Deep Neural Network Compression: from Sufficient to Scarce Data

Chen Shangyu

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

19/08/2020 .... CHEN Shangyu ....

Date Chen Shangyu

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

19/08/2020

. . . .

Date Prof. Sinno Jialin Pan

Authorship Attribution Statement

This thesis contains material from 3 papers published in the following peer-reviewed papers accepted at conferences in which I am listed as an author.

Chapter 3 is published as **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "Deep Neural Network Quantization via Layer-Wise Optimization using Limited Training Data" in Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19).

The contributions of the co-authors are as follows:

- Wenya Wang and Sinno Jialin Pan fulfilled the idea and revise paper.

Chapter 4 is published as **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "Cooperative Pruning in Cross-Domain Deep Neural Network Compression", in Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-19).

The contributions of the co-authors are as follows:

- Wenya Wang and Sinno Jialin Pan fulfilled the idea and revise paper.

Chapter 5 is published as **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "MetaQuant: Learning to Quantize by Learning to Penetrate Non-differentiable Quantization" in Proceedings of 33rd Conference on Neural Information Processing Systems 2019 (NeurIPS-19).

The contributions of the co-authors are as follows:

- Wenya Wang and Sinno Jialin Pan fulfilled the idea and revise paper.

Chapter 6 is under review. The contributions of the co-authors are as follows:

- Sinno Jialin Pan discussed fulfilled the idea and revise paper.

19/08/2020	CHEN Shangyu
Date	Chen Shangyu

Acknowledgements

I wish to express my greatest gratitude to my supervisor: Sinno Jialin Pan, who has been tutoring and guiding me during the past four years. Back then, I came in this university without any sense of research. It is through Prof. Pan that I have learnt the meaning of research, taste of good reseach, and how to be an independent researcher. The things that my supervisor delivered me is more than sophistical in-domain knowlegde and writting skills, but the idea behind these knowledge itself, i.e. how to form my line of thinking. In other words, meta-knowledge or “Dao” in Chinese.

Last but not least, thanks to the support of my parents, I am able to reach this land and pursue my PhD degree.

Cogito, ergo sum. (I think therefore I am.)

—Rene Descartes

He who pursues the truth ends at the same.

—Tsugikuni Yoriichi

To me 4 years ago and in future

Abstract

The success of overparameterized deep neural networks (DNNs) poses a great challenge to deploy computationally expensive models on edge devices. Numerous model compression (pruning, quantization) methods have been proposed to overcome this challenge: Pruning eliminates unimportant parameters, while quantization converts full-precision parameters into integers. Both shrink model size and accelerate inference. However, existing methods reply on a large amount of training data. In real cases such as medical domain, it is consuming to collect training data, due to extensive human effort or data privacy. To tackle the problem of model compression in scarce data scenario, in this thesis, I have summarized my previous works on model compression, from using sufficient data to scarce data.

My early phase's work focused on model compression in a layer-wise manner: The loss of layer-wise compression is studied and corresponding compression solutions are proposed for alleviation. The layer-wise process enables fewer data dependency in quantization. This work is summarized in Chapter 3.

Following model quantization using scarce data, I proposed to prune model on a cross-domain setting in Chapter 4. It aims at improving compression performance on tasks with limited data, with the assistance of rich-resource tasks. Specially, a dynamic and cooperative pruning strategy is utilized to prune both source and target network simultaneously.

In Chapter 5, I try to solve the non-differentiable problem in training-based compression, where the pruning or quantization operations prevent gradient backward propagation from loss to trainable parameters. I proposed to use a meta neural network to penetrate the compression operation. The network receives input as trainable parameters and accessible gradients, and outputs gradients for parameters update. By incorporating the meta network into compression training, empirical experiments demonstrate a faster learning rate and better performance.

Although works on Chapter 3 and 4 alleviate model compression tasks on scarce data. They either required a pre-trained model or addition cost in compressing another model. In Chapter 6, an arbitrary scarce-data task is able to be compressed, with the inspiration from Chapter 5: I proposed to learn meta-knowledge from multiple model compression tasks using a meta-learning framework. The knowledge is embedded in an initialization for all tasks and a meta neural network which provides gradient during training. When a novel task arrives, it starts from the initialization and is trained by the guidance of meta neural network to reach compressed version in very few steps.

Contents

Acknowledgements ix

Abstract xiii

List of Figures xix

List of Tables xxiii

Symbols and Acronyms xxv

1 Introduction 1
- 1.1 Deep Neural Networks Deployment . . . 1
- 1.2 Attaining Compressed Model . . . 3
- 1.3 Problem Statement . . . 4
  - 1.3.1 Sufficient-data Model Compression . . . 4
  - 1.3.2 Scarce-data Model Compression . . . 5
- 1.4 Organization . . . 5

2 Related Work 7
- 2.1 Pruning . . . 9
  - 2.1.1 Unstructured Pruning . . . 9
    - 2.1.1.1 Magnitude-Based . . . 9
    - 2.1.1.2 Gradient-Based . . . 10
    - 2.1.1.3 Hessian-Based . . . 10
    - 2.1.1.4 Lookahead . . . 12
    - 2.1.1.5 Lottery Ticket Hypothesis . . . 12
  - 2.1.2 Structured Pruning . . . 13
    - 2.1.2.1 Filter/Channel Pruning . . . 14
    - 2.1.2.2 Miscellaneous . . . 17
- 2.2 Quantization . . . 19
  - 2.2.1 Weight Quantization . . . 19
  - 2.2.2 Weight & Activation & Gradient & Error Quantization . . . 24
  - 2.2.3 Differentiable Quantization . . . 30
  - 2.2.4 Post-Training / Data-Free Quantization . . . 30

2.2.5 Miscellaneous . . . 32
2.2.6 Comparison of Various Methods . . . 34
2.3 Distillation . . . 35
2.4 Architecture Modification . . . 37

3 Layer-Wise Model Quantization **41**
3.1 Introduction . . . 41
3.2 Problem Statement . . . 43
3.3 Layer-Wise Error . . . 43
3.4 Hessian Approximation . . . 45
3.5 Layer-Wise Quantization . . . 47
3.5.1 Alternative Direction Methods of Multipliers (ADMM) . . . 48
3.5.2 Quantization with ADMM . . . 49
3.5.2.1 Proximal Step . . . 50
3.5.2.2 Projection Step . . . 50
3.5.2.3 Dual Update Step . . . 51
3.5.3 Remaining Non-quantized Layers Update . . . 52
3.5.4 L-DNQ in Practice . . . 52
3.6 Theoretical Analysis . . . 53
3.7 Experiment . . . 54
3.7.1 Overall Experimental Results and Analysis . . . 56
3.7.2 Analysis of L-DNQ . . . 59
3.7.2.1 Bits' Effect Towards Performance . . . 59
3.7.2.2 Layer Output Error V.S. Performance . . . 59
3.7.2.3 Effectiveness of Weights Update . . . 60
3.7.2.4 Order of Quantization Process . . . 60
3.8 Conclusion . . . 60

4 Cooperative Pruning **61**
4.1 Introduction . . . 61
4.2 Cooperative Pruning . . . 63
4.2.1 Mask Generation . . . 64
4.2.2 Adaptive Domain Adaptation via Transfer Factor α . . . 65
4.2.3 Training-based Pruning . . . 67
4.2.4 Algorithm and Implementation Details . . . 68
4.3 Experiment . . . 68
4.3.1 Comparison Experiment . . . 69
4.3.2 Effect of Transfer Factor α . . . 71
4.3.2.1 Variation of Co-Prune . . . 72
4.3.2.2 Effect of Sensitivity β . . . 72
4.3.2.3 Effect of Initial α_0 . . . 73
4.3.2.4 Effect of Minimum $\alpha_{\min}$ . . . 73
4.3.3 Performance Under Various CRs . . . 74

4.3.4 Complexity Analysis . . . 74
4.3.5 Negative Transfer . . . 74
4.4 Conclusion . . . 75

5 Meta Compression 77
5.1 Introduction . . . 77
5.2 Problem Statement and Preliminary . . . 79
5.3 Meta-Compress . . . 81
5.3.1 Generation of Meta Gradient . . . 81
5.3.2 Training of Meta Compressor . . . 83
5.3.3 Incorporation of Side Information in Meta-Prune . . . 84
5.3.4 Design of Meta Compressor . . . 85
5.3.5 Algorithm and Implementation Details . . . 86
5.4 Experiment . . . 87
5.4.1 Experimental Results and Analysis . . . 88
5.4.1.1 Meta-Quant . . . 88
5.4.1.2 Meta-Prune . . . 89
5.4.2 Empirical Convergence Analysis . . . 89
5.4.2.1 Meta-Quant . . . 89
5.4.2.2 Meta-Prune . . . 92
5.4.3 Effect of Side Information in Meta-Prune . . . 92
5.4.4 Performance Comparison with Non-STE Training-based Quantization . . . 93
5.4.5 Performance Comparison with Non-training-based Pruning . 94
5.4.6 Meta-Compress Training Analysis . . . 94
5.4.7 Analysis of Hyperparameters and Architecture in Meta Compressor . . . 95
5.4.8 Looking Deeper into the Meta-Compress . . . 95
5.5 Conclusion . . . 96

6 Transferable Compressor 101
6.1 Problem Statement and Preliminary . . . 103
6.1.1 Optimization-based Meta Learning . . . 103
6.1.2 Compression Training . . . 104
6.2 Transferable Compressor . . . 105
6.2.1 Inner loop compression training . . . 106
6.2.2 Outer loop meta knowledge update . . . 106
6.3 TransComp in Practice . . . 108
6.3.1 Regularization for Pruning . . . 108
6.3.2 Design of $\mathcal{M}_\phi$ . . . 108
6.4 Experiments . . . 109
6.4.1 Overall Experiments . . . 112
6.4.1.1 CIFAR100 . . . 112
6.4.1.2 ImageNet . . . 113

6.4.1.3 Omniglot 113
6.4.1.4 MiniImageNet 114
6.4.1.5 CIFAR100 $\rightarrow$ STL10 115
6.4.2 Inner Loop Training Analysis 116
6.4.3 Data Scarcity 117
6.4.4 Convergence Analysis 117
6.4.5 Effect of Architecture in $\mathcal{M}_\phi$ 121
6.4.6 Interpretation of $\mathcal{M}_\phi$ 121
6.5 Conclusion 122

7 Conclusion & Future Work **123**
7.1 Conclusion 123
7.2 Future Work 124
7.2.1 Model Compression Co-design of Algorithm and Hardware . 124
7.2.2 Data-Free Compression 125

List of Author's Awards, Patents, and Publications **127**

Bibliography **129**

List of Figures

1.1 An illustration of AlexNet. The first version of deep neural network. 2
1.2 Smartphones . . . 2
1.3 Cameras . . . 2
1.4 An illustration of unsturctured pruning. Figure is cited from [1] . . 3
1.5 An illustration of structure pruning. Figure is cited from [2] 3

2.1 Four main types of Structured Pruning in convolution layers. [2] . . 13
2.2 Pruning channels: pruning channels in layer l results in pruning filters in previous layer. Figure is cited from [3] . . . 15
2.3 Overflow of FBS. Figure is cited from [4] . . . 16
2.4 Weight is pruned with remaining diagnoal block. Figure is cited from [5] . . . 18
2.5 This figure contrasts the block structure in our XNOR-Network (right) with a typical CNN (left). Figure is cited from [6] . . . 21
2.6 Four operators $Q_W(\cdot)$, $Q_A(\cdot)$, $Q_G(\cdot)$, $Q_E(\cdot)$ added in WAGE computation dataflow to reduce precision, bit width of signed integers are below or on the right of arrows, activations are included in Multiply-Accumulate (MAC) Cycles for concision. Figure is cited from [7] . . . 27
2.7 Procedure of quantization in [7]. Figure is cited from [7] . . . 27
2.8 Procedure of [8]. Figure is cited from [8] . . . 29
2.9 Procedure of [9]. Figure is cited from [9] . . . 33
2.10 Procedure of quantization distillation in [10]. Figure is cited from [10] 36
2.11 Convolution in matrix multiplication: Input image is extracted into multiple patches, which share the same dimension of the kernels. The patches are arranged into a vectorized matrix, while the kernels are at the same time reshaped into vectorized matrix. Convolution is formulated as the matrix multiplication. . . . 37
2.12 Grouped convolution in matrix multiplication: Left part represents input feature, which is processed by divided into g groups. Each slides of input feature is multiplied with the corresponding diagonal part of parameters in right part, i.e. Only the diagonal blocks of parameters are remained. Grouped convolution reduce number of parameters and computation complexity to $1/g$. . . . 38
2.13 output channels from different grouped convolution are shuffled. . . 38

2.14 Illustrating the interleaved group convolution, with $L = 2$ primary partitions and $M = 3$ secondary partitions. The convolution for each primary partition in primary group convolution is spatial. The convolution for each secondary partition in secondary group convolution is point-wise (1×1). 39

3.1 Illustration of shape of Hessian. For feed-forward neural networks, unit z_1 gets its activation via forward propagation: $\mathbf{z} = \mathbf{W}^\top \mathbf{y}$, where $\mathbf{W} \in \mathbb{R}^{4\times3}$, $\mathbf{y} = [y_1, y_2, y_3, y_4]^\top \in \mathbb{R}^{4\times1}$, and $\mathbf{z} = [z_1, z_2, z_3]^\top \in \mathbb{R}^{3\times1}$. Then the Hessian matrix of z_1 w.r.t. all parameters is denoted by $\mathbf{H}^{[z_1]}$. As illustrated in the figure, $\mathbf{H}^{[z_1]}$'s elements are zero except for those corresponding to $\mathbf{W}_{*1}$ (the 1st column of $\mathbf{W}$), which is denoted by $\mathbf{H}_{11}$. $\mathbf{H}^{[z_2]}$ and $\mathbf{H}^{[z_3]}$ are similar. More importantly, $\mathbf{H}^{-1} = \text{diag}(\mathbf{H}_{11}^{-1}, \mathbf{H}_{22}^{-1}, \mathbf{H}_{33}^{-1})$, and $\mathbf{H}_{11} = \mathbf{H}_{22} = \mathbf{H}_{33}$. As a result, one only needs to compute $\mathbf{H}_{11}^{-1}$ to obtain $\mathbf{H}^{-1}$ which significantly reduces computational complexity. 47
3.2 Weights Update: After layer $l-1$ is quantized from $\bar{\mathbf{\Theta}}_{l-1}$ to $\hat{\mathbf{\Theta}}_{l-1}$. Input $\mathbf{Y}^{l-1}$ and $\hat{\mathbf{Y}}^{l-1}$ are fed into two networks to generate $f(\mathbf{Y}^{l-1}; \bar{\mathbf{\Theta}}_{[l,\dots,L]})$ and $f(\hat{\mathbf{Y}}^{l-1}; \bar{\mathbf{\Theta}}_{[l,\dots,L]})$, respectively. Squared difference is backpropated to update weights in higher layers: $\bar{\mathbf{\Theta}}_{l,\dots,L}^{new}$. 51
3.3 Fig.(a)/(b): Performance among L-DNQ, ExNN, TTQ, INQ, VQ, DQ using ResNet20/18 in CIFAR10/ImageNet with increasing instances. X-axis presents portion of training data used, Y-axis represents performance improvement after quantization (Higher the better). Fig.(c): Performance under different layer output error. X-axis presents final layer output error, Y-axis represents testing error after quantization (the lower the better). 58

4.1 Sketch architecture for Co-Prune: source / target task is trained and pruned cooperatively. Red dash lines indicate where the contribution to masks comes from. Blue dash lines point out where the masks are imposed to. Pruning mask of source task is determined by its own parameters. By contrast, both parameters from source and target model contribute to pruning mask for target. 63
4.2 Adjust α during cooperative pruning. Given updated parameters from the source model $\mathbf{W}_s$ and the target model $\mathbf{W}_t$, transfer factor α in fusing function f is imposed to generated a combined $\tilde{\mathbf{W}}$ for target mask 66
4.3 (a) Performance of Co-Prune under various β with fixed $\alpha_0 = 0.7$, $\alpha_{\min} = 0.3$; (b) Performance of Co-Prune under various α_0 with fixed $\beta = 3$, $\alpha_{\min} = 0.3$; (c) Performance of Co-Prune under various $\alpha_{\min}$ with fixed $\alpha_0 = 0.7$, $\beta = 3$. Y-axis represents the best performance under corresponding setting, X axis represents β, α_0, $\alpha_{\min}$, respectively 71
4.4 Performance improvement as CR changes in I$\rightarrow$ P 74

5.1 The overflow of Meta-Compress: During backward propagation, gradients are represented as blue line. Dash blue line means this propagation is non-differentiable and requires special handling. A shared meta network $\mathcal{M}$ is constructed which takes $g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}$ as input and its output will replace the gradient of $\tilde{\mathbf{W}}$ as $g_{\tilde{\mathbf{W}}}$. 82
5.2 Incorporation of meta compressor into compression training. Red dash box is composed of calibration, gradient refinement and multiplication of learning rate α. Output of meta compressor is involved in $\mathbf{W}$'s update and contributes to final loss, constructing a differential path from loss to ϕ-parameterized meta compressor. 83
5.3 Convergence speed of Meta-Quant V.S STE using SGD/Adam, dorefa/BWN in ResNet20, CIFAR10, dorefa. 90
5.4 Convergence speed of Meta-Prune V.S DNS using various model and datasets. 90
5.5 Meta-Quant V.S. STE in ResNet56-CIFAR100 91
5.6 Meta-Prune V.S. DNS in ResNet20-CIFAR10 91
5.7 Training convergence of DNS, Meta-Prune and Meta-Prune with side information. 93
5.8 Loss and gradient amplification during training. 96

6.1 Overflow of TransComp: Left: For a coming task, in inner loop training, it starts from a learnable initialization $\mathbf{\Theta}$ and is trained to a task-specific compressed model ($\hat{\mathbf{\Theta}}_{0,1,2}$) via guidance of $\mathcal{M}_\phi$. Source tasks' training update $\mathbf{\Theta}$ and $\mathcal{M}_\phi$ in outer loop update. When novel task arrives, it is adapted from $\mathbf{\Theta}$ to reach compressed model by $\mathcal{M}_\phi$. Right: Blue lines represent the flow of gradient. The non-differentiable compression operation $\mathcal{C}(\cdot)$ between $\mathbf{\Theta}(l)$ and $\hat{\mathbf{\Theta}}(l)$ in layer l leads to gradient obstruction (blue dash line). To overcome it and explore better gradient, layer-wise $\mathcal{M}_\phi^l$ receive gradient of $\hat{\mathbf{\Theta}}(l)$ (blue solid line) and parameters of $\mathbf{\Theta}(l)$ (green line) to provide gradient for $\mathbf{\Theta}(l)$. 105
6.2 Adaptation Procedure of Selected Novel Tasks in CIFAR100. 116
6.3 Relative Performance under different data scarcity of novel tasks. . 117
6.4 Source task loss and inner loop compression training performance in 10 selected novel tasks in 2-bit quantization on CIFAR100: $c = 5, k_i^{tr} = 100, k_i^{tst} = 100, k_*^{tr} = 100, k_*^{tst} = 100$. 118
6.5 Source task loss and loop compression training performance in 10 selected novel tasks in $\delta = 30$ pruning on CIFAR100 with $c = 5, k_i^{tr} = 100, k_i^{tst} = 100, k_*^{tr} = 100, k_*^{tst} = 100$. 118
6.6 Source task test loss and loop compression training performance in 10 selected novel tasks in 2-bit quantization on ImageNet with $c = 10, k_i^{tr} = 200, k_i^{tst} = 50, k_*^{tr} = 200, k_*^{tst} = 50$. 120
6.7 Omniglot-5-1 120
6.8 Omniglot-20-1 120
6.9 Omniglot-20-5 120

6.10 Convergence of various methods in 2-bit quantization omniglot training. 120
6.11 miniimagenet-5-1 120
6.12 miniimagenet-5-5 120
6.13 Convergence of various methods in 2-bit quantization miniimagenet training. 120
6.14 Amplification of $\mathcal{M}_\phi$ during training process. Left Y axis: magnitude of amplification after a linear $\mathcal{M}_\phi$; Right Y axis: training loss. 122

7.1 Inference speed comparison between unstructure pruning (CSR) and normal matrix multiplication(GEMM) 125

List of Tables

2.1 Comparison among five main types of model compression methods. 8
2.2 Comparison of different quantization method, "PT" represents **P**retrained **M**odels . . . 34
2.3 Performance Comparison of Various Quantization Methods in AlexNet 34
2.4 Performance Comparison of Various Quantization Methods in VGG16 35
2.5 Performance Comparison of Various Quantization Methods in ResNet18 35
2.6 Performance Comparison of Various Quantization Methods in ResNet50 35
2.7 Performance Comparison of Various Quantization Methods in GoogLenet 35

3.1 Comparison on CIFAR-10. All methods use 1% (500 images) of training instances. * indicates improvement. ** represents **F**ull **P**recision (pre-trained model) Accuracy. *** is a VGG-like model adopted by DistilQuant. . . . 56
3.2 Comparison on ImageNet. All methods use 1% (12,800 images) training instances. AlexNet[1]: in TTQ, the weights of the first and final layer remain full precision. L-DNQ, ExNN, DQ, VQ, DistilQuant are under the same setting. . . . 57
3.3 Overall experimental results of L-DNQ in various models using 1% of ImageNet dataset. Column 2-5 represent the quantization improvement (Top1/Top5) under different bits quantization. The last column represents full precision accuracy. . . . 57
3.4 Comparison between L-DNQ and L-DNQ^{-}. . . . 58

4.1 Overall results of CIFAR9-STL9 using CIFAR-Net . . . 70
4.2 Overall results of ImageNet→PASCAL, ImageNet→Caltech256, ImageNet→Bing using ImageNet pre-trained ResNet18. CR is 4% for each layer. . . . 72
4.3 Co-Prune's variation using different α. $\alpha = 0$ reduces to DNS, $\alpha = 1$ means that mask generation replies on source model . . . 73
4.4 Experiments of MNIST as source dataset and STL10 as target dataset. 75

5.1 Experimental result of Meta-Quant and STE using dorefa, BWN on CIFAR10 . . . 97
5.2 Experimental result of MetaQuant and STE using dorefa, BWN on CIFAR100 . . . 98
5.3 Experimental result of MetaQuant and STE using dorefa, BWN on ImageNet. . . . 98

5.4 Experimental result of Meta-Prune and DNS. 99
5.5 Comparison experiments for Meta-Prune with variant of side information. 99
5.6 Experimental result of Meta-Quant V.S ProxQuant, LAB, ELQ, TTQ. 99
5.7 Experimental result of Meta-Prune V.S LWC, L-OBS 100
5.8 Experimental result of MetaQuant-FC under different hidden dimension, activation 100

6.1 Relative Performance of Compression (2-bit Quantition, $\delta = 30$ Pruning) on CIFAR100. 112
6.2 Relative Performance of Compression (2-bit Quantization, $\delta = 30$ Pruning) on ImageNet. 113
6.3 Performance of Compression on Omniglot. 114
6.4 Performance of Compression on MiniImageNet. 114
6.5 Performance of compression on STL10 with $c = 5$. 115
6.6 Performance of 2-bit quantization on STL10 with $c = 10$. 116
6.7 Performance of 2-bit quantization in Omniglot 5-1 using `MultiFC`. . 121

Symbols and Acronyms

Symbols

$\mathbf{D}$, $\mathbf{D}^{tr}$, $\mathbf{D}^{te}$	Dataset, training dataset, test dataset
$\mathbf{W}/\mathbf{\Theta}$	Full-precision parameters of a model
$\tilde{\mathbf{W}}$	Pre-compressed and full-precision parameters of a model
$\hat{\mathbf{W}}/\hat{\mathbf{\Theta}}$	Compressed parameters of a model
Ω	Quantization constraint
$\mathbf{H}$	Hessian matrix
$\mathcal{C}(\,\cdot\,)$	Compression operation (quantize, prune)

Chapter 1

Introduction

In the past decade, deep learning has shown significant performance in a wide-range of real world applications [11–13]. Its contribution lies in the usage of overparameterized neural networks, which possesses millions of parameters and requires huge scale of training data to achieve satisfactory performance.

However, such performance comes with huge cost in computational resource and storage space: Consider a very deep model which is fully-trained and deployed, to use it for making predictions, most of the computations involve multiplications of a real-valued weight by a real-valued activation in forward propagation. These multiplications are expensive as they are all float-point to float-point multiplication operations. Besides, the number of parameters is huge: classical deep learning model AlexNet [11] possesses 62,378,344 parameters, occupying over 244Mb in storing. Although modern deep neural network such as ResNet-like network [13, 14] shrinks model size sharply, its feature reuse adds more memory consumption in inference. The situation requires most deep learning applications deployed on resource-abundant platform such as servers.

1.1 Deep Neural Networks Deployment

Recent practical and commercial scenarios are demanding deployment on edge devices such as cell phone, surveillance cameras. Devices at the edge typically have

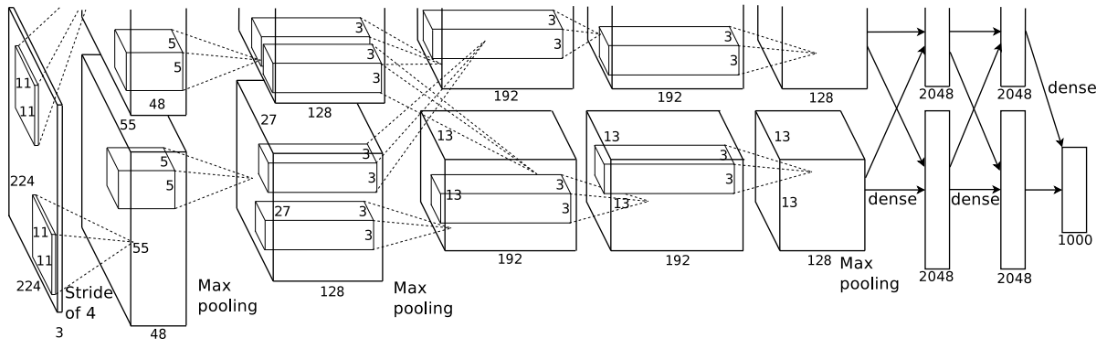


FIGURE 1.1: An illustration of AlexNet. The first version of deep neural network.

lower compute capabilities and are constrained in memory and power consumption. It is also necessary to reduce the amount of communication to the cloud for transferring models to the device to save on power and reduce network connectivity requirements. Therefore, there is a pressing need for techniques to compress models for reduced model size, faster inference and lower power consumption.

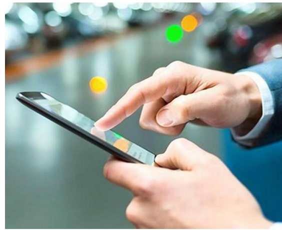


FIGURE 1.2: Smartphones

FIGURE 1.3: Cameras

There are extensive researches on this topic (**Model Compression**) with several approaches being considered:

Pruning: Pruning sparsifies parameters by setting unimportant ones to 0. It can further be divided into **Unstructured Pruning** and **Structured Pruning**. **Unstructured Pruning** prunes weights elements by elements, while **Structured Pruning** prunes entire structure / module of DNN as a whole. For example, it sparsifies the entire filter in convolution kernels, or entire row / column in fully-connected weights, which can be regarded as eliminating the whole input neuron or output neuron. By eliminating certain portion of parameters in deep neural models, it requires less storage space and is able to skip computation with corresponding parameters setting as 0.

Quantization: Quantization discretizes weights into limited values, such as $\{\pm 2, \pm 1, 0\}$. According to different quantization level, quantization can be further categorized

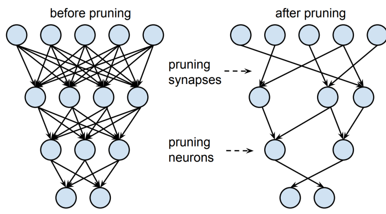


FIGURE 1.4: An illustration of unsturctured pruning. Figure is cited from [1]

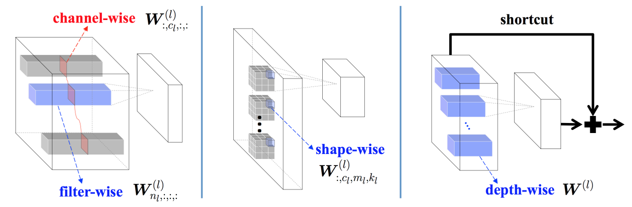


FIGURE 1.5: An illustration of structure pruning. Figure is cited from [2]

as **binarization** ($\{\pm1\}$) [6, 15], **ternarization** ($\{\pm1, 0\}$) [16, 17], **quantization** ($\{..., \pm2, \pm1, 0\}$) ([18]) and their corresponding variants, such as $\{\alpha, 0, \beta\}$ as ternarization. Quanzation can be regarded as converting full-precision weights and clustering them into ordered set. The resulting quantized weights can be stored in integer matrix, and turn float point - float point multiplication into float point - integer multiplication, which significantly reduces calculations.

Distillation: Distillation helps in converting knowledge in very deep neural network into a relatively shadow network. e.g. From ResNet-34 into ResNet-18. This method trains a new network given a pretrained network: Specifically, it combines the knowledge as finally layer output from pretrained and groundtrue labels to accelerate training and improve performance. Besides, the shadow network itself brings compression from original deep neural network.

Miscellaneous: Besides mathematical operations, some methods aim at optimize the procedure of convolution for acceleration, such as proposing new types of convolution, or arranging to compact computation based on prior knowledge.

1.2 Attaining Compressed Model

Model compression sheds light to attaining efficient deep neural networks for deployment in edge devices. However, attaining a compressed model requires as much effort as training a full-precision model, especially in the demanding for large scale of training data: Prevailing model compression methods heavily rely on supervised training, which follows a paradigm of fitting model parameters by iterative inference and backpropgation.

Practically, sufficient data is difficult to collect in real scenarios. Due to reasons such as data privacy, expensive human effort for annotation, etc. Reality poses great challenges in scarce data compression.

Similar problem have arisen in traditional machine learning and deep learning community: methods have been proposed to tackle scarce data situations, and progressed in serval fields: Transfer learning [19] proposed to extract knowledge from source tasks and applied it to improve the training of target tasks, especially tasks with insufficient training data; Meta learning [20] regarded each tasks with few shot data as a training instance and learned common knowledge. However, trials have not been conducted in model compression: **How can we generate a compressed model for task with scarce data?**

1.3 Problem Statement

The contribution of this thesis can be roughly categorized into **sufficient-data** and **scarce-data** model compression.

1.3.1 Sufficient-data Model Compression

Given sufficient training set $\mathbf{D}^{tr}$, we aim at producing a compressed model $\mathcal{N}(\hat{\mathbf{W}})$, which achieves satisfying results on test set $\mathbf{D}^{te}$.

The era of deep learning replies on huge amount of training data, though the number of parameters in deep neural network is still larger than the volume of dataset (ResNet18 occupies 33,161,024 parameters V.S. 1,280,000 training instances in ImageNet). Compared with traditional machine learning setting, deep neural network requires training set with scale of 10 thousand to reach convergence. Normally, size of $\mathbf{D}^{tr}$ scales up to $10,000 \sim 1,000,000$, such as CIFAR10 ($50,000$ training instances) and ImageNet ($1,280,000$ training instances).

Similar challenges happen in deep neural network model compression: to attain a compressed model, common methods require a large training set, which is sufficient to train a full-precision counterpart. Sufficient-data model compression shares the

same setting as deep neural model learning, with a different goal to achieve a compressed model.

1.3.2 Scarce-data Model Compression

Given a task of interest with scarce training set $\mathbf{D}^{tr}$ (normally 1% of full training set mentioned in Sec.1.3.1), we aim at generating a compressed model $\mathcal{N}(\hat{\mathbf{W}})$, which is still cable of make prediction on $\mathbf{D}^{te}$ with satisfying performance.

The lack of training instance leads to under-fitting and non-convergence for deep neural network. The reasons for the data insufficiency comes from two aspects: 1) Training data is trivial, such as simple text or data with limited dimensions. In these cases, deep neural network is overused and traditional machine learning model such as decision tree is enough. This situation is out of the scope of our discussion. 2) Training data is hard to attain (medical images) or unaccessible (commercial data, user logs). These data is difficult to analyze and thus require deep neural network. However, their number is limited, commonly only $10 \sim 100$ per classes.

Currently, few-shot learning has emerged as a new and practical topic in deep learning community. Few-shot learning requires a deep neural network to learn useful features by only 1 or 5 instances per classes. When instances from novel and unseen classes arrive, few-shot learned model can adapt to the new distribution and make satisfactory prediction.

These practical problems pose new challenges to both deep learning and model compression. I have especially study model compression with scarce data, where data is complex enough for usage of deep neural network while data is not sufficient to train a model, not to mention its compressed counterpart.

1.4 Organization

In this thesis, I organize my previous work on deep neural network compression from using sufficient data to scarce data.

In Chapter 2, I have reviewed the important progresses in model compression.

Traditional compression methods rely on heavy re-training process in pruning and require large amount of training data in quantization. To tackle these limitations, in Chapter 3, we proposed to generate quantized model using scarce data under a layer-wise framework: we performed weights quantization aiming at minimizing the change of layer-wise output after quantization. We introduced a cascade distillation training method, which uses distillation to train the remaining unquantized parameters after one-layer's quantization. Empirical experiments show that only a small portion of data is sufficient to finish quantization. The compression in each layer brings in layer-wise error, which is propagated and accumulated to final output. We proved that the final error is upper bounded by a constant. It builds the foundation that layer-wise compression will not lead to error explosion.

Aside to pruning under limited resource scenario, in Chapter 4, we proposed to co-operatively train the target compressed model and a source compression tasks. By utilizing knowledge transferred from source domain, target task is able to improve its compression performance even under limited training data.

Besides of the limitation in training process and data hunger. Most training-based compression methods face the problem of deriving gradient due to its non-differentiable compression operation $\mathcal{C}(\cdot)$. To enable compression training, $\frac{\partial \mathcal{C}(\mathbf{W})}{\partial \mathbf{W}}$ is assigned as 1 if $||\mathbf{W}|| \leq 1$ else 0. To tackle this problem and explore better gradient during training-based compression, in Chapter 5, we propose to learn $\frac{\partial \mathcal{C}(\mathbf{W})}{\partial \mathbf{W}}$ by another neural network, denoted as meta compressor. Experiments show that our proposed method is able to achieve faster convergence rate and better test performance.

One major limitation in model compression methods is that the task-specific training data is assumed to be sufficient to train a precise compressed version. However, in some real-world scenarios, high-quality labeled data is often in short supply, such as in the medical domain. To tackle the problem of conducting model compression in novel tasks with scarce data, in Chapter 6, we proposed to meta-learn the training process of compression from multiple model compression tasks (a.k.a. source tasks), and then transfer the meta knowledge to guide generating a precise compressed network to the novel task with scarce training data. Empirical experiments demonstrate that novel task's training on the transferred knowledge is able to achieve better performance in smaller number of steps.

Chapter 2

Related Work

In this survey of related work, model compression is roughly categorized into four main aspects, with their corresponding sub categorizations:

- Pruning
  - Unstructure Pruning
  - Structure Pruning
- Quantization
  - Training-based Quantization: Weight/Activation/Gradient/Error Quantization, Differentiable Quantization
  - Post-training Quantization
- Distillation
- Architecture Modification

Among these methods, pruning mainly focuses on finding important connections in deep neural network and aims at generating a sparse model. Unstructure pruning produces pruned model without pattern. It can compress the model to extreme size while maintaining acceptable performance. However, the unstructure sparse model can not be accelerated under current computing framework. On the contrary, structure pruning shrinks models to a compact size, by removing an entire structure (neuron, channel, filter) from the model. Normally, performance shows significant

Method	Compression	Acceleration
Unstructure Pruning	Low	Hard
Structure Pruning	High	Easy
Quantization	Medium	Medium
Distillation	Medium	Easy
Architecture Modification	Medium	Easy

Table 2.1: Comparison among five main types of model compression methods.

drop after compression rate reaches to 50%. However, structure pruned model can be easily accelerated, without any modification of current framework.

Quantization converts all full-precision parameters to a constraint set, forming the integer counterpart. The compression rate depends on the size of constraint set. Although integer computation is highly efficient in current computer architecture, special kernels are required to support the actual acceleration on quantization. Currently, 8-bit quantization kernel is widely available in GPU/CPU, however, more aggressive quantization support is still under progress.

Distillation trains a compact and small neural network by the assistance of a large model. Similar to structure pruning, distilled student model can be easily accelerated. However, human efforts are needed to determine the architecture of student model.

Architecture modification changes the core modules of neural network, to reduce the necessary computation while maintaining a satisfactory performance. Similar to structure pruning and distillation, the acceleration of modified neural networks are fully supported in current framework. However, generation of efficient core modules requires much endeavor.

I analyze the difference of five main categorization of model compression methods in Table 2.1. “Compression” represents the compression rate, i.e. compressed size divided by original size, the lower the better. “Acceleration” stands for whether the method can be easily supported to achieve actual acceleration in current computer framework.

2.1 Pruning

2.1.1 Unstructured Pruning

Unstructured pruning basically aims at finding the important parameters / neurons. Various methods have been proposed to measure the importance of parameters, pruning the unimportant ones while alternating the value of the remaining parameters. This selection and finetune procedure can be conducted by given a pre-trained full-precision model (Post-Pruning), or on the fly during the training of a pruned model (Training-based Pruning, Pruning Training).

2.1.1.1 Magnitude-Based

The first impression on measuring parameters' importance lies in the absolute values. It is natural to occur that the smaller the parameters, the less effect it will make to the final prediction. Formally, measuring by absolute value turns to be minimizing the Frobenius distortion after pruning:

$$\min_{\mathbf{M}:||\mathbf{M}||_0=s} ||\mathbf{W} - \mathbf{M} \cdot \mathbf{W}||_F \tag{2.1}$$

where $\mathbf{M}$ represents the parameter selection matrix, with elements being 1 for unpruned. The number of its non-zero elements need to be smaller than s, a predefined sparsity constraint.

Based on this simple but effective observation, LWC [1] pioneered element pruning in deep neural network. It proposed to prune elements by its absolute value based on a pre-trained full-precision model: it deleted those weights who approach to zero and remained the rest, which held the assumption that larger weights contribute more to the performance of neural network. After pruning, LWC retrained the un-pruned parameters.

[21] incorporated LWC into network training to prune parameters in an incremental way. Specifically, during the training of deep neural network, it occasionally pruned weight according to their absolute value. Then it continued to train with pruned ones fixed as zero. Those pruned weights can be restored in latter retrain.

2.1.1.2 Gradient-Based

[22] proposed to measure importance of connection, independently of the corresponding parameters. It introduced a connection matrix $\mathbf{M}$ imposed on parameters $\mathbf{W}$:

$$\min_{\mathbf{M},\mathbf{W}} \quad \ell(\mathbf{M} \odot \mathbf{W}; \mathbf{D}^{tr}) \tag{2.2}$$

$$\text{s.t.} \quad \mathbf{M} \in \{0,1\}^m, |\mathbf{M}|_0 \leq \kappa \tag{2.3}$$

It approximated the variation of ℓ by gradient of ℓ w.r.t pruned connection j, leading to $\mathbf{M}_j = 0$: $\Delta\ell_j(\mathbf{W}, \mathbf{D}^{tr}) \approx g_j(\mathbf{W}, \mathbf{D}^{tr})$. For each layer with m parameters, the importance of j-th connection is normalized as:

$$s_j = \frac{|g_j(\mathbf{W}, \mathbf{D}^{tr})|}{\sum_{k=1}^{m} |g_k(\mathbf{W}, \mathbf{D}^{tr})|} \tag{2.4}$$

Besides, it proposed to measure the connection importance on a random while exquisitely designed initialization , instead of the pre-trained full-precision model used widely. After using only one mini-batch data from $\mathbf{D}^{tr}$, it produced a sparse model by (2.4), which is ready for training.

2.1.1.3 Hessian-Based

Magnitude-based measuring neglects the mathematical connection between pruned parameters and leading prediction loss. A more delicate method try to dissect the connection between variation of parameters and corresponding loss. Dated back to 20th century, [23] formulated the final loss variation due to the pruning of elements. With the assumption of local identity and independence between element pruning. Formulation can be simplified as a quadratic objective function involving a dignoal hessian.

$$\delta\ell = \sum_i g_i \delta\mathbf{w}_i + \frac{1}{2}\sum_i h_{ii}\delta\mathbf{w}_i^2 + \sum_{i \neq j} h_{ij}\delta\mathbf{w}_i\delta\mathbf{w}_j \tag{2.5}$$

where g_i is the gradient of loss w.r.t $\mathbf{w}_i$, and h_{ij} are the elements of Hessian matrix. ℓ represents the predication loss, with δ ahead as the change of loss. With the simplification that off-diagonal elements in Hessian matrix and gradient can be

neglected, Eq.(2.5) can be reduced to:

$$\delta\ell = \frac{1}{2}\sum_i h_{ii}\delta\mathbf{w}_i^2 \tag{2.6}$$

Eq.(2.6) is the criterion of weights pruning.

[24] expanded [23] with weights update after one element is pruned, with objective is changed to measure the difference of layer output after pruning:

$$\delta\ell = (\frac{\partial\ell}{\partial\mathbf{w}})^\top\delta\mathbf{w} + \frac{1}{2}\delta\mathbf{w}^\top\mathbf{H}\delta\mathbf{w} \tag{2.7}$$

Minimizing Eq.(2.7) with the constraint that weights of index q is pruned leads to weights compensation and corresponding loss as:

$$\begin{aligned} \delta\mathbf{w} &= -\frac{\mathbf{w}_q}{[\mathbf{H}^{-1}]_{qq}}\mathbf{H}^{-1}e_q \\ L_q &= \frac{1}{2}\frac{\mathbf{w}_q^2}{[\mathbf{H}^{-1}]_{qq}} \end{aligned} \tag{2.8}$$

According to Eq.(2.8), [24] first measured the importance by l_q, then updated the rest of weights after pruning by $\delta\mathbf{w}$. Comparing with [23], it adopts a similar measuring criterion but it will compensate the un-pruned parameters to further minimize (2.7).

