# Search Algorithms and Search Spaces for Neural Architecture Search

Xiaofang Wang

CMU-RI-TR-22-12

April 21, 2022

The Robotics Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

**Thesis Committee:**
Kris M. Kitani (Chair)
Deva Ramanan
Jeff Schneider
Michael S. Ryoo, Stony Brook University & Google

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Robotics*

©Xiaofang Wang, 2022

## Abstract

Neural architecture search (NAS) is recently proposed to automate the process of designing network architectures. Instead of manually designing network architectures, NAS automatically finds the optimal architecture in a data-driven way. Despite its impressive progress, NAS is still far from being widely adopted as a common paradigm for architecture design in practice. This thesis aims to develop principled NAS methods that can automate the design of neural networks and reduce human efforts in architecture tuning as much as possible. To achieve this goal, we focus on developing better search algorithms and search spaces, both of which are important for the performance of NAS.

For search algorithms, we first present an efficient NAS framework based on Bayesian optimization (BO). Specifically, we propose a method to learn an embedding space over the domain of network architectures, which makes it possible to define a kernel function for the architecture domain, a necessary component to applying BO to NAS. Then, we propose a neighborhood-aware NAS formulation to improve the generalization of architectures found by NAS. The proposed formulation is general enough to be applied to various search algorithms, including both sampling-based algorithms and gradient-based algorithms.

For search spaces, we first extend NAS beyond discovering convolutional cells to attention cells. We propose a search space for spatiotemporal attention cells that use attention operations as the primary building block. Our discovered attention cells not only outperform manually designed ones, but also demonstrate strong generalization across different modalities, backbones, or datasets. Then, we show that committee-based models (ensembles or cascades) are an overlooked design space for efficient models. We find that simply building committees from off-the-shelf pre-trained models can match or exceed the accuracy of state-of-the-art models while being drastically more efficient. Finally, we point out the importance of controlling the cost in the comparison of different LiDAR-based 3D object detectors. We show that, SECOND, a simple baseline which is generally believed to have been significantly surpassed, can almost match the performance of the state-of-the-art method on the Waymo Open Dataset, if allowed to use a similar latency.

## Acknowledgements

First and foremost, I would like to thank my advisor Kris Kitani, for his tremendous support and help during my time at CMU. I have worked with Kris for seven years since I joined CMU as a master student and I still remember the first time we met each other during a party in RoboLounge. I thank Kris not only for his insights, wisdom, and guidance that have deeply influenced and shaped my research style and cultivated me from am innocent student to a more mature researcher, but also for his patience, perseverance, and kindness that have taught me how to better deal with difficulties and eventually made me a better person. I cannot imagine a better journey at CMU without having Kris as my advisor.

I am grateful to the rest of my Ph.D. thesis committee: Deva Ramanan, Jeff Schneider, and Michael Ryoo. Thank you for sparing the time to serve on my Ph.D. thesis committee and providing critical feedback and insightful comments that have made my thesis stronger. I am also thankful to Martial Hebert for being my master co-advisor, and thankful to Abhinav Gupta and Xinlei Chen for serving on my master thesis committee. Thank you all for the help on my master research.

I am fortunate to have worked with many great mentors, collaborators, and colleges during my two internships at Google. The two projects I did at Google have become an integral part of my Ph.D. thesis (Chapter 4&5). Thank you to Xuehan Xiong, Maxim Neumann, AJ Piergiovanni, Michael Ryoo, Anelia Angelova, Wei Hua, Guanhang Wu, Yinxiao Li, and the larger Google Cloud Video AI team for the support during my internship in 2019; and to Yair Alon (prev. Movshovitz-Attias), Elad Eban, Dan Kondratyuk, Eric Christiansenm, and the larger CoreML team in Google Perception for the support during my internship in 2020.

I have lucky to have met many great labmates and schoolmates at CMU, including the KLab members: Wei-Chiu Ma, Minghuang Ma, Namhoon Lee, Nicholas Rhinehart, Haoqi Fan, Yu Zhang, Syed Zahir Bokhari, Xinshuo Weng, Yifan Xing, Shangxuan Wu, Donghyun Yoo, Xiang Xu, Ye Yuan, Yan Xu, Navyata Sanghvi, Ben Newman, Alireza Golestaneh, Mohit Sharma, Arjun Sharma, Tanmay Shankar, Rawal Khirodkar, Tanya Marwah, Aashi Manglik, Scott Sun, Sandy Sun, Wen-Hsuan Chu, Harsh Agarwal, Shengcao Cao, Yunze Man, Dennis Melamed, Zhengyi Luo, Jack Li, Mariko Isogawa, Shinnosuke Usami, Xingyu Liu, Sean Crane, Qichen Fu, Yuda Song, Vivek Roy, Shun Iwase, Jinhyung Park, Jinkun Cao, Jenny Nan, Erica Weng; and friends at CMU: Chenyang Li, Yi Shi, Anqi Li, Shen Li, Zhe Cao, Yifan Hou, Rui Zhu, Yilin Yang, Yi Hua, Mengxin Li, Yujia Huang, Yuxin Wu, Tanmay Batra, Lin Ma, Jiaqi Liu, Yiming Wu, Tiancheng Zhi, Junwei Liang, Mengtian Li, Xian Zhou, Xingyu Lin, Allan Wang, Chen-Hsuan Lin, Minyoung Huh, Leonid Keselman, Senthil Purushwalkam, Aayush Bansal, Peiyun Hu, Rohit Girdhar, Chaoyang Wang, Wei Dong, Sibi Venkatesan, Wenhao Luo, Chao Cao, Chia Dai, Anqi Yang, Yufei Ye, Donglai Xiang, Gengshan Yang, Yi Sha; and many many others. Thank you all for all the chat, discussions, food, trips, and fun activities we have had together.

They have made my life at CMU so much more enjoyable.

Thank you to Suzanne Lyons Muth, Barbara (B.J.) Fecich, Nora Kazour, Lynnetta J. Miller, and Jessica Butterbaugh for making things easier for me. Thank you to Alison Day and other CMU OIE staff for all the help.

I would like to thank many people who have generously offered me help during my job search, including Yair Alon, Xuehan Xiong, Haoqi Fan, Tiancheng Zhi, Junwei Liang, Yifan Xing, and Linjie Yang. Thank you for giving me referrals and sharing interview tips. Special thanks to Yair Alon for answering my countless questions during the entire process.

I dedicate this thesis to my family. I am forever indebted to my parents and grandma for their unconditional love and endless support throughout my life. I cannot imagine the person I would be and my life would be without them. I also want to thank my girlfriend Xuhui Zhang and her cat Naitang for the accompany during the last part of my Ph.D. journey and filling my life with happiness.

# Contents

**1 Introduction** **1**
1.1 Overview . . . 3
1.2 Bibliographical Remarks . . . 5

**I Search Algorithms** **6**

**2 Efficient NAS with Bayesian Optimization** **7**
2.1 Introduction . . . 7
2.2 Related Work . . . 8
2.3 Approach . . . 10
2.4 Experiments . . . 16
2.5 Conclusion . . . 28

**3 Neighborhood-Aware NAS** **29**
3.1 Introduction . . . 29
3.2 Related Work . . . 31
3.3 Neighborhood-Aware Formulation . . . 32
3.4 Neighborhood-Aware Search Algorithms . . . 38
3.5 Experiments . . . 43
3.6 Conclusion . . . 51

**II Search Spaces** **52**

**4 AttentionNAS: Spatiotemporal Attention Cell Search** **53**
4.1 Introduction . . . 53
4.2 Related Work . . . 55
4.3 Attention Cell Search Space . . . 56
4.4 Attention Cell Search Algorithm . . . 61
4.5 Experiments . . . 64
4.6 Conclusion . . . 74

**5 Wisdom of Committees: An Overlooked Approach to Faster and More Accurate Models** **75**
5.1 Introduction . . . . . 75
5.2 Related Work . . . . . 77
5.3 Ensembles are Accurate, Efficient, and Fast to Train . . . . . 78
5.4 From Ensembles to Cascades . . . . . 80
5.5 Model Selection for Building Cascades . . . . . 83
5.6 Model Pool Details and Analysis . . . . . 92
5.7 Self-cascades . . . . . 94
5.8 Applicability beyond Image Classification . . . . . 95
5.9 Conclusion . . . . . 97

**6 Cost-Aware Evaluation and Model Scaling for LiDAR-Based 3D Object Detection** **98**
6.1 Introduction . . . . . 98
6.2 Related Work . . . . . 100
6.3 Architecture Overview . . . . . 102
6.4 Experimental Setup . . . . . 105
6.5 Scaling Depth, Width, and Pre-Head Resolution . . . . . 105
6.6 Comparing Scaled SECOND Against Recent Methods . . . . . 108
6.7 Conclusion . . . . . 113

