| | Top 6 S. | Top 6 S. | $67.1_{\pm 0.5}$ |
| Noisy Top-$k$ | Noisy Top 1 S. | Top 1 S. | $45.3_{\pm 2.2}$ |
| | Noisy Top 2 S. | Top 2 S. | $62.6_{\pm 0.2}$ |
| GS | GS ($\tau$=1/3) | Top 1 S. | $57.1_{\pm 0.6}$ |
| | GS ($\tau$=1/3) | Top 2 S. | $57.3_{\pm 1.6}$ |
| ST-GS | ST-GS ($\tau$=1/3) | Top 1 S. | $55.6_{\pm 1.6}$ |
| | ST-GS ($\tau$=1/3) | Top 2 S. | $54.7_{\pm 1.2}$ |

S.: Softmax, GS: Gumbel-Softmax, ST-GS: Straight Through Gumbel-Softmax.

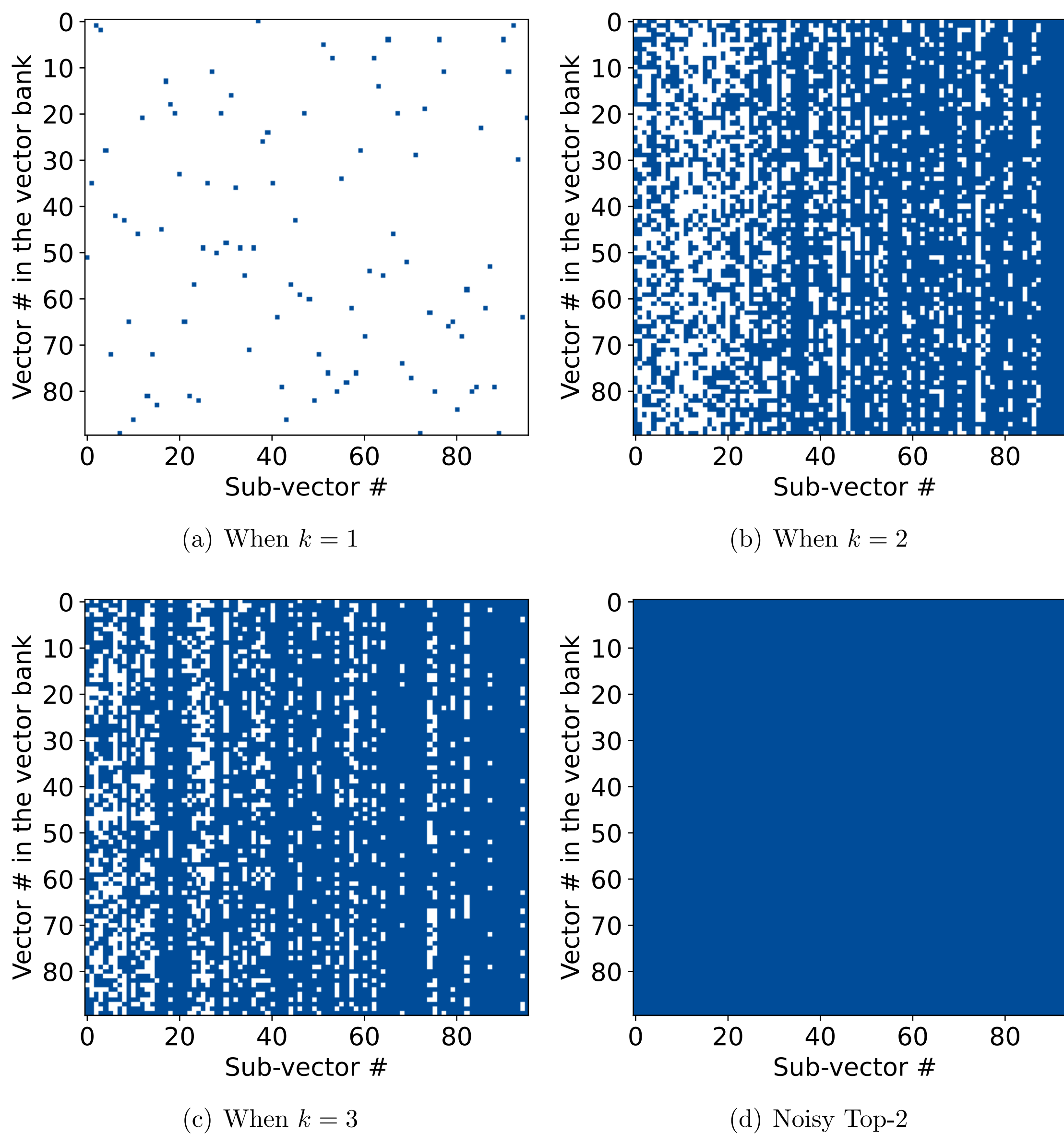
Table 5.8: Ablation study of sub-vector length.