[25] applied [24] into the era of deep learning. [24] dealed with the scenario of single layer neural with tractable parameters for calculation of Hessian matrix, which poses great challenge in deep neural networks with millions of parameters and complex structure. [25]'s application treated (2.7) in a layer-wise framework: in each layer, it minimized the loss difference before and after pruning. These differences are accumulated and propagated to the final layer, which is proven to be bounded by a constant number. Besides, it adopted an approximation to calculate the Hessian matrix in each layer. By using the second order information in measuring parameters, [25] is able to achieve less performance drop after first pruning than [1], and recover much faster during finetuning.

2.1.1.4 Lookahead

[26] took into account the weight tensors of neighboring layers $\mathbf{W}^{l-1}$, $\mathbf{W}^{l+1}$ in addition to the original weight tensor $\mathbf{W}^{l}$. Basically, it measured a joint-effect of pruning one parameter in $\mathbf{W}^{l}$ while leaving the rest in $\mathbf{W}^{l-1}$, $\mathbf{W}^{l+1}$ unchanged:

$$\begin{aligned} \mathcal{L}^{l}(w) &= ||\mathbf{W}^{l+1}\mathbf{W}^{l}\mathbf{W}^{l-1} - \mathbf{W}^{l+1}\mathbf{W}^{l}|_{w=0}\mathbf{W}^{l-1}||_F \\ &= |w| \cdot ||\mathbf{W}^{l-1}[j.\ :]||_F \cdot ||\mathbf{W}^{l+1}[:, k]||_F \end{aligned} \tag{2.9}$$

where $\mathbf{W}^{l-1}[j.\ :]$ denoted the slice in $\mathbf{W}^{l-1}$ connecting to w, similar applies in $\mathbf{W}^{l+1}[:, k]$.

2.1.1.5 Lottery Ticket Hypothesis

Similar to the above-mentioned works which aim at finding importance parameters, [27] conjectured that: in a dense, randomly initialized deep neural network, there exist a subnetwork that can be trained in isolation and is capable of achieving the comparable performance using similar training efforts. Such subnetwork is denoted as “winning tickets”. To identify the existence of winning tickets, it designs the following experiments steps:

- Step 1: Randomly initialize a neural network $\mathcal{N}(\theta_0)$.
- Step 2: Train the network for j iterations, arriving at θ_j.
- Step 3: Prune $p\%$ of the parameters according to the magnitude in θ_j, creating mask matrix $\mathbf{M}$.
- Step 4: Reset the remaining parameters to initial value in θ_0, creating the winning ticket: $\mathcal{N}(\mathbf{M} \odot \theta_0)$. Repeat to Step 2 as iterative pruning.

Identification of winning ticket required an iterative pruning-training procedure. [28] extended lottery ticket to deeper neural network (such as ResNet50 on imageNet). It introduced “Learning Rate Warmup” and “late Resetting”. Specially, it use a learning rate an order of magnitude lower in [27]. And reset remaining parameters in Step 4 to the values after a small amount of training, instead of initialization.

Although lottery ticket hypothesis is able to retrieve a smaller core model inside the original model, it requires large effort due to the iterative pruning-training phases. [29] proposed to reuse the same winning tickets across a variety of datasets and optimizers. Specially, it generate the winning tickets for one training configuration (optimizer and dataset), then evalaute the performance on another configuration.

[30] analyzed lottery ticket by investigating more pruning criterions, instead of magnitude-based. Such as the increase in magnitude during training, which is empirically shown to achieve similar and best performance with magnitude-based. Besides, it proposed "Supermask", which is a trainable binary matrix imposed on parameters. Based on a random and fix initialization, training only the Supermask leads to a somehow improvement in the task.

2.1.2 Structured Pruning

Unstructured pruning possesses flexibility in choosing unimportant parameters, which is able to bring in higher sparsity and require smaller storage space. However, current hardware does not support efficient computation in irregular sparse matrix, or it requires extreme low compression rate to achieve significant computation acceleration, which on another hand harms the performance of pruned neural network.

This phenomenon advocate the usage of structured pruning, which combines the benefit of shrinking neural network but also take advantage of hardware acceleration.

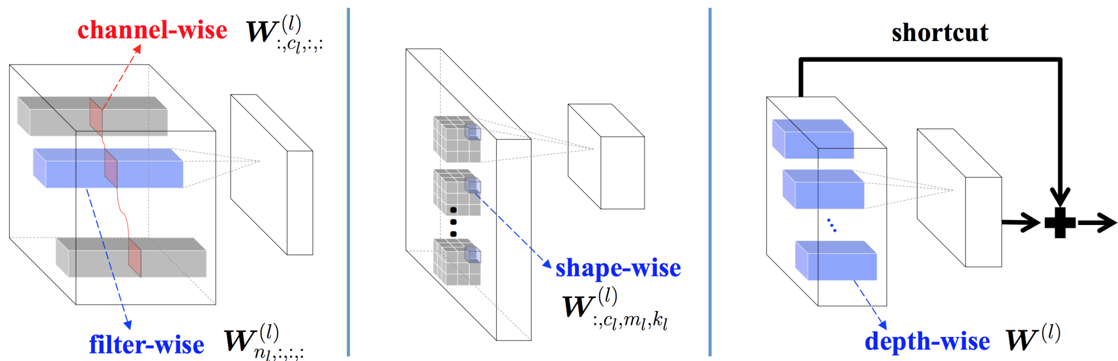


Figure 2.1: Four main types of Structured Pruning in convolution layers. [2]

Fig.2.1 demonstrates the four main types of Structured Pruning in convolution layer. Suppose convolution weights' dimension as $[c_{in} \times c_{out} \times w \times h]$.

- **Filter-wise** pruning reduces the entire filter, which results in pruned convolution weights' dimension as $[c_{in} \times c_{out}^{'} \times w \times h]$, where $c_{out}^{'} < c_{out}$.
- **Channel-wise** pruning can be regarded as "cut out of slices in convolution weights": $[c_{in}^{'} \times c_{out} \times w \times h]$, where $c_{in}^{'} < c_{in}$.
- **Shape-wise** methods prune weights in the same position among all filters, making a different shape of filters, leading to a same shape $[c_{in} \times c_{out} \times w \times h]$ of convolution weights.
- **Layer-wise** methods prune the whole layer.

[2] proposed to prune entire structure of deep neural network to improve memory locality, which can accelerate computation than individual pruning with low locality. It come up with 4 schemes to prune: 1) Filter-wise 2) Channel-wise 3) Shape-wise and 4) Layer-wise. Pruning is performed by imposing $L_{2,1}$ norm in corresponding scheme then training.

- Penalizing unimportant filers and channels:

$$\ell = \ell(\mathbf{W}; \mathbf{D}^{tr}) + \lambda_n \times \sum_{l=1}^{L} \left(\sum_{n_l=1}^{N_l} ||\mathbf{W}^l_{:,n_l,:,:}||_g \right) + \lambda_c \times \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} ||\mathbf{W}^l_{c_l,:,:,:}||_g \right) \quad (2.10)$$

  where N_l, C_l represents the number of filters and channels in layer l, λ_c, λ_n are the corresponding penalty parameters. $||\mathbf{W}||_g = |\sum ||\mathbf{W}_i||_2|_1$.

- Learning arbitrary shapes of filers:

$$\ell = \ell(\mathbf{W}; \mathbf{D}^{tr}) + \lambda_s \times \sum_{l=1}^{L} \left(\sum_{c_l=1}^{C_l} \sum_{m_l=1}^{M_l} \sum_{k_l=1}^{K_l} ||\mathbf{W}^l_{c_l,:,m_l,k_l}||_g \right) \quad (2.11)$$

  where M_l, K_l represents the width and height in each filter.

2.1.2.1 Filter/Channel Pruning

Filter and channel pruning are coupling: pruning a filter leads to pruning a corresponding channel in next layer, vice versa. Fig.2.2 visualizes how channel/filter

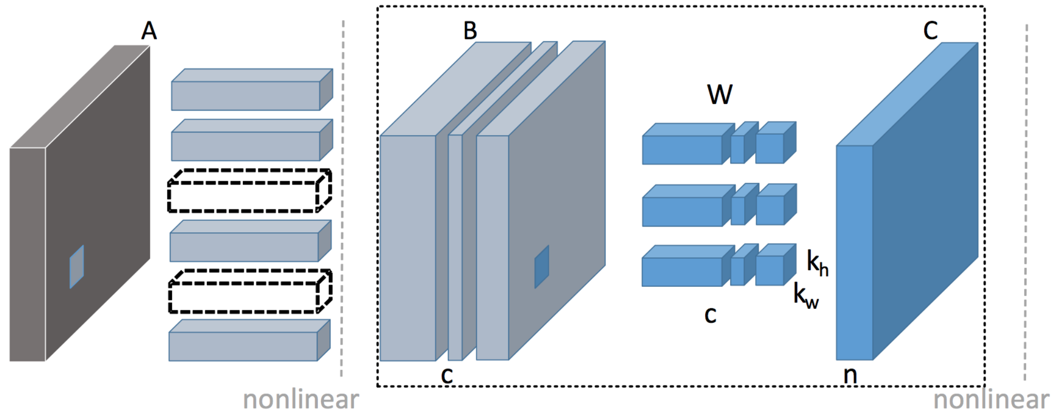


FIGURE 2.2: Pruning channels: pruning channels in layer l results in pruning filters in previous layer. Figure is cited from [3]

pruning is conducted and how it effects the before and latter layers: In l-th layer, if the n_l-th filter is pruned, it leads to the n_l-th channel of output feature being zeros, which inactivates the n_l-th channel in all filters in $l+1$ layer.

Similar with [1], [31] pruned entire filter in convolution layers by measuring the sum of their absolute value.

[32] recorded neurons the non-activate percentage during a batch of data to measure whether to prune this neuron.

[33] introduced a scaling factor γ for each channel. By training entire neural network with γ, filters with less importance is neglected with $\gamma = 0$.

[3] proposed a two-step layer-wise method: it firstly selected representative channels and pruned redundant ones by a LASSO regression based method. Then, it minimized the reconstruction loss. The whole process is formulated as:

$$\begin{aligned} &\arg\min_{\beta,\mathbf{W}} \frac{1}{N}||\mathbf{Y} - \sum_{i=1}^{c} \beta_i \mathbf{X}_i \mathbf{W}_i|| \\ &\text{s.t.} \qquad ||\beta||_0 \leq \epsilon \end{aligned} \tag{2.12}$$

For each layer, after c channels are selected by β, it adjusts the remaining $\mathbf{W}$ to compensate pruned error in layer output. It alternatively optimizes β and $\mathbf{W}$.

[34] and [3] solved the same problem but with different approaches. As a comparison, [34] transformed the problem of pruning filter in layer i as a input channel

combination problem to minimize reconstruction error in layer $i+1$. Specifically, it formulated the problem as:

$$\arg\min_{S} \sum_{i=1}^{m} (\hat{y} - \sum_{j \in S} \hat{x}_{i,j})^2 \tag{2.13}$$

where m is number of instance, S is the selected channels. It simplifies Eq.(2.13) as:

$$\arg\min_{S} \sum_{i=1}^{m} (\sum_{j \in T} \hat{x}_{i,j})^2 \tag{2.14}$$

where T is the complementary set of S. However, Solving Eq.(2.14) is still NP hard, thus it used a greedy strategy by adding one element to T at a time, and choosing the channel leading to the smallest objective value.
After pruning, it will further minimize the reconstruction error (Eq.(2.13)) by weighing the channels, which can be defined as:

$$\mathbf{W}^* = \arg\min_{\mathbf{W}} \sum_{i=1}^{m} (\sum_{j \in T} \hat{y}_i - \mathbf{W}_{i,j}^{\top} \hat{x}_i)^2 \tag{2.15}$$

Previous works generate a static structured pruned network, which is deployed to propose all input instances in the same structure. FBS [4] proposed to enhance or depress channel activation based on the input features. This approach shares a similar high-level idea with [35] to adjust output by channel reweighting. Instead, [4] focused on channel pruning. As Fig.2.3 visualizes the overflow of FBS: In layer

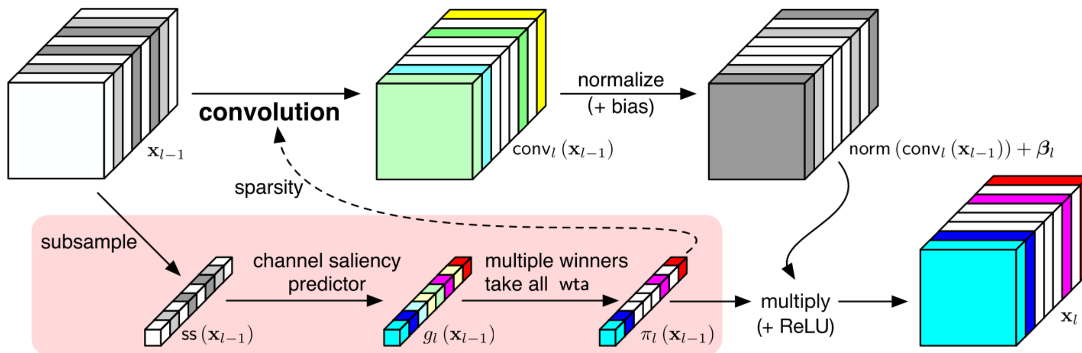


FIGURE 2.3: Overflow of FBS. Figure is cited from [4]

l, the input features $\mathbf{x}_{l-1}$ is fed in a saliency predictor $\pi_l(\cdot)$, which learns the

importance of $\mathbf{x}_{l-1}$ and amplify or suppress channels in output:

$$x_l = \pi_l(\mathbf{x}_{l-1}) \cdot (\text{norm}(\text{conv}_l(\mathbf{x}_{l-1}, \theta_l))) \tag{2.16}$$

$\pi_l(\cdot)$ is learned with L_1 penalty to produce sparse activation.

2.1.2.2 Miscellaneous

[36] proposed a data-free approach to prune entire neuron by finding the similar effect neurons. For a single neuron output with:

$$z_n = a_1 h(W_1^T X) + ... + a_i h(W_i^T X) = a_j h(W_j^T X) + ... \tag{2.17}$$

It tried to find the similar $h(W_i^T X)$ and $h(W_j^T X)$ in order to combine them as one term to eliminate the other one.

[37] proposed to train a sparse network by incorporating two constraints: Tensor Low Rank Constraints and Group Sparse Constraints. And study the effect of local minimum's momentum towards sparse performance: In practice, once a neuron reaches 0, it set its momentum to zero, forcing the neuron to maintain its value. To solve the tensor low rank constraints, it add penalty of tensor trace norm by using Low Rank Tensor Completion (LRTC)[38].

[5] learned to split the network weights into either a set or a hierarchy of multiple groups that use disjoint sets of features, by learning both the class-to-group and feature-to-group assignment matrices along with the network weights. This produced a tree-structured network that involves no connection between branched subtrees of semantically disparate class groups. In the viewpoint from weight itself, weights are learned into a block-diagonal sparse matrix, as Fig.2.4 demonstrates.

Such partition benefits parallelization, where each non-zero block is computed independently, thus greatly accelerate inference. To achieve block-diagonal sparsity, it enforces regularization as:

$$\min_{\omega,\mathbf{P},\mathbf{Q}} L(\omega, \mathbf{X}, \mathbf{y}) + \tag{2.18}$$

$$\sum_{l=1}^{L} \lambda ||\mathbf{W}^l||_2^2 + \sum_{l=1}^{L} \left(\gamma_1 R_W(\mathbf{W}^l, \mathbf{P}^l, \mathbf{Q}^l) + \gamma_2 R_D(\mathbf{P}^l, \mathbf{Q}^l) + \gamma_3 R_E(\mathbf{P}^l, \mathbf{Q}^l)\right) \tag{2.19}$$

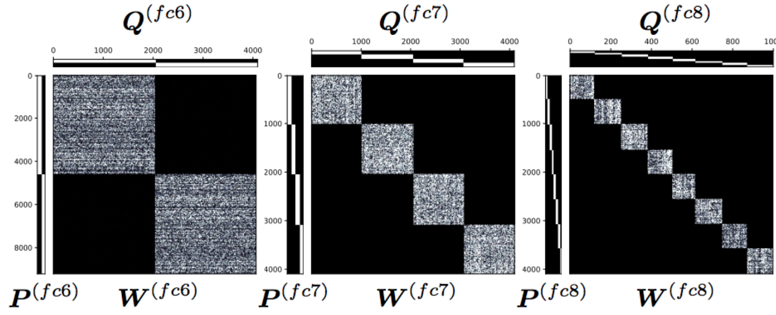


FIGURE 2.4: Weight is pruned with remaining diagnoal block. Figure is cited from [5]

where and $\mathbf{P}^l$ and $\mathbf{Q}^l$ are the set of feature-to-group and class-to-group assignment vectors respectively, for each layer l. For each layer, let $\mathbf{P}_g = \text{diag}(\mathbf{p}_g)$ and $\mathbf{Q}_g = \text{diag}(\mathbf{q}_g)$ be the feature and class group assignment matrix for group g respectively. Then $\mathbf{P}_g\mathbf{W}\mathbf{Q}_g$ represents the weight parameters associated with group. R_W $R_W(\mathbf{W}^l, \mathbf{P}^l, \mathbf{Q}^l)$ suppressed the off-diagonal elements by:

$$R_W(\mathbf{W}^l, \mathbf{P}^l, \mathbf{Q}^l) = \sum_g \sum_i ||((\mathbf{I} - \mathbf{P}_g)\mathbf{W}\mathbf{Q}_g)_{i*}||_2 \tag{2.20}$$

$$+ \sum_g \sum_j ||(\mathbf{P}_g\mathbf{W}(\mathbf{I} - \mathbf{Q}_g)_{*j})||_2 \tag{2.21}$$

where $(\mathbf{M})_i$ and $(\mathbf{M})_j$ denote i-th row and j-th column of $\mathbf{m}$. Eq.(2.20) imposes row/column-wise $L_{2,1}$-norm on the inter-group connections.
$R_D(\mathbf{P}^l, \mathbf{Q}^l)$ forced group assignment to be mutually exclusive by:

$$R_D(\mathbf{P}^l, \mathbf{Q}^l) = \sum_{i<j} \mathbf{p}_i \times \mathbf{p}_j + \sum_{i<j} \mathbf{q}_i \times \mathbf{q}_j \tag{2.22}$$

$R_E(\mathbf{P}^l, \mathbf{Q}^l)$ constraint the group assignments to be balanced:

$$R_E(\mathbf{P}^l, \mathbf{Q}^l) = \sum_g \left((\sum_i p_{gi})^2 + (\sum_j p_{gj})^2 \right) \tag{2.23}$$

[39] leveraged structured sparsity patterns of specific computation masks such as object detection and semantic segmentation. In particular, it exploit the sparsity from the road and sidewalk map mask as well as the model predicted foreground mask at lower resolution. For speed-up purposes, the same sparsity mask is reused

for every layer in our experiments, but it can also be computed from a different source per layer.

2.2 Quantization

Quantization aims at generating a deep neural network with parameters or activation within constraint values Ω. For example:

- $\Omega = \alpha \times \{-1, +1\}$: Binarization
- $\Omega = \alpha \times \{-1, 0, +1\}$: Ternarization
- $\Omega = \alpha \times \{-k+1, (0), k-1\}$: k-bit Shift-Wise Quantization
- $\Omega = \alpha \times \{-2^{k-1}, (0), +2^{k-1}\}$: k-bit Shift-Wise Quantization
- $\Omega = \{Q_0, ..., Q_{2^k}\}$: k-bit Asymmetric Quantization

Where k is an integer. By converting full-precision (FLOAT32) parameters or activation into quantized counterpart (INT8), float point turns into float-fix point or fix point multiplication for computation acceleration, which is widely supported in most frameworks.

2.2.1 Weight Quantization

[40] proposed network binarization through deterministic and stochastic rounding for parameters update after backpropagation. Specifically, it defined forward weights rounding as:

- Deterministic:

$$w_b = \begin{cases} +1 & \text{if} \quad w > 0 \\ -1 & \text{otherwise} \end{cases} \tag{2.24}$$

- Stochastic:

$$w_b = \begin{cases} +1 & \text{with probability p} = \sigma(w) \\ -1 & \text{with probability 1 - p} \end{cases} \tag{2.25}$$

In the meanwhile, it kept a set of full-precision weights w as assistance. In the forward propagation, w is firstly rounded, then final loss is calculated. In backward propagation, gradients are attained via w and w is updated as usual by gradients. These procedures continue until convergence. Besides, the updated w will be clipped within $[-1, 1]$ since w_b is constrained in $\{-1, 1\}$.

[15] extended the idea by introducing binary activation. The deterministic and stochastic rounding can be also applied in activation. Compared with [41], it instead used the "straight-through-estimator" (STE) [42] to solve the gradient vanish problem in rounding, thus gets rid of the full-precision weight w as assistance. STE estimates gradient after rounding by restraining the gradient before rounding:

$$\text{Forward:} \quad q = \text{round}(r) \tag{2.26}$$

$$\text{Backward:} \quad \frac{\partial C}{\partial r} = \frac{\partial C}{\partial q} 1_{|r|<1} \tag{2.27}$$

$1_{|r|<1} = 1$ when $r < 1$, 0 otherwise, this preserves the gradients information and cancels the gradient when r is too large. Besides, it considered how batch-normalization is used in binary network by shift-based methods.

[6] introduced a float value α_l known as the *scaling factor* in layer l, turning binarized weights as $\alpha_l \times \{\pm 1\}$. To accelerate inference, α_l is multiplied by layer input, such that weights become integer values $\{\pm 1\}$, converting float-point multiplication into float-point to integer computation.
Specifically, it estimated binary weights by formulating as:

$$J(\mathbf{B}, \alpha) = ||\mathbf{W} - \alpha \mathbf{B}||^2 = \underbrace{\mathbf{W}^\top \mathbf{W}}_{\text{fixed}} - 2\alpha \mathbf{W}^\top \mathbf{B} + \alpha^2 \underbrace{\mathbf{B}^\top \mathbf{B}}_{n} \tag{2.28}$$

where n is the number of elements in weights. Because $\mathbf{B} \in \{+1, -1\}, \mathbf{B}^\top \mathbf{B} = n$. By taking derivatives of J w.r.t to $\mathbf{B}, \alpha$, they reach optimal at:

$$\mathbf{B}^* = \text{sign}(\mathbf{W}) \tag{2.29}$$

$$\alpha = \frac{1}{n} ||\mathbf{W}||_{l_1} \tag{2.30}$$

Similar with [41], it used an extra full-precision weight for backpropagation, which is rounded by Eq.(2.29) in forward.
It extended weights binarization into activation binarization as "XNOR-Net", which

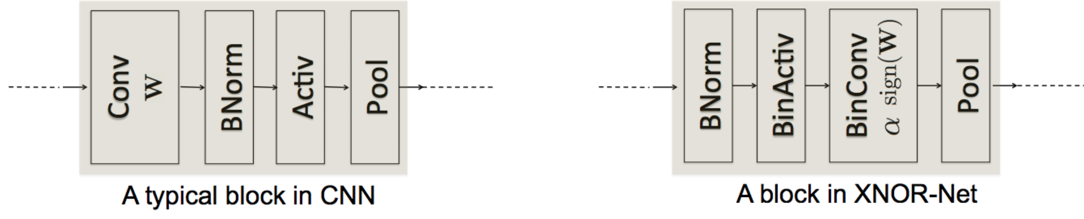


FIGURE 2.5: This figure contrasts the block structure in our XNOR-Network (right) with a typical CNN (left). Figure is cited from [6]

approximates layer output ($\mathbf{X}^\top \mathbf{W}$) by binary dot product:

$$\alpha^*, \mathbf{B}^*, \beta^*, \mathbf{H}^* = \underset{\alpha,\mathbf{B},\beta,\mathbf{B}}{\arg\min} ||\mathbf{X}^\top \mathbf{W} - \beta\alpha \mathbf{H}^\top \mathbf{B}||_2 \tag{2.31}$$

Similarly, it solved Eq.(2.31) as: $\mathbf{H} = \text{sign}(\mathbf{X}), \mathbf{B} = \text{sign}(\mathbf{W}), \alpha = \frac{1}{n}||\mathbf{W}||_{l_1}, \beta = \frac{1}{n}||\mathbf{X}||_{l_1}$. To decrease the information loss due to binarization, it altered the process produce by putting batch-normalization before input binarization as Fig.2.5 demonstrates.

[43] proposed a proximal Newton algorithm with diagonal Hessian approximation that directly minimized the loss w.r.t. the binarized weights. It formulated weight binarization as the following optimization problem:

$$\min_{\hat{\mathbf{W}}} \quad l(\hat{\mathbf{W}}) \tag{2.32}$$

$$\text{s.t.} \quad \hat{\mathbf{W}} = \alpha_l \mathbf{b}_l, \alpha > 0, \mathbf{b} \in \{\pm 1\}^n \tag{2.33}$$

It solved Eq.(2.32) using the proximal Newton method. At iteration t, the smooth term $l(\hat{\mathbf{W}})$ is replaced by the second-order expansion:

$$l(\hat{\mathbf{W}}^t) = l(\hat{\mathbf{W}}^{t-1}) + \Delta l(\hat{\mathbf{W}}^{t-1})^\top (\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1}) + \frac{1}{2}(\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1})^\top \mathbf{H}^{t-1} (\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1}) \tag{2.34}$$

For neural networks, the exact Hessian is rarely positive semi-definite and difficult to attain. To alleviate these problems, a popular approach is to approximate the Hessian by a diagonal positive definite matrix $\mathbf{D}$. Specifically, the second moment v in RMSprop, ADAM algorithms is an estimator of diag($\mathbf{H}^2$). It uses the square root of this v, which is readily available in Adam, to construct $\mathbf{D}$. Eq.(2.34) can be simplified and at the tth iteration of the proximal Newton algorithm, the following

subproblem is solved:

$$\min_{\hat{\mathbf{W}}^t} \quad \Delta l(\hat{\mathbf{W}}^{t-1})^\top(\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1}) + \frac{1}{2}(\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1})^\top \mathbf{D}^{t-1}(\hat{\mathbf{W}}^t - \hat{\mathbf{W}}^{t-1}) \quad (2.35)$$

$$\text{s.t.} \quad \hat{\mathbf{W}} = \alpha_l \mathbf{b}_l, \alpha > 0, \mathbf{b} \in \{\pm 1\}^n \quad (2.36)$$

The optimal solution of Eq.(2.35) can be obtained in closed-form as:

$$a_l^t = \frac{||\mathbf{d}_l^{t-1} \odot \mathbf{W}_l^t||}{\mathbf{d}_l^{t-1}} \quad (2.37)$$

$$\mathbf{b}_l^t = \text{sign}(\mathbf{W}_l^t) \quad (2.38)$$

Pay attention that Eq.(2.37) is an extension of Eq.(2.29). When $\mathbf{d}^{t-1} = \mathbf{I}$ (the second order information is not considered), Eq.(2.37) degenerates to Eq.(2.29).

[44] extended binarization in [43] into quantization. It kept the objective function in Eq.(2.35) but changed constraints as, for example, $\mathbf{b} \in \{\pm 1, 0\}$. Solving the optimization function leads to the solution: For a fixed $\mathbf{b}$, $\alpha = \frac{||\mathbf{b} \odot \mathbf{d}_l^{t-1} \odot \mathbf{w}_l^t||_1}{||\mathbf{b} \odot \mathbf{d}_l^{t-1}||_1}$, whereas when α is fixed, $\mathbf{b} = \mathbf{I}_{\frac{\alpha}{2}}(\mathbf{w}_l^t)$.

Ternarization introduces extra 0s. [16] assumed a prior distribution for parameters to find an approximated threshold for ternarization. Specifically, it firstly minimized the Euclidean distance between the full precision weights $\mathbf{W}$ and the ternary-valued weights $\mathbf{W}^t$ along with a nonnegative scaling factor α, which is formulated as:

$$\alpha^*, \mathbf{W}^{t*} = \arg\min_{\alpha, \mathbf{W}^t} J(\alpha, \mathbf{W}^t) = ||\mathbf{W} - \alpha \mathbf{W}^t||_2^2 \quad (2.39)$$

Following the similar solutions in [6] would get interdependent between α and $\mathbf{W}^t$. Thus, there is no deterministic solution in this way. To overcome this, it tried to find an approximated optimal solution with a threshold-based ternary function:

$$W_t^i = \begin{cases} +1 & \text{if} \quad W^i > \Delta \\ 0 & \text{if} \quad |W^i| < \Delta \\ -1 & \text{if} \quad W^i < -\Delta \end{cases} \quad (2.40)$$

For any given Δ, the optimal α can be computed as follows:

$$\alpha_\Delta^* = \frac{1}{|\mathbf{I}_\Delta|} \sum_{i \in \mathbf{I}_\Delta} |W_i| \quad (2.41)$$

where $\mathbf{I}_\Delta$ represents the weights fall into corresponding interval. By substituting α into Eq.(2.39), it gets a Δ-dependent equation, which can be simplified as follows:

$$\Delta^* = \arg\max_{\Delta>0} \frac{1}{|\mathbf{I}_\Delta|}(\sum_{i\in\mathbf{I}_\Delta} |W_i|)^2 \tag{2.42}$$

Eq.(2.42) has no straightforward solutions. Though discrete optimization can be made to solve the problem (due to states of W_is are finite), it can be time-consuming. Instead, it made a simple assumption that W_is are generated from uniform or normal distribution. In case of W_is are uniformly distributed in $[a, a]$ and Δ lies in $(0, a]$, the approximated Δ^* is $\frac{1}{3}a$, which equals to $\frac{2}{3} \times E(\mathbf{W})$. When W_is are generated from normal distributions $\mathbb{N}(0, \sigma^2)$, the approximated Δ^* is 0.6σ which equals to $0.75 \times E(|\mathbf{W}|)$.

[17] expanded ternaization $(\{0, \pm 1\})$ to $\{W^n, 0, W^p\}$, which set different scaling factors for positive and negative parameters ternarization. To update these factors, it first performed gradient descent, then gathered and averaged gradients for different factors:

$$\frac{\partial L}{\partial W_l^p} = \textstyle\sum_{i\in I_l^p} \frac{\partial L}{\partial W_l^p(i)} \tag{2.43}$$

$$\frac{\partial L}{\partial W_l^n} = \textstyle\sum_{i\in I_l^n} \frac{\partial L}{\partial W_l^n(i)} \tag{2.44}$$

where W_l^p, W_l^n are the projection points for assistant fully-precision weights $\hat{w}_l$ which is determined by heuristic thresholds Δ:

$$w_l^t = \begin{cases} W_l^p & \text{if} \quad \hat{w}_l > \Delta \\ 0 & \text{if} \quad |\hat{w}_l| < \Delta \\ W_l^n & \text{if} \quad \hat{w}_l < -\Delta \end{cases} \tag{2.45}$$

Finally, it borrowed the idea from STE and redefined gradients of $\hat{w}_l$ as:

$$\frac{\partial L}{\partial \hat{w}_l} = \begin{cases} W_l^p \times \dfrac{\partial L}{\partial w_l^t} & \text{if} \quad \hat{w}_l > \Delta \\ \dfrac{\partial L}{\partial w_l^t} & \text{if} \quad |\hat{w}_l| < \Delta \\ W_l^n \times \dfrac{\partial L}{\partial w_l^t} & \text{if} \quad \hat{w}_l < -\Delta \end{cases} \tag{2.46}$$

[45] is a combination of some above methods. It proposed to integrate the loss perturbation from the weight quantization and an incremental quantization strategy. The loss perturbation jointly considers the weight approximation error and the accompanying quantization impact to the loss function. Specifically, it is formulated as:

$$\min_{\hat{\mathbf{W}}_l} \quad L(\mathbf{W}_l) + a_1 L_p(\mathbf{W}_l, \hat{\mathbf{W}}_l) + a_2 E(\mathbf{W}_l, \hat{\mathbf{W}}_l) \tag{2.47}$$

$$\text{s.t.} \quad \hat{\mathbf{W}}_l \in \Omega \tag{2.48}$$

$L_p = |L(\mathbf{W}_l) - L(\hat{\mathbf{W}}_l)|$ encodes the loss difference between the quantized and full-precision models, $E = ||\mathbf{W}_l - \hat{\mathbf{W}}_l||^2$ represents the approximation error between quantized weight sets and full-precision counterparts. First order Taylor expansion of L_p is considered to approximate L_p:

$$L_p(\mathbf{W}_l, \hat{\mathbf{W}}_l) = |L(\mathbf{W}_l) - L(\mathbf{W}_l) - \tfrac{\partial L}{\partial \mathbf{W}_l}(\mathbf{W}_l - \hat{\mathbf{W}}_l)| \tag{2.49}$$

$$= |\tfrac{\partial L}{\partial \mathbf{W}_l}(\mathbf{W}_l - \hat{\mathbf{W}}_l)| \tag{2.50}$$

By a linear assumption that $\frac{\partial L}{\partial \mathbf{W}_l} \propto (\mathbf{W}_l - \hat{\mathbf{W}}_l)$, L_p, E can be reshaped into a uniform expression and gradient descent is updated as:

$$\mathbf{W}_l^t = \mathbf{W}_l^{t-1} - \gamma \frac{\partial L}{\partial \mathbf{W}_l^{t-1}} - \lambda \text{sign}(\mathbf{W}_l - \hat{\mathbf{W}}_l) \tag{2.51}$$

It further incorporated an incremental strategy to train the network by setting a partition of weights update effected by quantization while the rest are updated normally.

2.2.2 Weight & Activation & Gradient & Error Quantization

[18] further quantized activation and gradients. It defined quantize function $\mathbf{quantize}_k$ as:

$$r_o = \frac{1}{2^k - 1} \text{round}((2^k - 1) r_i) \tag{2.52}$$

For weights and gradient quantization, it firstly normalized float value, then applied quantization function. Specially:

- Weights:

$$\textbf{Forward:} \quad r_o = f_\omega^k(r_i) = 2\text{quantize}_k\left(\frac{\tanh(r_i)}{2\max|\tanh(r_i)|} + \frac{1}{2}\right) - 1 \tag{2.53}$$

$$\textbf{Backward:} \quad \frac{\partial c}{\partial r_i} = \frac{\partial r_o}{\partial r_i}\frac{\partial c}{\partial r_o} \tag{2.54}$$

- Gradients:

$$f_\gamma^k(dr) = 2\max(|dr|)\left[\text{quantize}_k\left(\frac{dr}{2\max(|dr)} + \frac{1}{2}\right) - \frac{1}{2}\right] \tag{2.55}$$

For activation, it directly applied quantization function.

[46] considered quantizing the activation of deep neural network by exploiting the statistics of network activation. It mainly studied the problem of gradient mismatch by using different forward and backward approximation. For general quantization function, optimal ones comes from:

$$Q^*(x) = \arg\min_Q E_x[(Q(x) - x)^2] \tag{2.56}$$

$$= \arg\min_Q \int p(x)(Q(x) - x)^2 dx \tag{2.57}$$

where $p(x)$ is the probability density function of x, thus the optimal quantizer of dot-product depends on their statistics. It further found that the distribution of activation after batch normalization is close to normal Gaussian. Therefore, it proposed the half-wave Gaussian quantizer as:

$$Q(x) = \begin{cases} q_i, & \text{if} \quad x \in (t_i, t_{i+1}) \\ 0, & x \leq 0 \end{cases} \tag{2.58}$$

t_is, q_is are the optimal quantization parameters which is attained by Lloyd's algorithm. It then proposed 3 types of backward functions to solve the problem of derivatives vanishing.

- Vanilla ReLU:

$$\tilde{Q}'(x) = \begin{cases} 1, & \text{if} \quad x > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.59}$$

- Clipped ReLU:

$$\tilde{Q}'(x) = \begin{cases} q_m, & x > q_m, \\ x, & x \in (0, q_m], \\ 0, & \text{otherwise} \end{cases} \tag{2.60}$$

- Log-tailed ReLU:
  Forward:

$$\tilde{Q}(x) = \begin{cases} q_m + \log(x - \tau), & x > q_m, \\ x, & x \in (0, q_m], \\ 0, & \text{otherwise} \end{cases} \tag{2.61}$$

  Backward:

$$\tilde{Q}'(x) = \begin{cases} \frac{1}{x - \tau}, & x > q_m, \\ 1, & x \in (0, q_m], \\ + + + + 0, & \text{otherwise} \end{cases} \tag{2.62}$$

  where $\tau = q_m - 1$, the log-tailed ReLU is identical to the vanilla ReLU for dot products of amplitude smaller than q_m, but gives decreasing weight to amplitudes larger than this.

[7] (WAGE) pioneered to extend quantization into training, by discretizing activation, weight, gradient and error (where error is defined as the gradient of loss w.r.t activation). Weight, activation, gradient and error is assigned with bit width as k_W, k_A, k_G, k_E. During training and inference, bit width is changed as Fig.2.6 shows:

It adopted a linear mapping with k-bit integers for simplicity, where continuous and unbounded values are discretized with uniform distance σ:

$$\sigma(k) = 2^{1-k}, k \in \mathbb{N}_+ \tag{2.63}$$

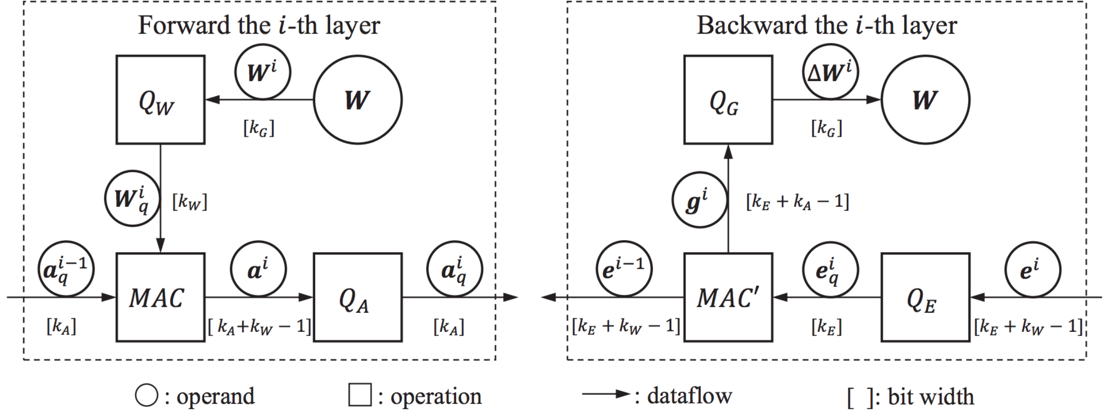


FIGURE 2.6: Four operators $Q_W(\cdot)$, $Q_A(\cdot)$, $Q_G(\cdot)$, $Q_E(\cdot)$ added in WAGE computation dataflow to reduce precision, bit width of signed integers are below or on the right of arrows, activations are included in Multiply-Accumulate (MAC) Cycles for concision. Figure is cited from [7]

As Fig.2.7 shows, it firstly introduced an additional monolithic scaling factor for shifting values distribution to an appropriate order of magnitude, then applied linear mapping, finally it proposed stochastic rounding. Quantization of variable

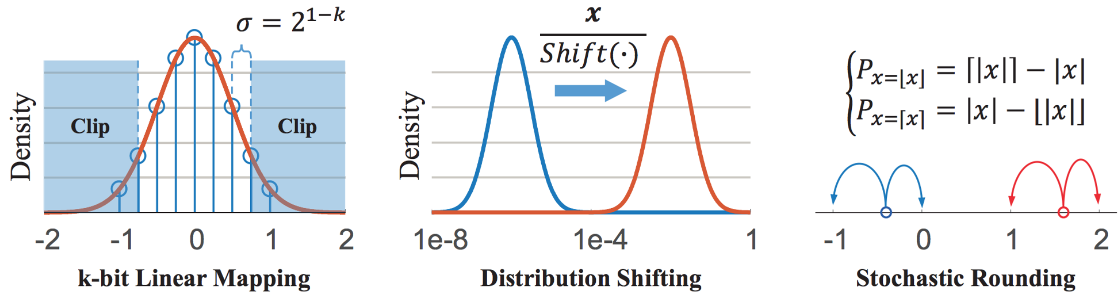


FIGURE 2.7: Procedure of quantization in [7]. Figure is cited from [7]

x into k bits is defined as:

$$Q(x,k) = \text{clip}\left\{\sigma(k)\cdot\text{round}\left[\frac{x}{\sigma(k)}\right], -1+\sigma(k), 1-\sigma(k)xw\right\} \tag{2.64}$$

Besides, it introduced some tricks to assist convergence:

- Weight Initialization:
  Because batch normalization is simplified to a constant scaling layer, weights should be cautiously initialized as:

$$\mathbf{W} \approx U(-L,L), L = \max\{\sqrt{\frac{6}{n_{in}}}, L_{min}\}, L_{min} = \beta\sigma \tag{2.65}$$

where n_{in} is the layer fan-in number, and the original limit $\sqrt{\frac{6}{n_{in}}}$ in MSRA [47] is calculated to keep same variance between inputs and outputs of the same layer theoretically. The additional limit L_{min} is a minimum value that the uniform distribution U should reach, and β is a constant greater than 1 to create overlaps between minimum step size σ and maximum value L.