**7 Conclusion & Discussion** **114**
7.1 Future Work . . . . . 114
7.2 Concluding Summary . . . . . 116
7.3 Publication List . . . . . 118

# List of Figures


2.1 Visualization of architectures compressed from VGG-19. . . . . . . . 19
2.2 Visualization of architectures compressed from ResNet-18. . . . . . . 20
2.3 Comparison of methods to maximize the acquisition function. . . . . 27
2.4 Comparison between our method and random search baseline. . . . 27

3.1 Loss landscape visualization of the found architecture. . . . . . . . . 30
3.2 Loss landscape visualization of the found architecture (more runs). . 49
3.3 Cell Visualization. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

4.1 Illustration of the search space. . . . . . . . . . . . . . . . . . . . . . 55
4.2 Illustration of the supergraph used by the differentiable method. . . 62
4.3 Visualization of the position-agnostic cell. . . . . . . . . . . . . . . . . 71
4.4 Visualization of the position-specific cells. . . . . . . . . . . . . . . . . 71

5.1 Committee-based models achieve a higher accuracy than single models on ImageNet while using fewer FLOPs. . . . . . . . . . . . . . . . 76
5.2 Ensembles work well in the *large* computation regime and cascades show benefits in *all* computation regimes. . . . . . . . . . . . . . . . . 79
5.3 Different metrics for the confidence function. . . . . . . . . . . . . . . 82
5.4 Cascades with different confidence thresholds. . . . . . . . . . . . . . 82
5.5 Cascades of EfficientNet, ResNet, MobileNetV2 or ViT models on ImageNet. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
5.6 Impact of the number of models in cascades. . . . . . . . . . . . . . . 93

6.1 Main components in a LiDAR-based 3D object detector. . . . . . . . . 102
6.2 Backbone Architecture of SECOND. . . . . . . . . . . . . . . . . . . . 103
6.3 Scaled SECOND vs. Selected Two-Stage Detectors. . . . . . . . . . . . 110

# List of Tables


2.1 Compression Results on CIFAR-100. . . . . . . . . . . . . . . . . . . . . 17
2.2 Compression Results on CIFAR-10. . . . . . . . . . . . . . . . . . . . . . 18
2.3 Comparison with ShuffleNet on CIFAR-100. . . . . . . . . . . . . . . . 21
2.4 Comparison with TPE on CIFAR-100. . . . . . . . . . . . . . . . . . . . 21
2.5 Summary of Kernel Transfer Results. . . . . . . . . . . . . . . . . . . . 22
2.6 Comparison of different objective functions . . . . . . . . . . . . . . . 24
2.7 Ablation study of the number of kernels $K$. . . . . . . . . . . . . . . . 25
2.8 Ablation study of adding skip connections. . . . . . . . . . . . . . . . 26
2.9 Ablation study of the maximization of the acquisition function. . . . 26

3.1 Average error of flat-minima architectures and sharp-minima architectures. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.2 Additional results for average error of flat-minima architectures and sharp-minima architectures. . . . . . . . . . . . . . . . . . . . . . . . . 36
3.3 Kendall's Tau obtained by $f(\alpha)$ (baseline) and $g\left(f(\mathcal{N}(\alpha))\right)$ (ours). . 36
3.4 Average neighborhood variance and test error of architectures found by $f(\alpha)$ (baseline) and $g\left(f(\mathcal{N}(\alpha))\right)$ (ours). . . . . . . . . . . . . . . 37
3.5 Test error of NA-RS and the standard random search (RS). . . . . . . 43
3.6 Ablation study on the aggregation function in NA-RS. . . . . . . . . . 44
3.7 Ablation study on $n_{\text{nbr}}$ in NA-RS. . . . . . . . . . . . . . . . . . . . . 44
3.8 Test error of NA-DARTS and DARTS. . . . . . . . . . . . . . . . . . . . 45
3.9 Comparison with state-of-the-art NAS methods on CIFAR-10&100 . . 45
3.10 Comparison with state-of-the-art NAS methods on ImageNet. . . . . 46
3.11 Ablation study of NA-DARTS. . . . . . . . . . . . . . . . . . . . . . . . 47
3.12 Test error of architectures found from the S3 search space. . . . . . . 48

4.1 Search results on Kinetics-600 and MiT using GPB. . . . . . . . . . . . 65
4.2 Search results on Kinetics-600 and MiT using differentiable search. . 65
4.3 Generalization across different modalities. . . . . . . . . . . . . . . . . 67
4.4 Generalization across different backbones. . . . . . . . . . . . . . . . . 67
4.5 Generalization across different datasets. . . . . . . . . . . . . . . . . . 67
4.6 Comparison with the state-of-the-art methods. . . . . . . . . . . . . . 68
4.7 Comparison between different supergraph designs. . . . . . . . . . . 69

4.8 Inference FLOPs on Kinetics-600. . . . . . . . . . . . . . . . . . . . . . . 74

5.1 Training time (TPUv3 days) of EfficientNet. . . . . . . . . . . . . . . . . 80
5.2 Training time (TPU v3 days) of ensembles. . . . . . . . . . . . . . . . . 80
5.3 Cascades of EfficientNet, ResNet or MobileNetV2 models on ImageNet. 86
5.4 Cascades of ViT models on ImageNet. . . . . . . . . . . . . . . . . . . . 86
5.5 Average latency on TPUv3 for the case of online processing with batch size 1. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.6 Throughput on TPUv3 for the case of offline processing. . . . . . . . 88
5.7 Comparison with SOTA NAS methods. . . . . . . . . . . . . . . . . . 88
5.8 Cascades can be built with a guarantee on worst-case FLOPs. . . . . . 89
5.9 Exit ratios of cascades. . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.10 A Family of Cascades C0 to C7. . . . . . . . . . . . . . . . . . . . . . 91
5.11 Max, min, mean, and standard deviation of the performance of single models, ensembles, and cascades. . . . . . . . . . . . . . . . . . . . . . 94
5.12 Cascades of models of same architectures vs. Cascades of models of different architectures. . . . . . . . . . . . . . . . . . . . . . . . . . . 94
5.13 Self-cascades. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
5.14 Cascades of X3D models on Kinetics-600. . . . . . . . . . . . . . . . . 96
5.15 Cascades of DeepLabv3 models on Cityscapes. . . . . . . . . . . . . . 96

6.1 Component Overview of Relevant Detectors. . . . . . . . . . . . . . . 102
6.2 List of architectures used in our analysis. . . . . . . . . . . . . . . . . 103
6.3 Performance of SECOND-Anchor with different backbones. . . . . . 106
6.4 Performance of SECOND-Center with different backbones. . . . . . . 106
6.5 Depth vs. Width vs. Pre-Head Resolution. . . . . . . . . . . . . . . . . 108
6.6 Scaled SECOND vs. Recent 3D Object Detection Methods. . . . . . . 111
6.7 Overfitting of SECOND-Center on Cyclist. . . . . . . . . . . . . . . . . 113

# Chapter 1

# Introduction

Neural networks are almost ubiquitous in modern learning-based methods for computer vision. Equipped with large-scale datasets and appropriate training techniques, neural networks can approximate sophisticated high-dimensional functions and learn about the non-trivial relationships between input and expected output, *e.g.*, images and object labels or segmentation masks. One major contributing factor to the power of neural networks is the novel design of network architectures.

Traditionally, network architectures are manually designed by humans through trial and error based on their knowledge and experience, which can be tedious and is sometimes regarded as *more of an art than a science*. Towards a more principled approach for designing network architectures, neural architecture search (NAS) is recently proposed to automate the process of network architecture design.

Instead of manually designing network architectures, NAS automatically finds the optimal architecture in a data-driven way, where the design process of architectures is formulated as an optimization problem. For a specific task, NAS first defines a *search space* containing all the possible architectures to be explored, which represents the domain of the optimization problem. Then a *search algorithm* is applied to solve the optimization problem, *i.e.*, finding the architecture with the highest performance on the given task in the search space.

While NAS has demonstrated impressive progress and discovered architectures outperforming human-designed ones for a wide range of tasks [46,119,189,190], it is still far from being widely adopted as a common paradigm for architecture design in practice. We highlight the following two challenges that need to be addressed to make NAS more widely used.

- **Develop efficient and reliable search algorithms.** State-of-the-art NAS methods usually require significant computational resources. For example, it took 2,000 GPU days to discover NASNet [190], and as estimated by Strubell et al. [135], it took ∼1,300 TPUv2 days to find the Evolved Transformer [131]. Such computational cost is prohibitive and also raises concerns on the huge carbon footprint of NAS [135].

To speed up the search, weight sharing across architectures is proposed to amortize the training cost of candidate architectures [8,107]. Weight-sharing NAS, *a.k.a.*, one-shot NAS, can reduce the search time by orders of magnitude, *e.g.*, DARTS [91] only needs 0.4 GPU days to find a high-performing architecture. However, weight-sharing methods still suffer from several weaknesses, such as being prone to overfitting [181], failing when the search space changes [181], and yielding unstable search performance among multiple runs [164].

