On Designing Resource-Constrained CNNs Efficiently

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Ting-Wu Chin

B.S., Computer Science, National Chiao Tung University
M.S., Computer Science and Engineering, National Chiao Tung University

Carnegie Mellon University
Pittsburgh, PA

August 2021

© Ting-Wu Chin, 2021
All rights reserved.

Acknowledgements

Pursuing a doctorate degree in Electrical and Computer Engineering at Carnegie Mellon University is one of my best decisions in life. It has been a challenging, life-changing, wonderful, and fulfilling journey and it would not have been possible for me to begin writing this thesis without the tremendous support and guidance from my advisors, mentors, friends, and family.

First and foremost, I would like to express my best gratitude to my advisor, Prof. Diana Marculescu. Diana has been a truly amazing mentor to me and has never failed to support and encourage me throughout the journey of my program. Specifically, she has given me freedom in exploring new ideas and new directions for research, she has always been present in our weekly meetings to guide me along the way, she shared her connections with me, leading to an internship at Facebook Reality Labs, and she has always had my back when facing rejections and negative results. Dealing with rejections can be hard especially when they come in a row. One of my papers has faced four consecutive rejections before it was accepted at CVPR as an oral presentation. Diana has shown her great support in those hard times by letting me know that great work eventually gets recognized and our work is great. I am very fortunate and grateful to have Diana as my advisor.

I would also like to thank my co-advisor, Prof. Gauri Joshi, who has been willing to co-advise me when Diana moved from CMU to UT Austin. Gauri has been a great mentor who cares a lot about her students. The weekly meeting with Gauri's research group (OPAL) has been inspiring and fun amid the COVID-19 pandemic. I am very fortunate and thankful to be a part of OPAL and I greatly appreciate all the insightful feedback Gauri has given me to improve my work further. Specifically, I really appreciate the feedback of encouraging me to understand the problems from a more theoretical aspect and encouraging me to assess the problem of interest with varying metrics, which have played an important role in shaping my thoughts for my later projects.

I would like to thank both Dr. Cha Zhang and Prof. Virginia Smith for serving as my thesis committee members. I greatly appreciate the affirmations and the invaluable feedback throughout the process.

I would like to thank my mentors at industry, Dr. Cha Zhang from Microsoft, Dr. Pierce Chuang from Facebook Reality Labs, and Dr. Ari Morcos from Facebook AI Research. I am extremely grateful that Cha took me as a research intern at Microsoft even though I did not have relevant publication records at the time of application. Cha has contributed greatly to shaping my research skills throughout and after the internship and I aspire to be a great critical thinker like him. I am very grateful to get to know Pierce, who has taught me a lot in the field of network quantization and hardware implementation. Pierce has been a very friendly mentor and I am fortunate to learn from him. Last but not least, I would like to thank

Ari for being such a great and supportive mentor. Even though I have only worked with Ari in a virtual format, it does not stop me to feel strongly about Ari's empathetic, enthusiastic, and inclusive personality. Ari's curiosity has greatly influenced me to pay attention to the details of my empirical results and I am very grateful for Ari taking the time to guide me and provide me tips in my academic writings.

I would like to thank my collaborators: Dr. Zhuo Chen, Dr. Dimitrios Stamoulis, Dr. Ruizhou Ding, Dr. Zeye Liu, Dr. Dilin Wang, Mr. Ahmet Inci, Ms. Natasha Frumkin, and Mr. Yang Zhou, for the fruitful discussions and their contributions. I am very fortunate to work with all these talented people. Especially, I would like to thank Ruizhou for being a great mentor and collaborator, who has contributed a lot in helping me improve my research skills during the early stage of my PhD.

I would like to thank all the friends and colleagues I have met along the way who have made my life as a PhD student filled with fun and energy: Dr. Kai-Chiang Wu, Dr. Da-Cheng Juan, and Dr. Ermao Cai from EnyAC. Mr. Jianyu Wang, Mr. Ankur Mallick, Mr. Samarth Gupta, Mr. Tuhinangshu Choudhury, Ms. Yae Jee Cho, and Mr. Divyansh Jhunjhunwala from OPAL. Dr. Chieh Lo, Dr. Kartikeya Bhardwaj, Dr. Kai-Chun Lin, Dr. Xi He, Dr. Qicheng Huang, Dr. Chenlei Fang, Mr. Ching-Yi Lin, Mr. Tyler Vuong, Mr. Yi-Chung Lin, and Mr. Jin-Dong Dong from CMU ECE.

Additionally, I would like to thank all members of CMU Street Styles and CMU Dancer's Symposium, and special thanks to Mr. Daniel See and Mr. Randal Miller for getting me into knowing the dance scene at Pittsburgh. Dancing with you all was fun and amazing and it has been very helpful for me to get out of the stress incurred during my PhD.

I would also like to acknowledge the funding support received from National Science Foundation, Pittsburgh Supercomputing Center, and Carnegie Mellon University. They have been critical in making my research possible.

Lastly, I sincerely thank my wife, Hsuan-Yi Hsu, who has been a critical support for me throughout the entire process. Hsuan-Yi has been a truly amazing, empathetic, and supportive partner. It took tremendous courage and love for her to move away from her family and life just to accompany me to the United States to pursue a doctorate degree and I can not thank her enough for her dedication and understanding along the way.

Abstract

Deep Convolutional Neural Networks (CNNs) have been adopted in many computer vision applications to achieve high performance. However, the growing computational demand of CNNs has made it increasingly difficult to deploy state-of-the-art CNNs onto resource-constrained platforms. As a result, model compression/acceleration has emerged to be an important field of research. In this thesis, we intend to make CNNs more friendly for resource-limited platforms from two perspectives. The first perspective is to introduce novel ways of compressing/accelerating CNNs and the second perspective is to reduce the overhead of existing methodologies for constructing resource-constrained CNNs.

In the first perspective, we propose one novel technique for model acceleration and another for model compression. First, we propose *AdaScale* which is an algorithm that automatically scales the resolution of input images to improve both the speed and accuracy of a video object detection system. Second, we identify the *Winning-Bitwidth* phenomenon, where we found some weight bitwidth is more efficient than others for model compression when the filter counts of the CNNs are allowed to change.

In the second perspective, we propose three novel algorithms for accelerating existing filter pruning methods for constructing resource-constrained CNNs. First, we propose LeGR, an algorithm that aims to learn a global ranking among filters of a pre-trained CNN so that compressing the CNN to different target constraint levels using filter pruning can be done efficiently by greedily pruning the filters following the learned ranking. Second, we improve upon LeGR and propose *Joslim*, which is an algorithm that trains a CNN from scratch by jointly optimizing its weights and filter counts such that the trained CNN can be pruned without fine-tuning. Joslim improves upon LeGR in terms of efficiency as LeGR requires the pruned models to be fine-tuned to be usable. Lastly, we propose *Width Transfer*, which improves the efficiency for filter pruning methods that are derived from a neural architecture search perspective. Width Transfer assumes that the optimized filter counts are regular across depths and widths of a CNN architecture and are invariant to the size and the resolution of the training dataset. As a result, Width Transfer performs neural architecture search for filter counts by solving a proxy problem that has a much lower overhead.

Contents

Contents vi

List of Tables viii

List of Figures x

1 Introduction 1

1.1 Contributions . . . 2

1.2 Thesis Organization . . . 4

2 Background 5

2.1 Convolutional Neural Networks (CNNs) . . . 5

2.2 Model Compression/Acceleration . . . 6

3 AdaScale: Scale Inputs Adaptively for Improved Speed and Accuracy 10

3.1 Motivation . . . 10

3.2 Adaptive Scaling . . . 12

3.3 Experiments . . . 14

3.4 Discussion . . . 22

3.5 Carbon Footprint Analysis . . . 22

4 Winning-Bitwidth: Beyond Quantization for Fixed CNNs 23

4.1 Motivation . . . 23

4.2 Experiments . . . 25

4.3 Discussion . . . 34

4.4 Carbon Footprint Analysis . . . 35

5 LeGR: Towards Efficient Filter Pruning 36

5.1 Motivation . . . 36
5.2 Learned Global Ranking . . . 38
5.3 Experiments . . . 42
5.4 Ablation Study . . . 47
5.5 Discussion . . . 49
5.6 Carbon Footprint Analysis . . . 49

6 Joslim: Efficient Filter Pruning without Fine-tuning **50**
6.1 Motivation . . . 50
6.2 Methodology . . . 52
6.3 Experiments . . . 57
6.4 Discussion . . . 62
6.5 Carbon Footprint Analysis . . . 62

7 Width Transfer: On the (In)variance of Filter Count Optimization **64**
7.1 Motivation . . . 64
7.2 Width Optimization . . . 67
7.3 Approach . . . 68
7.4 Experiments . . . 70
7.5 Discussion . . . 76
7.6 Carbon Footprint Analysis . . . 76

8 Synergies and Discussion of Presented Approaches **77**
8.1 LeGR, Joslim, and Width Transfer . . . 77
8.2 Robustness to Distributional Shifts . . . 81
8.3 Relation to Neural Architecture Search . . . 81
8.4 Applicability to Networks besides CNNs . . . 82

9 Related Work **84**
9.1 Model Compression/Acceleration . . . 84
9.2 Efficient Model Compression . . . 86
9.3 Object Detection . . . 87

10 Conclusions **90**

Bibliography **93**

A Appendix for Chapter 4 **110**

A.1 Network Architectures 110

A.2 Proof For Proposition 4.2.1 111

B Appendix for Chapter 5 **113**

B.1 Optimization Interpretation Of LeGR 113

B.2 LeGR-DDPG 114

C Appendix for Chapter 6 **116**

C.1 Width Parameterization 116

C.2 Width Differences 117

C.3 Training Hyperparameters 117

C.4 Theoretical Analysis For Temporal Sharing 118

C.5 Inference Memory Footprint Calculation 119

C.6 Comparisons With AutoSlim 120

List of Tables

3.1 Evaluation of the proposed method. We denote methods by their approach of training and testing, *e.g.*, MS/SS stands for multi-scale (MS) training and single-scale (SS) testing. Blue text and red text indicate ≥ 1 AP improvement and degradation compared to SS/SS, respectively. . 16

3.2 mAP and runtime for different multi-scale training settings. 20

3.3 mAP and runtime for different regressor architectures. 22

4.1 Quantizing depth-wise convolution introduces large accuracy degradation across model sizes. $\Delta Acc_Q = Acc_{1bit} - Acc_{4bit}$ denotes the accuracy introduced by quantization and $\Delta Acc_G = Acc_{1bit,2\times} - Acc_{1bit}$ denotes the accuracy improvement by increasing channel counts. The CNN is VGG variant C with and without quantizing the depth-wise convolutions from 4 bits to 1 bit. 30

4.2 bitwidth ordering for MobileNetV2 and ResNet50 with *the model size aligned to the* 0.25× *8 bits models* on ImageNet. Each cell reports the top-1 accuracy of the corresponding model. The trend for the optimal bitwidth is similar to that of CIFAR-100 (4 bit for MobileNetV2 and 1 bit for ResNet). 33
4.3 The optimal bitwidth selected in Table 4.2 is indeed better than 8 bit when scaled to larger model sizes and more surprisingly, it is better than mixed-precision quantization. All the activations are quantized to 8 bits. 34

5.1 Comparison with prior art on CIFAR-10. We group methods into sections according to different FLOP counts. Values for our approaches are averaged across three trials and we report the mean and standard deviation. We use boldface to denote the best numbers and use * to denote our implementation. The accuracy is represented in the format of *pre-trained* ↦ *pruned-and-fine-tuned*. 45
5.2 Summary of pruning on ImageNet. The sections are defined based on the FLOP count left. The accuracy is represented in the format of *pre-trained* ↦ *pruned-and-fine-tuned*. 46

6.1 Comparing the top-1 accuracy among Slim, BigNAS, and Joslim on ImageNet. Bold represents the highest accuracy of a given FLOPs. 59

7.1 Compound width transfer for other CNNs. Width optimization overhead measured with 8 NVIDIA V100 GPUS on a single machine. 74

8.1 Comparing the overhead of different channel searching methods. WT stands for Width Transfer. 78

A.1 ResNet20 to ResNet56 110
A.2 Inv-ResNet26 111
A.3 VGGs 112

List of Figures

2.1 Finding best a_i for different precision values empirically through simulation using Gaussian with various σ^2. 9

3.1 Examples where down-sampled images have better detection results. Blue boxes are the detection results, and the numbers are the confidence. The detector is trained on a single scale (pixels of the shortest side) of 600. Column (a) and (c) are tested at scale 600. Column (b) is tested at scale 240 and column (d) is tested at scale 480. 11

3.2 The AdaScale methodology. 12

3.3 Optimal scale determination. First, the same number of predicted foregrounds from four scales are selected as $A_{m,i}$. Then, the scale with the lowest loss $\hat{L_i^m}$ is selected as the optimal scale. . . 13

3.4 The scale regressor module. 13

3.5 Precision-Recall curves for categories that MS/AdaScale has (a)(b)(c) better performance, (d) on-par performance, and (e)(f) worse performance compared to SS/SS. 18

3.6 Normalized true positives and false positives for different methods across all the images in validation set for three selected categories. 19

3.7 mAP and speed comparison with prior art on ImageNet VID dataset. Applying our AdaScale to RFCN [35], DFF [216] and SeqNMS [65] can further improve both speed and accuracy. 20

3.8 Comparing the results of SS/SS and MS/AdaScale qualitatively. Column (a) and (c) are results produced by SS/SS; column (b) and (d) are results produced by MS/AdaScale. The scales used in MS/AdaScale are labeled in black rectangle with white text. 21

3.9 The investigation of the dynamics of AdaScale. The scales of the images are labeled in bottom-right. 21

3.10 The regressed scale distribution of AdaScale tested on ImageNet VID validation set. (a)-(d) use different S_{train}. 22

4.1 Some bitwidth is consistently better than other bitwidths across model sizes. $\mathcal{C}_{size}$ denotes model size. xWyA denotes x-bit weight quantization and y-bit activation quantization. The experiments are done on the CIFAR-100 dataset. For each network, we sweep the width-multiplier to cover points at multiple model sizes. For each dot, we plot the mean and standard deviation of three random seeds. The standard deviation might not be visible due to little variances. 27

4.2 The optimal bitwidth for ResNet26 changes from 1 bit (a) to 4 bit (b) when the building blocks change from basic blocks (c) to inverted residual blocks (d). $\mathcal{C}_{size}$ in (a) and (b) denotes model size. (C_{out}, C_{in}, K, K) in (c) and (d) indicate output channel count, input channel count, kernel width, and kernel height of a convolution. 28

4.3 The optimal bitwidth for VGG shifts from 1 bit to 4 bit as more convolutions are replaced with depth-wise separable convolutions (DWSConv), *i.e.*, from (a) to (c). Variant A, B, and C have 30%, 60%, and 90% of the convolution layers replaced with DWSConv, respectively. As shown in (d), the optimal bitwidth changes back to 1 bit if we only quantize point-wise convolution but not depth-wise convolutions. 29

4.4 Visualization of our accuracy decomposition, which is used for analyzing depth-wise convolutions. 30

4.5 The average estimate Var($|\bar{\boldsymbol{w}}|$) for each depth-wise convolution under different $d = (C_{in} \times K_w \times K_h)$ values. 31

4.6 d negatively correlates with the variance and positively correlates with the accuracy difference induced by quantization $\Delta Acc_Q = Acc_{1bit} - Acc_{4bit}$. 32

5.1 Using filter pruning to optimize CNNs for embodied AI applications. Instead of producing one CNN for each pruning procedure as in prior art, our proposed method produces a set of CNNs for practitioners to efficiently explore the trade-offs. 37

5.2 The flow of LeGR-Pruning. $\|\Theta\|_2^2$ represents the filter norm. Given the learned layer-wise affine transformations, *i.e.*, the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair, LeGR-Pruning returns filter masks that determine which filters are pruned. After LeGR-Pruning, the pruned network will be fine-tuned to obtain the final network. 40

5.3 (a) The trade-off curve of pruning ResNet-56 and MobileNetV2 on CIFAR-100 using various methods. We average across three trials and plot the mean and standard deviation. (b) Training cost for seven CNNs across FLOP counts using various methods targeting ResNet-56 on CIFAR-100. We report the average cost considering seven FLOP counts, *i.e.*, 20% to 80% FLOP count in a step of 10% on NVIDIA GTX 1080 Ti. The cost is normalized to the cost of LeGR. 43
5.4 Results for ImageNet. LeGR is better or comparable compared to prior methods. Furthermore, its goal is to output a set of CNNs instead of one CNN. 45
5.5 Results for Bird-200. 46
5.6 Robustness to the hyper-parameter $\hat{\zeta}_l$. Prior art is plotted as a reference (c.f. Figure 5.3a). 47
5.7 Pruning ResNet-56 for CIFAR-100 with LeGR by learning $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ using different $\hat{\tau}$ and FLOP count constraints. 48
5.8 Latency reduction *vs.* FLOP count reduction. FLOP count reduction is indicative for latency reduction. 48

6.1 Schematic overview comparing our proposed method with existing alternatives and channel pruning. Channel pruning has a fundamentally different goal compared to ours, *i.e.*, training slimmable nets. Joslim jointly optimizes both the widths and the shared weights. 52
6.2 Comparisons among Slim, BigNAS, and Joslim. C10 and C100 denote CIFAR-10/100. We perform three trials for each method and plot the mean and standard deviation. 58
6.3 Comparisons among Slim, BigNAS, and Joslim on ImageNet. 59
6.4 A latency-*vs.*-error view of Fig. 6.3a. 59
6.5 Prediction error *vs.* inference memory footprint for MobileNetV2 and ResNet18 on ImageNet. . 59
6.6 Ablation study for the introduced binary search and the number of gradient descent updates per full iteration using ResNet20 and CIFAR-100. Experiments are conducted three times and we plot the mean and standard deviation. 60

7.1 The top row shows the conventional width optimization approach, which takes a training dataset and a seed network, and outputs a network with optimized widths. The bottom row depicts our idea of width transfer, where width optimization operates on the down-scaled dataset and seed network. We then use a simple function to extrapolate the optimized architecture to match the original network. Compared to direct width optimization, our empirical findings suggest that width transfer has similar performance, but has the benefit of drastically lower overhead. 65
7.2 The two width optimization strategies proposed in prior art. 67

7.3 An example for extrapolation. The projected network has fewer layers and channel counts per layer compared to the original network. After width optimization on the projected network, we propose two methods, *i.e.*, stack-last-block and stack-average-block, to match the layer counts to the original network. Finally, we match the FLOPs to the original network with a width multiplier. 68

7.4 Experiments for width transfer under network projection. We plot the ImageNet top-1 accuracy for uniform baseline, width transfer, and direct optimization (the leftmost points). On the x-axis, we plot the width optimization overhead saved by using width transfer. 71

7.5 The average optimized width for ResNet18 and MobileNetV2. They are averaged across the optimized widths. We plot the mean in solid line with shaded area representing standard deviation. 72

7.6 We compare the two layer-stacking strategies using DMCP for both ResNet18 and MobileNetV2. We can observe that both stack-average-block and stack-last-block perform similarly. 73

7.7 Experiments for width transfer under dataset projection. We plot the ImageNet top-1 accuracy for uniform baseline, width transfer, and direct optimization (the leftmost points). On the x-axis, we plot the width optimization overhead saved by using width transfer. 74

7.8 Width transfer with compound projection. 75

7.9 Comparing the proposed using DMCP with width transfer, DMCP, and network slimming. . . 76

8.1 (a,c) The trade-off curve of pruning ResNet-56 on CIFAR-100 using various methods. (b,d) Training cost and its scaling with respect to the number of target compression ratios for different methods targeting ResNet-56 on CIFAR-100. The cost is calculated using the number of forward passes for a ResNet-56 while approximating one backward pass as two forward passes. Fig. a and b are for optimizing many target networks while Fig. c and d are for optimizing weight-sharing networks. 79

8.2 The mean corruption errors for different number of target compression ratios of different methods targeting ResNet-56 on CIFAR-100. Note that the networks are optimized with CIFAR-100 and test using CIFAR-100-C. 80

B.1 Comparison between searching the layer-wise filter norms and searching the layer-wise filter percentage. (a) compares the searching progress for 50% FLOP count ResNet-56 and (b) compares the final performance for ResNet-56 with various constraint levels. 115

C.1 Comparing the width-multipliers between Joslim and Slim. The title for each plot denotes the relative differences (Joslim - Slim) and the numbers in the parenthesis are for Joslim. 117

C.2 Comparing Joslim and AutoSlim on ResNet18. Since ResNet18 has similar FLOPs across different layers, greedy pruning used by AutoSlim perform comparably to Joslim. However, Joslim outperforms AutoSlim when it comes to optimizing for memory consumption since the greedy pruning procedure adopted by AutoSlim is not multi-objective. 120

Chapter 1

Introduction

Convolutional neural networks (CNNs) have become the state-of-the-art model for tasks in vision, including recognition [68], detection [149], segmentation [21], image generation [85], and tasks that involve multiple modalities, such as visual question answering [3], visual-language navigation [2], and image generation based on language cues [142]. A common trend among the advances in these various tasks is to scale up the CNNs in terms of model sizes to achieve increased predictive power [68, 81, 199]. While it is effective in improving the predictive performance, it makes deployment of these CNNs onto resource-constrained devices such as embedded platforms or mobile phones increasingly challenging. Deploying CNNs to resource-constrained devices has gained surged interests from the research community [41]. Directly running CNNs on edge devices allows the inputs (*e.g.*, images, texts, or sound waves) to stay locally on the devices. This is in stark contrast to the Machine-Learning-as-a-Service paradigm where one uploads inputs to the cloud and receives the predicted outputs from the cloud. Deploying models onto edge devices has the following advantages:

- Better privacy as the user data never leaves their devices.
- Better control over the latency of the service as it is not dependent on the network connectivity.

While the advantages are desirable, deploying CNNs onto resource-constrained devices also poses several challenges:

- Embedded devices are usually battery-powered while CNNs are power-hungry [165, 188]. Hence, it is desirable to reduce the power consumption of CNNs.
- Embedded devices have low memory capacity while CNNs introduce large memory footprint [33, 98]. Hence, it is desirable to reduce the memory consumption of CNNs.

- Embedded devices have limited compute capability while CNNs are compute intensive [68, 54]. Hence, it is desirable to reduce the number of operations for CNNs to meet the latency constraints posed by real-time applications.

Toward addressing these challenges, model compression [63, 33, 99] and hardware-aware neural architecture search (NAS) [159, 183, 37, 16, 15, 133] have emerged to be active fields of research. While methods from model compression and hardware-aware NAS have introduced better ways to meet the resource-constraints of the embedded devices without losing much accuracy, they typically have large design-time overhead. Specifically, most approaches treat compression as a hyperparameter optimization problem, which requires large runtimes. Additionally, many approaches look for a specific solution that meets the resource constraints posed by a certain device, which makes the solution not reusable for other constraint levels. This makes it costly for designing models that will be deployed to many different devices with different constraint levels.

1.1 Contributions

In light of the aforementioned challenges in deploying CNNs onto resource-constrained devices, in this thesis we propose novel ways to compress and accelerate CNNs and novel ways to accelerate the model compression process.

Novel methods for model compression and acceleration

- **Chapter 3 – AdaScale for Improved Video Object Detection**: Complementary to conventional approaches for model compression and acceleration such as pruning [99], quantization [33, 86], and neural architecture search [162], we propose AdaScale, which adapts input resolution for different images to achieve better speed and accuracy. More specifically, we propose to learn the optimal resolution for the current frame and apply the output resolution for the next frame in a video object detection setting. We show that the proposed approach can improve R-FCN object detector [35] by *1.3 mAP* and achieve 1.6× speedup on the ImageNet VID dataset [152]. This chapter is published at the Conference of Machine Learning and Systems (MLSys) 2019 [24].

- **Chapter 4 – Tuning Channel Counts for Better Weight Quantization**: We identify a novel finding that weight quantization [98] benefits from some neural architectures more than others. More specifically, as opposed to studying weight quantization by fixing the neural architecture of the model to be quantized, we propose to study the setting where the channel counts of a neural architecture

can be altered when considering weight quantization. Under this setting, we show that even a single bitwidth throughout the network can outperform mixed precision (*i.e.*, having different weight bitwidths for different layers) quantization methods that do not take network architectures into considerations. Additionally, we characterize how weight quantization depends on neural architectures. Specifically, we show that the quantization error negatively correlates with the fan-in channel counts for the convolutional layers being quantized. Quantitatively, one can improve the top-1 accuracy by 1.8% for MobileNetV2 [154] on the ImageNet dataset [152] when jointly considering channel counts and weight quantization compared to conventional mixed precision quantization methods at the same model size. This chapter is published at the European Conference on Computer Vision Workshops (ECCVW) 2020 [23] and has received the best paper award at the Embedded Vision Workshop.

Novel methods for accelerating model compression

- **Chapter 5 – Learned Global Ranking for Efficient Filter Pruning**: While filter pruning is one of the dominant approaches for compressing and accelerating CNNs [99], discovering which filters to prune to lead to good performance often requires costly searching procedures [71, 59, 60, 193, 115]. More importantly, the results of the search procedures can not be reused across different levels of compression ratios, which makes filter pruning less scalable if targeting many compression ratios. In light of this inefficiency, we proposed LeGR, which learns a ranking of all the filters in a pre-trained CNN to determine which filters should be pruned first greedily. We show empirically that the proposed algorithm prunes networks up to 3$\times$ faster than prior work while having comparable or better performance when targeting seven pruned ResNet-56 [68] with different accuracy and computational requirement profiles on the CIFAR-100 dataset [93]. Moreover, the performance of the pruned ResNet-50 [68] and MobileNetV2 [154] achieve accuracy that are comparable to the state-of-the-art on the ImageNet dataset [152]. This chapter is published at the Conference on Computer Vision and Pattern Recognition (CVPR) 2020 [25] as oral presentation.

- **Chapter 6 – More Efficient Filter Pruning without Fine-tuning**: While LeGR [25] finds the pruned architectures fast across different compression ratios, each of the pruned models still requires a fine-tuning procedure to recover its accuracy, which can still be costly for certain applications. To remove fine-tuning altogether, we propose Joslim, which aims to find jointly the many filter configurations and the weights shared among the configurations using a unified optimization procedure. We show empirically that the proposed method outperforms existing alternatives that aims at the same goal

on the ImageNet dataset [152] across modern CNNs such as ResNet-18 [68] and MobileNetV2 [154]. This chapter is published at the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD) 2021 [27].

- **Chapter 7 – Efficient Filter Counts Optimization**: Treating filter pruning as a neural architecture search problem has gained increased interest [116]. However, existing approaches for doing so require an overhead equivalent to 2× training time of the input model. This can be costly when the input model is large. To reduce such an overhead, we propose Width Transfer, which performs existing algorithms on a more efficient proxy problem. More specifically, we propose to project the CNNs to be optimized to a shallower and narrower instance and project the training dataset to another with smaller resolution and fewer samples. Then, we perform existing algorithms to find optimal filter counts in the proxy setting. Finally, we transfer the found solution to the original space. We show empirically that the proposed procedure can achieve up to 320× overhead reduction without compromising the top-1 accuracy improvement obtained by performing filter counts optimization on the ImageNet dataset [152] for various modern CNNs. This chapter is published at the Conference on Computer Vision and Pattern Recognition Workshops (CVPRW) 2021 [26].

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides the necessary background for the thesis. Chapter 3 and Chapter 4 introduce novel methods for model acceleration and compression. Chapter 5 introduces a novel method, dubbed LeGR, for accelerating the model compression process that targets many compression ratios. Chapter 6 introduces a novel method to further get rid of the fine-tuning overhead in LeGR. Chapter 7 introduces a novel architecture transfer method for accelerating the process of filter counts optimization. Chapter 8 provides discussions for the proposed methods, their relationship to neural architecture search, and their applicability to architectures besides CNNs. Chapter 9 discusses the related work. Finally, Chapter 10 concludes this thesis and discusses future research directions.

Chapter 2

Background

2.1 Convolutional Neural Networks (CNNs)

Convolutional neural networks, or CNNs, are fundamental building blocks for state-of-the-art computer vision algorithms. At its core, a CNN is a deep neural network that is built by stacking many consecutive convolutional layers. A convolution operator is defined as follows:

$$(\boldsymbol{W} * \boldsymbol{X})_{i,j} \stackrel{\text{def}}{=} \sum_{m \in [K_w]} \sum_{n \in [K_h]} \boldsymbol{W}_{m,n} \boldsymbol{X}_{i+m-\frac{K_w}{2}, j+n-\frac{K_h}{2}} \quad \forall\, i, j, \tag{2.1}$$

where $*$ denotes the convolution operator, $\boldsymbol{W} \in \mathbb{R}^{K_w \times K_h}$ denotes the weights of the convolutional layers where K_w and K_h are the kernel size of the convolution, $\boldsymbol{X} \in \mathbb{R}^{W \times H}$ denotes the input tensor where H and W are the height and width of the input image, and (i, j) are 2-D indices of the location to be convolved. With this notion, we can define several types of convolution adopted in modern deep neural networks.

Standard convolution A standard convolutional layer $\boldsymbol{O} = Conv(\boldsymbol{X}, \boldsymbol{W})$ takes in an input tensor $\boldsymbol{X} \in \mathbb{R}^{C_{in} \times W \times H}$, has a trainable weight tensor $\boldsymbol{W} \in \mathbb{R}^{C_{out} \times C_{in} \times K_w \times K_h}$, and has an output tensor $\boldsymbol{O} \in \mathbb{R}^{C_{out} \times W \times H}$,

$$Conv(\boldsymbol{W}, \boldsymbol{X})_i \stackrel{\text{def}}{=} \sum_{c \in [C_{in}]} \boldsymbol{W}_{i,c} * \boldsymbol{X}_c \quad \forall\, i \in [C_{out}], \tag{2.2}$$

where C_{in} denotes the number of input channels and C_{out} denotes the number of output channels or also known as the number of filters. The number of floating-point operations (FLOPs) required for a forward pass of a standard convolution is given by $W'H'K_wK_hC_{in}C_{out}$, where H' and W' is the height and width of the output feature maps. On the other hand, the number of trainable parameters in a standard convolution is characterized as $K_wK_hC_{in}C_{out}$. Note that if we use a width-multiplier [77] to control the width of the CNN, *i.e.*, $C_{in}w$ and $C_{out}w$, the overall FLOPs and parameter counts of the CNN scale about quadratically with the width-multiplier w.

Group convolution A group convolution structurally eliminates some of the connections between input and output channels. Specifically, a group convolution splits input and output channels into G groups of equal size and the convolution is done by executing G standard convolutions in parallel and concatenating their respective results to form the output of the group convolution.

The FLOPs of a group convolution is characterized by $\left(WHK_wK_h\frac{C_{in}}{G}\frac{C_{out}}{G}\right)G$ while the number of trainable parameters is characterized $K_wK_h\frac{C_{in}}{G}\frac{C_{out}}{G}$.

Depth-wise convolution Depth-wise convolution is a special case of group convolution where $C_{in} = C_{out} = G$. As a result, the FLOPs of a depth-wise convolution is given by $WHK_wK_hC_{out}$ while its parameter counts is characterized as K_wK_h.

Point-wise convolution Point-wise convolution is a special case of standard convolution where $K_w = K_h = 1$. As a result, the FLOPs of a depth-wise convolution is given by $WHC_{in}C_{out}$ while its parameter counts is characterized as $C_{in}C_{out}$.

To incorporate the non-linearity into deep neural networks, activation functions (Act) are inserted after convolutional layers. In modern CNNs, batch normalization (BN) layers [84] are often adopted to enable better trainability. As a result, when referring to a single layer of the CNN, we mean $Act(\cdot) \circ BN(\cdot) \circ Conv(\cdot)$.

There are many successful architectures for CNNs, including the VGG network that stacks convolutional layers [157], ResNet that stacks convolutional layers with skip connections [68], DenseNet [79] that has dense connections among convolutional layers, MobileNet that factorizes a standard convolution layer into depth-wise convolution and point-wise convolution [77], and ShuffleNet [201] that replaces the point-wise convolutional layers with a group convolution and a channel shuffling operation for more a efficient computation. Instead of designing CNNs manually, neural architecture search has emerged to be a promising direction for automatically finding a high accuracy CNN configuration [52] .

2.2 Model Compression/Acceleration

Model compression and acceleration is an emerging field of research for deep learning [41] given the need to put deep models onto resource-constrained devices to enhance the applicability of deep neural networks. In this section, we discuss the compression and acceleration techniques that are closely related to this thesis.

2.2.1 Pruning

Pruning removes redundant connections of an existing neural network to compress and/or accelerate the execution of the neural network. Pruning was originally proposed as a means to combat overfitting [94]. With the success of deep learning, pruning has evolved to become a means for removing redundancies to improve efficiency [64]. Typically, pruning can be either *unstructured* or *structured*. In unstructured pruning the pruned unnecessary connections do not obey specific structures that can be exploited by the underlying software and hardware. To accelerate the execution of a sparse CNN derived from unstructured pruning, sparse computation software and hardware support is necessary. On the other hand, in structured pruning the connections are removed in a structural fashion such that the resulting pruned network can be accelerated without specialized software or hardware implementation; in this case, the dominant structure pruning directions address channel or filter pruning [99, 130]. To exploit existing software and hardware implementation, we mainly consider filter pruning in this thesis.

In general, we are interested in finding the weights of a neural network $\boldsymbol{\theta}$ that minimizes the expected loss L over the training data $\boldsymbol{x}$ and label y such that the cost of executing a neural network with such weights $C(\boldsymbol{\theta})$ is below a desired budget δ:

$$\min_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x},y}\ L(\boldsymbol{\theta}, \boldsymbol{x}, y)\ \ \text{s.t.}\ \ C(\boldsymbol{\theta}) \leq \delta. \tag{2.3}$$

In filter pruning, $C(\hat{\boldsymbol{\theta}}) < C(\boldsymbol{\theta})$ only if $\hat{\boldsymbol{\theta}}$ has fewer non-zero filters than $\boldsymbol{\theta}$. The cost function can be different for different application scenarios. Some use the number of floating operations (FLOPs) while others use measured latency. Note that equation 2.3 does not restrict the algorithms to use a notion of a pre-trained model. However, solving it is hard as C is not smooth. In the literature, there are two families of methods to approach equation 2.3. The first family of approaches relaxes the cost function C to be a convex upper-bound and the notable approach is to add group-Lasso regularization on filters [181] or on the weight of the BatchNorm layer [59]. The second family of approaches tries to approximate a pre-trained model with a discrete optimization problem. Specifically, one first trains a neural network without any constraints and obtains a pre-trained model $\boldsymbol{\theta}_{\text{pre}}$. Then, one tries to find a vector of binary masks $\boldsymbol{m}$, which has the dimension of the total number of filters, such that when the mask is applied to the pre-trained model $\boldsymbol{\theta}_{\text{pre}} \odot \boldsymbol{m}$, it results in the least loss degradation $\min_{\boldsymbol{m}} \mathbb{E}_{\boldsymbol{x},y} L(\boldsymbol{\theta}_{\text{pre}} \odot \boldsymbol{m}, \boldsymbol{x}, y) - L(\boldsymbol{\theta}_{\text{pre}}, \boldsymbol{x}, y)$ while satisfying the constraint $C(\boldsymbol{\theta}_{\text{pre}} \odot \boldsymbol{m}) < \delta$. To satisfy the constraint, one can resort to a greedy algorithm that stops once the constraint is satisfied [99, 130, 129, 72, 69], black-box optimization methods [71, 180, 122], sampling methods [105, 95], and differentiable approximation methods [56].

2.2.2 Quantization

Weight quantization is an effective method for reducing the model size of a CNN [211, 98, 208, 76, 43]. The key idea of weight quantization is to constrain the value of network weights to a fixed set of discrete values that admit low-bit representations. To train networks with quantization, one has to overcome the challenge of the quantization function not being differentiable. In quantization-aware training, the straight-through estimator [11] is often adopted to approximate the gradient of the quantization function. Specifically, for bitwidth values no less than 2 bit ($b \geq 2$), the following quantization function is used for weights during the forward pass:

$$Q(\boldsymbol{W}_{i,:}) = \lfloor \frac{clamp(\boldsymbol{W}_{i,:}, -a_i, a_i)}{r_i} \rceil \times r_i, \quad r_i = \frac{a_i}{2^{b-1}-1} \tag{2.4}$$

where

$$clamp(w, min, max) = \begin{cases} w, & \text{if } min \leq w \leq max \\ min, & \text{if } w < min \\ max & \text{if } w > max \end{cases}$$

and $\lfloor \cdot \rceil$ denotes the round-to-nearest-neighbor function, $\boldsymbol{W} \in \mathbb{R}^{C_{out} \times d}$, $d = C_{in} K_w K_h$ denotes the real-value weights for the ith output filter of a convolutional layer that has C_{in} channels and $K_w \times K_h$ kernel size. $\boldsymbol{a} \in \mathbb{R}^{C_{out}}$ denotes the vector of clipping factors which are selected to minimize $\|Q(\boldsymbol{W}_{i,:}) - \boldsymbol{W}_{i,:}\|_2^2$ by assuming $\boldsymbol{W}_{i,:} \sim \mathcal{N}(0, \sigma^2 \boldsymbol{I})$ as suggested in [97].

If we denote

$$|\bar{\boldsymbol{W}}_{i,:}| = \frac{1}{d} \sum_{j=1}^{d} |\boldsymbol{W}_{i,j}|, \tag{2.5}$$

with the normal distribution assumption, prior work [97] suggests that one can determine a_i based on the estimation of $|\bar{\boldsymbol{W}}_{i,:}|$. We run simulations for weights drawn from a zero-mean Gaussian distribution with several variances and identify the best $a_i^* = \arg\min_{a_i} \|Q_{a_i}(\boldsymbol{W}_{i,:}) - \boldsymbol{W}_{i,:}\|_2^2$ empirically. Indeed, we find that one can infer a_i from the sample mean $|\bar{\boldsymbol{W}}_{i,:}|$, which is shown in Figure 2.1. As a result, for the different bitwidth values considered, we find $c = \frac{|\bar{\boldsymbol{W}}_{i,:}|}{a_i^*}$ via simulation and use the obtained c to calculate a_i on-the-fly throughout training.

For special cases such as 1 bit, we follow DoReFaNets [211] and define the quantization function as follows:

$$Q(\boldsymbol{W}_{i,:}) = sign(\boldsymbol{W}_{i,:}) \times \left(|\bar{\boldsymbol{W}}_{i,:}|\right). \tag{2.6}$$

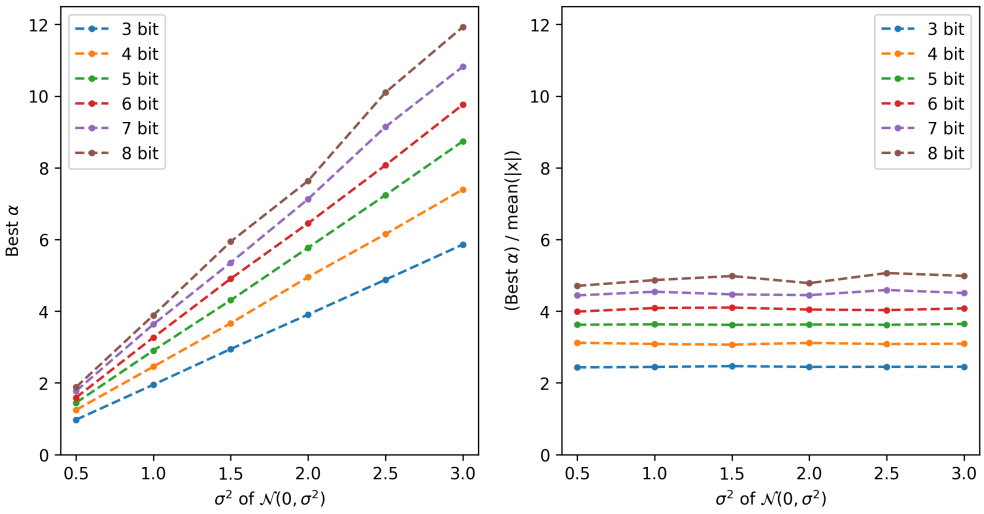


Figure 2.1: Finding best a_i for different precision values empirically through simulation using Gaussian with various σ^2.