| Length $b$ | Vector Bank Size | CoLA |
|---|---|---|
| 128 | 240 | $67.0_{\pm 0.8}$ |
| 256 | 120 | $\mathbf{68.7}_{\pm 0.7}$ |
| 512 | 60 | $67.8_{\pm 0.8}$ |
| 1024 | 30 | $67.3_{\pm 1.1}$ |

As shown in Table 5.8, we maintain the same number of parameters in the vector bank while varying the sub-vector length $b$. The best performance is achieved with a sub-vector length of 256.

## 5.5 Conclusion

This paper introduces a "divide-and-share" paradigm and a differentiable top-$k$ admixture module for extreme parameter-efficient fine-tuning with vector banks. Our proposed VB-LoRA achieves the competitive or higher accuracy while using significantly smaller number of stored parameters compared to the state-of-the-art PEFT methods, including LoRA, VeRA, and Tied-LoRA. In addition, VB-LoRA is model-agnostic and applicable to other PEFT methods (Ding et al. 2023), including inserted adapters (Karimi Mahabadi et al. 2021),




(a) When $k = 1$

(b) When $k = 2$

(c) When $k = 3$

(d) Noisy Top-2

Figure 5.3: VB-LoRA's vector selection footprints during training.
The x-axis represents the 96 sub-vectors formed by the vectors from a bank of 90 vectors, while the y-axis represents the indices of selected vectors from the bank. The blue blocks indicate the selection footprint during training.

prompt tuning (Qin et al. 2021), and BitFit (Ben Zaken et al. 2022). Although VB-LoRA focuses on reducing the storage and transmission costs for LLM fine-tuning, we believe the proposed scheme can be extended to memory-efficient fine-tuning and parameter-efficient pre-training. But we leave these for future exploration.

# CHAPTER 6
# CONCLUSION AND FUTURE WORK

This dissertation revolves around the efficiency of deep neural networks. In Chapter 2, we proposed an $L_0$ regularization-based network pruning method, $L_0$-ARM. In Chapter 3, we introduced Neural Pruning Networks (NPNs), which unify network sparsification and network expansion in an end-to-end training pipeline. In Chapter 4, we examined the limitations of existing pruning algorithms on large datasets and proposed Dep-$L_0$, which enhances $L_0$-based network pruning algorithms through dependency modeling. In Chapter 5, we presented a "divide-and-share" paradigm and a differentiable top-$k$ admixture module for extreme parameter-efficient fine-tuning with vector banks.

For future work, several promising directions can be pursued. One potential avenue is the application of the $L_0$-based pruning method to more diverse and complex neural network architectures, such as transformers. Another direction involves extending the "divide-and-share" paradigm to multi-task learning and distributed training scenarios, potentially unveiling new strategies for parameter-efficient transfer learning and adaptation.

## 6.1 Pruning Transformer-based Models

The transformer (Vaswani et al. 2017) architecture has been widely used for natural language processing (NLP) tasks over the past years, significantly improving the performance of Neural Machine Translation (NMT) tasks. Moreover, the transformer structure has also been applied to the computer vision area, such as image classification (Touvron et al. 2021), object detection (Carion et al. 2020), and video segmentation (Wang et al. 2021). However,

the growing size of transformer models makes them challenging to deploy on smartphones, drones, and Internet of Things (IoT) devices. Thus, compressing transformer models is a critical problem. There are existing works on pruning transformer models (Fan et al. 2019; Behnke & Heafield 2020; Zhu et al. 2021). For instance, Zhu et al. (2021) proposed a method to prune Vision Transformer (ViT).