- Weight Quantization:
  After applying direct quantization of weight based on Eq.(2.63), the variance of weights is scaled compared to the original limit, which will cause exploding of networks outputs. To alleviate the amplification effect it introduced a layer-wise shift-based scaling factor α to attenuate the amplification effect:

$$\alpha = \max\{\text{shift}\{\frac{L_{min}}{L}\}, 1\} \tag{2.66}$$

- Activation Quantization:

$$\mathbf{a}_q = Q(\frac{\mathbf{a}}{\alpha}, k_A) \tag{2.67}$$

- Error Quantization:
  Experiments uncovered that it is the orientations rather than orders of magnitude in errors that guides previous layers to converge. It firstly scaled error into $[-\sqrt{2}, +\sqrt{2}]$, then performed quantization:

$$\mathbf{e}_q = Q(\frac{\mathbf{e}}{\text{shift}(\max(|\mathbf{e}|))}, k_e) \tag{2.68}$$

- Gradient Quantization:
  It firstly define a minimum step and directions for updating weights:

$$\mathbf{g}_s = \eta \times \frac{\mathbf{g}}{\text{shift}(\max\{|\mathbf{g}|\})} \tag{2.69}$$

  To substitute accumulation of small gradients in latter case, it separated $\mathbf{g}_s$ into integer parts and decimal parts, then used a 16-bit random number generator to constrain high bit width $\mathbf{g}_s$ to k_G-bit integers stochastically:

$$\delta\mathbf{W} = Q_G(\mathbf{g}) = \sigma(k_G) \times \text{sgn}(\mathbf{g}_s) \times \{\lfloor|\mathbf{g}_s|\rfloor + \text{Bernoulli}(|\mathbf{g}_s| - \lfloor|\mathbf{g}_s|\rfloor)\} \tag{2.70}$$

[48] proposed a standard framework and pipeline to convert full-precision model into 8-bit quantization representation for both training and inference. It is able to maintain a satisfactory performance, even outperformance than full-precision model. The conversion scheme is conducted by applying affine mapping of integers q to real numbers r, in order to permit efficient arithmetic computation on SIMD hardware, i.e.:

$$r = S(q - Z) \tag{2.71}$$

where S, Z are quantization parameters. In inference using only integer arithmetic, consider multiplication of real numbers r_1, r_2, whose result is $r_3 = r_1 \times r_2$, we have:

$$S_3(q_3 - Z_3) = S_1(q_1 - Z_1)S_2(q_2 - Z_2) \tag{2.72}$$

which can be rewritten as:

$$q_3 = Z_3 + M(q_1 - Z_1)(q_2 - Z_2) \tag{2.73}$$

where $M = \frac{S_1 S_2}{S_3}$. With empirical experiments, M can be represented as $M = 2^{-n}M_0$, with $M_0 \in [0.5, 1)$. Therefore, in the whole computation, most arithmetic operation is conducted by integer addition and multoplication.

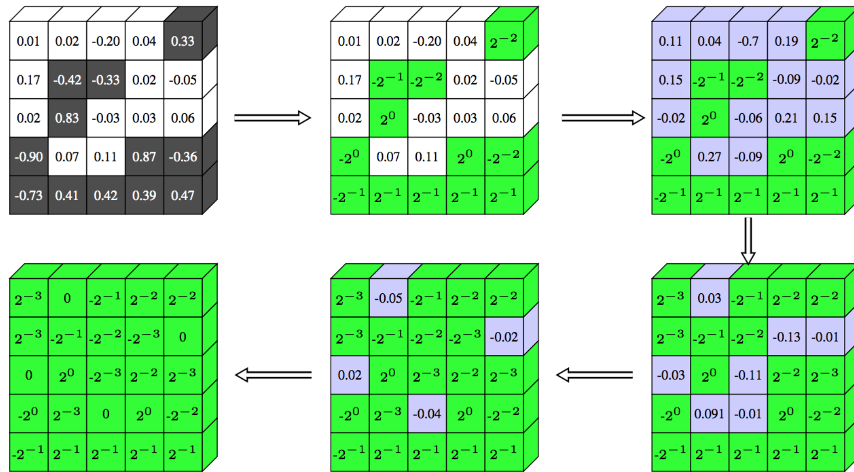


FIGURE 2.8: Procedure of [8]. Figure is cited from [8]

2.2.3 Differentiable Quantization

[49] proposed to quantize weights into more bits. It incorporated quantization loss into training objective. During the network training, it maintained a trainable fully-precision weight $\mathbf{W}$ while projects the updated $\mathbf{W}$ into quantized weights $\hat{\mathbf{W}}$ by ADMM [50].

[8] proposed to incrementally quantize a portion of parameters into $\{\pm 2^{n_1}, \pm 2^{n_2}, ..., 0\}$. Un-quantized parts are kept training. Fig.2.8 demonstrates the procedure.

2.2.4 Post-Training / Data-Free Quantization

Previous quantization relies on a machine learning framework to attain the quantized model, specially, model is trained with huge training data, under quantization constraint. However, in practical scenario, training data may be unaccessible, quantization is conducted only given a pre-trained full-precision model without data, or with limited data. This pipeline is named as "post-training / data-free quantization", for it does not reply on the traditional training diagram to attain the quantized model.

[51] distinguished four levels of quantization solution, in decreasing order of practical applicability, i.e. the higher the level, the more difficult it can be applied in real applications:

- **Level 1**: No data and no backpropagation is required.
- **Level 2**: Requires data but no backpropagation.
- **Level 2.5**: No data and backpropagation is required. Training instances may be attained by synthetic generation.
- **Level 3**: Requires both data and backpropagation, need fine-tuning to reach acceptable performance.
- **Level 4**: Requires both data and backpropagation, quantized model needs to be trained from scratch.

I have added level 2.5 for a recent progress on quantization with synthetic data.

For level 1 quantization, [52] presented an overview for model quantization, including the standard quantization methods. Other level 1 quantization methods hardly circumvent the basic quantization in [52], most efforts lie in parameters pre-processing for robust quantization or activation adjustment after quantization.

Based on the quantization method proposed in [52], [51] made progress by the following 4 steps:

- Equalizing channel's ranges over multiple layers: It noticed that the different distribution of channels pose difficulty in quantization. Also by exploiting the scaling equivalence in neural network:

$$\begin{aligned} x_2 &= f(W_2 f(W_1 x_0 + b1) + b2) \\ &= f(W_2 S \hat{f}(S^{-1} W_1 x_0 + S^{-1} b1) + b2) \\ &= f(\hat{W}_2 f(\hat{W}_1 x_0 + \hat{b}_1) + b2) \end{aligned} \tag{2.74}$$

  It used (2.74) to constrain the ranges of each channel to be similar, leading to more robust quantization.

- Biases absorbing: Running the equalization procedure on weights potentially increases the biases of the layers. This paper proposed to absorbs the biases in the subsequent layer. It noticed that: for a layer with ReLU activation r, there is a non-negative vector c such that: $r(Wx + b - c) = r(Wx + b) - c$ for all values of x. By using this observation, biases absorb from layer $i - 1$ to layer i is described as:

$$\begin{aligned} x_i &= W_i x_{i-1} + b_i \\ &= W_i(r(W_{i-1} x_{i-2} + b_{i-1}) + c - c) + b_i \\ &= W_i(r(W_{i-1} x_{i-2} + \hat{b}_{i-1}) + c) + b_i \\ &= W_i \hat{x}_{i-1} + \hat{b}_i \end{aligned} \tag{2.75}$$

  where c is approximated by the batch normalization statistics as: $c = \max(0, \min x_{i-1})$

- Quantization: now the normal quantization conversion ([52]) is applied to the parameters.

- Correction of quantizaton bias in activation.

[53] explored the equivalent weight arrangement and quantization noise analysis for robust quantization.

For level 2.5, most methods rely on GAN or batch normalization statistics matching to generate training data from a given pre-trained model. [54] proposed three types of schemes to generate synthetic data, which is then used in a distillation framework to train the quantized model.

[55] proposed quantization without access to the training or validation dataset. Instead, it optimized a distilled dataset by matching the statistics of batch normalization across layers. Then, the generated dataset is utilized to perform post-training quantization.

2.2.5 Miscellaneous

[56] proposed a method to approach full-precision weight by an addition of a series of pow of 2. In each layer, it estimates the real-value weight filter $\mathbf{W}$ using the linear combination of M binary filters $\mathbf{B}_1, ..., \mathbf{B}_M$:

$$\mathbf{W} \approx \sum_{i=1}^{M} \alpha_i \mathbf{B}_i \tag{2.76}$$

$\mathbf{B}_i$ is attained by: $\mathbf{B}_i = \text{sign}(\mathbf{W} - \text{mean}(\mathbf{W}) + u_i \text{std}(\mathbf{W}))$, $u_i = -1 + (i-1)\frac{2}{M-1}, i = 1, ..., M$ which to shift evenly over the range $[-\text{std}(\mathbf{W}), \text{std}(\mathbf{W})]$, or leave it to be trained by the network. Similar to [46], it observes that the full-precision weights tend to have a symmetric, non-sparse distribution, which is close to Gaussian.
For backward, STE is used to pass gradient from $\mathbf{B}_i$ to $\mathbf{W}$:

$$\frac{\partial c}{\partial \mathbf{W}} = \sum_{i}^{M} \alpha_i \frac{\partial c}{\partial \mathbf{B}_i} \tag{2.77}$$

It further binarizes activation with a clip function to normalize as assistance, specially:

$$\textbf{Forward:} \qquad \mathbf{A} = 2\mathbb{I}_{h_v(\mathbf{R})>0.5} - 1 \tag{2.78}$$

$$h_v(\mathbf{R}) = \text{clip}(x + v, 0, 1) \tag{2.79}$$

$$\textbf{Forward:} \qquad \frac{\partial c}{\partial \mathbf{R}} = \frac{\partial c}{\partial \mathbf{A}} \odot \mathbb{I}_{0 \geq \mathbf{R}-v \leq 1} \tag{2.80}$$

[9] combined network pruning and weight quantization in a single learning framework that performed pruning and quantization jointly, and in parallel with fine-tuning. As Fig.2.9 demonstrates: it firstly eliminates weights with small absolute value, then partitions the remaining weights into intervals, finally averages weights falling within the corresponding quantization interval to get quantized weights. In the meanwhile, it preserves a set of full-precision weights as reference and use loss w.r.t quantized weights to update.

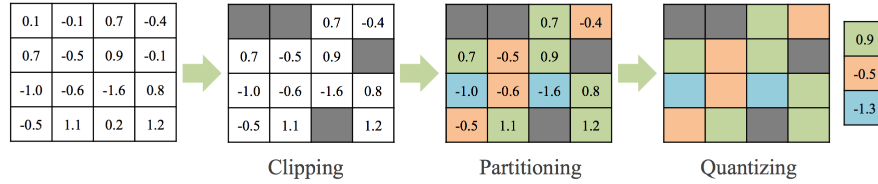


FIGURE 2.9: Procedure of [9]. Figure is cited from [9]

[57] proposed Hessian-weighted k-means clustering to quantize network parameters. Following the derivation in [23], target loss brought by weights distortion can be represented as:

$$\delta L \approx \frac{1}{2} \sum_{i=1}^{N} h_{ii}(\hat{\mathbf{w}}) |\hat{w}_i - \tilde{w}_i|^2 \tag{2.81}$$

where $\hat{w}$ are quantized weights and $\tilde{w}$ are pretrained weights. Based on such observation, it minimized Eq.(2.81) by finding optimal clustering centroids in $\tilde{w}$:

$$\underset{C_1,\ldots C_k}{\arg\min} \sum_{j=1}^{k} \sum_{\tilde{w}_i \in C_j} h_{ii} |\tilde{w}_i - c_j|^2 \tag{2.82}$$

where C_is are the clustering centroids calculated as: $C_j = \frac{\sum_{\tilde{w} \in C_j} h_{ii} \tilde{w}_i}{\sum_{\tilde{w} \in C_j} h_{ii}}$.

2.2.6 Comparison of Various Methods

I have compared most state-of-the-art quantization methods, including their bit width in weight/activation/gradient/error in Table 2.2 and performance of various architecture in Table 2.3 (AlexNet), 2.4 (VGG16), 2.5 (ResNet-18), 2.6 (ResNet-50), 2.7 (GoogleNet).

As shown in Table 2.3, in traditional deep network architecture, binarization (BWN) is able to achieve lossless performance. Quantizaion on both parameters and activation still face challenges (DoReFa-Net).

In more recent architecture such as ResNet-18, ResNet-50 and GoogLenet, models become more compact and informative. Binarization (BWN) leads to significant degradation. More bit quantization (INQ, ExNN) is required to achieve comparable performance.

Method	weights	activation	gradient	error	init
BC [41]	1	32	32	32	PT
BNN [58]	1	1	32	32	PT
BWN [6]	$\alpha \times 1$	32	32	32	PT
XNOR [6]	$\alpha \times 1$	$\alpha \times 1$	32	32	PT
TWN [16]	$\alpha \times 2$	32	32	32	PT
TTQ [17]	$\{\alpha, 0, \beta\}$	32	32	32	PT
INQ [8]	$\alpha \times 16$	32	32	32	PT
DoReFa [18]	8	8	8	32	PT/Scratch
WAGE [7]	2	8	8	8	PT

TABLE 2.2: Comparison of different quantization method, "PT" represents **P**retrained **M**odels

<table>
<tr><th>Method</th><th>Top-1/Top-5 Increase(%)</th><th>Top-1/Top-5 Original(%)</th><th>Remarks{W,A,G,E}</th></tr>
<tr><td>BC</td><td>-21.2/-19.2</td><td rowspan="4">56.6/80.2</td><td rowspan="2">{1,32,32,32}</td></tr>
<tr><td>BWN</td><td>0.2/-0.8</td></tr>
<tr><td>BNN</td><td>-27.8/-29.8</td><td rowspan="2">{1,1,32,32}</td></tr>
<tr><td>XNOR-Net</td><td>-12.4/ -11</td></tr>
<tr><td>TTQ</td><td>0.3/-0.6</td><td>57.2/80.3</td><td>{2,32,32,32}</td></tr>
<tr><td rowspan="2">DoReFa-Net</td><td>-9.8/-</td><td rowspan="2">55.9/-</td><td>{1,2,6,32}</td></tr>
<tr><td>-2.9/-</td><td>{1,4,32,32}</td></tr>
<tr><td>INQ</td><td>0.15/0.23</td><td>57.24/80.23</td><td>{17,32,32,32}</td></tr>
<tr><td rowspan="2">ExNN</td><td>-0.8/-0.6</td><td rowspan="2">60.0/82.4</td><td>{3,32,32,32}</td></tr>
<tr><td>0/-0.2</td><td>{4,32,32,32}</td></tr>
</table>

TABLE 2.3: Performance Comparison of Various Quantization Methods in AlexNet

Method	Top-1/Top-5 Increase(%)	Top-1/Top-5 Original(%)	Remarks{W,A,G,E}
INQ	-2.28/-1.65	68.54/88.65	{17,32,32,32}
ExNN	0.6/0.8	71.1/89.9	{3,32,32,32}
	1.1/1.0		{4,32,32,32}

TABLE 2.4: Performance Comparison of Various Quantization Methods in VGG16

Method	Top-1/Top-5 Increase(%)	Top-1/Top-5 Original(%)	Remarks{W,A,G,E}
BWN	-8.5/-6.2	69.3/89.2	{1,32,32,32}
XNOR-Net	-18.1/ -16.0		{1,1,32,32}
TWN	-3.6/-2.6	65.4/86.8	{2,32,32,32}
TTQ	-3/-2	69.6/89.2	{2,32,32,32}
ExNN	-1.6/-1.1	69.1/89.0	{3,32,32,32}
	-1.1/-0.7		{4,32,32,32}
INQ	0.71/0.41	68.27/88.69	{17,32,32,32}
	-2.25/-1.56		{2,32,32,32}

TABLE 2.5: Performance Comparison of Various Quantization Methods in ResNet18

Method	Top-1/Top-5 Increase(%)	Top-1/Top-5 Original(%)	Remarks{W,A,G,E}
INQ	0.71/0.41	73.22/88.69	{17,32,32,32}
ExNN	-1.4/-0.7	75.3/92.2	{3,32,32,32}
	-1.3/-0.6		{4,32,32,32}

TABLE 2.6: Performance Comparison of Various Quantization Methods in ResNet50

Method	Top-1/Top-5 Increase(%)	Top-1/Top-5 Original(%)	Remarks{W,A,G,E}
BWN	-5.8/-3.9	71.3/90.0	{1,32,32,32}
INQ	0.1/0.25	68.69/89.03	{17,32,32,32}
ExNN	-2.8/-1.6	68.7/88.9	{3,32,32,32}
	-2.4/-1.4		{4,32,32,32}

TABLE 2.7: Performance Comparison of Various Quantization Methods in GoogLenet

2.3 Distillation

[59] proposed distillation in neural network, which aimed at improving performance by using deeper and pretrained model's final output as ground-truth, together with labels to train a shadow network. It assumed that these operations act like teachers teaching students or knowledge distillation from "high temperature" (pretrained models) to "low temperature" (models to be trained). By using distillation, shadow networks preserve performance as deeper ones with much less computations and

storage space. In this viewpoint, distillation itself can be regarded as a type of optimizing neural network.

[10] extended pure distillation to combine quantization techniques. It proposed two methods. The first method is called *quantized distillation*, which leveraged distillation during the training process, by incorporating distillation loss into the training of a smaller network whose weights are quantized to a limited set of levels. The second method, *differentiable quantization*, optimized the location of quantization points through stochastic gradient descent, to better fit the behavior of the pretrained model.

- **Quantized Distillation**
  As Fig.2.10 demonstrates, it performed the SGD step on the full-precision model, but computed the gradient on quantized model. Besides, it accumulated the error at each projection step into the gradient for the next step. One can think of this process as if collecting evidence for whether each weight needs to move to the next quantization point or not. Crucially, the error accumulation prevents the algorithm from getting stuck in the current solution if gradients are small, which would occur in a naive projected gradient approach.

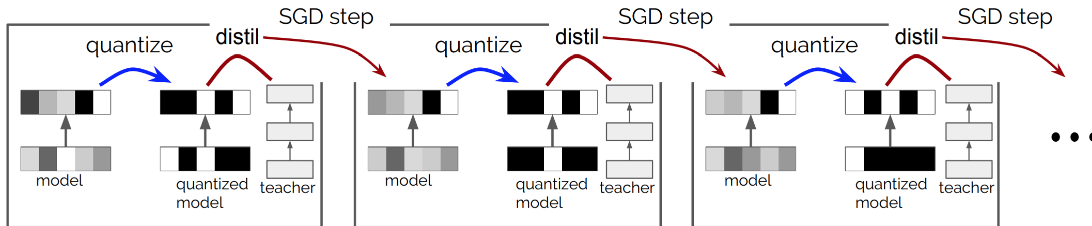


FIGURE 2.10: Procedure of quantization distillation in [10]. Figure is cited from [10]

- **Differentiable Quantization**
  This method acted as a general approach of improving the accuracy of a quantized neural network, by exploiting non-uniform quantization point placement. Specially, it let $p = (p_1, ..., p_s)$ be the vector of quantization points, and let $Q(v, p)$ be quantization function. Gradient of Q w.r.t p is defined as:

$$\frac{\partial Q(v,p)_i}{\partial p_j} = \begin{cases} \alpha_i & \text{if } v_i \text{ has been quantized to } p_i \\ 0 & \text{otherwise} \end{cases} \tag{2.83}$$

where α_i is i-th element of the scaling factor.

Besides, it proved that stochastic quantization is equivalent to adding Gaussian noise. Specially:

$$Q(v)^\top x = v^\top x + \epsilon$$

2.4 Architecture Modification

Grouped convolution is a improvement of normal convolution on inference and training efficiency. Most modification for acceleration and storage reduction can be traced back to grouped convolution. It can be dated back to [11] for fitting in GPU memory in training. However, [60] analyzed the effect of grouped convolution and showed that it learn better representations. By studying the correlation matrix between adjacent layers' responses (output), it found that filter relationships are sparse. While filter groups can be learnt with a block-diagonal structured sparsity on the channel dimension.

In practice, grouped convolution as "Diagonal Block Sparse Matrix Multiplication'. Basically, convolution can be represented as matrix multiplication as shown in Fig.2.11 while grouped convolution, it can be represented as in Fig.2.12:

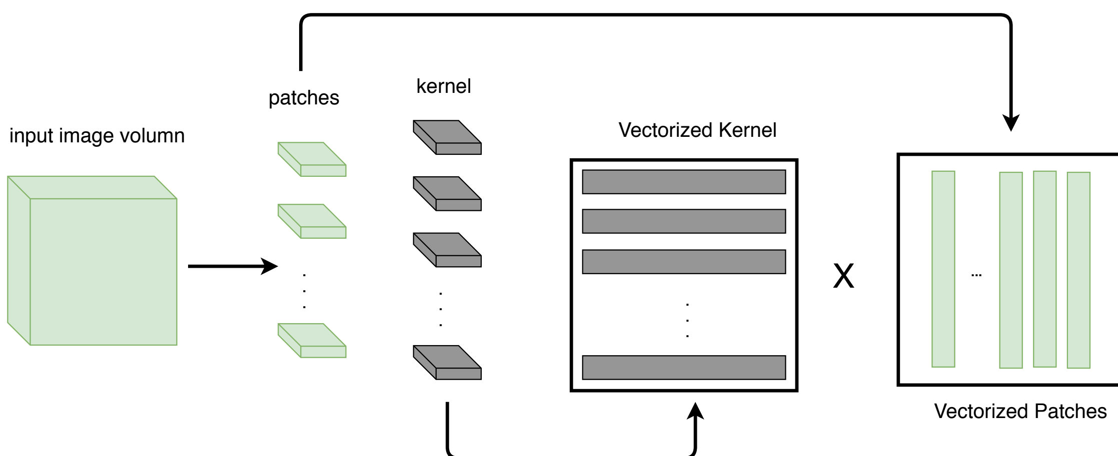


FIGURE 2.11: Convolution in matrix multiplication: Input image is extracted into multiple patches, which share the same dimension of the kernels. The patches are arranged into a vectorized matrix, while the kernels are at the same time reshaped into vectorized matrix. Convolution is formulated as the matrix multiplication.

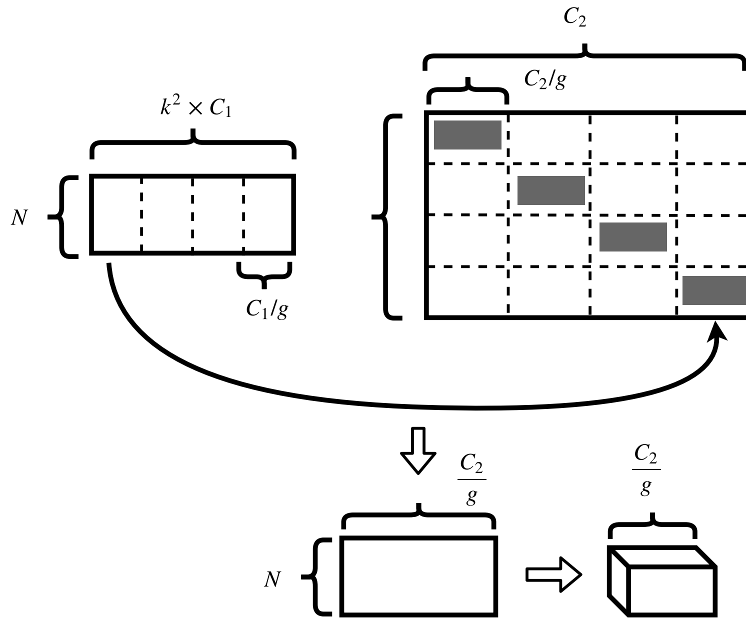


FIGURE 2.12: Grouped convolution in matrix multiplication: Left part represents input feature, which is processed by divided into g groups. Each slides of input feature is multiplied with the corresponding diagonal part of parameters in right part, i.e. Only the diagonal blocks of parameters are remained. Grouped convolution reduce number of parameters and computation complexity to $1/g$.

[61] proposed to incorporate grouped convolution, and shuffle the output channels from different group for information fusion, as shown in Fig.2.13

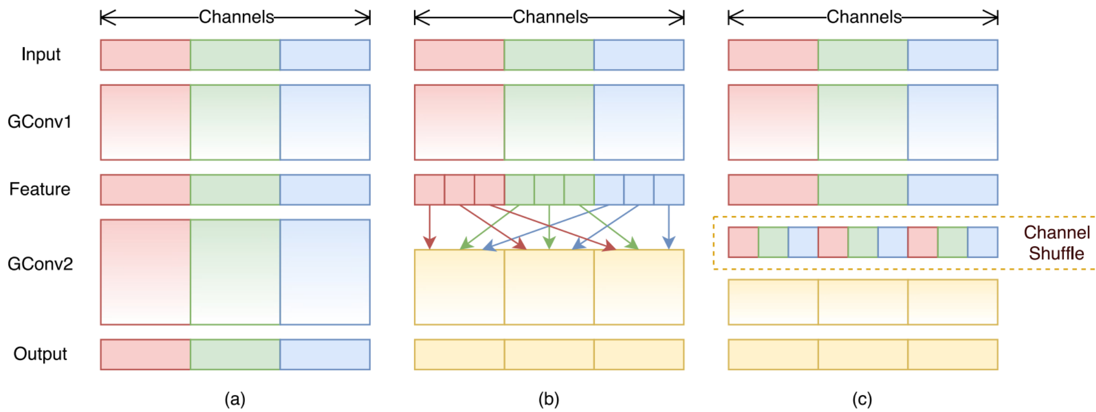


FIGURE 2.13: output channels from different grouped convolution are shuffled.

[62] proposed multi-stage grouped convolution in different layers, as shown in Fig.2.14.

Extended from grouped convolution, [63] set number of group as number of input channels, which conducted convolution channel-wisely. After that, a 1×1 pointwise convolution to fuse the learned feature map.

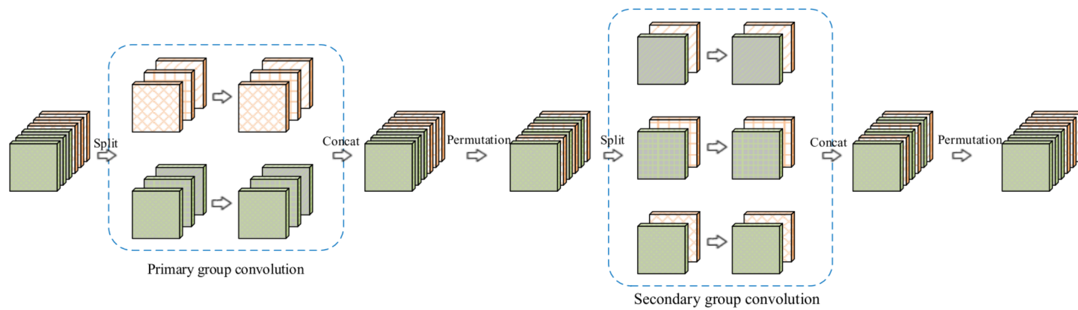
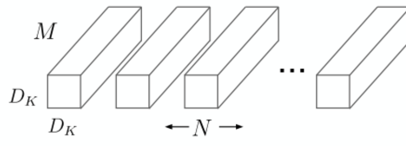
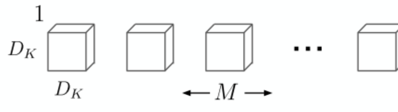


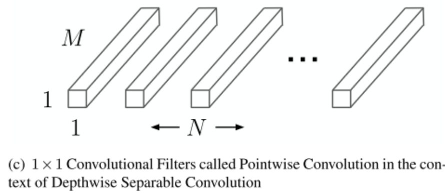
FIGURE 2.14: Illustrating the interleaved group convolution, with $L = 2$ primary partitions and $M = 3$ secondary partitions. The convolution for each primary partition in primary group convolution is spatial. The convolution for each secondary partition in secondary group convolution is point-wise (1×1).



(a) Standard Convolution Filters



(b) Depthwise Convolutional Filters



(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Method	#. Parameter	#.Computation
Standard	$H \times W \times M \times N$	$H^2 \times W^2 \times M \times N$
Depth-Wise	$H \times W \times M + N$	$H^2 \times W^2 \times M + M \times N \times H \times W$

Chapter 3

Layer-Wise Model Quantization

3.1 Introduction

Traditional [1] model compression methods build their algorithms on the framework of supervised learning, i.e. Given training set $\mathbf{D}^{tr} = \{\mathbf{X}^{tr}, \mathbf{y}^{tr}\}$ and specific model architecture $\mathcal{N}(\cdot)$, it forwards $\mathbf{X}^{tr}$ in $\mathcal{N}$, with the outputs comparing $\mathbf{y}^{tr}$ to generate loss ℓ, which is backprogated to $\mathbf{W}$ with gradient $g_{\mathbf{W}}$. Then $\mathbf{W}$ is updated by $g_{\mathbf{W}}$ using gradient descent. Such trend of methods doesn't utilize any property of model compression. Instead, it entirely replies on the paradigm of supervised learning, hoping to achieve training convergence. Though its success in many applications and empirical experiments, the mathematical defects of gradient descent in training compressed model have not been solved, which does neither guarantee its future usage, nor shed light to the investigation of model compression.

We try to avoid such dilemma by finding the compressed version of a well-train full precision model. Specially, instead of training a compressed model using the same data based on the well-train initialization, we approximate its network output by a compressed model, under mathematical formulation and constraints.

In this chapter, based on the previous work on layer-wise model pruning [25], we propose a new layer-wise quantization method for deep neural networks, aiming to achieve the following goals: 1) For each layer, parameters can be highly quantized,

[1]The work in this chapter has been published in 33rd AAAI Conference on Artificial Intelligence (AAAI 2019).

while the reconstructed error is small. 2) There is a theoretical guarantee on the overall prediction performance of the compressed deep neural network in terms of reconstructed errors for each layer. 3) Only a limited number of training instances is required to finish the whole process, enabling model quantization in a low-resource scenario.

To achieve our first goal, we borrow an idea from some classic pruning approaches for shallow neural networks, such as optimal brain damage (OBD) [23] and optimal brain surgeon (OBS) [24]. These classic methods approximate a change in the error function via functional Taylor Series, and identify unimportant weights based on second order derivatives. Though these approaches have proven to be effective for shallow neural networks, it remains challenging to extend them for deep neural networks because of the high computational cost on computing second order derivatives, i.e., the inverse of the Hessian matrix over all the parameters. In this work, as we restrict the computation on second order derivatives w.r.t. the parameters of each individual layer only, i.e., the Hessian matrix is only over parameters for a specific layer, the computation becomes tractable. Moreover, we utilize characteristics of back-propagation for fully-connected layers in well-trained deep networks to further reduce computational complexity of the inverse operation of the Hessian matrix.

To achieve our second goal, based on the theoretical results in [64], we provide a proof on the bound of performance drop before and after compression in terms of the reconstructed errors for each layer. With such a layer-wise compression framework using second-order derivatives for trimming parameters for each layer, we empirically show that: after quantization is conducted for a layer, distillation with scarce data is sufficient to tune un-compressed parameters. After quantizing all layers, it is able to achieve negligible performance drop.

The third goal is achieved by adopting model distillation between a full-precision model and the targeted quantized counterpart. Specially, we finetune the unquantized layers using MSE (Mean-Square-Error) from full-precision model's output. It empirically shows that distillation brings with more stable and faster convergence, leading to less data burden.

The proposed method is denoted as **L**ayer-Wise **D**eep **N**eural Network **Q**uatization (L-DNQ). In the following, we first formulate the problem statement in Sec.3.2.

Then introduce the core optimization objective involved in both compression in Sec.3.3. A Hessian approximation method is explained in Sec.3.4. Then we introduce detailed quantization method in Sec.3.5.

3.2 Problem Statement

Given a training set of n instances, $\{(\mathbf{x}_j, y_j)\}_{j=1}^n$, and a well-trained deep neural network of L layers (excluding the input layer)[2]. Denote the input and the output of the whole deep neural network by $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{d\times n}$ and $\mathbf{Y} \in \mathbb{R}^{n\times 1}$, respectively. For a layer l, we denote the input and output of the layer by $\mathbf{Y}^{l-1} = [\mathbf{y}_1^{l-1}, ..., \mathbf{y}_n^{l-1}] \in \mathbb{R}^{m_{l-1}\times n}$ and $\mathbf{Y}^l = [\mathbf{y}_1^l, ..., \mathbf{y}_n^l] \in \mathbb{R}^{m_l\times n}$, respectively, where $\mathbf{y}_i^l$ can be considered as a representation of $\mathbf{x}_i$ in layer l, and $\mathbf{Y}^0 = \mathbf{X}$, $\mathbf{Y}^L = \mathbf{Y}$, and $m_0 = d$. Using one forward-pass step, we have $\mathbf{Y}^l = \sigma(\mathbf{Z}^l)$, where $\mathbf{Z}^l = \mathbf{W}_l{}^\top \mathbf{Y}^{l-1}$ with $\mathbf{W}_l \in \mathbb{R}^{m_{l-1}\times m_l}$ being the matrix of parameters for layer l, and $\sigma(\cdot)$ is the activation function. For convenience in presentation and proof, we define the activation function $\sigma(\cdot)$ as the rectified linear unit (ReLU). We further denote by $\mathbf{\Theta}_l \in \mathbb{R}^{m_{l-1}m_l\times 1}$ the vectorization of $\mathbf{W}_l$. For a well-trained neural network, $\mathbf{Y}^l$, $\mathbf{Z}^l$ and $\mathbf{\Theta}_l^*$ are all fixed matrices and contain most information of the neural network. The goal of pruning is to set the values of some elements in $\mathbf{\Theta}_l$ to be zero. In quantization, we discretize the values of all elements of $\mathbf{\Theta}_l$ for each layer into a finite set $\mathbf{\Omega}_l$, e.g. symmetric: $\mathbf{\Omega}_l = \{-\alpha_l, 0, \alpha_l\}$ or random: $\mathbf{\Omega}_l = \{\alpha_l, \beta_l, \gamma_l\}$. We denote the quantized $\mathbf{\Theta}_l$ by $\hat{\mathbf{\Theta}}_l$.

3.3 Layer-Wise Error

During layer-wise compression in layer l, the input $\mathbf{Y}^{l-1}$ is fixed as the same as the well-trained network. Suppose we set the q-th element of $\mathbf{\Theta}_l$, denoted by $\mathbf{\Theta}_{l_{[q]}}$, to be zero; or quantize $\mathbf{\Theta}_l$ by some quantization rule (such as nearest), and get a new parameter vector, denoted by $\hat{\mathbf{\Theta}}_l$. With $\mathbf{Y}^{l-1}$, we obtain a new output for layer l, denoted by $\hat{\mathbf{Y}}^l$. Consider the mean square root error between $\hat{\mathbf{Y}}^l$ and $\mathbf{Y}^l$ over the

[2]For simplicity in presentation, we suppose the neural network is a feed-forward (fully-connected) network. We will show how to extend our method to filter layers in Convolutional Neural Networks.

whole training data as the layer-wise error:

$$\varepsilon^l = \sqrt{\frac{1}{n}\sum_{j=1}^{n}\left((\hat{\mathbf{y}}_j^l - \mathbf{y}_j^l)^\top(\hat{\mathbf{y}}_j^l - \mathbf{y}_j^l)\right)} = \frac{1}{\sqrt{n}}\|\hat{\mathbf{Y}}^l - \mathbf{Y}^l\|_F, \tag{3.1}$$

where $\|\cdot\|_F$ is the Frobenius Norm. Note that for any single parameter pruning, one can compute its error ε_q^l, where $1 \leq q \leq m_{l-1}m_l$, and use it as a pruning criterion. This idea has been adopted by some existing methods [65]. However, in this way, for each parameter at each layer, one has to pass the whole training data once to compute its error measure, which is very computationally expensive. A more efficient approach is to make use of the second order derivatives of the error function to help identify importance of each parameter.

We first define an error function $E(\cdot)$ as

$$E^l = E(\hat{\mathbf{Z}}^l) = \frac{1}{n}\left\|\hat{\mathbf{Z}}^l - \mathbf{Z}^l\right\|_F^2, \tag{3.2}$$

where $\mathbf{Z}^l$ is outcome of the weighted sum operation right before performing the activation function $\sigma(\cdot)$ at layer l of the well-trained neural network, and $\hat{\mathbf{Z}}^l$ is outcome of the weighted sum operation after compression at layer l. Note that $\mathbf{Z}^l$ is considered as the desired output of layer l before activation. The following lemma shows that the layer-wise error is bounded by the error defined in (3.2).

Lemma 3.3.1. With the error function (3.2) and $\mathbf{Y}^l = \sigma(\mathbf{Z}^l)$, the following holds: $\varepsilon^l \leq \sqrt{E(\hat{\mathbf{Z}}^l)}$.

Proof. Given $x, y \in \mathbb{R}$, $f(\cdot) : \mathbb{R} \to \mathbb{R}$, it is easy to show that: for $f' \leq 1$, we always have: $[f(x) - f(y)]^2 \leq (x - y)^2$.

For activation function ReLU or sigmoid, $\sigma' \leq 1$, which leads to $[\sigma_l(\hat{\mathbf{Z}}_i^l) - \sigma_l(\mathbf{Z}_i^l)]^2 \leq (\hat{\mathbf{Z}}_i^l - \mathbf{Z}_i^l)^2$, where i represents the index of each elements in $\hat{Z}/Z$ and $\sigma(\hat{Z})/\sigma(Z)$.

Finally, $\varepsilon^2 = \frac{1}{n}||\sigma(\hat{\mathbf{Z}}) - \sigma(\mathbf{Z})||_F^2 \leq \frac{1}{n}||\hat{\mathbf{Z}} - \mathbf{Z}||_F^2 = E(\hat{\mathbf{Z}})$. □

Therefore, to find compression strategy (parameters whose deletion in pruning, quantized value of parameters in quantization) minimizes (3.1) can be translated to find compression strategy minimizes the error function (3.2). Following [23, 24],

the error function can be approximated by functional Taylor series as follows,

$$E(\hat{\mathbf{Z}}^l) - E(\mathbf{Z}^l) = \delta E^l = \left(\frac{\partial E^l}{\partial \mathbf{\Theta}_l}\right)^\top \delta\mathbf{\Theta}_l + \frac{1}{2}\delta\mathbf{\Theta}_l{}^\top \mathbf{H}_l \delta\mathbf{\Theta}_l + O\left(\|\delta\mathbf{\Theta}_l\|^3\right), \tag{3.3}$$

where δ denotes a perturbation of a corresponding variable, $\mathbf{H}_l \equiv \partial^2 E^l / \partial \mathbf{\Theta}_l{}^2$ is the Hessian matrix w.r.t. $\mathbf{\Theta}_l$, and $O(\|\delta\mathbf{\Theta}_l\|^3)$ is the third and all higher order terms. It can be proven that with the error function defined in (3.2), the first (linear) term $\frac{\partial E^l}{\partial \mathbf{\Theta}_l}\Big|_{\mathbf{\Theta}_l = \mathbf{\Theta}_l^*}$ and $O(\|\delta\mathbf{\Theta}_l\|^3)$ are equal to 0, which finally reduce to:

$$\begin{aligned} \min_{\hat{\mathbf{\Theta}}_l} \quad & f(\hat{\mathbf{\Theta}}_l) = \tfrac{1}{2}(\hat{\mathbf{\Theta}}_l - \mathbf{\Theta}_l)^\top \mathbf{H}_l (\hat{\mathbf{\Theta}}_l - \mathbf{\Theta}_l), \\ \text{s.t.} \quad & \hat{\mathbf{\Theta}}_l \in \boldsymbol{\Omega_l} \quad \text{or} \quad |\hat{\mathbf{\Theta}}_l|_0 < \delta, \end{aligned} \tag{3.4}$$

(3.4) illustrates that, in order to recover the full-precision pre-trained model $\boldsymbol{\Omega_l}$ by its compressed counterpart, we need to find the optimal compression strategy represented by $\hat{\mathbf{\Theta}}_l$ by solving the quadratic programing under the quantization or pruning constraints.

3.4 Hessian Approximation

To selectively prune parameters, our approach needs to compute the inverse Hessian matrix at each layer to measure the sensitivities of each parameter of the layer, which is still computationally expensive though tractable. In this section, we present an efficient algorithm that can reduce the size of the Hessian matrix and thus speed up computation on its inverse.

For each layer l, according to the definition of the error function used in Lemma 3.3.1, the first derivative of the error function with respect to $\hat{\mathbf{\Theta}}_l$ is

$$\frac{\partial E^l}{\partial \mathbf{\Theta}_l} = -\frac{1}{n}\sum_{j=1}^{n} \frac{\partial z_j^l}{\partial \mathbf{\Theta}_l}(\hat{\mathbf{z}}_j^l - \mathbf{z}_j^l)$$

, where $\hat{\mathbf{z}}_j^l$ and $\mathbf{z}_j^l$ are the j-th columns of the matrices $\hat{\mathbf{Z}}^l$ and $\mathbf{Z}^l$, respectively, and the Hessian matrix is defined as:

$$\mathbf{H}_l \equiv \frac{\partial^2 E^l}{\partial (\mathbf{\Theta}_l)^2} = \frac{1}{n}\sum_{j=1}^{n}\left(\frac{\partial z_j^l}{\partial \mathbf{\Theta}_l}\left(\frac{\partial z_j^l}{\partial \mathbf{\Theta}_l}\right)^\top - \frac{\partial^2 z_j^l}{\partial (\mathbf{\Theta}_l)^2}(\hat{\mathbf{z}}_j^l - \mathbf{z}_j^l)^\top\right)$$

Note that for most cases $\hat{\mathbf{z}}_j^l$ is quite close to $\mathbf{z}_j^l$, we simply ignore the term containing $\hat{\mathbf{z}}_j^l - \mathbf{z}_j^l$. Even in the late-stage of pruning when this difference is not small, we can still ignore the corresponding term [24]. For layer l that has m_l output units, $\mathbf{z}_j^l = [z_{1j}^l, \ldots, z_{m_l j}^l]$, the Hessian matrix can be calculated via

$$\mathbf{H}_l = \frac{1}{n}\sum_{j=1}^{n}\mathbf{H}_l^j = \frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{m_l}\frac{\partial z_{ij}^l}{\partial \mathbf{\Theta}_l}\left(\frac{\partial z_{ij}^l}{\partial \mathbf{\Theta}_l}\right)^\top, \tag{3.5}$$

where the Hessian matrix for a single instance j at layer l, $\mathbf{H}_l^j$, is a block diagonal square matrix of the size $m_{l-1} \times m_l$. Specifically, the gradient of the first output unit z_{1j}^l w.s.t. $\mathbf{\Theta}_l$ is $\frac{\partial z_{1j}^l}{\partial \mathbf{\Theta}_l} = \left[\frac{\partial z_{1j}^l}{\partial \mathbf{w}_1}, \ldots, \frac{\partial z_{1j}^l}{\partial \mathbf{w}_{m_l}}\right]$, where $\mathbf{w}_i$ is the i-th column of $\mathbf{W}_l$. As z_{1j}^l is the layer output before activation function, its gradient is simply to calculate, and more importantly all output units' gradients are equal to the layer input: $\frac{\partial z_{ij}^l}{\partial \mathbf{w}_k} = \mathbf{y}_j^{l-1}$ if $k = i$, otherwise $\frac{\partial z_{ij}^l}{\partial \mathbf{w}_k} = 0$. An illustrated example is shown in Figure 3.1, where we ignore the scripts j and l for simplicity in presentation.