For the backward pass for all the bitwidths, we use a straight-through estimator as in prior literature to make the training differentiable. That is,

$$\frac{\partial Q(\boldsymbol{W}_{i,:})}{\partial \boldsymbol{W}_{i,:}} = \boldsymbol{I}. \tag{2.7}$$

With these definitions, prior work has shown great success in training low-precision CNNs [144, 214, 87, 89, 198, 76, 31].

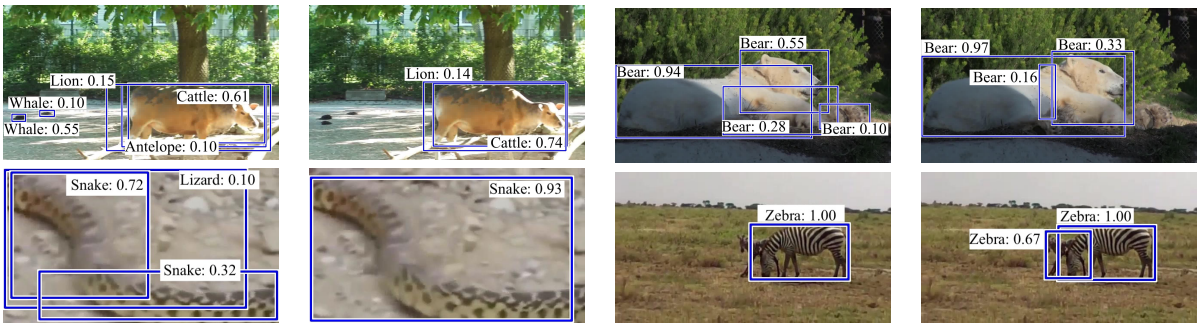
Chapter 3

AdaScale: Scale Inputs Adaptively for Improved Speed and Accuracy

In this chapter, we focus on improving both the speed and accuracy of a video object detection system by introducing a novel algorithm dubbed AdaScale. We first demonstrate the motivation behind scaling the input images for different inputs. Then, we propose AdaScale, which is a methodology to equip existing object detectors with the capability to scale the input resolutions adaptively to improve both speed and accuracy for the video object detection setting. The effectiveness of AdaScale is demonstrated by experiments on ImageNet VID [152] and Youtube-BoundingBox datasets [146].

3.1 Motivation

Video object detection acts as a fundamental building block for visual cognition in future autonomous agents such as autonomous cars, drones, and robots. Therefore, to build systems with reliable performance, it is critical for the detectors to be fast and accurate. Though object detection is well-studied for static images [35, 57, 67, 112, 149], there are unique challenges in the case of *video* object detection, including motion blur caused by the moving objects, failure of camera focus [216], and also real-time speed constraints when it comes to autonomous agents. Besides these *challenges*, however, video object detection also brings new *opportunities* to be exploited. Some of the prior work that focuses on video object detection tries to improve average precision by leveraging a unique characteristic of video [216, 55, 91], which is the temporal consistency (*i.e.*, consecutive frames have similar content). On the other hand, from a speed perspective, prior work [217, 218, 14] counts on the temporal consistency to reduce the computation needed for a standalone object detector. Similarly, we aim to leverage temporal consistency, but to improve both speed and accuracy of the standalone object detectors with a novel technique called adaptive-scale testing,



(a) 600 (b) 240 (c) 600 (d) 480

Figure 3.1: Examples where down-sampled images have better detection results. Blue boxes are the detection results, and the numbers are the confidence. The detector is trained on a single scale (pixels of the shortest side) of 600. Column (a) and (c) are tested at scale 600. Column (b) is tested at scale 240 and column (d) is tested at scale 480.

or *AdaScale*.

The scale of input image affects both the speed and accuracy of modern CNN-based object detectors [80]. Prior work related to image scaling addresses two directions: (i) multi-scale testing for better accuracy, and (ii) down-sampling images for higher speed. Examples from the first category include re-sizing images to various scales (image pyramid) and pushing them through the CNN for feature extraction at various scales [35, 57, 67], as well as fusing feature maps from different layers generated by a single-scale input image [109, 18, 9]. However, these approaches introduce extra computational overhead compared to object detectors with single-scale inputs. Examples from the second category include Pareto optimal search by tuning the input image scale [110, 112, 147, 80] and dynamically re-sizing the image according to the input image [28]. However, results for these approaches demonstrate that higher speed comes at the cost of lower accuracy when it comes to image scaling.

In contrast with prior work, we find that down-sampling images is sometimes beneficial in terms of accuracy. Specifically, there are two sources of improvement brought by image down-sampling: (i) Reducing the number of false positives that may be introduced by focusing on unnecessary details. (ii) Increasing the number of true positives by scaling the objects that are too large to a size at which the object detector is more confident. Fig. 3.1 shows images that are better when down-sampled in our experiments using Region-based Fully Convolutional Network (R-FCN) [35] object detector on ImageNet VID dataset.

Motivated by this, our goal is to re-size the images to their best scale aiming for both higher speed and accuracy. In this chapter, we propose *AdaScale* to boost both the accuracy and the speed of the standalone object detector. Specifically, we use the current frame to predict the optimal scale for the next frame. Our results on ImageNet VID and mini YouTube-BB datasets demonstrate 1.3 points and 2.7 points mAP improvement with 1.6× and 1.8× speedup, respectively. Moreover, by combining with the state-of-the-art

video acceleration work [217], we improve its speed by an an extra 25% with a slight mAP increase on ImageNet VID dataset.

3.2 Adaptive Scaling

To adaptively scale the input resolution, we propose to learn a scale regressor given the current image content and apply its output to resize the next frame based on the temporal consistency assumption in video object detection. Fig. 3.2 provides an overview for AdaScale methodology. It includes fine-tuning the object detector, using the resulting detector to generate the optimal scale labels, training the scale regressor with the generated labels, and the deployment of AdaScale in video object detection. We discuss each component in detail in the following sections.

Figure 3.2: The AdaScale methodology.

3.2.1 Optimal Scale

To define the optimal scale (pixels of the shortest side) of a given image, we need to first define a finite set of scales S (*e.g.*, in our case $S = \{600, 480, 360, 240\}$) and we must have a metric that evaluates the quality of the detection results at these different scales. Naïvely, we can use the commonly used mean average precision (mAP) to compare different scales, and define the scale with the largest mAP as the optimal scale. However, the mAP evaluated for a single image is sparse due to limited number of ground truths per image. Hence, we opt to count on the loss function that is used to train the object detector as the metric to compare results at different scales. In general, the loss function for an object detector used in training often includes the bounding box regression loss and classification loss [57, 149, 35]:

$$L(\mathbf{p}, u, \mathbf{t}, \hat{\mathbf{t}}) = L_{cls}(\mathbf{p}, u) + \lambda[u \geq 1] L_{reg}(\mathbf{t}, \hat{\mathbf{t}}), \tag{3.1}$$

where $\mathbf{p}$ is a vector of predicted probability for each pre-defined class, u is the ground truth class label (0 means background), $\hat{\mathbf{t}}$ is a four-dimension vector that indicates the location information of the bounding box [57], and $\mathbf{t}$ is also a four-dimension vector that represents the ground truth location of the bounding box. Noted that $[u \geq 1]$ indicates that regression loss only applies to the bounding box whose ground truth label is not background. Practically [35, 110], a predicted bounding box is assigned to foreground when there is at least one ground truth bounding box that has over 0.5 Jaccard overlap (intersection over union) [53] with it; otherwise, it is assigned to background. However, since this loss function naturally

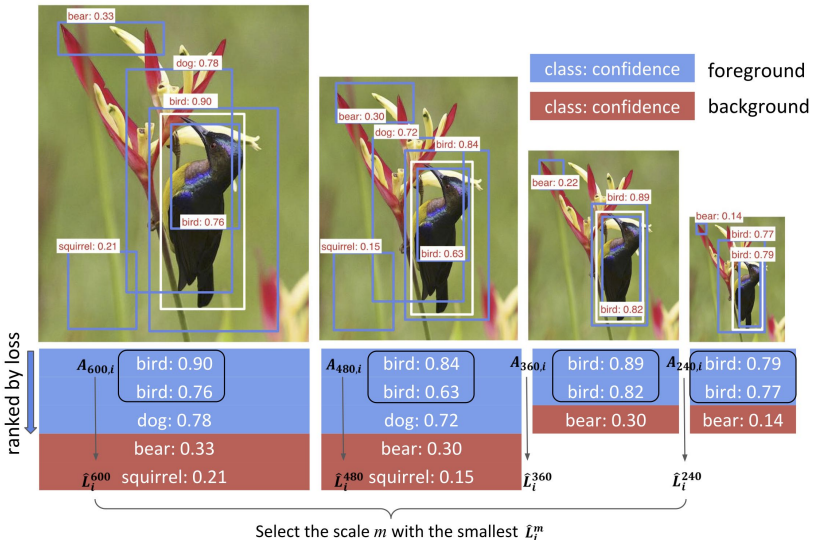


Figure 3.3: Optimal scale determination. First, the same number of predicted foregrounds from four scales are selected as $A_{m,i}$. Then, the scale with the lowest loss L_i^m is selected as the optimal scale.

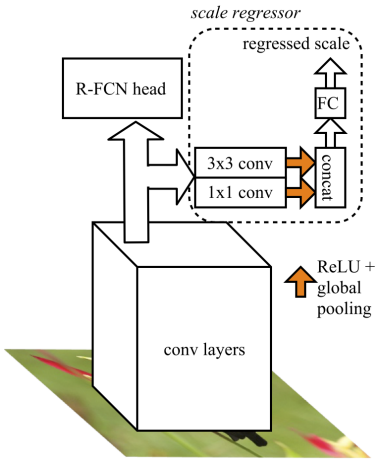


Figure 3.4: The scale regressor module.

assumes that the regression loss for background is 0, directly using it to *assess* different image scales will favor the image scale with fewer foreground bounding boxes.

Hence, to deal with this, we devise a new metric that focuses only on the same number of foreground bounding boxes to compare different image scales. To explain our proposed metric, we denote $L_{i,a}^m, m \in S$ as the loss of predicted bounding box a of image i at scale m using (3.1), and denote $\hat{L}_i^m, m \in S$ for image i at scale m, as our proposed metric. To obtain $\hat{L}_i^m$, we first compute the number of predicted foreground bounding boxes, $n_{m,i}$, for image i at each scale $m \in S$, then let $n_{min,i} = \min_m(n_{m,i})$. Concretely, the proposed metric can be computed as: $\hat{L}_i^m = \sum_{a \in A_{m,i}} L_{i,a}^m$, where $A_{m,i}$ is a set of predicted foreground bounding boxes of image i at scale m and $|A_{m,i}| = n_{min,i}$. To obtain $A_{m,i}$, for each scale, we sort the predicted foreground bounding boxes of image i with respect to $L_{i,a}^m$ in ascending order and pick the first $n_{min,i}$ predictions into the set $A_{m,i}$. The visual illustration of the process is shown in Fig. 3.3. With the proposed metric, we define optimal scale $m_{opt,i}$ for image i as:

$$m_{opt,i} = argmin_m \hat{L}_i^m. \tag{3.2}$$

3.2.2 Scale Regressor

Now that we understand which scale is better for a given image, we may be able to predict the optimal scale for the image. Intuitively, if the object is large or has simple texture, it is likely that we would down-sample the image to let the object detector focus on the salient objects rather than the distracting details. On the other hand, if the object is small or there are many salient objects, the image should remain

in a large scale. Since R-FCN head [35] counts on the deep features (*i.e.*, the last convolutional layer of the backbone feature extractor) to regress bounding box locations, we think that the channels of the deep features already contain size information. As a result, we build a scale regressor using deep features to predict the optimal scale, as shown in Fig. 3.4. Specifically, we use a 1x1 convolutional layer to capture the size information from different feature maps. Additionally, we use a parallel 3x3 convolutional layer to capture the complexity of each 3x3 patch in the feature maps. After the non-linear unit, we use global pooling that acts as a voting process. Lastly, we combine the two streams with a fully connected layer to regress the output scale. To be precise, we define the deep features as $X \in R^{C \times H \times W}$, where C is number of channels, H and W are height and width of the deep feature maps. We define our regressor as $g : R^{C \times H \times W} \rightarrow R$. It is important to note that we do not regress the optimal scale m_{opt} directly since what matters is the content instead of the image size itself. Hence, we regress a relative scale so that the module learns to react (up-sample, down-sample, or stay the same) given the current content of the image. Specifically, the target of the regressed scale for image i is defined as:

$$t(m_i, m_{opt,i}) = 2 \times \frac{m_{opt,i}/m_i - m_{min}/m_{max}}{m_{max}/m_{min} - m_{min}/m_{max}} - 1, \tag{3.3}$$

where m_i is the current scale of the image i, m_{min} is the minimum defined scale, *e.g.*, 128, while m_{max} is the maximum defined scale, *e.g.*, 600. That is, we are regressing to normalized, *i.e.*, [-1, 1], relative scales. To generate labels for the regressor, we calculate (3.2) over the training data to obtain $m_{opt,i} \ \ \forall i \in D_{train}$, where D_{train} is the training data. As commonly used in regression problems, we adopt mean square error (3.4) as the loss function to train the regressor:

$$L_{scalereg} = \frac{1}{|D_{train}|} \sum_{i \in D_{train}} (g(X_i) - t(m_i, m_{opt,i}))^2. \tag{3.4}$$

To incorporate adaptive scaling, or AdaScale, in the video setting, we impose a temporal consistency assumption. More precisely, we assume that the optimal scales for the two consecutive frames are similar; our results empirically justify this assumption. Algorithm 1 shows an example of leveraging AdaScale for video object detection, which is elaborated in section 3.3.2.

3.3 Experiments

3.3.1 Setup

All of our experiments are done using Nvidia GTX 1080 Ti. We base our implementation on the code released by prior work [217], where MXNet [22] is used as the deep learning framework. We conduct our experiments mainly on the ImageNet VID dataset [153], which contains 3862 and 555 training and

```
Input: detector, video, S: pre-defined scale set
image = video.next_frame();
targetScale = 600; // Initialize image scale
while image do
    image = resize(image, targetScale);
    base_size = minimum(image.height, image.width);
    // Regress t of Eq. (3.3)
    bboxes, scores, targetScale = detector.detect(image);
    // Invert Eq. (3.3)
    targetScale = decode(targetScale, base_size, S);
    targetScale.clip_(min(S), max(S)).round_();
    image = video.next_frame();
end
```

Algorithm 1: Pseudo-code for using AdaScale in the testing phase.

validation video snippets, respectively. We use a pre-trained R-FCN model [217], which is trained on both ImageNet DET and ImageNet VID training set. For DET dataset, only the 30 categories that overlap with the VID dataset are selected for training. The evaluation of ImageNet VID is performed on validation set, which follows prior work [217]. In addition to ImageNet VID, we also evaluate our performance on the recently released YouTube-BB dataset [146], which contains 23 categories and around 380,000 video segments. Due to resource and time limitation, we randomly sample 100 segments per category and cut 20 frames per segment to form our mini training set. We also sample 10 segments per category for the validation set to form our mini testing set. To train the model for mini Youtube-BB dataset, we use the model trained on ImageNet VID and DET as a pre-trained model to further fine-tune on mini Youtube-BB.

3.3.2 Training and Testing

Object Detector: First, to avoid the object detector to be biased toward a single scale, we fine-tune the R-FCN model pre-trained at scale 600, for four epochs using multi-scale training [57]. The hyperparameters used follow prior work [217]. Specifically, we use a learning rate of 0.00025 and divide it by 10 after 1.3 and 2.6 epochs, respectively. We use two GPUs with a single image per GPU. Therefore, the training batch size is two. In a addition, we pick the scale (the shortest side of the image) from the set $S_{train} = \{600, 480, 360, 240\}$, and use the maximum bound for the longer side as 2000. Our re-sizing protocol follows Fast R-CNN [57]. In the following sections, we will refer to the shortest side size as the *image scale*. All the detection results in this work use Non-Maximum Suppression (NMS) with threshold 0.3 [35]. For each image, the top-300 confident bounding boxes after NMS are selected as the final output. Note that in terms of data augmentation, we add multi-scale training as an additional data augmentation strategy.

Scale Regressor: With the multi-scale trained object detector, we generate the scale label for each frame

Table 3.1: Evaluation of the proposed method. We denote methods by their approach of training and testing, *e.g.*, MS/SS stands for multi-scale (MS) training and single-scale (SS) testing. Blue text and red text indicate ≥ 1 AP improvement and degradation compared to SS/SS, respectively.

(a) ImageNet VID

Method	airplane	antelope	bear	bike	bird	bus	car	cattle	dog	cat	elephant	fox	g.panda	hamster	horse
SS/SS	88.9	84.5	86.0	65.8	72.2	76.1	58.3	71.0	69.4	76.0	76.4	87.2	81.6	89.8	69.6
MS/SS	88.5	86.2	75.3	62.8	74.2	74.5	55.4	71.2	72.1	75.8	77.1	87.5	82.1	87.5	74.4
MS/AdaScale	88.2	87.0	80.2	67.4	73.7	75.3	57.8	73.4	74.1	81.7	77.7	89.1	81.5	93.5	75.6

lion	lizard	monkey	motorcycle	rabbit	r.panda	sheep	snake	squirrel	tiger	train	turtle	watercraft	whale	zebra	mAP(%)	Runtime(ms)
51.9	79.1	51.2	84.0	63.4	76.8	56.3	75.6	53.9	89.5	82.4	79.0	65.1	74.5	91.3	74.2	75
57.1	78.7	51.2	83.8	61.0	58.7	61.5	68.9	57.3	89.8	81.4	78.2	64.5	74.3	89.2	73.3	75
62.6	78.7	52.2	84.6	63.6	66.4	62.2	73.0	61.0	90.7	82.3	79.7	65.6	75.6	90.4	75.5	**47**

(b) Mini YouTube-BB

Method	person	bird	boat	bike	bus	bear	cow	cat	giraffe	p.plant	horse	motorcycle	knife	airplane	skateboard	train	truck	zebra	toilet	dog	elephant	umbrella	car	mAP(%)	Runtime(ms)
SS/SS	24.9	45.3	39.3	49.1	83.1	67.8	71.8	86.5	83.7	55.0	74.4	51.8	65.1	89.9	54.2	86.7	87.1	88.5	79.7	53.5	82.8	61.1	83.5	68.0	75
MS/SS	22.4	49.4	42.0	61.7	84.2	71.0	71.3	85.1	85.9	49.5	69.3	52.1	62.1	88.8	56.1	88.1	86.8	89.2	83.1	52.5	79.9	61.5	83.4	68.5	75
MS/AdaScale	26.2	53.2	41.9	63.6	83.4	72.6	72.0	87.6	86.8	57.8	75.4	59.0	70.4	89.7	52.5	86.7	87.2	89.0	83.8	53.3	81.4	66.4	85.7	70.7	**41**

in the training data with a set of pre-defined scales using the proposed metric in section 3.2.1. To enable adaptive scaling, the regressor needs to learn to scale up or down according to the current content. To best train the regressor, we should scale the image to every possible scales for the regressor to learn the dynamics. That is, when training the regressor, the input image scale is randomly drawn from a uniform distribution of the pre-defined scale set S_{reg}. In practice, we find $S_{reg} = \{600, 480, 360, 240, 128\}$ is enough to cover the dynamics between 600 and 128. Note that we pick 128 since it is the scale of smallest pre-defined bounding box or anchor used in the Region Proposal Network [149] inside R-FCN and we want to push the image to an as small as possible scale for the largest potential speed improvement. With the generated label, we then train our scale regressor using the training data and freeze the weights of the entire network, except for the scale regressor module. We train the scale regressor for two epochs with an initial learning rate of 10^{-4} and divide by 10 after 1.3 epoch. For the testing phase, as shown in Algorithm 1, we begin every video snippet by re-sizing the first frame to 600. Then, we use the decoded regressed scale for the next frame. As for decoding the regressed scale, we first count on the inverse of (3.3) to obtain a scale in floating point. Then, we round it to an integer, and clip it to the range $[S_{min}, S_{max}]$.

3.3.3 Evaluation

To evaluate the proposed AdaScale, we progressively compare the three methods: (i) SS/SS - a detector trained and tested at 600, which is usually adopted by prior art [149, 35, 217, 216, 55], (ii) MS/SS - a

detector trained at S_{train} and tested at 600, and (iii) MS/AdaScale - a detector trained at S_{train} and tested on an adaptively changing scale between 128 and 600, given the range of S_{reg}. Note that the scale for MS/AdaScale can be any integer value within this range since it is predicted by the scale regressor. The evaluation results are shown in Table 3.1. From this point on, for the sake of simplicity, we base our analysis on ImageNet VID only. The analysis holds for mini YouTube-BB as well.

Accuracy: Compared to the baseline SS/SS, MS/AdaScale increases mAP by 1.3 points. For better visualization, blue numbers in Table 3.1 indicate ≥ 1 AP improvement while red numbers represent ≥ 1 AP degradation. Our approach achieves ≥ 1 AP improvements in half of the categories with only three categories having ≥ 1 AP degradation. In general, multi-scale training can enrich the training data and achieve better generalization of the model. However, this is not always the case. For categories like *red panda* and *bear*, there is a huge AP degradation for all the multi-scale training-based approaches. We find that multi-scale training could potentially lead to some confusion for certain categories. We leave the in-depth study of this phenomena to future work.

We further dive into the precision-recall curve to understand the dynamics of precision and recall for all the methods. Fig. 3.5 shows that precision-recall curves for three most improved categories (a)-(c), one on-par category (d), and two most degraded categories (e)-(f). To give a more comprehensive analysis, we add multi-scale training and multi-scale testing (MS/MS) here for comparison. In addition, we also compare with multi-scale training and random testing scenario, which selects one of the five scales in S_{reg} randomly at test time. Compared to random scaling, MS/AdaScale clearly learns the dynamics of when and how to scale to be able to have consistently higher average precision. Additionally, we can tell from the figure that irrespective of getting better or worse compared to SS/SS, MS/AdaScale follows the curve of MS/MS closely.

Speed: Our scale regressor incurs only 2ms of overhead, which is 3% of the runtime of R-FCN. To see the speed improvement brought by the MS/AdaScale, Fig. 3.10(a) shows the size distribution produced by the scale regressor on ImageNet VID validation dataset and we conduct speed sensitivity analysis on scale set S_{train} in section 3.3.7. We note that, to profile the runtime, we warm up the GPU memory in order to remove the impact of memory allocation overhead of MXNet [22].

3.3.4 Higher Precision with AdaScale

We further dig into what our method actually improves - precision or recall. As mentioned earlier in section 3.2.2, adaptive scaling could possibly increase true positives by scaling the object into a better scale for the detector or reduce false positives by not focusing too much on unnecessary details. To

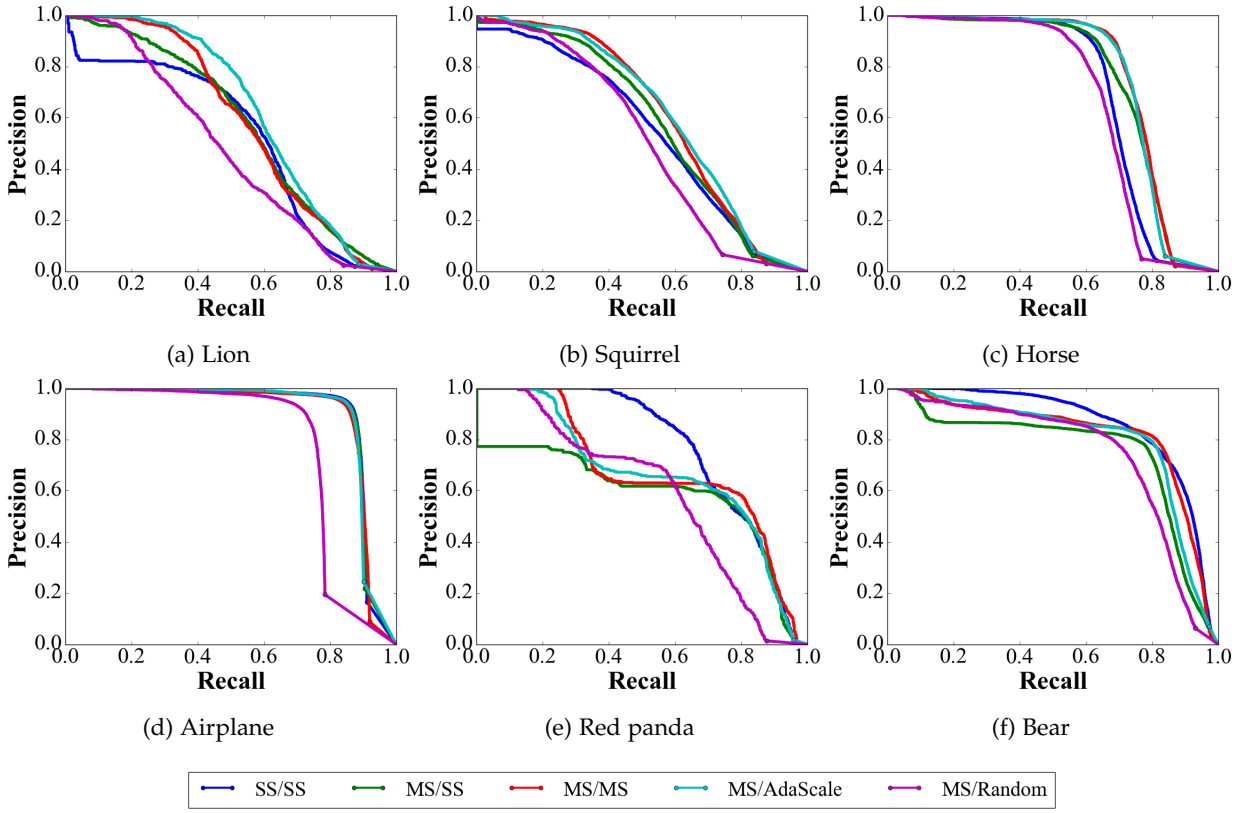


Figure 3.5: Precision-Recall curves for categories that MS/AdaScale has (a)(b)(c) better performance, (d) on-par performance, and (e)(f) worse performance compared to SS/SS.

conduct this analysis, we compute the number of true positives and false positives across all the images in the validation set for method SS/SS, MS/SS, MS/MS, MS/AdaScale, as well as MS/Random. Fig. 3.6 shows the number of true positives and the number of false positives normalized to method SS/SS. First, by comparing SS/SS and MS/SS, we can observe that multi-scale training is able to lower the number of false positives dramatically. This is reasonable since multi-scale training reduces the chance that the classifier counts on scale information as a discriminating feature. The results of MS/SS and MS/Random show that simply down-sampling images can also reduce false positive, but it reduces true positives as well. In addition to the false positive reduction brought by multi-scale training and image down-sampling, MS/AdaScale manages to reduce even more false positives, with true positives comparable to SS/SS. In general, MS/AdaScale is able to increase precision at a slight cost of recall degradation.

3.3.5 Qualitative Results

In Fig. 3.8, we show some example images for the detection results of both the baseline SS/SS and MS/AdaScale. First, we observe that the regressor learns to down-sample the image when there is a

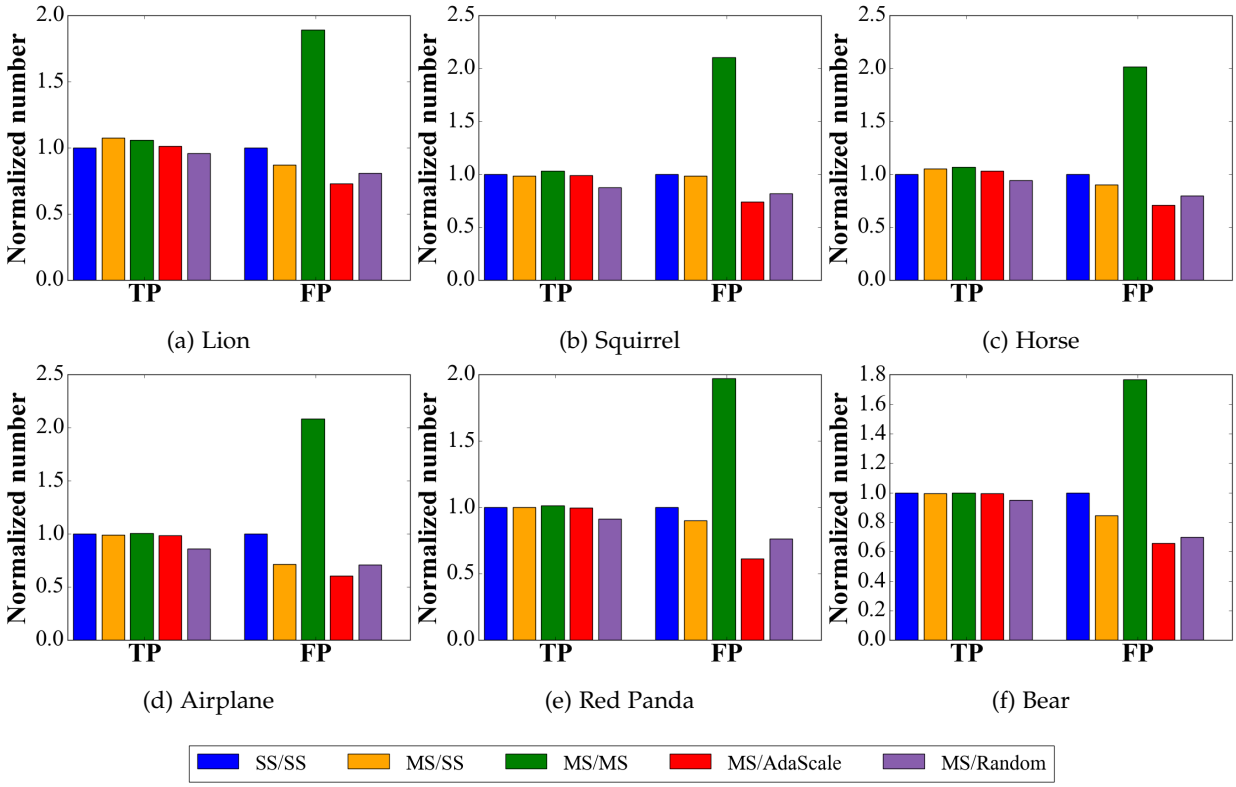


Figure 3.6: Normalized true positives and false positives for different methods across all the images in validation set for three selected categories.

large object in the image. On the other hand, it stays in higher scales if there is a small object in the image. Also, we notice that the regressor learns to scale to the right size to avoid false positives and even correct predictions with false classes.

To understand AdaScale more in terms of the sequential decisions, Fig. 3.9 shows the AdaScale dynamics of three clips. Specifically, it shows that (i) it stably down-samples images with a large object; (ii) it stably scales the images into larger scales when the object is small; and (iii) it jitters when there are multiple objects with varying sizes. The scale jittering in the third clip indicates that if there are size-varying multiple objects in the frame, it is harder to decide what constitutes a better size, which can also be observed in the watercraft of Fig. 3.8. To enhance the current design, it is possible to apply AdaScale recursively on the attention of the given image, to obtain results from multiple regressed scales. We leave improvements of the current design to future work.

3.3.6 Comparison with Prior Work

To our best knowledge, our work is the first to exploit the use of images with smaller scales for improving both speed and accuracy, rather than treating them as a trade-off [80, 35, 147, 28, 110]. For video object

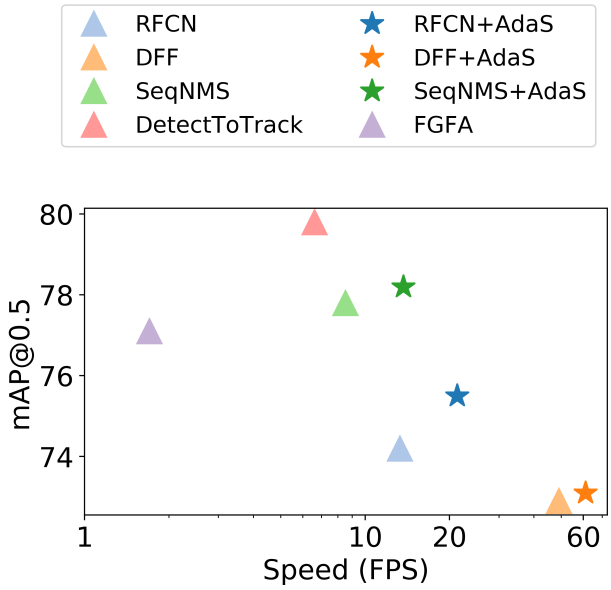


Figure 3.7: mAP and speed comparison with prior art on ImageNet VID dataset. Applying our AdaScale to RFCN [35], DFF [216] and SeqNMS [65] can further improve both speed and accuracy.

detection, our work is complementary to some of the prior work that tries to benefit from the detection results of multiple frames to improve accuracy or speed.

In Fig. 3.7, the baseline object detector is R-FCN [35] with 74.2 mAP and 13.3 frame-per-second (FPS). We run the prior work approaches [216, 217, 65, 55] that provide source code for our experiment setup to profile both speed and mAP. Additionally, we combine our work with SeqNMS [65] and Deep Feature Flow (DFF) [217] to further push the Pareto frontier by maintaining the accuracy while speeding up testing by an additional 61% and 25%, respectively.

3.3.7 Ablation Study

Table 3.2: mAP and runtime for different multi-scale training settings.

S_{train}	{600,480,360,240}		{600,480,360}		{600,360}		{600}	
testing method	SS	Ada.	SS	Ada.	SS	Ada.	SS	Ada.
mAP (%)	73.3	75.5	73.3	74.8	73.4	74.8	74.2	74.2
runtime (ms)	75	47	75	55	75	57	75	68

Training Scales of Object Detector: To understand how multi-scale training affects the performance of AdaScale, we try different sets of training scales S_{train} and the results are shown in Table 3.2 and Fig. 3.10. We find that a larger set of S_{train} improves both the mAP and speed of AdaScale. From Fig. 3.10(a)-(d), we can also observe higher speed with smaller *training* scales. We postulate that it is due to two reasons: (i) Multi-scale trained object detector is able to generate more meaningful labels for the regressor to learn

(a) SS/SS (b) MS/AdaScale (c) SS/SS (d) MS/AdaScale

Figure 3.8: Comparing the results of SS/SS and MS/AdaScale qualitatively. Column (a) and (c) are results produced by SS/SS; column (b) and (d) are results produced by MS/AdaScale. The scales used in MS/AdaScale are labeled in black rectangle with white text.

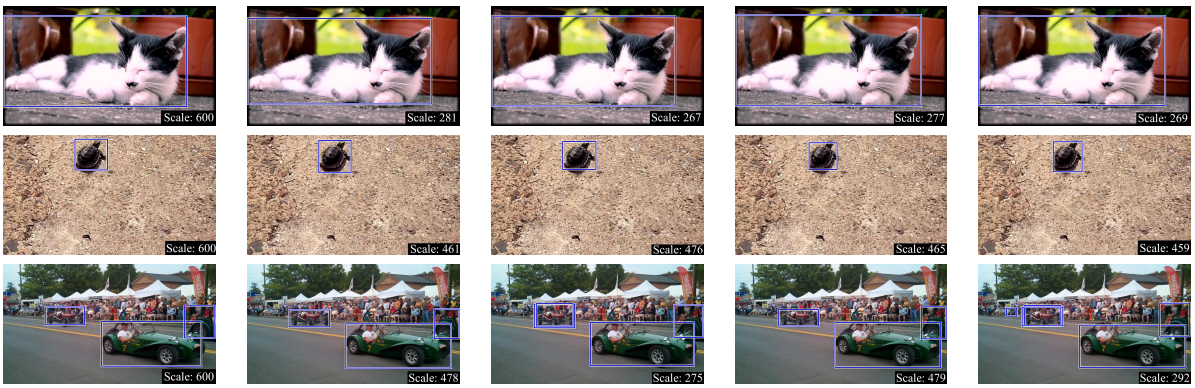
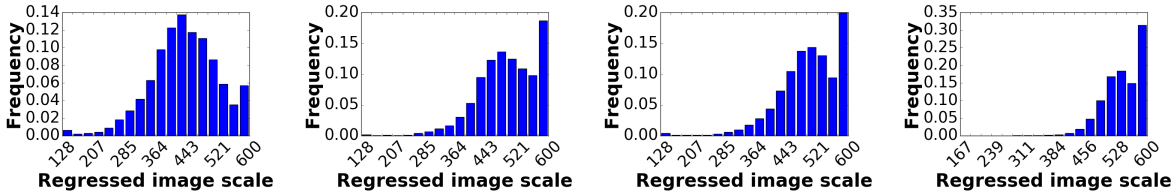


Figure 3.9: The investigation of the dynamics of AdaScale. The scales of the images are labeled in bottom-right.

since it is less biased toward some scales. (ii) The object detector becomes good at multiple scales that could be better exploited by the scale regressor.

Regressor Architectures: We try using different sizes of filter for the regressor module and we show the results in Table 3.3. Interestingly, since the *accuracy* of the regressor directly affects the *speed* of the object detector, both regressor's accuracy and the overhead of the module affect the final overall speed.



(a) {600,480,360,240} (b) {600,480,360} (c) {600,360} (d) {600}

Figure 3.10: The regressed scale distribution of AdaScale tested on ImageNet VID validation set. (a)-(d) use different S_{train}.

Table 3.3: mAP and runtime for different regressor architectures.

kernel size	1	1&3	1&3&5
mAP (%)	75.3	75.5	75.5
runtime (ms)	51	47	50

3.4 Discussion

In this chapter, we present a thorough study of the possibility of improving both speed and accuracy in video object detection with adaptive scaling. Our contributions are three-fold: (i) to the best of our knowledge, our work is the first work to demonstrate the use of down-sampled images for improving both speed and accuracy for video object detection, (ii) we provide comprehensive empirical results that demonstrate improvement in both ImageNet VID as well as mini YouTube-BB datasets, and (iii) we combine our technique with state-of-the-art video object detection acceleration techniques and further improve the speed by an additional 25% with the added benefit of slightly higher accuracy.

3.5 Carbon Footprint Analysis

When compared to standard object detectors, our methodology saves carbon footprint during inference but increases it during training. During training, in addition to obtaining standard object detectors, we conduct multi-scale fine-tuning for object detectors and multi-scale training for the scale regressor. Training the baseline object detector takes 120k iterations [35] and it takes extra 60k iterations for our proposed method. That is, we have increased the training overhead by 1.5×. According to our analysis on ImageNet VID with R-FCN, we have reduced the inference latency by 1.6×. According to a recent study by Patterson *et al.* [139], the ratio of training and inference is roughly nine to one in current cloud providers. With the above calculation, adopting the AdaScale methodology can reduce the total carbon footprint by 29% compared to using a standard object detector.

Chapter 4

Winning-Bitwidth: Beyond Quantization for Fixed CNNs

In this chapter, we describe our finding that model compression via weight quantization benefits some neural architectures more than others. More specifically, if we allow the number of channels to be changed for the network to be quantized, one can achieve better a trade-off between accuracy and compression rate.