Therefore, we need to develope efficient and reliable search algorithms that can find satisfying architectures with an affordable computational cost, produce stable search results, and work robustly for new search spaces and tasks.

- **Design flexible and generic search spaces.** Currently, the majority of NAS research focuses on image classification, where the most widely used search spaces are the cell-based search space [89,91,190] and the MobileNet-based search space [16, 139, 162]. While these spaces contain a diverse range of high-performing architectures, recent work show that they are still not flexible enough. For example, the cell-based search space is shown to have a narrow performance range [171] and even random search can find competitive architectures [171,179]. Architectures in the MobileNet-based space resemble MobileNetV2 [120] by design, therefore the lower bound of accuracy is still high [164]. As pointed by Yang et al. [171], although such constraints of the search space design can avoid the exploration of bad architectures and guarantee good performance, they also limit the possibility for NAS to discover truly innovative solutions. Therefore, we need more flexible search spaces for NAS to find significantly better architectures.

  Other than image classification, NAS has also been successfully applied to many other tasks, *e.g.*, object detection [46], semantic segmentation [23,88], and video classification [119,153]. When applying NAS to a new task, it is common to propose a new search space customized for this task. Customizing the search space allows us to maximally leverage human insight on the task and data, but also introduces additional complexity in practice when applying NAS to new tasks or data. This calls for generic search spaces that can work for a wide range of tasks and real-world data without heavy modification [164].

The ultimate goal of this thesis is to develop principled NAS methods that can automate the design of neural networks and reduce human efforts in architecture tuning as much as possible. Both the search algorithm and search space are important for the performance of NAS, so we focus on both of them.

## 1.1 Overview

Below is an overview of the thesis.

### Part I: Search Algorithms

#### Chapter 2: Efficient NAS with Bayesian Optimization

One significant bottleneck of NAS is the need to constantly evaluate different network architectures, as each evaluation is extremely costly (*e.g.*, backpropagation to learn the parameters of a single deep network can take days on a single GPU). This means that any efficient search algorithm must be judicious when selecting architectures to evaluate. We explore the usage of Bayesian optimization (BO) for NAS. BO has been proven to be a highly useful algorithm for solving optimization problems where the objective function is a black box and expensive to evaluate, *e.g.*, hyperparameter optimization.

We propose a method to learn an embedding space over the domain of network architectures, which makes it possible to define a kernel function for the architecture domain, a necessary component to applying BO to NAS. Based on the proposed embedding space, we present an efficient architecture search framework using BO. Extensive experiments demonstrate that our algorithm can significantly outperform other search algorithms, *e.g.*, random search and reinforcement learning [3], and manually designed compact architecture ShuffleNet [183]. We describe further details of this work in Chapter 2.

#### Chapter 3: Neighborhood-Aware NAS

NAS is typically formulated as an optimization problem that tries to find the architecture with the highest performance in the search space. However, directly optimizing architecture performance may cause the search algorithm to overfit to the search setting, *i.e.*, selecting an architecture performing well during search but generalizing poorly under the test setting, due to the inevitable differences between the search setting and test setting, such as the number of training epochs, the usage of weight sharing, or proxy datasets during search.

Motivated by the understanding in neural network training that flat minima generalize better than sharp minima [57], we propose a neighborhood-aware NAS formulation to encourage the selection of flat minima architectures in the search space, *i.e.*, architectures with strong generalization capability. The proposed formulation is general enough to be applied to various search algorithms, such as random search, reinforcement learning, and differentiable NAS methods [91]. By simply augmenting DARTS [91] with our formulation, we find architectures that perform better or on par with those found by state-of-the-art NAS methods on established benchmarks. Further details of this work are included in Chapter 3.

## Part II: Search Spaces

### Chapter 4: AttentionNAS: Spatiotemporal Attention Cell Search

Most work in computer vision [88,89,91,189,190] uses the convolutional operation as the primary building block to construct the network. While being successful, convolutional operations still have their limitations and NAS should not be limited to only convolutional operations. It has been shown that attention is complementary to convolutional operations, and they can be combined to further improve performance on vision tasks [6, 154, 161]. However, many design choices still remain to be determined to use attention, especially when applying attention to videos.

We propose a search space for spatiotemporal attention cells that use attention operations as the primary building block. We also develop a differentiable formulation of the search space, allowing us to efficiently search for the optimal attention cell design. The discovered attention cell can be seamlessly inserted into a wide range of backbone networks, *e.g.*, I3D [19] or S3D [167], to improve the performance on video understanding tasks. The discovered attention cells not only outperform manually designed ones, but also demonstrate strong generalization across different modalities, backbones, or datasets. We include more details of this work in Chapter 4.

### Chapter 5: Wisdom of Committees: An Overlooked Approach to Faster and More Accurate Models

Committee-based models (ensembles or cascades) construct models by combining existing pre-trained ones. While ensembles and cascades are well-known techniques that were proposed before deep learning, they are not considered a core building block of deep model architectures and are rarely compared to in recent literature on developing efficient models. In this work, we go back to basics and conduct a comprehensive analysis of the efficiency of committee-based models.

We find that even the most simplistic method for building committees from existing, independently pre-trained models can match or exceed the accuracy of state-of-the-art models while being drastically more efficient. These simple committee-based models also outperform sophisticated neural architecture search methods. These findings hold true for several tasks, including image classification, video classification, and semantic segmentation, and various architecture families, such as ViT [37], EfficientNet [140], ResNet [54], MobileNetV2 [120], and X3D [41]. Our results show that an EfficientNet cascade can achieve a 5.4x speedup over B7 and a ViT cascade can achieve a 2.3x speedup over ViT-L-384 while being equally accurate. We provide more details of this work in Chapter 5.

**Chapter 6: Cost-Aware Evaluation and Model Scaling for LiDAR-Based 3D Object Detection**

Considerable research efforts have been devoted to LiDAR-based 3D object detection and its empirical performance has been significantly improved. While progress has been encouraging, we notice an issue about the current state of 3D detection research: it is not yet common practice to compare different detectors under the same cost, *e.g.*, inference latency. This can cause unfair comparison and makes it is difficult to quantify the true performance gain brought by recently proposed architecture designs.

To address this issue, we take a step forward by analyzing the performance of a simple grid-based one-stage detector, *i.e.*, SECOND [170], under different costs by scaling its original architecture. We compare the family of scaled SECOND against recent 3D detection methods. The results are surprising. We find that, SECOND can easily outperform a number of recent methods after being scaled up. Notably, scaled SECOND can nearly match the performance of PV-RCNN++, the current state-of-the-art on the Waymo Open Dataset, if allowed to use a similar inference latency. Our results indicate that the gain brought by the architectural innovation in some recent methods is not as significant as what was shown in their papers. We recommend future research control the inference cost in their empirical comparison and include the family of scaled SECOND as baselines when presenting novel 3D object detectors. More details of this work are included in Chapter 6.

## 1.2 Bibliographical Remarks

Chapter 2 is based on joint work with Shengcao Cao and Kris Kitani [17]. Chapter 3 is based on joint work with Shengcao Cao, Mengtian Li, and Kris Kitani [150]. Chapter 4 is based on joint work with Xuehan Xiong, Maxim Neumann, AJ Piergiovanni, Michael Ryoo, Anelia Angelova, Kris Kitani, and Wei Hua [153]. Chapter 5 is based on joint work with Dan Kondratyuk, Eric Christiansenm, Kris Kitani, Yair Alon (prev. Movshovitz-Attias), and Elad Eban [152]. Chapter 6 is based on joint work with Kris Kitani [151].

# Part I

# Search Algorithms

# Chapter 2

# Efficient NAS with Bayesian Optimization

## 2.1 Introduction

In many application domains, it is common practice to make use of well-known deep network architectures (*e.g.,* VGG [129], GoogleNet [138], ResNet [54]) and to adapt them to a new task without optimizing the architecture for that task. While this process of transfer learning is surprisingly successful, it often results in over-sized networks which have many redundant or unused parameters. Inefficient network architectures can waste computational resources and over-sized networks can prevent them from being used on embedded systems. There is a pressing need to develop algorithms that can take large networks with high accuracy as input and compress their size while maintaining similar performance. In this paper, we focus on the task of compressed architecture search – the automatic discovery of compressed network architectures based on a given large network.

One significant bottleneck of compressed architecture search is the need to repeatedly evaluate different compressed network architectures, as each evaluation is extremely costly (*e.g.*, back-propagation to learn the parameters of a single deep network can take several days on a single GPU). This means that any efficient search algorithm must be judicious when selecting architectures to evaluate. Learning a good embedding space over the domain of compressed network architectures is important because it can be used to define a distribution on the architecture space that can be used to generate a priority ordering of architectures for evaluation. To enable the careful selection of architectures for evaluation, we propose a method to incrementally learn an embedding space over the domain of network architectures.