In terms of pruning granularity level, most methods fall into four categories: layer-wise, head-wise, line-wise, and element-wise. (1) Layer-wise pruning takes transformer layers as the smallest granularity. For example, Fan et al. (2019) proposed LayerDrop, which reduces the depth of transformer layers at inference time. (2) Head-wise pruning takes attention heads as the smallest granularity. One example is Behnke & Heafield (2020), which applies the lottery ticket hypothesis to prune attention heads in the early stages of training. (3) Line-wise pruning focuses on rows or columns in Transformer layer matrices. (4) Element-wise pruning uses random weights as the smallest pruning granularity. For example, Cheong (2019) prunes based on the magnitude of weights.

In our previous work, we mainly focused on pruning Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN). However, we can naturally extend our $L_0$-based pruning method to transformers. We can attach binary masks to transformers at all different granularity levels and penalize the number of pruning units by adding $L_0$-norm regularization. Considering the large scale of transformers, we may benefit from dependency learning and potentially improve the accuracy of the pruned network.

### 6.2 Multi-task Parameter-efficient Fine-tuning

Fine-tuning a pre-trained model requires making design choices about which layers of the model should be frozen or updated. Multitask fine-tuning adds extra complexity about which parameters should be shared or task-specific. Along this line of work, Polytropon (Ponti et al. 2022) jointly learns a small inventory of LoRA adapters and a routing function that selects a variable-sized subset of adapters for few-shot adaptation. Caccia et al. (2023) emphasize the importance of routing granularity and further propose a finer-grained mixing across multiple heads. Following these works, it would be interesting to explore a finer-grained parameter transfer across tasks, heads, types, and layers at the sub-vector level for multitask fine-tuning.

### 6.3 Distributed Parameter-efficient Fine-tuning

Distributed parameter-efficient fine-tuning is an emerging area that addresses the challenge of fine-tuning large-scale models across multiple devices or nodes in a distributed system. This approach leverages the strengths of distributed computing to enable efficient training and fine-tuning of deep learning models, particularly in scenarios where computational resources are spread across various locations.

One promising direction in this area is to develop techniques that can distribute the fine-tuning process across multiple devices while minimizing communication overhead and ensuring consistent model updates. For instance, federated learning paradigms can be adapted to fine-tune models in a decentralized manner, where each node performs local updates on its

subset of data and shares only the necessary gradients or model parameters with a central server (McMahan et al. 2017). This approach not only reduces the communication burden but also enhances data privacy. It would be interesting to extend our proposed VB-LoRA to this new scenario, as the vector bank is a compact representation of model parameters.

# REFERENCES


Aghajanyan, A., Zettlemoyer, L., & Gupta, S. 2020, arXiv preprint arXiv:2012.13255

Akaike, H. 1998, Selected Papers of Hirotugu Akaike (Springer), 199–213

Arora, S., Ge, R., Halpern, Y., Mimno, D., Moitra, A., Sontag, D., Wu, Y., & Zhu, M. 2013, in International Conference on Machine Learning, PMLR, 280–288

Ash, T. 1989, Connection Science, 1, 365–375

Behnke, M., & Heafield, K. 2020, in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2664–2674

Ben Zaken, E., Goldberg, Y., & Ravfogel, S. 2022, in Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ed. S. Muresan, P. Nakov, & A. Villavicencio, Dublin, Ireland, 1–9

Bengio, Y., Leonard, N., & Courville, A. 2013, arXiv preprint arXiv:1308.3432

Bird, T., Kunze, J., & Barber, D. 2018, arXiv preprint arXiv:1809.04855

Bishop, C. M. 2007, Pattern Recognition and Machine Learning (Information Science and Statistics) (Springer)

Blei, D. M., Kucukelbir, A., & McAuliffe, J. D. 2017, Journal of the American Statistical Association, 859