4.1 Motivation

Recent success of CNNs in computer vision applications such as image classification and semantic segmentation has fueled many important applications in storage-constrained devices, *e.g.*, virtual reality headsets, drones, and IoT devices. As a result, improving the parameter-efficiency (the top-1 accuracy to the parameter counts ratio) of CNNs while maintaining their attractive features (*e.g.*, accuracy for a task) has gained tremendous research momentum recently.

Among the efforts of improving CNNs' efficiency, weight quantization was shown to be an effective technique [211, 208, 76, 43]. The majority of research efforts in quantization has targeted quantization algorithms for finding the lowest possible weight bitwidth without compromising the figure-of-merit (*i.e.*, accuracy). Mixed-precision quantization methods, which allow different bitwidths to be selected for different layers in the network, have recently been proposed to further compress deep CNNs [176, 185, 50]. Nevertheless, having different bitwidths for different layers greatly increases the neural network implementation complexity from both hardware and software perspectives. For example, hardware and software implementations optimized for executing an 8 bits convolution are sub-optimal for executing a 4 bits convolution, and vice versa.

To minimize the efforts of hardware and software support, it is natural to wonder: "Is some weight

bitwidth better than others?" However, this is an ill-posed problem as one cannot decide optimality between two bitwidths if one has smaller model size while the other has better accuracy. This work takes a first step towards understanding if some bitwidth is better than other bitwidths *under a given model size constraint*. Given the multi-objective nature of the problem, we need to align different bitwidths to the same model size to further decide the optimality for the bitwidth selection. To realize model size alignment for different bitwidths, we relax the commonly adopted notion that the network architecture stays orthogonal to weight quantization. In more details, we propose to change the network architecture so that different bitwidths can have the same model size and we use the width-multiplier[1] [77] as a tool to compare the performance of different weight bitwidths under *the same model size*.

With this setting, we find that there exists some weight bitwidth that consistently outperforms others across different model sizes when all are considered under a given model size constraint. This suggests that one can decide the optimal bitwidth for small model sizes to save computing cost and the result generalizes to large model sizes[2]. Additionally, we show that the optimal bitwidth of a convolutional layer negatively correlates to the convolutional kernel fan-in. As an example, depth-wise convolutional layers turn to have optimal bitwidth values that are higher than that of all-to-all convolutions. We further provide a theoretical reasoning for this phenomenon. These findings suggest that architectures such as VGG and ResNets are more parameter-efficient when they are wide and use binarized weights. On the other hand, networks such as MobileNets [77] might require different weight bitwidths for all-to-all convolutions and depth-wise convolutions. Somewhat surprisingly, we find that on ImageNet, under a given model size constraint, a single bitwidth for both ResNet-50 and MobileNetV2 can outperform mixed-precision quantization using reinforcement learning [176] that targets minimum total bitwidth without accuracy degradation. This suggests that searching for the minimum bitwidth configuration while holding the network architecture to be fixed is a sub-optimal strategy. Our results suggest that when the number of channels becomes one of the hyperparameters under consideration, a single weight bitwidth throughout the network shows great potential for model compression.

In summary, we systematically analyze the model size and accuracy trade-off considering both weight bitwidths and the number of channels for various modern networks architectures (variants of ResNet, VGG, and MobileNet) and datasets (CIFAR and ImageNet) and have the following contributions:

- We empirically show that when allowing the network width to vary, lower weight bitwidths outperform higher ones in a Pareto sense (accuracy *vs.* model size) for networks with standard convolu-

[1] Width-multiplier grows or shrinks the number of channels across the layers with identical proportion for a certain network, *e.g.*, grow the number of channels for all the layers by $2\times$.

[2] Note that we use width-multiplier to scale model across different sizes.

tions. This suggests that for such CNNs, further research on wide binary weight networks is likely to identify better network configurations which will require further hardware/software platform support.

- We empirically show that the optimal bitwidth of a convolutional layer negatively correlates to the convolutional kernel fan-in and provide theoretical reasoning for such a phenomenon. This suggests that one could potential categorize CNNs based on the convolutional kernel fan-in when designing the corresponding bitwidth support from both software and hardware.

- We empirically show that one can achieve a more accurate model (under a given model size) by using a single bitwidth when compared to mixed-precision quantization that uses deep reinforcement learning to search for layer-wise weight precision values. Moreover, the results are validated on a large-scale dataset, *i.e.*, ImageNet.

The remainder of this chapter is organized as follows. Section 4.2 discusses our experiments for all our findings. In particular, Section 4.2.2 shows that some bitwidth can outperform others consistently across model sizes when both are compared under the same model size constraint using width-multipliers. Section 4.2.3 discusses how fan-in channel count per convolutional kernel affects the resilience of quantization for convolution layers, which further affects the optimal bitwidth for a convolution layer. Section 4.2.4 scales up our experiments to ImageNet and demonstrates that a single weight bitwidth manages to outperform mixed-precision quantization given the same model size. Section 4.3 concludes the chapter.

4.2 Experiments

We conduct all our experiments on image classification datasets including CIFAR-100 [93] and ImageNet. All experiments are trained from scratch to ensure different weight bitwidths are trained equally long. While we do not start from a pre-trained model, we note that our baseline fixed-point models (*i.e.*, 4 bits for CIFAR and 8 bits for ImageNet) have accuracy comparable to their floating-point counterparts. For all the experiments on CIFAR, we run the experiments three times and report the mean and standard deviation.

4.2.1 Training hyper-parameters

For CIFAR, we use a learning rate of 0.05, cosine learning rate decay, linear learning rate warmup (from 0 to 0.05) with 5 epochs, batch size of 128, total training epoch of 300, weight decay of $5e^{-4}$, SGD optimizer with Nesterov acceleration and 0.9 momentum.

For ImageNet, we have identical hyper-parameters as CIFAR except for the following hyper-parameters batch size of 256, 120 total epochs for MobileNetV2 and 90 for ResNets, weight decay $4e^{-5}$, and 0.1 label smoothing.

4.2.2 Bitwidth comparisons

In this subsection, we are primarily interested in the following question:

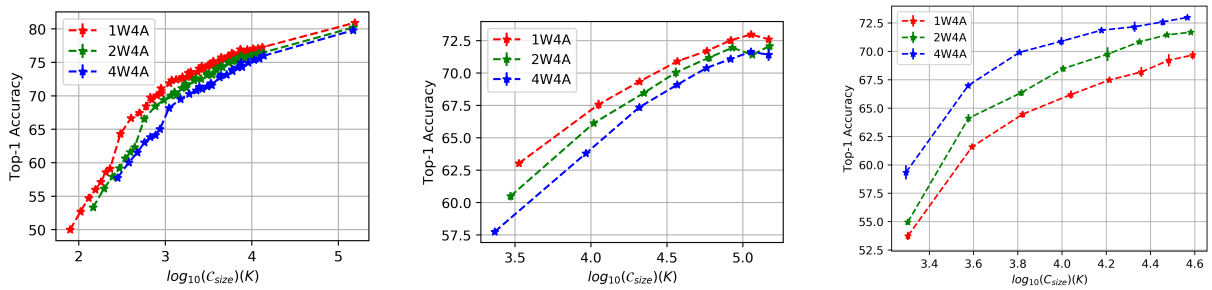
When taking network width into account, does one bitwidth consistently outperform others across model sizes?

To our best knowledge, this is an open question and we take a first step to answer this question empirically. If the answer is affirmative, it may be helpful to focus the software/hardware support on the better bitwidth when it comes to parameter-efficiency. We consider three kinds of commonly adopted CNNs, namely, ResNets with basic block [68], VGG [157], and MobileNetV2 [154]. These networks differ in the convolution operations, connections, and filter counts. For ResNets, we explored networks from 20 to 56 layers in six layer increments. For VGG, we investigate the case of eleven layers. Additionally, we also study MobileNetV2, which is a mobile-friendly network. We note that we modify the stride count in of the original MobileNetV2 to match the number of strides of ResNet for CIFAR. The architectures that we introduce for the controlled experiments are discussed in detail in Appendix A.1.

For CIFAR-100, we only study weight bitwidths below 4 since this configuration achieves performance comparable to its floating-point counterpart. Specifically, we consider 4 bits, 2 bits, and 1 bit weights. To compare different weight bitwidths, we use the width-multiplier to align the model size among them. For example, one can make a 1-bit CNN twice as wide to match the model size of a 4-bit CNN [3]. For each of the networks we study, we sweep the width-multiplier to consider points at multiple model sizes. Specifically, for ResNets, we investigate seven depths, four model sizes for each depth, and three bitwidths, which results in $7 \times 4 \times 3 \times 3$ experiments. For both VGG11 and MobileNetV2, we consider eight model sizes and three bitwidths, which results in $2 \times 8 \times 3 \times 3$ experiments.

As shown in Figure 4.1, across the three types of networks we study, there exists some bitwidth that is better than others. That is, the answer to the question we raised earlier in this subsection is affirmative. For ResNets and VGG, this value is 1 bit. In contrast, for MobileNetV2, it is 4 bits. The results for ResNets and VGG are particularly interesting since lower weight bitwidths are better than higher ones. In other words, binary weights in these cases can achieve the best accuracy and model size trade-off. On the other

[3] Increase the width of a layer increases the number of output filters for that layer as well as the number of channels for the subsequent layer. Thus, number of parameters and number of operations grow approximately quadratically with the width-multiplier.



(a) ResNets (20 to 56 layers in increments of 6)

(b) VGG11

(c) MobileNetV2

Figure 4.1: Some bitwidth is consistently better than other bitwidths across model sizes. $\mathcal{C}_{size}$ denotes model size. xWyA denotes x-bit weight quantization and y-bit activation quantization. The experiments are done on the CIFAR-100 dataset. For each network, we sweep the width-multiplier to cover points at multiple model sizes. For each dot, we plot the mean and standard deviation of three random seeds. The standard deviation might not be visible due to little variances.

hand, MobileNetV2 exhibits a different trend where higher bitwidths are better than lower bitwidths up to 4 bits[4].

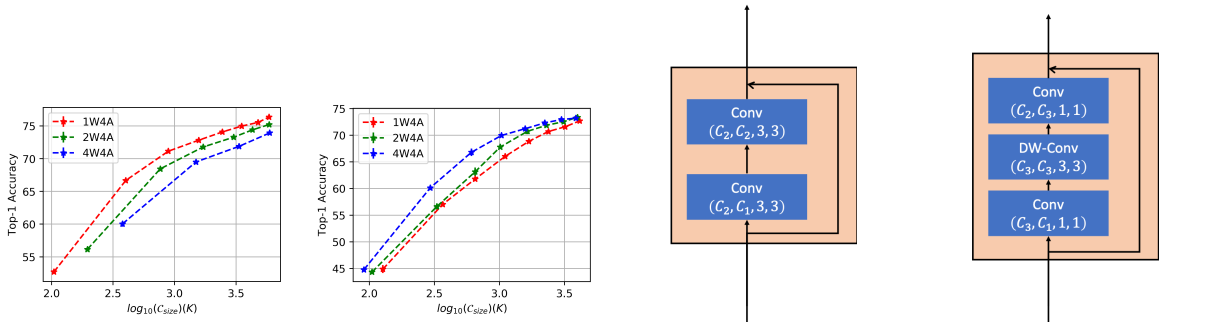
4.2.3 CNN architectures and quantization

While there exists an ordering among different bitwidths as shown in Fig. 4.1, it is not clear what determines the optimal weight bitwidth. To further uncover the relationship between CNN's architectural parameters and its optimal weight bitwidth, we ask the following questions.

What architectural components determine the MobileNetV2 optimal weight bitwidth of 4 bits as opposed to 1 bit?

As it can be observed in Fig. 4.1, MobileNetV2 is a special case where the higher bitwidth is better than lower ones. When comparing MobileNetV2 to the other two networks, there are many differences, including how convolutions are connected, how many convolutional layers are there, how many filters in each of them, and how many channels for each convolution. To narrow down which of these aspects result in the reversed trend compared to the trend exhibits in ResNets and VGG, we first consider the inverted residual blocks, *i.e.*, the basic component in MobileNetV2. To do so, we replace all basic blocks (two consecutive convolutions) of ResNet26 with the inverted residual blocks as shown in Fig. 4.2c and 4.2d. We refer to this new network as Inv-ResNet26. As shown in Fig. 4.2a and 4.2b, the optimal bitwidth shifts from 1 bit to 4 bit once the basic blocks are replaced with inverted residual blocks. Thus, we can infer that the inverted residual block itself or its components are responsible for such a reversed trend.

[4]However, not higher than 4 bits since the 4-bit model has accuracy comparable to the floating-point model.



(a) ResNet26 (b) Inv-ResNet26 (c) Basic block (d) Inverted residual block

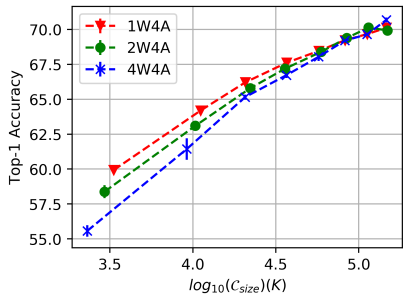
Figure 4.2: The optimal bitwidth for ResNet26 changes from 1 bit (a) to 4 bit (b) when the building blocks change from basic blocks (c) to inverted residual blocks (d). $\mathcal{C}_{size}$ in (a) and (b) denotes model size. (C_{out}, C_{in}, K, K) in (c) and (d) indicate output channel count, input channel count, kernel width, and kernel height of a convolution.

Since an inverted residual block is composed of a point-wise convolution and a depth-wise separable convolution, we further consider the case of depth-wise separable convolution (DWSConv). To identify whether DWSConv can cause the inverted trend, we use VGG11 as a starting point and gradually replace each of the convolutions with DWSConv. We note that doing so results in architectures that gradually resemble MobileNetV1 [77]. Specifically, we introduce three variants of VGG11 that have an increasing number of convolutions replaced by DWSConvs. Starting with the second layer, *variant A* has one layer replaced by DWSConv, *variant B* has four layers replaced by DWSConvs, and *variant C* has all of the layers except for the first layer replaced by DWSConvs (the architectures are detailed in Appendix A.1).

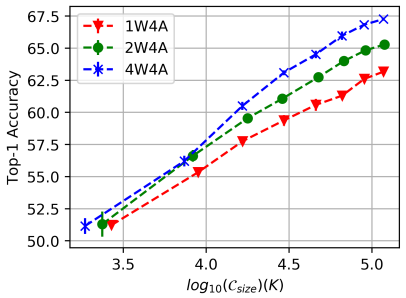
As shown in Fig. 4.3, as the number of DWSConv increases (from variant A to variant C), the optimal bitwidth shifts from 1 bit to 4 bits, which implies that depth-wise separable convolutions or the layers within it are affecting the optimal bitwidth. To identify which of the layers of the DWSConv (*i.e.*, the depth-wise convolution or the point-wise convolution) has more impact on the optimal bitwidth, we keep the bitwidth of depth-wise convolutions fixed at 4 bits and quantize other layers. As shown in Fig. 4.3d, the optimal curve shifts from 4 bits being the best back to 1 bit, with a similarly performing 2 bits. Thus, depth-wise convolutions appear to directly affect the optimal bitwidth trends.

Is depth-wise convolution less resilient to quantization or less sensitive to channel increase?

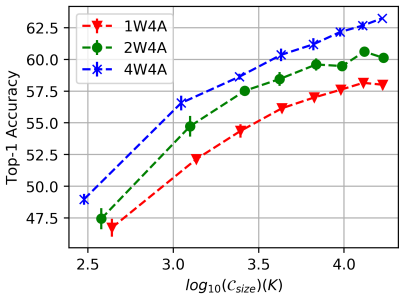
After identifying that depth-wise convolutions have a different characteristic in optimal bitwidth compared to standard all-to-all convolutions, we are interested in understanding the reason behind this. In our setup, the process to obtain a lower bitwidth network that has the same model size as a higher bitwidth network can be broken down into two steps: (1) quantize a network to lower bitwidth and (2) grow the network with width-multiplier to compensate for the reduced model size. As a result, the fact that depth-



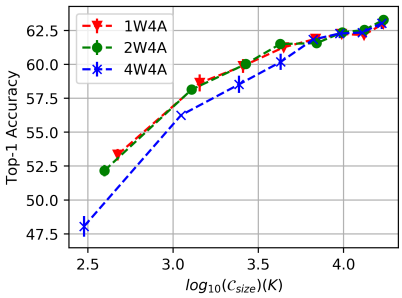
(a) Variant A



(b) Variant B



(c) Variant C



(d) Variant C without quantizing depth-wise convolutions

Figure 4.3: The optimal bitwidth for VGG shifts from 1 bit to 4 bit as more convolutions are replaced with depth-wise separable convolutions (DWSConv), *i.e.*, from (a) to (c). Variant A, B, and C have 30%, 60%, and 90% of the convolution layers replaced with DWSConv, respectively. As shown in (d), the optimal bitwidth changes back to 1 bit if we only quantize point-wise convolution but not depth-wise convolutions.

wise convolution has higher weight bitwidth better than lower weight bitwidth might potentially be due to the large accuracy degradation introduced by quantization or the small accuracy improvements from the use of more channels.

To further diagnose the cause, we decompose the accuracy difference between a lower bitwidth but wider network and a higher bitwidth but narrower network into accuracy differences incurred in the aforementioned two steps as shown in Fig. 4.4. Specifically, let ΔAcc_Q denote the accuracy difference incurred by quantizing a network and let ΔAcc_G denote the accuracy difference incurred by increasing the channel count of the quantized network.

We analyze ΔAcc_G and ΔAcc_Q for networks with and without quantizing depth-wise convolutions, *i.e.*, Fig. 4.3c and Fig. 4.3d. In other words, we would like to understand how depth-wise convolutions affect ΔAcc_G and ΔAcc_Q. On one hand, ΔAcc_Q is evaluated by comparing the accuracy of the 4-bit model and the corresponding 1-bit model. On the other hand, ΔAcc_G is measured by comparing the accuracy of the 1-bit model and its 2× grown counterpart. As shown in Table 4.1, when quantizing depth-wise convolutions, ΔAcc_Q becomes more negative such that $\Delta Acc_Q + \Delta Acc_G < 0$. This implies that the main reason for the

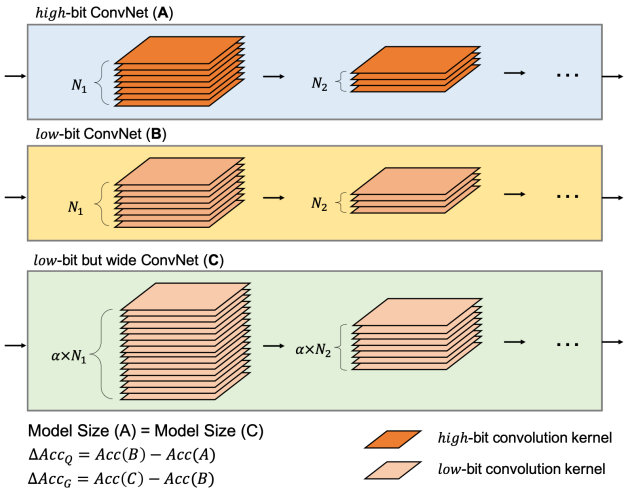


Figure 4.4: Visualization of our accuracy decomposition, which is used for analyzing depth-wise convolutions.

Table 4.1: Quantizing depth-wise convolution introduces large accuracy degradation across model sizes. $\Delta Acc_Q = Acc_{1bit} - Acc_{4bit}$ denotes the accuracy introduced by quantization and $\Delta Acc_G = Acc_{1bit,2\times} - Acc_{1bit}$ denotes the accuracy improvement by increasing channel counts. The CNN is VGG variant C with and without quantizing the depth-wise convolutions from 4 bits to 1 bit.

Width-multiplier	1.00×		1.25×		1.50×		1.75×		2.00×		Average	
Variant C	ΔAcc_Q	ΔAcc_G	ΔAcc_Q	ΔAcc_G	ΔAcc_Q	ΔAcc_G	ΔAcc_Q	ΔAcc_G	ΔAcc_Q	ΔAcc_G	ΔAcc_Q	ΔAcc_G
w/o Quantizing DWConv	-1.54	+2.61	-2.76	+2.80	-1.77	+1.74	-1.82	+1.64	-1.58	+1.55	-1.89	+2.07
Quantizing DWConv	-8.60	+4.39	-7.60	+3.41	-7.74	+3.19	-8.61	+4.09	-7.49	+2.25	-8.01	+3.47

optimal bitwidth change is that quantizing depth-wise convolutions introduce more accuracy degradation than it can be recovered by increasing the channel count when going below 4 bits compared to all-to-all convolutions. We note that it is expected that quantizing the depth-wise convolutions would incur smaller ΔAcc_Q compared to their no-quantization baseline because we essentially quantized more layers. However, depth-wise convolutions only account for 2% of the model size but incur on average near 4× more accuracy degradation when quantized.

We would like to point out that Sheng *et al.* [156] also find that quantizing depth-wise separable convolutions incurs large accuracy degradation. However, their results are based on post-training layer-wise quantization. As mentioned in their work [156], the quantization challenges in their setting could be resolved by quantization-aware training, which is the scheme considered in this chapter. Hence, our observation is novel and interesting.

Why is depth-wise convolution less resilient to quantization?

Having uncovered that depth-wise convolutions introduce large accuracy degradation when weights are quantized below 4 bits, in this section, we investigate depth-wise convolutions from a quantization

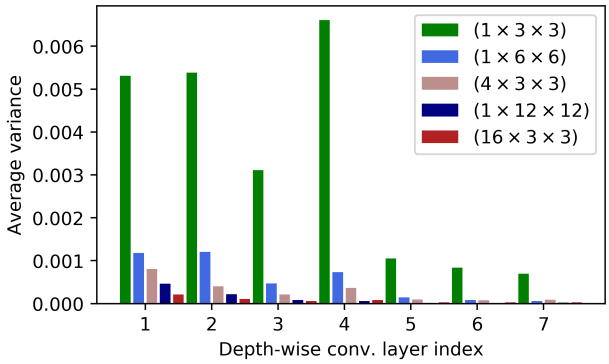


Figure 4.5: The average estimate $\text{Var}(|\bar{\boldsymbol{w}}|)$ for each depth-wise convolution under different $d = (C_{in} \times K_w \times K_h)$ values.

perspective. When comparing depth-wise convolutions and all-to-all convolutions in the context of quantization, they differ in the number of elements to be quantized, *i.e.*, $C_{in} = 1$ for depth-wise convolutions and $C_{in} >> 1$ for all-to-all convolutions.

Why does the number of elements matter? In quantization-aware training, one needs to estimate some statistics of the vector to be quantized (*i.e.*, $\boldsymbol{a}$ in Equation 2.4 and $|\bar{\boldsymbol{w}}|$ in Equations 2.6) based on the elements in the vector. The number of elements affect the robustness of the estimate that further decides the quantized weights. More formally, we provide the following proposition.

Proposition 4.2.1 *Let $\boldsymbol{w} \in \mathbb{R}^d$ be the weight vector to be quantized where $\boldsymbol{w}_i$ is characterized by normal distribution $\mathcal{N}(0, \sigma^2)$ $\forall$ i without assuming samples are drawn independently and $d = C_{in}K_wK_h$. If the average correlation of the weights is denoted by ρ, the variance of $|\bar{\boldsymbol{w}}|$ can be written as follows:*

$$\text{Var}(|\bar{\boldsymbol{w}}|) = \frac{\sigma^2}{d} + \frac{(d-1)\rho\sigma^2}{d} - \frac{2\sigma^2}{\pi}. \tag{4.1}$$

The proof is in Appendix A.2. This proposition states that, as the number of elements (d) increases, the variance of the estimate can be reduced (due to the first term in equation (4.1)). The second term depends on the correlation between weights. Since the weights might not be independent during training, the variance is also affected by their correlations.

We empirically validate Proposition 4.2.1 by looking into the sample variance of $|\bar{\boldsymbol{w}}|$ across the course of training[5] for different d values by increasing (K_w, K_h) or C_{in}. Specifically, we consider the 0.5× VGG variant C and change the number of elements of the depth-wise convolutions. Let $d = (C_{in} \times K_w \times K_h)$ for a convolutional layer, we consider the original depth-wise convolution, *i.e.*, $d = 1 \times 3 \times 3$ and increased channels with $d = 4 \times 3 \times 3$ and $d = 16 \times 3 \times 3$, and increased kernel size with $d = 1 \times 6 \times 6$ and $d = 1 \times 12 \times 12$. The numbers are selected such that increasing the channel count results in the same d

[5]We treat the calculated $|\bar{\boldsymbol{w}}|$ at each training step as a sample and calculate the sample variance across training steps.

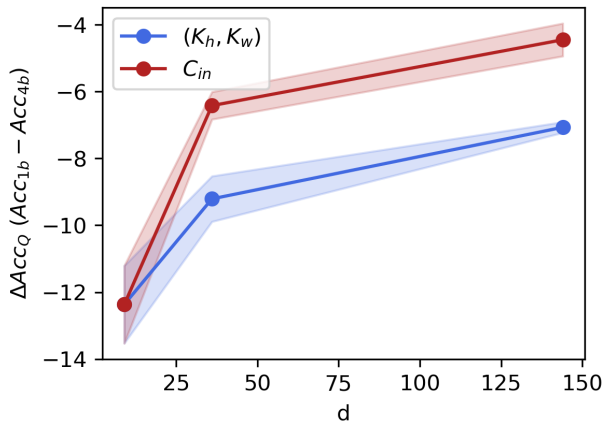


Figure 4.6: d negatively correlates with the variance and positively correlates with the accuracy difference induced by quantization $\Delta Acc_Q = Acc_{1bit} - Acc_{4bit}$.

compared to increasing the kernel sizes. We note that when the channel count (C_{in}) is increased, it is no longer a depth-wise convolution, but rather a group convolution.

In Fig. 4.5, we analyze layer-level sample variance by averaging the kernel-level sample variance in the same layer. First, we observe that results align with Proposition 4.2.1. That is, one can reduce the variance of the estimate by increasing the number of elements along both the channel (C_{in}) and kernel size dimensions (K_w, K_h). Second, we find that increasing the number of channels (C_{in}) is more effective in reducing the variance than increasing kernel size (K_w, K_h), which could be due to the weight correlation, *i.e.*, intra-channel weights have larger correlation than inter-channel weights.

Nonetheless, while lower variance suggests a more stable value during training, it might not necessarily imply lower quantization error for the quantized models. Thus, we conduct an accuracy sensitivity analysis with respect to quantization for different d values. More specifically, we want to understand how d affects the accuracy difference between lower bitwidth (1 bit) and higher bitwidth (4 bits) models (ΔAcc_Q). As shown in Fig. 4.6, we empirically find that d positively correlates with ΔAcc_Q, *i.e.*, the larger the d, the smaller the accuracy degradation is. On the other hand, when comparing channel counts and kernel sizes, we observe that increasing the number of channels is more effective than increasing the kernel size in reducing accuracy degradation caused by quantization. This analysis sheds light on the two different trends observed in Fig. 4.1.

4.2.4 Remarks and scaling up to ImageNet

We have two intriguing findings so far. First, there exists some bitwidth that is better than others across model sizes when compared under a given model size. Second, the optimal bitwidth is architecture-

Table 4.2: bitwidth ordering for MobileNetV2 and ResNet50 with *the model size aligned to the* 0.25× *8 bits models* on ImageNet. Each cell reports the top-1 accuracy of the corresponding model. The trend for the optimal bitwidth is similar to that of CIFAR-100 (4 bit for MobileNetV2 and 1 bit for ResNet).

Weight bitwidth for Convs \ DWConvs	MobileNetV2				ResNet50
	8 bits	4 bits	2 bits	1 bit	None
8 bits	52.17	53.89	50.51	48.78	71.11
4 bits	56.84	**59.51**	57.37	55.91	74.65
2 bits	53.89	57.10	55.26	54.04	75.12
1 bit	54.82	58.16	56.90	55.82	**75.44**

dependent. More specifically, the optimal weight bitwidth negatively correlates with the fan-in channel counts per convolutional kernel. These findings show promising results for the hardware and software researchers to support only a certain set of bitwidths when it comes to parameter-efficiency. For example, use binary weights for networks with all-to-all convolutions.

Next, we scale up our analysis to the ImageNet dataset. Specifically, we study ResNet50 and MobileNetV2 on the ImageNet dataset. Since we keep the bitwidth of the first and last layer quantized at 8 bits, scaling them in terms of width will grow the number of parameters much more quickly than other layers. As a result, we keep the number of channels for the first and last channel fixed for the ImageNet experiments. As demonstrated in Section 4.2.2, the bit ordering is consistent across model sizes, we conduct our analysis for ResNet50 and MobileNetV2 by scaling them down with a width-multiplier of 0.25× for computational considerations. The choices of bitwidths are limited to $\{1, 2, 4, 8\}$.

As shown in Table 4.2, we can observe a trend similar to the CIFAR-100 experiments, *i.e.*, for networks without depth-wise convolutions, the lower weight bitwidths the better, and for networks with depth-wise convolutions, there are sweet spots for depth-wise and other convolutions. Specifically, the final weight bitwidth selected for MobileNetV2 is 4 bits for both depth-wise and standard convolutions. On the other hand, the selected weight bitwidth for ResNet50 is 1 bit. If bit ordering is indeed consistent across model sizes, these results suggest that the optimal bitwidth for MobileNetV2 is 4 bit and it is 1 bit for ResNet50. However, throughout our analysis, we have not considered mixed-precision, which makes it unclear if the so-called optimal bitwidth (4 bit for MobileNetV2 and 1 bit for ResNet-50) is still optimal when compared to mixed-precision quantization.

As a result, we further compare with mixed-precision quantization that uses reinforcement learning to find the layer-wise bitwidth [176]. Specifically, we follow [176] and use a reinforcement learning approach to search for the lowest bitwidths without accuracy degradation (compared to the 8 bits fixed point models). To compare the searched model with other alternatives, we use width-multipliers on top of the searched network match the model size of the 8 bit quantized model. We consider networks of three sizes,

Table 4.3: The optimal bitwidth selected in Table 4.2 is indeed better than 8 bit when scaled to larger model sizes and more surprisingly, it is better than mixed-precision quantization. All the activations are quantized to 8 bits.

Width-multiplier for 8-bit model		1×		0.5×		0.25×	
Networks	Methods	Top-1 (%)	$\mathcal{C}_{size}$ (10^6)	Top-1 (%)	$\mathcal{C}_{size}$ (10^6)	Top-1 (%)	$\mathcal{C}_{size}$ (10^6)
ResNet50	Floating-point	76.71	816.72	74.71	411.48	71.27	255.4
	8 bits	76.70	204.18	74.86	102.87	71.11	63.85
	Flexible [176]	77.23	204.18	76.04	102.90	74.30	63.60
	Optimal (1 bit)	**77.58**	204.08	**76.70**	102.83	**75.44**	63.13
MobileNetV2	Floating-point	71.78	110.00	63.96	61.76	52.79	47.96
	8 bits	71.73	27.50	64.39	15.44	52.17	11.99
	Flexible [176]	72.13	27.71	65.00	15.54	55.20	12.10
	Optimal (4 bit)	**73.91**	27.56	**68.01**	15.53	**59.51**	12.15

i.e., the size of $1\times, 0.5\times$ and $0.25\times$ 8-bit fixed point models. As shown in Table 4.3, we find that a single bitwidth (selected via Table 4.2) outperforms both 8 bit quantization and mixed-precision quantization by a significant margin for both networks considered. This results suggest that searching for the bitwidth without accuracy degradation is indeed a sub-optimal strategy and can be improved by incorporating channel counts into the search space and reformulate the optimization problem as maximizing accuracy under storage constraints. Moreover, our results also imply that when the number of channels are allowed to be altered, a single weight bitwidth throughout the network shows great potential for model compression, which has the potential of greatly reducing the software and hardware optimization costs for quantized CNNs.

4.3 Discussion

In this chapter, we provide the first attempt to understand the ordering between different weight bitwidths by allowing the channel counts of the considered networks to vary using the width-multiplier. If there exists such an ordering, it may be helpful to focus on software/hardware support for higher-ranked bitwidth when it comes to parameter-efficiency, which in turn reduces software/hardware optimization costs. To this end, we have three findings: (1) there exists a weight bitwidth that is better than others across model sizes under a given model size constraint, (2) the optimal weight bitwidth of a convolutional layer negatively correlates to the fan-in channel counts per convolutional kernel, and (3) with a single weight bitwidth for the whole network, one can find configurations that outperform model-oblivious layer-wise mixed-precision quantization using reinforcement learning when compared under a given same model size constraint. Our results suggest that when the CNNs to be quantized are allowed to be altered architecturally, a single weight bitwidth throughout the network shows great potential for model compression.

Additionally, our results show that it may be promising to conduct neural architecture search (NAS) jointly with network quantization. While Wang *et al.* [178] have conducted a similar study, their architecture search space does not include the number of groups for convolutional kernel, which is the crucial aspect for quantization as hinted by the analysis in this chapter. Hence, it would be desirable for future work to explore joint NAS and quantization while specifically taking the convolutional groups into consideration.

4.4 Carbon Footprint Analysis

Without assuming specialized hardware that can benefit from quantized representation, our approach incurs extra carbon footprint. More specifically, we grow the network wider and use lower weight bitwidth during training and inference. Compared to an 8-bit model, our approach makes the model 1-bit and 2.83× wider for ResNets and 4-bit and 2× wider for MobileNets. This effectively introduces additional 8× and 4× carbon footprint for ResNets and MobileNets. However, if we assume the underlying hardware can benefit from lower precision during inference and assume that the training to inference ratio is one to nine [139], then our method improves the top-1 accuracy by incurring 1.7× and 1.3× carbon footprint for ResNets and MobileNets, respectively.

Chapter 5

LeGR: Towards Efficient Filter Pruning

Starting with this chapter, we switch gears from proposing novel ways for model compression to proposing novel ways to scale model compression across many target compression ratios. In this chapter, we propose to make filter pruning more efficient across multiple target compression ratios by introducing a new way to formulate the problem. Specifically, we propose to learn a global ranking among filters in a pre-trained network such that a greedy pruning procedure, which is efficient for many target compression ratios, can be effective.

5.1 Motivation

Building on top of the success of visual perception [150, 66, 68], natural language processing [38, 42], and speech recognition [30, 138] with deep learning, researchers have started to explore the possibility of embodied AI applications. In embodied AI, the goal is to enable agents to take actions based on perceptions in some environments [155]. We envision that next generation embodied AI systems will run on mobile devices such as autonomous robots and drones, where compute resources are limited and thus, will require model compression techniques for bringing such intelligent agents into our lives.

In particular, pruning the convolutional filters in CNNs, also known as filter pruning, was shown to be an effective technique [190, 114, 181, 99] for trading accuracy for inference speed improvements. The core idea of filter pruning is to find the least important filters to prune by minimizing the accuracy degradation and maximizing the speed improvement. State-of-the-art filter pruning methods [59, 71, 114, 213, 140, 34] require a target model complexity of the whole CNN (*e.g.*, total filter count, FLOP count[1], model size, inference latency, *etc.*) to obtain a pruned network. However, deciding a target model complexity for optimizing embodied AI applications can be hard. For example, considering delivery with autonomous

[1]The number of floating-point operations to be computed for a CNN to carry out an inference.

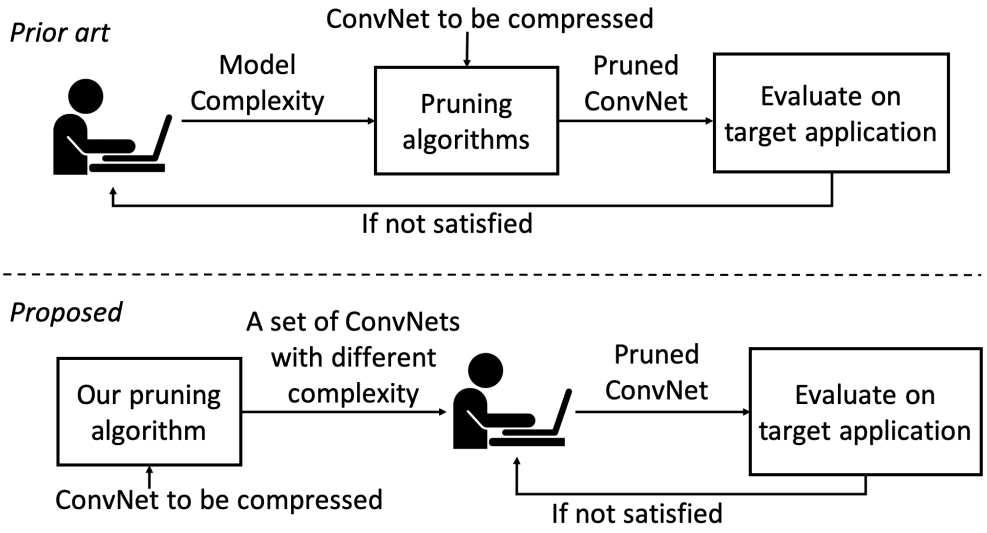


Figure 5.1: Using filter pruning to optimize CNNs for embodied AI applications. Instead of producing one CNN for each pruning procedure as in prior art, our proposed method produces a set of CNNs for practitioners to efficiently explore the trade-offs.

drones, both inference speed and precision of object detectors can affect the drone velocity [13], which in turn affects the inference speed and precision[2]. For an user-facing autonomous robot that has to perform complicated tasks such as MovieQA [166], VQA [3], and room-to-room navigation [2], both speed and accuracy of the visual perception module can affect the user experience. These aforementioned applications require many iterations of trial-and-error to find the optimal trade-off point between speed and accuracy of the CNNs.

More concretely, in these scenarios, practitioners would have to determine the sweet-spot for model complexity and accuracy in a trial-and-error fashion. Using an existing filter pruning algorithm many times to explore the impact of the different accuracy-*vs.*-speed trade-offs can be time-consuming. Figure 5.1 demonstrates the usage of filter pruning for optimizing CNNs in aforementioned scenarios. With prior approaches, one has to go through the process of finding constraint-satisfying pruned-CNNs via a pruning algorithm for every model complexity considered until practitioners are satisfied with the accuracy-*vs.*-speedup trade-off. Our work takes a first step toward alleviating the inefficiency in the aforementioned paradigm. We propose to alter the objective of pruning from outputting a single CNN with pre-defined model complexity to producing a set of CNNs that have different accuracy/speed trade-offs, while achieving comparable accuracy with state-of-the-art methods (as shown in Figure 5.4). In this fashion, the model compression overhead can be greatly reduced, which results in a more practical usage of

[2]Higher velocity requires faster computation and might cause accuracy degradation due to the blurring effect of the input video stream.

filter pruning.

To this end, we propose *learned global ranking (or LeGR)*, an algorithm that learns to rank convolutional filters across layers such that the CNN architectures of different speed/accuracy trade-offs can be obtained easily by dropping the bottom-ranked filters. The obtained architectures are then fine-tuned to generate the final models. In such a formulation, one can obtain a set of architectures by learning the ranking *once*. We demonstrate the effectiveness of the proposed method with extensive empirical analyses using ResNet and MobileNetV2 on CIFAR-10/100, Bird-200, and ImageNet datasets. The main contributions of this chapter are as follows:

- We propose learned global ranking (LeGR), which produces a set of pruned CNNs with different accuracy/speed trade-offs. LeGR is shown to be faster than prior art in CNN pruning, while achieving comparable accuracy with state-of-the-art methods on three datasets and two types of CNNs.
- Our formulation towards pruning is the first work that considers learning to rank filters across different layers globally, which addresses the limitation of prior art in magnitude-based filter pruning.