In the network compression paradigm, we are given a teacher network and we aim to search for a compressed network architecture, a student network that contains as few parameters as possible while maintaining similar performance to the teacher network. We address the task of compressed architecture search by using

Bayesian Optimization (BO) with a kernel function defined over our proposed embedding space to select architectures for evaluation. As modern neural architectures can have multiple layers, multiple branches and multiple skip connections, defining an embedding space over all architectures is non-trivial. In this work, we propose a method for mapping a diverse range of discrete architectures to a continuous embedding space through the use of recurrent neural networks. The learned embedding space allows us to perform BO to efficiently search for compressed student architectures that are also expected to have high accuracy.

We demonstrate that our search algorithm can significantly outperform various baseline methods, such as random search and reinforcement learning [3]. For example, our search algorithm can compress VGG-19 [129] by $8\times$ on CIFAR-100 [72] while maintaining accuracy on par with the teacher network. The automatically found compressed architectures can also achieve higher accuracy than the state-of-the-art manually-designed compact architecture ShuffleNet [183] with a similar size. We also demonstrate that the learned embedding space can be transferred to new settings for architecture search, such as a larger teacher network or a teacher network in a different architecture family, without any training.

**Contributions.** (1) We propose a novel method to incrementally learn an embedding space over the domain of network architectures. Based on the learnable embedding space, we present a framework of searching for compressed network architectures with BO. The learned embedding provides a feature space over which the kernel function of BO is defined. (2) We propose a set of architecture operators for generating architectures for search. Operators for modifying the teacher network are: layer removal, layer shrinkage and skip connection addition. (3) We propose a multiple kernel strategy to prevent the premature convergence of the search and encourage the search algorithm to explore more diverse architectures during the search process.

## 2.2 Related Work

**Computationally Efficient Architecture.** There has been great progress in designing computationally efficient network architectures. Representative examples include SqueezeNet [67], MobileNet [59], MobileNetV2 [120], CondenseNet [63] and ShuffleNet [183]. Different from them, we aim to develop an algorithm that can automatically search for an efficient network architecture with minimal human efforts involved in the architecture design.

**Neural Architecture Search (NAS).** NAS has recently been an active research topic [89–91, 96, 107, 115, 189, 190]. Some existing works in NAS are focused on searching for architectures that not only can achieve high performance but also respect some resource or computation constraints [3, 35, 38, 60, 139, 186]. NAO [96]

and our work share the idea of mapping network architectures into a latent continuous embedding space. But NAO and our work have fundamentally different motivations, which further lead to different architecture search frameworks. NAO maps network architectures to a continuous space such that they can perform gradient based optimization to find better architectures. However, our motivation for learning the embedding space is to find a principled way to define a kernel function between architectures with complex skip connections and multiple branches.

Our work is also closely related to N2N [3], which searches for a compressed architecture based on a given teacher network using reinforcement learning. Our search algorithm is developed based on Bayesian Optimization (BO), which is different from N2N and many other existing works. We will compare our approach to other BO based NAS methods in the next paragraph. Readers can refer to [39] for a more complete literature review of NAS.

**Bayesian Optimization (BO).** BO is a popular method for hyper-parameter optimization in machine learning. BO has been used to tune the number of layers and the size of hidden layers [10, 137], the width of a network [130] or the size of the filter bank [9], along with other hyper-parameters, such as the learning rate, number of iterations. [99], [68] and [182] also fall into this category. Our work is also related to [55], which presents a Bayesian method for identifying the Pareto set of multi-objective optimization problems and applies the method to searching for a fast and accurate neural network.

However, most existing works on BO for NAS only show results on tuning network architectures where the connections between network layers are fixed, i.e., most of them do not optimize how the layers are connected to each other. [69] proposes a distance metric OTMANN to compare network architectures with complex skip connections and branch layers, based on which NASBOT is developed, a BO based NAS framework, which can tune how the layers are connected. Although the OTMANN distance is designed with thoughtful choices, it is defined based on some empirically identified factors that can influence the performance of a network, rather than the actual performance of networks. Different from OTMANN, the distance metric (or the embedding) for network architectures in our algorithm is automatically learned according to the actual performance of network architectures instead of manually designed.

Our work can also be viewed as carrying out optimization in the latent space of a high dimensional and structured space, which shares a similar idea with previous literature [47, 95]. For example, [95] presents a new variational auto-encoder to map kernel combinations produced by a context-free grammar into a continuous latent space.

**Deep Kernel Learning.** Our work is also related to recent work on deep kernel learning [159, 160]. They aim to learn more expressive kernels by representing the

kernel function as a neural network to incorporate the expressive power of deep networks. The follow-up work [2] extends the kernel representation to recurrent networks to model sequential data. Our work shares a similar motivation with them and tries to learn a kernel function for the neural architecture domain by leveraging the expressive power of deep networks.

## 2.3 Approach

In this work, we focus on searching for a compressed network architecture based on a given teacher network and our goal is to find a network architecture which contains as few parameters as possible but still can obtain a similar performance to the teacher network. Formally, we aim to solve the following optimization problem:

$$x^* = \arg\max_{x \in \mathcal{X}} f(x), \tag{2.1}$$

where $\mathcal{X}$ denotes the domain of neural architectures and the function $f(x) : \mathcal{X} \mapsto \mathbb{R}$ evaluates how well the architecture $x$ meets our requirement.

We aim to find a network architecture which contains as few parameters as possible but still can obtain a similar performance to the teacher network. Usually compressing a network leads to the decrease in the performance. So the function $f$ needs to provide a balance between the compression ratio and the performance. In particular, we hope the function $f$ favors architectures of high performance but low compression ratio more than architectures of low performance but high compression ratio. So we adopt the reward function design in N2N [3] for the function $f$. Formally, $f$ is defined as:

$$f(x) = C(x)\,(2 - C(x)) \cdot \frac{A(x)}{A(x_{\text{teacher}})}, \tag{2.2}$$

where $C(x)$ is the compression ratio of the architecture $x$, $A(x)$ is the validation performance of $x$ and $A(x_{\text{teacher}})$ is the validation performance of the teacher network. The compression ratio $C(x)$ is defined as $C(x) = 1 - \frac{\#\text{params}(x)}{\#\text{params}(x_{\text{teacher}})}$.

Note that for any $x$, to evaluate $f(x)$ we need to train the architecture $x$ on the training data and test on the validation data. Therefore, evaluating $f(x)$ for a specific architecture $x$ is extremely costly. This requires that the algorithm must judiciously select the architectures to evaluate. To enable the careful selection of architectures for evaluation, we propose a method to incrementally learn an embedding space over the domain of network architecture that can be used to generate a priority ordering of architectures for evaluation. In particular, we develop the search algorithm based on BO with a kernel function defined over our proposed embedding space. In the following text, we will first introduce the sketch of the BO algorithm and then explain how the proposed embedding space is used in the loop of BO.

We adopt the Gaussian process (GP) based BO algorithms to maximize the function $f(x)$, which is one of the most popular algorithms in BO. A GP prior is placed on the function $f$, parameterized by a mean function $\mu(\cdot) : \mathcal{X} \mapsto \mathcal{R}$ and a covariance function or kernel $k(\cdot, \cdot) : \mathcal{X} \times \mathcal{X} \mapsto \mathcal{R}$. To search for the solution, we start from an arbitrarily selected architecture $x_1$. At step $t$, we evaluate the architecture $x_t$, *i.e.*, obtaining the value of $f(x_t)$. Using the $t$ evaluated architectures up to now, we compute the posterior distribution on the function $f$:

$$p\left(f(x) \mid f(x_{1:t})\right) \sim \mathcal{N}(\mu_t(x), \sigma_t^2(x)), \tag{2.3}$$

where $f(x_{1:t}) = [f(x_1), \ldots, f(x_t)]$ and $\mu_t(x)$ and $\sigma_t^2(x)$ can be computed analytically based on the GP prior [158]. We then use the posterior distribution to decide the next architecture to evaluate. In particular, we obtain $x_{t+1}$ by maximizing the expected improvement acquisition function $\mathrm{EI}_t(x) : \mathcal{X} \mapsto \mathbb{R}$, *i.e.*, $x_{t+1} = \arg\max_{x \in \mathcal{X}} \mathrm{EI}_t(x)$. The expected improvement function $\mathrm{EI}_t(x)$ [101] measures the expected improvement over the current maximum value according to the posterior distribution:

$$\mathrm{EI}_t(x) = \mathbb{E}_t[\max(0, f(x) - f_t^*)], \tag{2.4}$$

where $\mathbb{E}_t$ indicates the expectation is taken with respect to the posterior distribution at step $t$ $p\left(f(x) \mid f(x_{1:t})\right)$ and $f_t^*$ is the maximum value among $\{f(x_1), \ldots, f(x_t)\}$. Once obtaining $x_{t+1}$, we repeat the above described process until we reach the maximum number of steps. Finally, we return the best evaluated architecture as the solution.