Borzunov, A., Ryabinin, M., Chumachenko, A., Baranchuk, D., Dettmers, T., Belkada, Y., Samygin, P., & Raffel, C. A. 2024, Advances in Neural Information Processing Systems, 36

Brown, T. et al. 2020, Advances in Neural Information Processing Systems, 33, 1877

Caccia, L., Ponti, E., Su, Z., Pereira, M., Le Roux, N., & Sordoni, A. 2023, in Advances in Neural Information Processing Systems

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. 2020, in European conference on computer vision, Springer, 213–229

Cer, D., Diab, M., Agirre, E., Lopez-Gazpio, I., & Specia, L. 2017, in Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017), ed. S. Bethard, M. Carpuat, M. Apidianaki, S. M. Mohammad, D. Cer, & D. Jurgens, Vancouver, Canada, 1–14

Chan, T.-H., Chi, C.-Y., Huang, Y.-M., & Ma, W.-K. 2009, IEEE Transactions on Signal Processing, 57, 4418

Cheong, R. 2019

Chin, T.-W., Ding, R., Zhang, C., & Marculescu, D. 2019, arXiv, arXiv

Cichocki, A. 2014, arXiv preprint arXiv:1403.2048

Cortes, C., Gonzalvo, X., Kuznetsov, V., Mohri, M., & Yang, S. 2017, in International Conference on Machine Learning (ICML)

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., & Bengio, Y. 2016, arXiv preprint arXiv:1602.02830

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. 2009a, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. 2009b, in 2009 IEEE conference

on computer vision and pattern recognition, Ieee, 248–255

Deng, L., Li, G., Han, S., Shi, L., & Xie, Y. 2020, in Proceedings of the IEEE, Vol. 108, 485–532

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., & Fergus, R. 2014, in Advances in neural information processing systems, 1269–1277

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. 2023, in Advances in Neural Information Processing Systems, ed. A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine, Vol. 36, 10088–10115

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. 2018, arXiv preprint arXiv:1810.04805

Ding, C. H., Li, T., & Jordan, M. I. 2008, IEEE transactions on pattern analysis and machine intelligence, 32, 45

Ding, N. et al. 2023, Nature Machine Intelligence, 5, 220

Ding, X., Ding, G., Guo, Y., Han, J., & Yan, C. 2019a, arXiv preprint arXiv:1905.04748

Ding, X., Ding, G., Zhou, X., Guo, Y., Han, J., & Liu, J. 2019b, in Advances in neural information processing systems

Dolan, W. B., & Brockett, C. 2005, in Proceedings of the Third International Workshop on Paraphrasing (IWP2005)

Dong, X., & Yang, Y. 2019, in Advances in Neural Information Processing Systems, 760–771

Fan, A., Grave, E., & Joulin, A. 2019, arXiv preprint arXiv:1909.11556

Fu, X., Ma, W.-K., Huang, K., & Sidiropoulos, N. D. 2015, IEEE Transactions on Signal Processing, 63, 2306

Gale, T., Elsen, E., & Hooker, S. 2019a, arXiv preprint arXiv:1902.09574

——. 2019b, arXiv preprint arXiv:1902.09574

Gers, F. A., Schmidhuber, J., & Cummins, F. 1999

Grathwohl, W., Choi, D., Wu, Y., Roeder, G., & Duvenaud, D. 2018, in International Conference on Learning Representations (ICLR)

Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. 2015, in International Conference on Machine Learning, 1737–1746

Han, S., Mao, H., & Dally, W. J. 2016, in International Conference on Learning Representations (ICLR)

Han, S., Pool, J., Tran, J., & Dally, W. 2015, in Advances in neural information processing systems, 1135–1143

He, J., Zhou, C., Ma, X., Berg-Kirkpatrick, T., & Neubig, G. 2021, in International Conference on Learning Representations

He, K., Zhang, X., Ren, S., & Sun, J. 2016a, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778

He, K., Zhang, X., Ren, S., & Sun, J. 2016b, in Proceedings of the IEEE conference on computer vision and pattern recognition, 770–778