It can be shown that the block diagonal square matrix $\mathbf{H}_l^j$'s diagonal blocks $\mathbf{H}_{l_{ii}}^j \in \mathbb{R}^{m_{l-1} \times m_{l-1}}$, where $1 \le i \le m_l$, are all equal to $\boldsymbol{\psi}_l^j = \mathbf{y}_j^{l-1}(\mathbf{y}_j^{l-1})^\top$, and the inverse Hessian matrix $\mathbf{H}_l^{-1}$ is also a block diagonal square matrix with its diagonal blocks being $(\frac{1}{n}\sum_{j=1}^{n}\boldsymbol{\psi}_l^j)^{-1}$. In addition, normally $\mathbf{\Psi}^l = \frac{1}{n}\sum_{j=1}^{n}\boldsymbol{\psi}_l^j$ is degenerate and its pseudo-inverse can be calculated recursively via Woodbury matrix identity [24]:

$$(\mathbf{\Psi}_{j+1}^l)^{-1} = (\mathbf{\Psi}_j^l)^{-1} - \frac{(\mathbf{\Psi}_j^l)^{-1}\mathbf{y}_j^{l-1}(\mathbf{y}_j^{l-1})^\top(\mathbf{\Psi}_j^l)^{-1}}{n + (\mathbf{y}_{j+1}^{l-1})^\top(\mathbf{\Psi}_j^l)^{-1}\mathbf{y}_{j+1}^{l-1}},$$

where $\mathbf{\Psi}_t^l = \frac{1}{t}\sum_{j=1}^{t}\boldsymbol{\psi}_l^j$ with $(\mathbf{\Psi}_0^l)^{-1} = \alpha\mathbf{I}$, $\alpha \in [10^4, 10^8]$, and $(\mathbf{\Psi}^l)^{-1} = (\mathbf{\Psi}_n^l)^{-1}$. The size of $\mathbf{\Psi}^l$ is then reduced to m_{l-1}, and the computational complexity of calculating $\mathbf{H}_l^{-1}$ is $O(nm_{l-1}^2)$.

To make the estimated minimal change of the error function optimal in (**??**), the layer-wise Hessian matrices need to be exact. Since the layer-wise Hessian matrices

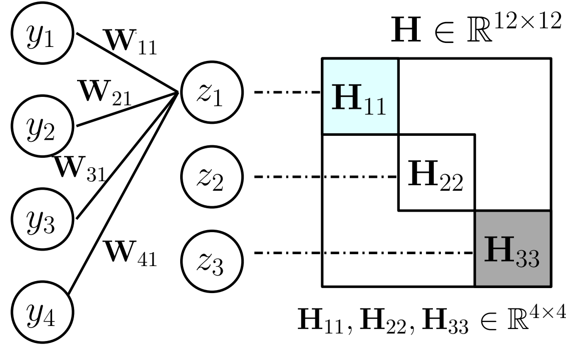


FIGURE 3.1: Illustration of shape of Hessian. For feed-forward neural networks, unit z_1 gets its activation via forward propagation: $\mathbf{z} = \mathbf{W}^\top\mathbf{y}$, where $\mathbf{W} \in \mathbb{R}^{4\times3}$, $\mathbf{y} = [y_1, y_2, y_3, y_4]^\top \in \mathbb{R}^{4\times1}$, and $\mathbf{z} = [z_1, z_2, z_3]^\top \in \mathbb{R}^{3\times1}$. Then the Hessian matrix of z_1 w.r.t. all parameters is denoted by $\mathbf{H}^{[z_1]}$. As illustrated in the figure, $\mathbf{H}^{[z_1]}$'s elements are zero except for those corresponding to $\mathbf{W}_{*1}$ (the 1st column of $\mathbf{W}$), which is denoted by $\mathbf{H}_{11}$. $\mathbf{H}^{[z_2]}$ and $\mathbf{H}^{[z_3]}$ are similar. More importantly, $\mathbf{H}^{-1} = \text{diag}(\mathbf{H}_{11}^{-1}, \mathbf{H}_{22}^{-1}, \mathbf{H}_{33}^{-1})$, and $\mathbf{H}_{11} = \mathbf{H}_{22} = \mathbf{H}_{33}$. As a result, one only needs to compute $\mathbf{H}_{11}^{-1}$ to obtain $\mathbf{H}^{-1}$ which significantly reduces computational complexity.

only depend on the corresponding layer inputs, they are always able to be exact even after several compression operations. The only parameter we need to control is the layer-wise error ε^l. Note that there may be a "compression inflection point" after which layer-wise error would drop dramatically. In practice, user can incrementally increase the size of compressed parameters based on the sensitivity L_q, and make a trade-off between the compression ratio and the performance drop to set a proper tolerable error threshold or compression ratio.

In the following sections, we illustrate method to solve (3.4) under pruning and quantization constraints, respectively.

3.5 Layer-Wise Quantization

Our proposed layer-wise quantization is a cascade algorithm, where the output of a quantized layer is used as the input for quantizing the subsequent layer. Specifically, suppose one has quantized the well-trained network up to the (l–1)-th layer. Then, to quantize the l-th layer, we consider $\hat{\mathbf{Y}}^{l-1} = f(\mathbf{Y}^0; \hat{\mathbf{\Theta}}_{[1,\dots,l-1]})$ as the input, where $f(\mathbf{Y}^0; \hat{\mathbf{\Theta}}_{[1,\dots,l-1]})$ denotes the output of the $(l-1)$-th layer with the first $(l-1)$ layers being quantized, given input $\mathbf{Y}^0$. In the subsequent steps, L-DNQ aims at

quantize the weights from layer l to the last layer L, denoted by $\hat{\mathbf{\Theta}}_{[l,\ldots,L]}$, such that the divergence of the final layer output between quantized network and pre-trained network is minimized:

$$\begin{aligned} \min_{\hat{\mathbf{\Theta}}_{[l,\ldots,L]}} \quad & ||f(\hat{\mathbf{Y}}^{l-1};\hat{\mathbf{\Theta}}_{[l,\ldots,L]}) - f(\mathbf{Y}^{l-1};\bar{\mathbf{\Theta}}_{[l,\ldots,L]})||_F^2, \\ \text{s.t.} \quad & \hat{\mathbf{\Theta}}_{[l,\ldots,L]} \in \mathbf{\Omega}_{[l,\ldots,L]}. \end{aligned} \tag{3.6}$$

Directly solving the above problem is difficult as the inputs to quantized network ($\hat{\mathbf{Y}}^{l-1}$) and the reference network ($\mathbf{Y}^{l-1}$) are different. Here, instead we propose to optimize the upper bound of the objective in (3.6), which is given by the following triangle inequality:

$$\begin{aligned} & ||f(\hat{\mathbf{Y}}^{l-1};\hat{\mathbf{\Theta}}_{[l,\ldots,L]}) - f(\mathbf{Y}^{l-1};\bar{\mathbf{\Theta}}_{[l,\ldots,L]})||_F^2 \\ & \leq \underbrace{||f(\hat{\mathbf{Y}}^{l-1};\hat{\mathbf{\Theta}}_{[l,\ldots,L]}) - f(\hat{\mathbf{Y}}^{l-1};\bar{\mathbf{\Theta}}^{new}_{[l,\ldots,L]})||_F^2}_{\text{Quantization}} \\ & + \underbrace{||f(\hat{\mathbf{Y}}^{l-1};\bar{\mathbf{\Theta}}^{new}_{[l,\ldots,L]}) - f(\mathbf{Y}^{l-1};\bar{\mathbf{\Theta}}_{[l,\ldots,L]})||_F^2}_{\text{Weights Update}}, \end{aligned} \tag{3.7}$$

where $f(\hat{\mathbf{Y}}^{l-1};\hat{\mathbf{\Theta}}_{[l,\ldots,L]})$ presents the final output with quantized weights $\hat{\mathbf{\Theta}}_{[l,\ldots,L]}$ and input $\hat{\mathbf{Y}}^{l-1}$. $\bar{\mathbf{\Theta}}^{new}_{[l,\ldots,L]}$ is introduced as updated full precision weights learned during training. The objective in (3.6) is upper bounded by the summation of the **Quantization** term and the **Weights Update** term. To optimize its upper bound, we present an Alternating Direction Method of Multipliers (ADMM) [50] algorithm to minimize the quantization term for each layer and a back-propagation algorithm to minimize the Weights Update term right after the quantization on each layer. These two procedures are conducted alternatively until all layers are quantized.

3.5.1 Alternative Direction Methods of Multipliers (ADMM)

We give a brief introduction of ADMM in this section. ADMM is a widely-used optimization method. It combines the decomposability of dual ascent and convergence properties of the methods of multipliers. Given the following minimization problem:

$$\min f(\mathbf{x}) + g(\mathbf{z}), \quad \text{s.t.} h(\mathbf{x},\mathbf{z}) = \mathbf{0}.$$

In ADMM, we first reformulate (3.8) with augmented Lagrangian as:

$$L_\rho(\mathbf{x}, \mathbf{y}, \mathbf{z}) = f(\mathbf{x}) + g(\mathbf{z}) + \mathbf{y}^\top h(\mathbf{x}, \mathbf{z}) + \frac{\rho}{2}||h(\mathbf{x}, \mathbf{z})||_2^2, \tag{3.8}$$

where $\mathbf{y}$ is the Lagrangian multipliers. In practice, $\mathbf{y}$ is replaced by $\boldsymbol{\lambda} = (\frac{1}{\rho})\mathbf{y}$ [50] to convert (3.8) to

$$L_\rho(\mathbf{x}, \mathbf{y}, \mathbf{z}) = f(\mathbf{x}) + g(\mathbf{z}) + \frac{\rho}{2}||h(\mathbf{x}, \mathbf{z}) + \boldsymbol{\lambda}||_2^2 - \frac{\rho}{2}||\boldsymbol{\lambda}||^2. \tag{3.9}$$

We then breaks (3.9) into subproblems with respect to $\mathbf{x}, \mathbf{z}, \lambda$, respectively, and solve them iteratively using the following updates:

$$\begin{aligned}
\text{Proximal step} : \mathbf{x}^{k+1} &= \text{argmax}_{\mathbf{x}} L_\rho(\mathbf{x}, \mathbf{z}^k, \boldsymbol{\lambda}^k) && (3.10)\\
\text{Projection step} : \mathbf{z}^{k+1} &= \text{argmax}_{\mathbf{z}} L_\rho(\mathbf{x}^{k+1}, \mathbf{z}, \boldsymbol{\lambda}^k) && (3.11)\\
\text{Dual update Step} : \boldsymbol{\lambda}^{k+1} &= \boldsymbol{\lambda}^k + \mathbf{x}^{k+1} - \mathbf{z}^{k+1} && (3.12)
\end{aligned}$$

3.5.2 Quantization with ADMM

In our problem setting, we apply ADMM to separately optimize continuous variables and discrete variables in (3.4). To be specific, we introduce an auxiliary parameter $\mathbf{G}$ and reformulate (3.4) as follows,

$$\min_{\hat{\boldsymbol{\Theta}}, \mathbf{G}} f(\hat{\boldsymbol{\Theta}}) + I_{\boldsymbol{\Omega}}(\mathbf{G}), \text{ s.t. } \hat{\boldsymbol{\Theta}} = \mathbf{G}, \tag{3.13}$$

where $I_{\boldsymbol{\Omega}}(\mathbf{G})$ is an indicator function that induces great penalty if $\mathbf{G} \notin \boldsymbol{\Omega}$. Here we drop the subscript l for simplification in presentation. By introducing $\mathbf{G}$ and applying the ADMM algorithm, the optimization problem (3.13) can be converted to

$$L_\rho(\hat{\boldsymbol{\Theta}}, \mathbf{G}, \boldsymbol{\lambda}) = f(\hat{\boldsymbol{\Theta}}) + I_{\boldsymbol{\Omega}}(\mathbf{G}) + \frac{\rho}{2}\|\hat{\boldsymbol{\Theta}} - \mathbf{G} + \boldsymbol{\lambda}\|_2^2 - \frac{\rho}{2}\|\boldsymbol{\lambda}\|_2^2. \tag{3.14}$$

(3.14) can be broken into 3 subproblems that are solved alternatively and iteratively by repeating the following steps:

3.5.2.1 Proximal Step

At iteration $k+1$, the proximal step involves the update on $\hat{\mathbf{\Theta}}$ via

$$\hat{\mathbf{\Theta}}^{k+1} = \arg\min_{\hat{\mathbf{\Theta}}} L_\rho(\hat{\mathbf{\Theta}}, \mathbf{G}^k, \boldsymbol{\lambda}^k), \tag{3.15}$$

where

$$L_\rho(\hat{\mathbf{\Theta}}, \mathbf{G}^k, \boldsymbol{\lambda}^k) = f(\hat{\mathbf{\Theta}}) + \frac{\rho}{2}\|\hat{\mathbf{\Theta}} - \mathbf{G}^k + \boldsymbol{\lambda}^k\|_2^2. \tag{3.16}$$

Since $f(\hat{\mathbf{\Theta}})$ is a quadratic function with continuous variable, setting the gradient to $\mathbf{0}$ leads to the optimal solution by solving the following linear equation:

$$\big(\mathbf{H} + \mathrm{diag}(\rho)\big)\hat{\mathbf{\Theta}}^{\boldsymbol{k+1}} = \mathbf{H}\bar{\mathbf{\Theta}}^{new} + \mathrm{diag}(\rho)(\mathbf{G}^k - \boldsymbol{\lambda}^k). \tag{3.17}$$

This is far more efficient than gradient descent [49], and costs only a few seconds even if $\mathbf{H}$ is large. Besides, gradient descent requires fine-tuning a number of hyper-parameters and has unpredictable convergence. These issues are avoided using (3.17). As we can observe in (3.16), ρ acts as an importance weight for the discrete term. A large ρ leads $\hat{\mathbf{\Theta}}^{k+1}$ to approach discrete feasible solution $\mathbf{G}^k - \boldsymbol{\lambda}^k$, while a small ρ guides it to original weights $\bar{\mathbf{\Theta}}^{new}$.

3.5.2.2 Projection Step

In projection step, we optimize $\mathbf{G}$ by solving the following optimization problem:

$$\min_{\mathbf{G}} \|\hat{\mathbf{\Theta}}^{k+1} - \mathbf{G} + \boldsymbol{\lambda}^k\|_2^2, \ \text{ s.t. } \ \mathbf{G} \in \mathbf{\Omega}. \tag{3.18}$$

We define $\mathbf{V}^k = \hat{\mathbf{\Theta}}^{k+1} + \boldsymbol{\lambda}^k$, which is fixed in the projection step. The goal of (3.18) is to find $\mathbf{G}$ that is closest to $\mathbf{V}^k$ and lie in the discrete set $\mathbf{\Omega}$. Take $\mathbf{\Omega} = \{-\alpha, 0, \alpha\}$ as an example. We further denote by $\mathbf{Q} \in \{-1, 0, 1\}$ an intermediate variable such that $\mathbf{G} = g(\alpha, \mathbf{Q}) = \alpha \cdot \mathbf{Q}$. Here $g(\,\cdot\,)$ represents a mapping from integers to discrete real values. Thus, (3.18) can be rewritten as:

$$\min_{\mathbf{G},\alpha} \|\mathbf{V}^k - \alpha \cdot \mathbf{Q}\|_2^2, \ \text{ s.t. } \ \mathbf{Q} \in \{-1, 0, 1\}, \tag{3.19}$$

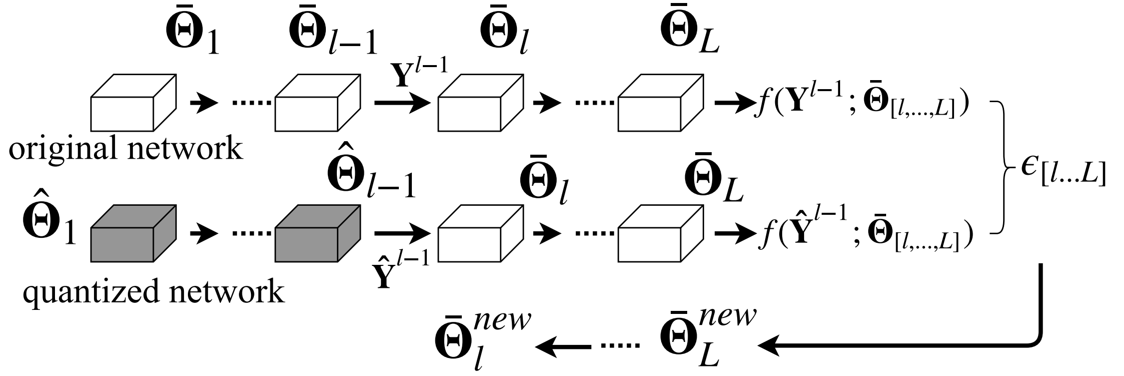


FIGURE 3.2: Weights Update: After layer $l-1$ is quantized from $\bar{\Theta}_{l-1}$ to $\hat{\Theta}_{l-1}$. Input $\mathbf{Y}^{l-1}$ and $\hat{\mathbf{Y}}^{l-1}$ are fed into two networks to generate $f(\mathbf{Y}^{l-1}; \bar{\Theta}_{[l,...,L]})$ and $f(\hat{\mathbf{Y}}^{l-1}; \bar{\Theta}_{[l,...,L]})$, respectively. Squared difference is backpropated to update weights in higher layers: $\bar{\Theta}^{new}_{l,...,L}$.

which consists of two types of variables to be optimized: the scaling factor α that is continuous and the discrete constraints $\mathbf{Q}$. The problem is non-convex and non-smooth. We propose to solve it alternatively: optimize α with $\mathbf{Q}$ fixed and vice versa. Specifically, given a fixed $\mathbf{Q}$, (3.19) is a quadric function w.r.t α, which can be easily solved by $\alpha = \frac{\mathbf{V}^\top \mathbf{Q}}{\mathbf{Q}^\top \mathbf{Q}}$. With α fixed, the optimal $\mathbf{Q}$ is obtained by projecting $\mathbf{V}^k$ to the nearest feasible solution as $\mathbf{Q} = \text{Proj}_{\{-1,0,1\}}\left(\frac{\mathbf{V}^k}{\alpha}\right)$, where $\text{Proj}_{\mathbf{\Omega}}(x)$ denotes the nearest point in the set $\mathbf{\Omega}$ for x. The projection step is efficient to compute. Empirical experiments show that α and $\mathbf{Q}$ could reach a stable range in less than 10 iterations. Finally, we can have $\mathbf{G}^{k+1} = g(\alpha, \mathbf{Q})$.

3.5.2.3 Dual Update Step

After obtaining $\hat{\Theta}^{k+1}$ and $\mathbf{G}^{k+1}$, the dual variable $\boldsymbol{\lambda}$ is updated using the following rule:

$$\boldsymbol{\lambda}^{k+1} = \boldsymbol{\lambda}^k + \hat{\Theta}^{k+1} - \mathbf{G}^{k+1}. \tag{3.20}$$

The ADMM-based layer-wise optimization saves much effort compared to existing gradient-descent-based methods. Computation complexity for the three steps are: $O(n^3)$,$O(n)$,$O(n)$, where n is the number of weights in one kernel. Most importantly, solving the optimization objective requires no label information, which is beneficial when labeled data is not available in real world applications.

3.5.3 Remaining Non-quantized Layers Update

In order to optimize the weights update term in (3.7), we first obtain the network where the previous $l-1$ layers are quantized while the remaining ones are not. Denote the difference or error of the final layer output between the current partially-quantized network and the original network by

$$\epsilon_{[l...L]} = \|f(\hat{\mathbf{Y}}^{l-1}; \bar{\mathbf{\Theta}}^{new}_{[l,...,L]}) - f(\mathbf{Y}^{l-1}; \bar{\mathbf{\Theta}}_{[l,...,L]})\|^2_F$$

Here, we consider $f(\mathbf{Y}^{l-1}; \bar{\mathbf{\Theta}}_{[l,...,L]})$ as the ground-truth, and use back-propagation to learn $\bar{\mathbf{\Theta}}^{new}_{[l,...,L]}$. After we update $\bar{\mathbf{\Theta}}_{[l,...,L]}$ to $\bar{\mathbf{\Theta}}^{new}_{[l,...,L]}$, we can apply the ADMM algorithm introduced in the previous section to quantize layer l by replacing $\bar{\mathbf{\Theta}}_l$ with $\bar{\mathbf{\Theta}}^{new}_l$ in (3.6). Fig.3.2 demonstrates the weights update procedure.

Algorithm 1: Layer-wise Unsupervised Network Quantization

Require: $\bar{\mathbf{\Theta}} = \{\bar{\mathbf{\Theta}}_l\}_{1\leq l\leq L}$, $\boldsymbol{C} = \{\mu_l, \sigma_l, \gamma_l, \beta_l\}_{1\leq l\leq L}$: well-trained network, $\mathbf{X}$: training data

Ensure: $\hat{\mathbf{\Theta}} = \{\hat{\mathbf{\Theta}}_l\}_{1\leq l\leq L}$, $\hat{\boldsymbol{C}} = \{\hat{\mu}_l, \hat{\sigma}_l, \hat{\gamma}_l, \hat{\beta}_l\}_{1\leq l\leq L}$, $\{\boldsymbol{\alpha_l}\}_{1\leq l\leq L}$: quantized network

1: **for** $l = 1, 2, ...L$ **do**

2: Calculate Hessian matrix $\mathbf{H}_l$ of layer l from $\bar{\mathbf{\Theta}}^{new}_l$ and $\mathbf{X}$. (If $l = 1$, $\bar{\mathbf{\Theta}}^{new}_1 = \bar{\mathbf{\Theta}}_1$)

3: Perform ADMM to obtain quantized weight $\hat{\mathbf{\Theta}}_l$ by solving (3.14)

4: Learn $\bar{\mathbf{\Theta}}^{new}_{[l+1,...,L]}$ by minimizing $\epsilon_{[l+1...L]} = \|f(\hat{\mathbf{Y}}^l; \bar{\mathbf{\Theta}}^{new}_{[l+1,...,L]}) - f(\mathbf{Y}^l; \bar{\mathbf{\Theta}}_{[l+1,...,L]})\|^2_F$, and update $\bar{\mathbf{\Theta}}_{[l+1,...,L]} \leftarrow \bar{\mathbf{\Theta}}^{new}_{[l+1,...,L]}$.

5: **end for**

6: Retrain.

3.5.4 L-DNQ in Practice

The overall procedure of L-DNQ is summarized in Algorithm 1. In the algorithm, $\boldsymbol{C}$ denotes the set of parameters for batch normalization, which is also updated in the quantized network. Steps 1-5 corresponds to the quantization procedure using ADMM. After quantization, models can be further boosted by retraining: using label information to fine-tune parameters in batch normalization layers and un-quantized layer. In practice, we adopt approximated Hessian calculation in [25], which only needs to calculate one block matrix of the original Hessian, making

our Hessian computation suitable to handle. About 200 images are sufficient to generate Hessian matrix for each layer.

3.6 Theoretical Analysis

Recall that our goal is to control the consistency of the network's final output $\mathbf{Y}^L$ before and after compression. In the following, we show how the layer-wise errors propagate to the final output layer. We prove that the accumulated error over multiple layers is upper bounded by a constant.

Theorem 3.6.1. Given a quantized network via layer-wise quantization introduced in Section 3.3, each layer has its own layer-wise error ε^l for $1 \leq l \leq L$, then the accumulated error of ultimate network output $\hat{\varepsilon}^L = \frac{1}{\sqrt{n}}\|\hat{\mathbf{Y}}^L - \mathbf{Y}^L\|_F$ obeys:

$$\hat{\varepsilon}^L \leq \sum_{k=1}^{L-1}\left(\prod_{l=k+1}^{L} \mathcal{A}\sqrt{\varepsilon^k}\right) + \sqrt{\varepsilon^L}, \tag{3.21}$$

where $\hat{\mathbf{Y}}^l = \sigma(\hat{\mathbf{W}}_l^\top \hat{\mathbf{Y}}^{l-1})$ for $2 \leq l \leq L$ denotes the "accumulated pruned output" of layer l,The term $\mathcal{A} = \|\hat{\mathbf{\Theta}}_l\|_F$ is upper bounded according to different quantizations.

Consider $\mathbf{\Omega} = \{\pm\alpha, 0\}$ as an example. It can be proven that $\mathcal{A} \leq \alpha^2 \times (m_l \times m_{l-1})$. Moreover, empirical experiments shows that 0 occupies 50%-70% of the quantized parameters, thus $\mathcal{A}$ is much smaller in practice. In summary, Theorem 3.6.1 shows that: 1) Layer-wise error for layer l will be scaled by continued multiplication of parameters' Frobenius Norm over the following layers when it propagates to final output. For quantization, this Frobenius Norm is upper bounded by a constant determined by the quantization intervals. 2) The final error of the ultimate network output is bounded by the weighted sum of layer-wise errors.

Proof. We prove Theorem 3.6.1 via induction. First, for $l = 1$:

$$\hat{\varepsilon}^l = \varepsilon^l = \sqrt{\delta E^l}. \tag{3.22}$$

Then suppose that Theorem 3.6.1 holds up to layer l:

$$\hat{\varepsilon}^l \leq \sum_{h=1}^{l-1}\left(\prod_{k=h+1}^{l} \|\hat{\mathbf{\Theta}}_k\|_F\sqrt{\delta E^h}\right) + \sqrt{\delta E^l}. \tag{3.23}$$

In order to show that (3.23) holds for layer $l+1$ as well, we refer to $\tilde{\mathbf{Y}}^{l+1} = \sigma(\hat{\mathbf{W}}_{l+1}^{\top}\mathbf{Y}^{l})$ as 'layer-wise quantized output', where the input $\mathbf{Y}^{l}$ is fixed as the same as the originally well-trained network. An accumulated input $\hat{\mathbf{Y}}^{l+1} = f(\mathbf{Y}^{0}; \hat{\mathbf{\Theta}}_{[1,\ldots,l]})$, and have the following theorem. □

Theorem 3.6.2. Consider layer $l+1$ in a quantized deep network, the difference between its accumulated quantized output, $\hat{\mathbf{Y}}^{l+1}$, and layer-wise quantized output, $\tilde{\mathbf{Y}}^{l+1}$, is bounded by:

$$\|\tilde{\mathbf{Y}}^{l+1} - \hat{\mathbf{Y}}^{l+1}\|_F^2 \leq \sqrt{n}\|\hat{\mathbf{\Theta}}^{l+1}\|_F^2 \hat{\varepsilon}^{l}. \tag{3.24}$$

By using (3.22), (3.24) and the triangle inequality, we are now able to extend (3.23) to layer $l+1$:

$$\begin{aligned}
\hat{\varepsilon}^{l+1} \quad &= \frac{1}{\sqrt{n}}\|\hat{\mathbf{Y}}^{l+1} - \mathbf{Y}^{l+1}\|_F^2 \\
&\leq \frac{1}{\sqrt{n}}\|\tilde{\mathbf{Y}}^{l+1} - \hat{\mathbf{Y}}^{l+1}\|_F^2 + \frac{1}{\sqrt{n}}\|\tilde{\mathbf{Y}}^{l+1} - \mathbf{Y}^{l+1}\|_F^2 \\
&\leq \sum_{h=1}^{l}\left(\prod_{k=h+1}^{l+1}\|\hat{\mathbf{\Theta}}^{k+1}\|_F^2 \cdot \sqrt{\delta E^{h}}\right) + \sqrt{\delta E^{l+1}}.
\end{aligned}$$

Finally, we prove that (3.23) holds up for all layers, and Theorem 3.6.1 is a special case when $l=L$. We further set $\mathcal{A}=\|\hat{\mathbf{\Theta}}_l\|_F$, which is the Frobenius norm of quantized weights. $\mathcal{A}$ is upper bounded according to different compression. Consider $\mathbf{\Omega}=\{\pm\alpha, 0\}$ as an example. It can be proven that $\mathcal{A}\leq\alpha^2 \times (m_l \times m_{l-1})$.

3.7 Experiment

We verify the effectiveness of Layer-Wise Quantization (L-DNQ) in this section. We conduct comparison experiments with the following baseline approaches: 1) Extremely Low Bit Neural Network (ExNN) [49] 2) Trained Ternary Quantization (TTQ) [17], 3) Incremental Network Quantization (INQ) [8] 4) Loss-Aware weight Ternarized network (LAT) [44]. 5) Compressing Deep Convolution Networks using Vector Quantization (VQ) [66]. 6) Direct Quantization (DQ) which is commonly used in production [48], [67]. 7) Ternary Residual Networks (TRN) [68]. 8) Model compression via distillation and quantization (DistilQuant) [69]. Two benchmark

datasets are used including ImageNet ILSVRC-2012 and CIFAR-10. Regarding deep architectures, we experiment with ResNet-18 [13], AlexNet [3] [11] on ImageNet dataset, and with CIFARNet[4], VGG, WRN[5], ResNet-20, ResNet-32, ResNet-56 on CIFAR-10. All these deep models are well trained at the first place. Due to different deep learning framework, performance of well-trained networks show slight difference.

As L-DNQ pioneers in limited-instance quantization, few works have been published for comparison. VQ conducts compression based on original pre-trained model by using K-means clustering directly; DQ essentially projects full-precision weights into the nearest discrete points; TRN approximated full-precision weights by a combination of quantized weights. After quantization, training instances are used for retraining and fine-tunning. These methods can be considered as baselines for limited-instance quantization. For fair comparison with training-based quantization, we reduce training data to 1% of the original training dataset. Specifically, we re-implement ExNN, TTQ, INQ, LAT and DistilQuant to generate their reported results and then apply the same sets of parameters to produce the results with limited instances. 500 training instances in CIFAR-10 and 12,800 in ImageNet are randomly sampled to simulate the scenario of limited instances. All experiments are conducted 5 times and the average result is reported. Note that all methods use different initial pre-trained models. For fair comparison, we record the percentage of the improvements for these quantized models over their corresponding pre-trained models, which is positive for improvement while negative for degradation after quantization (The higher the better).

L-DNQ adopts the following quantization intervals: $\mathbf{\Omega}_l = \alpha_l \times \{0, \pm 2^0, \pm 2^1, \pm 2^2 ... \pm 2^b\}$ for each layer. Using power of 2 is efficient for inference, because quantized weights can be stored and calculated as integer $(\pm 1, 2, ...)$, with layer output multiplying by α_l to retrieve the actual layer output. To facilitate notation, we use $(2b+3)$-bit to denote the above quantization set, which can be interpreted as the total number of different values in the quantization set, e.g., $\alpha \times \{0, \pm 2^0, \pm 2^1, \pm 2^2\}$ is denoted as 7-bit. In INQ [8], [48] and [68], a different presentation form for bits

[3] AlexNet with batch normalization layers is adopted.

[4] The network architecture is: $(2 \times 128C3) - MP2 - (2 \times 256C3) - MP2 - (2 \times 512C3) - MP2 - (2 \times 1024FC) - 10SVM$, where $C3$ is a 3×3 ReLU convolution layer, $MP2$ is a 2×2 max-pooling layer.

[5] Widen factor:20, depth:28, dropout rate:0.3

is used. Here we convert the number of bits using our notation for fair comparison.

Network	Method	bits	Imp*(%)	FP**
ResNet20	TTQ	3	-77.25	91.77
	INQ	15	-48.48	90.02
	ExNN	3	-11.15	91.5
	VQ	3	-11.27	
	DQ	3	-19.92	
	L-DNQ	3	**-4.30**	
ResNet32	TTQ	3	-79.99	92.33
	INQ	15	-48.02	86.83
	ExNN	3	-12.03	92.13
	VQ	3	-8.98	
	DQ	3	-21.07	
	L-DNQ	3	**-3.66**	
ResNet56	TTQ	3	-80.64	93.20
	INQ	15	-15.84	93.40
	ExNN	3	-12.15	92.66
	VQ	3	-11.43	
	DQ	3	-18.67	
	L-DNQ	3	**-3.49**	
CIFARNet	LAT	3	-11.62	89.62
	VQ	3	-11.83	92.27
	DQ	3	-21.72	
	L-DNQ	3	**-1.96**	
VGG***	DistilQuant	3	-53.9	90.77
	L-DNQ	3	**-1.47**	89.42
WRN	DistilQuant	3	-6.57	92.25
	L-DNQ	3	**-2.22**	91.43

TABLE 3.1: Comparison on CIFAR-10. All methods use 1% (500 images) of training instances. * indicates improvement. ** represents **F**ull **P**recision (pre-trained model) Accuracy. *** is a VGG-like model adopted by DistilQuant.

3.7.1 Overall Experimental Results and Analysis

On CIFAR-10, we compare our method with ExNN, TTQ, INQ, LAT, VQ, DQ, DistilQuant in Table 3.1. ExNN incorporated ADMM quantization in training. TTQ ternarized the full-precision model into 3 different values: $\{\beta_l, 0, \alpha_l\}$ in layer l. INQ converted weights into either power of two or zero. LAT quantized weights into 3 bits. DistilQuant utilize distillation to train quantized model. We reproduce

Network	Method	bits	Improvement(%)	FP Accuracy
ResNet18	TTQ	3	-69.48/-88.49	69.6/89.2
	INQ	15	-61.27/-64.22	68.27/88.69
	ExNN	3	-43.53/-37.82	69.76/89.02
	VQ	3	-35.69/-29.08	
	DQ	3	-61.22/-65.64	
	L-DNQ	3	**-16.43/-10.67**	
	DQ	8	-56.78/-58.92	
		16	-13.81/-8.68	
		32	-2.82/-1.51	
	L-DNQ	9	**-2.73/-0.90**	
ResNet34	DistilQuant	3	-32.03/24.3	56.55/79.09
	L-DNQ	3	**-29.31/18.37**	
AlexNet[1]	TTQ	3	-56.18/-78.26	57.2/80.3
	DistilQuant		-55.07/-72.79	56.55/79.09
	ExNN		-28.34/-26.06	58.34/80.80
	VQ		-35.95/-36.06	
	DQ		-57.04/-76.49	
	L-DNQ		**-17.78/-14.00**	
AlexNet	INQ	15	-42.82/-46.83	57.24/79.80
	TRN	12.5	≈ -1	N.A.
	ExNN	9	-13.47/-10.33	58.34/80.80
	VQ		-5.94/-4.22	
	DQ		-7.84/-5.72	
	L-DNQ		**-0.88/-0.57**	

TABLE 3.2: Comparison on ImageNet. All methods use 1% (12,800 images) training instances. AlexNet[1]: in TTQ, the weights of the first and final layer remain full precision. L-DNQ, ExNN, DQ, VQ, DistilQuant are under the same setting.

Network	3-bit	5-bit	7-bit	9-bit	Full Precision
ResNet18	-16.43/-10.67	-8.61/-4.92	-4.67/-2.40	-2.73/-0.90	69.76/89.02
ResNet34	-29.31/-18.37	-11.22/-6.10	-3.69/-1.88	-2.10/-0.87	73.30/91.42
ResNet50	-19.66/-11.32	-7.55/-4.10	-2.79/-1.24	-1.71/-0.53	76.15/92.87

TABLE 3.3: Overall experimental results of L-DNQ in various models using 1% of ImageNet dataset. Column 2-5 represent the quantization improvement (Top1/Top5) under different bits quantization. The last column represents full precision accuracy.

TTQ, INQ experiments on ResNet-20, ResNet-32, ResNet-56, LAT experiments on CIFARNet and DistilQuant on VGG, WRN with the source code released by [17] , [8] , [44] and [69], respectively. ExNN is reimplemented by us since its source code is not released. After VQ and DQ generate quantized weights, we use limited

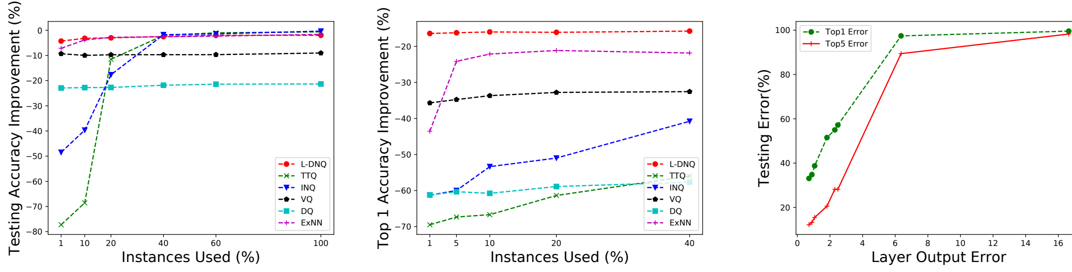


FIGURE 3.3: Fig.(a)/(b): Performance among L-DNQ, ExNN, TTQ, INQ, VQ, DQ using ResNet20/18 in CIFAR10/ImageNet with increasing instances. X-axis presents portion of training data used, Y-axis represents performance improvement after quantization (Higher the better). Fig.(c): Performance under different layer output error. X-axis presents final layer output error, Y-axis represents testing error after quantization (the lower the better).

Dataset	Network	Method	kbits	TopK	Improvement	Full Precision (%)
CIFAR10	ResNet20	L-DNQ^{-}	3	1	-13.66	91.5
		L-DNQ	3	1	-4.3	
ImageNet	AlexNet	L-DNQ^{-}	3	1/5	-56.80/-79.30	58.34/80.80
		L-DNQ	3	1/5	-17.78/-14.00	

TABLE 3.4: Comparison between L-DNQ and L-DNQ^{-}.

instances for retraining. As Table 3.1 shows, all training-based methods (ExNN, TTQ, INQ, DistilQuant) experienced great degradation using limited instances. VQ and DQ performed slightly better, but still showed considerable drop. As a contrast, L-DNQ can preserve original performance, even under a dramatic reduction of training data down to 1%.

On ImageNet, which is a much larger dataset than CIFAR-10, we compare our model with ExNN, INQ, TTQ, VQ, DQ, DistilQuant using ResNet18/34 and AlexNet, with the results showing in Table 3.2. Similar to CIFAR-10, different methods use different pre-trained models and the improvements are shown. When only 1% instances are used, L-DNQ outperforms other methods by a large margin. Training-based methods suffer under-fitting problem in limited-instance scenario. Direct quantization-based methods fail to capture data distribution for better compression. For TRN, since it doesn't release the source code, we compare with its reported result using AlexNet. As Table 3.2 shows, L-DNQ achieves much better performance compared with other methods using 3 bits. In practice, DQ uses more bits in quantization, hence we also compare our model using 9 bits with DQ using more bits from 8 to 32 in ResNet18 and other baselines with more bits in AlexNet.

Clearly, the baseline models retrieve better performances as the number of bits increases, but still hardly achieve comparable performances as L-DNQ, which almost approximates the full-precision model with 9 bits.

We also compare L-DNQ, ExNN, TTQ, INQ, VQ and DQ in ResNet20/18 using increasing number of training instances on CIFAR-10 and ImageNet. As Fig 4.3(a) and 4.3(b) show, L-DNQ maintains high performances even when only a few training instances are used, while training-based methods degrade severely if data is scarce.

We conjecture that L-DNQ minimizes the divergence from original network under quantized constraints instead of regenerating a new network given label supervisions, which enables L-DNQ to utilize much fewer data to preserve the original performance.

3.7.2 Analysis of L-DNQ

3.7.2.1 Bits' Effect Towards Performance

We conduct experiments on various quantization levels on ImageNet using 1% instances. As Table 3.3 shows, the prediction accuracy increases as the number of bits increases. It can be observed that L-DNQ approximately recovers the original accuracy under 9-bit quantization.

3.7.2.2 Layer Output Error V.S. Performance

L-DNQ aims at minimizing the final layer output error between original and quantized networks. It is interesting to explore the relationship between layer output error and performance of quantized network. We measure the performance of quantized ResNet18 network on ImageNet dataset under different final output errors. As Fig. 4.3(c) shows, testing error increases as the final output error increases. Empirically, testing error is positively correlated with final output error. Hence, by minimizing the final output error, L-DNQ is capable of attaining good performance.

3.7.2.3 Effectiveness of Weights Update

To verify the effectiveness of weights update, we generate another baseline called L-DNQ$^-$ by removing weights update in L-DNQ, which can be considered as a reduction of the proposed L-DNQ. Comparison results between L-DNQ and L-DNQ$^-$ are shown in Table 3.4: weights update in L-DNQ brings significant performance gain by narrowing the divergence between quantized network and the un-quantized one.

3.7.2.4 Order of Quantization Process

We conduct an ablation study for the quantization order in cascade algorithm. Specially, we adopt a up-bottom (from high to low layers) cascade quantization scheme: we first quantize output (last) layer, then the layer before, all the way down to first layer. In ResNet20, CIFAR10, 3-bit L-DNQ achieves -50.6% improvement, which is far worse than bottom-up quantization (-4.3%).