The main difficulty in realizing the above optimization procedure is the design of the kernel function $k(\cdot, \cdot)$ for $\mathcal{X}$ and the maximization of the acquisition function $\mathrm{EI}_t(x)$ over $\mathcal{X}$, since the neural architecture domain $\mathcal{X}$ is discrete and highly complex. To overcome these difficulties, we propose to learn an embedding space for the neural architecture domain and define the kernel function based on the learned embedding space. We also propose a search space, a subset of the neural architecture domain, over which maximizing the acquisition function is feasible and sufficient.

### 2.3.1 Learnable Embedding Space and Kernel Function

The kernel function, which measures the similarity between network architectures, is fundamental for selecting the architectures to evaluate during the search process. As modern neural architectures can have multiple layers, multiple branches and multiple skip connections, comparing two architectures is non-trivial. Therefore, we propose to map a diverse range of discrete architectures to a continuous embedding space through the use of recurrent neural networks and then define the kernel function based on the learned embedding space.

We use $h(\cdot; \theta)$ to denote the architecture embedding function that generates an embedding for a network architecture according to its configuration parameters. $\theta$

represents the weights to be learned in the architecture embedding function. With $h(\cdot;\theta)$, we define the kernel function $k(x, x';\theta)$ based on the RBF kernel:

$$k(x, x';\theta) = \exp\left(-\frac{||h(x;\theta) - h(x';\theta)||^2}{2\sigma^2}\right), \tag{2.5}$$

where $\sigma$ is a hyper-parameter. $h(\cdot;\theta)$ represents the proposed learnable embedding space and $k(x, x';\theta)$ is the learnable kernel function. They are parameterized by the same weights $\theta$. In the following text, we will first introduce the architecture embedding function $h(\cdot;\theta)$ and then describe how we learn the weights $\theta$ during the search process.

The architecture embedding function needs to be flexible enough to handle a diverse range of architectures that may have multiple layers, multiple branches and multiple skip connections. Therefore we adopt a Bidirectional LSTM to represent the architecture embedding function, motivated by the layer removal policy network in N2N [3]. The input to the Bi-LSTM is the configuration information of each layer in the network, including the layer type, how this layer connects to other layers, and other attributes. After passing the configuration of each layer to the Bi-LSTM, we gather all the hidden states, apply average pooling to these hidden states and then apply $L_2$ normalization to the pooled vector to obtain the architecture embedding.

We would like to emphasize that our representation for layer configuration encodes the skip connections between layers. Skip connections have been proven effective in both human designed network architectures, such as ResNet [54] and DenseNet [64], and automatically found network architectures [189]. N2N only supports the kind of skip connections used in ResNet [54] and does not generalize to more complex connections between layers, where our representation is still applicable. We give the details about our representation for layer configuration in Section 2.3.1.1.

The weights of the Bi-LSTM, *i.e.*, $\theta$, are learned during the search process. The weights $\theta$ determine the architecture embedding function $h(\cdot;\theta)$ and the kernel $k(\cdot,\cdot;\theta)$. Further, $\theta$ controls the GP prior and the posterior distribution of the function value conditioned on the observed data points. The posterior distribution guides the search process and is essential to the performance of our search algorithm. Our goal is to learn a $\theta$ such that the function $f$ is consistent with the GP prior, which will result in a posterior distribution that accurately characterizes the statistical structure of the function $f$.

Let $D$ denote the set of evaluated architectures. In step $t$, $D = \{x_1, \ldots, x_t\}$. For any architecture $x_i$ in $D$, we can compute $p\left(f(x_i) \mid f(D \setminus x_i);\theta\right)$ based on the GP prior, where $\setminus$ refers to the set difference operation, $f(x_i)$ is the value obtained by evaluating the architecture $x_i$ and $f(D\setminus x_i) = [f(x_1), \ldots, f(x_{i-1}), f(x_{i+1}) \ldots, f(x_t)]$. $p\left(f(x_i) \mid f(D \setminus x_i);\theta\right)$ is the posterior probability of $f(x_i)$ conditioned on the other evaluated architectures in $D$. The higher the value of $p\left(f(x_i) \mid f(D \setminus x_i);\theta\right)$ is, the

more accurately the posterior distribution characterizes the statistical structure of the function $f$ and the more the function $f$ is consistent with the GP prior. Therefore, we learn $\theta$ by minimizing the negative log posterior probability:

$$L(\theta) = -\frac{1}{|D|} \sum_{i: x_i \in D} \log p\left(f(x_i) \mid f(D \setminus x_i); \theta\right). \tag{2.6}$$

$p\left(f(x_i) \mid f(D \setminus x_i); \theta\right)$ is a Gaussian distribution and its mean and covariance matrix can be computed analytically based on $k(\cdot, \cdot; \theta)$. Thus $L$ is differentiable with respect to $\theta$ and we can learn the weights $\theta$ using backpropagation.

#### 2.3.1.1 Representation for Layer Configuration

We represent the configuration of one layer by a vector of length $(m+2n+6)$, where $m$ is the number of types of layers we consider and $n$ is the maximum number of layers in the network. The first $m$ dimensions of the vector are a one-hot vector, indicating the type of the layer. Then the following 6 numbers indicate the value of different attributes of the layer, including the kernel size, stride, padding, group, input channels and output channels of the layer. If one layer does not have any specific attribute, the value of that attribute is simply set to zero.

The following $2n$ dimensions encode the edge information of the network, if we view the network as a directed acyclic graph with each layer as a node in the graph. In particular, the $2n$ dimensions are composed of two $n$-dim vectors, where one represents the edges incoming to the code and the other one represents the edges outgoing from the node. The nodes in the directed acyclic graph can be topologically sorted, which will give each layer an index. For an edge from node $i$ to $j$, the $(j-i)^{th}$ element in the outgoing vector of node $i$ and the incoming vector of node $j$ will be $1$. We are sure that $j$ is larger than $i$ because all the nodes are topologically sorted. With this representation, we can describe the connection information in a complex network architecture.

### 2.3.2 Acquisition Function and Search Space

In each optimization step, we obtain the next architecture to evaluate by maximizing the acquisition function $\mathrm{EI}_t(\cdot)$ over the neural architecture domain $\mathcal{X}$. On one hand, maximizing $\mathrm{EI}_t(\cdot)$ over all the network architectures in $\mathcal{X}$ is unnecessary. Since our goal is to search for a compressed architecture based on the given teacher network, we only need to consider those architectures that are smaller than the teacher network. On the other hand, maximizing $\mathrm{EI}_t(\cdot)$ over $\mathcal{X}$ is non-trivial. Gradient-based optimization algorithms cannot be directly applied to optimize $\mathrm{EI}_t(\cdot)$ as $\mathcal{X}$ is discrete. Also, exhaustive exploration of the whole domain is infeasible. This calls for a search space that covers the compressed architectures of our interest and easy to explore. Motivated by N2N [3], we propose a search space for maximizing the

acquisition function, which is constrained by the teacher network, and provides a practical method to explore the search space.

We define the search space based on the teacher network. The search space is constructed by all the architectures that can be obtained by manipulating the teacher network with the following three operations: (1) layer removal, (2) layer shrinkage and (3) adding skip connections.

**Layer removal and shrinkage.** The two operations ensure that we only consider architectures that are smaller than the given big network. Layer removal refers to removing one or more layers from the network. Layer shrinkage refers to shrinking the size of layers, in particular, the number of filters in convolutional layers, as we focus on convolutional networks in this work. Different from N2N, we do not consider shrinking the kernel size, padding or other configurable variables and we find that only shrinking the number of filters already yields satisfactory performance.

**Adding skip connections.** The operation of adding skip connections is employed to increase the network complexity. N2N [3], which uses reinforcement learning to search for compressed network architectures, does not support forming skip connections in their action space. We believe when searching for compressed architectures, adding skip connections to the compressed network is crucial for it to achieve similar performance to the teacher network and we will show ablation study results to verify this.

The way we define the search space naturally allows us to explore it by sampling the operations to manipulate the architecture of the teacher network. To optimize the acquisition function over the search space, we randomly sample architectures in the search space by randomly sampling the operations. We then evaluate $\mathrm{EI}_t(\cdot)$ over the sampled architectures and return the best one as the solution. We also have tried using evolutionary algorithm to maximize $\mathrm{EI}_t(\cdot)$ but it yields similar results with random sampling. So for the sake of simplicity, we use random sampling to maximize $\mathrm{EI}_t(\cdot)$. We attribute the good performance of random sampling to the thoughtful design of the operations to manipulate the teacher network architecture. These operations already favor the compressed architectures of our interest.