He, Y., Kang, G., Dong, X., Fu, Y., & Yang, Y. 2018a, arXiv preprint arXiv:1808.06866

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., & Han, S. 2018b, in Proceedings of the European Conference on Computer Vision (ECCV), 784–800

He, Y., Liu, P., Wang, Z., Hu, Z., & Yang, Y. 2019, in Proceedings of the IEEE Conference

on Computer Vision and Pattern Recognition, 4340–4349

Hinton, G., Vinyals, O., & Dean, J. 2015, arXiv preprint arXiv:1503.02531

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., & Gelly, S. 2019, in International Conference on Machine Learning, PMLR, 2790–2799

Hu, E. J., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. 2021, in International Conference on Learning Representations

Hu, H., Peng, R., Tai, Y.-W., & Tang, C.-K. 2016, arXiv preprint arXiv:1607.03250

Huang, G., Liu, Z., van der Maaten, L., & Weinberger, K. Q. 2017, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Huang, Z., & Wang, N. 2018, in Proceedings of the European conference on computer vision (ECCV), 304–320

Inouye, D., Ravikumar, P., & Dhillon, I. 2014, in International Conference on Machine Learning, PMLR, 683–691

Ioffe, S., & Szegedy, C. 2015, International Conference on Machine Learning (ICML)

Jaderberg, M., Vedaldi, A., & Zisserman, A. 2014, arXiv preprint arXiv:1405.3866

Jang, E., Gu, S., & Poole, B. 2016, arXiv preprint arXiv:1611.01144

Jang, E., Gu, S., & Poole, B. 2017, in International Conference on Learning Representations (ICLR)

Jiang, A. Q. et al. 2024, arXiv preprint arXiv:2401.04088

Jie, S., & Deng, Z.-H. 2023in , 1060–1068

Karimi Mahabadi, R., Henderson, J., & Ruder, S. 2021, Advances in Neural Information Processing Systems, 34, 1022

Khoromskij, B. N. 2011, Constructive Approximation, 34, 257

Kingma, D. P., & Ba, J. 2015, in International Conference on Learning Representations (ICLR)

Kingma, D. P., Salimans, T., & Welling, M. 2015, in Advances in Neural Information Processing Systems, 2575–2583

Kolda, T. G., & Bader, B. W. 2009, SIAM review, 51, 455

Kopiczko, D. J., Blankevoort, T., & Asano, Y. M. 2024, in International Conference on Learning Representations

Krizhevsky, A. 2009, Learning multiple layers of features from tiny images, Tech. rep.

Krizhevsky, A., & Hinton, G. 2009, Master's thesis, Department of Computer Science, University of Toronto

Le, Q., Sarlós, T., Smola, A., et al. 2013in

Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. 1998, in Proceedings of the IEEE, 2278–2324

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. 1998, Proceedings of the IEEE, 86, 2278

LeCun, Y., Denker, J. S., & Solla, S. A. 1990, in Advances in neural information processing systems, 598–605

Lee, J., Kim, S., Yoon, J., Lee, H. B., Yang, E., & Hwang, S. J. 2018, arXiv preprint arXiv:1805.10896

Lester, B., Al-Rfou, R., & Constant, N. 2021, in Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics

Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H. P. 2016, arXiv preprint arXiv:1608.08710

Li, J., & Bioucas-Dias, J. M. 2008, in IGARSS 2008-2008 IEEE International Geoscience and Remote Sensing Symposium, Vol. 3, IEEE, III–250

Li, X. L., & Liang, P. 2021, in Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 4582–4597

Li, Y., Han, S., & Ji, S. 2024, arXiv preprint arXiv:2405.15179

Li, Y., & Ji, S. 2019, in The European Conference on Machine Learning (ECML)

Li, Y., & Ji, S. 2021a, in Joint European Conference on Machine Learning and Knowledge Discovery in Databases, Springer, 167–183

Li, Y., & Ji, S. 2021b, in 2021 International Joint Conference on Neural Networks (IJCNN), IEEE, 1–9

Li, Y., Ma, X., Sunderraman, R., Ji, S., & Kundu, S. 2023, Human Brain Mapping, 44, 4772