3.8 Conclusion

We formulate deep neural network quantization in a layer-wise optimization framework. In each layer, we minimize the layer output variation before and after compression. Specially, we solve the optimization problem under quantization constraint by ADMM. Only a small portion of data is sufficient to finish the quantization procedure.

Chapter 4

Cooperative Pruning

4.1 Introduction

Previous [1] chapter (L-DNQ) targets at model quantization using scarce data. However, it requires a full-precision pre-trained as reference for compression approximation. The pre-trained model still relies on training processes with a large amount of data. Besides, L-DNQ's focus lies in model quantization.

For pruning, most prevailing methods relied on training processes with a large amount of data and a well-trained model. Specifically, given a well pre-trained deep neural network as the initial parameters, which is usually trained in a cloud environment, most methods conduct the pruning process in a supervised learning manner with sufficient training data to minimize the error between the outputs of the pruned network and the ground-truth labels. However, practical scenarios pose more strict challenges: a large amount of labeled data is hard to obtain due to costly annotation effort, making it hard to train a good model, not even to mention pruning. Without the access of sufficient data and well-trained full-precision model, existing pruning methods are no more effective. In these situations, a desirable solution is to transfer knowledge from a well pre-trained model in a resource-rich source domain to the target domain with limited data for pruning. To this end, we develop a novel pruning method under the cross-domain setting: given a pre-trained model and data from the source domain and only limited data in the target

[1] The work in this chapter has been published in 28th International Joint Conference on Artificial Intelligence (IJCAI 2019).

domain, the model could utilize knowledge from the source domain to assist pruning in the target domain. Although transfer learning has been applied in many learning problems, there is little study on cross-domain model compression. In the sequel, we name our proposed model as **Co**pertivative **Prun**ing (Co-Prune).

Specifically, we employ a dynamic and cooperative pruning strategy to prune both source and target network simultaneously. We employ a mask matrix consisting of '0's and '1's with the same shape as each layer's parameters to indicate if the parameter at each position should be pruned $\{0\}$ or kept $\{1\}$. The entries in the mask are determined by the absolute values of their corresponding parameters in the full-precision model. In each iteration, parameters are updated by gradient descent: for the parameters pruned by the mask, their gradients are still recorded according to the gradients before penetrating through the mask. The updated parameters lead to a modified mask, which in turn affects the computation in the next iteration. To transfer knowledge, the mask of the target network for each layer is jointly determined by its own parameters and the ones from the source network via a weighted scalar parameter α, named as "transfer factor". Transfer factor α determines how much knowledge is leveraged from the source domain to the target domain. We dynamically set α in the training process to reflect dynamic knowledge transfer. Specifically, α is set larger at the beginning of training because target domain is assumed to rely heavily on source supervision to achieve a good initial state. As training proceeds, we gradually decrease α to allow the target network to learn from itself. We show in the experiment that the dynamic transfer factor could achieve a smooth knowledge transfer for better prediction performance.

Compared with most existing methods, Co-Prune is capable of conducting model pruning in the domain with only limited training data and without a well-trained initial state. Our contributions are listed in the following: 1) Our method can deal with neural network pruning under the scenario of limited training data. 2) We pioneer the idea of knowledge transfer from a resource-rich domain to conduct neural network pruning in target domains. 3) Extensive experiments are conducted to verify the effectiveness of our proposed method compared with several state-of-the-art approaches in the setting of limited training data.

4.2 Cooperative Pruning

Given a pre-trained neural network model in terms of $\mathbf{W}_s$, a training dataset D_s with n_s data from the source domain, and a limited training dataset D_t with n_t data from the target domain, we aim to produce a pruned model $\mathbf{W}'_t = \mathbf{W}_t\mathbf{M}_t$ for the target domain, where $\mathbf{W}_t$ denotes the target-domain full-precision network, which is unknown at the beginning, $\mathbf{W}'_t$ parameterizes the target-domain network after pruning, and $\mathbf{M}_t$ is the mask matrix. Normally, the target dataset is less than 1/10 of the source dataset which is insufficient to train a good deep neural network model. For each layer of a model, a mask matrix ($\mathbf{M}$) of the same shape as model parameters is introduced to indicate whether a parameter is pruned or not (denoted by 0 or 1). We preserve two networks and their mask matrices for the source domain ($\mathbf{W}_s$, $\mathbf{M}_s$) and the target domain ($\mathbf{W}_t$, $\mathbf{M}_t$), respectively. $\mathbf{W}_t$ is initialized from the fine-tuned model in the target domain trained from $\mathbf{W}_s$. The marks $\mathbf{M}_s$ and $\mathbf{M}_t$ are initialized with all 1's. During training, both $\mathbf{W}_s$ and $\mathbf{W}_t$ are updated through gradient descent, while $\mathbf{M}_s$ and $\mathbf{M}_t$ are directly computed accordingly. We update $\mathbf{W}_s$ during training to dynamically affect $\mathbf{M}_t$. A Compression Ratio (CR) is set for each layer to control the percentage of remaining parameters after pruning.

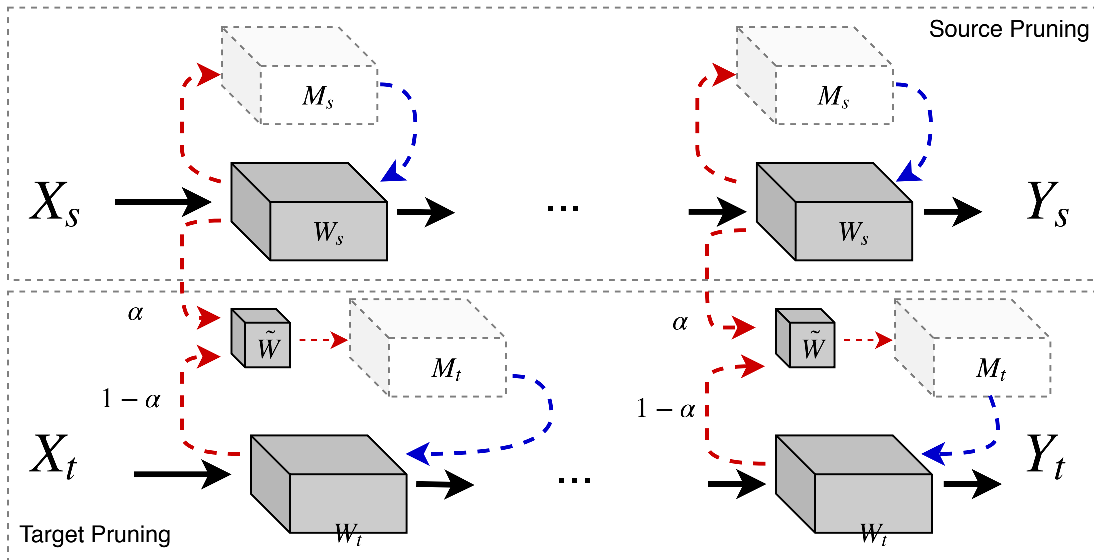


FIGURE 4.1: Sketch architecture for Co-Prune: source / target task is trained and pruned cooperatively. Red dash lines indicate where the contribution to masks comes from. Blue dash lines point out where the masks are imposed to. Pruning mask of source task is determined by its own parameters. By contrast, both parameters from source and target model contribute to pruning mask for target.

Fig.4.1 illustrates a sketch architecture for Co-Prune: Pruning is conducted by imposing layer-wise pruning mask for model's parameters $\mathbf{W}_s$ ($\mathbf{W}_t$). During training, the mask $\mathbf{M}_s$ ($\mathbf{M}_t$) is updated in each iteration according to current value of parameters, which is explained in detail in Sec.4.2.1. Pruning mask $\mathbf{M}_s$ for the source domain is independently generated from its own parameters, as in (4.3). For the target domain, the pruning mask $\mathbf{M}_t$ is obtained from the parameters of both source and target networks via (4.5), where knowledge from the source domain is selectively transferred to the target domain.

4.2.1 Mask Generation

It is rather difficult to train a target model from scratch using only limited target data. On the other hand, directly applying and fine-tuning the pre-trained model from a source domain for target predictions is not feasible due to large domain shift. Therefore, we propose to leverage information from the source model to assist target predictions through cooperative mask generation. We regard the mask as the bridge connecting source/target domain and transferring knowledge between them. Specially, given parameters $\mathbf{W}$, masking function ($\mathcal{M}$) and its leading mask $\mathbf{M}$ is defined as:

$$\begin{aligned} \mathbf{M}_{i,j} &= \mathcal{M}(\mathbf{W}_{i,j}) \\ &= \begin{cases} 0, & \text{if } \operatorname{Imp}(\mathbf{W}_{i,j}) < \sigma_{\mathrm{Imp}}(\mathrm{CR}), \\ 1, & \text{if } \operatorname{Imp}(\mathbf{W}_{i,j}) \geq \sigma_{\mathrm{Imp}}(\mathrm{CR}), \end{cases} \end{aligned} \tag{4.1}$$

where $\operatorname{Imp}(\cdot)$ is a function measuring parameters' importance. For example, [23] defined it as: $\operatorname{Imp}(w) = \frac{\partial^2 L}{\partial w^2} \times w^2$, involving hessian of each parameter and its square. In Co-Prune, to efficiently calculate parameters' importance, it is defined as: $\operatorname{Imp}(\mathbf{W}_{i,j}) = |\mathbf{W}_{i,j}|$. $\sigma_{\mathrm{Imp}}(\mathrm{CR})$ represents a mapping from parameters (with n elements) to pruning threshold given $\operatorname{Imp}(\cdot)$ and CR set for this layer:

$$\sigma_{\mathrm{Imp}}(\mathrm{CR}) = \operatorname{Increasing}(\operatorname{Imp}(\mathbf{W}), n \times \mathrm{CR}). \tag{4.2}$$

$\operatorname{Increasing}(\mathbf{W}, i)$ takes out the i-th element of an increasing ranking of elements from $\operatorname{Imp}(\mathbf{W})$.

The mask for each layer of **source** parameters $\mathbf{W}_s$ is generated using (4.1) independently:

$$\mathbf{M}_s = \mathcal{M}(\mathbf{W}_s). \tag{4.3}$$

The mask matrix embeds information about parameters' importance in each domain. We assign a scalar variable α, named as "transfer factor" to control the knowledge flow by a fusion function $f(\mathbf{W}_s, \mathbf{W}_t, \alpha)$ for generating the mask in the **target** domain. In Co-Prune, transfer factor produces a weighted sum of both source and target model parameters as the following:

$$\tilde{\mathbf{W}} = f(\mathbf{W}_s, \mathbf{W}_t, \alpha) = \alpha \times \mathbf{W}_s + (1 - \alpha) \times \mathbf{W}_t. \tag{4.4}$$

Then $\tilde{\mathbf{W}}$ is utilized to compute the mask matrix for target model using (4.1) as

$$\mathbf{M}_t = \mathcal{M}(\tilde{\mathbf{W}}). \tag{4.5}$$

The pruned positions in $\mathbf{M}_t$ are set as 0, while the rest are set as 1. During this process, knowledge from the source model is transferred to the target, regulated through α. The mask is imposed into each layer of the network for inference and calculation of loss, which is denoted by:

$$\text{Loss}(\mathbf{W}_d \odot \mathbf{M}_d), \tag{4.6}$$

with $d \in \{s, t\}$, where $\odot$ represents element-wise multiplication between $\mathbf{W}_d$ and $\mathbf{M}_d$.

4.2.2 Adaptive Domain Adaptation via Transfer Factor α

Cooperative pruning can be regarded as source model (teacher) transferring knowledge to target model (student). Because of limited data in the target domain, learning at the beginning requires more guidance from the source domain by exploring the commonalities across different domains. As training proceeds, it is assumed that the target model is capable of learning more domain-dependent information from itself. This dynamic mechanism is implemented by adjusting α: assigning a large α at the beginning, and reducing it gradually along with the training process.

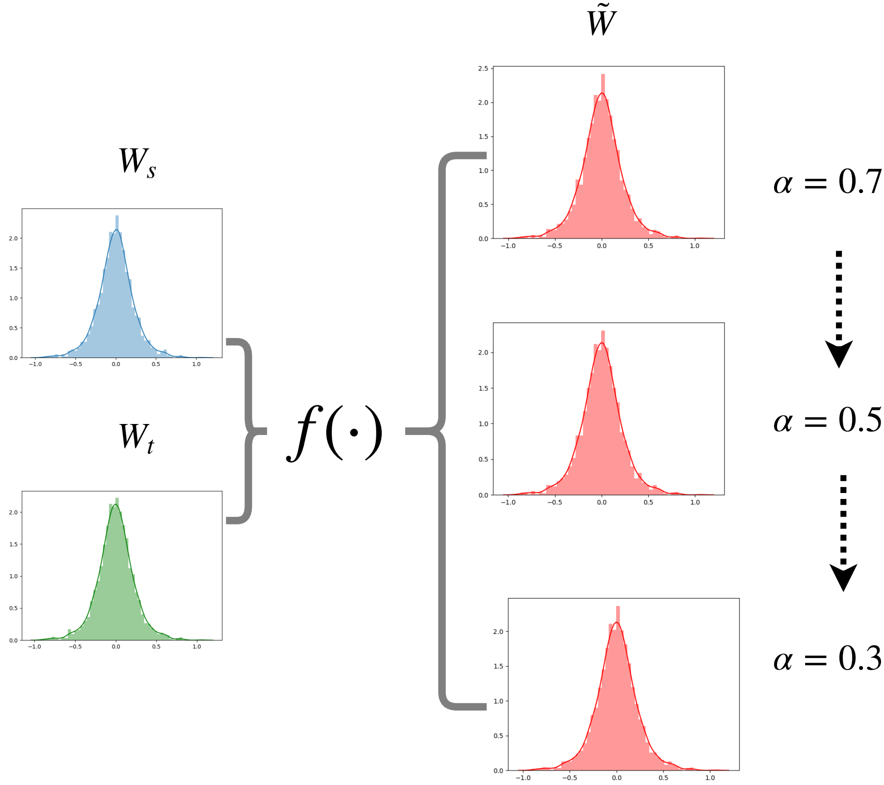


FIGURE 4.2: Adjust α during cooperative pruning. Given updated parameters from the source model $\mathbf{W}_s$ and the target model $\mathbf{W}_t$, transfer factor α in fusing function f is imposed to generated a combined $\tilde{\mathbf{W}}$ for target mask

Specially, in Co-Prune a non-increasing discrete function for α_k w.r.t iteration of training optimum k is defined:

$$\alpha_k = \alpha_0 - k \times \frac{\alpha_0 - \alpha_{\min}}{\beta}, \quad \text{s.t.} \quad \alpha_{\min} \leq \alpha_k \leq \alpha_0. \tag{4.7}$$

α_0 is the initial factor, $\alpha_{\min}$ is the minimal value that α_k can reach. When training comes at optimum using current α_k, it will be updated by (4.7) to obtain α_{k+1}. β represents sensitivity of $\{\alpha_k\}$. In Co-Prune, $\alpha_0 = 0.7$, $\alpha_{\min} = 0.3$, $\beta = 3$ for trade-off between computational time and accuracy. Further studies on how to choose α_0, $\alpha_{\min}$ and β are experimented in Sec4.3.2.

As shown in Fig. 4.2 for an example of adaptive α: parameters from source and target are weighted summed by a gradually changing α, leading to combined parameters with the importance of each domain weighed by α.

4.2.3 Training-based Pruning

Traditional pruning is conducted in a "Prune-Retrain" fashion: Firstly, prune less important parameters based on the pre-trained model by corresponding $\text{Imp}(\cdot)$, then retrain the remaining parameters while fixing the pruned ones. Such one-time pruning strategy fails to retrieve pruned parameters that appear to be essential in a latter training stage. This could be the case in cross-domain pruning: some weights need to be retrieved in the source model to be adapted to the target model. Compared to direct pruning, imposing a mask matrix over parameters to perform pruning is a better way to keep original parameters.

However, when conducting backward update, the gradients of those parameters with 0 mask values are 0, making it similar to one-time pruning. To overcome this limitation and enable model adaptation during cross-domain training, we borrow the idea of "straight-through estimator" (STE) from neural network quantization: For non-differentiable quantization function $\mathcal{Q}$ imposed on value r to obtain $q = \mathcal{Q}(r)$, the gradient of loss w.r.t q is attained as $\frac{\partial C}{\partial q} = g_q$. The straight-through estimator of $\frac{\partial C}{\partial r} = g_r$ is simply: $g_r = g_q 1_{|r| \leq 1}$.

Specially, during inference, consider that pruning function is conducted on $\mathbf{W}$, leading to $\hat{\mathbf{W}} = \mathbf{M} \odot \mathbf{W} = \mathcal{M}(\mathbf{W}) \odot \mathbf{W}$. where we omit subscripts $\{s, t\}$ for the ease of illustration. During inference, loss of neural network is:

$$\text{Forward} : L = \text{Loss}(\hat{\mathbf{W}}). \tag{4.8}$$

For backward computation, assume that gradient of L w.r.t $\hat{\mathbf{W}}$ is attained as $g_{\hat{\mathbf{W}}} = \frac{\partial L}{\partial \hat{\mathbf{W}}}$. In Co-Prune, estimator of $g_{\mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}}$ is obtained by

$$\text{Backward} : g_{\mathbf{W}} = g_{\hat{\mathbf{W}}}. \tag{4.9}$$

During training-based pruning, each layer consists of a Compression Ratio (CR) that is set heuristically, representing the proportion of the remaining parameters. After parameter updates using (4.9) in each iteration, Parameters below the threshold attained by (4.2) are pruned by setting the corresponding positions in the mask matrix as 0 for this iteration, as indicated in (4.1). In the subsequent training iterations, pruned parameters can be recovered according to their persistent update as

(4.9). This training-based pruning enables 1) parameter recovery to optimize final performance and 2) model adaptation when applying the model on target data.

Algorithm 2: Co-Prune

Require: Source training data $D_s = \{\mathbf{X}_s, Y_s\}^{n_s}$, target training data $D_t = \{\mathbf{X}_t, Y_t\}^{n_t}$, source pre-trained model $\mathbf{W}_s$

Ensure: Final target model $\mathbf{W}_t$ and mask $\mathbf{M}_t$.

Initialize source model using $\mathbf{W}_s$, target model as target fine-tuned model based on $\mathbf{W}_s$, source-specific mask $\mathbf{M}_s$, target-specific mask $\mathbf{M}_t$ with all 1s. $\alpha = \alpha_0$

while $\alpha \geq \alpha_{\min}$ **do**
  while Not Optimum **do**
    Train $\mathbf{W}_s$ using D_s and $\mathbf{M}_s$, $\mathbf{W}_t$ using D_t and $\mathbf{M}_t$ by (4.8) and (4.9).
    Update mask $\mathbf{M}_s$ by (4.3).
    Update mask $\mathbf{M}_t$ by (4.5).
  end while
  α is tuned by (4.7).
end while

4.2.4 Algorithm and Implementation Details

Alg.2 illustrates the whole process of Co-Prune: Source and target models are trained using their corresponding data and current masks. Masks are updated accordingly with the updates of model parameters by (4.3) and (4.5). In practice, Adam [70] with initial learning rate 10^{-3} is used for Co-Prune and all retraining processes. Learning rate will be divided by 10 when training loss increases for 3 consecutive epochs. Training to optimum is considered as learning rate becomes smaller than 10^{-6}.

4.3 Experiment

To simulate practical scenario, two datasets with abundant source data and limited target data are utilized.

CIFAR9-STL9 is a modified version of combined CIFAR10 and STL10 dataset. CIFAR10 is a classical 10-class dataset with 50000 32×32-pixel training data. Inspired by this dataset, STL10 is designed with 10 similar classes that consists of very limited labeled samples: 5000 96×96-pixel training data. We exclude one class from CIFAR10/STL10 that is not shared in both datasets and name the

resulting data as CIFAR9/STL9. After exclusion, we treat CIFAR9 as the source domain and STL9 as the target domain.

ImageCLEF is a 4-domain image dataset. It extracts 600 images of 12 classes from ImageNet [71], Caltech-256 [72], PASCAL [73] and Bing, respectively. We regard ImageNet dataset as the source domain. Specifically, an ImageNet pre-trained model is downloaded online. Then the last layer is replaced with a 12-output fully-connected layer in order to be used for 12-class classification problem. We use the 600-image ImageNet data to finetune the whole model, which is utilized as the source pre-trained model. The rest three datasets in ImageCLEF are regarded as target domains.

To verify Co-Prune's generalization ability, we experiment with CIFAR-Net [74] in CIFAR9-STL9, ResNet18 [13] in ImageCLEF. For CIFAR-Net, CR of each layer is set manually. For ResNet18, a unified CR is set for every layer.

4.3.1 Comparison Experiment

We conduct comparison experiments using the following baseline pruning methods: 1) LWC [1], 2) OBD [23], 3) DNS [21], 4) L-OBS [25]. LWC measured parameters' importance using their absolute value. OBD used hessian and their squared value to indicate the parameters importance. L-OBS formulated pruning for each layer as an optimization problem, which is solved to attain parameters' sensitivity. Given a pre-trained model, these three methods performed one-time pruning according to their measurement of weights' importance. A retraining process is then conducted while fixing the pruned weights. DNS performed pruning in a dynamic way: after parameters updates, their importance will be re-calculated by their corresponding absolute values, then parameters are pruned under certain probability related to the importance. Since these methods are not designed specifically for cross-domain pruning, to make fair comparison, all methods firstly retrain the target model based on provided pre-trained model in the source domain as an initial state. Then target data is utilized for the pruning process.

Besides, we use a classical domain adaptation algorithms with modification for pruning as baselines from perspective of domain adaptation: DDC [75]. DDC trained source and target network with additional minimization of Maximum Mean Discrepancy (MMD) distance of intermediate features. To revise DDC for pruning

with supervised domain adaptation, we use DNS networks as target network, with additional target label loss to assist training. We name it as DDC-DNS.

Since data is quite limited in ImageCLEF, we divide each domain into 80% for training and 20% for testing (with class balance). Every experiment are conducted 10 times with random data partition. For each partition, performance improvement with variance of each method is recorded.

CR (%)	Method	FP Acc (%)	Prune Acc (%)
10.4	LWC	68.03	66.26
	OBD		65.78
	DNS		66.25
	L-OBS		66.01
	DDC-DNS		66.49
	Co-Prune		**66.99**
1.3	LWC		57.47
	OBD		50.82
	DNS		58.89
	L-OBS		56.00
	DDC-DNS		56.79
	Co-Prune		**60.5**
	One-Time Co-Prune		55.36
	Distillation		53.16
0.9	LWC		49.53
	OBD		48.34
	DNS		53.32
	L-OBS		53.04
	DDC-DNS		47.86
	Co-Prune		**56.21**

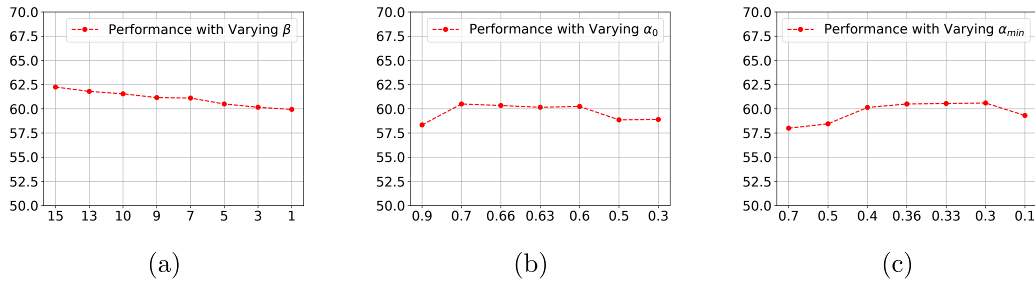
TABLE 4.1: Overall results of CIFAR9-STL9 using CIFAR-Net

We compare Co-Prune with LWC, OBD, DNS, L-OBS, DDC-DNS in CIFAR9-STL9 using CIFAR-Net under different CRs. Table 4.1 illustrates performance of the pruned model under different methods: Co-Prune outperforms all baseline methods in all selected CR. Especially, as CR decreases, Co-Prune shows more obvious advantage over other methods.

To validate the effect of training-based pruning used in Co-Prune, which differentiates from one-time pruning in that: pruning strategy is changing during training.

We conduct an experiment of Co-Prune using one-time pruning (named as "One-Time Co-Prune") in Table 4.1, specially, mask is updated only at the beginning of training. The performance drops significantly from 60.5% to 55.36%. This demonstrates that training-based pruning contributes to Co-Prune.

Comparison experiments with non-pruning compression is conducted using distillation. We construct a slim CIFARNet which contains 10.4% parameters of the original one. This slim network is trained by ground-truth and distilled from original target-retrained CIFARNet (who is retrained using target data based on source pre-trained network). As Table 4.1 shows it reaches 53.16%, worse than Co-Prune.



(a) (b) (c)

FIGURE 4.3: (a) Performance of Co-Prune under various β with fixed $\alpha_0 = 0.7$, $\alpha_{\min} = 0.3$; (b) Performance of Co-Prune under various α_0 with fixed $\beta = 3$, $\alpha_{\min} = 0.3$; (c) Performance of Co-Prune under various $\alpha_{\min}$ with fixed $\alpha_0 = 0.7$, $\beta = 3$. Y-axis represents the best performance under corresponding setting, X axis represents β, α_0, $\alpha_{\min}$, respectively

Similarly, we conduct 3 cross-domain pruning from ImageNet to PASCAL, Caltech256, Bing, respectively. In Table 4.2, we compare the performances of different methods in various target domains, using ResNet18 with 4% CR in each layer. For all the cross-domain directions, Co-Prune shows the best average performance over other methods.

4.3.2 Effect of Transfer Factor α

To examine α's properties in Co-Prune, CIFAR9-STL9 in CIFAR-Net under CR at 1.3 is used as an example.

Direction	Method	FP Acc (%)	Improve Acc (%)
I→P	LWC	60.917±3.809	-16.167±3.617
	OBD		-28.333±7.188
	DNS		-11.417±3.556
	L-OBS		-8.317±3.435
	DDC-DNS		-12.917±4.075
	Co-Prune		**-6.583±3.525**
I→C	LWC	87.417±2.898	-6.833±3.851
	OBD		-30.500±14.664
	DNS		-5.583±2.912
	L-OBS		-5.018±3.214
	DDC-DNS		-6.167±3.009
	Co-Prune		**-3.083±2.936**
I→B	LWC	49.667±3.232	-11.667±2.609
	OBD		-18.250±2.661
	DNS		-7.667±2.577
	L-OBS		-8.973±4.735
	DDC-DNS		-6.500±5.711
	Co-Prune		**-5.648±3.659**

TABLE 4.2: Overall results of ImageNet→PASCAL, ImageNet→Caltech256, ImageNet→Bing using ImageNet pre-trained ResNet18. CR is 4% for each layer.

4.3.2.1 Variation of Co-Prune

When α=0, Co-Prune reduces to DNS as it only replies on target data to generate mask and conduct training. For $\alpha = 1$, Co-Prune depends on source model to generate mask for target pruning. Total dependence on source or target does harm in finding optimal pruning strategy. As Table 4.3 shows, Co-Prune with adaptive α outperforms other variations, showing that proper knowledge transfer from source assists in improving pruning on target model.

4.3.2.2 Effect of Sensitivity β

(4.7) determines the sensitivity of transfer factor. We experiments with various β by fixing $\alpha_0 = 0.7$, $\alpha_{\min} = 0.3$ and record its best performance in Fig.4.3(a): Performance of Co-Prune under various β shows changes with mean and variance as

α	FP Acc	Prune Acc
0 (DNS)	68.03	58.89
0.7→0.5→0.3		**60.5**
1		57

TABLE 4.3: Co-Prune's variation using different α. $\alpha = 0$ reduces to DNS, $\alpha = 1$ means that mask generation replies on source model

60.91±0.8%, whose average exceeds all baseline methods by a large margin. This result shows that Co-Prune is consistent and stable with different sensitivity when tuning α.

4.3.2.3 Effect of Initial α_0

α_0 is set as initial value for transferring knowledge. We examine the effect of this initial state to final performance of Co-Prune. $\beta = 3$ and $\alpha_{\min} = 0.3$ are fixed and the best performance is reported in Fig.4.3(b). Performance drops when α_0 is too large (starting from learning too much from source and intense tuning of α) or too small (starting from learning too few from source). The performance is rather stable when α_0 ranges within $[0.6, 0.7]$ and shows better performance.

4.3.2.4 Effect of Minimum $\alpha_{\min}$

Co-Prune terminates after training reaches optimum using $\alpha_{\min}$ Effect of this final state to performance of Co-Prune is experimented with $\beta = 3$ and $\alpha_0 = 0.7$ fixed. The best performance with various $\alpha_{\min}$ is reported in Fig.4.3(c). $\alpha_{\min} = 0.7$ gets the worst performance, which is even lower than DNS. If $\alpha_{\min}$ is getting too large, it means there is almost no changes in α. When $\alpha_{\min}$ is set within $[0.3, 0.4]$, Co-Prune shows the best performance.

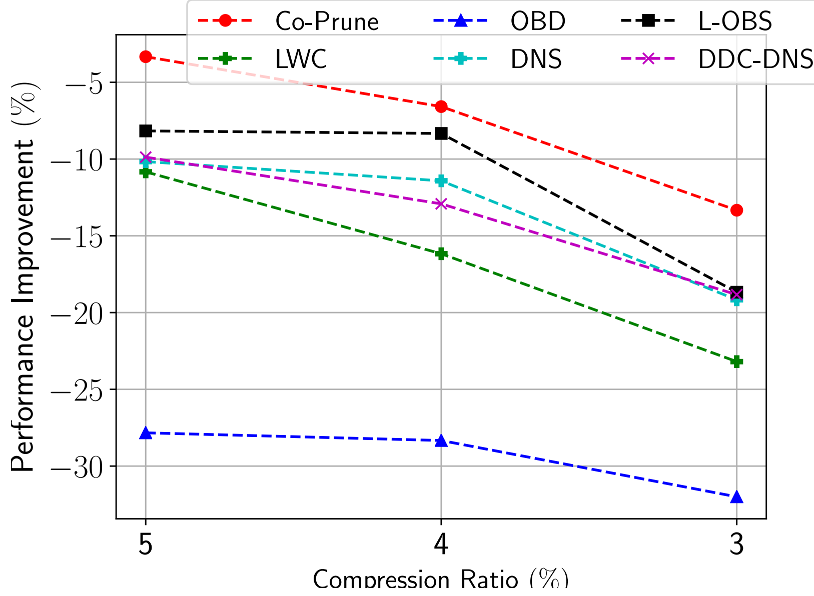


FIGURE 4.4: Performance improvement as CR changes in I→ P

4.3.3 Performance Under Various CRs

We further study how different methods perform in different CRs. I→P from ImageCLEF using ResNet18 is utilized as an example to verify the performances of different methods from CR: 5% → 4% → 3%. Fig.4.3(c) illustrates the variation of performance improvement under pruned methods. Among all the methods, Co-Prune performs the best in all CRs.

4.3.4 Complexity Analysis

We set number of parameters as n, training iteration as T. training-based pruning methods, such as Co-Prune, DNS, their time complexity can be represented as $O(t(n + n\text{log}n))$. For one-time pruning methods (pruning strategy is determined at the beginning), LWC: $O(tn+n\text{log}n)$, OBD: $O(tn+n\text{log}n+n^2)$, L-OBS: $O(tn+n\text{log}n + D^2)$, where D is the size of dataset used in hessian approximation.

4.3.5 Negative Transfer

We have conducted experiments using CIFAR10-STL9 and ImageCLEF, which are both commonly used dataset in domain adaptation. To study the negative transfer

CR (%)	Method	FP Acc(%)	Prune Acc (%)
10.4	LWC	65.3	60.2
	Co-Prune		57.3

Table 4.4: Experiments of MNIST as source dataset and STL10 as target dataset.

effect in Co-Prune, we especially choose MNIST as source dataset and STL10 as target dataset for ablation study.

We reshape instances in MNIST to 96×96 and copy their single channel information to form a 3-channel RGB input, to be compatible with data in STL10. We apply Co-Prune on this pair of datasets and follow the same training hyperparameter as mentioned above.

As Table 4.4 shows, compared with single-domain pruning (LWC), Co-Prune using a non-related dataset (MNIST) causes negative transfer to target task's pruning.

4.4 Conclusion

In this chapter, we propose a novel cross-domain pruning algorithm (Co-Prune) for deep neural network. Traditionally, pruning algorithms reply on a well-trained network and a huge amount of training data, which is not realistic in many domains. Co-Prune solves this problem by training-based pruning, with the construction of target's pruning mask that incorporates knowledge from the source domain. An adaptive transfer factor that controls the knowledge flow from the source domain for target pruning is introduced to further boost the performance. Extensive experiments are conducted on two benchmark datasets to demonstrate Co-Prune's performance over baseline methods.

Chapter 5

Meta Compression

5.1 Introduction

Most [1] compression methods rely on a supervised training framework. Specially, given training data $\{\mathbf{X}, y\}^n$ and initial model with full-precision weights $\mathbf{W}$. In forward pass, $\mathbf{W}$ is compressed by non-differentiable "compressor" ($\mathcal{C}$) into compressed value $\hat{\mathbf{W}}$, which is utilized to generate training loss:

$$\ell = \text{Loss}(\hat{\mathbf{W}}; \mathbf{x}, y) = \text{Loss}(\mathcal{C}(\mathbf{W}); \mathbf{X}, y) \tag{5.1}$$

In quantization, $\mathcal{C}$ discretizes $\mathbf{W}$ into finite value $\hat{\mathbf{W}}$, such as $\{\pm 2^k, 0\}, k = 1, 2, ...$ [49]. While $\mathcal{C}$ sets parts of $\mathbf{W}$ to be 0 in pruning, by imposing masks which share the same shape of corresponding weights.

During backward, gradient of $\mathbf{W}$ is unattainable due to $\mathcal{C}$'s non-differentiability. To enable training, Straight-Through-Estimator (STE) is widely used in quantization to propagate gradients with clipping:

$$\frac{\partial L}{\partial \mathbf{W}} = \left(\frac{\partial L}{\partial \hat{\mathbf{W}}}\right)_{||\mathbf{W}||_{l1} \leq 1} \tag{5.2}$$

[1]The work in this chapter has been published in 33st Conference on Neural Information Processing Systems (NeurIPS 2019).

Gradient of $\mathbf{W}$ is assigned as the that of $\hat{\mathbf{W}}$ under the condition that $\mathbf{W} \leq 1$, otherwise gradient is cancelled.

Similar technique is utilized in pruning: $\mathcal{C}$ prunes portion of $\mathbf{W}$ into 0 according to different criterion. During training, gradient of un-pruned weights is accessible. While the gradient of pruned weights is passed from $\hat{\mathbf{W}}$ to $\mathbf{W}$ without clipping [21]:

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \hat{\mathbf{W}}} \tag{5.3}$$

Although these STE-like training-based compression [2] provides an end-to-end training method under non-smooth constraints, it inevitably brings the problem of *gradient mismatch*: the gradients of $\mathbf{W}$ are not generated using the value of $\mathbf{W}$, but rather its compressed value $\hat{\mathbf{W}}$. Few works have progressed to investigate how to obtain better gradients for compression training.

To overcome the problem of gradient mismatch and explore better gradients in training-based compression, inspired by [76], we propose to learn $\frac{\partial \mathcal{C}(\mathbf{W})}{\partial \mathbf{W}}$ by a neural network ($\mathcal{M}$) during compression training. Such neural network is called *meta compressor* and is trained together with the base compressed model. This process is named as **Meta Compress**ion (Meta-Compress): **Meta-Quant** in quantization and **Meta-Prune** in pruning. Specially, in each backward propagation, $\mathcal{M}$ takes $\frac{\partial \ell}{\partial Q(\mathbf{W})}$ and $\mathbf{W}$ as inputs in a coordinate-wise manner, then its output is assigned to $\frac{\partial \ell}{\partial \mathbf{W}}$ for weights update using common optimization methods such as SGD and Adam. In the forward pass, inference is conducted using the compressed version of the updated weights, which produce the final outputs to be compared with the ground-truth labels for backward computation. During this process, gradient propagation from the compressed weights to the full-precision weights is handled by $\mathcal{M}$, which avoids the problem of non-differentiability and gradient mismatch. Besides, the gradients generated by the meta compressor are loss-aware, contributing to better performance of the compression training.

[2] In the following description, training-based compression refers to STE-like training-based compression

The exploration of gradient is guided by the loss of base compressed model, where the base network and meta network don't share a same task. Training meta compressor using only information from base network doesn't guarantee that the generated gradient truly reflects the desired gradient used in compression training. To achieve gradients that not only decrease base training loss but also resemble "exact" gradient of compressed weights, we propose to incorporate side information for the training of Meta-Prune. Specially, in Meta-Prune, Meta-Compress is trained using the loss from base model and reconstruction loss of gradient in un-pruned weights.

Compared with commonly-used STE and manually designed gradient propagation in compression training, Meta-Compress learns to generate proper gradients without any expert knowledge. The whole process is end-to-end and meta compressor can be viewed as a plug-in to any base model, making it easy and general to be implemented in modern architectures. After compression training is finished, meta compressor can be removed and consumes no extra space for inference. We compare Meta-Compress with STE under different compression functions: dorefa [18], BWN [6] in quantization and magnitude-based pruning [21], and various optimization techniques (SGD, Adam [70]) using CIFAR10/100, ImageNet in various base models to verify Meta-Compress's generalizability.

5.2 Problem Statement and Preliminary

Given a training set of n labeled instances $\{\mathbf{x}, y\}^n$, a pre-trained full-precision base model f with $\mathcal{L}$-layer is parameterized as $\mathbf{W} = [\mathbf{W}_1, ..., \mathbf{W}_{\mathcal{L}}]$. We define a pre-processing function $\mathcal{A}(\cdot)$ and a compression function $\mathcal{C}(\cdot)$. $\mathcal{A}(\cdot)$ converts $\mathbf{W}$ into $\tilde{\mathbf{W}}$ which is scaled and centralized to make it easier for compression. $\mathcal{C}(\cdot)$ compresses $\tilde{\mathbf{W}}$ into $\hat{\mathbf{W}}$. In quantization, $\hat{\mathbf{W}}$ is constrained into $\alpha \times \{\pm 2^0, \pm 2^1, ..., \pm 2^{k-1}\}$ as k-bit quantization. In pruning, $\hat{\mathbf{W}}$ contains at most $C\%$ of non-zero elements, C is denoted as "Compression Rate" (CR).

Quantization: 2 pre-processing functions and corresponding quantization methods (dorefa[3], BWN) are studied in this work:

[3] In this work, we only consider the forward quantization function for weights quantization used in [18], and denote it as "dorefa"

- dorefa:

$$\tilde{\mathbf{W}} = \mathcal{A}(\mathbf{W}) = \frac{\tanh(\mathbf{W})}{2\max(|\tanh(\mathbf{W})|)} + \frac{1}{2} \tag{5.4}$$

$$\hat{\mathbf{W}} = \mathcal{C}(\tilde{\mathbf{W}}) = 2 \times \frac{\text{round}\left[(2^k - 1)\tilde{\mathbf{W}}\right]}{2^k - 1} - 1 \tag{5.5}$$

- BWN:

$$\tilde{\mathbf{W}} = \mathcal{A}(\mathbf{W}) = \mathbf{W} \tag{5.6}$$

$$\hat{\mathbf{W}} = \mathcal{C}(\tilde{\mathbf{W}}) = \frac{1}{n}||\tilde{\mathbf{W}}||_{l1} \times \text{sign}(\tilde{\mathbf{W}}) \tag{5.7}$$

Pruning: Dynamic pruning [21] is studied. Specially, $\hat{\mathbf{W}}$ is attained by imposing a pruning mask $\mathbf{M}$ in $\tilde{\mathbf{W}}$, where $\mathbf{M}$ is dynamically updated according to $\tilde{\mathbf{W}}$ during training.

$$\tilde{\mathbf{W}} \;=\; \mathcal{A}(\mathbf{W}) = \mathbf{W} \tag{5.8}$$

$$\hat{\mathbf{W}} \;=\; \mathcal{C}(\tilde{\mathbf{W}}) = \tilde{\mathbf{W}} \odot \mathbf{M} \tag{5.9}$$

$\mathbf{M}$ is determined by the current value of $\tilde{\mathbf{W}}$, function of measuring importance: $\text{Imp}(\cdot)$, and function to determining pruning threshold $\sigma_{\text{Imp}}(C)$ as:

$$\mathbf{M}_{i,j} \;=\; \begin{cases} 0, & \text{if } \; \text{Imp}(\hat{\mathbf{W}}_{i,j}) < \sigma_{\text{Imp}}(C), \\ 1, & \text{if } \; \text{Imp}(\hat{\mathbf{W}}_{i,j}) \geq \sigma_{\text{Imp}}(C), \end{cases} \tag{5.10}$$

$$\text{Imp}(\mathbf{W}_{i,j}) \;=\; ||\mathbf{W}_{i,j}||_{l1}. \tag{5.11}$$

$$\sigma_{\text{Imp}}(C) \;=\; \text{Increasing}(\text{Imp}(\mathbf{W}), |\mathbf{W}| \times C). \tag{5.12}$$

$\text{Imp}(\mathbf{W}_{i,j})$ measures the importance of weight. For example, [23] defined it as: $\text{Imp}(w) = \frac{\partial^2 L}{\partial w^2} \times w^2$, involving hessian of each parameter and its square. In this work, we use absolute magnitude as criterion [1, 21]. $\sigma_{\text{Imp}}(C)$ represents a mapping from weights to a pruning threshold. $\text{Increasing}(\mathbf{W}, i)$ takes out the i-th element

of an increasing ranking of elements from Imp($\mathbf{W}$). $|\mathbf{W}|$ retrieves the number of elements in $\mathbf{W}$. The updated weights with higher absolute magnitude is kept by indicating corresponding elements as 1 in $\mathbf{M}$.