5.2 Learned Global Ranking

The core idea of the proposed method is to learn a ranking for filters across different layers such that a CNN of a given complexity can be obtained easily by pruning out the bottom rank filters. In this section, we discuss our assumptions and formulation toward achieving this goal.

As mentioned earlier in Section 5.1, often both accuracy and latency of a CNN affect the performance of the overall application. The goal for model compression in these settings is to explore the accuracy-*vs.*-speed trade-off for finding a sweet-spot for a particular application using model compression. Thus, in this chapter, we use FLOP count for the model complexity to sample CNNs. As we will show in Section 5.4.3, we find FLOP count to be predictive for latency.

5.2.1 Global Ranking

To obtain pruned-CNNs with different FLOP counts, we propose to learn the filter ranking globally across layers. In such a formulation, the global ranking for a given CNN just needs to be learned once and can be used to obtain CNNs with different FLOP counts. However, there are two challenges for such a formulation. First, the global ranking formulation enforces an assumption that the top-performing smaller CNNs are a proper subset of the top-performing larger CNNs. As there are many ways to set the filter counts across different layers to achieve a given FLOP count, it implies that there are opportunities where

the top-performing smaller network can have more filter counts in some layers but fewer filter counts in others compared to a top-performing larger CNN. Nonetheless, this assumption enables the idea of global filter ranking, which can generate pruned CNNs with different FLOP counts efficiently. In addition, the experiment results in Section 5.4.1 show that the pruned CNNs under this assumption are competitive in terms of performance with the pruned CNNs obtained without this assumption. We state the subset assumption more formally below.

Assumption 5.2.1 (Subset Assumption) *We assume there are many local optima for a pruned CNN with FLOP count f. Let $\mathcal{F}(f)$ be a set pruned architectures with FLOP count f that achieve similar local optima. Let $A \in \mathcal{F}(f)$, the subset assumption states that one can find an architecture $B \in \mathcal{F}(f')$ where $f' \leq f$ by only reducing the channel counts of A.*

Another challenge for learning a global ranking is the hardness of the problem. Obtaining an optimal global ranking can be expensive, *i.e.*, it requires $O(K \times K!)$ rounds of network fine-tuning, where K is the number of filters. Thus, to make it tractable, we assume the filter norm is able to rank filters locally (within layer) but not globally (across layers).

Assumption 5.2.2 (Norm Assumption) *ℓ_2 norm can be used to compare the importance of a filter within each layer, but not across layers.*

We note that the *norm assumption* is adopted and empirically verified by prior art [99, 189, 71]. For filter norms to be compared across layers, we propose to learn layer-wise affine transformations over filter norms. Specifically, the importance of filter i is defined as follows:

$$I_i = \alpha_{l(i)} \left\| \mathbf{\Theta_i} \right\|_2^2 + \kappa_{l(i)}, \tag{5.1}$$

where $l(i)$ is the layer index for the ith filter, $\|\cdot\|_2$ denotes ℓ_2 norms, $\mathbf{\Theta_i}$ denotes the weights for the ith filter, and $\boldsymbol{\alpha} \in \mathbb{R}^L$, $\boldsymbol{\kappa} \in \mathbb{R}^L$ are learnable parameters that represent layer-wise scale and shift values, and L denotes the number of layers. We will detail in Section 5.2.2 how $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pairs are learned so as to maximize overall accuracy.

Based on these learned affine transformations from Eq. (5.1) (*i.e.*, the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair), the LeGR-based pruning proceeds by ranking filters globally using $\boldsymbol{I}$ and prunes away bottom-ranked filters, *i.e.*, smaller in $\boldsymbol{I}$, such that the FLOP count of interest is met, as shown in Figure 5.2. This process can be done efficiently without the need of training data (since the knowledge of pruning is encoded in the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair).

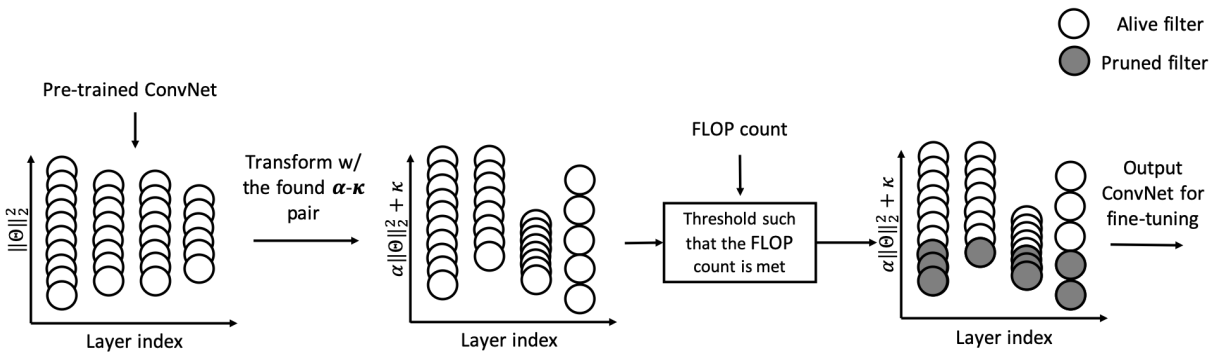


Figure 5.2: The flow of LeGR-Pruning. $\|\Theta\|_2^2$ represents the filter norm. Given the learned layer-wise affine transformations, *i.e.*, the α-κ pair, LeGR-Pruning returns filter masks that determine which filters are pruned. After LeGR-Pruning, the pruned network will be fine-tuned to obtain the final network.

5.2.2 Learning Global Ranking

To learn α and κ, one can consider constructing a ranking with α and κ and then uniformly sampling CNNs across different FLOP counts to evaluate the ranking. However, CNNs obtained with different FLOP counts have drastically different validation accuracy, and one has to know the Pareto curve[3] of pruning to normalize the validation accuracy across CNNs obtained with different FLOP counts. To address this difficulty, we propose to evaluate the validation accuracy of the CNN obtained from the lowest considered FLOP count as the objective for the ranking induced by the α-κ pair. Concretely, to learn α and κ, we treat LeGR as an optimization problem:

$$\arg\max_{\alpha,\kappa} \; Acc_{val}(\hat{\Theta}_{\hat{l}}) \tag{5.2}$$

where

$$\hat{\Theta}_{\hat{l}} = \text{LeGR-Pruning}(\alpha, \kappa, \hat{\zeta}_l). \tag{5.3}$$

LeGR-Pruning prunes away the bottom-ranked filters until the desired FLOP count is met as shown in Figure 5.2. $\hat{\zeta}_l$ denotes the lowest FLOP count considered. As we will discuss later in Section 5.4.1, we have also studied how $\hat{\zeta}$ affects the performance of the learned ranking, *i.e.*, how the learned ranking affects the accuracy of the pruned networks.

Specifically, to learn the α-κ pair, we rely on approaches from hyper-parameter optimization literature. While there are several options for the optimization algorithm, we adopt the regularized evolutionary algorithm (EA) proposed in [145] for its effectiveness in the neural architecture search space. The pseudo-code for our EA is outlined in Algorithm 2. We have also investigated policy gradients for solving for the

[3] A Pareto curve describes the optimal trade-off curve between two metrics of interest. Specifically, one cannot obtain improvement in one metric without degrading the other metric. The two metrics we considered in this chapter are accuracy and FLOP count.

Input: model $\mathbf{\Theta}$, lowest constraint $\hat{\zeta}_l$, random walk size σ, total search iterations E, sample size S, mutation ratio u, population size P, fine-tune iterations $\hat{\tau}$
Output: $\boldsymbol{\alpha}, \boldsymbol{\kappa}$
Initialize *Pool* to a size P queue
for $e = 1$ **to** E **do**
  $\boldsymbol{\alpha} = \mathbf{1},\ \boldsymbol{\kappa} = \mathbf{0}$
  if *Pool* has S samples **then**
    V = *Pool*.sample(S)
    $\boldsymbol{\alpha}, \boldsymbol{\kappa}$ = argmaxFitness(V)
  end if
  Layer= Sample u% layers to mutate
  for $l \in$ *Layer* **do**
    std$_l$=computeStd([$M_i\ \forall\ i \in l$])
    $\boldsymbol{\alpha}_l = \boldsymbol{\alpha}_l \times \hat{\boldsymbol{\alpha}}_l$, where $\hat{\boldsymbol{\alpha}}_l \sim e^{\mathcal{N}(0,\sigma^2)}$
    $\boldsymbol{\kappa}_l = \boldsymbol{\kappa}_l + \hat{\boldsymbol{\kappa}}_l$, where $\hat{\boldsymbol{\kappa}}_l \sim \mathcal{N}(0, \text{std}_l)$
  end for
  $\hat{\mathbf{\Theta}}_l$ = LeGR-Pruning-and-fine-tuning($\boldsymbol{\alpha}, \boldsymbol{\kappa}$, $\hat{\zeta}_l$, $\hat{\tau}$, $\mathbf{\Theta}$)
  $Fitness = Acc_{val}(\hat{\mathbf{\Theta}}_l)$
  Pool.replaceOldestWith($\boldsymbol{\alpha}, \boldsymbol{\kappa}, Fitness$)
end for

Algorithm 2: Learning $\boldsymbol{\alpha}, \boldsymbol{\kappa}$ with regularized EA

$\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair, which is shown in Appendix B.2. We can equate each $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair to a network architecture obtained by LeGR-Pruning. Once a pruned architecture is obtained, we fine-tune the resulting architecture by $\hat{\tau}$ gradient steps and use its accuracy on the validation set[4] as the fitness (*i.e.*, validation accuracy) for the corresponding $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair. We note that we use $\hat{\tau}$ to approximate τ (fully fine-tuned steps) and we empirically find that $\hat{\tau} = 200$ gradient updates work well under the pruning settings across the datasets and networks we study. More concretely, we first generate a pool of candidates ($\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ values) and record the fitness for each candidate, and then repeat the following steps: (i) sample a subset from the candidates, (ii) identify the fittest candidate, (iii) generate a new candidate by mutating the fittest candidate and measure its fitness accordingly, and (iv) replace the oldest candidate in the pool with the generated one. To mutate the fittest candidate, we randomly select a subset of the layers *Layer* and conduct one step of random-walk from their current values, *i.e.*, $\alpha_l, \kappa_l\ \forall\ l \in$ *Layer*.

We note that our layer-wise affine transformation formulation (Eq. 5.1) can be interpreted from an optimization perspective. That is, one can upper-bound the loss difference between a pre-trained CNN and its pruned-and-fine-tuned counterpart by assuming Lipschitz continuity on the loss function, as detailed in Appendix B.1.

[4]We split 10% of the original training set to be used as validation set.

5.3 Experiments

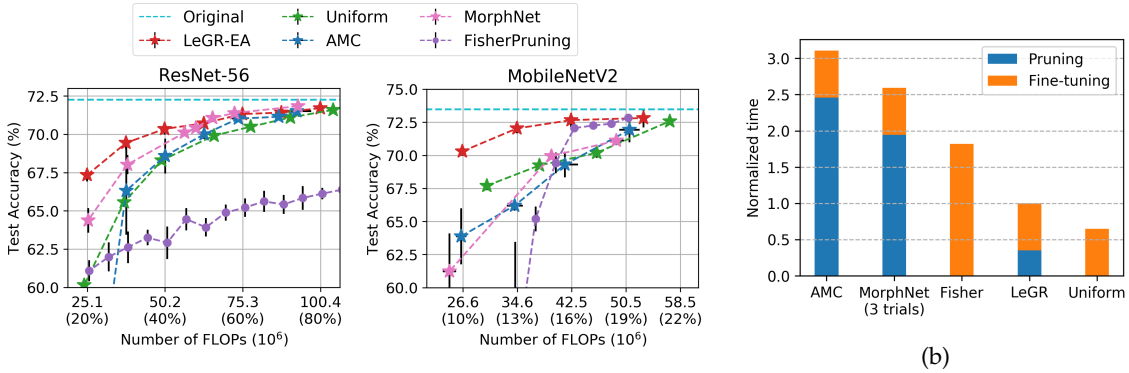
5.3.1 Datasets and Training Setting

Our work is evaluated on various image classification benchmarks including CIFAR-10/100 [93], ImageNet [153], and Birds-200 [170]. CIFAR-10/100 consists of 50k training images and 10k testing images with a total of 10/100 classes to be classified. ImageNet is a large scale image classification dataset that includes 1.2 million training images and 50k testing images with 1k classes to be classified. Also, we benchmark the proposed algorithm in a transfer learning setting since in practice, we want a small and fast model on some target datasets. Specifically, we use the Birds-200 dataset that consists of 6k training images and 5.7k testing images covering 200 bird species.

For Bird-200, we use 10% of the training data as the validation set used for early stopping and to avoid over-fitting. The training scheme for CIFAR-10/100 follows [70], which uses stochastic gradient descent with nesterov [134], weight decay $5e^{-4}$, batch size 128, $1e^{-1}$ initial learning rate with decrease by $5\times$ at epochs 60, 120, and 160, and train for 200 epochs in total. For control experiments with CIFAR-100 and Bird-200, the fine-tuning after pruning is done as follows: we keep all training hyper-parameters the same but change the initial learning rate to $1e^{-2}$ and train for 60 epochs (*i.e.*, $\tau \approx$ 21k). We drop the learning rate by $10\times$ at 30%, 60%, and 80% of the total epochs, *i.e.*, epochs 18, 36, and 48. To compare numbers with prior art on CIFAR-10 and ImageNet, we follow the number of iterations in [219]. Specifically, for CIFAR-10 we fine-tuned for 400 epochs with initial learning rate $1e^{-2}$, drop by $5\times$ at epochs 120, 240, and 320. For ImageNet, we use pre-trained models and we fine-tuned the pruned models for 60 epochs with initial learning rate $1e^{-2}$, drop by $10\times$ at epochs 30 and 45.

For the hyper-parameters of LeGR, we select $\hat{\tau} = 200$, *i.e.*, fine-tune for 200 gradient steps before measuring the validation accuracy when searching for the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair. We note that we do the same for AMC [71] for a fair comparison. Moreover, we set the number of architectures explored to be the same with AMC, *i.e.*, 400. We set mutation rate $u = 10$ and the hyper-parameter of the regularized evolutionary algorithm by following prior art [145]. In the following experiments, we use the smallest ζ considered as $\hat{\zeta}_l$ to search for the learnable variables $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$. The found $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair is used to obtain the pruned networks at various FLOP counts. For example, for ResNet-56 with CIFAR-100 (Figure 5.3a), we use $\hat{\zeta}_l = 20\%$ to obtain the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair and use the same $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair to obtain the seven networks ($\zeta = 20\%, ..., 80\%$) with the flow described in Figure 5.2. The ablation of $\hat{\zeta}_l$ and $\hat{\tau}$ are detailed in Sec. 5.4.2.

We prune filters across all the convolutional layers. We group dependent channels by summing up their importance measure and prune them jointly. The importance measure refers to the measure after learned affine transformations. Specifically, we group a channel in depth-wise convolution with its corresponding



(a) (b)

Figure 5.3: (a) The trade-off curve of pruning ResNet-56 and MobileNetV2 on CIFAR-100 using various methods. We average across three trials and plot the mean and standard deviation. (b) Training cost for seven CNNs across FLOP counts using various methods targeting ResNet-56 on CIFAR-100. We report the average cost considering seven FLOP counts, *i.e.*, 20% to 80% FLOP count in a step of 10% on NVIDIA GTX 1080 Ti. The cost is normalized to the cost of LeGR.

channel in the preceding layer. We also group channels that are summed together through residual connections.

5.3.2 CIFAR-100 Results

In this section, we consider ResNet-56 and MobileNetV2 and we compare LeGR mainly with four filter pruning methods, *i.e.*, MorphNet [59], AMC [71], FisherPruning [167], and a baseline that prunes filters uniformly across layers. Specifically, the baselines are determined such that one dominant approach is selected from different groups of prior art. We select one approach [59] from pruning-while-learning approaches, one approach [71] from pruning-by-searching methods, one approach [167] from continuous pruning methods, and a baseline extending magnitude-based pruning to various FLOP counts. We note that FisherPruning is a continuous pruning method where we use 0.0025 learning rate and perform 500 gradient steps after each filter pruned following [167].

As shown in Figure 5.3a, we first observe that FisherPruning does not work as well as other methods and we hypothesize the reason for it is that the small fixed learning rate in the fine-tuning phase makes it hard for the optimizer to get out of local optima. Additionally, we find that FisherPruning prunes away almost all the filters for some layers. On the other hand, we find that all other approaches outperform the uniform baseline in a high-FLOP-count regime. However, both AMC and MorphNet have higher variances when pruned more aggressively. In both cases, LeGR outperforms prior art, especially in the low-FLOP-count regime.

More importantly, our proposed method aims to alleviate the cost of pruning when the goal is to

explore the trade-off curve between accuracy and inference latency. From this perspective, our approach outperforms prior art by a significant margin. More specifically, we measure the average time of each algorithm to obtain the seven pruned ResNet-56 across the FLOP counts in Figure 5.3a using our hardware (*i.e.*, NVIDIA GTX 1080 Ti). Figure 5.3b shows the efficiency of AMC, MorphNet, FisherPruning, and the proposed LeGR. The cost can be broken down into two parts: (1) pruning: the time it takes to search for a network that has some pre-defined FLOP count and (2) fine-tuning: the time it takes for fine-tuning the weights of a pruned network. For MorphNet, we consider three trials for each FLOP count to find an appropriate hyper-parameter λ to meet the FLOP count of interest. The numbers are normalized to the cost of LeGR. In terms of pruning time, LeGR is $7\times$ and $5\times$ faster than AMC and MorphNet, respectively. The efficiency comes from the fact that LeGR only searches the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair once and re-uses it across FLOP counts. In contrast, both AMC and MorphNet have to search for networks for every FLOP count considered. FisherPruning always prune one filter at a time, and therefore the lowest FLOP count level considered determines the pruning time, regardless of how many FLOP count levels we are interested in.

5.3.3 Comparison with Prior Art

Although the goal of this chapter is to develop a model compression method that produces a set of CNNs across different FLOP counts, we also compare our method with prior art that focuses on generating a CNN for a specified FLOP count.

CIFAR-10 In Table 5.1, we compare LeGR with prior art that reports results on CIFAR-10. First, for ResNet-56, we find that LeGR outperforms most of the prior art in both FLOP count and accuracy dimensions and performs similarly to [70, 219]. For VGG-13, LeGR achieves significantly better results compared to prior art.

ImageNet Results For ImageNet, we prune ResNet-50 and MobileNetV2 with LeGR to compare with prior art. For LeGR, we learn the ranking using 47% FLOP count for ResNet-50 and 50% FLOP count for MobileNetV2, and use the learned ranking to obtain CNNs for other FLOP counts of interest. We have compared to 17 prior methods that report pruning performance for ResNet-50 and/or MobileNetV2 on the ImageNet dataset. While our focus is on the fast exploration of the speed and accuracy trade-off curve for filter pruning, our proposed method is better or comparable compared to the state-of-the-art methods as shown in Figure 5.4. The detailed numerical results are in Table 5.2. We would like to emphasize that to obtain a pruned-CNN with prior methods, one has to run the pruning algorithm for every FLOP count

Table 5.1: Comparison with prior art on CIFAR-10. We group methods into sections according to different FLOP counts. Values for our approaches are averaged across three trials and we report the mean and standard deviation. We use boldface to denote the best numbers and use * to denote our implementation. The accuracy is represented in the format of *pre-trained* $\mapsto$ *pruned-and-fine-tuned*.

Network	Method	Acc. (%)	MFLOP count
ResNet-56	PF [99]	93.0 → 93.0	90.9 (72%)
	Taylor [130]*	**93.9** → 93.2	90.8 (72%)
	LeGR	**93.9** → **94.1±0.0**	**87.8** (70%)
	DCP-Adapt [219]	93.8 → **93.8**	66.3 (53%)
	CP [74]	92.8 → 91.8	62.7 (50%)
	AMC [71]	92.8 → 91.9	62.7 (50%)
	DCP [219]	93.8 → 93.5	62.7 (50%)
	SFP [70]	93.6±0.6 → **93.4±0.3**	59.4 (47%)
	LeGR	**93.9** → **93.7±0.2**	**58.9** (47%)
VGG-13	BC-GNJ [118]	91.9 → 91.4	141.5 (45%)
	BC-GHS [118]	91.9 → 91	121.9 (39%)
	VIBNet [34]	91.9 → 91.5	70.6 (22%)
	LeGR	91.9 → **92.4±0.2**	70.3 (22%)

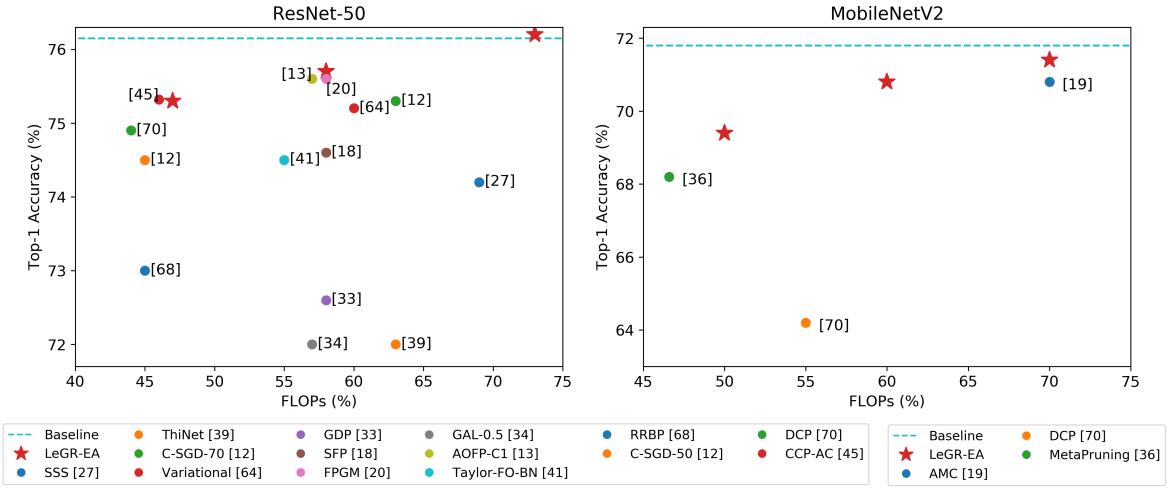


Figure 5.4: Results for ImageNet. LeGR is better or comparable compared to prior methods. Furthermore, its goal is to output a set of CNNs instead of one CNN.

considered. In contrast, our proposed method learns the ranking once and uses it to obtain CNNs across different FLOP counts.

5.3.4 Transfer Learning: Bird-200

We analyze how LeGR performs in a transfer learning setting where we have a model pre-trained on a large dataset, *i.e.*, ImageNet, and we want to transfer its knowledge to adapt to a smaller dataset, *i.e.*, Bird-200. We prune the fine-tuned network on the target dataset directly following the practice in prior art [207, 121]. We first obtain fine-tuned MobileNetV2 and ResNet-50 on the Bird-200 dataset with top-1

Table 5.2: Summary of pruning on ImageNet. The sections are defined based on the FLOP count left. The accuracy is represented in the format of *pre-trained* $\mapsto$ *pruned-and-fine-tuned*.

Network	Method	Top-1	Top-1 Diff	Top-5	Top-5 Diff	FLOP count (%)
ResNet-50	NISP [197]	- → -	-0.2	- → -	-	73
	LeGR	76.1 → **76.2**	**+0.1**	92.9 → **93.0**	**+0.1**	73
	SSS [82]	76.1 → 74.2	-1.9	92.9 → 91.9	-1.0	69
	ThiNet [121]	72.9 → 72.0	-0.9	91.1 → 90.7	-0.4	63
	C-SGD-70 [45]	75.3 → 75.3	+0.0	92.6 → 92.5	-0.1	63
	Variational [202]	75.1 → 75.2	+0.1	92.8 → 92.1	-0.7	60
	GDP [107]	75.1 → 72.6	-2.5	92.3 → 91.1	-1.2	58
	SFP [70]	76.2 → 74.6	-1.6	92.9 → 92.1	-0.8	58
	FPGM [72]	76.2 → 75.6	-0.6	92.9 → 92.6	-0.3	58
	LeGR	76.1 → **75.7**	-0.4	92.9 → **92.7**	-0.2	58
	GAL-0.5 [108]	76.2 → 72.0	-4.2	92.9 → 91.8	-1.1	57
	AOFP-C1 [46]	75.3 → 75.6	**+0.3**	92.6 → **92.7**	**+0.1**	57
	NISP [197]	- → -	-0.9	- → -	-	56
	Taylor-FO-BN [129]	76.2 → 74.5	-1.7	- → -	-	55
	CP [74]	- → -	-	92.2 → 90.8	-1.4	50
	SPP [175]	- → -	-	91.2 → 90.4	-0.8	50
	LeGR	76.1 → **75.3**	**-0.8**	92.9 → **92.4**	-0.5	47
	CCP-AC [140]	76.2 → **75.3**	-0.9	92.9 → 92.6	**-0.3**	44
	RRBP [213]	76.1 → 73.0	-3.0	92.9 → 91.0	-1.9	45
	C-SGD-50 [45]	75.3 → 74.5	**-0.8**	92.6 → 92.1	**-0.5**	45
	DCP [219]	76.0 → 74.9	-1.1	92.9 → 92.3	-0.6	44
MobileNetV2	AMC [71]	71.8 → 70.8	-1.0	→ **-**	**-**	70
	LeGR	71.8 → **71.4**	**-0.4**	→ **-**	**-**	70
	LeGR	71.8 → 70.8	-1.0	→ **-**	**-**	60
	DCP [219]	70.1 → 64.2	-5.9	→ -	-	55
	MetaPruning [115]	72.7 → 68.2	-4.5	→ -	-	50
	LeGR	71.8 → **69.4**	**-2.4**	→ **-**	**-**	50

accuracy 80.2% and 79.5%, respectively. These are comparable to the reported values in prior art [101, 123]. As shown in Figure 5.5, we find that LeGR outperforms Uniform and AMC, which is consistent with previous analyses in Section 5.3.2.

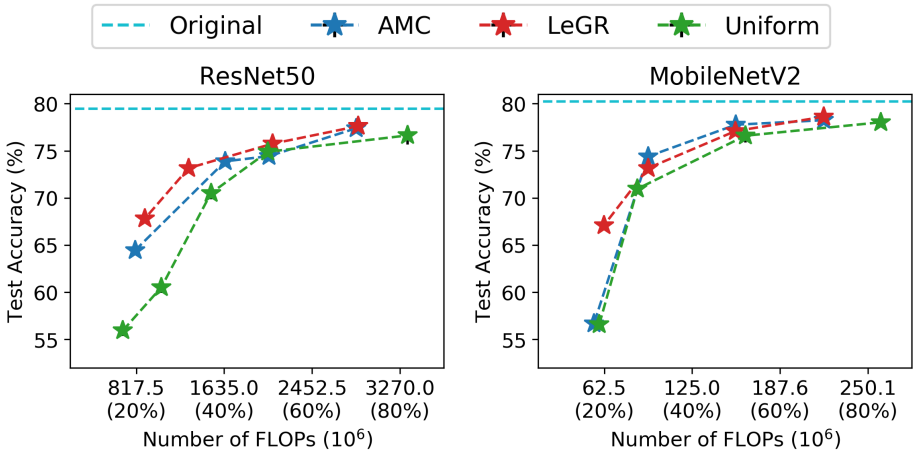


Figure 5.5: Results for Bird-200.

5.4 Ablation Study

5.4.1 Ranking Performance and $\hat{\zeta}_l$

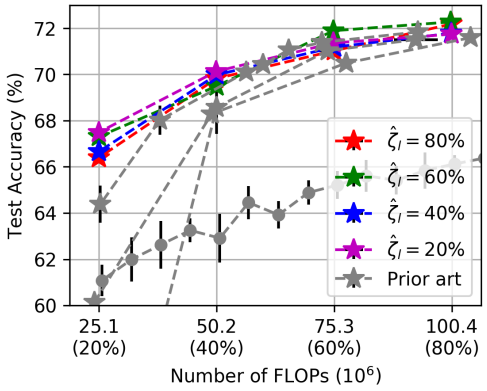


Figure 5.6: Robustness to the hyper-parameter $\hat{\zeta}_l$. Prior art is plotted as a reference (c.f. Figure 5.3a).

To learn the global ranking with LeGR without knowing the Pareto curve in advance, we use the minimum considered FLOP count ($\hat{\zeta}_l$) during learning to evaluate the performance of a ranking. We are interested in understanding how this design choice affects the performance of LeGR. Specifically, we try LeGR targeting ResNet-56 for CIFAR-100 with $\hat{\zeta}_l \in \{20\%, 40\%, 60\%, 80\%\}$. As shown in Figure 5.6, we first observe that rankings learned using different FLOP counts have similar performances, which empirically supports Assumption 5.2.1. More concretely, consider the network pruned to 40% FLOP count by using the ranking learned at 40% FLOP count. This case does not take advantage of the subset assumption because the entire learning process for learning $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ is done only by looking at the performance of the 40% FLOP count network. On the other hand, rankings learned using other FLOP counts but employed to obtain pruned-networks at 40% FLOP count have exploited the subset assumption (*e.g.*, the ranking learned for 80% FLOP count can produce a competitive network for 40% FLOP count). We find that LeGR with or without employing Assumption 5.2.1 results in similar performance for the pruned networks.

5.4.2 Fine-tuned Iterations

Since we use $\hat{\tau}$ to approximate τ when learning the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair, it is expected that the closer $\hat{\tau}$ to τ, the better the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair LeGR can find. We use LeGR to prune ResNet-56 for CIFAR-100 and learn $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ at three FLOP counts $\hat{\zeta}_l \in \{10\%, 30\%, 50\%\}$. We consider ζ to be exactly $\hat{\zeta}_l$ in this case. For $\hat{\tau}$, we experiment with $\{0, 50, 200, 500\}$. We note that once the $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ pair is learned, we use LeGR-Pruning to obtain the pruned CNN, fine-tune it for τ steps, and plot the resulting test accuracy. In this experiment, τ is set to 21120 gradient steps (60 epochs). As shown in Figure 5.7, the results align with our intuition in that there are

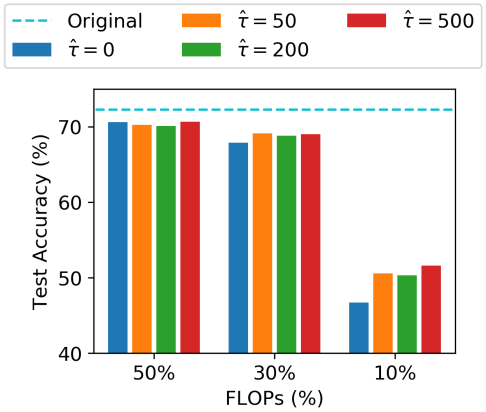


Figure 5.7: Pruning ResNet-56 for CIFAR-100 with LeGR by learning $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ using different $\hat{\tau}$ and FLOP count constraints.

diminishing returns in increasing $\hat{\tau}$. We observe that $\hat{\tau}$ affects the accuracy of the pruned CNNs more when learning the ranking at a lower FLOP count level, which means in low-FLOP-count regimes, the validation accuracy after fine-tuning a few steps might not be representative. This makes sense since when pruning away a lot of filters, the network can be thought of as moving far away from the local optimal, where the gradient steps early in the fine-tuning phase are noisy. Thus, more gradient steps are needed before considering the accuracy to be representative of the fully-fine-tuned accuracy.

5.4.3 FLOP count and Runtime

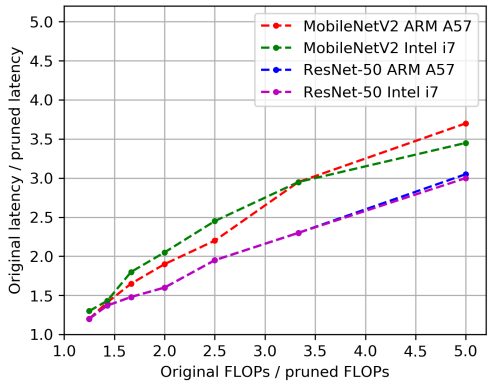


Figure 5.8: Latency reduction *vs.* FLOP count reduction. FLOP count reduction is indicative for latency reduction.

We demonstrate the effectiveness of filter pruning in wall-clock time speedup using ResNet-50 and MobileNetV2 on PyTorch 0.4 using two types of CPUs. Specifically, we consider both a desktop level CPU, *i.e.*, Intel i7, and an embedded CPU, *i.e.*, ARM A57, and use LeGR as the pruning methodology. The input is a single RGB image of size 224x224 and the program (Python with PyTorch) is run using a

single thread. As shown in Figure 5.8, filter pruning can produce near-linear acceleration (with a slope of approximately 0.6) without specialized software or hardware support.

5.5 Discussion

To alleviate the bottleneck of using model compression in optimizing the CNNs in a large system, we propose LeGR, a novel formulation for practitioners to explore the accuracy-*vs.*-speed trade-off efficiently via filter pruning. More specifically, we propose to learn layer-wise affine transformations over filter norms to construct a global ranking of filters. This formulation addresses the limitation that filter norms cannot be compared across layers in a learnable fashion and provides an efficient way for practitioners to obtain CNN architectures with different FLOP counts. Additionally, we provide a theoretical interpretation of the proposed affine transformation formulation. We conduct extensive empirical analyses using ResNet and MobileNetV2 on datasets including CIFAR, Bird-200, and ImageNet and show that LeGR has less training cost to generate the pruned CNNs across different FLOP counts compared to prior art while achieving comparable performance to state-of-the-art pruning methods.

5.6 Carbon Footprint Analysis

When compared to standard training of a neural network (*i.e.*, without pruning), LeGR saves carbon footprint during inference but increases it during training. During training, in addition to standard training, we conduct ranking learning and fine-tuning. If we take ImageNet as an example, it takes an extra 30% overhead for training. Considering an iso-accurate pruned ResNet-50, the pruned model reduces the computational overhead by 27%. According to a recent study by Patterson *et al.* [139], the ratio of training and inference is roughly nine to one in current cloud providers. With the above calculation, adopting the LeGR methodology can reduced the total carbon footprint by 21% compared to using a standard model. On the other hand, when compared to other pruning methods, LeGR reduces the carbon footprint during the training time while having similar inference overhead. More specifically, LeGR reduces the carbon footprint by 3$\times$ compared to AMC [71] during the training time.

Chapter 6

Joslim: Efficient Filter Pruning without Fine-tuning

In Chapter 5, we have proposed LeGR to improve the scalability of filter pruning. However, LeGR still requires fine-tuning each of the pruned models after pruning, which can be costly. In this chapter, we aim to remove the overhead of fine-tuning by sharing the weights among the pruned networks. From this perspective, we propose to jointly optimize a set of filter configurations, which have different accuracy and FLOPs profiles, and the weights that are shared among the configurations using alternating minimization.

6.1 Motivation

While LeGR can be efficient for filter pruning that aims for multiple compression ratios, it requires each of the pruned networks to be fine-tuned to achieve good performance. This can be limiting for various applications. For example, model compression is a useful tool to allow the designer to traverse the trade-off between accuracy and speed of the model. This characteristic can be useful if model compression can be carried out in an online fashion. More specifically, a real system might run multiple tasks at the same time, which makes it hard to guarantee the latency of a model since the information regarding other tasks are often unknown apriori. In this case, having a model that can be pruned at run-time without fine-tuning is beneficial for meeting the desired latency constraint. On the other hand, the fine-tuning process in LeGR not only adds overhead as the number of target compression ratios grows, but it also adds maintenance cost as the machine learning practitioner has to maintain n models for n target constraint levels and each of which have different weights. More specifically, this adds engineering costs if the training data is later improved and there will be n models to be fine-tuned or re-trained.

One idea to cope with the above drawbacks of LeGR is to train a weight-sharing network whose

weights are shared across all pruned models. Such a weight-sharing network can be pruned in an online fashion and only a single set of weights needs to be maintained. Along the direction of sharing weights across different networks that have different filter configurations, slimmable neural networks [196] have been proposed with the promise of enabling multiple neural networks with different trade-offs between prediction error and the FLOPs, *all at the storage requirement of only a single neural network*. This is in stark contrast to filter pruning methods [12, 193, 60] that aim for a small standalone model.

A slimmable neural network is trained by simultaneously considering networks with different widths (or filter counts) using a single set of shared weights. The width of a child network is specified by a real number between 0 and 1, which is known as the "width-multiplier" [77]. Such a parameter specifies how many filters per layer to use proportional to the full network. For example, a width-multiplier of $0.35\times$ represents a network with 35% of the channel counts of the full network for all the layers. While specifying child networks using a single width-multiplier for all the layers has shown empirical success [194, 196], such a specification neglects that different layers affect the network's output differently [200] and have different FLOPs and memory footprint requirements [59], which may lead to sub-optimal results. As an alternative, neural architecture search (NAS) methods such as BigNAS [195] optimizes the layer-wise widths for slimmable networks, however, a sequential greedy procedure is adopted to optimize the widths and weights. As a result, the optimization of weights is not adapted to the optimization of widths, thereby leaving rooms for improvement by joint width and weight optimization.

In this chapter, we propose a framework for optimizing slimmable nets by formalizing it as minimizing the area under the trade-off curve between prediction error and some metric of interest, *e.g.*, memory footprint or FLOPs, with alternating minimization. Our framework subsumes both the universally slimmable networks [194] and BigNAS [195] as special cases. The framework is general and provides us with insights to improve upon existing alternatives and justifies our new algorithm Joslim, the first approach that jointly optimizes both shared-weights and widths for slimmable nets. To this end, we demonstrate empirically the superiority of the proposed algorithm over existing methods using various datasets, networks, and objectives. We visualize the algorithmic differences between the proposed method and existing alternatives in Fig. 6.1.

The contributions of this chapter are as follows:

- We propose a general framework that enables the joint optimization of the widths and their corresponding shared weights of a *slimmable net*. The framework is general and subsumes existing algorithms as special cases.

- We propose Joslim, an algorithm that jointly optimizes the widths and weights of slimmable nets.

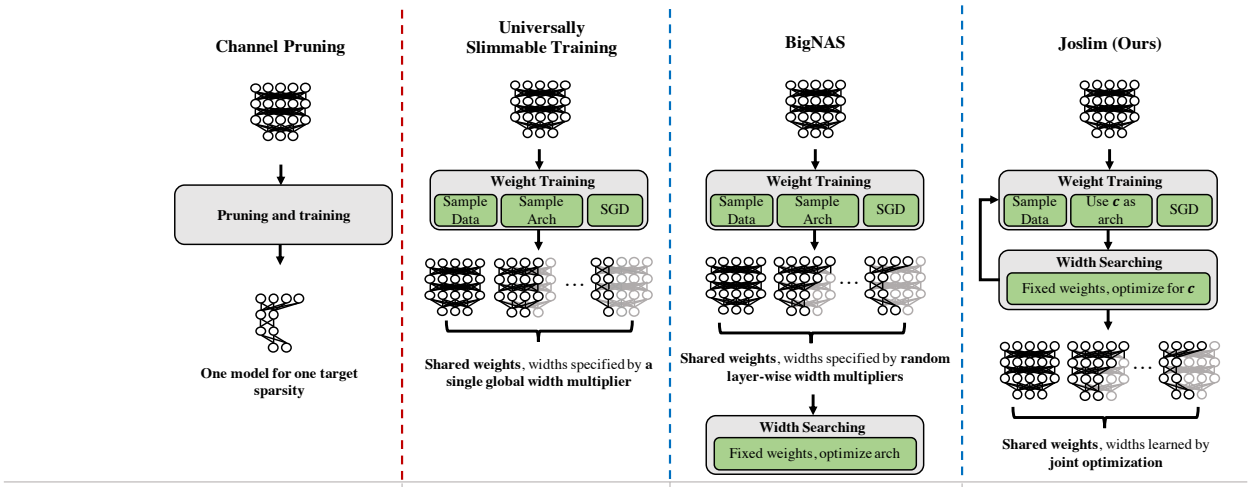


Figure 6.1: Schematic overview comparing our proposed method with existing alternatives and channel pruning. Channel pruning has a fundamentally different goal compared to ours, *i.e.*, training slimmable nets. Joslim jointly optimizes both the widths and the shared weights.