### 2.3.3 Multiple Kernel Strategy

We implement the search algorithm with the proposed learnable kernel function but notice that the highest function value among evaluated architectures stops increasing after a few steps. We conjecture this is due to that the learned kernel is overfitted to the training samples since we only evaluate hundreds of architectures in the whole search process. An overfitted kernel may bias the following sampled architectures for evaluation.

To encourage the search algorithm to explore more diverse architectures, we propose a multiple kernel strategy, motivated by the bagging algorithm, which is usually employed to avoid overfitting. In bagging, instead of training one single model on the whole dataset, multiple models are trained on different subsets of

the whole dataset. Likewise, in each step of the search process, we train multiple kernel functions on uniformly sampled subsets of $D$, the set of all the available evaluated architectures. Technically, learning multiple kernels refers to learning multiple architecture embedding spaces, *i.e.*, multiple sets of weights $\theta$. After training the kernels, each kernel is used separately to compute one posterior distribution and determine one architecture to evaluate in the next step. That is to say, if we train $K$ kernels in the current step, we will obtain $K$ architectures to evaluate in the next step. The proposed multiple kernel strategy encourages the search process to explore more diverse architectures and can help find better architectures than training one single kernel only.

When training kernels, we randomly initialize their weights and train from the scratch on subsets of evaluated architectures. We do not learn the weights of the kernel based on the weights learned in the last step, *i.e.*, fine-tuning the Bi-LSTM from the last step. The training of the Bi-LSTM is fast since we usually only evaluate hundreds of architectures during the whole search process. A formal sketch of our search algorithm in shown Algorithm 1.

**Algorithm 1** Neural Architecture Search with Bayesian Optimization

**Input**: Number of steps $T$. Number of kernels $K$. Teacher network $x_{\text{teacher}}$.
Randomly sample $K$ architectures $x_1^1, \dots, x_1^K$ from the search space defined based on $x_{\text{teacher}}$.
Initialize the set of evaluated architectures $D = \emptyset$.
**for** $t = 1, \dots, T$ **do**
  Evaluate the $K$ architectures $x_t^1, \dots, x_t^K$.
  $D = D \cup \{x_t^1, \dots, x_t^K\}$.
  **for** $k = 1, \dots, K$ **do**
    Randomly initialize the weights of kernel $k$, denoted as $\theta^k$.
    Randomly sample a subset of $D$, denoted as $D^k$.
    Learn $\theta^k$ on $D^k$ using the objective function in Eq. 2.6.
    Compute the posterior distribution conditioned on the architectures in $D_k$ with kernel $k$.
    Maximize the acquisition function and denote the solution as $x_{t+1}^k$.
  **end for**
**end for**
Return the best architecture in $D$ as the solution.

## 2.4 Experiments

### 2.4.1 Compression Experiments

We evaluate our algorithm with different teacher architectures and datasets. We use two datasets: CIFAR-10 and CIFAR-100 [72]. CIFAR-10 contains 60K images in 10 classes, with 6K images per class. CIFAR-100 also contains 60K images but in 100 classes, with 600 images per class. Both CIFAR-10 and CIFAR-100 are divided into a training set with 50K images and a test set with 10K images. We sample 5K training images as the validation set. We consider four networks as the teacher architecture: VGG-19 [129], ResNet-18, ResNet-34 [54] and ShuffleNet [183].

We consider two baselines algorithms for comparison: random search (RS) and a reinforcement learning based approach, N2N [3]. Here we use RS to directly maximize the compression objective $f(x)$. To be more specific, RS randomly samples architectures in the search space, then evaluates all of them and returns the best architecture as the optimal solution. In the following experiments, RS evaluates 160 architectures. For our proposed method, we run 20 architecture search steps, where each step generates $K = 8$ architectures for evaluation based on the the $K$ different kernels. This means our proposed method also evaluates 160 $(20 \times 8)$ architectures in total during the search process.

When evaluating an architecture during the search process, we only train it for 10 epochs to reduce computation time. We also employ the Knowledge Distillation (KD) strategy [56] for faster training as we are given a teacher network. But when we fully train the architecture $x$ to see its true performance, we fine tune it from the weights obtained by early stopping with cross entropy loss without using KD. For both RS and our method, we fully train the top 4 architectures among the 160 evaluated architectures and choose the best one as the solution.

When learning the kernel function parameters, we randomly sample from the set of the evaluated architectures with a probability of 0.5 to form the training set for one kernel. The results of N2N are from the original paper [3].

The compression results on are summarized in Table 2.1& 2.2. For a compressed network $x$, 'Ratio' refers to the compression ratio of the network $x$, which is defined as $\left(1 - \frac{\#\text{params}(x)}{\#\text{params}(x_{\text{teacher}})}\right)$. 'Times' refers to the ratio between the size of the teacher network and the size of the compressed network, *i.e.*, $\frac{\#\text{params}(x_{\text{teacher}})}{\#\text{params}(x)}$. We also show the value of $f(x)$ as an indication of how well each architecture $x$ meets our requirement in terms of both the accuracy and the compression ratio. For 'Random Search' and 'Ours', we run the experiments for three times and report the average results as well as the standard deviation.

We first apply our algorithm to compress three popular network architectures: VGG-19, ResNet-18 and ResNet-34, and use them as the teacher network. We can see that on both CIFAR-10 and CIFAR-100, our proposed method consistently finds architectures that can achieve higher value of $f(x)$ than all baselines. For VGG-19

Table 2.1: Compression Results on CIFAR-100.

| CIFAR-100 | | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|---|
| VGG-19 | Teacher | 73.71% | 20.09M | - | - | - |
| | Random Search | 68.17% | 2.83M | 0.8593 | 8.04× | 0.9046 |
| | | ±1.28% | ±1.05M | ±0.0525 | ±3.78× | ±0.0074 |
| | Ours | 71.41% | 2.61M | 0.8699 | 7.99× | **0.9518** |
| | | ±0.75% | ±0.61M | ±0.0306 | ±1.99× | ±**0.0158** |
| ResNet-18 | Teacher | 78.68% | 11.22M | - | - | - |
| | Random Search | 69.86% | 1.26M | 0.8878 | 10.10× | 0.8752 |
| | | ±1.90% | ±0.54M | ±0.0477 | ±4.33× | ±0.0137 |
| | N2N | 68.01% | 2.42M | 0.7845 | 4.64× | 0.8242 |
| | Ours | 73.83% | 1.87M | 0.8335 | 6.01× | **0.9123** |
| | | ±1.11% | ±0.08M | ±0.0073 | ±0.26× | ±**0.0151** |
| ResNet-34 | Teacher | 78.71% | 21.33M | - | - | - |
| | Random Search | 72.33% | 3.61M | 0.8308 | 5.95× | 0.8924 |
| | | ±1.53% | ±0.35M | ±0.0162 | ±0.60× | ±0.0154 |
| | N2N - removal | 70.11% | 4.25M | 0.8008 | 5.02× | 0.8554 |
| | Ours - removal | 74.05% | 3.18M | 0.8508 | 6.88× | 0.9192 |
| | | ±0.52% | ±0.65M | ±0.0307 | ±1.31× | ±0.0033 |
| | Ours | 73.68% | 2.36M | 0.8895 | 9.08× | **0.9246** |
| | | ±0.57% | ±0.15M | ±0.0069 | ±0.59× | ±**0.0076** |
| ShuffleNet | Teacher | 71.14% | 1.06M | - | - | - |
| | Random Search | 64.75% | 0.18M | 0.8264 | 6.37× | 0.8803 |
| | | ±2.15% | ±0.06M | ±0.0588 | ±2.68× | ±0.0152 |
| | Ours | 68.45% | 0.23M | 0.7855 | 4.74× | **0.9171** |
| | | ±1.38% | ±0.04M | ±0.0337 | ±0.78× | ±**0.0088** |

on CIFAR-100, the architecture found by our algorithm is $8$ times smaller than the original teacher network while the accuracy only drops by $2.3\%$. For ResNet-18 on CIFAR-100, the architecture found by our algorithm has a little bit more parameters than that found by RS but has higher accuracy by about $4\%$. For ResNet-34 on CIFAR-100, the architecture found by our proposed method has a higher accuracy as the architecture discovered by RS but only uses about $65\%$ of the number of parameters. Also for ResNet-34 on CIFAR-100, N2N only provides the results of layer removal, denoted as 'N2N - removal'. 'Ours - removal' refers to only considering the layer removal operation in the search space for fair comparison. We can see that 'Ours - removal' also significantly outperforms 'N2N - removal' in terms of both the accuracy and the compression ratio. We visualize architectures compressed from VGG-19 and ResNet-18 in Figure 2.1&2.2.