Lin, C.-H., Ma, W.-K., Li, W.-C., Chi, C.-Y., & Ambikapathi, A. 2015, IEEE Transactions on Geoscience and Remote Sensing, 53, 5530

Lin, M., Ji, R., Wang, Y., Zhang, Y., Zhang, B., Tian, Y., & Shao, L. 2020, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 1529–1538

Lin, S., Ji, R., Li, Y., Deng, C., & Li, X. 2019a, IEEE transactions on neural networks and learning systems, 31, 574

Lin, S., Ji, R., Yan, C., Zhang, B., Cao, L., Ye, Q., Huang, F., & Doermann, D. 2019b, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2790–2799

Liu, B., Wang, M., Foroosh, H., Tappen, M., & Pensky, M. 2015, in Proceedings of the IEEE conference on computer vision and pattern recognition, 806–814

Liu, H., Tam, D., Muqeeth, M., Mohta, J., Huang, T., Bansal, M., & Raffel, C. A. 2022, Advances in Neural Information Processing Systems, 35, 1950

Liu, S., & Deng, W. 2015, in 2015 3rd IAPR Asian conference on pattern recognition (ACPR), IEEE, 730–734

Liu, Y. et al. 2019, arXiv preprint arXiv:1907.11692

Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., & Zhang, C. 2017, in Proceedings of the IEEE International Conference on Computer Vision, 2736–2744

Louizos, C., Ullrich, K., & Welling, M. 2017, in Advances in Neural Information Processing Systems, 3288–3298

Louizos, C., Welling, M., & Kingma, D. P. 2018a, International Conference on Learning Representations (ICLR)

Louizos, C., Welling, M., & Kingma, D. P. 2018b, in International Conference on Learning Representations (ICLR)

Maddison, C. J., Mnih, A., & Teh, Y. W. 2016, arXiv preprint arXiv:1611.00712

Maddison, C. J., Mnih, A., & Teh, Y. W. 2017, in International Conference on Learning Representations (ICLR)

McMahan, B., Moore, E., Ramage, D., Hampson, S., & Arcas, B. A. y. 2017, in Proceedings of Machine Learning Research, Vol. 54, Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, ed. A. Singh & J. Zhu (PMLR), 1273–1282

Mitchell, T. J., & Beauchamp, J. J. 1988, Journal of the american statistical association, 83, 1023

Miyato, T., Maeda, S.-i., Ishii, S., & Koyama, M. 2018, IEEE transactions on pattern analysis and machine intelligence

Molchanov, D., Ashukha, A., & Vetrov, D. 2017, in Proceedings of the 34th International Conference on Machine Learning-Volume 70, JMLR. org, 2498–2507

Molchanov, P., Mallya, A., Tyree, S., Frosio, I., & Kautz, J. 2019, in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 11264–11272

Neklyudov, K., Molchanov, D., Ashukha, A., & Vetrov, D. 2017, in Advances in Neural Information Processing Systems (NIPS)

Novikova, J., Dušek, O., & Rieser, V. 2017, in Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue, ed. K. Jokinen, M. Stede, D. DeVault, & A. Louis, Saarbrücken, Germany, 201–206

Oseledets, I. V. 2010, SIAM Journal on Matrix Analysis and Applications, 31, 2130

Ouyang, L. et al. 2022, Advances in Neural Information Processing Systems, 35, 27730

Park, S., Lee, J., Mo, S., & Shin, J. 2020, arXiv preprint arXiv:2002.04809

Ponti, E. M., Sordoni, A., Bengio, Y., & Reddy, S. 2022, arXiv preprint arXiv:2202.13914

Pritchard, J. K., Stephens, M., & Donnelly, P. 2000, Genetics, 155, 945

Qin, Y. et al. 2021, arXiv preprint arXiv:2110.07867

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. 2019

Rajpurkar, P., Jia, R., & Liang, P. 2018, in Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ed. I. Gurevych & Y. Miyao, Melbourne, Australia, 784–789