We show empirically that Joslim outperforms existing methods on various networks, datasets, and objectives. Quantitatively, improvements up to 1.7% and 8% in top-1 accuracy on ImageNet are attained for MobileNetV2 considering FLOPs and memory footprint, respectively.

6.2 Methodology

In this chapter, we are interested in jointly optimizing the network widths and network weights. Ultimately, when evaluating the performance of a slimmable neural network, we care about the trade-off curve between multiple objectives, *e.g.*, theoretical speedup and accuracy. This trade-off curve is formed by evaluating the two objectives at multiple width configurations using the same shared-weights. Viewed from this perspective, both the widths and shared-weights should be optimized in such a way that the resulting networks have a better trade-off curve (*i.e.*, larger area under curve). This section formalizes this idea and provides an algorithm to solve it in an approximate fashion.

6.2.1 Problem formulation

Our goal is to find both the weights and the width configurations that optimize the area under the trade-off curve between two competing objectives, *e.g.*, accuracy and inference speed. Without loss of generality, we use cross entropy loss as the accuracy objective and FLOPs as the inference speed objective throughout

the text for clearer context. Note that FLOPs can also be replaced by other metrics of interest such as memory footprint. Since in this case both objectives are better when lower, the objective for optimizing the slimmable nets becomes to *minimize* the area under curve. To quantify the area under curve, we start with a Riemann integral. Let $w(c)$ be a width configuration of c FLOPs, one can quantify the Riemann integral by evaluating the cross entropy loss $L_{\mathcal{S}}$ on the training set $\mathcal{S}$ using the shared weights $\boldsymbol{\theta}$ for the architectures that spread uniformly on the FLOPs-axis between a lower bound l and an upper bound u of FLOPs: $\{\boldsymbol{a}|\boldsymbol{a} = w(c), c \in [l, u]\}$. More formally, the area under curve $\mathbb{A}$ for the widths $w(\cdot)$ and weights $\boldsymbol{\theta}$ is characterized as

$$\mathbb{A}(\boldsymbol{\theta}, w) \stackrel{\text{def}}{=} \int_l^u L_{\mathcal{S}}\left(\boldsymbol{\theta}, w(c)\right) dc \tag{6.1}$$

$$\approx \sum_{i=0}^{N} L_{\mathcal{S}}\left(\boldsymbol{\theta}, w(c_i)\right) \delta, \tag{6.2}$$

where equation 6.2 approximates the Riemann integral with the Riemann sum using N architectures that spread uniformly on the FLOPs-axis with a step size δ. With a quantifiable area under curve, our goal for optimizing slimmable neural networks becomes finding both the shared-weights $\boldsymbol{\theta}$ and the architecture function w to minimize their induced area under curve:

$$\arg\min_{\boldsymbol{\theta}, w} \mathbb{A}(\boldsymbol{\theta}, w) \approx \arg\min_{\boldsymbol{\theta}, w} \sum_{i=0}^{N} L_{\mathcal{S}}\left(\boldsymbol{\theta}, w(c_i)\right) \delta \tag{6.3}$$

$$= \arg\min_{\boldsymbol{\theta}, w} \frac{1}{N} \sum_{i=0}^{N} L_{\mathcal{S}}\left(\boldsymbol{\theta}, w(c_i)\right) \tag{6.4}$$

$$\approx \arg\min_{\boldsymbol{\theta}, w} \mathbb{E}_{c \sim U(l,u)} L_{\mathcal{S}}\left(\boldsymbol{\theta}, w(c)\right), \tag{6.5}$$

where $U(l, u)$ denotes a uniform distribution over a lower bound l and an upper bound u. Note that the solution to equation 6.5 is the shared-weight vector and a set of architectures, which is drastically different from the solution to the formulation used in the NAS literature [111, 162], which is an architecture.

6.2.2 Proposed approach: Joslim

Since both the shared-weights $\boldsymbol{\theta}$ and the architecture function w are optimization variables of two natural groups, we start by using alternating minimization:

$$w^{(t+1)} = \arg\min_{w} \mathbb{E}_{c \sim U(l,u)} L_{\mathcal{S}}\left(\boldsymbol{\theta}^{(\boldsymbol{t})}, w(c)\right) \tag{6.6}$$

$$\boldsymbol{\theta}^{(\boldsymbol{t+1})} = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{c \sim U(l,u)} L_{\mathcal{S}}\left(\boldsymbol{\theta}, w^{(t+1)}(c)\right). \tag{6.7}$$

In equation 6.6, we maintain the shared-weights $\boldsymbol{\theta}$ fixed and for each FLOPs between l and u, we search for a corresponding architecture that minimizes the cross entropy loss. This step can be seen as a multi-

objective neural architecture search given a fixed set of pre-trained weights, and can be approximated using smart algorithms such as multi-objective Bayesian optimization [137] or evolutionary algorithms [39]. However, even with smart algorithms, such a procedure can be impractical for every iteration of the alternating minimization.

In equation 6.7, one can use stochastic gradient descent by sampling from a set of architectures that spread uniformly across FLOPs obtained from solving equation 6.6. However, training such a weight-sharing network is practically $4\times$ the training time of the largest standalone subnetwork [194] (it takes 6.5 GPU-days to train a slimmable ResNet18), which prevents it from being adopted in the alternating minimization framework.

To cope with these challenges, we propose targeted sampling, local approximation, and temporal sharing to approximate both equations.

Targeted sampling

We propose to sample a set of FLOPs to approximate the expectation in equations 6.6 and 6.7 with empirical estimates. Moreover, the sampled FLOPs are shared across both steps in the alternating minimization so that one does not have to solve for the architecture function w (needed for the second step), but only solve for a set of architectures that have the corresponding FLOPs. Specifically, we approximate the expectation in both equations 6.6 and 6.7 with the sample mean:

$$c_i^{(t)} \sim U(l, u) \quad \forall\, i = 1, \ldots, M \tag{6.8}$$

$$w^{(t+1)} \approx \arg\min_{w} \frac{1}{M} \sum_{i=1}^{M} L_{\mathcal{S}} \left(\boldsymbol{\theta}^{(\boldsymbol{t})}, w(c_i^{(t)}) \right) \tag{6.9}$$

$$\boldsymbol{\theta}^{(\boldsymbol{t+1})} \approx \arg\min_{\boldsymbol{\theta}} \frac{1}{M} \sum_{i=1}^{M} L_{\mathcal{S}} \left(\boldsymbol{\theta}, w^{(t+1)}(c_i^{(t)}) \right). \tag{6.10}$$

From equation 6.9 and 6.10, we can observe that at any timestamp t, we only query the architecture function $w^{(t)}$ and $w^{(t+1)}$ at a fixed set of locations $c_i \; \forall\, i = 1, \ldots, M$. As a result, instead of solving for the architecture function w, we can solve for a fixed set of architectures $\mathcal{W}^{(t+1)}$ at each timestamp as follows:

$$\mathcal{W}^{(t+1)} := \{ w^{(t+1)}(c_i), \ldots, w^{(t+1)}(c_M) \} \tag{6.11}$$

where

$$w^{(t+1)}(c_i) = \arg\min_{\boldsymbol{a}} L_{\mathcal{S}} \left(\boldsymbol{\theta}^{(\boldsymbol{t})}, \boldsymbol{a} \right) \qquad \text{s.t. FLOPs}(\boldsymbol{a}) = c_i. \tag{6.12}$$

With these approximations, for each iteration in the alternating minimization, we solve for M architectures with targeted FLOPs as opposed to solving for the entire approximate trade-off curve.

Local approximation

To reduce the overhead for solving equation 6.10, we propose to approximate it with a few steps of gradient descent. Specifically, instead of training a slimmable neural network with sampled architectures until convergence in each iteration of alternating minimization (equation 6.10), we propose to perform K steps of gradient descent:

$$\begin{aligned} \boldsymbol{x}^{\mathbf{0}} &\overset{\text{def}}{=} \boldsymbol{\theta}^{(t)} \\ \boldsymbol{x}^{(k+1)} &\overset{\text{def}}{=} \boldsymbol{x}^{(k)} - \eta \frac{1}{M} \sum_{i=1}^{M} \nabla_{\boldsymbol{x}^{(k)}} L_{\mathcal{S}} \left(\boldsymbol{x}^{(k)}, \mathcal{W}_i^{(t+1)} \right) \\ \boldsymbol{\theta}^{(t+1)} &\approx \boldsymbol{x}^{(K)}, \end{aligned} \tag{6.13}$$

where η is the learning rate. Larger K indicates better approximation with higher training overhead.

Temporal sharing

Since we use local approximation, $\boldsymbol{\theta}^{(t+1)}$ and $\boldsymbol{\theta}^{(t)}$ would not be drastically different. As a result, instead of performing constrained neural architecture search *from scratch* (*i.e.*, solving for equation 6.12) in every iteration of the alternating minimization, we propose to share information across the search procedures in different iterations of the alternation.

To this end, we propose to perform temporal sharing for multi-objective Bayesian optimization with random scalarization (MOBO-RS) [137] to solve equation 6.12. MOBO-RS itself is a sequential model-based optimization algorithm, where one takes a set of architectures $\mathcal{H}$, builds models (typically Gaussian Processes [143]) to learn a mapping from architectures to cross entropy loss g_{CE} and FLOPs g_{FLOPs}, scalarizes both models into a single objective with a random weighting λ (λ controls the preference for cross entropy and FLOPs), and finally optimizes the scalarized model to obtain a new architecture and stores the architecture back to the set $\mathcal{H}$. This entire procedure repeats for T iterations for one MOBO-RS.

To exploit temporal similarity, we propose MOBO-TS2, which stands for multi-objective Bayesian optimization with targeted scalarization and temporal sharing. Specifically, we propose to let $T = 1$ and share $\mathcal{H}$ across alternating minimization. Importantly, we replace the random scalarization with targeted scalarization where we use binary search to search for the λ that results in the desired FLOPs. As such, $\mathcal{H}$ grows linearly with the number of alternations. In such an approximation, for each MOBO in the alternating optimization, we reevaluate the cross-entropy loss for each $\boldsymbol{a} \in \mathcal{H}$ to build faithful GPs. We further provide theoretical analysis for approximation via temporal similarity for Bayesian optimization in Appendix C.4.

Input : Model parameters $\boldsymbol{\theta}$, lower bound for width-multipliers $w_0 \in [0, 1]$, number of full iterations F, number of gradient descent updates K, number of λ samples M
Output: Trained parameter $\boldsymbol{\theta}$, approximate Pareto front $\mathcal{N}$

$\mathcal{H} = \{\}$ (*Historical minimizers* $\boldsymbol{a}$)
for i = *1...F* **do**
  $\boldsymbol{x}, y$ = sample_data()
  $\boldsymbol{u}_{\text{CE}}$, $\boldsymbol{u}_{\text{FLOPs}}$ = $L_{\text{CE}}(\mathcal{H}; \boldsymbol{\theta}, \boldsymbol{x}, y)$, FLOPs($\mathcal{H}$)
  $\boldsymbol{g}_{\text{CE}}$, $\boldsymbol{g}_{\text{FLOPs}}$ = GP_UCB($\mathcal{H}$, $\boldsymbol{u}_{\text{CE}}$, $\boldsymbol{u}_{\text{FLOPs}}$)
  widths = []
  for m = *1...M* **do**
    $\boldsymbol{a}$ = MOBO_TS2($\boldsymbol{g}_{\text{CE}}, \boldsymbol{g}_{\text{FLOPs}}, \mathcal{H}$) (*Algorithm 4*)
    widths.append($\boldsymbol{a}$)
  end
  $\mathcal{H} = \mathcal{H} \cup$ widths (*update historical data*)
  widths.append($\boldsymbol{w_0}$)
  for j = *1...K* **do**
    SlimmableTraining($\boldsymbol{\theta}$, widths)
    (*line 3-16 of Algorithm 1 in [194]*)
  end
  $\mathcal{N}$=nonDominatedSort($\mathcal{H}$, $\boldsymbol{u}_{\text{CE}}$, $\boldsymbol{u}_{\text{FLOPs}}$)
end

Algorithm 3: Joslim

Input : Acquisition functions $\boldsymbol{g}_{\text{CE}}, \boldsymbol{g}_{\text{FLOPs}}$, historical data $\mathcal{H}$, search precision ϵ
Output: channel configurations $\boldsymbol{a}$

c = Uniform(l, u) (*Sample a target FLOPs*)
λ_{FLOPs}, λ_{min}, λ_{max} = 0.5, 0, 1
while $|\frac{FLOPs(\boldsymbol{a})-c}{FullModelFLOPs}| > \epsilon$ **do** `// binary search`
  $\boldsymbol{c}$ =arg min$_c$ Scalarize(λ_{FLOPs}, $\boldsymbol{g}_{\text{CE}}, \boldsymbol{g}_{\text{FLOPs}}$)
  if $FLOPs(\boldsymbol{a}) > c$ **then**
    $\lambda_{\text{min}} = \lambda_{\text{FLOPs}}$
    $\lambda_{\text{FLOPs}} = (\lambda_{\text{FLOPs}} + \lambda_{\text{max}})/2$
  else
    $\lambda_{\text{max}} = \lambda_{\text{FLOPs}}$
    $\lambda_{\text{FLOPs}} = (\lambda_{\text{FLOPs}} + \lambda_{\text{min}})/2$
  end
end

Algorithm 4: MOBO-TS2

Joslim

Based on this preamble, we present our algorithm, Joslim, in Algorithm 3. In short, Joslim has three steps: (1) build surrogate functions (*i.e.*, GPs) and acquisition functions (*i.e.*, UCBs) using historical data $\mathcal{H}$ and their function responses, (2) sample M target FLOPs and solve for the corresponding widths (*i.e.*, $\boldsymbol{a}$) via binary search with the scalarized acquisition function and store them back to $\mathcal{H}$, and (3) perform K gradient descent steps using the solved widths. The first two steps solve equation 6.12 with targeted sampling and temporal sharing, and the final step solves equation 6.10 approximately with local approximation. In the end, to obtain the best widths, we use non-dominated sorting based on the training loss and FLOPs for $a \in \mathcal{H}$.

6.2.3 Relation to existing approaches

For direct comparisons with our method we consider the universally slimmable neural networks [194], which uses a single width multiplier to specify the widths of a slimmable network and NAS-based approaches such as OFA [15] and BigNAS [195], which use a sequential greedy process for weights and widths optimization. To demonstrate the generality of the proposed framework, we show how these previously published works are special cases of our framework.

Slim

Universally slimmable networks [194], or Slim for short, is a special case of our framework where the widths are not optimized but pre-specified by a single global width multiplier. This corresponds to solving equation 6.5 with w given as a function that returns the width that satisfies some FLOPs by controlling a single global width multiplier. Our framework is more general as it introduces the freedom for optimizing the widths of slimmable nets.

OFA and BigNAS

OFA and BigNAS use the same approach when it comes to the channel search space[1]. They are also a special case of our framework where the optimization of the widths and the shared-weights are carried out greedily. Specifically, BigNAS first trains the shared-weights by random layer-wise width multipliers. After convergence, BigNAS performs evolutionary search to optimize the layer-wise width multipliers considering both error and FLOPs. This greedy algorithm can be seen as performing one iteration of alternating minimization by solving equation 6.7 followed by solving equation 6.6. From this perspective, one can observe that the shared-weights $\boldsymbol{\theta}$ are not jointly optimized with the widths. Our framework is more general and enables joint optimization for both widths and weights.

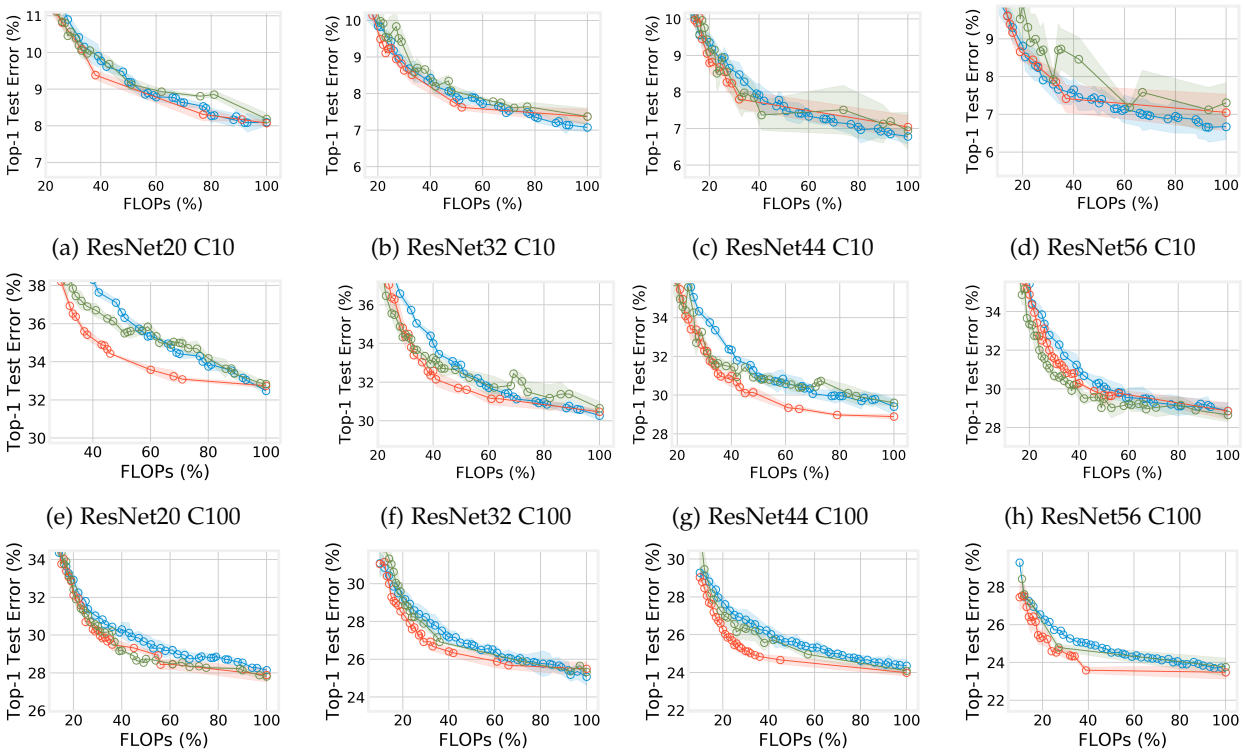
As we demonstrate in Section 6.3.2, our comprehensive empirical analysis reveals that Joslim is superior to either approach when compared across multiple networks, datasets, and objectives.

6.3 Experiments

6.3.1 Experimental setup

For all the Joslim experiments in this sub-section, we set K such that Joslim only visits 1000 width configurations throughout the entire training ($|\mathcal{H}| = 1000$). Also, we set M to be 2, which follows the conventional

[1]Since we only search for channel counts, the progressive shrinking strategy proposed in OFA does not apply. As a result, both OFA and BigNAS have the same approach.



(a) ResNet20 C10 (b) ResNet32 C10 (c) ResNet44 C10 (d) ResNet56 C10

(e) ResNet20 C100 (f) ResNet32 C100 (g) ResNet44 C100 (h) ResNet56 C100

(i) 2×ResNet20 C100 (j) 3×ResNet20 C100 (k) 4×ResNet20 C100 (l) 5×ResNet20 C100

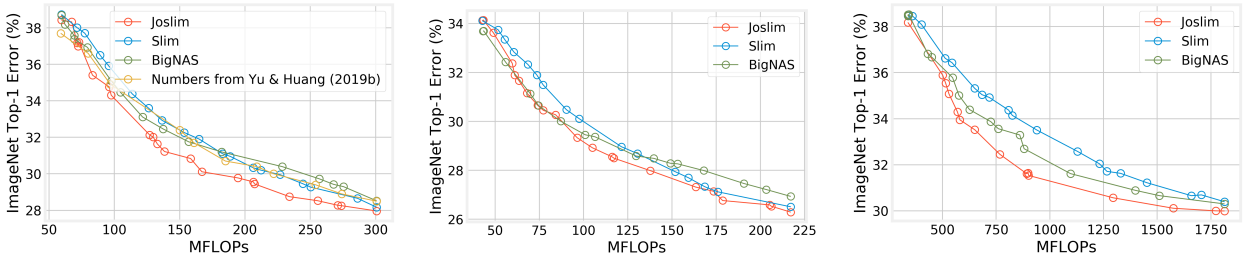
Figure 6.2: Comparisons among Slim, BigNAS, and Joslim. C10 and C100 denote CIFAR-10/100. We perform three trials for each method and plot the mean and standard deviation.

slimmable training method [194] that samples two width configurations in between the largest and the smallest widths. As for binary search, we conduct at most 10 binary searches with ϵ set to 0.02, which means that the binary search terminates if the FLOPs difference is within a two percent margin relative to the full model FLOPs. On average, the procedure terminates by using 3.4 binary searches for results on ImageNet. The dimension of $\boldsymbol{a}$ is network-dependent and is specified in Appendix C.1 and the training hyperparameters are detailed in Appendix C.3. To arrive at the final set of architectures for Joslim, we use non-dominated sort based on the training loss and FLOPs for $\boldsymbol{a} \in \mathcal{H}$.

6.3.2 Performance gains introduced by Joslim

We consider three datasets: CIFAR-10, CIFAR-100, and ImageNet. To provide informative comparisons, we verify our implementation for the conventional slimmable training with the reported numbers in [194] using MobileNetV2 on ImageNet. Our results follow closely to the reported numbers as shown in Fig. 6.3a, which makes our comparisons on other datasets convincing.

We compare to the following baselines:



(a) MobileNetV2 (b) MobileNetV3 (c) ResNet18

Figure 6.3: Comparisons among Slim, BigNAS, and Joslim on ImageNet.

MobileNetV2				MobileNetV3				ResNet18			
MFLOPs	Slim	BigNAS	Joslim	MFLOPs	Slim	BigNAS	Joslim	MFLOPs	Slim	BigNAS	Joslim
59	61.4	61.3	**61.5**	43	65.8	**66.3**	65.9	339	61.5	61.5	**61.8**
84	63.0	63.1	**64.6**	74	68.1	68.1	**68.8**	513	63.4	64.2	**64.5**
102	64.7	**65.5**	**65.5**	85	69.1	**70.0**	**70.0**	650	64.7	65.6	**66.5**
136	67.1	67.5	**68.2**	118	71.0	**71.4**	**71.4**	718	65.1	66.1	**67.5**
149	67.6	68.2	**69.1**	135	71.5	71.5	**72.1**	939	66.5	67.3	**68.5**
169	68.2	68.8	**69.9**	169	72.7	72.0	**72.8**	1231	68.0	68.4	**69.4**
212	69.7	69.6	**70.6**	184	73.0	72.5	**73.2**	1659	69.3	69.3	**69.9**
300	71.8	71.5	**72.1**	217	73.5	73.1	**73.7**	1814	69.6	69.7	**70.0**

Table 6.1: Comparing the top-1 accuracy among Slim, BigNAS, and Joslim on ImageNet. Bold represents the highest accuracy of a given FLOPs.

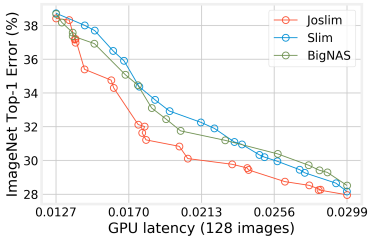


Figure 6.4: A latency-*vs*.-error view of Fig. 6.3a.

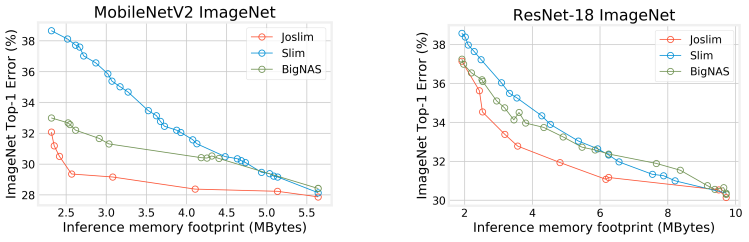
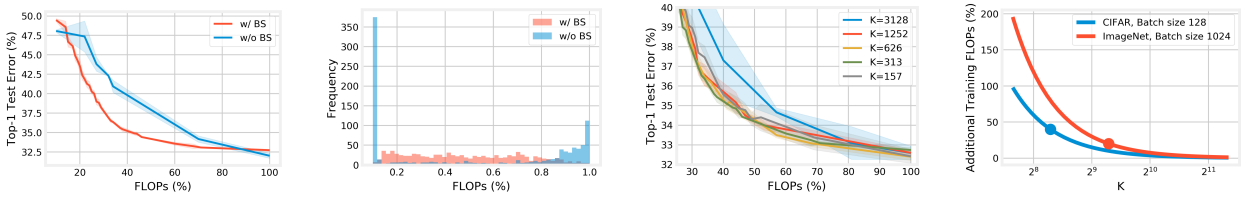


Figure 6.5: Prediction error *vs*. inference memory footprint for MobileNetV2 and ResNet18 on ImageNet.

- **Slim**: the conventional slimmable training method (the universally slimmable networks by [194]). We select 40 architectures uniformly distributed across FLOPs and run a non-dominated sort using training loss and FLOPs.

- **BigNAS**: disjoint optimization that first trains the shared-weights, then uses search methods to find architectures that work well given the trained weights (similar to OFA [15]). To compare fairly with Joslim, we use MOBO-RS for the search. After optimization, we run a non-dominated sort for all the visited architectures $\mathcal{H}$ using training loss and FLOPs.

The main results for the CIFAR dataset are summarized in Fig. 6.2 with results on ImageNet summarized in Figure 6.3 and Table 6.1. Compared to *Slim*, the proposed Joslim has demonstrated much



(a) Impact of binary search (BS).

(b) Histogram of FLOPs for $\mathcal{H}$ w/ and w/o BS.

(c) Performance for different K.

(d) Additional overhead over Slim for different K.

Figure 6.6: Ablation study for the introduced binary search and the number of gradient descent updates per full iteration using ResNet20 and CIFAR-100. Experiments are conducted three times and we plot the mean and standard deviation.

better results across various networks and datasets. This suggests that channel optimization can indeed improve the efficiency of slimmable networks. Compared to *BigNAS*, Joslim is better or comparable across networks and datasets. This suggests that joint widths and weights optimization leads to better overal performance for slimmable nets. From the perspective of training overhead, Joslim introduced minor overhead compared to Slim due to the temporal similarity approximation. More specifically, on ImageNet, Joslim incurs approximately 20% extra overhead compared to Slim.

Note that the performance among these three methods are similar for the CIFAR-10 dataset. This is plausible since when a network is more over-parameterized, there are many solutions to the optimization problem and it is easier to find solutions with the constraints imposed by weight sharing. In contrast, when the network is relatively less over-parameterized, compromises have to be made due to the constraints imposed by weight sharing. In such scenarios, Joslim outperforms Slim significantly, as it can be seen in CIFAR-100 and ImageNet experiments. We conjecture that this is because Joslim introduces a new optimization variable (width-multipliers), which allows better compromises to be attained. Similarly, from the experiments with ResNets on CIFAR-100 (Fig. 6.2e to Fig. 6.2h), we find that shallower models tend to benefit more from joint channel and weight optimization than their deeper counterparts.

Interestingly, while it might be intuitive that larger models are more amenable to pruning without accuracy degradation, we find that ResNet18, MobileNetV2, and MobileNetV3 dropped their accuracy similarly while ResNet18 has much more parameters than MobileNetV2 and MobileNetV3 as can be seen from Figure 6.3. This suggests that we may be able to build training algorithms to better utilize the parameter counts of a model.

As FLOPs may not necessarily reflect latency improvements since FLOP does not capture memory accesses, we in addition plot latency-*vs.*-error for the data in Fig. 6.3a in Fig. 6.4. The latency is measured on a single V100 GPU using a batch size of 128. When visualized in latency, Joslim still performs favorably compared to Slim and BigNAS for MobileNetV2 on ImageNet.

Lastly, we consider another objective that is critical for on-device machine learning, *i.e.*, inference memory footprint [196]. Inference memory footprint decides whether a model is executable or not on memory-constrained devices. Since Joslim is general, we can replace the FLOPs calculation with memory footprint calculation to optimize for memory-*vs.*-error. As shown in Fig. 6.5, Joslim significantly outperform other alternatives. Notably, Joslim outperforms Slim by up to 8% top-1 accuracy for MobileNetV2. Such a drastic improvement comes from the fact that memory footprint depends mostly on the largest layers. As a result, slimming all the layers equally to arrive at networks with smaller memory footprint (as done in Slim) is less than ideal since only one layer contributes to the reduced memory. In addition, when comparing Joslim with BigNAS, we can observe significant improvements as well, *i.e.*, around 2% top-1 accuracy improvements for MobileNetV2, which demonstrates the effectiveness of joint width and weights optimization.

6.3.3 Ablation studies

In this subsection, we ablate the hyperparameters that are specific to Joslim to understand their impact. We use ResNet20 and CIFAR-100 for the ablation with the results summarized in Fig. 6.6.

Binary search

Without binary search, one can also consider sampling the scalarization weighting λ uniformly from $[0, 1]$, which does not require any binary search and is easy to implement. However, the issue with this sampling strategy is that uniform sampling λ does not necessarily imply uniform sampling in the objective space, *e.g.*, FLOPs. As shown in Fig. 6.6a and Fig. 6.6b, sampling directly in the λ space results in non-uniform FLOPs and worse performance compared to binary search.

Number of gradient descent steps

In the approximation, the number of architectures ($|\mathcal{H}|$) is affected by the number of gradient descent updates K. In previous experiments for CIFAR, we have $K = 313$, which results in $|\mathcal{H}| = 1000$. Here, we ablate K to $156, 626, 1252, 3128$ such that $|\mathcal{H}| = 2000, 500, 250, 100$, respectively. Given a fixed training epoch and batch size, Joslim produces a better approximation for equation 6.10 but a worse approximation for equation 6.9 with larger K. The former is because of the local approximation while the latter is because there are overall fewer iterations put into Bayesian optimization due to temporal sharing. As shown in Fig. 6.6c, we observe worse results with higher K. On the other hand, the improvement introduced by

lower K saturates quickly. The overhead of Joslim as a function of K compared to Slim is shown in Fig. 6.6d where the dots are the employed K.

6.4 Discussion

In this chapter, we are interested in developing a method for filter pruning without fine-tuning. To achieve this goal, we consider training a network such that its weights are shared across various pruned models. We formalize this idea by searching for both the shared weights and the width configurations that minimize the area under the trade-off curve between cross entropy and FLOPs (or memory footprint) with alternating minimization. We further show that the proposed framework subsumes existing methods as special cases and provides flexibility for devising better algorithms. To this end, we propose Joslim, an algorithm that jointly optimizes the weights and widths of slimmable nets, which empirically outperforms existing alternatives that either neglect width optimization or conduct widths and weights optimization independently. We extensively verify the effectiveness of Joslim over existing techniques on 15 dataset and network combinations and two types of objectives, *i.e.*, FLOPs and memory footprint. Our results highlight the feasibility of removing fine-tuning in filter pruning.

6.5 Carbon Footprint Analysis

When compared to standard training of a neural network (*i.e.*, without pruning), our methodology saves carbon footprint during inference but increases it during training. During training, in contrast to standard training, we conduct joint width and weight optimization. If we take ImageNet as an example, it takes an extra 4.2× overhead for training. An iso-accurate pruned MobileNetV2, model reduces the computational overhead by 20%. According to a recent study by Patterson *et al.* [139], the ratio of training and inference is roughly nine to one in current cloud providers. This leads to a 14% increased in carbon footprint. However, Joslim trains a weight-sharing model which may in turn reduce the frequency of training. Joslim ends up saving overall carbon footprint compared to standard training if we assume training a weight-sharing network can reduce the training frequency of a standard model by 4× (*i.e.*, instead of training four standalone models of different sizes, we can now train a single slimmable model). More specifically, adopting the Joslim methodology can reduce the total carbon footprint by 11% compared to using a standard model. On the other hand, when compared to other weight-sharing methods (*i.e.*, BigNAS), Joslim increases carbon footprint at training but reduces it during inference since Joslim can produce a smaller model compared BigNAS when compared at the same top-1 accuracy. More specifically, with the

nine-to-one ratio for training and inference, Joslim ends up saving 20% carbon footprint when considering ResNet18 on ImageNet with the top-1 accuracy being 69%.

Chapter 7

Width Transfer: On the (In)variance of Filter Count Optimization

Inspired by Liu *et al.* [116], many filter pruning approaches are developed from a neural architecture search perspective [60, 160, 12, 193]. More specifically, Liu *et al.* have found that the resulting neural architecture after filter pruning play an important role in the effectiveness of filter pruning. That is, the goal is to identify layer-wise filter counts as opposed to finding the redundant weights to prune. Inspired by this perspective, in this chapter, we design a novel method for optimizing the filter counts of a neural network efficiently by assuming some invariance properties for the filter count (or width) optimization problem.

7.1 Motivation

Better designs for the number of filters for each layer of a CNN can lead to improved test performance for image classification without requiring additional FLOPs during the forward pass at test time. Specifically, by optimizing the channel widths, improvements of up to 2% top-1 accuracy for image classification on ImageNet can be achieved without additional FLOPs [60, 59, 193, 27, 103]. However, designing the layer by layer width multipliers for efficient CNNs is a non-trivial task that often requires intuition and domain expertise together with trial-and-error to do well. To alleviate the labor-intensive trial-and-error procedure, width optimization algorithms have been proposed [115, 71, 60, 49, 59] to automatically determine the width of a convolutional neural network. A width optimization algorithm takes as input an initial network and a training dataset, and outputs a set of optimized widths for each layer. When these optimized widths are applied to the initial network and train the network from scratch, one can achieve better validation accuracy compared to training a network of the original widths without incurring additional test-time

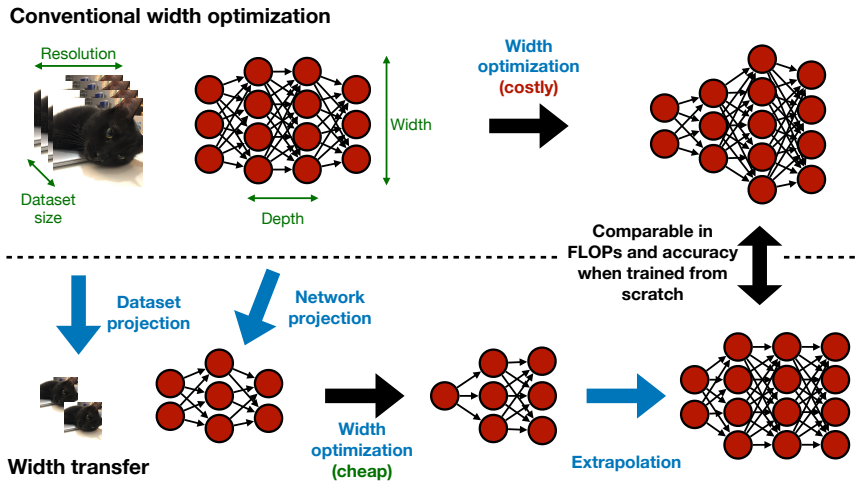


Figure 7.1: The top row shows the conventional width optimization approach, which takes a training dataset and a seed network, and outputs a network with optimized widths. The bottom row depicts our idea of width transfer, where width optimization operates on the down-scaled dataset and seed network. We then use a simple function to extrapolate the optimized architecture to match the original network. Compared to direct width optimization, our empirical findings suggest that width transfer has similar performance, but has the benefit of drastically lower overhead.

FLOPs. Such algorithms can be seen as neural architecture search algorithms that search for layer-wise channel counts that maximizes validation accuracy subject to test-time FLOPs constraints.

However, these methods often add a large computational overhead necessary for the width optimization procedure. Concretely, even for efficient methods that use differentiable parameterization [60], width optimization takes an additional $2\times$ the training time. To contextualize this overhead, using distributed training on 8 V100 GPUs, it takes approximately 100 GPU hours to train a ResNet50 on the ImageNet dataset [141]. Including the width optimization overhead, it therefore takes 300 GPU hours for both width optimization using differentiable methods [60] and training the optimized ResNet50. Reducing the overhead for width optimization, therefore, would have material practical benefits.

Fundamentally, one of the key reasons why width optimization is so costly is due to its limited understanding by the community. Without assuming or understanding the structure of the problem, the best hope is to conduct black-box optimization whenever training configurations, datasets, or architectures are changed. In this work, we take the first step to systematically and empirically understand the structure underlying the width optimization problem by changing network architectures and the properties of training datasets, and observing how they affect width optimization.

If similar inputs to the width optimization algorithms result in similar outputs, one can exploit this commonality to reduce the width optimization overhead, especially if the two input configurations have markedly different FLOPs requirements. As a concrete example, if optimizing the widths of a wide CNN

(high FLOPS) and a narrow CNN (low FLOPs) results in widths that differ only by a multiplier, one can reduce the computational overhead of width optimization by computing widths for the low FLOPS, narrow CNN and adjusting them to accommodate the high FLOPs, wide CNN. In particular, we assume the following invariances for the optimized widths and validate them empirically:

1. **Sample size**: The optimized widths are minimally affected by the size of the dataset when the dataset's distribution is approximately identical (*i.e.*, uniform down-sampling in a class-balanced fashion).

2. **Spatial resolution**: The optimized widths are merely affected by the image resolutions.

3. **Channel magnitude**: The ratios between the optimized widths and the un-optimized ones are roughly constant regardless of the absolute channel counts of the un-optimized network.

4. **Within-stage channel counts**: The optimized widths are similar when they belong to the same stage of a network where stage is defined by grouping the blocks with the same input resolution [68].

We further propose *Width Transfer*, a novel paradigm for efficient width optimization that harnesses the above assumed invariances. In width transfer, one first projects the network and the dataset to their smaller counterparts, then one executes width optimization with the smaller network and dataset, and finally one extrapolates the optimized result back as shown schematically in Figure 7.1.

Based on a comprehensive empirical analysis, we provide the following contributions:

- We systematically study the four assumed invariances with comprehensive experiments. Our findings suggest that the four invariances largely hold for the optimized widths.

- We propose Width Transfer, a novel paradigm for efficient width optimization. Additionally, we propose two novel layer-stacking methods to transfer width across networks with different layer counts.

- We demonstrate a practical implication of the previous finding by showing that one can achieve 320× reduction in width optimization overhead for a scaled-up MobileNetV2 and ResNet18 on ImageNet with similar accuracy improvements, effectively making the cost of width optimization negligible relative to initial model training.

- With controlled hyperparameters over multiple random seeds on a large-scale image classification dataset, we verify the effectiveness of width optimization methods proposed in prior art. This is in contrast with prior work which borrows numbers from other papers that might not have the same

training hyperparameters. However, we also find that, for a deeper ResNet on ImageNet, width optimization has limited benefits (Fig. 7.4c).