In training-based compression, $\mathcal{C}$ is non-differentiable. Traditional methods approximated $\frac{\partial \ell}{\partial \mathbf{W}}$ by (5.3) or similar techniques. We aim at improving compressed model with better accuracy and faster convergence with the assistance of Meta-Compress.

5.3 Meta-Compress

5.3.1 Generation of Meta Gradient

Our proposed Meta-Compress incorporates a shared meta compressor $\mathcal{M}_\phi$ parameterized by ϕ across layers into compression training. After $\mathbf{W}$ is compressed as $\hat{\mathbf{W}}$ (subscript $\mathcal{L}$ is omitted for ease of notation), $f(\hat{\mathbf{W}}; \mathbf{x})$ generates a loss ℓ as in (5.1), such that the gradient of ℓ w.r.t $\hat{\mathbf{W}}$ is obtained by chain rules, which is denoted by $g_{\hat{\mathbf{W}}} = \frac{\partial \ell}{\partial \hat{\mathbf{W}}}$. The meta compressor$\mathcal{M}_\phi$ receives $g_{\hat{\mathbf{W}}}$ and $\tilde{\mathbf{W}}$ as inputs, and outputs the gradient on $\tilde{\mathbf{W}}$, denoted by $g_{\tilde{\mathbf{W}}} = \frac{\partial \ell}{\partial \tilde{\mathbf{W}}}$:

$$g_{\tilde{\mathbf{W}}} = \mathcal{M}_\phi(g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}). \tag{5.13}$$

The gradient $g_{\tilde{\mathbf{W}}}$ is further back-propagated to $\mathbf{W}$, whose gradient, $g_{\mathbf{W}}$, is computed via:

$$g_{\mathbf{W}} = \frac{\partial L}{\partial \tilde{\mathbf{W}}} \frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}} = g_{\tilde{\mathbf{W}}} \frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}} = \mathcal{M}_\phi(g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}) \frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}}, \tag{5.14}$$

where $\frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}}$ describes the differentiable relation between $\mathbf{W}$ and $\tilde{\mathbf{W}}$, $\frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}} = \frac{1-\tanh^2(\mathbf{W})}{\max(|\tanh(\mathbf{W})|)}$ for dorefa according to (5.4), and $\frac{\partial \tilde{\mathbf{W}}}{\partial \mathbf{W}} = \mathbf{1}$ for BWN and pruning according to (5.6), (5.8). This process is denoted as *calibration*.

Before updating $\mathbf{W}$, $g_{\mathbf{W}}$ is firstly processed according to different optimization methods to produce the final update value for each weight. This process is named *gradient refinement*, which is denoted by $\pi(\cdot)$ in the sequel. Specifically, for Stochastic Gradient Descent (SGD), $\pi(g_{\mathbf{W}}) = g_{\mathbf{W}}$. For other optimization methods such as Adam, $\pi(\cdot)$ can be implemented as $\pi(g_{\mathbf{W}}) = g_{\mathbf{W}} + \text{residual}$, where

residual is computed according to different gradient refinement methods. Finally, the full-precision weights $\mathbf{W}$ is updated as:

$$\mathbf{W}^{t+1} = \mathbf{W}^{t} - \alpha\pi(g^{t}_{\mathbf{W}}), \tag{5.15}$$

where t denotes training iteration and α indicates the learning rate. Fig.5.1 illus-

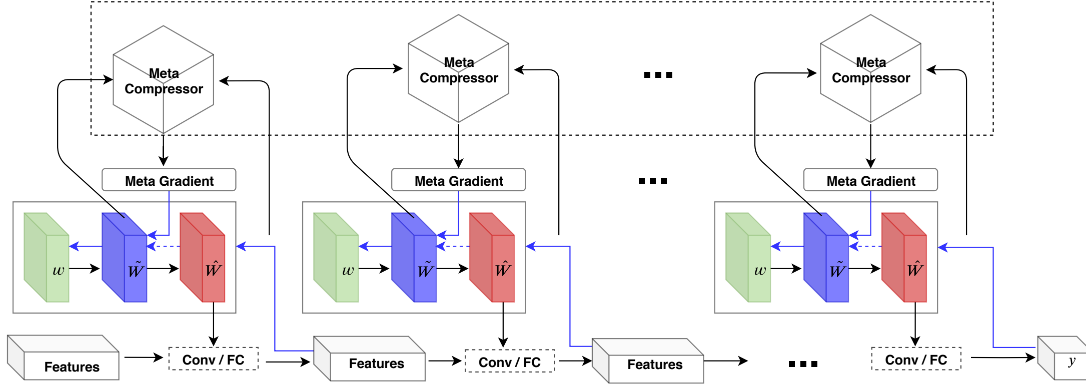


FIGURE 5.1: The overflow of Meta-Compress: During backward propagation, gradients are represented as blue line. Dash blue line means this propagation is non-differentiable and requires special handling. A shared meta network $\mathcal{M}$ is constructed which takes $g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}$ as input and its output will replace the gradient of $\tilde{\mathbf{W}}$ as $g_{\tilde{\mathbf{W}}}$.

trates the overall procedure of Meta-Compress. After $g_{\hat{\mathbf{W}}}$ is obtained from backpropagation, it is fed into the meta compressor together with $\tilde{\mathbf{W}}$ to generate a meta gradient $g_{\tilde{\mathbf{W}}}$, which is backpropagated to $\mathbf{W}$ using (5.14). Finally, $\mathbf{W}$ is updated with (5.15), with the assistance of different optimization methods reflected in $\pi(\cdot)$.

Compared with [76], which directly learns $g_{\mathbf{W}}$, Meta-Compress applies a neural network to learn $g_{\tilde{\mathbf{W}}}$, which is unaccessible in compression training due to the property of non-differentiability of the compression functions. Our work resolves the issue of non-differentiability and is general to different optimization methods.

In Meta-Prune, only portion of $\mathbf{W}$ is compressed into 0, whose gradient is generated by (5.14). For the unp-runed weights, their gradient is attained using usual backpropagation.

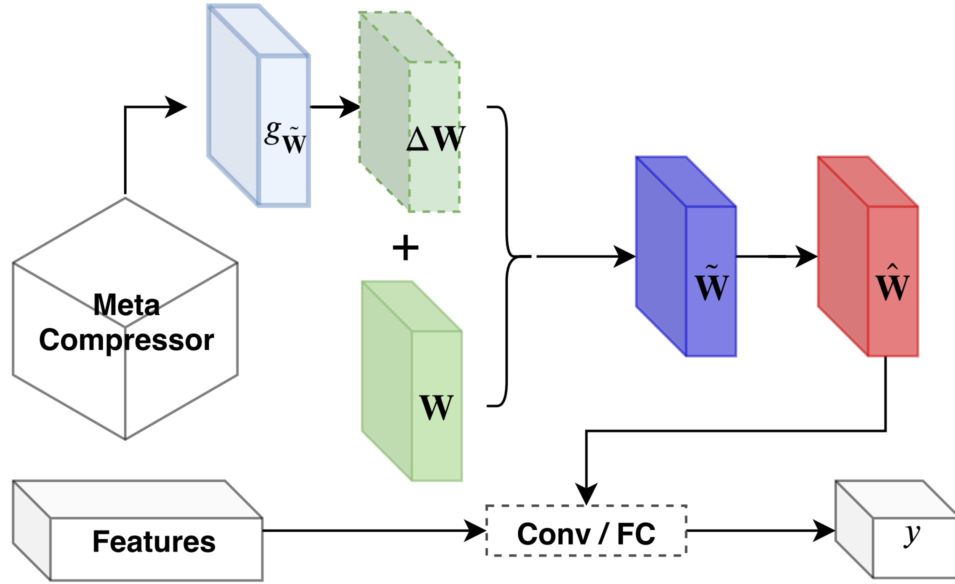


FIGURE 5.2: Incorporation of meta compressor into compression training. Red dash box is composed of calibration, gradient refinement and multiplication of learning rate α. Output of meta compressor is involved in $\mathbf{W}$'s update and contributes to final loss, constructing a differential path from loss to ϕ-parameterized meta compressor.

5.3.2 Training of Meta Compressor

Similar to [76], meta compressor is a **coordinate-wise** neural network, which means that each weight parameter is processed independently. For a *single* weight index i in $g_{\hat{\mathbf{W}}_i}$, $\tilde{\mathbf{W}}_i$ receives its corresponding gradient $g_{\tilde{\mathbf{W}}_i}$ via $g_{\tilde{\mathbf{W}}_i} = \mathcal{M}_\phi(g_{\hat{\mathbf{W}}_i}, \tilde{\mathbf{W}}_i)$.

For efficient processing, during each inference, the inputs in (5.13) are arranged as batches with size 1. Specially, suppose $\mathbf{W}$ comes from a convolution layer with shape $\mathbb{R}^{o \times i \times k \times k}$, where o, i, k denotes the number of output channels, input channels and kernel size, respectively. Then $\tilde{\mathbf{W}}$, $\hat{\mathbf{W}}$ and the corresponding gradient share the same shape, which is a reshaping of inputs in (5.13) to $\mathbb{R}^{(o \times i \times k^2) \times 1}$.

Recall from (5.15) and (5.14), the output of $\mathcal{M}_\phi$ is incorporated into the value of updated $\mathbf{W}^t$, which is then quantized in next iteration's inference. Therefore, $\mathcal{M}_\phi$ is connected to the final quantization training loss, which receives gradient update on ϕ backpropagated from the final loss. By introducing the meta compressor to produce $g_{\tilde{\mathbf{W}}}$, Meta-Compress not only avoids the non-differentiability issue for parameters in the base model, but also provides an end-to-end training throughout the whole network. Moreover, the meta compressor is loss-aware, hence it is trained

to generate more accurate update for $\mathbf{W}$ for reducing the final loss, which explores how gradient can be modified to assist compression training. Figure.5.2 illustrates the detailed process when incorporating the meta compressor into the compression training of the base model, which forms a differentiable path from the final loss to ϕ. In the meantime of compression training in $\mathbf{W}$, ϕ is also learned for each training iteration t:

Forwards:

$$\begin{aligned}
\tilde{\mathbf{W}}^t &= \mathcal{A}(\mathbf{W}^t) = \\
\mathcal{A}&\left[\mathbf{W}^{t-1} - \alpha \times \pi(\mathcal{M}_\phi(g_{\hat{\mathbf{W}}}^{t-1}, \tilde{\mathbf{W}}^{t-1})\frac{\partial \tilde{\mathbf{W}}^{t-1}}{\partial \mathbf{W}^{t-1}})\right], \qquad (5.16)\\
\ell &= \text{Loss}\left\{f\left[\mathcal{C}(\tilde{\mathbf{W}}^t); \mathbf{x}\right], y\right\}, \qquad (5.17)
\end{aligned}$$

Backward:

$$\frac{\partial \ell}{\partial \phi^t} = \frac{\partial \ell}{\partial \tilde{\mathbf{W}}^t}\frac{\partial \tilde{\mathbf{W}}^t}{\partial \phi^t} = \mathcal{M}_\phi(g_{\hat{\mathbf{W}}^t}, \tilde{\mathbf{W}}^t)\frac{\partial \tilde{\mathbf{W}}^t}{\partial \phi^t}. \qquad (5.18)$$

Here $\frac{\partial \tilde{\mathbf{W}}}{\partial \phi}$ is differentiable because $\mathcal{A}$ is differentiable. Furthermore, a differentiable meta neural network is chosen.

Pay attention to the time step in forward, we use combination of $\mathbf{W}^{t-1}$ and meta gradient to represent $\mathbf{W}^t$, in order to incorporate $\mathcal{M}_\phi$. $\mathbf{W}_t$ will be updated after backward, which can be regarded as late weights update.

5.3.3 Incorporation of Side Information in Meta-Prune

(5.18) calculates the gradient of Meta-Compress by loss of base model. The updated Meta-Compress leads to decrease of base loss. We provide side information to assist the training of Meta-Compress in pruning.

In training-based pruning, un-pruned / remain weights ($\mathbf{W}_r$) is updated using standard backpropagation. Based on this observation, an assumption is hold that meta compressor should provide unchanged gradient for $\mathbf{W}_r$ if they are processed by meta compressor. Although meta compressor focuses on gradient of pruned

weight in practice, meta compressor is ought to learn identical mapping for $g_{\mathbf{W}}$ under certain criterion that the corresponding $\mathbf{W}$ is un-pruned.

To tackle this problem, a reconstruction loss for gradient of $\mathbf{W}_r$ is introduced. Specially, gradient of $\mathbf{W}_r$ is denoted as $g_{\mathbf{W}_r}$ and fed into Meta-Compress, whose output is compared with the input using L_2 norm:

$$\ell_{side} = ||\mathcal{M}_\phi(g_{\mathbf{W}_r}) - g_{\mathbf{W}_r}||_2^2 \tag{5.19}$$

By enforcing the resemblance between output and input of Meta-Compress when $\mathbf{W}_r$ is processed, meta compressor learns to provide less unchanged gradient for pruned weights that resemble the un-pruned counterpart. Besides, it regularizes meta compressor to generate more stable gradient. A hyper-parameter λ as trade-off factor between (5.19) and (5.17) is introduced to balance the effects:

$$\ell_{prune} = \ell + \lambda \times \ell_{side} \tag{5.20}$$

which brings in the modified gradient for meta compressor as:

$$g_\phi = \frac{\partial \ell}{\partial \phi} + \lambda \times \frac{\partial \ell_{side}}{\partial \phi} \tag{5.21}$$

5.3.4 Design of Meta Compressor

$\mathcal{M}_\phi$ is a parameterized and differentiable neural network to generate the meta gradient. It can be viewed as a generalization of STE. For example, $\mathcal{M}_\phi$ degrades into STE if it clip $g_{\tilde{\mathbf{W}}}$ according to the absolute magnitude of $\tilde{\mathbf{W}}$: $g_{\tilde{\mathbf{W}}} = \mathcal{M}_\phi(g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}) = g_{\hat{\mathbf{W}}} \cdot \mathbf{1}_{|\tilde{\mathbf{W}}| \leq 1}$. We design 3 different realizations of meta compressor. The first formulation simply uses a neural network composed of 2 or multiple layers of fully-connected layer. It only requires $g_{\hat{\mathbf{W}}}$ as input, which is named as `FCGrad`:

$$\mathcal{M}_\phi(g_{\hat{\mathbf{W}}}) = \mathrm{FCs}(\phi, \sigma, g_{\hat{\mathbf{W}}}), \tag{5.22}$$

where σ represents the nonlinear activation. Since previous successful experimental results brought by STE shows that a good $g_{\tilde{\mathbf{W}}}$ should be generated by considering the value of $\tilde{\mathbf{W}}$. Based on this observation, we construct another 2 forms of meta compressor with $\tilde{\mathbf{W}}$ fed as input and multiply the output of these neural networks

with $g_{\hat{\mathbf{W}}}$ to incorporate gradient information from its subsequent step. Specifically, we have `MultiFC`:

$$\mathcal{M}_\phi(g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}) = g_{\hat{\mathbf{W}}} \cdot \text{FCs}(\phi, \sigma, \tilde{\mathbf{W}}). \tag{5.23}$$

Another network incorporates LSTM and FC to construct $\mathcal{M}$, which is inspired by [76] that uses memory-based neural network as the meta learner. We denote it as `LSTMFC`:

$$\mathcal{M}_\phi(g_{\hat{\mathbf{W}}}, \tilde{\mathbf{W}}) = g_{\hat{\mathbf{W}}} \cdot \text{FCs}(\phi_{FCs}, \sigma, (\text{LSTM}(\phi_{LSTM}, \tilde{\mathbf{W}}))). \tag{5.24}$$

When using LSTM as meta compressor, each coordinate of the weights keeps a track of the hidden states generated by LSTM, which contains the memory of historical information of $g_{\hat{\mathbf{W}}}$ and $\tilde{\mathbf{W}}$.

5.3.5 Algorithm and Implementation Details

Algorithm 3: Meta-Compress

Require: Training dataset $\{\mathbf{x}, y\}^n$, well-trained full-precision base model $\mathbf{W}$.
Ensure: Quantized base model $\hat{\mathbf{W}}$.

1: Construct meta compressor $\mathcal{M}_\phi$, training iteration $t = 0$.
2: **while** not optimal **do**
3:   **for** Layer l from 1 to L **do**
4:     $\hat{\mathbf{W}}_l^t = \mathcal{C}(\tilde{\mathbf{W}}_l^t) = \mathcal{C}\left\{\mathcal{A}\left[\mathbf{W}_l^{t-1} - \alpha \times \pi(\mathcal{M}_\phi(g_{\hat{\mathbf{W}}_l}^{t-1}, \tilde{\mathbf{W}}_l^{t-1}) \cdot \frac{\partial \tilde{\mathbf{W}}_l^{t-1}}{\partial \mathbf{W}_l^{t-1}})\right]\right\}$
5:   **end for**
6:   Calculate loss: $L = \text{Loss}\left\{f\left[\mathcal{C}(\tilde{\mathbf{W}}^t); \mathbf{x}\right], y\right\}$
7:   Generate $g_{\hat{\mathbf{W}}^t}$ using chain rules.
8:   Calculate meta gradient $g_{\tilde{\mathbf{W}}^t}$ using $\mathcal{M}_\phi$.
9:   Calculate $\frac{\partial \ell}{\partial \phi^t}$ by (5.18) or (5.21)
10:   **for** Layer l from 1 to $\mathcal{L}$ **do**
11:     $\mathbf{W}_l^t = \mathbf{W}_l^{t-1} - \alpha \times \pi(\mathcal{M}_\phi(g_{\hat{\mathbf{W}}_l}^{t-1}, \tilde{\mathbf{W}}_l^{t-1}) \cdot \frac{\partial \tilde{\mathbf{W}}_l^{t-1}}{\partial \mathbf{W}_l^{t-1}})$
12:   **end for**
13:   $\phi^{t+1} = \phi^t - \gamma \times \frac{\partial \ell}{\partial \phi^t}$ (γ is the learning rate of the meta compressor)
14:   $t = t + 1$
15: **end while**

Algorithm.3 illustrates the detailed process of Meta-Compress. A shared meta compressor $\mathcal{M}_\phi$ is firstly constructed and randomly initialized. During each training iteration, line 3-6 describes the forward process: for each layer, $g_{\hat{\mathbf{W}}}$ and $\tilde{\mathbf{W}}$

from the previous iteration are fed into $\mathcal{M}_\phi$ to generate the meta gradient $g_{\tilde{\mathbf{W}}}$ to perform inference, as indicated from line 3-5. Since $g_{\hat{\mathbf{W}}}$ hasn't been calculated in the first iteration, normal compression training is conducted at the first iteration: $\hat{\mathbf{W}} = \mathcal{C}(\tilde{\mathbf{W}}) = \mathcal{C}\left[\mathcal{A}(\mathbf{W})\right]$ to replace line 4. Line 7-9 shows the backward process: $\hat{\mathbf{W}}$'s gradient can be attained through error backpropagation, which is shown in line 7. During the backward process, $g_{\hat{\mathbf{W}}}$ and $\tilde{\mathbf{W}}$ of the current iteration are obtained and their outputs from $\mathcal{M}_\phi$ are saved for computation in the next iteration, denoted as $g_{\tilde{\mathbf{W}}^{t+1}}$ as in line 7-8. By incorporating $\mathcal{M}_\phi$ into the inference graph, its gradient is obtained in line 9. Finally, $g_{\tilde{\mathbf{W}}}$ is used to calculate $g_{\mathbf{W}}$, which is then processed by different optimization methods using $\pi(\cdot)$, leading to the update of $\mathbf{W}$ shown in line 10-12. In the first iteration, due to the lack of $g_{\tilde{\mathbf{W}}}$, weights update of $\mathbf{W}$ is not conducted. Note that ϕ from the meta compressor is updated in line 13.

5.4 Experiment

Meta-Compress focuses on the penetration of non-differentiable compressor function during training-based methods. We conduct comparison experiments with STE as baseline. In Meta-Quant, we compare using 2 forward quantization methods: 1) dorefa [18], 2) BWN [6] and 2 optimization methods: 1) SGD 2) Adam [70]. In Meta-Prune, magnitude-based pruning [21] is utilized as forward pruning method with Adam as optimization.

Three benchmark datasets are used including ImageNet ILSVRC-2012 and CIFAR10/100. Regarding deep architectures, we experiment with ResNet20/32/44 on CIFAR10. Since CIFAR10/100 share the same input dimension, we modify the output dimension of the last fully-connected layer from 10 to 100 in ResNet56/110 for CIFAR100. For ImageNet, ResNet18 is utilized for comparison. For all the experiments conducted and compared in Meta-Quant, **all** layers in the networks are quantized using **1 bit**: each layer contains only 2 values. In Meta-Prune, CR is set as 5% and 10% in each layer: every layer contains only 5% or 10% remain weights with rest pruned.

For experiments on CIFAR10/100, we set the initial learning rate as $\alpha = 1e^{-3}$ for base models and the initial learning rate as $\gamma = 1e^{-3}$ for meta compressor. For

fair comparison, we set total training epochs as 100 for all experiments, α and γ will be divided by 10 after every 30 epochs. For ImageNet, the initial learning rate is set as $\alpha = 1e^{-4}$ for the base model using dorefa and BWN. Initial γ is set as $1e^{-3}$. α decreases to $\{1e^{-5}, 1e^{-6}\}$ when training comes to 10 / 20 epochs. γ reduces to $\{1e^{-4}, 1e^{-5}\}$ in accordance to the change of the learning rate in base models with total epoch as 30. Batch size is 128 for CIFAR/ImageNet. Specially in Meta-Prune, a weight decay of $5e^{-4}$ is imposed on training of meta compressor and λ to control side information penalty is set as $1e^{-3}$.

All experiments are conducted for 5 times, the statistics of last 10/5 epochs' test accuracy are reported as the performance of both proposed and baseline methods in CIFAR/ImageNet datasets. We also demonstrate the empirical convergence speed among different methods through training loss curves.

Detailed hyper-parameters in different realizations of Meta-Quant in CIFAR experiments are the following: In `MultiFC`, a 2-layer fully-connected layer is used with hidden size as 100, no non-linear activation is used. In `LSTMFC`, a 1-layer LSTM and a fully-connected layer are utilized, with the hidden dimension set as 100. In `FCGrad`, a 2-layer fully-connected meta model is used with hidden size as 100 without non-linear activation.

In ImageNet experiments, we use `MultiFC`/`FCGrad` with 2/1-layer fully-connected layer, whose hidden dimension is 100.

5.4.1 Experimental Results and Analysis

5.4.1.1 Meta-Quant

Table 5.1 shows the overall experimental results on CIFAR10 for Meta-Quant and STE using different forward quantization methods and optimizations. Variants of Meta-Quant shows significant improvement over STE baseline, especially SGD is used.

CIFAR100 is a more difficult task than CIFAR10, which contains much more fine-grained classes with a total number of 100 classes. Table 5.2 shows the overall experimental results on CIFAR100 for MetaQuant and STE using different forward

quantization methods and optimizations. Similar to CIFAR10, Meta-Quant outperforms by a large margin than STE in all cases, showing that Meta-Quant has significant improvement in more challenging tasks than traditional methods.

5.4.1.2 Meta-Prune

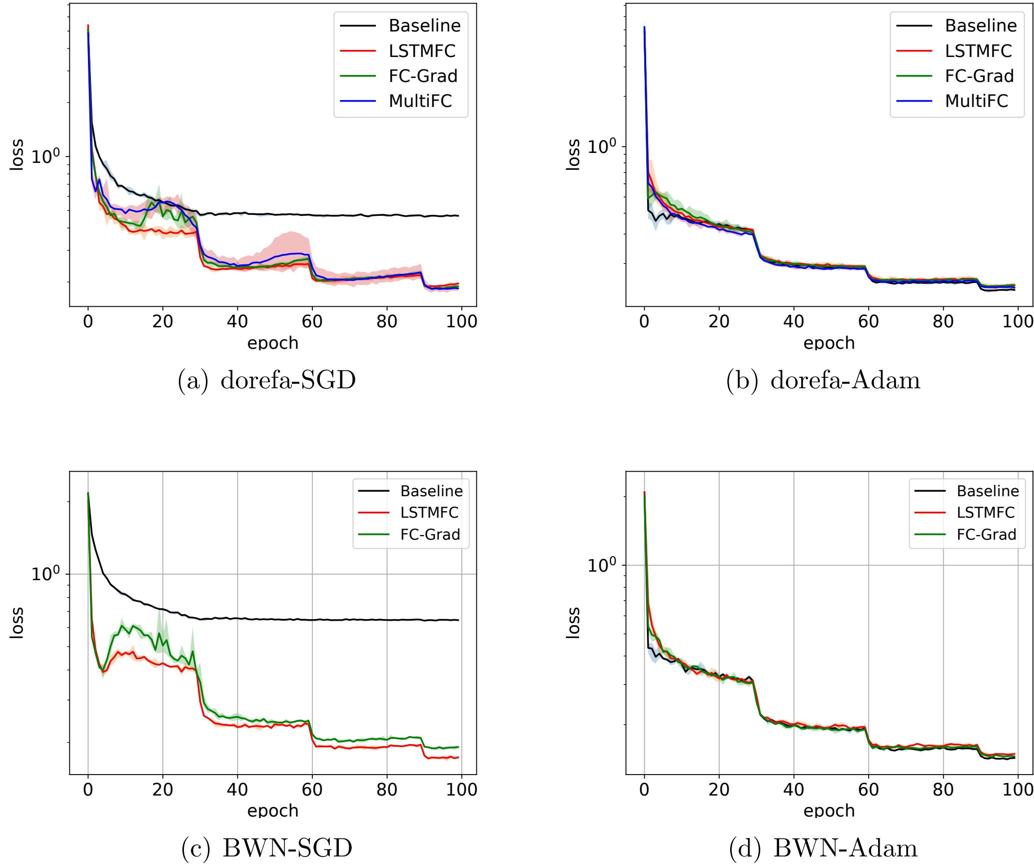
Table 5.4 shows the overall experimental results of Meta-Prune in various architectures and datasets. Meta-Prune outperforms DNS in all the setting. Especially in more difficult task such as CIFAR100 and ImageNet datasets, Meta-Prune achieves large improvement over DNS. Meta-Prune is able to retrieve similar or better performance than DNS with lower compression rate: In ImageNet, Meta-Prune's test accuracy in 10% is even above the corresponding of DNS in 20%.

We conjecture that complex model architecture brings in difficulty of searching pruned weights and adjusting un-pruned weights. Standard backpropagation adopted by DNS provides bias gradients for pruned weights, which hinders the whole training process. While Meta-Prune generates gradient based on the loss of base model, indirectly guiding to provide proper gradient for pruned weights. Besides, the side information illustrated in Sec.5.3.3 regularizes the learning of meta compressor: For pruned weights with large magnitude, meta compressor tends to provides unchanged gradient.

5.4.2 Empirical Convergence Analysis

5.4.2.1 Meta-Quant

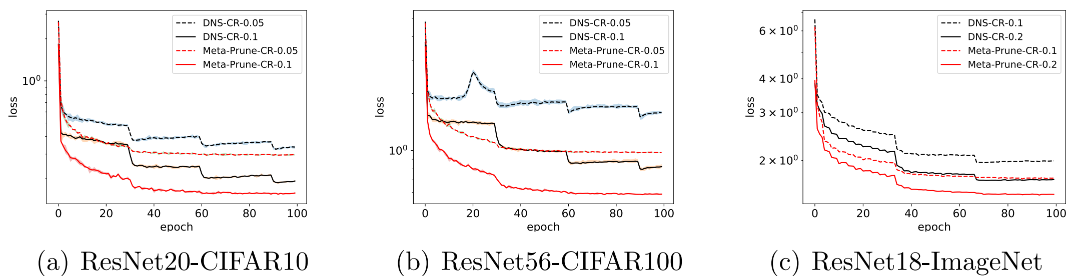
In this experiment, we compare the performances of variants of Meta-Quant and STE during the training process to demonstrate their convergence speeds. ResNet20 using dorefa is utilized as an example. As Fig.5.3 shows, under the same task and forward quantization method, Meta-Quant shows tremendous convergence advantage over STE using SGD, including much faster descending speed of loss and obviously lower loss values. In Adam, although all the methods show similar decreasing speed, Meta-Quant methods finally reach to lower loss values, which is also reflected in the test accuracy reported in Table 5.1.



(a) dorefa-SGD (b) dorefa-Adam

(c) BWN-SGD (d) BWN-Adam

FIGURE 5.3: Convergence speed of Meta-Quant V.S STE using SGD/Adam, dorefa/BWN in ResNet20, CIFAR10, dorefa.



(a) ResNet20-CIFAR10 (b) ResNet56-CIFAR100 (c) ResNet18-ImageNet

FIGURE 5.4: Convergence speed of Meta-Prune V.S DNS using various model and datasets.

Overall, Meta-Quant shows better convergence than STE using different forward quantization methods and optimizations. The improvement is more obvious when SGD is chosen. We conjecture that the performance difference between SGD and Adam is due to the following reason: SGD simply updates full-precision weights using the calibrated gradient from $g_{\tilde{\mathbf{W}}}$, which directly reflects the output of meta

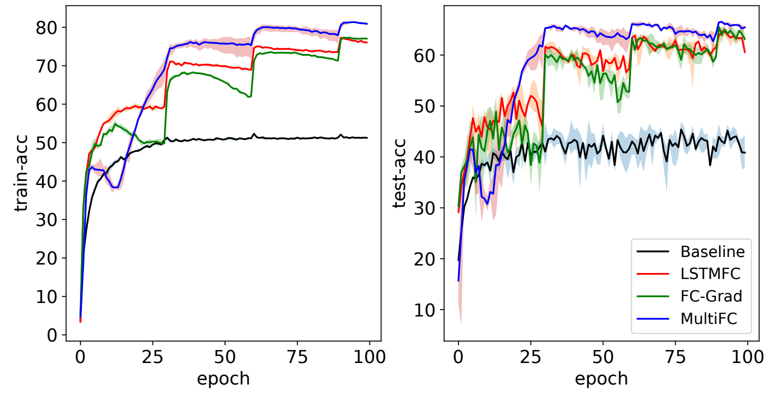


FIGURE 5.5: Meta-Quant V.S. STE in ResNet56-CIFAR100

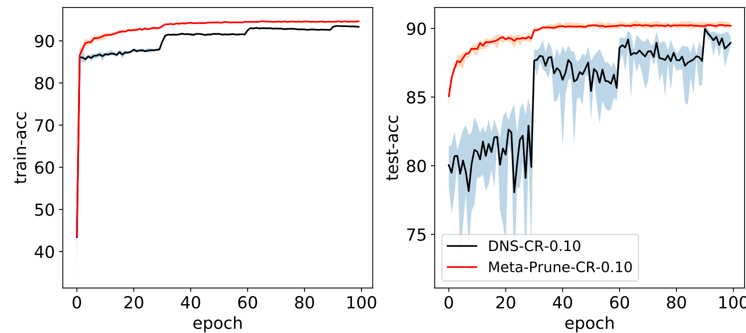


FIGURE 5.6: Meta-Prune V.S. DNS in ResNet20-CIFAR10

compressorcompared to STE. Adam aggregates the historical information of $g_{\mathbf{W}}$ and normalizes the current gradient, which to a certain degree shrinks the difference of meta compressor and STE.

Training and testing accuracy is shown in Fig.5.5 using ResNet56, CIFAR100, dorefa as forward and SGD as optimization. After training exceeded 3th epoch, performance of Meta-Quant began to grow much faster than STE. After 30th epoch, STE tended to remain stable and change slightly, without continuous growth even learning rate decreased. While variants of Meta-Quant kept growing with obvious uprising when learning rate changed.

5.4.2.2 Meta-Prune

We utilize ResNet20-CIFAR10, ResNet56-CIFAR100 and ResNet18-ImageNet under different compression rate to illustrate the training convergence between Meta-Prune and DNS. As shown in Fig.5.4: For each compression rate, Meta-Prune and DNS start from the similar loss while Meta-Prune achieves much faster decreasing speed than DNS. Finally, Meta-Prune reaches much lower training loss than DNS. In ResNet18-ImageNet, Meta-Prune even attains a similar loss with DNS at 10% lower compression ratio.

Besides, training process of Meta-Prune is more stable than DNS. As shown in Fig.5.4(b), when compression rate comes at 0.05, an unpredictable uprising appeared around 20th epoch for DNS. On the contrary, Meta-Prune demonstrates stable decreasing in loss during the whole process.

We further analyze the training and testing performance during pruning training in Fig.5.6. ResNet20-CIFAR10 with compression rate as 0.05 is utilized as example. At the beginning, Meta-Prune and DNS showed similar training accuracy while Meta-Prune demonstrated much better generalization ability in testing performance. As training proceeded, Meta-Prune kept maintaining overperformance than DNS in training accuracy. This difference is more obvious in testing accuracy, where Meta-Prune showed better and more stable than DNS during the whole training.

5.4.3 Effect of Side Information in Meta-Prune

We compare baseline method against Meta-Prune with / without side information to demonstrate the effect of side information in Table 5.5. Penalty factor λ is tested at $1e^{-3}/1e^{-2}$. Without side information, Meta-Prune is able to outperform baseline. Side information further boosts the performance of Meta-Prune by 2-3%.

Besides, training curve is analyzed in Fig.5.7: Meta-Prune with side information is able to achieve a stable training process and fast decreasing loss, which finally leads to lower training error than DNS and basic Meta-Prune.

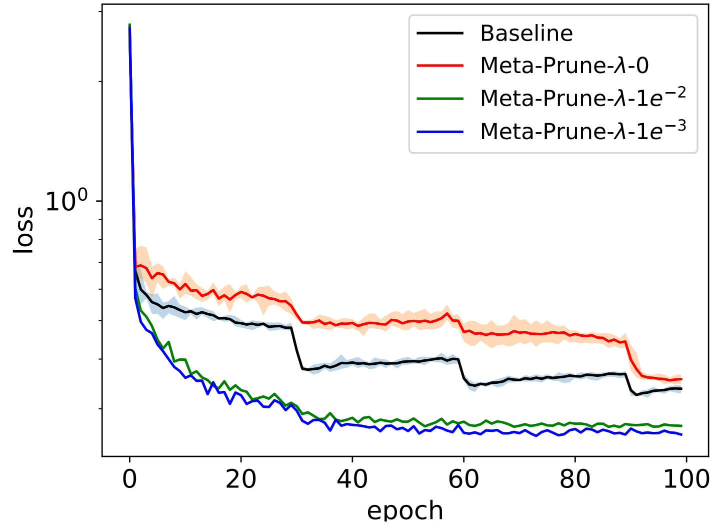


FIGURE 5.7: Training convergence of DNS, Meta-Prune and Meta-Prune with side information.

5.4.4 Performance Comparison with Non-STE Training-based Quantization

Meta-Quant aims at improving training-based quantization by learning better gradients for penetration of non-differentiable quantization functions. Some advanced quantization methods avoid discrete quantization. In this section, we compare Meta-Quant with Non-STE training-based quantization: ProxQuant ([77]), LAB ([43]) to demonstrate that traditional STE training-based quantization is able to achieve better performance by using Meta-Quant.

Due to the difference of the initial full-precision model used, we only report the performance drop in terms of test accuracy after quantization (the smaller the better). We compare MetaQuant with ProxQuant using ResNet20/32/44, LAB using its proposed architecture[4] on CIFAR10 with all layers quantized to binary values. As shown in Table 5.6, MetaQuant shows better performance than both baselines.

ELQ ([45]) and TTQ ([17]) are compared in 3rd row in Table 5.6 using ImageNet datasets. Although over-performance, ELQ is a combination of a series of previous quantization methods and tricks on incremental quantization. Meta-Quant focuses

[4](2x128C3)-MP2-(2x256C3)-MP2-(2x512C3)-MP2-(2x1024FC)-10FC

more on how to improve STE-based training quantization, without any extra loss and training tricks. TTQ is a non-symmetric ternarization with $\{0, \alpha, -\beta\}$ as ternary points. Meta-Quant follows dorefa using a symmetric quantization which leads to efficient inference.

5.4.5 Performance Comparison with Non-training-based Pruning

Meta-Prune is a training-based pruning method, which incorporates a dynamic pruning strategy in training process. LWC [1] and L-OBS [25] can be categorized into non-training pruning, for they adopts a "prune-retrain" scheme. Specially, LWC uses weights' magnitude as criterion for pruning, while L-OBS measure weights' importance by the combination of hessian and power of magnitude. Unpruned weights are adjusted before retrain in L-OBS.

LWC and L-OBS are compared with Meta-Prune using ResNet20/32 in CIFAR10, compression rate as 0.05. For LWC and L-OBS, after pruning is conducted, we retrain the un-pruned weights for 100 epochs, with $1e^{-3}$ as initial learning rate which decrease by a factor of 0.1 every 30 epochs. In Table 5.7, the comparison is demonstrated. Meta-Prune achieves better performance than LWC and L-OBS by around 4-7%.

5.4.6 Meta-Compress Training Analysis

Training of Meta-Compress involves computation in training of meta compressor. To analyze the additional training time, training time per iteration as for Meta-Quant using `MultiFC` and dorefa with STE using ResNet20 in CIFAR10 (Intel Xeon CPU E5-1650 with GeForce GTX 750 Ti). Meta-Quant costs **51.15** seconds to finish one iteration of training while baseline method uses **38.17s**. However, In real deployment meta compressor is removed, Meta-Quant is able to provide better test performance without any extra inference time.

5.4.7 Analysis of Hyperparameters and Architecture in Meta Compressor

(5.23), (5.22), (5.24) describes the general architecture of meta compressor. However, how to choose the proper network and hidden dimension is important. The sensitivity of these hyper-parameters affects the utility of Meta-Quant. In this section, we take ResNet20 in CIFAR10 using dorefa, Adam as an example to demonstrate how hidden dimension, activation and number of layers affect performance in `MultiFC`.

As Table 5.8 shows, after hidden dimension climbs up to 30, the variation of the hidden size doesn't affect the performance of Meta-Quant. Besides, the activation shows similar effects to the final performance. These results show that MetaQuant is insensitive to the hyper-parameters once it has already been equipped with enough capacity for meta training.

5.4.8 Looking Deeper into the Meta-Compress

This section investigates how Meta-Compress actually accelerates training convergence and improves final performance. Meta-Quant with `FCGrad` is taken as example: In `FCGrad`, if no non-linear activation is used, the 2-layer fully-connected meta compressor is equivalent to an adaptive factor multiplied in $g_{\hat{\mathbf{W}}}$ as shown in (5.22). We studied the changing of this amplification factor with different number of hidden sizes in meta compressor during training using ResNet20, CIFAR10, dorefa, SGD as an example. As shown in Fig.5.8, `FCGrad` magnifies $g_{\hat{\mathbf{W}}}$'s magnitude adaptively (shown in dotted lines) during training, in accordance with the descending of the training loss. Larger hidden dimensions bring higher increase of the amplification, leadings to faster convergence. Finally, all 3 meta compressor climb to similar performances. In the analysis of `FCGrad`, it shows that Meta-Quant adaptively tunes $g_{\tilde{\mathbf{W}}}$ by a multiplication of $g_{\hat{\mathbf{W}}}$, which is able to produce faster convergence and better performance in training-based quantization.

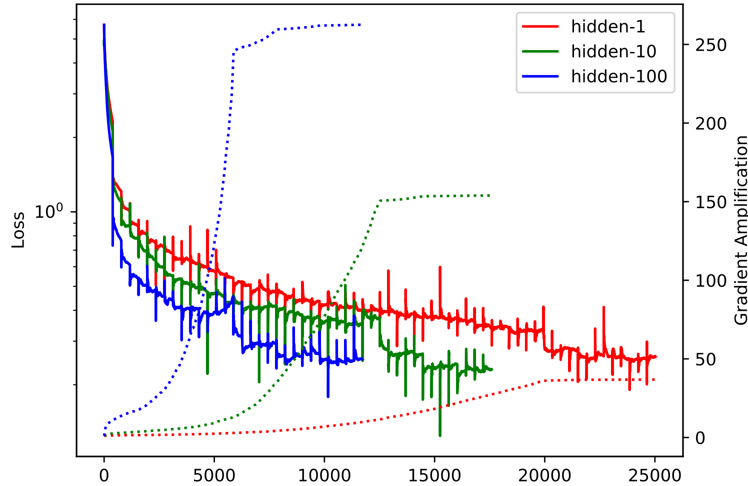


FIGURE 5.8: Loss and gradient amplification during training.

5.5 Conclusion

In this chapter, we propose a novel method (Meta-Compress) to learn the gradient for penetration of the non-differentiable compression function in training-based compression by a meta compressor. This meta network is general enough to be incorporated into various base models and can be updated using the loss of the base models. We further introduce side information to regularize the training of Meta-Prune.

In various quantization and pruning tasks, we show that the meta gradients generated by Meta-Compress are able to provide better convergence speed and final performance, under different forward compression functions, optimization methods and compression rate.