Real, E., Aggarwal, A., Huang, Y., & Le, Q. V. 2019, in AAAI

Reisinger, J., Waters, A., Silverthorn, B., & Mooney, R. J. 2010, in International Conference on Machine Learning, Citeseer, 903–910

Renduchintala, A., Konuk, T., & Kuchaiev, O. 2023, arXiv preprint arXiv:2311.09578

Sarker, K., Yang, X., Li, Y., Belkasim, S., & Ji, S. 2020, arXiv preprint arXiv:2009.12027

Schwarz, G. 1978, The Annals of Statistics, 6, 461

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. 2016, in International Conference on Learning Representations

Sheng, Y. et al. 2023, arXiv preprint arXiv:2311.03285

Silver, D. et al. 2016, Nature, 529, 484

Simonyan, K., & Zisserman, A. 2014, arXiv preprint arXiv:1409.1556

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. 2013, in Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, ed. D. Yarowsky, T. Baldwin, A. Korhonen, K. Livescu, & S. Bethard, Seattle,

Washington, USA, 1631–1642

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. 2014, Journal of Machine Learning Research, 15, 1929

Stiles, J., & Jernigan, T. L. 2010, Neuropsychology Review, 20, 327

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., & Hashimoto, T. B. 2023, Stanford Alpaca: An Instruction-following LLaMA model, https://github.com/tatsu-lab/stanford_alpaca

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., & Jégou, H. 2021, in International Conference on Machine Learning, PMLR, 10347–10357

Touvron, H. et al. 2023, arXiv preprint arXiv:2307.09288

Tucker, G., Mnih, A., Maddison, C. J., Lawson, J., & Sohl-Dickstein, J. 2017, in Advances in Neural Information Processing Systems (NIPS)

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. 2017

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. 2018, in Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, ed. T. Linzen, G. Chrupała, & A. Alishahi, Brussels, Belgium, 353–355

Wang, Y., Xu, Z., Wang, X., Shen, C., Cheng, B., Shen, H., & Xia, H. 2021, in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 8741–8750

Wang, Y.-X., Ramanan, D., & Hebert, M. 2017, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR)

Warstadt, A., Singh, A., & Bowman, S. R. 2019, Transactions of the Association for Computational Linguistics, 7, 625

Wen, W., Wu, C., Wang, Y., Chen, Y., & Li, H. 2016a, in Advances in Neural Information Processing Systems (NIPS)

Wen, W., Wu, C., Wang, Y., Chen, Y., & Li, H. 2016b, in Advances in neural information processing systems

Williams, R. J. 1992, Machine Learning, 8, 229

Yaman, F., Li, Y., Han, S., Inoue, T., Mateo, E., & Inada, Y. 2023, in Optical Fiber Communication Conference, Optica Publishing Group, W1J–7

Ye, J., Lu, X., Lin, Z., & Wang, J. Z. 2018, arXiv preprint arXiv:1802.00124

Yin, M., & Zhou, M. 2019, in International Conference on Learning Representations (ICLR)

You, Z., Yan, K., Ye, J., Ma, M., & Wang, P. 2019, in Advances in Neural Information Processing Systems, 2133–2144

Zagoruyko, S., & Komodakis, N. 2016a, in The British Machine Vision Conference (BMVC)

Zagoruyko, S., & Komodakis, N. 2016b

Zhao, X. et al. 2023, arXiv preprint arXiv:2305.18703

Zheng, L. et al. 2024, Advances in Neural Information Processing Systems, 36

Zhou, G., Sohn, K., & Lee, H. 2012, in International Conference on Artificial Intelligence and Statistics (AIStats), 1453–1461

Zhu, M., Tang, Y., & Han, K. 2021, arXiv preprint arXiv:2104.08500

Zhuang, Z., Tan, M., Zhuang, B., Liu, J., Guo, Y., Wu, Q., Huang, J., & Zhu, J. 2018, in

Advances in Neural Information Processing Systems, 875–886

Zoph, B., & Le, Q. V. 2017, in International Conference on Learning Representations (ICLR)

Zoph, B., Vasudevan, V., Shlens, J., & Le, Q. V. 2018, in IEEE Conference on Computer Vision and Pattern Recognition (CVPR)