7.2 Width Optimization

The layer-by-layer widths of a deep CNN are often regarded as a hyperparameter optimized to improve classification accuracy [71, 60, 160]. A width optimization algorithm, $\mathcal{A}$, takes in a training configuration, $\mathcal{C} = (\mathcal{D}, \mathcal{N})$, and outputs a set of optimized widths, $\boldsymbol{w}^*$. $\mathcal{C}$ consists of initial network, $\mathcal{N}$, and training dataset, $\mathcal{D}$. Concretely, the goal of $\mathcal{A}$ is to solve the following optimization problem:

$$\begin{aligned} \boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \; & Acc_{\text{val}}\left(\theta(\mathcal{N} \odot \boldsymbol{w}, \mathcal{D}), \mathcal{N} \odot \boldsymbol{w}\right) \\ \text{s.t. } & \text{FLOPs}(\mathcal{N}) = \text{FLOPs}(\mathcal{N} \odot \boldsymbol{w}), \end{aligned} \tag{7.1}$$

where $\boldsymbol{w}$ is a width vector with L dimensions, where L is the number of convolutional layers. $\mathcal{N} \odot \boldsymbol{w}$ is applying width $\boldsymbol{w}$ to a network $\mathcal{N}$ by scaling the number of channels for layer i from F_i to $w_i F_i$. $\theta()$ is the standard training procedure that takes in a CNN and a training dataset and outputs trained weights, *e.g.*, stochastic gradient descent with a fixed training epoch. Lastly, Acc_{val} is a function that evaluates the validation accuracy given the trained weights and the CNN architecture.

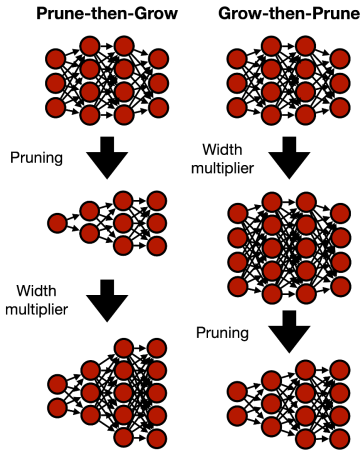


Figure 7.2: The two width optimization strategies proposed in prior art.

To optimize the width of a CNN, there are in general two approaches proposed in prior art: *Prune-then-Grow* and *Grow-then-Prune*. *Prune-then-Grow* uses channel pruning methods to arrive at a down-sized CNN with non-trivial layer-wise channel counts followed by re-growing the CNN to its original FLOPs using the width multiplier method [59]. On the other hand, *Grow-then-Prune* first uses the width multiplier method to enlarge the CNN followed by channel pruning methods to trim down channels to match its

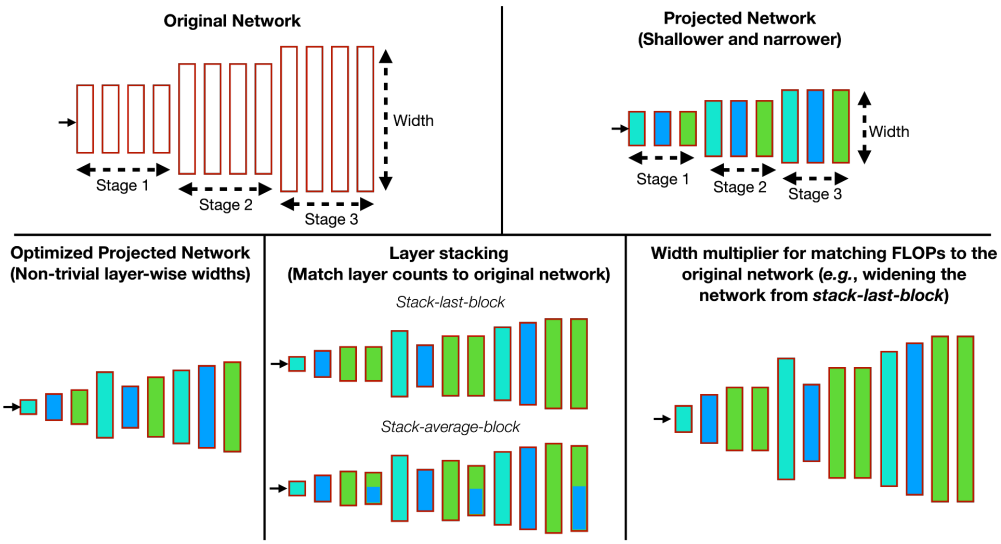


Figure 7.3: An example for extrapolation. The projected network has fewer layers and channel counts per layer compared to the original network. After width optimization on the projected network, we propose two methods, *i.e.*, stack-last-block and stack-average-block, to match the layer counts to the original network. Finally, we match the FLOPs to the original network with a width multiplier.

pre-grown FLOPs [193, 60]. The aim of both of these methods is to improve accuracy of the network while maintaining a given FLOPs count. The schematic view of the two approaches is visualized in Figure 7.2.

While there are many papers on channel pruning [99, 129], they mostly focus their analysis on decreasing the inference-time FLOPs of pre-trained models whereas we focus on improving the classification accuracy of a network by optimizing its width while holding inference-time FLOPs constant. While one can use either the *Prune-then-Grow* or *Grow-then-Prune* strategies to arrive at a CNN of equivalent FLOPs, it is not clear if such strategies generally improve performance over the un-optimized baseline as it is not verified in most channel pruning papers. As a result, in this chapter, we focus on analyzing algorithms that have demonstrated the effectiveness over the baseline (uniform) width configurations in either *Prune-then-Grow* or *Grow-then-Prune* settings.

7.3 Approach

7.3.1 Width optimization methods

Theoretically, we only care about algorithms $\mathcal{A}$ that "*solve*" the width optimization problem (equation 7.1). However, the problem is inherently combinatorially hard. As a result, we use state-of-the-art width optimization algorithms as probes to understand the width optimization problem. More specifically, we consider methods that have reported improved accuracy compared to the un-optimized network given the same test-time FLOPs, and have publicly available code to ensure correctness of implementation.

With these criteria, we consider MorphNet [59], AutoSlim [193], and DMCP [60]. Note that we use these algorithms to find the “network architecture" which will be trained from scratch using the normal training configurations with randomly initialized weights.

7.3.2 Projection and extrapolation

Projection In projection, there are two aspects: network projection and dataset projection. For network projection, we propose to use the width multiplier to uniformly shrink the channel counts for all the layers to arrive at a narrower model. Additionally, we also propose to use the depth multiplier to uniformly shrink the block counts per each stage of a neural network to arrive at a shallower model. For dataset projection, on the one hand, we propose to sub-sample the training sample in a dataset in a class-balanced way. On the other hand, we propose to sub-sample the spatial dimension of the training images to arrive at images with lower resolutions. When keeping the width optimization algorithm fixed, *i.e.*, training the input network with a certain batch size and training epochs using the input dataset, all the aforementioned projections immediately result in width optimization overhead reduction.

Extrapolation We consider two aspects for extrapolation: dimension-matching and FLOPs-matching, which are schematically shown in Figure 7.3. First, we want the extrapolated network to have the same number of layers as the original network. This is particularly crucial when the original network is projected in the depth dimension, in which case we propose two layer stacking strategies:

- **Stack-last-block**: Stack the width multipliers of the last block of each stage until the desired depth is met. A stage includes convolutional blocks with the same output resolution in the original network. A convolutional block consists of several convolutional layers such as the bottleneck block in ResNet [68] and the inverted residual block in MobileNetV2 [154].
- **Stack-average-block**: To avoid mismatches among residual connection, we exclude the first block of each stage and compute the average of the width multipliers across all the rest blocks in a stage, then stack the average width multipliers until the desired depth is met.

Note that since existing network designs share the same channel widths for all the blocks in each stage, the above two layer stacking strategies will have the same results when applied to networks with un-optimized widths.

Second, we want to extrapolate the optimized projected network to a larger one such that it has the FLOPs of the original network. To do so, we propose to use the width multiplier to widen the optimized width. This procedure basically assumes that what determines the optimized widths is the ratio among

layers and we show that this assumption is largely correct in Section 7.4 as the optimized widths are largely transferable across network's initial widths.

7.4 Experiments

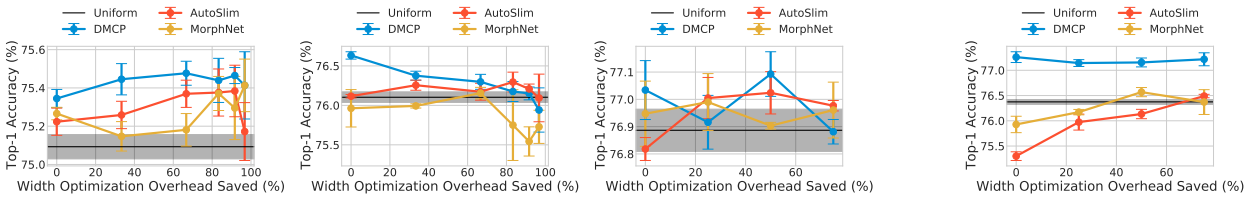
In this section, we empirically investigate the transferability of the optimized widths across different projection and extrapolation strategies. Specifically, we study projection across architectures by evaluating different widths and depths as well as across dataset properties by dataset sub-sampling and resolution sub-sampling. In addition to analyzing each of these four settings independently, we also investigate a compound projection that involves all four jointly. To measure the transferability, we plot the ImageNet top-1 accuracy of network obtained by direct optimization and width transfer across different projection scales that have different width optimization overhead. Width optimization overhead refers to the FLOPs needed to carry out width optimization. If transferable, we should observe a horizontal line across different width optimization overheads, suggesting that performance is not compromised by deriving the optimized widths from a smaller FLOP configuration. Moreover, we also plot the ImageNet top-1 accuracy for the un-optimized baseline to characterize whether width optimization or width transfer is even useful for some configurations.

7.4.1 Experimental Setup

We used the ImageNet dataset [40] throughout the experiments. Unless stated otherwise, we use 224 input resolution. For CNNs, we considered the meta-architecture of ResNet18 [68] and MobileNetV2 [154]. Note that we adjusted the depth and width of ResNet-18 and MobileNetV2 to arrive at a wide variety of models for our width transfer study. Models were each trained on a single machine with 8 V100 GPUs for all the experiments. The width multiplier method applies to all the layers in the CNNs while the depth-multiplier excludes the first and the last stage of MobileNetV2 as there is only one block for each of them. After we obtained the optimized architecture, we trained the corresponding network from scratch with random initialization using the same hyperparameters to analyze their performance. We repeated each experiment three times with different random seeds and reported the mean and standard deviation.

7.4.2 Projection: width

Here, we focus on answering the following question: "*Do optimized widths obey the channel magnitude invariance?*" The answer to this question is unclear from existing literature as the current practice is to re-run the optimization across different networks [60, 59, 115]. If the optimized widths are similar across



(a) Res18, width (b) MBv2, width (c) Res18, depth (d) MBv2, depth

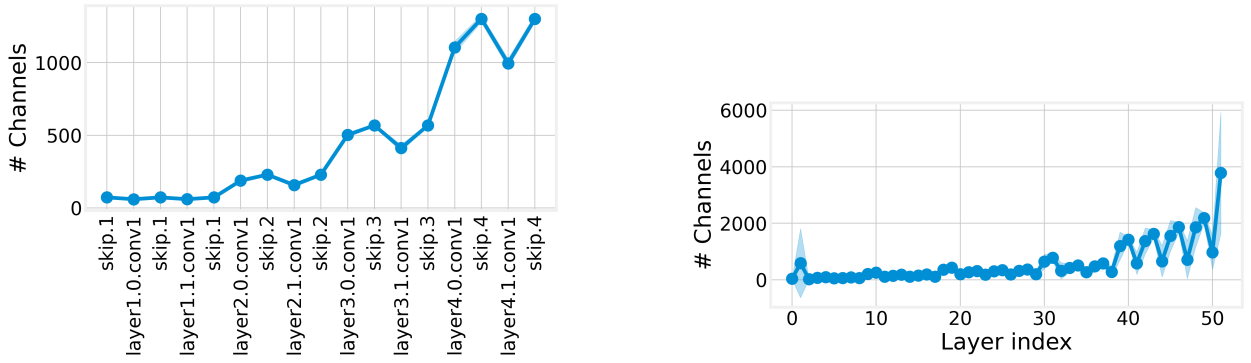
Figure 7.4: Experiments for width transfer under network projection. We plot the ImageNet top-1 accuracy for uniform baseline, width transfer, and direct optimization (the leftmost points). On the x-axis, we plot the width optimization overhead saved by using width transfer.

different initial widths, this suggests that the quality of the vector of channel counts are scale-invariant given the current practice of training deep CNNs and the dataset. Additionally, it also has practical benefits where one can use width transfer to reduce the overhead incurred in width optimization. On the other hand, if the optimized widths are dissimilar, this suggests that not only the direction of the vector of channel counts is important, but also its magnitude. That is, for different magnitudes, we may need different orientations. In other words, this suggests that existing practice, though costly, is empirically proved to be necessary.

To empirically study the aforementioned question, we considered the source width multipliers of {0.312, 0.5, 0.707, 1, 1.414, 1.732} and the target width multiplier of 1.732, and we analyzed if the source optimized architecture transfers to the target architecture. The set of initial width multipliers is chosen based on square roots of width optimization overhead. We analyzed the transferability in the accuracy space. In Figure 7.4a and 7.4b, we plot the ImageNet top-1 accuracy for the baseline (a 1.732× wide network) and networks obtained by direct optimization and width transfer. For ResNet18, the width optimization overhead can be saved by up to 96% for all three algorithms without compromising the accuracy improvements gained by the width optimization.

On MobileNetV2, AutoSlim and MorphNet can transfer well and save up to 80% width optimization overhead. While DMCP for MobileNetV2 results in 0.4% top-1 accuracy loss when using width transfer, the transferred width can still outperform the uniform baseline, which is encouraging for applications that allow such accuracy degradation in exchange for 83% width optimization overhead savings. More specifically, that would reduce compute time from 160 GPU-hours all the way to 30 GPU-hours for MobileNetV2 measured using a batch size of 1024, a major saving. Our results suggest that a good orientation for the optimized channel vector continues to be suitable across a wide range of magnitudes.

Since the optimized widths are highly transferable, we are interested in the resulting widths for both CNNs. We find that the later layers tend to increase a lot compared to the un-optimized ones. Concretely, in un-optimized networks, ResNet18 has 512 channels in the last layer and MobileNetV2 has 1280 channels



(a) ResNet18 (b) MobileNetV2

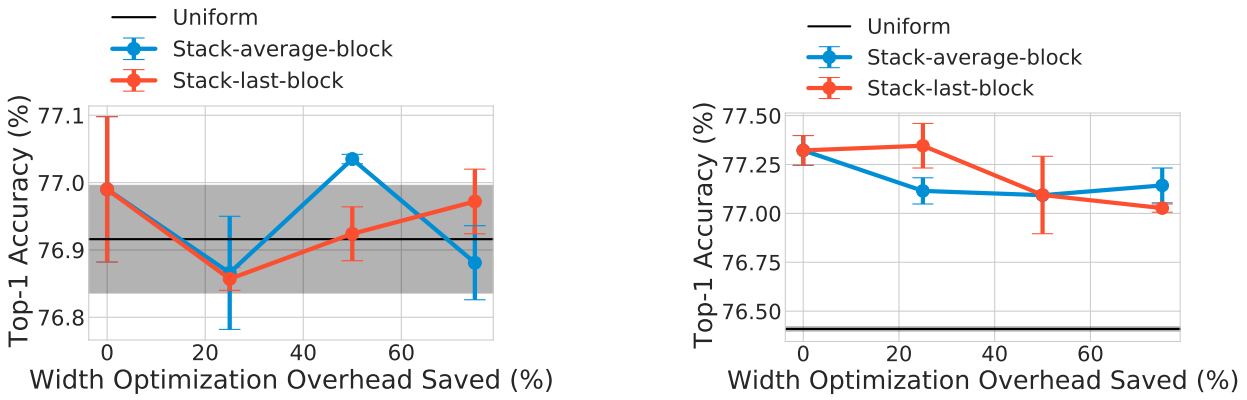
Figure 7.5: The average optimized width for ResNet18 and MobileNetV2. They are averaged across the optimized widths. We plot the mean in solid line with shaded area representing standard deviation.

in the last layer. In contrast, the average optimized width has 1300 channels for ResNet18 and 3785 channels for MobileNetV2. We visualize the average widths for ResNet18 and MobileNetV2 (average across optimized widths) in Figure 7.5.

7.4.3 Projection: depth

Next, we asked whether networks with different initial depths share common structure in their optimized widths. Specifically, "*Do the optimized widths obey the within-stage channel counts invariance?*" Because making a network deeper will add new layers with no corresponding optimized width, we will need a mechanism to map the vector optimized widths to a vector with far more elements. Here, we first compared the two layer-stacking methods proposed in Section 7.3.2 using DMCP for ResNet18 and MobileNetV2. As shown in Figure 7.6, both *stack-last-block* and *stack-average-block* layer stacking strategies perform similarly. As a result, we use *stack-average-block* for all other experiments. We considered {1, 2, 3, 4} as the source depth multipliers and use 4 as the target depth multiplier. Similar to the analysis done in Section 7.4.2, we analyzed the similarity in the accuracy space.

As shown in Figure 7.4c and 7.4d, we find that the optimized widths stay competitive via simple layer stacking methods and up to 75% width optimization overhead can be saved if we were to optimize the width using width transfer for all three algorithms and two networks. This finding also suggests that the relative values of optimized widths share common structure across networks that differ in depth. In other words, the pattern of width multipliers across depth is scale-invariant. Interestingly, we find that width transfer *improves* direct optimization in terms of accuracy when it comes to AutoSlim and MorphNet as we see a positive slope for these two methods on both networks. We conjecture that this is due to



(a) ResNet18

(b) MobileNetV2

Figure 7.6: We compare the two layer-stacking strategies using DMCP for both ResNet18 and MobileNetV2. We can observe that both stack-average-block and stack-last-block perform similarly.

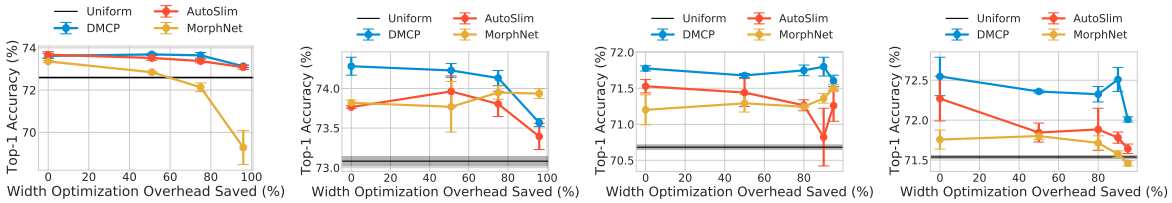
both AutoSlim and MorphNet being affected more by the dimensionality[1] of the problem (the number of widths to be learned), and that the within-stage channel invariance largely holds.

7.4.4 Projection: resolution

The input resolution and the channel counts of a CNN are known to be related when it comes to the test accuracy of a CNN. As an example, it is known empirically that a wider CNN can benefit from inputs with a higher resolution than a narrower net can [164]. As a result, it is not clear *if the optimized widths obey the spatial resolution invariance*. If the optimized widths indeed obey the spatial resolution invariance, this suggests that although wider networks benefit more from a higher resolution inputs, the non-uniform widths that result in better performance are similar. On the other hand, if the optimized widths are different, it suggests that, when it comes to the test accuracy, the relationship between channel counts and input resolution is more involved than the level of over-parameterization.

To study the aforementioned question, we considered the input resolution for the source to be {64, 160, 224, 320} and choose a target of 320. As shown in Figure 7.7a and 7.7b, we find that except for MorphNet targeting ResNet18, all other algorithm and network combinations can achieve up to 96% width optimization overhead savings with the optimized widths that are still better than the uniform baseline. By saving 75% width optimization overhead, we can stay close to the performance obtained via direct optimization. Interestingly, we find that MorphNet had a very different optimized widths when transferred from resolution 64 for ResNet18, which leads to the worse performance for ResNet18 compared to direct optimization.

[1]Width depth projection, we effectively reduce the dimensionality of the search problem.



(a) Res18, resolution (b) MBv2, resolution (c) Res18, dataset size (d) MBv2, dataset size

Figure 7.7: Experiments for width transfer under dataset projection. We plot the ImageNet top-1 accuracy for uniform baseline, width transfer, and direct optimization (the leftmost points). On the x-axis, we plot the width optimization overhead saved by using width transfer.

Network	Baseline	DMCP	Width transfer	Overhead (direct→width transfer)
ResNet50	77.97 ± 0.09	78.23 ± 0.11	78.07 ± 0.12	$37.4 \rightarrow 1.3$
ResNet101	79.43 ± 0.07	79.70 ± 0.05	79.54 ± 0.07	$66.7 \rightarrow 2.7$
EfficientNetB3	80.02 ± 0.09	80.24 ± 0.02	80.19 ± 0.10	$80 \rightarrow 3$

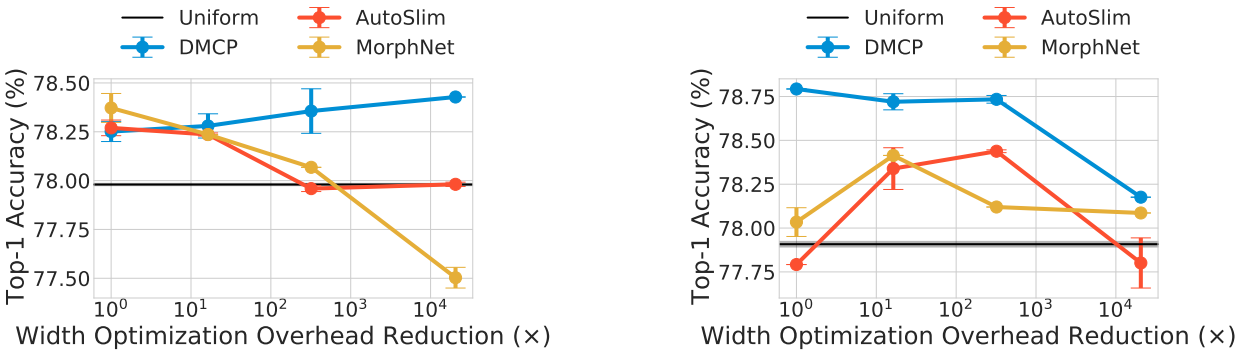
Table 7.1: Compound width transfer for other CNNs. Width optimization overhead measured with 8 NVIDIA V100 GPUS on a single machine.

7.4.5 Projection: dataset size

The dataset size is often critical for understanding the generalization performance of a learning algorithm. Here, we would like to understand *if the optimized widths obey the sample size invariance*. We considered subsampling the ImageNet dataset to result in a source of $\{5\%, 10\%, 20\%, 50\%, 100\%\}$ of the original training data. Similar to previous analysis, we tried to transfer the optimized widths obtained using the smaller configurations to the largest configuration, *i.e.*, 100% of the original training data. As shown in Figure 7.7c and 7.7d, widths optimized for smaller dataset sizes transfer well to large dataset sizes. That is, 95% width optimization overhead can be saved and still outperforms the uniform baseline for both networks. On the other hand, 90% width optimization overhead can be saved and can still match the performance of direct optimization for DMCP. This suggests that the amount of training data barely affects width optimization, especially for DMCP, which is surprising.

7.4.6 Compound projection

From previous analyses, we find that the optimized widths are largely transferable across various projection methods independently. Here, we further empirically analyzed if the optimized width can be transferable across compound projection. To do so, we considered linearly interpolating all four projection methods and analyzed if the width optimized using cost-efficient settings can transfer to the most costly setting. Specifically, let a tuple (width, depth, resolution, dataset size) denote a training configuration. We considered the source to be {(0.312,1,64,5%), (0.707,1,160,10%), (1,1,224,50%), (1.414,2,320,100%)} and the target to be (1.414,2,320,100%). As shown in Figure 7.8, the optimized width is transferable across



(a) ResNet18

(b) MobileNetV2

Figure 7.8: Width transfer with compound projection.

compound projection. Specifically, we can achieve up to 320× width optimization overhead reduction with width transfer for the best performing algorithm, DMCP. Additionally, it also suggests that the four projection dimensions are not tightly coupled for width optimization.

Applications to other target CNNs We considered using compound width transfer for ResNet50, ResNet101, and EfficientNetB3. For projection, we consider the width, depth, resolution and dataset size to be 0.707, 0.5, 160, and 20%, respectively. As shown in Table 7.1, up to 30× wall-clock time reduction is achieved with less than 0.2% top-1 accuracy degradation. Consider a scenario where one wants to optimize the width of a network and train such a network for deployment. Width optimization reduces the overall training cost from 3× to 1.06×. Such a huge optimization cost reduction can enable fast exploration for the benefits of width optimization for large models without paying considerable costs.

7.4.7 Comparing to cheap pruning methods

While adopting state-of-the-art channel optimization directly can be costly, one may consider using cheap pruning methods and adopt a Prune-then-Grow strategy to carry out width optimization. We compare to magnitude-based channel pruning: network slimming (NS) [114] that prunes channels based on the magnitude of γ of the batch normalization layer. NS-xw-ye follows a three-step procedure: train an $x\times$ wider network for y epochs, prune the network with global γ ranking, and re-train the pruned network using full training schedule. The induced overhead for width optimization lies in the first step. The comparisons with NS for ResNet18 on ImageNet is shown in Fig. 7.9. Using DMCP directly is indeed the most expensive one, but it has the best performance. Our width transfer achieves similar performance compared to DMCP with overhead lower than magnitude-based pruning.

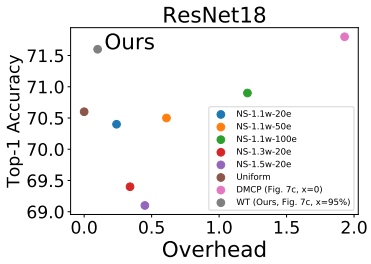


Figure 7.9: Comparing the proposed using DMCP with width transfer, DMCP, and network slimming.

Channel pruning Channel pruning is an active research topic for efficient network design. More specifically, channel pruning determines how we can prune an existing network in the channel dimension so as to retain the most accuracy [25, 70, 71, 99, 129, 102]. A channel pruning procedure often has the weights and the optimized channel counts coupled together. Inspired by Liu *et al.* [116], who has empirically shown the importance of the optimized channel counts, we take a step further by understanding the transferability of the searched channel counts across different input network and dataset transformations.

7.5 Discussion

In this chapter, we take a first step in understanding the transferability of the optimized widths across different width optimization algorithms and invariance dimensions. Our empirical analysis sheds light on the structure of the width optimization problem, which can be used to design better optimization methods. More specifically, by exploiting the channel magnitude and within-stage channel counts invariances, we not only can reduce the computational cost needed to width optimization, but also reduce the dimension of the optimization variables[2]. Per our analysis, we can achieve up to 320× reduction in width optimization overhead without compromising the top-1 accuracy on ImageNet.

7.6 Carbon Footprint Analysis

When compared to methods that incorporate width optimization in the training time, our methodology saves carbon footprint during the training time. Applying Width Transfer to DMCP is practically about 3× faster than DMCP, which means Width Transfer saves 3× carbon footprint than alternatives when it comes to training.

[2]In the channel search space, we have one optimization variable for each layer.

Chapter 8

Synergies and Discussion of Presented Approaches

8.1 LeGR, Joslim, and Width Transfer

While all three aforementioned algorithms contribute to advancing the efficiency of filter counts optimization, they have different competitive edges depending on different application scenarios. In this section, we quantitatively compare and contrast the three proposed algorithms in various application scenarios to shed some light on their relationship.

We consider the following two application scenarios:

- Trying to optimize the filter counts of a CNN to obtain a family of efficient networks. More specifically, the use case focuses on outputting **many** optimized networks with the objective of achieving Pareto-optimality.
- Trying to obtain **a weight-sharing network** of which the filter counts of the sub-networks are optimized to achieve Pareto-optimality. The weight-sharing network can be used in an online adaptive setting.

Note that Width Transfer is complementary to LeGR and Joslim as Width Transfer can operate on any width optimization algorithm. As a result, we perform Width Transfer for both LeGR and Joslim in this section. To compare the three algorithms in two application scenarios, we use the CIFAR-100 dataset [93] and compare the overhead of the algorithms and the resulting accuracy of the trained networks. For CNNs, we adopt ResNet-56 [68] for this analysis.

	Searching Cost	Training Cost
Uniform	None	Training the uniform architectures
LeGR	• Training the unpruned model • Learning global ranking	Fine-tuning the pruned models
WT-LeGR	• Training the unpruned model with proxies • Learning global ranking with proxies	Training the optimized architectures
Joslim	• Joint widths and weights optimization	None
WT-Joslim	• Joint widths and weights optimization with proxies	Training the optimized architectures

Table 8.1: Comparing the overhead of different channel searching methods. WT stands for Width Transfer.

8.1.1 Optimizing for Many Target Networks

In this task, we focus on searching for many optimized networks pruned from ResNet-56. Specifically, we consider models of FLOP counts ratios of the original model that range from 20% to 80% in a step of 10%. For this task, the overhead of the algorithms includes finding the set of pruned networks, which we call *searching*, and fine-tuning or training the found networks, which we call *training*. One straightforward baseline for this task is to use the uniform width multiplier to arrive at networks of different FLOPs and train them using standard training procedures, which we call *Uniform*.

For Uniform, searching cost is effectively zero and the training cost is the overhead to train the seven models from scratch until convergence. For LeGR, the searching cost includes training the unpruned model and learning the global ranking. While LeGR has higher searching cost than Uniform, LeGR has much lower training cost as LeGR fine-tunes the pruned model instead of training each of the pruned model from scratch. For Joslim, the searching cost includes the joint optimization of both widths and weights while the training cost is effectively zero. For Width Transfer, the searching cost is greatly reduced compared to LeGR and Joslim due to the usage of proxy networks and datasets. However, the training cost is increased as Width Transfer only transfer the widths (or the architectures) but not the weights, which makes its training cost similar to that of Uniform. The overhead comparisons are detailed in Table 8.1.

Figure 8.1a and 8.1b show the results for different methods and their overhead analysis. We can observe that Joslim achieves the best prediction error *vs.* FLOPs trade-off compared to other methods while being the most expensive method when the number of target compression ratios is below ten. The second competitive approach is LeGR, which has a much lower overhead compared to that of Joslim when the number of target compression ratios. Lastly, Width Transfer scales similarly compared to Uniform in terms of overhead while being able to provide better architectures than Uniform. Width Transfer is more attractive when the number of target compression ratios is low and when the model under optimized is large. The better performance of Joslim and LeGR compared to their Width Transfer counterparts indicates that not only the architectures matter for pruning, but also their weights, which aligns with the

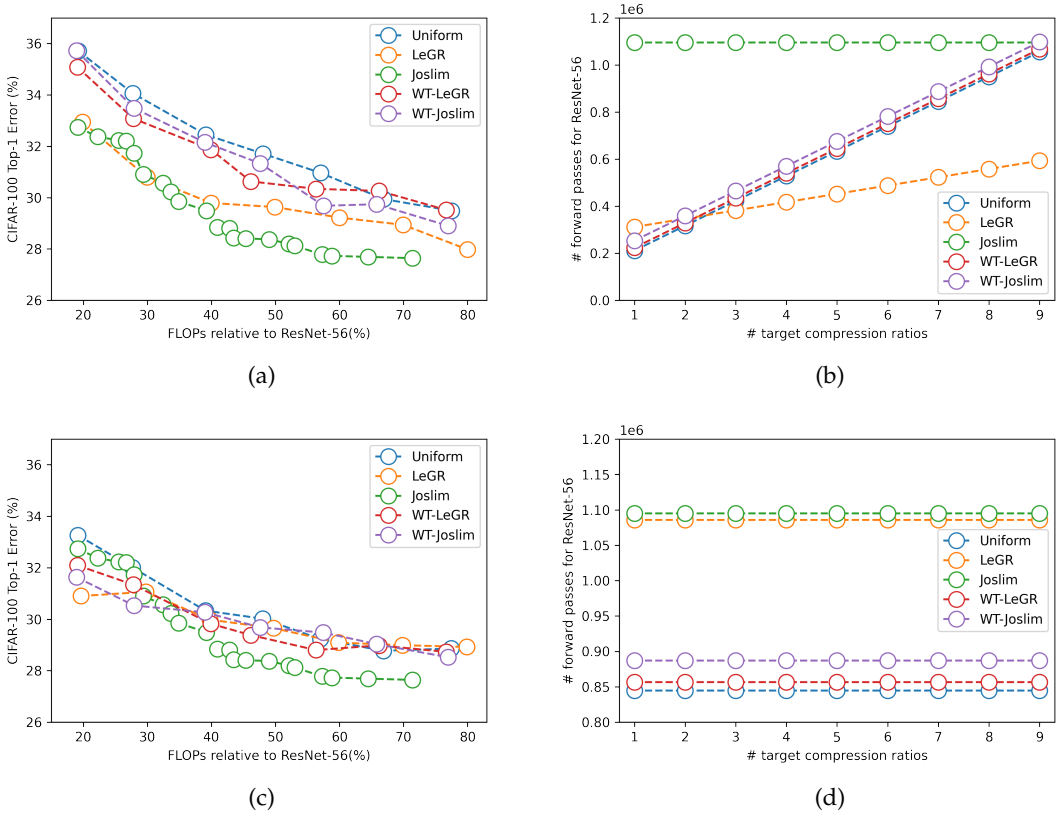
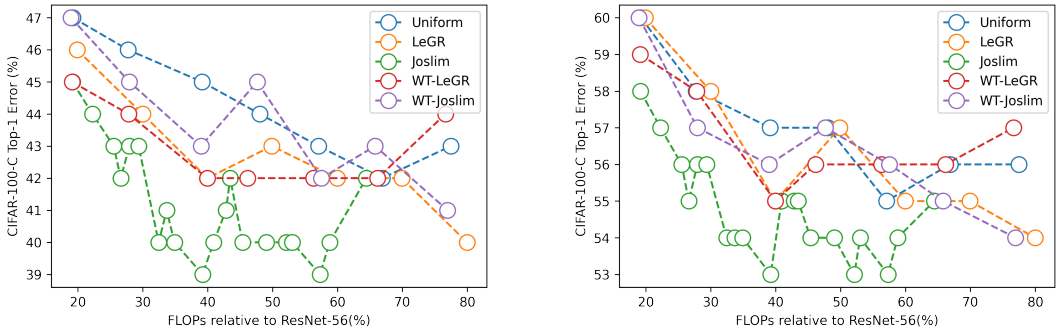


Figure 8.1: (a,c) The trade-off curve of pruning ResNet-56 on CIFAR-100 using various methods. (b,d) Training cost and its scaling with respect to the number of target compression ratios for different methods targeting ResNet-56 on CIFAR-100. The cost is calculated using the number of forward passes for a ResNet-56 while approximating one backward pass as two forward passes. Fig. a and b are for optimizing many target networks while Fig. c and d are for optimizing weight-sharing networks.

recent findings by Ye *et al.* [191].

8.1.2 Optimizing for a Weight-sharing Network

This task is similar to previous task with the difference being we would like to obtain a weight-sharing model. That is, we only use a single set of weights for all the target compression ratios. For methods other than Joslim, we simply follow the procedures in previous tasks while training the obtained architectures with weight-sharing training [194]. Figure 8.1c and 8.1d show our results. From the results, we can observe that Joslim generally performs the best, which is expected as Joslim is designed for this particular task by jointly optimizing both the weights and the width configurations. Interestingly, Joslim performs worse than LeGR and Width Transfer for the smallest target compression ratio, which is because the smallest target compression ratio in Joslim has an uniform architecture while it has a non-uniform architecture for LeGR and Width Transfer. This implies that one can further improve Joslim by having the widths of the



(a) Severity: 1 (b) Severity: 3

Figure 8.2: The mean corruption errors for different number of target compression ratios of different methods targeting ResNet-56 on CIFAR-100. Note that the networks are optimized with CIFAR-100 and test using CIFAR-100-C.

smallest model being searched as well. Lastly, we find that the gap between LeGR and Uniform close as we move from no weight-sharing to weight-sharing, which suggest that LeGR benefits a lot from its pre-trained weights and the architectures found by LeGR are not drastically better than Uniform. In terms of overhead, Joslim has the largest overhead while all methods require constant overhead across various target compression ratios thanks to the weight-sharing training.

8.1.3 Extending to Transfer Learning

Extending the proposed pruning methods to transfer learning is straightforward if the source dataset is available. Specifically, we conduct both LeGR, Joslim, and Width Transfer on the source dataset and fine-tune the pruned models on the target dataset. However, it is less clear if the source dataset is not available. If we do not have access to the source dataset and only have access to a pre-trained model that is trained on the source dataset, we need to adapt the methodology a little. The key problem that might arise in this scenario is that pruning might lead to forgetting important features that are only learnable through the large-scale source dataset. To combat this issue, we can add feature distillation loss to the pre-trained model [29, 100] for the training and fine-tuning procedures during the pruning methodology.

Overall, we find that Joslim performs the best in terms of the performance if the overhead can be accepted. On the other hand, Width Transfer gives the lowest overhead with minor improvements over Uniform if only targeting a few compression ratios. As an alternative, LeGR provides a nice balance between Joslim and Width Transfer when considering a moderate number of target compression ratios. All methods can be extended to target transfer learning easily.

8.2 Robustness to Distributional Shifts

In addition to the standard accuracy metric, we further analyze the performance of the pruned models on CIFAR-100-C [75] to understand how filter counts affect networks' robustness in distribution shifts. Specifically, CIFAR-100-C perturbs the images in CIFAR-100 with various natural perturbations such as Gaussian Blur, Motion Blur, Frost, and Zoom Blur just to name a few. For each of the perturbation, the dataset provides severity from level 1 to 5. In this subsection, we report the mean corruption error for level 1 and level 3 for all kinds of perturbations provided in the dataset. As shown in Figure 8.2, we find that the ranking among methods stay roughly the same. Interestingly, when it comes to Joslim, we find that the best model in mean corruption error is not necessarily the best model in clean error, which means that it might be desirable to do pruning for improved robustness. While standard pruning has shown in prior work to be harmful for the out-of-distribution setting [104], a weight-sharing training approach (*e.g.*, Joslim [27], Slimmable Nets [196], and BigNAS [195]) might behave differently.

8.3 Relation to Neural Architecture Search

Filter counts search is closely related to neural architecture search (NAS). In filter pruning, one typically starts with a pre-trained model and tries to identify redundant filters to prune away, which is followed by a fine-tuning procedure to improve the accuracy of the pruned model. The resulting model has both the neural architecture and the weights that are different from the pre-trained ones. Hence, filter pruning algorithms can be recognized as a variant of NAS where the search space is the number of filters for each layer. In light of this, Liu *et al.* [116] conduct empirical analysis on various filter pruning algorithms to understand the performance of the pruned architectures by re-initializing the weights of the pruned models and train them from scratch. They conclude that the main sources of effectiveness of the studied pruning algorithms comes from the resulting architecture. Later, Ye *et al.* [191] prove both empirically and theoretically that not only the architectures, but also the pre-trained weights are useful. Hence, effective filter pruning algorithms play a role in identifying not only a good neural architecture, but also good weights.

Due to the importance of the neural architecture in filter pruning, various NAS techniques have been adopted in filter pruning to develop new filter pruning methods. For example, both the one-shot supernet method [12, 193, 27, 60, 160] and the Gumbel relaxation method [171] are explored in filter pruning. It is worth noting that the choices for the number of filters is large for each searchable variable compared to the commonly studied neural architecture search space [111, 162, 183, 16] (*i.e.*, hundreds vs. less then ten). As a result, the differentiable categorical formulation adopted in DARTS [111] or SNAS [186] is less suitable

for the filter pruning space. To use those formulation, FBNetV2 [171] discretizes the filter counts into at most 14 choices per layer. Due to the large number of choices per layer, the one-shot neural architecture search method is more popular in filter pruning.