Network	Forward	Backward	Optimization	Test Acc (%)	FP Acc (%)
ResNet20	dorefa	STE	SGD	80.745(2.113)	91.5
		`MultiFC`		**88.942(0.466)**	
		`LSTMFC`		88.305(0.810)	
		`FCGrad`		88.840(0.291)	
		STE	Adam	89.782(0.172)	
		`MultiFC`		89.941(0.068)	
		`LSTMFC`		**89.979(0.103)**	
		`FCGrad`		89.962(0.068)	
	BWN	STE	SGD	75.913(3.495)	
		`LSTMFC`		**89.289(0.212)**	
		`FCGrad`		88.949(0.231)	
		STE	Adam	89.896(0.182)	
		`LSTMFC`		90.036(0.109)	
		`FCGrad`		**90.042(0.098)**	
ResNet32	dorefa	STE	SGD	82.911(1.680)	92.13
		`MultiFC`		89.637(0.380)	
		`LSTMFC`		**90.397(0.149)**	
		`FCGrad`		89.934(0.246)	
		STE	Adam	90.172(0.077)	
		`MultiFC`		90.966(0.064)	
		`LSTMFC`		90.948(0.074)	
		`FCGrad`		**90.976(0.068)**	
	BWN	STE	SGD	79.768(2.062)	
		`LSTMFC`		**90.568(0.169)**	
		`FCGrad`		90.241(0.316)	
		STE	Adam	91.015(0.087)	
		`LSTMFC`		91.002(0.077)	
		`FCGrad`		**91.034(0.067)**	
ResNet44	dorefa	STE	SGD	86.686(1.020)	93.56
		`MultiFC`		90.546(0.218)	
		`LSTMFC`		91.494(0.163)	
		`FCGrad`		**91.539(0.097)**	
		STE	Adam	91.079(0.064)	
		`MultiFC`		91.772(0.073)	
		`LSTMFC`		91.870(0.022)	
		`FCGrad`		**91.989(0.067)**	
	BWN	STE	SGD	82.647(0.334)	
		`LSTMFC`		91.498(0.057)	
		`FCGrad`		**91.614(0.081)**	
		STE	Adam	91.121(0.023)	
		`LSTMFC`		91.498(0.271)	
		`FCGrad`		**92.107(0.059)**	

TABLE 5.1: Experimental result of Meta-Quant and STE using dorefa, BWN on CIFAR10

Network	Forward	Backward	Optimization	Test Acc (%)	FP Acc (%)
ResNet56	dorefa	STE	SGD	42.265(8.143)	71.22
		MultiFC		**65.791(0.415)**	
		LSTMFC		63.645(2.183)	
		FCGrad		64.351(0.935)	
		STE	Adam	66.419(0.533)	
		MultiFC		**66.588(0.375)**	
		LSTMFC		66.483(0.793)	
		FCGrad		66.564(0.351)	
	BWN	STE	SGD	34.479(11.737)	
		LSTMFC		63.346(2.253)	
		FCGrad		**64.402(1.434)**	
		STE	Adam	64.297(1.309)	
		LSTMFC		66.584(0.349)	
		FCGrad		**67.018(0.329)**	
ResNet110	dorefa	STE	SGD	43.419(18.902)	72.54
		MultiFC		**68.269(0.136)**	
		LSTMFC		64.753(2.850)	
		FCGrad		66.145(2.490)	
		STE	Adam	66.836(1.198)	
		MultiFC		68.418(0.235)	
		LSTMFC		67.138(1.286)	
		FCGrad		**68.741(0.363)**	
	BWN	STE	SGD	35.227(19.408)	
		LSTMFC		**66.242(2.979)**	
		FCGrad		64.791(4.096)	
		STE	Adam	66.265(1.429)	
		LSTMFC		67.767(1.391)	
		FCGrad		**69.114(0.181)**	

Table 5.2: Experimental result of MetaQuant and STE using dorefa, BWN on CIFAR100

Network	Forward	Backward	Optimization	FP Top1/Top5(%)	Quant Top1/Top5 (%)
ResNet18	dorefa	STE	Adam	69.76/89.08	58.349(2.072)/81.477(1.567)
		MultiFC			59.472(0.025)/82.410(0.010)
		FCGrad			**59.835(0.359)/82.671(0.232)**
	BWN	STE			59.503(0.835)/82.549(0.506)
		FCGrad			**60.328(0.391)/83.025(0.234)**

Table 5.3: Experimental result of MetaQuant and STE using dorefa, BWN on ImageNet.

Dataset	Network	CR (%)	Backward	Test Acc (%)	FP Acc (%)
CIFAR10	ResNet20	5	DNS	83.494(7.663)	91.5
			`MultiFC`	**88.560(0.009)**	
		10	DNS	89.191(0.425)	
			`MultiFC`	**90.220(0.023)**	
	ResNet32	5	DNS	87.306(3.304)	92.13
			`MultiFC`	89.276(0.027)	
		10	DNS	90.874(0.303)	
			`MultiFC`	**91.147(0.036)**	
	ResNet44	5	DNS	89.434(14.745)	93.56
			`MultiFC`	**89.974(0.004)**	
		10	DNS	92.163(0.107)	
			`MultiFC`	**92.268(0.007)**	
CIFAR100	ResNet56	5	DNS	46.556(69.982)	71.22
			`MultiFC`	63.247(0.025)	
		10	DNS	64.720(2.073)	
			`MultiFC`	**67.922(0.011)**	
	ResNet110	5	DNS	55.039(66.415)	72.54
			`MultiFC`	67.118(0.011)	
		10	DNS	67.981(0.342)	
			`MultiFC`	**69.630(0.037)**	
ImageNet	ResNet18	10	DNS	58.783(0.307)/82.170(0.121)	69.76/89.08
			`MultiFC`	**64.013(0.003)/85.614(0.004)**	
		20	DNS	63.704(0.091)/85.525(0.047)	
			`MultiFC`	**67.065(0.005)/87.509(0.001)**	

Table 5.4: Experimental result of Meta-Prune and DNS.

Method	λ	Pruned Acc(%)
Baseline	-	83.494(7.663)
Meta-Prune	0	85.837(4.409)
	$1e^{-3}$	**88.56(0.009)**
	$1e^{-2}$	87.589(0.006)

Table 5.5: Comparison experiments for Meta-Prune with variant of side information.

Network	Method	Acc Drop (%)	Network	Method	Acc Drop (%)
ResNet20	ProxQuant	1.29	ResNet32	ProxQuant	1.28
	Meta-Quant	**0.7**		Meta-Quant	**0.39**
ResNet44	ProxQuant	0.99	LABNet	LAB	1.4
	Meta-Quant	**0.08**		Meta-Quant	**-0.2**
ResNet18	ELQ	**3.55/2.65**	ResNet18-2bits	TTQ [17]	**3.00/2.00**
	Meta-Quant	6.32/4.31		Meta-Quant	5.17/3.59

Table 5.6: Experimental result of Meta-Quant V.S ProxQuant, LAB, ELQ, TTQ.

Network	Method	Pruned Acc (%)	Network	Method	Pruned Acc (%)
ResNet20	LWC	81.68(0.365)	ResNet32	LWC	83.92(0.112)
	L-OBS	84.34(0.370)		L-OBS	85.27(0.497)
	Meta-Prune	**88.56(0.009)**		Meta-Prune	**89.276(0.027)**

Table 5.7: Experimental result of Meta-Prune V.S LWC, L-OBS

Hidden	Activation	Acc
1	None	89.91
30		90.48
100		90.48
1500		90.46
100	ReLU	90.49
	tanh	90.46

Table 5.8: Experimental result of MetaQuant-FC under different hidden dimension, activation

Chapter 6

Transferable Compressor

Most benchmark deep learning architectures are overparameterized, which contain millions of parameters and require huge scale of training data to train a precise model for a specific task. To make deploying such resource-intensive deep learning models on edge devices possible, recently, much attention has been paid on training-based model compression methods. The basic idea is to learn a compressed version of a well-trained deep model by using training data of a specific task of interest, and then deploy the compressed model on an edge device used for the same specific task. One major limitation in these methods is that the task-specific training data is assumed to be sufficient to train a precise compressed version, especially when the benchmark data used to train the full-precision deep model is very different from the task-specific data. However, in some real-world scenarios, high-quality labeled data is often in short supply, such as in the medical domain.

Thus, a natural question raises "*Can we generate a precise compressed network for any novel task with scarce training data?*" In this chapter, we attempt to provide a positive answer. Our high-level idea is to meta-learn the training process of compression from multiple model compression tasks (a.k.a. source tasks), and then transfer the meta knowledge to guide generating a precise compressed network to a novel task with scarce training data. We denote this overall process by **Trans**ferable **Comp**ressor (TransComp). The key of TransComp lies in the form of meta knowledge and how it is learned. Note that in conventional training-based compression methods, the overall process consist of two phases: 1) given a deep architecture, e.g., ResNet, GoogleNet, etc., denoted by $\mathcal{N}(\cdot)$, to use sufficient

training data to train a full-precision network, and 2) to use sufficient training data again together with some pre-defined update rules (e.g., the form of gradients to update compressed weights) to train a compressed version from the well-trained full-precise network. Therefore, in TransComp, we consider two types of meta knowledge: 1) the process of learning a good initialization of the full-precision network across all the source tasks, from which each tasks-specific compressed network is generated. Our intuition is that with a good generalizable initialization, different precise compressed versions ($\{\hat{\boldsymbol{\Theta}}_i\}$'s) for different source tasks can be produced with only scarce training data. 2) the process of automatically generating gradients to update parameters for compressed networks across all the source tasks. This is a key operation for the task of network compression.

To capture the process of learning a good initialization $\boldsymbol{\Theta}$ of the full-precision network, we borrow the idea of Model-Agnostic Meta-Learning (MAML) [20] to design a gradient-based meta-training framework. To automatically generate compression update rules, we design a learnable blackbox, which is a small neural network whose parameters are trained across all the source tasks. In the sequel, we denote this by $\mathcal{M}_\phi$ the meta compressor, where ϕ denotes its learnable parameters. Note that in [78], a similar idea was proposed to design a meta network to produce gradients for updating quantized weights. However, their work is focused on network quantization on a single task/domain. After the above two types of meta knowledge are learned, when a novel task comes, we first initialize the full-precise network using $\boldsymbol{\Theta}$: $\mathcal{N}(\boldsymbol{\Theta})$, and then use $\mathcal{M}_\phi$ with scarce training data of the novel task to learn the task-specific compressed network.

To test the effectiveness of TransComp we construct experiments in different problem settings. Specifically, we construct experiments on CIFAR100 and ImageNet in the setting where the novel tasks have scarce training data while the source tasks may have sufficient or scarce training data, and on Omniglot and MiniImageNet in the setting where both the novel tasks and the source tasks have extremely scarce data (few-shot). Empirical results show that compared with the existing compression methods, TransComp is able to learn a better compressed model for novel tasks in general.

To summarize, our contributions are two folds: 1) We offer the TransComp framework that is able to adaptively learn a precise compression model for any novel

task with scarce training data. 2) We conduct extensive experiments in different settings to demonstrate the effectiveness of TransComp.

6.1 Problem Statement and Preliminary

Given a specific deep architecture $\mathcal{N}(\cdot)$ and any novel task with scarce training data $\mathbf{D}_*^{tr} = \{\mathbf{X}_*^{tr}, \mathbf{y}_*^{tr}\}$, where $\mathbf{X}$ denotes the input data matrix and $\mathbf{y}$ denotes a vector of corresponding labels, the goal is to generate a compressed model $\mathcal{N}(\hat{\mathbf{\Theta}}_*)$ for the task, which is expected to make precise predictions on unseen test data of the task $\mathbf{D}_*^{tst} = \{\mathbf{X}_*^{tst}\}$. In this work, regarding the form of model compression, we only focus on pruning or quantization, i.e., $\hat{\mathbf{\Theta}}$ is either sparse or quantized. In addition, suppose we have a set of source compression tasks, each of which is associated with a dataset $\mathbf{D}_i = \{\mathbf{X}_i, \mathbf{y}_i\}_{i=1}^n$. Each dataset is split into a training set $\mathbf{D}_i^{tr} = \{\mathbf{X}_i^{tr}, \mathbf{y}_i^{tr}\}$ and a test set $\mathbf{D}_i^{tst} = \{\mathbf{X}_i^{tst}, \mathbf{y}_i^{tst}\}$. In training of TransComp, we aim to meta-learn a good common initialization $\mathbf{\Theta}$ for $\mathcal{N}(\cdot)$, and a transferable meta compressor $\mathcal{M}_\phi$ from $\{\mathbf{D}_i\}$'s. Before presenting our method in detail, we briefly introduce some preliminaries in the rest of this section.

6.1.1 Optimization-based Meta Learning

Gradient-based meta learning aims to learn a universal initialization $\mathbf{\Theta}$, which can quickly adapt to different tasks with a small number of training instances. MAML [20] is a representative in this line of research, which consists of two training phases: inner loop and outer loop. In the inner loop, each task is trained from $\mathbf{\Theta}$ with its own training dataset via backpropagation to obtain a task-specific model in terms of $\mathbf{\Theta}_i$. In the outer loop, and losses of all the tasks are used to update $\mathbf{\Theta}$ via backpropagation. The two phases are performed alternatingly until some stop criterion is met:

$$\text{Inner Loop Training :} \quad \mathbf{\Theta}_i^t = \mathbf{\Theta}_i^{t-1} - \alpha \times \frac{\partial \ell(\mathbf{\Theta}_i^{t-1}; \mathbf{D}_i^{tr})}{\partial \mathbf{\Theta}_i^{t-1}} \tag{6.1}$$

$$\text{Outer Loop Update :} \quad g_{\mathbf{\Theta}} = \frac{\partial \ell(\mathbf{\Theta}_i^T; \mathbf{D}_i^{tst})}{\partial \mathbf{\Theta}} \tag{6.2}$$

6.1.2 Compression Training

Compression training generates compressed neural networks under a training mechanism. Specifically, during forward inference, full-precision parameters $\mathbf{\Theta}$ are converted into compressed parameters $\hat{\mathbf{\Theta}}$. Input instances $\mathbf{X}$ are then fed into the compressed network to make predictions and compute errors against the groundtruth $\mathbf{y}$. In backward update, the errors are backpropagated via gradient descent with a learning rate α to update parameters. Formally, we have

$$\text{Forward}: \quad \hat{\mathbf{\Theta}} = \mathcal{C}(\mathbf{\Theta}) \quad \text{Loss} = \ell(\hat{\mathbf{\Theta}}; \mathbf{X}, \mathbf{y}) \tag{6.3}$$

$$\text{Backward}: \quad g_{\hat{\mathbf{\Theta}}} = \tfrac{\partial \ell}{\partial \hat{\mathbf{\Theta}}} \quad g_{\mathbf{\Theta}} = \frac{\partial \ell}{\partial \mathcal{C}(\mathbf{\Theta})} \times \frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}} \tag{6.4}$$

$$\text{Update}: \quad \mathbf{\Theta} := \mathbf{\Theta} - \alpha \times g_{\mathbf{\Theta}} \tag{6.5}$$

Note that the compression operation $\mathcal{C}(\,\cdot\,)$ is usually non-differentiable, which leads to the inaccessibility of $\frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}}$. To enable training, "Straight-Through-Estimator" (STE) is commonly used to penetrate $\frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}}$ by passing gradients of $\hat{\mathbf{\Theta}}$ to $\mathbf{\Theta}$ with corresponding full-precision parameters' magnitude being below 1, and canceling the rest with higher values i.e., $\frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}} \approx 1_{||\mathbf{\Theta}|| \leq 1}$.

As mentioned , in this work, we consider two types of compression: quantization and pruning. We follow the traditional compression techniques [18, 21] to define the compression operation $\mathcal{C}(\,\cdot\,)$ as:

$$\text{Quantization}: \quad \hat{\mathbf{\Theta}} = 2 \times \frac{\text{round}\left[(2^k - 1)\tilde{\mathbf{\Theta}}\right]}{2^k - 1} - 1, \text{ where } \tilde{\mathbf{\Theta}} = \frac{\tanh(\mathbf{\Theta})}{2\max(|\tanh(\mathbf{\Theta})|)} \tag{6.6}$$

$$\text{Pruning}: \quad \hat{\mathbf{\Theta}} = \mathbf{\Theta} \odot \mathbf{M}, \text{ where } \mathbf{M}_{i,j} = \begin{cases} 0, & \text{if } \operatorname{Imp}(\mathbf{\Theta}_{i,j}) < \sigma_{\text{Imp}}(\delta), \\ 1, & \text{if } \operatorname{Imp}(\mathbf{\Theta}_{i,j}) \geq \sigma_{\text{Imp}}(\delta), \end{cases} \tag{6.7}$$

where $\operatorname{Imp}(\mathbf{\Theta}_{i,j})$ measures the importance of a weight. Here we use $\operatorname{Imp}(\mathbf{\Theta}_{i,j}) = ||\mathbf{\Theta}_{i,j}||_{l_1}$. And $\sigma_{\text{Imp}}(\delta)$ denotes a mapping from parameters to a pruning threshold given a compression rate δ, which is the remaining percentage of parameters, the smaller the sparser. Specifically, $\sigma_{\text{Imp}}(\delta) = \text{Increasing}(\operatorname{Imp}(\mathbf{\Theta}), |\mathbf{\Theta}| \times \delta)$, where $\text{Increasing}(\mathbf{\Theta}, i)$ takes out the i-th element of an increasing ranking of elements from $\operatorname{Imp}(\mathbf{\Theta})$, and $|\mathbf{\Theta}|$ retrieves the number of elements in $\mathbf{\Theta}$.

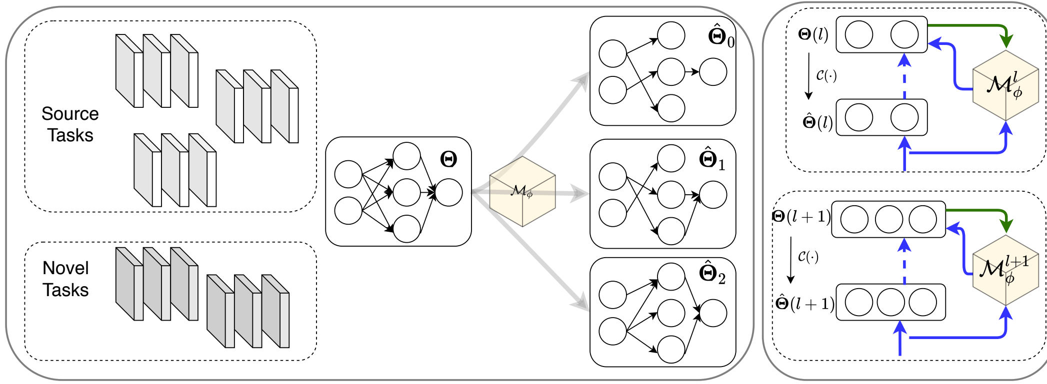


FIGURE 6.1: Overflow of TransComp: Left: For a coming task, in inner loop training, it starts from a learnable initialization $\boldsymbol{\Theta}$ and is trained to a task-specific compressed model ($\hat{\boldsymbol{\Theta}}_{0,1,2}$) via guidance of $\mathcal{M}_\phi$. Source tasks' training update $\boldsymbol{\Theta}$ and $\mathcal{M}_\phi$ in outer loop update. When novel task arrives, it is adapted from $\boldsymbol{\Theta}$ to reach compressed model by $\mathcal{M}_\phi$. Right: Blue lines represent the flow of gradient. The non-differentiable compression operation $\mathcal{C}(\cdot)$ between $\boldsymbol{\Theta}(l)$ and $\hat{\boldsymbol{\Theta}}(l)$ in layer l leads to gradient obstruction (blue dash line). To overcome it and explore better gradient, layer-wise $\mathcal{M}_\phi^l$ receive gradient of $\hat{\boldsymbol{\Theta}}(l)$ (blue solid line) and parameters of $\boldsymbol{\Theta}(l)$ (green line) to provide gradient for $\boldsymbol{\Theta}(l)$.

6.2 Transferable Compressor

Given a specific deep learning architecture $\mathcal{N}(\cdot)$ with L layers, TransComp aims to capture meta knowledge for compression across tasks and apply it to any novel task with scarce data to generate a precisely compressed version in the same architecture as $\mathcal{N}(\cdot)$. As discussed, the meta knowledge is embedded in a full-precision initialization $\boldsymbol{\Theta} = \{\boldsymbol{\Theta}_1, ..., \boldsymbol{\Theta}_L\}$ of $\mathcal{N}(\cdot)$ and a meta compressor $\mathcal{M}_\phi = \{\mathcal{M}_\phi^1, ..., \mathcal{M}_\phi^L\}$, which is used to automatically generate gradients to update compressed parameters. $\boldsymbol{\Theta}$ serves as a starting state for compression training of each task, aiming to provide a commonly good initialization for all the source tasks such that precise task-specific compression models $\hat{\boldsymbol{\Theta}}_i$ can be learned efficiently and effectively.

TransComp adopts a similar training protocol as in MAML to learn $\boldsymbol{\Theta}$ and $\mathcal{M}_\phi$ via alternating inner loop compression training and outer loop mete-knowledge update. To be specific, $\mathcal{N}(\cdot)$ is first initialized with $\boldsymbol{\Theta}$ that is a randomly initialized at the beginning, $\mathcal{N}(\boldsymbol{\Theta})$. The meta compressor $\mathcal{M}_\phi$ is also randomly initialized. After that we sample a subset of m tasks out of the n source tasks.

6.2.1 Inner loop compression training

For each task i in the sampled subset of source tasks, we start by initializing $\mathbf{\Theta}_i$ by $\mathbf{\Theta}$, i.e., $\mathbf{\Theta}_i = \mathbf{\Theta}$. In each training step, we convert $\mathbf{\Theta}_i$ to $\hat{\mathbf{\Theta}}_i$ via $\hat{\mathbf{\Theta}}_i = \mathcal{C}(\mathbf{\Theta}_i)$, where $\mathcal{C}$ is based on either (6.6) or (6.7), and perform a forward pass with $\mathbf{X}_i^{tr}$ to generate predictions $\mathcal{N}(\hat{\mathbf{\Theta}}_i; \mathbf{X}_i^{tr})$ as shown in (6.3). For a backward pass, by computing the loss between $\mathcal{N}(\hat{\mathbf{\Theta}}_i; \mathbf{X}_i^{tr})$ and $\mathbf{y}_i^{tr}$, we use gradient descent to update the full-precision parameters of the task $\mathbf{\Theta}_i$. Note that rather than using STE to approximate $\frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}}$ in (6.4), we use the meta compressor $\mathcal{M}_\phi$ to generate gradients. Formally, for layer l of $\mathcal{N}(\cdot)$ at iteration t, we approximate $\frac{\partial \mathcal{C}(\mathbf{\Theta})}{\partial \mathbf{\Theta}}$ by $\mathcal{M}_\phi^l \left(g_{\hat{\mathbf{\Theta}}_i^{t-1}(l)}, \mathbf{\Theta}_i^{t-1}(l) \right)$, where $\mathcal{M}_\phi^l$ denotes the meta compressor for the l-th layer and $\hat{\mathbf{\Theta}}_i^{t-1}(l)$ and $\mathbf{\Theta}_i^{t-1}(l)$ denote the parameters of the l-th layer of $\hat{\mathbf{\Theta}}$ and $\mathbf{\Theta}$ at iteration $t-1$, respectively. Thus, $\mathbf{\Theta}_i(l)$ is updated as follows:

$$\mathbf{\Theta}_i^t(l) = \mathbf{\Theta}_i^{t-1}(l) - \alpha \mathcal{M}_\phi^l \left(g_{\hat{\mathbf{\Theta}}_i^{t-1}(l)}, \mathbf{\Theta}_i^{t-1}(l) \right) \times g_{\hat{\mathbf{\Theta}}_i^{t-1}(l)}. \tag{6.8}$$

6.2.2 Outer loop meta knowledge update

After T steps inner loop training, in the outer loop, we update $\mathbf{\Theta}$ and $\mathcal{M}_\phi$ by using $\hat{\mathbf{\Theta}}_i^T$, which is the task-specific compressed model after T-step training, and $\mathbf{D}_i^{tst}$. Denote by $\ell_i^T = \ell(\hat{\mathbf{\Theta}}_i^T; \mathbf{D}_i^{tst})$ the loss of $\mathcal{N}(\hat{\mathbf{\Theta}}_i^T)$ on $\mathbf{D}_i^{tst}$. The gradient of ℓ_i^T w.r.t. $\mathbf{\Theta}$ is

$$\begin{aligned} g_{\mathbf{\Theta}}^{(i)} &\triangleq \frac{\partial \ell_i^T}{\partial \mathbf{\Theta}} = \frac{\partial \ell_i^T}{\partial \hat{\mathbf{\Theta}}_i^T} \frac{\partial \hat{\mathbf{\Theta}}_i^T}{\partial \mathbf{\Theta}_i^T} \prod_{t=1}^{t=T} \left(\frac{\partial \mathbf{\Theta}_i^t}{\partial \mathbf{\Theta}_i^{t-1}} \right) \\ &= \frac{\partial \ell_i^T}{\partial \hat{\mathbf{\Theta}}_i^T} \times (\mathcal{M}_\phi^T)_i \times \prod_{t=1}^{t=T} \left(\frac{\partial (\mathbf{\Theta}_i^{t-1} + (\mathcal{M}_\phi^{t-1})_i \times \frac{\partial \ell_i^{t-1}}{\partial \hat{\mathbf{\Theta}}_i^{t-1}})}{\partial \mathbf{\Theta}_i^{t-1}} \right) \end{aligned} \tag{6.9}$$

This gradient form is applied to every layer. For ease in presentation, we omit the layer script l. Abbreviation is applied for $(\mathcal{M}_\phi^t)_i = \mathcal{M}_\phi(g_{\mathbf{\Theta}_i^t}, \hat{\mathbf{\Theta}}_i^t)$. In inner loop compression training, i.e. $t < T$, we denote $\ell_i^t = \ell(\hat{\mathbf{\Theta}}_i^t, \mathbf{D}_i^{tr})$. In outer loop update when $t = T$, we use test data from source task to compute loss: $\ell_i^T = \ell(\hat{\mathbf{\Theta}}_i^T, \mathbf{D}_i^{te})$. And learning rate is omitted. First and second term in last derivation of (6.9) is accessible in current deep learning framework.

Each element in third term is reduced as: $(1+(\mathcal{M}_{\phi}^{t-1})_i \times \partial(\frac{\partial \ell_i^{t-1}}{\partial \hat{\mathbf{\Theta}}_i^{t-1}})/\partial \mathbf{\Theta}_i^{t-1})$. In practice, we assign $\partial(\frac{\partial \ell_i^{t-1}}{\partial \hat{\mathbf{\Theta}}_i^{t-1}})/\partial \mathbf{\Theta}_i^{t-1}$ as the hessian of $\mathbf{\Theta}_i^{t-1}$ (i.e. $\frac{\partial (\ell_i^{t-1})^2}{\partial^2 \mathbf{\Theta}_i^{t-1}}$) by neglecting compression operation from $\mathbf{\Theta}_i^{t-1} \to \hat{\mathbf{\Theta}}_i^{t-1}$. For first order approximation is used, last derivation of (6.9) reduces to only $\frac{\partial \ell_i^T}{\partial \hat{\mathbf{\Theta}}_i^T} \times (\mathcal{M}_{\phi}^T)_i$.

To update $\mathcal{M}_{\phi}$ in terms of parameters ϕ, we detach the gradient of input for $\mathcal{M}_{\phi}$, leading it free from higher order gradient computation. By observing that in the T-step training of the inner loop, we have $\hat{\mathbf{\Theta}}_i^T = \mathcal{C}(\mathcal{M}_{\phi}(g_{\mathbf{\Theta}_i^{T-1}}, \hat{\mathbf{\Theta}}_i^{T-1}) \times g_{\hat{\mathbf{\Theta}}_i^{T-1}} + ... + \mathcal{M}_{\phi}(g_{\mathbf{\Theta}^1}, \hat{\mathbf{\Theta}}^1) \times g_{\hat{\mathbf{\Theta}}_i^1} + \mathcal{M}_{\phi}(g_{\mathbf{\Theta}^0}, \hat{\mathbf{\Theta}}^0) \times g_{\hat{\mathbf{\Theta}}_i^0}) + \mathbf{\Theta})$. Thus, the gradient of ℓ_i^T w.r.t. ϕ is

$$g_{\phi}^{(i)} \triangleq \frac{\partial \ell_i^T}{\partial \hat{\mathbf{\Theta}}_i^T} \times \frac{\partial \hat{\mathbf{\Theta}}_i^T}{\partial \mathbf{\Theta}_i^T} \times \sum_{t=0}^{T-1} \left(g_{\hat{\mathbf{\Theta}}_i^t} \times \frac{\partial(\mathcal{M}_{\phi}(g_{\mathbf{\Theta}_i^t}, \hat{\mathbf{\Theta}}_i^t)}{\partial \phi} \right). \tag{6.10}$$

The first term in (6.10) is accessible. The second term reduces to $\mathcal{M}_{\phi}(g_{\mathbf{\Theta}_i^T}, \hat{\mathbf{\Theta}}_i^T)$. The third term is a summation of gradients of $\mathcal{M}_{\phi}$ w.r.t ϕ in each update step. As $\mathcal{M}_{\phi}$ is differentiable, $\partial \mathcal{M}_{\phi}/\partial \phi$ is accessible.

We accumulate $g_{\mathbf{\Theta}}^{(i)}$ and $g_{\phi}^{(i)}$ over the subset of m tasks, and update $\mathbf{\Theta}$ and ϕ, respectively, via

$$\mathbf{\Theta} = \mathbf{\Theta} - \alpha_{\mathbf{\Theta}} \times \sum_{i=1}^{m} g_{\mathbf{\Theta}}^{(i)}, \qquad \phi = \phi - \alpha_{\phi} \sum_{i=1}^{m} g_{\phi}^{(i)}. \tag{6.11}$$

After $\mathbf{\Theta}$ and ϕ (i.e., $\mathcal{M}_{\phi}$) are updated, we then re-sample another subset of m source tasks. The procedure of processing m source tasks with update on $\mathbf{\Theta}$ and ϕ is denoted by "episode". We repeat the inner loop and outer loop until some stop

criterion is met.

Algorithm 4: Training of TransComp

Require: A subset of m source tasks with $\{\mathbf{D}_i^{tr}\}$'s and $\{\mathbf{D}_i^{tst}\}$'s.

Ensure: Initialization $\mathbf{\Theta}$ and meta compressor $\mathcal{M}_\phi$.

1: **for** Task i in the subset **do**
2: **for** Update step t from 1 to T **do**
3: Convert $\mathbf{\Theta}_i^{t-1}$ to $\hat{\mathbf{\Theta}}_i^{t-1}$ via (6.6) or (6.7).
4: Forward inference on $\hat{\mathbf{\Theta}}_i^{t-1}$ with $\mathbf{X}_i^{tr}$ and get ℓ with $\mathbf{y}_i^{tr}$.
5: Compute gradient of ℓ w.r.t $\hat{\mathbf{\Theta}}_i^{t-1}$ as $g_{\hat{\mathbf{\Theta}}_i^{t-1}}$.
6: Feed $\mathbf{\Theta}_i^{t-1}$, $g_{\hat{\mathbf{\Theta}}_i^{t-1}}$ into $\mathcal{M}_\phi$ to generate gradient on $\mathbf{\Theta}_i^{t-1}$ as $g_{\mathbf{\Theta}_i^{t-1}}$.
7: Update $\mathbf{\Theta}_i^t = \mathbf{\Theta}_i^{t-1} - \alpha \times g_{\mathbf{\Theta}_i^{t-1}}$
8: **end for**
9: Convert $\mathbf{\Theta}_i^T$ to $\hat{\mathbf{\Theta}}_i^T$. Compute gradients for $\mathbf{\Theta}$ and ϕ using (6.9) and (6.10).
10: **end for**
11: Update $\mathbf{\Theta}$ and ϕ by (6.11).

6.3 TransComp in Practice

6.3.1 Regularization for Pruning

As in pruning, those unpruned parameters are able to retrieve their true gradients, we modify the gradient passing for pruning variants in TransComp as follow. We directly deliver $g_{\hat{\mathbf{\Theta}}^t}$ to $g_{\mathbf{\Theta}^t}$ for unpruned parameters. In the meanwhile, we aim to ensure $\mathcal{M}_\phi$ to produce $\mathbf{1}$ for unpruned parameters. Therefore, we incorporate a regularization term in $\mathcal{M}_\phi$ for pruning as $\ell_{reg} = \beta \times ||\mathcal{M}_\phi(\mathbf{\Theta}_{up}^t, g_{\hat{\mathbf{\Theta}}_{up}^t}) - \mathbf{1}||_2^2$, where $\mathbf{\Theta}_{up}^t$ represents the un-pruned parameters in step t and similarly for $g_{\hat{\mathbf{\Theta}}_{up}^t}$, and β is a trade-off parameter.

6.3.2 Design of $\mathcal{M}_\phi$

The layer-wise meta compressor $\mathcal{M}_\phi$ receives each layer's pre-compressed parameters $\mathbf{\Theta}^t$ and the gradient $g_{\hat{\mathbf{\Theta}}_i^t}$ as inputs, and generates meta gradients to achieve better performance. Inspired by [78], we design three types of neural networks for

$\mathcal{M}_\phi$ as follows. For each layer l:

$$\texttt{FCGrad}: \mathcal{M}^l_\phi(g_{\hat{\mathbf{\Theta}}^t_i(l)}) = \mathrm{FCs}_{\phi_l}(\sigma, g_{\hat{\mathbf{\Theta}}^t_i(l)}).$$

`FCGrad` takes $g_{\hat{\mathbf{\Theta}}^t_i(l)}$ as input only, which is fed into a fully-connected neural network with hidden activation as σ. In experiments, we use a two-layer fully-connected model with σ as ReLU for all layers.

$$\texttt{MultiFC}: \mathcal{M}^l_\phi(\mathbf{\Theta}^t_i(l)) = \mathrm{FCs}_{\phi_l}(\sigma, \mathbf{\Theta}^t_i(l)).$$

Similarly, a fully-connected neural network is designed in `MultiFC`, which only receives $\mathbf{\Theta}^t_i(l)$ as input.

$$\texttt{LSTMFC}: \mathcal{M}^l_\phi(\mathbf{\Theta}^t_i(l)) = \mathrm{FCs}_{\phi_l}(\sigma, \mathrm{LSTM}_{\phi_l}(\mathbf{\Theta}^t_i(l))).$$

In `LSTMFC`, we first process $\mathbf{\Theta}^t_i(l)$ using a LSTM, whose output is fed into a fully-connected model. Each layer in the base model contain one $\mathcal{M}^l_\phi$ from the same type. Inputs of $\mathcal{M}^l_\phi$ are processed independently: $\mathbf{\Theta}^t_i(l)$ and $g_{\hat{\mathbf{\Theta}}^t_i(l)}$ are fed into $\mathcal{M}^l_\phi$ without interaction with others. To process efficiently, inputs in one iteration are arranged into batch shape with number of parameters as batch size. Outputs are reshaped back into dimension of gradients for further process. In experiments, we use the same architecture of $\mathcal{M}^l_\phi$ for each layer, and the parameters of $\mathcal{M}^l_\phi$ are shared across tasks.

6.4 Experiments

We deign experiments to answer the following two questions: 1) Whether TransComp is able to leverage knowledge from multiple source tasks via meta-learning initialization $\mathbf{\Theta}$ and $\mathcal{M}_\phi$ to improve novel task compression. 2) Whether TransComp outperforms other algorithms that are designed for network compression with scarce data.

To answer the first question, we introduce two baselines: 1) 'Straight-Through-Estimator' (STE). It follows the framework of TransComp to learn an initialization. However, $g_{\mathbf{\Theta}^t}$ in each inner loop training is assigned as $g_{\hat{\mathbf{\Theta}}^t}$ as in (6.4). STE acts as a MAML version in compression training, which embeds source tasks' knowledge

in $\boldsymbol{\Theta}$ while does not utilize the meta compressor in inner loop training. 2) Meta-SGD [79]. It is proposed to learn a common gradient in inner loop training and initialization for all source and novel tasks. Although it is not designed for compression training, we modify it by replacing $\partial \ell^t / \partial \boldsymbol{\Theta}^t$ with a learnable parmater, which is updated using source tasks. Novel tasks directly use the learned gradient in inner loop compression training without further update, leading to the lack of specific knowledge in novel tasks' inner training. To answer the second question, we compare TransComp with other scarce data compression methods. In quantization, L-DNQ [80] is utilized, which is able to generate quantized models with limited data by distilling knowledge from full-precision well-trained model, with constraints on layer-wise output approximation. For each novel task, we pre-train a full-precision method using its corresponding scarce data. Then apply L-DNQ to recover its performance under quantization constraint. In pruning, we compare with Co-Prune [81]. This method cooperatively trains a pair of tasks, with the data-abundant task to assist providing pruning strategy for task with scarce data. We implement Co-Prune by sampling one source task with sufficient data to assist novel task's pruning.

Task i sampled in source tasks is consist of c classes, each with k_i^{tr}/k_i^{tst} training/test instances, i.e. $|\mathbf{D}_i^{tr}| = c \times k_i^{tr}, |\mathbf{D}_i^{tst}| = c \times k_i^{tst}$. We inner train T steps for each source task to reach its task-specific model. Novel tasks are sampled to include unseen c categories, each of which contain k_*^{tr}/k_*^{tst} training/test instances, leading to $|\mathbf{D}_*^{tr}| = c \times k_*^{tr}$ and $|\mathbf{D}_*^{tst}| = c \times k_*^{tst}$. We train both TransComp and baseline methods (STE, Meta-SGD) using source tasks and evaluate their performance in novel task: we adapt $\mathbf{X}_*^{tr}$ on trained $\boldsymbol{\Theta}$ and $\mathcal{M}_\phi$ to reach task-specific compressed model $\hat{\boldsymbol{\Theta}}_*$ after T_* steps inner loop compression training. $\hat{\boldsymbol{\Theta}}_*$ makes prediction on $\mathbf{X}_*^{tst}$ and its accuracy is calculated as performance. In all experiments, **all** layers are compressed using a same compression rate, e.g. same quantization bits and compression ratio.

Scarce Data: Two datasets and corresponding architectures are utilized: CIFAR100 and ImageNet. CIFAR100 consists of 100 classes with 500 instances in each category. We sample source tasks from the data of first 64 classes while the rest as novel tasks set. ImageNet possesses 1000 classes, each contains approximately 1300 images. The first 800 classes are categorized into source tasks set with the rest as novel task set. $\mathbf{D}_i^{tr}, \mathbf{D}_*^{tr}$ are sampled from original training data,

with $\mathbf{D}_i^{tst}, \mathbf{D}_*^{tst}$ using full data from test/validation dataset. Since the compression difficulty of different tasks varies with high variance. We design an evaluation protocol for fair comparison. Specially, for each novel task, we use its full data (FD-Compression) to perform compression training as a maximum performance bound: a full-precision model is trained with full data, then compression training (Sec.6.1) is applied on the well-trained initialization to achieve a compressed model. Besides, a portion of full data is selected as its scarce data partition, which is trained and compressed similarly, providing as a baseline (SD-Compression) for traditional compression training in scarce data scenario. 200 epochs are applied in both phases, and we record the test performance in last epoch. For each novel task, FD-Compression is regarded as a criterion for measuring performance of different methods: suppose performance of FD-Compression is a while comparison method reaches at b (denoted as absoluted performance). **Relative Performance** (RP) of the comparison method is calculated as $b - a$, which represents that for the exact novel task, comparison method in scarce-data setting achieves $b-a$ outpeformance compared with full-data baseline, the higher the better. For each method, we average its RP of the 100 novel tasks and report its mean and variance for comparison.

Extremely Scarce Data (Few-Shot): Besides, we compare TransComp in few-shot problem setting in Omniglot and MiniImageNet benchmark. Few-shot learning aims at generate task-specific model using extreme limited instances. We conduct experiments in both quantization and pruning. We use the same architectures and comparison protocol in [20]: we inner compression train 1000 novel tasks using trained $\mathbf{\Theta}$ and $\mathcal{M}_\phi$, and report their average performance with mean and variance. Full-precision MAML model is provided as maximum performance for comparison.
Cross Datasets: We further conduct experiments on cross datasets. Specially, we use different datasets for source / novel tasks sampling. We train $\mathcal{M}_\phi$ and $\mathbf{\Theta}$ using source tasks. Then the learned $\mathcal{M}_\phi$ and $\mathbf{\Theta}$ are applied to novel tasks for training and evaluation. We use CIFAR100 as dataset for source tasks sampling and STL10 for novel task generation.

6.4.1 Overall Experiments

6.4.1.1 CIFAR100

We use ResNet20 [13] as the base network for compression. Source tasks are sampled as $c = 5, k_i^{tr} = 200, k_i^{tst} = 100$. We construct novel tasks using $c = 5, k_*^{tr} = 100, k_*^{tst} = 100$, which contributes to 20% of its full data (500 instances in each category). Source tasks are inner trained for $T = 5$ steps. Novel tasks are adapted using $T_* = 10$ steps for TransComp and STE/ Meta-SGD. For methods of TransComp, hidden size of all variants is set as 10 with activation as ReLU. In pruning, β in Sec.6.3 is set as 10.0. We train TransComp and STE/Meta-SGD for 2000 episode.

Method	Quantization		Pruning	
	bit	RP	CR	RP
FD-Compression	2	0	30	0
SD-Compression		-5.56 (3.17)		-5.47 (2.94)
STE		3.37 (4.15)		9.51 (3.85)
Meta-SGD		1.68 (4.73)		9.62 (4.37)
L-DNQ/Co-Prune		5.32 (3.30)		0.58 (2.88)
`MultiFC`		4.27 (5.54)		9.30 (3.85)
`FCGrad`		5.92 (4.82)		10.32 (4.29)
`LSTMFC`		**9.42 (4.45)**		**10.46 (4.08)**

TABLE 6.1: Relative Performance of Compression (2-bit Quantition, $\delta = 30$ Pruning) on CIFAR100.

As shown in Table 6.1, comparing STE with SD-Compression, they use the same compression technique, but are different in initialization state. STE learns initialization in a meta-learning framework while SD-Compression's initialization is based on the scarce data pretraining. STE achieves better average relative performance than SD-Compression, which demonstrates that knowledge from source tasks is embedded in initialization under the meta-learning framework. However, STE does not learn knowledge in how to guide inner loop training from source tasks. In novel tasks' inner loop compression training, Meta-SGD produces gradients without accessing knowledge from novel tasks. Both leads to their underperformance than variants of TransComp. L-DNQ is able to preserve the performance of full-precision model trained by novel tasks with scarce data. Meta-learning based methods can outperform full-data compression with 20% scarce training data. Among them, `LSTMFC` is able to achieve the best performance in quantization and pruning.