Table 2.2: Compression Results on CIFAR-10.

| CIFAR-10 | | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|---|
| VGG-19 | Teacher | 93.91% | 20.04M | - | - | - |
| | Random Search | 91.76% | 2.27M | 0.8865 | 10.54× | 0.9628 |
| | | ±0.88% | ±1.03M | ±0.0515 | ±5.83× | ±0.0149 |
| | N2N | 91.64% | 0.98M | 0.9513 | 20.53× | 0.9735 |
| | Ours | 92.27% | 0.81M | 0.9595 | 25.39× | **0.9809** |
| | | ±0.49 % | ±0.17M | ±0.0084 | ±4.85× | ±**0.0050** |
| ResNet-18 | Teacher | 95.24% | 11.17M | - | - | - |
| | Random Search | 92.29% | 0.79M | 0.9293 | 14.42× | 0.9641 |
| | | ±0.83% | ±0.14M | ±0.012 | ±2.39× | ±0.0093 |
| | N2N | 91.81% | 1.00M | 0.9099 | 11.10× | 0.9562 |
| | Ours | 92.99% | 0.85M | 0.9239 | 14.44× | **0.9701** |
| | | ±1.03% | ±0.34M | ±0.0302 | ±5.05× | ±**0.0070** |
| ResNet-34 | Teacher | 95.57% | 21.28M | - | - | - |
| | Random Search | 92.87% | 1.70M | 0.9199 | 12.59× | 0.9655 |
| | | ±0.40% | ±0.18M | ±0.0084 | ±1.38× | ±0.0046 |
| | N2N | 92.35% | 2.07M | 0.9020 | 10.20× | 0.9570 |
| | Ours | 92.70% | 1.32M | 0.9379 | 17.00× | **0.9660** |
| | | ±0.74% | ±0.35M | ±0.0163 | ±5.11× | ±**0.0072** |
| ShuffleNet | Teacher | 90.87% | 0.99M | - | - | - |
| | Random Search | 88.25% | 0.15M | 0.8490 | 7.38× | 0.9471 |
| | | ±0.51% | ±0.05M | ±0.0529 | ±3.24× | ±0.0095 |
| | Ours | 89.36% | 0.10M | 0.8995 | 10.43× | **0.9729** |
| | | ±1.05% | ±0.03M | ±0.0284 | ±2.54× | ±**0.0055** |

ShuffleNet is an extremely computation-efficient human-designed CNN architecture [183]. We also have tried to use ShuffleNet as the teacher network and see if we can further optimize this architecture. As shown in Table 2.1& 2.2, our search algorithm successfully compresses 'ShuffleNet $1\times(g=2)$' by $10.43\times$ and $4.74\times$ on CIFAR-10 and CIFAR-100 respectively and the compressed architectures can still achieve similar accuracy to the original teacher network. Here '$1\times$' indicates the width multiplier in the teacher ShuffleNet and '$(g=2)$' indicates that the number of groups is 2. Readers can refer to Zhang et al. [183] for more details.

### 2.4.2 Comparison with ShuffleNet

We now compare the compressed architectures found by our algorithm with the state-of-the-art manually-designed compact network architecture ShuffleNet [183].

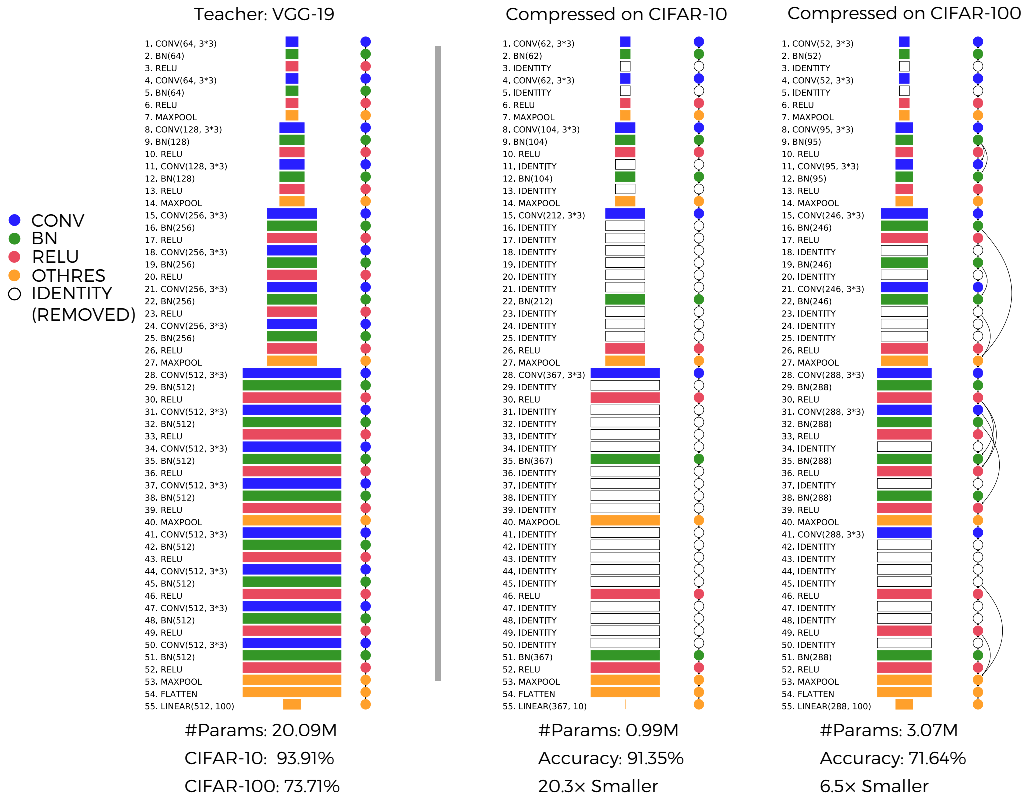



Figure 2.1: Visualization of architectures compressed from VGG-19.

We vary the number of channels and the number of groups in ShuffleNet and compare the compressed architectures found by our proposed method against these different configurations of ShuffleNet. We conduct experiments on CIFAR-100 and the results are summarized in Table 2.3. For 'Ours' in Table 2.3, we use the mean results of 3 runs of our method. In Table 2.3, VGG-19, ResNet-18, ResNet-34 and ShuffleNet refer to the compressed architectures found by our algorithm based on the corresponding teacher network and do *not* refer to the original architecture indicated by the name. The teacher ShuffleNet used in the experiments is 'ShuffleNet $1 \times (g = 2)$' as mentioned above. '$0.5 \times (g = 1)$' and so on in Table 2.3 refer to different configurations of ShuffleNet and we show the accuracy of these original ShuffleNet in the table. The compressed architectures found based on ResNet-18 and ResNet-34 have a similar number of parameters with ShuffleNet $1.5\times$ but they can all achieve much higher accuracy than ShuffleNet $1.5\times$. The compressed architecture found based on ShuffleNet $1 \times (g = 2)$ can obtain higher accuracy than ShuffleNet $0.5\times$ while using a similar number of parameters.

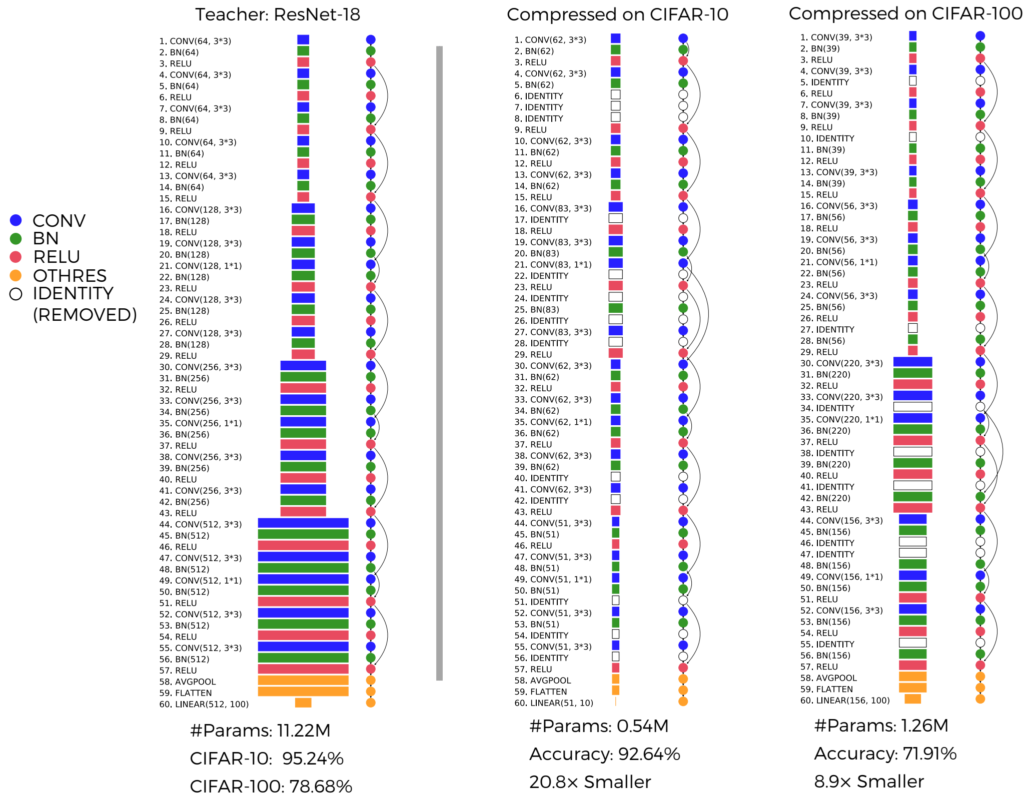



Figure 2.2: Visualization of architectures compressed from ResNet-18.