In this thesis, we discuss three approaches to make filter pruning more efficient across multiple target constraint levels. Our proposed methods are also closely related to efficient neural architecture search methods that aim at the same goal. Specifically, Once-for-all (OFA) networks [15] and BigNAS [195] both try to search for resource-constrained architectures efficiently across different constraint levels. Besides the difference in the search space in our methods compared to theirs, our methods are also fundamentally different. In LeGR, we essentially propose to learn to rank different architectures such that the ranking generates a continuum trade-off between accuracy and compute resources. Such an idea has not been explored in the neural architecture search space. In Joslim, our algorithm shares some similarity with OFA and BigNAS. As discussed in Chapter 6, our proposed Joslim generalizes OFA and BigNAS by having a multi-objective formulation, which we approach with alternating minimization. Both OFA and BigNAS can be seen as greedy methods that sequentially train the supernet and search for resource-constrained architectures while Joslim alternate between the two. In Width Transfer, we propose a novel proxy for the filter pruning search space to drastically reduce the search overhead. While ECO-NAS [209], which is the closest related idea in the NAS literature, has also explored various proxy settings for NAS, our proposed Width Transfer adds dimensionality reduction of the optimization variable into the picture.

8.4 Applicability to Networks besides CNNs

In this thesis, we mainly discuss approaches to improve the efficiency of CNNs, which is the dominant neural architectures for visual tasks. Since Transformers have recently achieved great success in various computer vision tasks [51, 19, 10, 177, 206, 92, 169, 148], we discuss the applicability of the methods proposed in this thesis to neural architectures other than CNNs.

In AdaScale, we propose to apply different image resolutions for different images to improve both speed and accuracy. This exploits one characteristic of modern CNNs that they are faster with smaller image resolutions. Fortunately, this characteristic also holds for Visual Transformers [51]. More specifically, instead of partitioning images into 16×16 patches, we can partition them into larger patches, which reduces the total input tokens for transformers, which in turn lead to reduction in end-to-end latency. This idea is recently explored by Wang *et al.* [179], which demonstrates the potential usefulness of AdaScale in Transformers.

By equating the number of filters per layer in a CNN to the number of hidden units per layer in a

Transformer, all three methods proposed in this thesis regarding efficient filter pruning (LeGR, Joslim, and Width Transfer) can be directly applied to Transformers. Similarly, the theoretical analysis provided in Winning-Bitwidth (Chapter 4) still holds for Transformer networks. One key characteristic of CNNs these methods build upon is that having non-trivial filter counts per layer improves the performance of a CNN without increasing the computational overhead. Fortunately, various papers have shown structural pruning to be effective for Transformers [8, 125, 174], which demonstrates the potential usefulness of Winning-Bitwidth, LeGR, Joslim, and Width Transfer for Transformers.

Chapter 9

Related Work

9.1 Model Compression/Acceleration

Various methods have been developed to compress and/or accelerate CNNs including weight quantization [144, 214, 87, 89, 198, 76, 44, 31], efficient convolution operators [77, 73, 184, 78, 205], neural architecture search [212, 37, 16, 47, 162, 159, 158, 62, 183], adjusting image resolution [163, 24, 164], and filter pruning [99, 130, 129, 71, 69, 181, 105, 193, 114, 59, 189].

9.1.1 Filter Pruning

In filter pruning, we are interested in finding out redundant channels to remove such that the CNN can be accelerated without hurting its predictive performance. Some methods attack this problem by trying to remove filters to minimize the loss difference between the pruned model and the pre-trained model. These methods typically use some metric for approximating filters' impact on the loss function. Examples include using the norm of filter weights [99, 124, 71], Taylor expansion [130, 129, 167, 172], learned criteria [25, 69], random sampling to gauge the importance [95, 105], and greedily probing each channel to measure the loss [191]. On the other hand, methods that directly seek for a network with fewer channel counts are also proposed. Specifically, these methods include adding a regularization term to introduce filter-level sparsity during standard training [181, 114, 59, 119, 83, 213], and making the channel counts differentiable with approximation methods [168, 135]. Liu *et al.* [116] found that the architectures of the pruned models (the non-trivial channel counts per layer) may be the main source of effectiveness for filter pruning, which leads to a family of methods that aim to perform channel counts search using one-shot neural architecture search [60, 12, 193, 27].

9.1.2 Quantization

Quantization for neural networks allows one to store the weights of the neural networks in a lower precision format, which leads to model compression. Additionally, with activation being quantized, low-precision computation can achieve energy and latency reduction. There are in general two directions for quantization in prior literature, post-training quantization [132, 127, 203, 156] and quantization-aware training [144, 214, 87, 89, 198, 76, 31]. The former assumes training data is not available when quantization is applied. While being fast and training-data-free, its performance is worse compared to quantization-aware training.

In post-training quantization, Sung *et al.* propose to optimize for the clipping location for quantization that minimizes the L2-norm of the quantization error [161]. Banner *et al.* provide the analytical solutions to optimal clipping point assuming the parameters to be quantized follow a Gaussian or Laplace distribution [7]. Zhao *et al.* propose to expand the network in the channel dimension by splitting existing channels to reduce the outliers in quantization [204]. Nagel *et al.* show that rounding to the nearest neighbor for quantization can be sub-optimal and propose to optimize the rounding decision to achieve better results [131].

In quantization-aware training, Rastegari *et al.* [144] introduce binary neural networks, which lead to significant efficiency gain by replacing multiplications with XNOR operations at the expense of significant accuracy degradation. Later, Zhu *et al.* [214] propose ternary quantization while others [211, 87, 98] bridge the gap between floating-point and binarized neural networks by introducing fixed-point quantization. Building upon prior art, the vast majority of existing work focuses on reducing the accuracy degradation by improving the training strategy [208, 187, 117, 43] and developing better quantization schemes [89, 176, 198].

There are also efforts that try to bridge the gap between the two family by leveraging knowledge distillation [32, 17].

Additionally, deciding which layer to use which bit-widths is also an important research direction. In particular, HAQ [176] uses reinforcement learning to conduct the search, HAWQv2 [50] uses the trace of the Hessian to perform layer-level sensitivity analysis, and differentiable NAS methods are also adopted in various papers [192, 185, 58].

Related to our findings in Chapter 4, Mishra *et al.* [128] have also considered the impact of channel count in quantization. In contrast, our work has the following novel features. First, we find that in ConvNets with standard convolutions, *a lower bitwidth outperforms higher ones under a given model size constraint*. Second, we find that the Pareto optimal bitwidth negatively correlates to the convolutional

kernel fan-in and we provide theoretical insights for it. Last, we show that a single weight bitwidth can outperform *mixed-precision* quantization on ImageNet for ResNet50 and MobileNetV2.

9.1.3 Hardware-aware neural architecture search

To achieve better efficiency, one can consider searching for neural architectures while subject to some hardware constraints. In particular, DPPNet [48], ProxylessNAS [16] and MnasNet [162] are the early methods for hardware-aware neural architecture search. Later methods focus on different search spaces [37, 171, 106, 6, 178], a more efficient search procedure [183, 159, 62, 15, 195], different ways to conduct weight-sharing [173, 96], or uses a constrained formulation [133]. It is worth noting that the search space in these methods usually do not include the channel counts per layer [16, 183, 159, 61, 15, 133], or include very small variation in the channel counts [195, 173]. The only exceptions are ChamNet [37], FBNetv2 [171], MnasNet [162], and MCUNet [106]. However, the channel counts space in these methods is relatively coarse-grained (per-stage variables or a per-network variable) compared to the filter pruning literature. Hence, combining filter pruning methods together with neural architecture search may be worth exploring to unleash the full potential of hardware-aware NAS. APQ [178] is the only recent attempt of combining filter pruning, quantization, and NAS.

9.2 Efficient Model Compression

9.2.1 Search once and reuse for multiple constraint levels

Model compression techniques usually comes with a hyperparameter to determine the trade-offs between the accuracy and the compression level. Examples include having different reward formulation when searching for the optimal compressed models [176, 71, 162], having different weights for combining conflicting objectives [59, 114, 181], having different constraint level in optimization [189, 12], or having different condition for the greedy compression procedure to stop [129, 1, 167, 172]. Besides the greedy methods, other methods require the constraint level to be an input to a costly optimization process, which means searching for compressed models over a large set of constraint levels is going to be a prohibitively expensive procedure. To address the scalability of optimization given many constraint levels of interest, various methods were proposed. For example, we proposed LeGR [25] that builds learning on top of a greedy procedure and Joslim [27] that directly uses a multi-objective formulation to solve for a trade-off front with weight-sharing. In addition to our efforts, slimmable neural networks [196] enable multiple sub-networks with different compression ratios to be generated from a single network with one set of weights. While slimmable networks only vary the channel counts in the networks to arrive at different

networks, once-for-all network [15] and BigNAS [195] extend the search space to kernel size, expansion ratios, and depths of a neural network. Similarly, AdaBit [88] and BatchQuant [4] present weight-sharing networks for quantization such that quantization to different bit-width can be done online efficiently without re-training.

9.2.2 Transferrability of neural architectures

In light of the findings of Liu *et al* [116], which demonstrate that the effectiveness of filter pruning mainly lies in the neural architecture, efficient neural architecture search (NAS) methods hence provide the means for efficient filter pruning. Our findings in Chapter 7 are tightly connected to understanding the transferability of the searched results from NAS algorithms where the search space is determined by the layer-wise channel counts of a seed network. The transferability of NAS has been recently explored in several papers considering different search spaces and perspectives. Zoph *et al*.[220] have proposed to search the best cell on a small dataset and use the searched cell on a large dataset. Panda *et al*.[136] have analyzed the transferability of the solutions of various NAS algorithms in the DARTS search space [111] and have concluded that the design of the proxy datasets for the search has a great impact on the transferability of the searched result. Critically however, prior work has neglected to include the channel width multipliers in the search space, instead only focusing on proxy datasets [136, 220]. Consequentially, the relationship between optimized widths across different architectures has not been examined previously. Others [182, 20, 120] have analyzed the transferability of the search processes as opposed to the searched solutions. The key difference among the two is that the transfer of the search processes is algorithm-dependent while the transfer of the searched solutions is not. EcoNAS [209] is closely related to our Chapter 7 as it also systematically investigated several proxy training configurations for neural architecture search. However, the crucial difference between EcoNAS and ours is that, in our study, the number of channels is not only a projection dimension, but also the optimization variable. As a result, the extrapolation step is necessary for our study but not for EcoNAS. Consequentially, the proposed width transfer has a dimensionality reduction effect for hyperparameter optimization while EcoNAS does not. Lastly, the search space for EcoNAS is the cell-based search space while our channel search space is orthogonal to theirs and can be applied to any network for architectural fine-tuning.

9.3 Object Detection

Chapter 3 focuses on applying adaptive scaling to video object detection in order to improve both speed and accuracy of object detectors. In the sequel we discuss prior work in scale-related object detection and

video object detection.

9.3.1 Image Scale for Object Detection

We discuss two categories of scale-related object detection work: (i) single-shot detection by exploiting feature maps from various layers of the CNN with inherently different scales, and (ii) multi-shot detection with input images at multiple scales.

Single-Shot: In this category, object detectors are designed to take an input image once and detect objects at various scales. That is, this category of prior work treats deep CNNs as scale-invariant. Prior work [9] uses features from different layers in the CNN and merge them with normalization and scaling. A similar idea is also adopted by other work [112, 18, 109, 210]. From a different viewpoint, prior art [113] proposes to use a recurrent network to approximate feature maps produced by images at different scales. Though single-shot approaches have shown great promise in better detecting various scales, the scale-invariant design philosophy generally requires a large model capacity [90, 113]. We note that, without perfect scale-invariance, different image scales will result in different accuracy, and prior art often uses a fixed single scale, *e.g.*, 600 pixels on the smallest side of the image. Hence, this line of work could be further improved in terms of speed and accuracy when augmented to adaptive scaling.

Multi-shot: This refers to scaling a single input image to various scales, forwarding each scaled image through the object detector, and merging the obtained results. Some work [67, 57, 151, 68, 35] forwards multiple scales of images to obtain feature maps with various scales. More recently, prior work [36] leverages multiple scales of images to infer multiple detection results and merge them using Non-Maximum Suppression. While multi-shot object detection alleviates the problem of imperfect scale-invariance, it incurs significant extra computation overhead, *i.e.*, up to 4$\times$ [57].

In contrast to the above work, Chapter 3 is aiming to alleviate the imperfect scale-invariance by selecting the best scale for each image, and hence, improves the accuracy compared to single-shot methods. Moreover, to improve the speed in the meantime, we consider down-sampling rather than up sampling. We note that our method could possibly be extended to multi-shot version, *i.e.*, adaptively select multiple scales for a given image, and we leave it for the future work.

9.3.2 Video Object Detection

We discuss prior work that aims at improving speed and/or accuracy of video object detection.

Speed: Optical flow was proposed to reduce detection overhead previously [217]. Similar to our idea, some prior art [28] proposes to adaptively scale the image to improve the detection speed. However,

both works improve speed at the expense of accuracy loss. *Accuracy:* Prior work [91] proposes to leverage contextual and temporal information across the video while some work [216] uses the idea from Deep Feature Flow (DFF) [217] to incorporate temporal information across consecutive frames. Another study [55] proposes to integrate detection with tracking into an end-to-end trainable deep CNN. *Both*: Some prior work [215] extends [216] and [217] to use both feature aggregation and propagation. Additionally, they propose to regress a quality metric of the optical flow to decide when and how to propagate the features.

Compared to the aforementioned related work, the main contribution that sets our Chapter 3 apart is that we focus on fixing the problem where existing object detectors use fixed single scale for each of the image while the object detectors are not scale-invariant, which is different from most of the prior work that focuses on exploiting the relationship of the detection results among the neighboring frames [91, 217, 216, 55, 215]. Moreover, we show that our work is complementary to state-of-the-art video object detection acceleration technique [217].

Chapter 10

Conclusions

While deep convolutional neural networks (CNNs) have shown their impact in various computer vision tasks, their growing complexity makes it hard to run them directly on embedded or mobile devices that have resource constraints. Moreover, there are a wide variety of devices with different hardware constraints and it is critical to design methods that can scale to many different resource constraint levels. In this thesis, we aim to design methods for improving the deployment of CNNs onto resource-constrained devices from two perspective.

In the first perspective, we propose new ways to conduct model acceleration and compression for CNNs. Specifically, we propose AdaScale and WinningBitwidth. While image resolution is often considered as a design knob to trade-off speed and accuracy of CNNs, in AdaScale, we identify that, for some images, smaller resolutions do not necessarily lead to lower mean average precision for object detection. This provides us the motivation to harness this phenomenon to improve both the speed and accuracy of video object detection by properly learning what image to down-sample and to what extent. By learning to adaptively down-sample the input images, our results demonstrate 1.6$\times$ speedup in wall-clock time can be achieved with 1.2 *points* mAP improvement on the ImageNet VID dataset. AdaScale provides a new means to accelerate CNNs and is a step towards real-time video object detection. On the other hand, quantization is a general technique to compress CNNs. As opposed to the commonly adopted formulation where the architecture of the CNN to be quantized is treated as fixed input, we propose to optimize the network architecture together with weight quantization to achieve better compression rates. Specifically, we show that by allowing the channel counts of a CNN to vary at the time of quantization, one can find better models. Moreover, by taking channel counts into consideration, a single weight-bitwidth throughout the networks can outperform conventional mixed-precision quantization that treats neural architecture as fixed inputs. More interestingly, we identify that a wider CNN with a lower bit-width can

lead to better performance compared to a narrower CNN with a higher bit-width. We further characterize this phenomenon and show that the quantization error is negatively correlated with the fan-in channel counts of the convolutional layer to be quantized. Overall, our findings suggest that one can achieve better model compression when taking the neural architectures into consideration.

In the second perspective, we aim to accelerate the process of model compression to achieve better scalability across different constraint levels impose by various embedded devices. Specifically, we propose LeGR, Joslim, and Width Transfer. In LeGR, we propose to learn a global ranking among filters in a CNN so that one can adopt greedy pruning to quickly prune models to various resource constraint levels. This is in stark contrast to existing methods that search for a compressed network that satisfies some resource constraint after a costly optimization procedure. The learned global ranking in LeGR can be reused for multiple constraint levels while other methods have to carry out the heavy optimization for every target constraint. In this fashion, LeGR can be up to $3\times$ faster than prior work while having comparable or better performance when targeting seven pruned ResNet-56 with different accuracy/FLOPs profiles on the CIFAR-100 dataset. Moreover, the performance of the pruned ResNet-50 and MobileNetV2 achieve accuracy that are comparable to the state-of-the-art on the ImageNet dataset.

However, LeGR does not address all efficiency problems. Specifically, even though one can find pruned CNNs fast, each of them needs to be fine-tuned to re-gain its accuracy, which can still be costly. Hence, we further remove the fine-tuning cost by introducing Joslim. In Joslim, our goal is to obtain a CNN that can be pruned to various resource constraint levels without needing any re-training. This can greatly broaden the application scenarios of pruning. One such example is adapting a CNN online based on the latency profiles of the CNN at run-time to meet some latency deadline. Since the latency of a CNN is a variable that depends on many things including the chip temperature and the background processes, adapting the CNN at run-time provides a means for meeting the deadline. To achieve the goal of pruning without fine-tuning, we introduce a multi-objective formulation to jointly optimize the channel counts as well as the shared-weights using alternating minimization. While the accuracy of the networks produced by Joslim are slightly lower than LeGR for MobileNetV2 on ImageNet, no re-training is needed for Joslim, which makes it more scalable when targeting many constraint levels. Moreover, Joslim achieves up to 1.7% top-1 accuracy improvement on the ImageNet dataset for MobileNetV2 compared to existing alternatives that aim to achieve the same goal.

Lastly, inspired by prior work [116] that shows the effectiveness of model compression mainly lies in the compressed neural architecture, we aim to find the compressed architecture efficiently and propose Width Transfer. In Width Transfer, we can obtain the compressed network architecture without training the largest model. To do so, we assume the optimized (compressed) channel counts are regular across

depths and widths of the network architectures and are invariant to the resolution and the size of the dataset. We empirically demonstrate that the assumed regularity and invariances largely hold and can lead to 320× overhead reduction in the cost of searching for the optimized channel counts without losing top-1 accuracy on the ImageNet dataset for large MobileNetV2 and large ResNet-18 when comparing Width Transfer to existing optimization methods.

While all of above three contributions are related to speeding up the procedure of model compression that uses channel optimization, they are characterized by a different competitive edge as we discussed in Chapter 8. Overall, we find that Joslim performs the best but with higher optimization overhead compared to LeGR. While Width Transfer is generally applicable to any width optimization algorithms, it only transfers the architectures but not the weights, which makes it less desirable if targeting many target compression ratios. That said, Width Transfer is efficient and effective when considering only a few target compression ratios.

Future work in efficient model compression can explore ways to combine the two perspective studied in this thesis. More specifically, to better accelerate or compress models, one can harness the temporal redundancy as in AdaScale. While AdaScale uses the input resolution as a knob to control efficiency by exploiting the temporal redundancy, one can imagine to morph the network architecture to harness the temporal redundancy. To achieve this, one needs some weight-sharing network such as the one introduced in Joslim to be able to adapt the architecture online according to temporal cues without needing any re-training. It is unclear how one can train such a weight-sharing network to best leverage the temporal redundancy. On the other hand, due to the rising awareness of data privacy, it is desirable to conduct training on resource-constrained devices. One natural future direction is to tackle the challenges posed by on-device training. More interestingly, as the data distribution changes online in embedded devices, it is unclear how to better adapt the model architectures on-device to reduce the cost of running CNNs to better fit the online distribution without labels. Lastly, with the rising importance of Transformers for visual tasks [51], efficient compression for Transformers is also a natural extension for future work.

Bibliography

[1] Yonathan Aflalo, Asaf Noy, Ming Lin, Itamar Friedman, and Lihi Zelnik. Knapsack pruning with inner distillation. *arXiv preprint arXiv:2002.08258*, 2020. 86

[2] Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian Reid, Stephen Gould, and Anton Van Den Hengel. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3674–3683, 2018. 1, 37

[3] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015. 1, 37

[4] Haoping Bai, Meng Cao, Ping Huang, and Jiulong Shan. Batchquant: Quantized-for-all architecture search with robust quantizer. *arXiv preprint arXiv:2105.08952*, 2021. 87

[5] Maximilian Balandat, Brian Karrer, Daniel R Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. Botorch: Programmable bayesian optimization in pytorch. *arXiv preprint arXiv:1910.06403*, 2019. 117

[6] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3, 2021. 86

[7] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Aciq: Analytical clipping for integer quantization of neural networks. 2018. 85

[8] Maximiliana Behnke and Kenneth Heafield. Losing heads in the lottery: Pruning transformer attention in neural machine translation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 2664–2674, 2020. 83

[9] Sean Bell, C Lawrence Zitnick, Kavita Bala, and Ross Girshick. Inside-outside net: Detecting objects in context with skip pooling and recurrent neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2874–2883, 2016. 11, 88

[10] Irwan Bello. Lambdanetworks: Modeling long-range interactions without attention. *arXiv preprint arXiv:2102.08602*, 2021. 82

[11] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 8

[12] Maxim Berman, Leonid Pishchulin, Ning Xu, Gérard Medioni, et al. Aows: Adaptive and optimal network width search with latency constraints. *Proceedings IEEE CVPR*, 2020. 51, 64, 81, 84, 86

[13] Behzad Boroujerdian, Hasan Genc, Srivatsan Krishnan, Wenzhi Cui, Aleksandra Faust, and Vijay Reddi. Mavbench: Micro aerial vehicle benchmarking. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 894–907. IEEE, 2018. 37

[14] M. Buckler, P. Bedoukian, S. Jayasuriya, and A. Sampson. Eva²: Exploiting temporal redundancy in live computer vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 533–546, June 2018. 10

[15] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. 2, 57, 59, 82, 86, 87

[16] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018. 2, 81, 84, 86

[17] Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13169–13178, 2020. 85

[18] Zhaowei Cai, Quanfu Fan, Rogerio S Feris, and Nuno Vasconcelos. A unified multi-scale deep convolutional neural network for fast object detection. In *European Conference on Computer Vision*, pages 354–370. Springer, 2016. 11, 88

[19] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pages 213–229. Springer, 2020. 82

[20] Francesco Paolo Casale, Jonathan Gordon, and Nicolo Fusi. Probabilistic neural architecture search. *arXiv preprint arXiv:1902.05116*, 2019. 87

[21] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017. 1

[22] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *Proceedings of LearningSys*, 2015. 14, 17

[23] Ting-Wu Chin, Pierce I-Jen Chuang, Vikas Chandra, and Diana Marculescu. One weight bitwidth to rule them all. In Adrien Bartoli and Andrea Fusiello, editors, *Computer Vision – ECCV 2020 Workshops*, pages 85–103, Cham, 2020. Springer International Publishing. 3

[24] Ting-Wu Chin, Ruizhou Ding, and Diana Marculescu. Adascale: Towards real-time video object detection using adaptive scaling. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 431–441, 2019. 2, 84

[25] Ting-Wu Chin, Ruizhou Ding, Cha Zhang, and Diana Marculescu. Towards efficient model compression via learned global ranking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1518–1528, 2020. 3, 76, 84, 86

[26] Ting-Wu Chin, Diana Marculescu, and Ari S. Morcos. Width transfer: On the (in)variance of width optimization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 2990–2999, June 2021. 4

[27] Ting-Wu Chin, Ari S Morcos, and Diana Marculescu. Joslim: Joint widths and weights optimization for slimmable neural networks. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, 2021. 4, 64, 81, 84, 86

[28] T. W. Chin, C. L. Yu, M. Halpern, H. Genc, S. L. Tsao, and V. J. Reddi. Domain-specific approximation for object detection. *IEEE Micro*, 38(1):31–40, January 2018. 11, 19, 88

[29] Ting-Wu Chin, Cha Zhang, and Diana Marculescu. Renofeation: A simple transfer learning method for improved adversarial robustness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3243–3252, 2021. 80

[30] Chung-Cheng Chiu, Tara N Sainath, Yonghui Wu, Rohit Prabhavalkar, Patrick Nguyen, Zhifeng Chen, Anjuli Kannan, Ron J Weiss, Kanishka Rao, Ekaterina Gonina, et al. State-of-the-art speech recognition with sequence-to-sequence models. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4774–4778. IEEE, 2018. 36

[31] Jungwook Choi, Pierce I-Jen Chuang, Zhuo Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Bridging the accuracy gap for 2-bit quantized neural networks (qnn). *arXiv preprint arXiv:1807.06964*, 2018. 9, 84, 85

[32] Yoojin Choi, Jihwan Choi, Mostafa El-Khamy, and Jungwon Lee. Data-free network quantization with adversarial knowledge distillation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 710–711, 2020. 85

[33] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv preprint arXiv:1511.00363*, 2015. 1, 2

[34] Bin Dai, Chen Zhu, and David Wipf. Compressing neural networks using the variational information bottleneck. *arXiv preprint arXiv:1802.10399*, 2018. 36, 45

[35] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Advances in neural information processing systems*, pages 379–387, 2016. x, 2, 10, 11, 12, 14, 15, 16, 19, 20, 22, 88

[36] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei. Deformable convolutional networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*, volume 00, pages 764–773, Oct. 2018. 88

[37] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019. 2, 84, 86

[38] Zihang Dai, Zhilin Yang, Yiming Yang, William W Cohen, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019. 36

[39] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002. 54

[40] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009. 70

[41] Lei Deng, Guoqi Li, Song Han, Luping Shi, and Yuan Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020. 1, 6

[42] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 36

[43] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 8, 23, 85

[44] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. Regularizing activation distribution for training binarized deep networks. 2019. 84

[45] Xiaohan Ding, Guiguang Ding, Yuchen Guo, and Jungong Han. Centripetal sgd for pruning very deep convolutional networks with complicated structure. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4943–4953, 2019. 46

[46] Xiaohan Ding, Guiguang Ding, Yuchen Guo, Jungong Han, and Chenggang Yan. Approximated oracle filter pruning for destructive cnn width optimization. *arXiv preprint arXiv:1905.04748*, 2019. 46

[47] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. *arXiv preprint arXiv:1806.08198*, 2018. 84

[48] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Ppp-net: Platform-aware progressive search for pareto-optimal neural architectures, 2018. 86

[49] Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. In *Advances in Neural Information Processing Systems*, pages 759–770, 2019. 64

[50] Zhen Dong, Zhewei Yao, Yaohui Cai, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *arXiv preprint arXiv:1911.03852*, 2019. 23, 85

[51] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 82, 92

[52] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55):1–21, 2019. 6

[53] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2154, 2014. 12

[54] Christoph Feichtenhofer, Haoqi Fan, Jitendra Malik, and Kaiming He. Slowfast networks for video recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6202–6211, 2019. 2

[55] Christoph Feichtenhofer, Axel Pinz, and Andrew Zisserman. Detect to track and track to detect. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3038–3046, 2017. 10, 16, 20, 89

[56] Shangqian Gao, Feihu Huang, Jian Pei, and Heng Huang. Discrete model compression with resource constraint for deep neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1899–1908, 2020. 7

[57] Ross Girshick. Fast r-cnn. In *Computer Vision (ICCV), 2015 IEEE International Conference on*, pages 1440–1448. IEEE, 2015. 10, 11, 12, 15, 88

[58] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z Pan. Mixed precision neural architecture search for energy efficient deep learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7. IEEE, 2019. 85

[59] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018. 3, 7, 36, 43, 51, 64, 67, 69, 70, 84, 86

[60] Shaopeng Guo, Yujie Wang, Quanquan Li, and Junjie Yan. Dmcp: Differentiable markov channel pruning for neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1539–1547, 2020. 3, 51, 64, 65, 67, 68, 69, 70, 81, 84

[61] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019. 86

[62] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *European Conference on Computer Vision*, pages 544–560. Springer, 2020. 84, 86

[63] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations*, 2016. 2

[64] 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. 7

[65] Wei Han, Pooya Khorrami, Tom Le Paine, Prajit Ramachandran, Mohammad Babaeizadeh, Honghui Shi, Jianan Li, Shuicheng Yan, and Thomas S Huang. Seq-nms for video object detection. *arXiv preprint arXiv:1602.08465*, 2016. x, 20

[66] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017. 36

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. In *european conference on computer vision*, pages 346–361. Springer, 2014. 10, 11, 88

[68] 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, 2, 3, 4, 6, 26, 36, 66, 69, 70, 77, 88, 110

[69] Yang He, Yuhang Ding, Ping Liu, Linchao Zhu, Hanwang Zhang, and Yi Yang. Learning filter pruning criteria for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 7, 84

[70] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *IJCAI*, pages 2234–2240, 2018. 42, 44, 45, 46, 76

[71] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018. 3, 7, 36, 39, 42, 43, 45, 46, 49, 64, 67, 76, 84, 86, 114, 115

[72] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4340–4349, 2019. 7, 46

[73] Yihui He, Xianggen Liu, Huasong Zhong, and Yuchun Ma. Addressnet: Shift-based primitives for efficient convolutional neural networks. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1213–1222. IEEE, 2019. 84

[74] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017. 45, 46

[75] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *International Conference on Learning Representations*, 2019. 81

[76] Lu Hou and James T. Kwok. Loss-aware weight quantization of deep networks. In *International Conference on Learning Representations*, 2018. 8, 9, 23, 84, 85

[77] 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. 5, 6, 24, 28, 51, 84

[78] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. *group*, 3(12):11, 2017. 84

[79] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017. 6

[80] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *IEEE CVPR*, 2017. 11, 19

[81] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018. 1

[82] Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *The European Conference on Computer Vision (ECCV)*, September 2018. 46

[83] Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 304–320, 2018. 84

[84] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015. 6

[85] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1125–1134, 2017. 1

[86] 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. 2

[87] 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 *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 9, 84, 85

[88] Qing Jin, Linjie Yang, and Zhenyu Liao. Adabits: Neural network quantization with adaptive bit-widths. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 87

[89] Sangil Jung, Changyong Son, Seohyung Lee, Jinwoo Son, Jae-Joon Han, Youngjun Kwak, Sung Ju Hwang, and Changkyu Choi. Learning to quantize deep networks by optimizing quantization intervals with task loss. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 9, 84, 85

[90] Angjoo Kanazawa, Abhishek Sharma, and David Jacobs. Locally scale-invariant convolutional neural networks. In *NIPS workshop*, 2014. 88

[91] Kai Kang, Hongsheng Li, Junjie Yan, Xingyu Zeng, Bin Yang, Tong Xiao, Cong Zhang, Zhe Wang, Ruohui Wang, Xiaogang Wang, et al. T-cnn: Tubelets with convolutional neural networks for object detection from videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 2017. 10, 89

[92] Bumsoo Kim, Junhyun Lee, Jaewoo Kang, Eun-Sol Kim, and Hyunwoo J Kim. Hotr: End-to-end human-object interaction detection with transformers. *arXiv preprint arXiv:2104.13682*, 2021. 82

[93] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009. 3, 25, 42, 77

[94] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990. 7

[95] Bailin Li, Bowen Wu, Jiang Su, and Guangrun Wang. Eagleeye: Fast sub-net evaluation for efficient neural network pruning. In *European Conference on Computer Vision*, pages 639–654. Springer, 2020. 7, 84

[96] Changlin Li, Tao Tang, Guangrun Wang, Jiefeng Peng, Bing Wang, Xiaodan Liang, and Xiaojun Chang. Bossnas: Exploring hybrid cnn-transformers with block-wisely self-supervised neural architecture search. *arXiv preprint arXiv:2103.12424*, 2021. 86

[97] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. 8

[98] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. *arXiv preprint arXiv:1706.02379*, 2017. 1, 2, 8, 85

[99] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 2, 3, 7, 36, 39, 45, 68, 76, 84

[100] Xingjian Li, Haoyi Xiong, Hanchao Wang, Yuxuan Rao, Liping Liu, Zeyu Chen, and Jun Huan. Delta: Deep learning transfer using feature map with attention for convolutional networks. *arXiv preprint arXiv:1901.09229*, 2019. 80

[101] Xingjian Li, Haoyi Xiong, Hanchao Wang, Yuxuan Rao, Liping Liu, and Jun Huan. DELTA: DEEP LEARNING TRANSFER USING FEATURE MAP WITH ATTENTION FOR CONVOLUTIONAL NETWORKS. In *International Conference on Learning Representations*, 2019. 46

[102] Yawei Li, Shuhang Gu, Kai Zhang, Luc Van Gool, and Radu Timofte. Dhp: Differentiable meta pruning via hypernetworks. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision – ECCV 2020*, pages 608–624, Cham, 2020. Springer International Publishing. 76

[103] Yawei Li, Wen Li, Martin Danelljan, Kai Zhang, Shuhang Gu, Luc Van Gool, and Radu Timofte. The heterogeneity hypothesis: Finding layer-wise dissimilated network architecture. *arXiv preprint arXiv:2006.16242*, 2020. 64

[104] Lucas Liebenwein, Cenk Baykal, Brandon Carter, David Gifford, and Daniela Rus. Lost in pruning: The effects of pruning neural networks beyond test accuracy. *Proceedings of Machine Learning and Systems*, 3, 2021. 81

[105] Lucas Liebenwein, Cenk Baykal, Harry Lang, Dan Feldman, and Daniela Rus. Provable filter pruning for efficient neural networks. In *International Conference on Learning Representations*, 2020. 7, 84

[106] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. *arXiv preprint arXiv:2007.10319*, 2020. 86

[107] Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. Accelerating convolutional networks via global & dynamic filter pruning. In *IJCAI*, pages 2425–2432, 2018. 46

[108] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2790–2799, 2019. 46

[109] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *CVPR*, volume 1, page 4, 2017. 11, 88

[110] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollar. Focal loss for dense object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2980–2988, 2017. 11, 12, 19

[111] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. 53, 81, 87

[112] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016. 10, 11, 88

[113] Yu Liu, Hongyang Li, Junjie Yan, Fangyin Wei, Xiaogang Wang, and Xiaoou Tang. Recurrent scale approximation for object detection in cnn. In *IEEE International Conference on Computer Vision*, 2017. 88

[114] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Computer Vision (ICCV), 2017 IEEE International Conference on*, pages 2755–2763. IEEE, 2017. 36, 75, 84, 86

[115] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Tim Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE International Conference on Computer Vision*, 2019. 3, 46, 64, 70

[116] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. In *International Conference on Learning Representations*, 2019. 4, 64, 76, 81, 84, 87, 91

[117] Christos Louizos, Matthias Reisser, Tijmen Blankevoort, Efstratios Gavves, and Max Welling. Relaxed quantization for discretized neural networks. In *International Conference on Learning Representations*, 2019. 85

[118] Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning. In *Advances in Neural Information Processing Systems*, pages 3288–3298, 2017. 45

[119] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017. 84

[120] Zhichao Lu, Kalyanmoy Deb, Erik Goodman, Wolfgang Banzhaf, and Vishnu Naresh Boddeti. Nsganetv2: Evolutionary multi-objective surrogate-assisted neural architecture search. *arXiv preprint arXiv:2007.10396*, 2020. 87

[121] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342*, 2017. 45, 46

[122] Xingchen Ma, Amal Rannen Triki, Maxim Berman, Christos Sagonas, Jacques Cali, and Matthew B Blaschko. A bayesian optimization framework for neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 10274–10283, 2019. 7

[123] Arun Mallya and Svetlana Lazebnik. Packnet: Adding multiple tasks to a single network by iterative pruning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 46

[124] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017. 84

[125] Jiachen Mao, Huanrui Yang, Ang Li, Hai Li, and Yiran Chen. Tprune: Efficient transformer pruning for mobile devices. *ACM Transactions on Cyber-Physical Systems*, 5(3):1–22, 2021. 83

[126] Bertil Matérn. *Spatial variation*, volume 36. Springer Science & Business Media, 2013. 117

[127] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different: Recovering neural network quantization error through weight factorization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 4486–4495, Long Beach, California, USA, 09–15 Jun 2019. PMLR. 85

[128] Asit Mishra, Eriko Nurvitadhi, Jeffrey J Cook, and Debbie Marr. WRPN: Wide reduced-precision networks. In *International Conference on Learning Representations*, 2018. 85

[129] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019. 7, 46, 68, 76, 84, 86

[130] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *International Conference on Learning Representation (ICLR)*, 2017. 7, 45, 84

[131] Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? adaptive rounding for post-training quantization. In *International Conference on Machine Learning*, pages 7197–7206. PMLR, 2020. 85

[132] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. *arXiv preprint arXiv:1906.04721*, 2019. 85

[133] Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik-Manor. Hardcore-nas: Hard constrained differentiable neural architecture search. *arXiv preprint arXiv:2102.11646*, 2021. 2, 86

[134] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. Akad. Nauk SSSR*, volume 269, pages 543–547, 1983. 42

[135] Xuefei Ning, Tianchen Zhao, Wenshuo Li, Peng Lei, Yu Wang, and Huazhong Yang. Dsa: More efficient budgeted pruning via differentiable sparsity allocation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2020. 84

[136] Rameswar Panda, Michele Merler, Mayoore Jaiswal, Hui Wu, Kandan Ramakrishnan, Ulrich Finkler, Chun-Fu Chen, Minsik Cho, David Kung, Rogerio Feris, et al. Nastransfer: Analyzing architecture transferability in large scale neural architecture search. *arXiv preprint arXiv:2006.13314*, 2020. 87

[137] Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. A flexible framework for multi-objective bayesian optimization using random scalarizations. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, page 267. AUAI Press, 2019. 54, 55, 119

[138] Daniel S Park, William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D Cubuk, and Quoc V Le. Specaugment: A simple data augmentation method for automatic speech recognition. *arXiv preprint arXiv:1904.08779*, 2019. 36

[139] David Patterson, Joseph Gonzalez, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *arXiv preprint arXiv:2104.10350*, 2021. 22, 35, 49, 62

[140] Hanyu Peng, Jiaxiang Wu, Shifeng Chen, and Junzhou Huang. Collaborative channel pruning for deep networks. In *International Conference on Machine Learning*, pages 5113–5122, 2019. 36, 46

[141] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollar. Designing network design spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10428–10436, 2020. 65

[142] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*, 2021. 1

[143] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003. 55

[144] 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. 9, 84, 85

[145] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. *arXiv preprint arXiv:1802.01548*, 2018. 40, 42

[146] Esteban Real, Jonathon Shlens, Stefano Mazzocchi, Xin Pan, and Vincent Vanhoucke. Youtube-boundingboxes: A large high-precision human-annotated data set for object detection in video. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7464–7473. IEEE, 2017. 10, 15

[147] Joseph Redmon and Ali Farhadi. Yolo9000: Better, faster, stronger. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*, pages 6517–6525. IEEE, 2017. 11, 19

[148] Bin Ren, Hao Tang, Fanyang Meng, Runwei Ding, Ling Shao, Philip HS Torr, and Nicu Sebe. Cloth interactive transformer for virtual try-on. *arXiv preprint arXiv:2104.05519*, 2021. 82

[149] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NIPS)*, 2015. 1, 10, 12, 16

[150] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015. 36

[151] Shaoqing Ren, Kaiming He, Ross Girshick, Xiangyu Zhang, and Jian Sun. Object detection networks on convolutional feature maps. *IEEE transactions on pattern analysis and machine intelligence*, 39(7):1476–1481, 2017. 88

[152] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. 2, 3, 4, 10

[153] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015. 14, 42

[154] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018. 3, 4, 26, 69, 70, 110, 116