Method	Quantization		Pruning	
	bit	PR	CR	PR
FD-Compression	2	0	30	0
SD-Compression		-13.15 (3.72)		-15.46 (3.83)
STE		-11.64 (4.88)		-4.67 (2.69)
Meta-SGD		-8.54 (2.95)		-4.47 (2.75)
L-DNQ/Co-Prune		-5.92 (2.27)		-7.16 (4.63)
`MultiFC`		-7.58 (4.84)		-4.61 (3.55)
`FCGrad`		-12.38 (5.04)		-4.05 (4.96)
`LSTMFC`		**0.54 (3.78)**		**-4.01 (4.05)**

TABLE 6.2: Relative Performance of Compression (2-bit Quantization, $\delta = 30$ Pruning) on ImageNet.

6.4.1.2 ImageNet

We construct tasks from ImageNet using $c = 10$ classes with $k_i^{tr} = 200, k_i^{tst} = 50, k_*^{tr} = 200, k_*^{tst} = 50$. Since the number of classified categories drop from 1000 to 10, to relieve overfitting, we divide the intermediate channel number in ResNet18[13] by a factor of 4, which leads to a slim version of ResNet18 with 6% parameters of the original one. And we resize data to 112×112. We train TransComp and STE/Meta-SGD for 1000 episodes. Due to the high volume of training data in each tasks, we set batch size as 500 and inner train for $T = 40$ update in source tasks, $T_* = 160$ for novel tasks. For TransComp, $\mathcal{M}_\phi$ is trained only using gradient in the first 4 updates. Hidden size is set as 10 with ReLU activation. In pruning $\beta = 10.0$. Table 6.2 illustrates the overall performance of scarce data compression in ImageNet: Overall, meta-learning based methods outperform SD-Compression due to the knowledge leverage from source tasks. In quantization, `LSTMFC` is able to achieved better performance than FD-Compression, at the same time outperforms baseline methods significantly. In pruning, TransComp shows an approximately 4% performance drop than FD-Compression. Still, `LSTMFC` outperforms other baseline methods.

6.4.1.3 Omniglot

We evaluate few-shot compression (2-bit quantization, $\delta = 20$ pruning) in c-way k-shot few shot setting, i.e. $k_i^{tr} = k_u^{tr} = k$ with c classes in both source / novel tasks. In Omniglot, three settings: 5-way 1-shot (5-1), 20-way 1-shot (20-1) and

Method	Quantization				Pruning			
	bit	5-1	20-1	20-5	δ	5-1	20-1	20-5
Full-Precision	32	97.70(0.87)	89.28(0.72)	96.72(0.39)	100	97.70(0.87)	89.28(0.72)	96.72(0.39)
STE	2	90.22(1.28)	75.88(1.59)	91.53(0.54)	20	94.89(0.73)	86.23(1.09)	94.21(0.53)
Meta-SGD		91.76(0.96)	76.61(0.92)	91.29(0.34)		94.42(0.93)	84.88(0.96)	94.78(0.32)
`MultiFC`		95.19(1.86)	80.34(1.59)	**94.58(0.19)**		94.99(0.64)	84.22(0.53)	94.61(0.44)
`FCGrad`		94.02(1.90)	34.11(18.9)	71.78(0.86)		**96.16(0.71)**	**87.25(0.96)**	93.86(0.51)
`LSTMFC`		**95.90(1.02)**	**80.45(1.88)**	94.19(0.30)		94.33(0.85)	82.19(0.99)	**96.29(0.25)**

TABLE 6.3: Performance of Compression on Omniglot.

20-way 5-shot (20-5) are utilized. TransComp uses hidden size as 10 with ReLU activation. $\beta = 1.0$ in pruning. We train all methods for equal episodes such that all converge to stable performance. The performance of last 10 episodes is averaged with mean and variance as reported. As Table 6.3 shows: in quantization, variants of TransComp achieves the best performance in all cases. In quantization, 2-bit `MultiFC` and `LSTMFC` drop approximately 2% in 5-1 and 20-5, compared to full-precision training, while STE and Meta-SGD shows great gap. Similar results are shown in pruning: Although overperformance is not as significant as in quantization, TransComp is able to achieve the best performance in all settings. `LSTMFC` in 20-5 even climbs to less than 0.5 performance drop than full-precision training.

Method	Quantization			Pruning		
	bit	5-1	5-5	δ	5-1	5-5
Full-Precision	32	46.82(3.39)	63.24(1.81)	100	46.82(3.39)	63.24(1.81)
STE	8	42.07(1.93)	55.17(1.71)	30	42.89(1.34)	56.44(2.05)
Meta-SGD		43.12(1.78)	52.89(1.97)		42.44(2.03)	52.54(2.10)
`MultiFC`		**46.89(1.16)**	**65.02(1.92)**		43.03(1.58)	57.05(2.32)
`FCGrad`		46.03(2.43)	61.12(1.94)		43.08(2.25)	**58.05(1.66)**
`LSTMFC`		44.90(2.17)	64.56(1.86)		**43.77(2.45)**	53.22(2.23)

TABLE 6.4: Performance of Compression on MiniImageNet.

6.4.1.4 MiniImageNet

2 settings are used in MiniImageNet: 5-way 1-shot (5-1), 5-way 5-shot(5-5). Similarly, we set hidden size in TransComp as 20 with ReLU activation, and $\beta = 1.0$ in pruning. We compare variants of TransComp with baseline methods in 8-bit quantization and $\delta = 30$ pruning in Table 6.4. For quantization, TransComp shows significant out-performance than baseline methods, with at least 1.5% and 6% improvement in both settings. Especially, `MultiFC` can even achieve better performance than full-precision model. In pruning, TransComp outperforms baseline with a margin of 1%.

6.4.1.5 CIFAR100 → STL10

Besides transfer compression within the same dataset, we conduct experiments across dataset: We train meta compressor and $\boldsymbol{\Theta}$ from source tasks which are sampled from source datasets. And then apply the learned meta compressor and $\boldsymbol{\Theta}$ to novel tasks which are generated from novel / target dataset.

In this section, we use CIFAR100 as source dataset and STL10 as target dataset. During meta-training, for each task, I sample c categories from CIFAR100, with all the corresponding training instances $(500 \times c)$ and test instances $(100 \times c)$. We meta-train for 5 steps. Similarly in novel tasks, we sample c categories from STL10, each category contains 500 training instances and 800 test instances. During novel tasks' inner loop training, I train each task for 200 epoch and record the last 10 epochs' average performance as final performance. Since all training instances in novel tasks (STL10) are utilized, SD-Compression is not compared. Instead, I have include a baseline method, named "Source Pretrain", which trains novel tasks based on a trained initialization, pretrained by source datasets. We use FD-Compression as baseline and report relative performance for other methods.

Method	Quantization		Pruning	
	bit	RP	CR	RP
FD-Compression	2	0	10	0
Source Pretrain		5.50(6.08)		-3.28(4.05)
STE		14.32(4.57)		**17.74(2.98)**
Meta-SGD		14.70(3.94)		14.24(2.73)
`MultiFC`		16.87(4.08)		13.40(2.57)
`FCGrad`		11.04(4.04)		17.01(2.59)
`LSTMFC`		**17.05(4.29)**		13.38(2.83)

TABLE 6.5: Performance of compression on STL10 with $c = 5$.

Table 6.5 demonstrates experiments on CIFAR100 → STL10 using $c = 5$. In 2-bit quantization, `LSTMFC` achieved the best performance. In pruning, STE outperformed `FCGrad` by small margin.

I further conducted experiments on $c = 10$ 2-bit quantization on STL10. Table 6.6 shows the performance comparison. In this experiment, STE demonstrates slightly better performance than TransComp, though TransComp still achieved outperformance than naive compression (FD-Compression).

Method	PR
FD-Compression	0
Source Pretrain	1.49(1.73)
STE	**14.61(1.25)**
Meta-SGD	13.13(1.06)
`MultiFC`	13.89(1.00)
`FCGrad`	8.99(1.19)
`LSTMFC`	13.55(1.04)

TABLE 6.6: Performance of 2-bit quantization on STL10 with $c = 10$.

6.4.2 Inner Loop Training Analysis

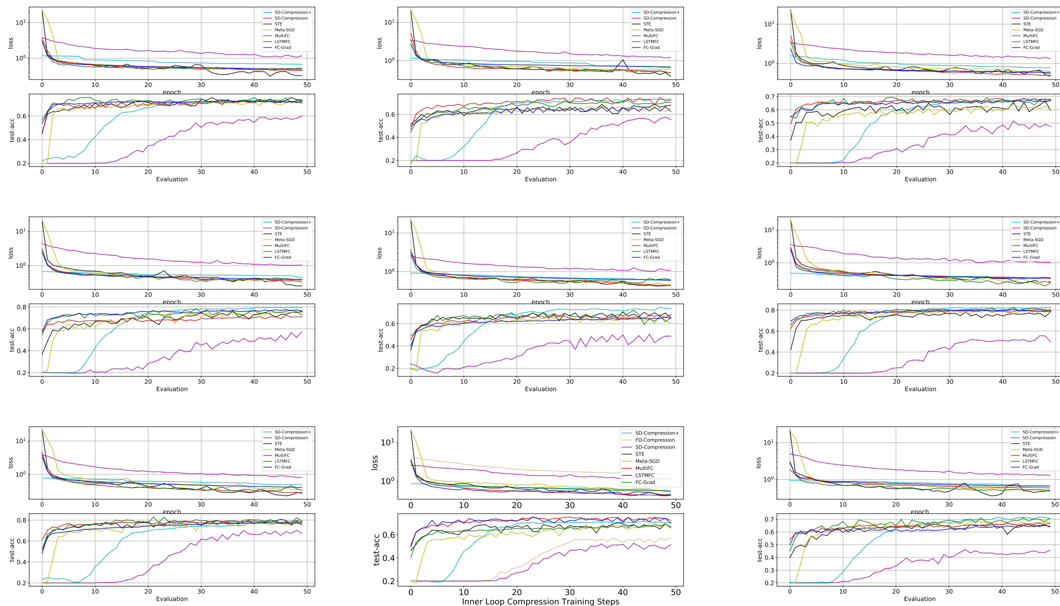


FIGURE 6.2: Adaptation Procedure of Selected Novel Tasks in CIFAR100.

We analyze the inner loop compression training procedure of TransComp and baseline methods in selected novel tasks. We further conduct SD-Compression on initialization pre-trained by source tasks set, denoted as 'SD-Compression+'. Based on the initialization and meta compressor brought by various TransComp and baseline methods, we train a compressed model until convergence. We apply novel tasks with $k_*^{tr} = 100, k_*^{tst} = 100$ on trained $\mathbf{\Theta}$ and $\mathcal{M}_\phi$ from $k_i^{tr} = 200, k_i^{tst} = 100$ of different methods. As Fig.6.2 shows, compared with SD-Compression/SD-Compression+, whose initialization is learned from scarce novel/source tasks data pretraining, meta-learning based method which utilizes knowledge from source tasks is able to achieve lower training loss and higher test performance in a very few

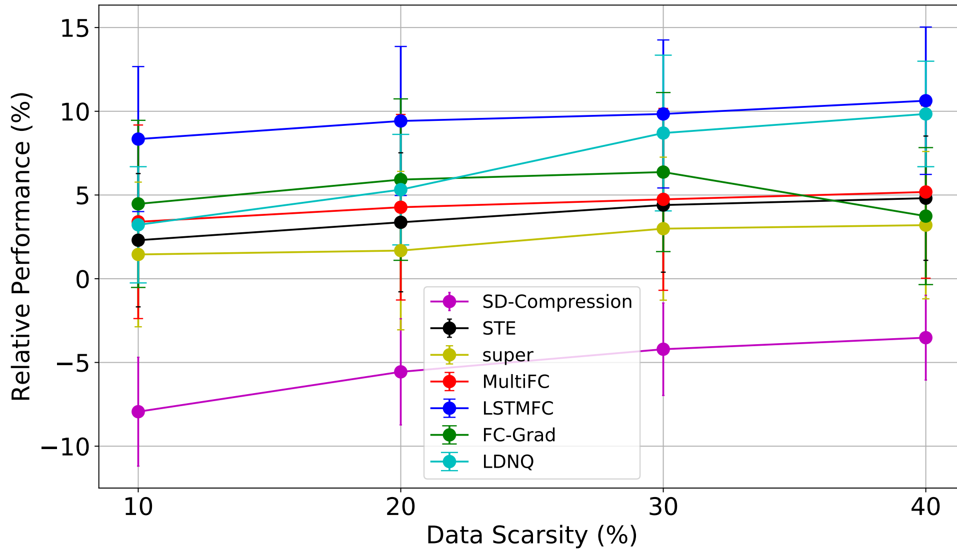


FIGURE 6.3: Relative Performance under different data scarcity of novel tasks.

steps. `LSTMFC` and `MultiFC` especially achieve better performance than STE/Meta-SGD by meta-learning compressor to guide the adaptation procedure.

6.4.3 Data Scarcity

We further analyze the performance of TransComp under different level of data scarcity in novel tasks. We construct novel tasks using 40%, 30%, 20%, 10% training data $\mathbf{D}_u^{tr}$ from full dataset and perform experiments on 2-bit quantization. We apply the trained $\mathbf{\Theta}$ and $\mathcal{M}_\phi$ trained by $c = 5, k_i^{tr} = 200, k_i^{tst} = 100$. As Fig.6.3 illustrates, performances of all methods decrease as number of training data declines. Variants of TransComp is able to maintain better performance than all baselines methods, with the drop in relative performance is less dramatic. Meta-learning based methods are able to achieve comparable performance to FD-Compression using only 10% of full training data. When data scarcity climbs to 40%, `LSTMFC` outperforms FD-Compression by more than 10% accuracy.

6.4.4 Convergence Analysis

We analyze the source tasks training procedure for all meta-learning based methods.

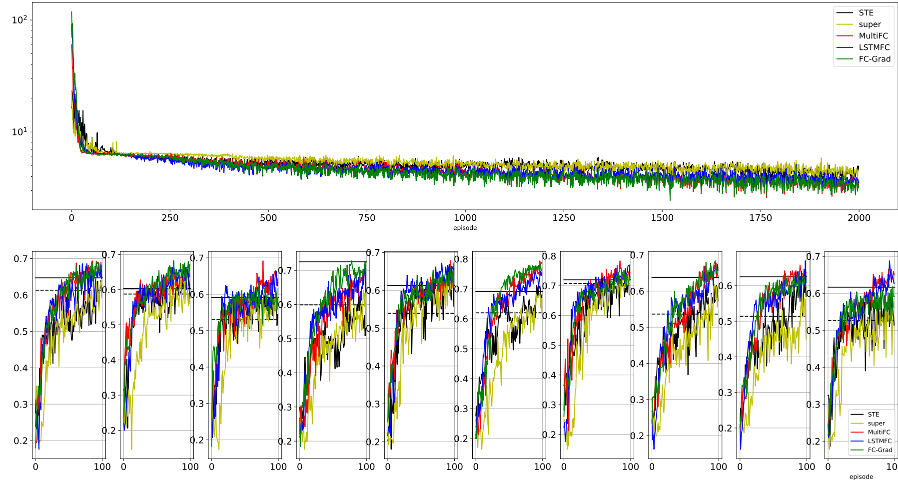


FIGURE 6.4: Source task loss and inner loop compression training performance in 10 selected novel tasks in 2-bit quantization on CIFAR100: $c = 5, k_i^{tr} = 100, k_i^{tst} = 100, k_*^{tr} = 100, k_*^{tst} = 100$.

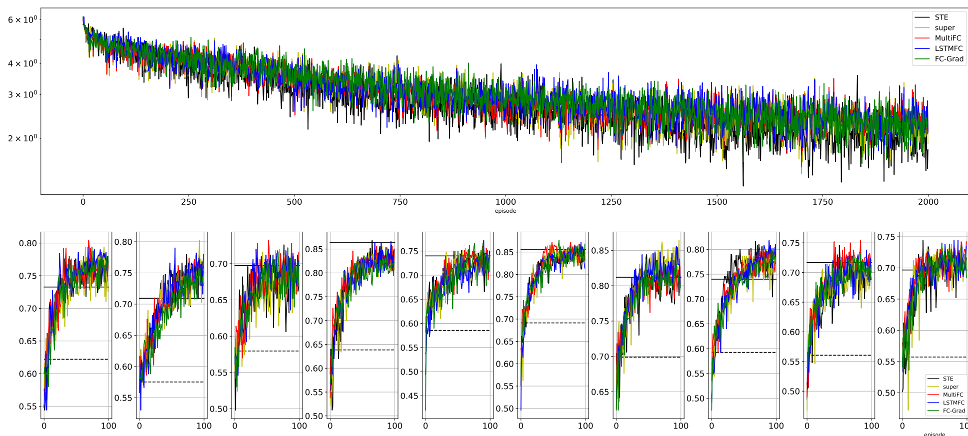


FIGURE 6.5: Source task loss and loop compression training performance in 10 selected novel tasks in $\delta = 30$ pruning on CIFAR100 with $c = 5, k_i^{tr} = 100, k_i^{tst} = 100, k_*^{tr} = 100, k_*^{tst} = 100$.

CIFAR10 Fig.6.4 and 6.5 illustrates the source training process and inner loop compression training performance of selected novel tasks in 2-bit quantization, $\delta = 30$ pruning, respectively. Source and novel tasks are sampled using $c = 5, k_i^{tr} = 100, k_i^{tst} = 100, k_*^{tr} = 100, k_*^{tst} = 100$.

Upper part shows that during source tasks training, variants of TransComp (colored by red, green, blue) achieve lower training loss than baseline methods (STE, Meta-SGD).

Lower part shows the inner loop compression training performance of different methods in 10 selected novel tasks: For each novel tasks, we first attain its maximum performance (FD-Compression) by training and compressing using all data, denoted as black solid line (absolute performance of FD-Compression in each novel task). Then we train and compress a minimum performance (SD-Compression) by using scarce data, in a similar way, as in black dash line (absolute performance of SD-Compression in each novel task). The color curves represent the change of inner loop compression training performance at $T_* = 10$, inner trained based on the initialization and meta compressor by different methods.

During source tasks training, all methods generate better initialization and meta compressor, which contributes to a higher performance after inner loop training on novel tasks. In quantization, for same training episode, variants of TransComp always achieve better inner loop training performance than STE and Meta-SGD. After certain episodes' training, TransComp climbs higher than SD-Compression and outperform FD-Compression in most tasks.

ImageNet

Fig.6.6 illustrate the source training process in 2-bit quantization experiments on ImageNet with $c = 10, k_i^{tr} = 200, k_i^{tst} = 50, k_*^{tr} = 200, k_*^{tst} = 50$. The novel tasks' performance is evaluated using $T_* = 40$ during training. Due to the long trace (20 update steps) of inner loop training, `MultiFC` and `FCGrad` tends to be similar as STE and Meta-SGD. However, `LSTMFC` demonstrates significant outperformance than the rest method. We assume it is due to the memory function in LSTM module.

Omniglot

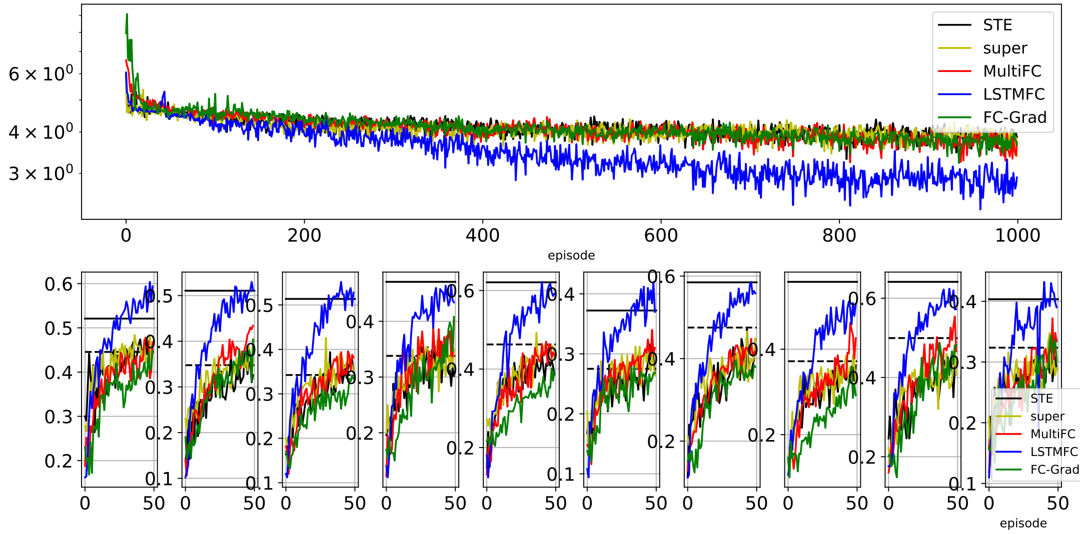


FIGURE 6.6: Source task test loss and loop compression training performance in 10 selected novel tasks in 2-bit quantization on ImageNet with $c = 10, k_i^{tr} = 200, k_i^{tst} = 50, k_*^{tr} = 200, k_*^{tst} = 50$.

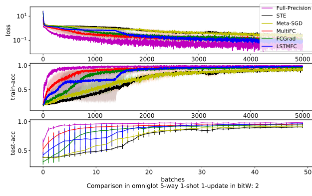


FIGURE 6.7: Omniglot-5-1

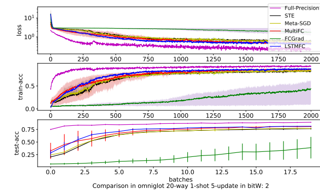


FIGURE 6.8: Omniglot-20-1

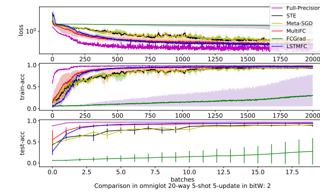


FIGURE 6.9: Omniglot-20-5

FIGURE 6.10: Convergence of various methods in 2-bit quantization omniglot training.

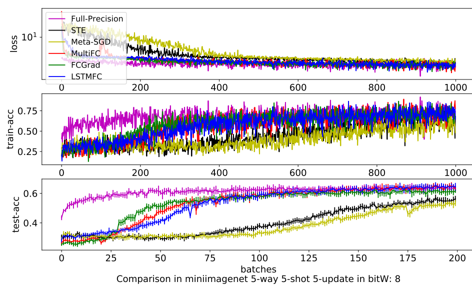


FIGURE 6.11: miniimagenet-5-1

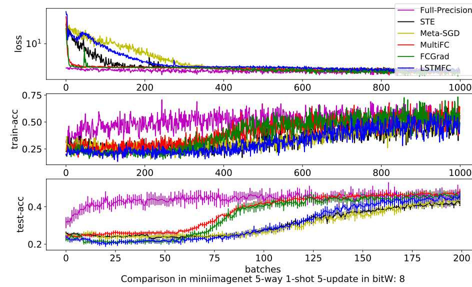


FIGURE 6.12: miniimagenet-5-5

FIGURE 6.13: Convergence of various methods in 2-bit quantization miniimagenet training.

Apart from producing better task-specific compressed model, TransComp is able to achieve faster convergence compared with other MAML-based baselines (STE, Meta-SGD). We record the trajectory of test data's loss in source tasks training phase and novel tasks' performance to show the quantization training of different MAML-based methods. We visualize the training trajectory of omniglot-5-1, 20-1, 20-5 in Fig.6.10, and miniimagenet-5-1, 5-5 in Fig.6.13. As figures shows, apart from the full-precision training, variants (except `FCGrad` in certain cases) of TransComp achieve much faster learning speed than STE and Meta-SGD.

6.4.5 Effect of Architecture in $\mathcal{M}_\phi$

In this section, we verify how hyper-parameters (hidden size, non-linear activation function) in $\mathcal{M}_\phi$'s architecture effect performance. We compare $\mathcal{M}_\phi$ with different architecture in 2-bit quantization of Omniglot 5-1 setting. As shown in Table 6.7, with proper hyper-parameter tuning, `MultiFC` is able to achieve nearly 2% boost in performance. Large hidden size and linear $\mathcal{M}_\phi$ can bring with improvement under certain cases.

Hidden Size	Non-linear Activation	Performance
10	ReLU	95.19(1.86)
100	ReLU	97.04(0.88)
10	None	97.10(0.78)

TABLE 6.7: Performance of 2-bit quantization in Omniglot 5-1 using `MultiFC`.

6.4.6 Interpretation of $\mathcal{M}_\phi$

Why TransComp is able to achieve better performance than baseline (STE, Meta-SGD)? Since all methods use a same initialization, the magic happens in $\mathcal{M}_\phi$. However, due to the non-linear property and high-dimension process in $\mathcal{M}_\phi$, interpretation of $\mathcal{M}_\phi$ is non-trivial. To understand the mechanism inside $\mathcal{M}_\phi$, we use `FCGrad` with linear activation as an example. After such simplification, $\mathcal{M}_\phi$ acts as an adjustor that multiply input gradient with a certain value. The change of amplification value is visualized as training progresses and shown in Fig.6.14: During source tasks training, amplification of each layer varies to decrease the training loss. Among them, amplification in lower layers (`features.0.0`, `features.1.0`,

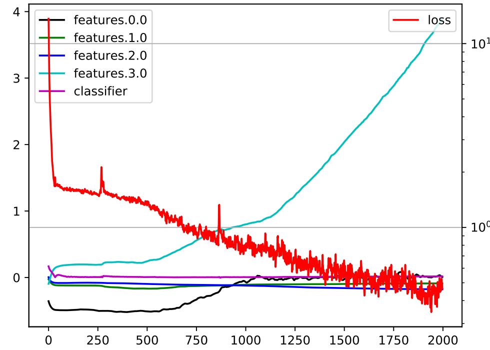


FIGURE 6.14: Amplification of $\mathcal{M}_\phi$ during training process. Left Y axis: magnitude of amplification after a linear $\mathcal{M}_\phi$; Right Y axis: training loss.

`features.2.0`) stay stable while in higher layers (`features.3.0`, `classifier`). This conforms to the observation in meta-learning [82] that general features is learnt in lower layers while task-specific knowledge are captured in higher layers.

6.5 Conclusion

We propose to solve problems of model compression on scarce data by meta-learning common knowledge from other similar tasks. This knowledge is captured by an initialization and a meta compressor. When a novel task arrive, it starts from the initialization and guided to learn by the meta compressor. Extensive experiments in scarce data and few shot setting demonstrates that our method is able to achieve better compressed model.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

This thesis summarizes my work over the past four years on model compression, under full data setting and scarce data scenario. Firstly, I given an introduction and motivation on model compression in Chapter 1: Modern neural network achieve significant performance, by the cost of millions of parameters and corresponding computation resources. Deployment of these neural network on edge devices requires both model storage optimization and inference acceleration. Moreover, it is challenging to produce a compressed model, especially when training data is insufficient.

Recent progress of model compression is surveyed in Chapter 2. I have categorized model compression methods into four main types: 1) Quantization 2) Pruning 3) Distillation 4) Architecture Modification. I have list a comprehensive illustration and comparison of various methods.

Chapter 3 illustrates a layer-wise method for model quantization under scarce data setting: Layer-wise output difference before and after quantization is minimized, which is formulated into a quadric programing with integer constraint. A cascade weights update method is introduced to further improve quantization performance. During the whole process, only scarce data (1% compared with the original training dataset) is required.

For scarce data setting in pruning, I have proposed cooperative pruning in Chapter 4: target task of scarce data is cooperatively trained and pruned with source task of abundant data. Knowledge from source task's pruning is transferred to target task by constructing target task's pruning mask. The mask construction is a weighted sum of current value from both source and target parameters.

I have made progress on training-based quantization in Chapter 5. I introduce a trainable meta neural network to provide gradient during quantization training, to circumvent the commonly used STE gradient estimator. The meta network is trained simultaneously with the based quantized model. The proposed meta quantization is able to achieve faster convergence and better performance in training-based quantization.

By incorporating meta quantization into meta learning, I proposed transferable compressor in Chapter 6, which comprises a meta neural network to provide gradient and meta initialization across tasks. The meta network and initialization is meta-trained in source tasks, then applied to novel tasks for learning and adaptation.

7.2 Future Work

7.2.1 Model Compression Co-design of Algorithm and Hardware

Algorithm of model compression has been making progress, however, there is still huge gap for algorithm deployment on hardware.

Fig.7.1 shows the inference comparison between unstructure pruning (CSR-specified multiplication) and original model (GEMM multiplication). Under the same dimension, original model achieves faster inference than unstructure pruning, though compression rate decreases. It demonstrates that it is hard for unstructure pruning (Chapter 4) to achieve acceleration on current computing framework (GPU).

Currently, most hardwares (CPU/GPU) support 8-bit integer multiplication, which enable actual acceleration for 8-bit quantization. However, algorithm has pushed to binarization (Chapter 5), which is far ahead of hardware support.

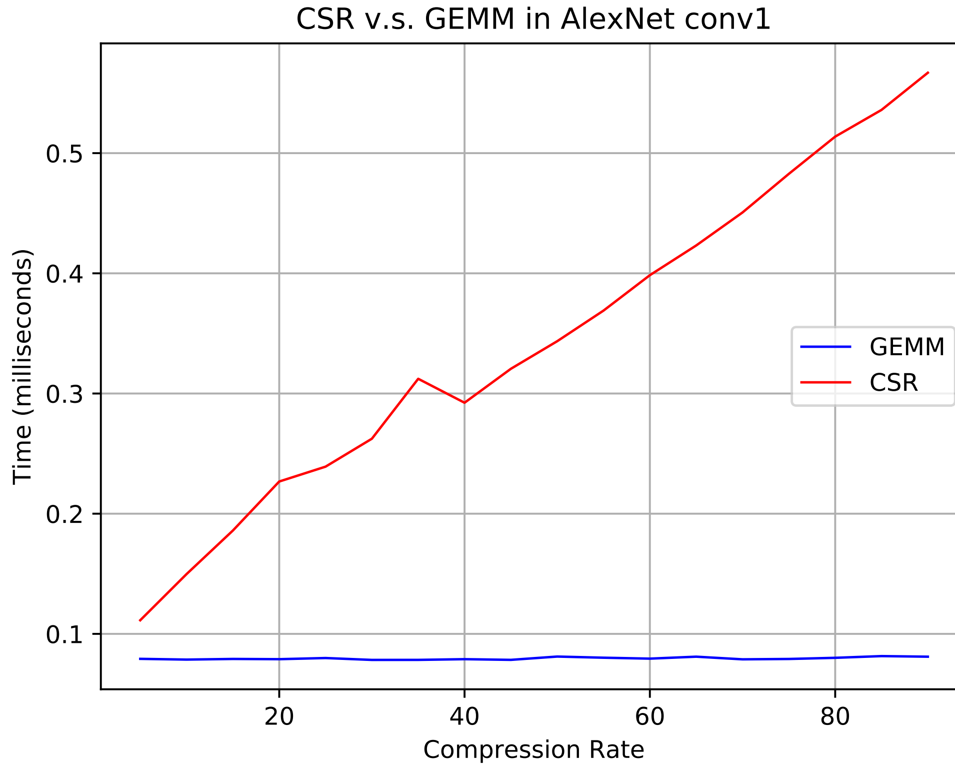


FIGURE 7.1: Inference speed comparison between unstructure pruning (CSR) and normal matrix multiplication(GEMM)

Under the mismatch of algorithm and hardware, a future work is to make alignment and design hardware-specific algorithm.

7.2.2 Data-Free Compression

Post-training/data-free compression benefits from its easy-to-use: Few or no data is required to perform compression. Chapter 3 illustrates a pioneered work on using limited data in model quantization. Chapter 6 incorporates meta-learning into model compression and enables model compression of novel tasks with scarce data.

Afterwards, [51], [83] has pushed data-free quantization to 8/4-bit. However, in few bit setting, post-training quantization still shows performance gap from training-based quantization.

A future work will be improving post-training quantization to even fewer bits, i.e. 3/2. Some techniques from meta-learning and transfer learning may be applied.

List of Author's Awards, Patents, and Publications

Conference Proceedings

- Xin Dong, **Shangyu Chen** and Sinno Jialin Pan, "Learning to Prune Deep Neural Networks via Layer-wise Optimal Brain Surgeon" in *31st Conference of Neural Information Processing Systems 2017 (NIPS-17).*

- **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "Deep Neural Network Quantization via Layer-Wise Optimization using Limited Training Data", in *33rd AAAI Conference on Artificial Intelligence (AAAI-19).*

- **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "Cooperative Pruning in Cross-Domain Deep Neural Network Compression", in *28th International Joint Conference on Artificial Intelligence (IJCAI-19).*

- **Shangyu Chen**, Wenya Wang and Sinno Jialin Pan. "MetaQuant: Learning to Quantize by Learning to Penetrate Non-differentiable Quantization", in *33rd Conference on Neural Information Processing Systems 2019 (NeurIPS-19).*

- Jianda Chen, **Shangyu Chen** and Sinno Jialin Pan. Storage Efficient and Dynamic Flexible Runtime Channel Pruning via Deep Reinforcement Learning, in *34rd Conference on Neural Information Processing Systems 2020 (NeurIPS-20).*

Bibliography

[1] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015. xix, 3, 9, 11, 15, 69, 80, 94

[2] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*, pages 2074–2082, 2016. xix, 3, 13, 14

[3] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision*, volume 2, page 6, 2017. xix, 15

[4] Xitong Gao, Yiren Zhao, Łukasz Dudziak, Robert Mullins, and Cheng-zhong Xu. Dynamic channel pruning: Feature boosting and suppression. *International Conference of Learning Representation*, 2018. xix, 16

[5] Juyong Kim, Yookoon Park, Gunhee Kim, and Sung Ju Hwang. Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization. In *International Conference of Machine Learning*, 2017. xix, 17, 18

[6] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. xix, 3, 20, 21, 22, 34, 79, 87

[7] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. Training and inference with integers in deep neural networks. *International Conference of Learning Representation*, abs/1802.04680, 2018. xix, 26, 27, 34

[8] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017. xix, 29, 30, 34, 54, 55, 57

[9] Frederick Tung and Greg Fraser. Clip-q : Deep network compression learning by in-parallel pruning-quantization. 2018. xix, 33

[10] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. *International Conference of Learning Representation*, abs/1802.05668, 2018. xix, 36

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105, 2012. 1, 37, 55

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

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1, 55, 69, 112, 113

[14] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017. 1

[15] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in Neural Information Processing Systems*, pages 4107–4115, 2016. 3, 20

[16] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. 3, 22, 34

[17] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *International Conference of Learning Representation*, 2017. 3, 23, 34, 54, 57, 93, 99

[18] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016. 3, 24, 34, 79, 87, 104

[19] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009. 4

[20] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017. 4, 102, 103, 111

[21] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016. 9, 69, 78, 79, 80, 87, 104

[22] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *International Conference of Learning Representation*, 2019. 10

[23] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems*, pages 598–605. Morgan-Kaufmann, 1990. 10, 11, 33, 42, 44, 64, 69, 80

[24] Babak Hassibi and David G Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems*, pages 164–171, 1993. 11, 42, 44, 46

[25] Xin Dong, Shangyu Chen, and Sinno Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Advances in Neural Information Processing Systems*, pages 4857–4867, 2017. 11, 41, 52, 69, 94

[26] Sejun Park, Jaeho Lee, Sangwoo Mo, and Jinwoo Shin. Lookahead: a far-sighted alternative of magnitude-based pruning. *International Conference of Learning Representation*, 2020. 12

[27] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *International Conference of Learning Representation*, 2019. 12

[28] Jonathan Frankle, Gintare Karolina Dziugaite, Daniel M Roy, and Michael Carbin. The lottery ticket hypothesis at scale. *arXiv preprint arXiv:1903.01611*, 2019. 12

[29] Ari Morcos, Haonan Yu, Michela Paganini, and Yuandong Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In *Advances in Neural Information Processing Systems*, pages 4933–4943, 2019. 13

[30] Hattie Zhou, Janice Lan, Rosanne Liu, and Jason Yosinski. Deconstructing lottery tickets: Zeros, signs, and the supermask. In *Advances in Neural Information Processing Systems*, pages 3592–3602, 2019. 13

[31] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *International Conference of Learning Representation*, 2017. 15

[32] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*, 2016. 15

[33] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *IEEE International Conference on Computer Vision*, pages 2755–2763. IEEE, 2017. 15

[34] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017. 15

[35] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *IEEE conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. 16

[36] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *British Machine Vision Conference*, 2015. 17

[37] Hao Zhou, Jose M Alvarez, and Fatih Porikli. Less is more: Towards compact cnns. In *European Conference on Computer Vision*, pages 662–677. Springer, 2016. 17

[38] Ji Liu, Przemyslaw Musialski, Peter Wonka, and Jieping Ye. Tensor completion for estimating missing values in visual data. *IEEE transactions on pattern analysis and machine intelligence*, 35(1):208–220, 2013. 17

[39] Mengye Ren, Andrei Pokrovsky, Bin Yang, and Raquel Urtasun. Sbnet: Sparse blocks network for fast inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8711–8720, 2018. 18

[40] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *Workshop in International Conference of Learning Representation*, 2015. 19

[41] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015. 20, 34

[42] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 20

[43] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *International Conference of Learning Representation*, 2016. 21, 22, 93

[44] Lu Hou and James T Kwok. Loss-aware weight quantization of deep networks. *International Conference of Learning Representation*, 2018. 22, 54, 57

[45] Wang K Chen Y Zhou A, Yao A. Explicit loss-error-aware quantization for low-bit deep neural networks. *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 24, 93

[46] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 5406–5414, 2017. 25, 32

[47] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *IEEE International Conference on Computer Vision*, pages 1026–1034, 2015. 28

[48] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018. 29, 54, 55

[49] Cong Leng, Hao Li, Shenghuo Zhu, and Rong Jin. Extremely low bit neural network: Squeeze the last bit out with admm. *AAAI Conference on Artificial Intelligence*, 2017. 30, 50, 54, 77

[50] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, January 2011. ISSN 1935-8237. doi: 10.1561/2200000016. URL `http://dx.doi.org/10.1561/2200000016`. 30, 48, 49

[51] Markus Nagel, Mart Van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. 30, 31, 125

[52] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv: Learning*, 2018. 31

[53] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different - recovering neural network quantization error through weight factorization. *arXiv: Learning*, 2019. 32

[54] Haroush Matan, Hubara Itay, Hoffer Elad, and Soudry Daniel. The knowledge within: Methods for data-free model compression. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 8491–8499, 2019. 32

[55] Cai Yaohui, Yao Zhewei, Dong Zhen, Gholami Amir, Michael Mahoney W., and Keutzer Kurt. Zeroq: A novel zero shot quantization framework. *IEEE Conference on Computer Vision and Pattern Recognition*, pages 13166–13175, 2020. 32

[56] Xiaofan Lin, Cong Zhao, and Wei Pan. Towards accurate binary convolutional neural network. In *Advances in Neural Information Processing Systems*, pages 344–352, 2017. 32

[57] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. *International Conference of Learning Representation*, 2017. 33

[58] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017. 34

[59] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the knowledge in a neural network. *CoRR*, abs/1503.02531, 2015. 35

[60] Yani Ioannou, Duncan Robertson, Roberto Cipolla, Antonio Criminisi, et al. Deep roots: Improving cnn efficiency with hierarchical filter groups. 2017. 37

[61] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018. 38

[62] Ting Zhang, Guo-Jun Qi, Bin Xiao, and Jingdong Wang. Interleaved group convolutions. *IEEE International Conference on Computer Vision*, pages 4383–4392, 2017. 38

[63] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 38

[64] Alireza Aghasi, Afshin Abdi, Nam Nguyen, and Justin Romberg. Net-trim: Convex pruning of deep neural networks with performance guarantee. In *Advances in Neural Information Processing Systems*, pages 3177–3186, 2017. 42

[65] Nikolas Wolfe, Aditya Sharma, Lukas Drude, and Bhiksha Raj. The incredible shrinking neural network: New perspectives on learning representations through the lens of pruning. 2016. 44

[66] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014. 54

[67] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011. 54

[68] Abhisek Kundu, Kunal Banerjee, Naveen Mellempudi, Dheevatsa Mudigere, Dipankar Das, Bharat Kaul, and Pradeep Dubey. Ternary residual networks. *arXiv preprint arXiv:1707.04679*, 2017. 54, 55

[69] Antonio Polino, Razvan Pascanu, and Dan Alistarh. Model compression via distillation and quantization. In *International Conference of Learning Representation*, 2018. 54, 57

[70] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference of Learning Representation*, 2015. 68, 79, 87

[71] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. 2009. 69

[72] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. 2007. 69

[73] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2):303–338, 2010. 69

[74] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 69

[75] Eric Tzeng, Judy Hoffman, Ning Zhang, Kate Saenko, and Trevor Darrell. Deep domain confusion: Maximizing for domain invariance. *arXiv preprint arXiv:1412.3474*, 2014. 69

[76] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016. 78, 82, 83, 86

[77] Yu Bai, Yu-Xiang Wang, and Edo Liberty. Proxquant: Quantized neural networks via proximal operators. *International Conference of Learning Representation*, 2018. 93

[78] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. Metaquant: Learning to quantize by learning to penetrate non-differentiable quantization. In *Advances in Neural Information Processing Systems*, pages 3918–3928, 2019. 102, 108

[79] Zhenguo Li, Fengwei Zhou, Fei Chen, and Hang Li. Meta-sgd: Learning to learn quickly for few-shot learning. *International Conference on Machine Learning*, 2017. 110

[80] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. Deep neural network quantization via layer-wise optimization using limited training data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3329–3336, 2019. 110

[81] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. Cooperative pruning in cross-domain deep neural network compression. *International Joint Conference on Artificial Intelligence*, 2019. 110

[82] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Advances in Neural Information Processing Systems*, pages 4077–4087, 2017. 122

[83] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*, pages 7948–7956, 2019. 125