[155] Manolis Savva, Abhishek Kadian, Oleksandr Maksymets, Yili Zhao, Erik Wijmans, Bhavana Jain, Julian Straub, Jia Liu, Vladlen Koltun, Jitendra Malik, et al. Habitat: A platform for embodied ai research. *arXiv preprint arXiv:1904.01201*, 2019. 36

[156] Tao Sheng, Chen Feng, Shaojie Zhuo, Xiaopeng Zhang, Liang Shen, and Mickey Aleksic. A quantization-friendly separable convolution for mobilenets. In *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, pages 14–18. IEEE, 2018. 30, 85

[157] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 6, 26

[158] Dimitrios Stamoulis, Ting-Wu Rudy Chin, Anand Krishnan Prakash, Haocheng Fang, Sribhuvan Sajja, Mitchell Bognar, and Diana Marculescu. Designing adaptive neural networks for energy-constrained image classification. In *Proceedings of the International Conference on Computer-Aided Design*, page 23. ACM, 2018. 84

[159] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. *arXiv preprint arXiv:1904.02877*, 2019. 2, 84, 86

[160] Xiu Su, Shan You, Tao Huang, Fei Wang, Chen Qian, Changshui Zhang, and Chang Xu. Locally free weight sharing for network width search. In *International Conference on Learning Representations*, 2021. 64, 67, 81

[161] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. *arXiv preprint arXiv:1511.06488*, 2015. 85

[162] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on*

Computer Vision and Pattern Recognition, pages 2820–2828, 2019. 2, 53, 81, 84, 86

[163] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114, Long Beach, California, USA, 09–15 Jun 2019. PMLR. 84

[164] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019. 73, 84

[165] Raphael Tang, Weijie Wang, Zhucheng Tu, and Jimmy Lin. An experimental analysis of the power consumption of convolutional neural networks for keyword spotting. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5479–5483. IEEE, 2018. 1

[166] Makarand Tapaswi, Yukun Zhu, Rainer Stiefelhagen, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Movieqa: Understanding stories in movies through question-answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4631–4640, 2016. 37

[167] Lucas Theis, Iryna Korshunova, Alykhan Tejani, and Ferenc Huszár. Faster gaze prediction with dense networks and fisher pruning. *arXiv preprint arXiv:1801.05787*, 2018. 43, 84, 86

[168] Rishabh Tiwari, Udbhav Bamba, Arnav Chavan, and Deepak K Gupta. Chipnet: Budget-aware pruning with heaviside continuous approximations. *arXiv preprint arXiv:2102.07156*, 2021. 84

[169] Jeya Maria Jose Valanarasu, Poojan Oza, Ilker Hacihaliloglu, and Vishal M Patel. Medical transformer: Gated axial-attention for medical image segmentation. *arXiv preprint arXiv:2102.10662*, 2021. 82

[170] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. The caltech-ucsd birds-200-2011 dataset. 2011. 42

[171] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 81, 82, 86

[172] Chaoqi Wang, Roger Grosse, Sanja Fidler, and Guodong Zhang. Eigendamage: Structured pruning in the kronecker-factored eigenbasis. In *International Conference on Machine Learning*, pages 6566–6575. PMLR, 2019. 84, 86

[173] Dilin Wang, Chengyue Gong, Meng Li, Qiang Liu, and Vikas Chandra. Alphanet: Improved training of supernet with alpha-divergence. *arXiv preprint arXiv:2102.07954*, 2021. 86

[174] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. Hat: Hardware-aware transformers for efficient natural language processing. *arXiv preprint arXiv:2005.14187*, 2020. 83

[175] Huan Wang, Qiming Zhang, Yuehai Wang, and Haoji Hu. Structured probabilistic pruning for convolutional neural network acceleration. In *Proceedings of the British Machine Vision Conference (BMVC)*, 2018. 46

[176] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8612–8620, 2019. 23, 24, 33, 34, 85, 86

[177] Ning Wang, Wengang Zhou, Jie Wang, and Houqaing Li. Transformer meets tracker: Exploiting temporal context for robust visual tracking. *arXiv preprint arXiv:2103.11681*, 2021. 82

[178] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. Apq: Joint search for network architecture, pruning and quantization policy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2078–2087, 2020. 35, 86

[179] Yulin Wang, Ruida Huang, Shiji Song, Zeyi Huang, and Gao Huang. Not all images are worth 16x16 words: Dynamic vision transformers with adaptive sequence length. 2021. 82

[180] Yunhe Wang, Chang Xu, Jiayan Qiu, Chao Xu, and Dacheng Tao. Towards evolutionary compression. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2476–2485, 2018. 7

[181] 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. 7, 36, 84, 86

[182] Catherine Wong, Neil Houlsby, Yifeng Lu, and Andrea Gesmundo. Transfer learning with neural automl. In *Advances in Neural Information Processing Systems*, pages 8356–8365, 2018. 87

[183] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019. 2, 81, 84, 86

[184] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. Shift: A zero flop, zero parameter alternative to spatial convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9127–9135, 2018. 84

[185] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090*, 2018. 23, 85

[186] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations*, 2019. 81

[187] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang, and Xian-sheng Hua. Quantization networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 85

[188] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5687–5695, 2017. 1

[189] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018. 39, 84, 86

[190] Jianbo Ye, Xin Lu, Zhe Lin, and James Z Wang. Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers. *International Conference on Learning Representation (ICLR)*, 2018. 36

[191] Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. *arXiv preprint arXiv:2003.01794*, 2020. 79, 81, 84

[192] Haibao Yu, Qi Han, Jianbo Li, Jianping Shi, Guangliang Cheng, and Bin Fan. Search what you want: Barrier panelty nas for mixed precision quantization. In *European Conference on Computer Vision*, pages 1–16. Springer, 2020. 85

[193] Jiahui Yu and Thomas Huang. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv preprint arXiv:1903.11728*, 8, 2019. 3, 51, 64, 68, 69, 81, 84, 120

[194] Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1803–1811, 2019. 51, 54, 56, 57, 58, 59, 79, 116, 118

[195] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. Bignas: Scaling up neural architecture search with big single-stage models. *arXiv preprint arXiv:2003.11142*, 2020. 51, 57, 81, 82, 86, 87

[196] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *International Conference on Learning Representations*, 2019. 51, 61, 81, 86, 119

[197] Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. Nisp: Pruning networks using neuron importance score propagation. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 46

[198] Xin Yuan, Liangliang Ren, Jiwen Lu, and Jie Zhou. Enhanced bayesian compression via deep reinforcement learning. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019. 9, 84, 85

[199] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016. 1

[200] Chiyuan Zhang, Samy Bengio, and Yoram Singer. Are all layers created equal? *arXiv preprint arXiv:1902.01996*, 2019. 51

[201] 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. 6

[202] Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2780–2789, 2019. 46

[203] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7543–7552, Long Beach, California, USA, 09–15 Jun 2019. PMLR. 85

[204] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning*, pages 7543–7552. PMLR, 2019. 85

[205] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhiru Zhang. Building efficient deep neural networks with unitary group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11303–11312, 2019. 84

[206] Sixiao Zheng, Jiachen Lu, Hengshuang Zhao, Xiatian Zhu, Zekun Luo, Yabiao Wang, Yanwei Fu, Jianfeng Feng, Tao Xiang, Philip HS Torr, et al. Rethinking semantic segmentation from a sequence-to-sequence perspective with transformers. *arXiv preprint arXiv:2012.15840*, 2020. 82

[207] Yang Zhong, Vladimir Li, Ryuzo Okada, and Atsuto Maki. Target aware network adaptation for efficient representation learning. *arXiv preprint arXiv:1810.01104*, 2018. 45

[208] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. In *International Conference on Learning Representations*, 2017. 8, 23, 85

[209] Dongzhan Zhou, Xinchi Zhou, Wenwei Zhang, Chen Change Loy, Shuai Yi, Xuesen Zhang, and Wanli Ouyang. Econas: Finding proxies for economical neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11396–11404, 2020. 82, 87

[210] Huajun Zhou, Zechao Li, Chengcheng Ning, and Jinhui Tang. Cad: Scale invariant framework for real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 760–768, 2017. 88

[211] 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. 8, 23, 85

[212] Yanqi Zhou, Siavash Ebrahimi, Sercan Ö Arık, Haonan Yu, Hairong Liu, and Greg Diamos. Resource-efficient neural architect. *arXiv preprint arXiv:1806.07912*, 2018. 84

[213] Yuefu Zhou, Ya Zhang, Yanfeng Wang, and Qi Tian. Accelerate cnn via recursive bayesian pruning. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3306–3315, 2019. 36, 46, 84

[214] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In *International Conference on Learning Representations*, 2017. 9, 84, 85

[215] Xizhou Zhu, Jifeng Dai, Lu Yuan, and Yichen Wei. Towards high performance video object detection. In *IEEE CVPR*, 2018. 89

[216] Xizhou Zhu, Yujie Wang, Jifeng Dai, Lu Yuan, and Yichen Wei. Flow-guided feature aggregation for video object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 408–417, 2017. x, 10, 16, 20, 89

[217] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. In *Proc. CVPR*, volume 2, page 7, 2017. 10, 12, 14, 15, 16, 20, 88, 89

[218] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 547–560, June 2018. 10

[219] Zhuangwei Zhuang, Mingkui Tan, Bohan Zhuang, Jing Liu, Yong Guo, Qingyao Wu, Junzhou Huang, and Jinhui Zhu. Discrimination-aware channel pruning for deep neural networks. In *Advances in Neural Information Processing Systems*, pages 883–894, 2018. 42, 44, 45, 46

[220] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018. 87

Appendix A

Appendix for Chapter 4

A.1 Network Architectures

For the experiments in Section 4.2.2, the ResNets used are detailed in Table A.1. Specifically, for the points in Fig. 4.1a, we consider ResNet20 to ResNet56 with width-multipliers of $0.5\times, 1\times, 1.5\times$, and $2\times$ for the 4-bit case. Based on these values, we consider additional width-multipliers $2.4\times$ and $2.8\times$ for the 2-bit case and $2.5\times, 3\times, 3.5\times$, and $3.9\times$ for the 1-bit case. We note that the right-most points in Fig. 4.1a is a $10\times$ ResNet26 for the 4 bits case. On the other hand, VGG11 is detailed in Table A.3 for which we consider width-multipliers from $0.25\times$ to $2\times$ with a step of 0.25 for the 4 bits case (blue dots in Fig. 4.1b). The architecture of MobileNetV2 used in the CIFAR-100 experiments follows the original MobileNetV2 (Table 2 in [154]) but we change the stride of all the bottleneck blocks to 1 except for the fifth bottleneck block, which has a stride of 2. As a result, we down-sample the image twice in total, which resembles the ResNet design for the CIFAR experiments [68]. Similar to VGG11, we consider width-multipliers from $0.25\times$ to $2\times$ with a step of 0.25 for MobileNetV2 for the 4 bits case (blue dots in Fig. 4.1c).

Table A.1: ResNet20 to ResNet56

Layers	20	26	32	38	44	50	56
Stem	Conv2d (16,3,3) Stride 1						
Stage 1	$3\times$ {Conv2d(16,3,3) Stride 1; Conv2d(16,3,3) Stride 1}	$4\times$	$5\times$	$6\times$	$7\times$	$8\times$	$9\times$
Stage 2	$3\times$ {Conv2d(32,3,3) Stride 2; Conv2d(32,3,3) Stride 1}	$4\times$	$5\times$	$6\times$	$7\times$	$8\times$	$9\times$
Stage 3	$3\times$ {Conv2d(64,3,3) Stride 2; Conv2d(64,3,3) Stride 1}	$4\times$	$5\times$	$6\times$	$7\times$	$8\times$	$9\times$

Table A.2: Inv-ResNet26

Stem	Conv2d (16,3,3) Stride 1
Stage 1	$4 \times \begin{cases} \text{Conv2d}(16 \times 6, 1, 1) \text{ Stride 1} \\ \text{DWConv2d}(16 \times 6, 3, 3) \text{ Stride 1} \\ \text{Conv2d}(16, 1, 1) \text{ Stride 1} \end{cases}$
Stage 2	$4 \times \begin{cases} \text{Conv2d}(32 \times 6, 1, 1) \text{ Stride 1} \\ \text{DWConv2d}(32 \times 6, 3, 3) \text{ Stride 2} \\ \text{Conv2d}(32, 1, 1) \text{ Stride 1} \end{cases}$
Stage 3	$4 \times \begin{cases} \text{Conv2d}(64 \times 6, 1, 1) \text{ Stride 1} \\ \text{DWConv2d}(64 \times 6, 3, 3) \text{ Stride 2} \\ \text{Conv2d}(64, 1, 1) \text{ Stride 1} \end{cases}$

A.2 Proof For Proposition 4.2.1

Based on the definition of variance, we have:

$$\begin{aligned}
\text{Var}(\frac{1}{d}\sum_{i=1}^{d}|\boldsymbol{w}_i|) &:= \mathbb{E}\left[\left(\frac{1}{d}\sum_{i=1}^{d}|\boldsymbol{w}_i|\right)^2 - \left(\mathbb{E}\frac{1}{d}\sum_{i=1}^{d}|\boldsymbol{w}_i|\right)^2\right] \\
&= \mathbb{E}\left[\left(\frac{1}{d}\sum_{i=1}^{d}|\boldsymbol{w}_i|\right)^2 - \frac{2\sigma^2}{\pi}\right] \\
&= \frac{1}{d^2}\mathbb{E}\left(\sum_{i=1}^{d}|\boldsymbol{w}_i|\right)^2 - \frac{2\sigma^2}{\pi} \\
&= \frac{\sigma^2}{d} + \frac{d-1}{d}\rho\sigma^2 - \frac{2\sigma^2}{\pi}.
\end{aligned}$$

Table A.3: VGGs

VGG11	Variant A	Variant B	Variant C
Conv2d (64,3,3)			
MaxPooling			
Conv2d (128,3,3)	$\begin{cases}\text{Conv2d}(128,1,1)\\ \text{DWConv2d}(128,3,3)\end{cases}$	$\begin{cases}\text{Conv2d}(128,1,1)\\ \text{DWConv2d}(128,3,3)\end{cases}$	$\begin{cases}\text{Conv2d}(128,1,1)\\ \text{DWConv2d}(128,3,3)\end{cases}$
MaxPooling			
Conv2d (256,3,3)	Conv2d (256,3,3)	$\begin{cases}\text{Conv2d}(256,1,1)\\ \text{DWConv2d}(256,3,3)\end{cases}$	$\begin{cases}\text{Conv2d}(256,1,1)\\ \text{DWConv2d}(256,3,3)\end{cases}$
Conv2d (256,3,3)	Conv2d (256,3,3)	$\begin{cases}\text{Conv2d}(256,1,1)\\ \text{DWConv2d}(256,3,3)\end{cases}$	$\begin{cases}\text{Conv2d}(256,1,1)\\ \text{DWConv2d}(256,3,3)\end{cases}$
MaxPooling			
Conv2d (512,3,3)	Conv2d (512,3,3)	$\begin{cases}\text{Conv2d}(512,1,1)\\ \text{DWConv2d}(512,3,3)\end{cases}$	$\begin{cases}\text{Conv2d}(512,1,1)\\ \text{DWConv2d}(512,3,3)\end{cases}$
Conv2d (512,3,3)	Conv2d (512,3,3)	Conv2d (512,3,3)	$\begin{cases}\text{Conv2d}(512,1,1)\\ \text{DWConv2d}(512,3,3)\end{cases}$
MaxPooling			
Conv2d (512,3,3)	Conv2d (512,3,3)	Conv2d (512,3,3)	$\begin{cases}\text{Conv2d}(512,1,1)\\ \text{DWConv2d}(512,3,3)\end{cases}$
Conv2d (512,3,3)	Conv2d (512,3,3)	Conv2d (512,3,3)	$\begin{cases}\text{Conv2d}(512,1,1)\\ \text{DWConv2d}(512,3,3)\end{cases}$
MaxPooling			

Appendix B

Appendix for Chapter 5

B.1 Optimization Interpretation Of LeGR

LeGR can be interpreted as minimizing a surrogate of a derived upper bound for the loss difference between (1) the pruned-and-fine-tuned CNN and (2) the pre-trained CNN. Concretely, we would like to solve for the filter masking binary variables $\boldsymbol{z} \in \{0,1\}^K$, with K being the number of filters. If a filter k is pruned, the corresponding mask will be zero ($z_k = 0$), otherwise it will be one ($z_k = 1$). Thus, we have the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{z}} \ \mathbb{L}(\boldsymbol{\theta} \odot \boldsymbol{z} - \eta \sum_{j=1}^{\tau} \Delta \boldsymbol{w}^{(j)} \odot \boldsymbol{z}) - \mathbb{L}(\boldsymbol{\theta}) \\ \text{s.t. } C(\boldsymbol{z}) \leq \zeta, \end{aligned} \tag{B.1}$$

where $\boldsymbol{\theta}$ denotes all the filters of the CNN, $\mathbb{L}(\boldsymbol{\theta}) = \frac{1}{|D|} \sum_{(x,y) \in D} \ L(f(x|\boldsymbol{\theta}), y)$ denotes the loss function of filters where x and y are the input and label, respectively. D denotes the training data, f is the CNN model and L is the loss function for prediction (*e.g.*, cross entropy loss). η denotes the learning rate, τ denotes the number of gradient steps, $\Delta \boldsymbol{w}^{(j)}$ denotes the gradient with respect to the filter weights computed at step j, and $\odot$ denotes element-wise multiplication. On the constraint side, $C(\cdot)$ is the modeling function for FLOP count and ζ is the desired FLOP count constraint. By fine-tuning, we mean updating the filter weights with stochastic gradient descent (SGD) for τ steps.

Let us assume the loss function $\mathbb{L}$ is Ω_l-Lipschitz continuous for the l-th layer of the CNN, then the

following holds:

$$\begin{aligned}
&\mathbb{L}(\boldsymbol{\theta} \odot \boldsymbol{z} - \eta \sum_{j=1}^{\tau} \Delta \boldsymbol{w}^{(j)} \odot \boldsymbol{z}) - \mathbb{L}(\boldsymbol{\theta}) \\
&\leq \mathbb{L}(\boldsymbol{\theta} \odot \boldsymbol{z}) + \sum_{i=1}^{K} \Omega_{l(i)} \eta \left\| \sum_{j=1}^{\tau} \Delta \boldsymbol{w}_i^{(j)} \odot \boldsymbol{z}_i \right\| - \mathbb{L}(\boldsymbol{\theta}) \\
&\leq \sum_{i=1}^{K} \Omega_{l(i)} \left\| \boldsymbol{\theta}_i \right\| \boldsymbol{h}_i + \sum_{i=1}^{K} \Omega_{l(i)}^2 \eta \tau \boldsymbol{z}_i \\
&= \sum_{i=1}^{K} (\Omega_{l(i)} \left\| \boldsymbol{\theta}_i \right\| - \Omega_{l(i)}^2 \eta \tau) \boldsymbol{h}_i + \Omega_{l(i)}^2 \eta \tau,
\end{aligned} \tag{B.2}$$

where $l(i)$ is the layer index for the i-th filter, $\boldsymbol{h} = \mathbf{1} - \boldsymbol{z}$, and $\|\cdot\|$ denotes ℓ_2 norms.

On the constraint side of equation (B.1), let $R_{l(i)}$ be the FLOP count of layer $l(i)$ where filter i resides. Analytically, the FLOP count of a layer depends linearly on the number of filters in its preceding layer:

$$R_{l(i)} = u_{l(i)} \left\| \{ \boldsymbol{z} : \boldsymbol{z}_j \; \forall j \in P(l(i)) \} \right\|_0, \; u_{l(i)} \geq 0, \tag{B.3}$$

where $P(l(i))$ returns a set of filter indices for the layer that precedes layer $l(i)$ and $u_{l(i)}$ is a layer-dependent positive constant. Let $\hat{R}_{l(i)}$ denote the FLOP count for layer $l(i)$ for the pre-trained network ($\boldsymbol{z} = \mathbf{1}$), one can see from equation (B.3) that $R_{l(i)} \leq \hat{R}_{l(i)} \; \forall i, \boldsymbol{z}$. Thus, the following holds:

$$C(\mathbf{1} - \boldsymbol{h}) = \sum_{i}^{K} R_{l(i)} (1 - \boldsymbol{h}_i) \leq \sum_{i}^{K} \hat{R}_{l(i)} (1 - \boldsymbol{h}_i). \tag{B.4}$$

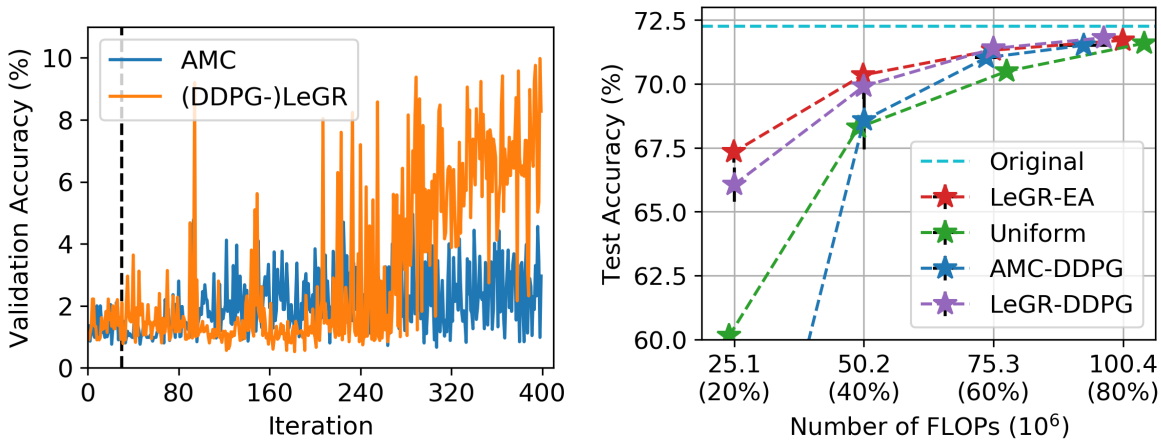
Based on equations (B.2) and (B.4), instead of minimizing equation (B.1), we minimize its upper bound in a Lagrangian form. That is,

$$\min_{\boldsymbol{h}} \sum_{i=1}^{K} \left(\alpha_{l(i)} \left\| \boldsymbol{\theta}_i \right\| + \kappa_{l(i)} \right) \boldsymbol{h}_i, \tag{B.5}$$

where $\alpha_{l(i)} = \Omega_{l(i)}$ and $\kappa_{l(i)} = \eta \tau \Omega_{l(i)}^2 - \lambda \hat{R}_{l(i)}$. To guarantee the solution will satisfy the constraint, we rank all filters by their scores $s_i = \alpha_{l(i)} \left\| \boldsymbol{\theta}_i \right\| + \kappa_{l(i)} \; \forall \; i$ and threshold out the bottom ranked (small in scores) filters such that the constraint $C(\mathbf{1} - \boldsymbol{h}) \leq \zeta$ is satisfied and $\|\boldsymbol{h}\|_0$ is maximized. That is, LeGR can be viewed as learning to estimate $\boldsymbol{\alpha}$ and $\boldsymbol{\kappa}$ by assuming that better estimates of $\boldsymbol{\alpha}$-$\boldsymbol{\kappa}$ produce a better solution for the original objective (B.1) by solving the surrogate of the upper bound (B.5).

B.2 LeGR-DDPG

We have also tried learning the layer-wise affine transformations with actor-critic policy gradient (DDPG), which is adopted in prior art [71]. We use DDPG in a sequential fashion that follows [71]. LeGR requires two continuous actions (*i.e.*, α_l and κ_l) for layer l while AMC needs only one action (*i.e.*, percentage).



(a) (b)

Figure B.1: Comparison between searching the layer-wise filter norms and searching the layer-wise filter percentage. (a) compares the searching progress for 50% FLOP count ResNet-56 and (b) compares the final performance for ResNet-56 with various constraint levels.

We conduct the comparison of pruning ResNet-56 to 50% of its original FLOP count targeting CIFAR-100 with $\hat{\tau} = 0$ and hyper-parameters following [71]. As shown in Fig. B.1a, while both LeGR and AMC outperform random search (iterations before the vertical black-dotted line), LeGR converges faster to a better solution. Beyond comparing the progress of searching, we also compare the performance of the final pruned networks. As shown in Fig. B.1b, searching layer-wise affine transformations is more efficient and effective compared to searching the layer-wise filter percentages. Comparing LeGR using the two policy improvement methods, we empirically find that DDPG incurs larger variance on the final network than evolutionary algorithm.

Appendix C

Appendix for Chapter 6

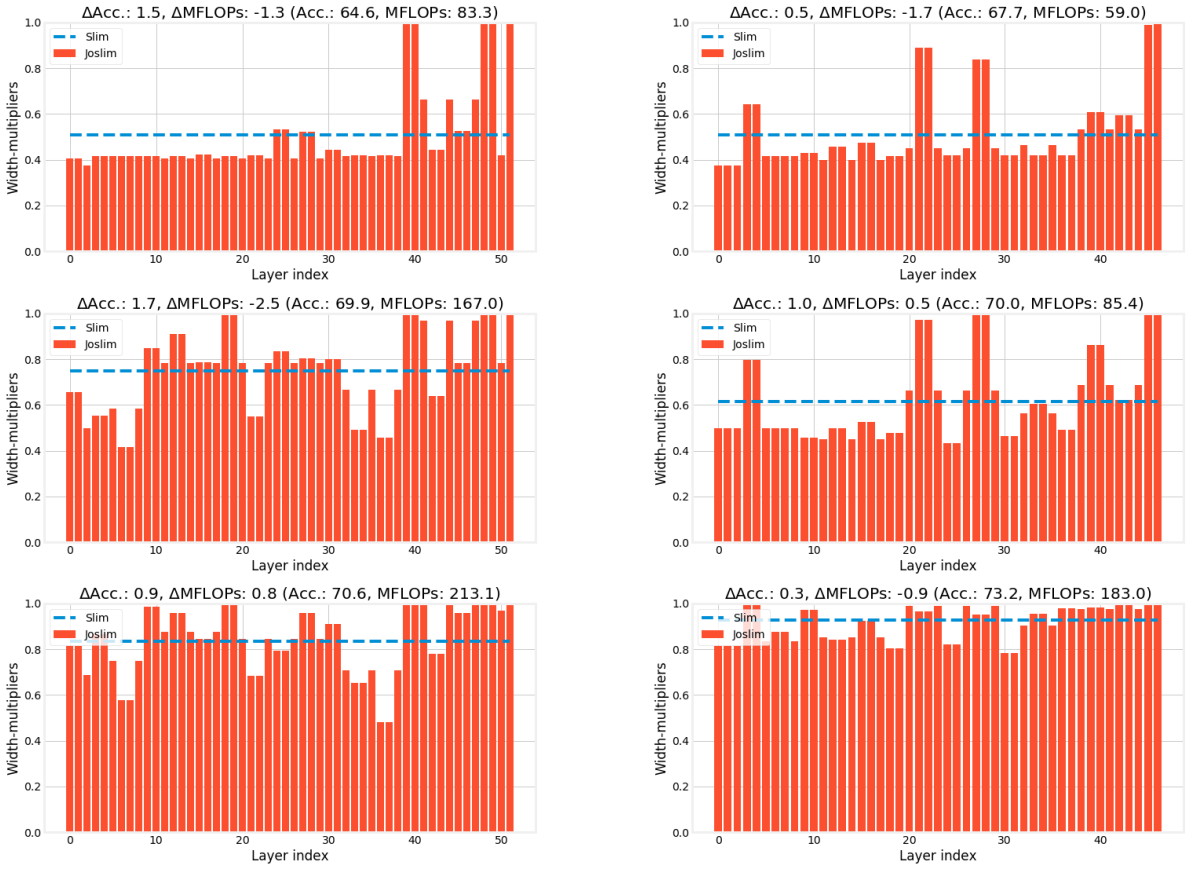
C.1 Width Parameterization

For ResNets with CIFAR, $\boldsymbol{a}$ has six dimensions and is denoted by $\boldsymbol{a}_{1:6} \in [0.316, 1]$, *i.e.*, one parameter for each stage and one for each residual connected layers in three stages. More specifically, the network is divided into three stages according to the output resolution, and as a result, there are three stages for all the ResNets designed for CIFAR. For example, in ResNet20, there are 7, 6, and 6 layers for each of the stages, respectively. Also, the layers that are added together via residual connection have to share the same width-multiplier, which results in one width-multiplier per stage for the layers that are connected via residual connections.

For MobileNetV2, $\boldsymbol{a}_{1:25} \in [0.42, 1]$, and therefore there is one dimension for each independent convolutional layer. Note that while there are in total 52 convolutional layers in MobileNetV2, not all of them can be altered independently. More specifically, for layers that are added together via residual connection, their widths should be identical. Similarly, the depth-wise convolutional layer should have the same width as its preceding point-wise convolutional layers. The same logic applies to MobileNetV3, which has 47 convolutional layers (excluding squeeze-and-excitation layers) and $\boldsymbol{a}_{1:22} \in [0.42, 1]$. In MobileNetV3, there are squeeze-and-excitation (SE) layers and we do not alter the width for the expansion layer in the SE layer. The output width of the SE layer is set to be the same as that of the convolutional layer where the SE layer is applied to. Note that there is no concept of expansion ratio for the inverted residual block in MobileNets in our width optimization. More specifically, the convolutional layer that acts upon expansion ratio is in itself just a convolutional layer with tunable width. Also, we do not quantize the width to be multiples of 8 as adopted in the previous work [154, 194]. Due to these reasons, our $0.42\times$ MobileNetV2 has 59 MFLOPs, which has the same FLOPs as the $0.35\times$ MobileNetV2 in [194, 154].

C.2 Width Differences

In Fig. C.1, we visualize the widths learned by Joslim and contrast them with Slim for MobileNetV2 and MobileNetV3. Note that both Joslim and Slim are slimmable networks with shared weights and from the top row to the bottom row represent three points on the trade-off curve for Fig. 6.3a and Fig. 6.3c.



(a) MobileNetV2 ImageNet

(b) MobileNetV3 ImageNet

Figure C.1: Comparing the width-multipliers between Joslim and Slim. The title for each plot denotes the relative differences (Joslim - Slim) and the numbers in the parenthesis are for Joslim.

C.3 Training Hyperparameters

We use PyTorch as our deep learning framework and we use BoTorch [5] for the implementation of MOBO-RS, which works seamlessly with PyTorch. More specifically, for the covariance function of Gaussian Processes, we use the commonly adopted Matérn Kernel [126] without changing the default hyperparameters provided in BoTorch. Similarly, we use the default hyperparameter provided in BoTorch for the Upper Confidence Bound acquisition function. To perform the optimization of line 6 in Algorithm 4, we make use of the API "*optimize_acqf*" provided in BoTorch. As a reference, with a single 1080Ti GPU, one can

train a Joslim-ResNet20 on CIFAR-100 with around 3 hours. On the other hand, with 8 V100 GPUs on a single machine, one can train a Joslim-ResNet18 on ImageNet with 19 hours.

CIFAR The training hyperparameters for the independent models are 0.1 initial learning rate, 200 training epochs, 0.0005 weight decay, 128 batch size, SGD with nesterov momentum, and cosine learning rate decay. The accuracy on the validation set is reported using the model at the final epoch. For slimmable training, we keep the same exact hyperparameters but train $2\times$ longer compared to independent models, *i.e.*, 400 epochs.

ImageNet Our training hyperparameters follow that of [194]. Specifically, we use initial learning rate of 0.5 with 5 epochs linear warmup (from 0 to 0.5), linear learning rate decay (from 0.5 to 0), 250 epochs, $4e^{-5}$ weight decay, 0.1 label smoothing, and we use SGD with 0.9 nesterov momentum. We use a batch size of 1024. For data augmentation, we use the "RandomResizedCrop" and "RandomHorizontalFlip" APIs in PyTorch. For MobileNetV2 we follow [194] and use random scale between 0.25 to 1. For MobileNetV3, we use the default scale parameters, *i.e.*, from 0.08 to 1. The input resolution we use is 224. Besides scaling and horizontal flip, we follow [194] and use color and lighting jitters data augmentataion with parameter of 0.4 for brightness, contrast, and saturation; and 0.1 for lighting. These augmentations can be found in the official repository of [194][1]. The hyperparamters for training ResNet18 is identical to MobileNetV2 except that we train it for 100 epochs only. The training for ImageNet is done using 8 NVIDIA V100 GPUs.

C.4 Theoretical Analysis For Temporal Sharing

The intuition behind the proposed approximation in Section 6.2.2 is the similarity for $\boldsymbol{\theta}$ across alternating minimization. In an extreme case, if we hold $\boldsymbol{\theta}$ constant throughout the training procedure, the approximation is equivalent to the original multi-objective BO. With that said, $\boldsymbol{\theta}$ changes gradually throughout training. To proceed with further theoretical understanding, we assume the loss $L_{\mathcal{S}}(\boldsymbol{\theta})$ is L-Lipschitz. More formally,

$$L_{\mathcal{S}}(\boldsymbol{\theta}^t) - L_{\mathcal{S}}(\boldsymbol{\theta}^{t+1}) \leq L\|\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t+1}\|_1, \forall \boldsymbol{\theta}^t, \boldsymbol{\theta}^{t+1}. \tag{C.1}$$

Now, consider using stochastic gradient descent to update the weights $\boldsymbol{\theta}$, *i.e.*, $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta^t \boldsymbol{g}^t$ where $\boldsymbol{g}^t$ is the gradient of loss with respect to the weights and η^t is the learning rate at iteration t. Since $L_{\mathcal{S}}$ is L-Lipschitz, we have $\|\boldsymbol{g}\|_1 \leq L$. Assuming using an exponential decaying learning rate with a factor $\gamma < 1$, we can further upper bound the functional differences across K iterations of gradient descents as

[1] https://github.com/JiahuiYu/slimmable_networks/blob/master/train.py#L43

follows:

$$L_{\mathcal{S}}(\boldsymbol{\theta}^t) - L_{\mathcal{S}}(\boldsymbol{\theta}^{t+n}) \leq \sum_{i=t}^{t+K} \eta^i \|g^i\| \leq K\eta^t L. \tag{C.2}$$

Aligning with our intuition, the analysis reveals that larger K implies poorer approximation for Bayesian optimization to share history. In multi-objective Bayesian optimization [137], the hyperparameter is searched over stationary objectives. In our case, due to temporal approximation, our cross entropy changes over time and the change is upper-bounded by $K\eta^t L$. As a result, we can plug such an upper bound in the regret bound analysis of Bayesian optimization [137] to understand how K, η, and γ affect the optimality of Bayesian optimization. Specifically, we upper bound $L_{\mathcal{S}}(\boldsymbol{\theta}^t)$ with $L_{\mathcal{S}}(\boldsymbol{\theta}^{t+K}) + K\eta^t L$ and use it in Lemma 2 and Lemma 3 from [137] in Appendix B.1. With such a technique, a regret bound will have the following overhead in addition to the original regret bound in equation (14) of [137]:

$$\frac{2\eta^0}{1-\gamma} KL\mathbb{E}[L_\lambda]K', \tag{C.3}$$

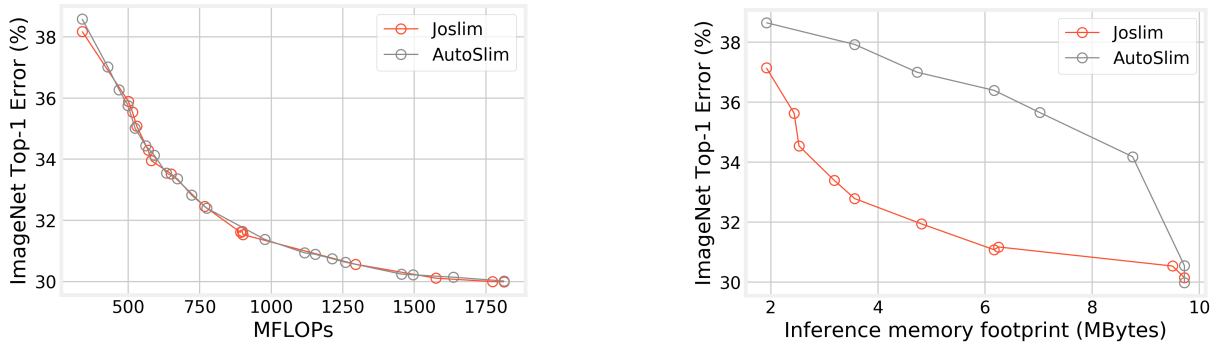
where we have utilized the geometric progression of the exponential learning rate decay and L, $\mathbb{E}[L_\lambda]$, and K' are the notations used by [137]. In other words, without a decaying learning rate, the overhead can be unbounded. This analysis reveals that larger initial learning rate η^0 and K results in a worse regret bound.

C.5 Inference Memory Footprint Calculation

To demonstrate the generality of proposed Joslim, we in addition consider optimizing for the trade-off curve between prediction error and inference memory footprint. The inference memory footprint is a critical factor when it comes to deploying deep CNNs onto resource-constrained devices such as mobile phones or micro-controllers as motivated in the original slimmable neural network paper [196]. We use a single image per batch to calculate the memory footprint. Specifically the inference memory footprint is characterized as follows:

$$\begin{aligned}
FM_{in}^l &= W_{in}^l \times H_{in}^l \times C_{in}^l \\
FM_{out}^l &= W_{out}^l \times H_{out}^l \times C_{out}^l \\
\text{Weights}^l &= K_w^l \times K_h^l \times C_{in}^l \times C_{out}^l / G^l \\
\text{Skip}^l &= W_{out}^l \times H_{out}^l \times C_{skip}^l \\
MEM &= \max_l \left(FM_{in}^l + FM_{out}^l + \text{Weights}^l + \text{Skip}^l \right),
\end{aligned} \tag{C.4}$$

where FM_{in}^l and FM_{out}^l denote the input and output feature map sizes of layer l, Weights^l denotes the size of the weights in layer l, and Skip^l denotes the memory requirement of storing the feature maps from



(a) Error *vs.* FLOPs

(b) Error *vs.* Memory

Figure C.2: Comparing Joslim and AutoSlim on ResNet18. Since ResNet18 has similar FLOPs across different layers, greedy pruning used by AutoSlim perform comparably to Joslim. However, Joslim outperforms AutoSlim when it comes to optimizing for memory consumption since the greedy pruning procedure adopted by AutoSlim is not multi-objective.

skip connections. W and H represent the width and height of the feature map. K_w and K_h denote the kernel size. Lastly, C_{in}, C_{out} and G denote the input channel, output channel, and the number of groups for convolutional layer l.

C.6 Comparisons With AutoSlim

AutoSlim [193] is a NAS method proposed to do channel search for *standalone models*. While it also provides non-uniform widths for different layers, it is not a method derived to solve for equation 6.5. Specifically, AutoSlim first perform weight-sharing training (equation 6.7) for a short amount of period, i.e., 10% to 20% of the full training epochs. Then, AutoSlim conducts greedy pruning to greedily remove a fixed amount of channels from the layer that affects the loss the least. Such a greedy procedure naturally results in a sequence of models of different computational requirements. Crucially, this greedy pruning procedure is not solving equation 6.12 since the computational requirement (FLOPs or memory footprint) does not affect the ranking among filters to be pruned. Nonetheless, we can adopt AutoSlim to obtain a sequence of models and train them via weight-sharing to form a slimmable network. We compared with AutoSlim using ResNet18 on ImageNet with both FLOPs and memory footprint. As shown in Fig. C.2, Joslim performs similarly with AutoSlim when it comes to FLOPs, and this is because ResNet18 has balanced FLOPs for all the layers. On the other hand, when it comes to memory footprint, AutoSlim performs much worse compared to Joslim. In hindsight, this result is not surprising as Joslim is designed to solve equation 6.5 while AutoSlim is not.