### 2.4.3 Comparison with TPE

Neural architecture search can be viewed as an optimization problem in a high-dimensional and discrete space. There are existing optimization methods such as TPE [10] and SMAC [66] that are proposed to handle such input spaces. To further justify our idea to learn a latent embedding space for the neural architecture domain, we now compare our method with directly using TPE to search for compressed architectures in the original hyperparameter value domain.

TPE [10] is a hyperparameter optimization algorithm based on a tree of Parzen estimator. In TPE, they use Gaussian mixture models (GMM) to fit the probability density of the hyperparameter values, which indicates that they determine the similarity between two architecture configurations based on the Euclidean distance in the original hyperparameter value domain. However, instead of comparing architecture configurations in the original hyperparameter value domain, our method transforms architecture configurations into a learned latent embedding space and compares them in the learned embedding space.

We first do not consider adding skip connections between layers and focus on

Table 2.3: Comparison with ShuffleNet on CIFAR-100.

| | Teacher | Accuracy | #Params | Teacher | Accuracy | #Params |
|---|---|---|---|---|---|---|
| Ours | VGG-19 | 71.41% | 2.61M | ResNet-18 | 73.83% | 1.87M |
| | ShuffleNet | 68.45% | 0.23M | ResNet-34 | 73.68% | 2.36M |
| | Configuration | Accuracy | #Params | Configuration | Accuracy | #Params |
| ShuffleNet | $0.5 \times (g = 1)$ | 67.71% | 0.26M | $1.5 \times (g = 1)$ | 72.43% | 2.09M |
| | $0.5 \times (g = 2)$ | 67.54% | 0.27M | $1.5 \times (g = 2)$ | 71.41% | 2.07M |
| | $0.5 \times (g = 3)$ | 67.23% | 0.27M | $1.5 \times (g = 3)$ | 71.05% | 2.03M |
| | $0.5 \times (g = 4)$ | 66.83% | 0.27M | $1.5 \times (g = 4)$ | 71.86% | 1.99M |
| | $0.5 \times (g = 8)$ | 66.74% | 0.31M | $1.5 \times (g = 8)$ | 71.04% | 2.08M |

Table 2.4: Comparison with TPE on CIFAR-100.

| CIFAR-100 | | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|---|
| ResNet-18 | TPE - removal + shrink | 70.60% | 1.30M | 0.8843 | 8.99x | 0.8849 |
| | | ±0.69% | ±0.28M | ±0.0249 | ±2.16x | ±0.0111 |
| | TPE | 65.17% | 1.54M | 0.8625 | 11.82x | 0.8041 |
| | | ±3.14% | ±1.42M | ±0.1267 | ±7.69x | ±0.0595 |
| | Ours - removal + shrink | 72.57% | 1.42M | 0.8733 | 8.85x | 0.9062 |
| | | ±0.58% | ±0.52M | ±0.0461 | ±3.97x | ±0.0081 |
| | Ours | 73.83% | 1.87M | 0.8335 | 6.01x | 0.9123 |
| | | ±1.11% | ±0.08M | ±0.0073 | ±0.26x | ±0.0151 |
| ResNet-34 | TPE - removal + shrink | 72.26% | 2.36M | 0.8893 | 9.24x | 0.9065 |
| | | ±0.83% | ±0.45M | ±0.0211 | ±1.59x | ±0.0072 |
| | Ours - removal + shrink | 73.72% | 2.75M | 0.8711 | 8.01x | 0.9205 |
| | | ±1.33% | ±0.55M | ±0.0257 | ±1.70x | ±0.0117 |
| | Ours | 73.68% | 2.36M | 0.8895 | 9.08x | 0.9246 |
| | | ±0.57% | ±0.15M | ±0.0069 | ±0.59x | ±0.0076 |

layer removal and layer shrinkage only, *i.e.*, we search for a compressed architecture by removing and shrinking layers from the given teacher network. Therefore, the hyperparameters we need to tune include for each layer whether we should keep it or not and the shrinkage ratio for each layer. This results in 64 hyperparameters for ResNet-18 and 112 hyperparameters for ResNet-34. We conduct the experiments on CIFAR-100 and the results are summarized in the Table 2.4. Comparing 'TPE - removal + shrink' and 'Ours - removal + shrink', we can see that our method outperforms TPE and can achieve higher accuracy with a similar size.

Now, we conduct experiments with adding skip connections. Besides the hyper-

Table 2.5: Summary of Kernel Transfer Results.

| | Method | Accuracy | Ratio | $f(x)$ | Method | Accuracy | Ratio | $f(x)$ |
|---|---|---|---|---|---|---|---|---|
| (a) $\rightarrow$ (b) | $K = 1$ | 93.13% | 0.8717 | 0.9584 | N2N on (b) | 92.35% | 0.9020 | 0.9570 |
| | $K = 8$ | 92.80% | 0.9627 | 0.9697 | Ours on (b) | 92.70% | 0.9379 | 0.9660 |
| (a) $\rightarrow$ (c) | $K = 1$ | 89.92% | 0.9793 | 0.9571 | N2N on (c) | 91.64% | 0.9513 | 0.9735 |
| | $K = 8$ | 92.79% | 0.9671 | 0.9870 | Ours on (c) | 92.27% | 0.9595 | 0.9809 |
| (a) $\rightarrow$ (d) | $K = 1$ | 68.77% | 0.9393 | 0.8708 | N2N on (d) | 68.01% | 0.7845 | 0.8242 |
| | $K = 8$ | 70.93% | 0.8586 | 0.8835 | Ours on (d) | 73.83% | 0.8335 | 0.9123 |

parameters mentioned above, for each pair of layers where the output dimension of one layer is the same as the input dimension of another layer, we tune a hyperparameter representing whether to add a skip connection between them. The results in 529 and 1717 hyperparameters for ResNet-18 and ResNet-34 respectively. In this representation, the original hyperparameter space is extremely high-dimensional and we think it would be difficult to directly optimize in this space. We can see from the table that for ResNet-18, the 'TPE' results are worse than 'TPE - removal + shrink'. We do not show the 'TPE' results for ResNet-34 here because the networks found by TPE have too many skip connections, which makes it very hard to train. The loss of those networks gets diverged easily and do not generate any meaningful results. Based on the results on 'layer removal + layer shrink' only and the results on the full search space, we can see that our method is better than optimizing in the original space especially when the original space is very high-dimensional.

We would like to point out that TPE [10] and SMAC [66] focus on improving Sequential Model-Based Optimization (SMBO) methods while our novelty is not in the use of Bayesian optimization methods. Our main contribution is the incrementally learning of an embedding to represent the configuration of network architectures such that we can carry out the optimization over the learned space instead of the original domain of the value of configuration parameters. Our method is complementary to TPE [10] and SMAC [66] and can be combined with them when being applied to NAS.

### 2.4.4 Kernel Transfer

We now study the transferability of the learned embedding space or the learned kernel. We would like to know to what extent a kernel learned in one setting can be generalized to a new setting. To be more specific about the kernel transfer, we first learn one kernel or multiple kernels in the source setting. Then we maximize the acquisition function within the search space in the target setting and the acquisition function is computed based on the kernel learned in the source setting. The

maximizer of the acquisition function is a compressed architecture for the target setting. We evaluate this architecture in the target setting and compare it with the architecture found by applying algorithms directly to the target setting.

We consider the following settings: (a) ResNet-18 on CIFAR-10, (b) ResNet-34 on CIFAR-10, (c) VGG-19 on CIFAR-10, and (d) ResNet-18 on CIFAR-100. 'ResNet-18 on CIFAR-10' refers to searching for a compressed architecture with ResNet-18 as the teacher network for the dataset CIFAR-10 and so on. We first run our search algorithm in setting (a) and transfer the learned kernel to setting (b), (c) and (d) respectively to see how much the learned kernel can transfer to a larger teacher network in the same architecture family (this means a larger search space), a different architecture family (this means a totally different search space) or a harder dataset.

We learn $K$ kernels in the source setting (a) and we transfer all the $K$ kernels to the target setting, which will result in $K$ compressed architectures for the target setting. We report the best one among the $K$ architectures. We have tried $K = 1$ and $K = 8$ and the results are shown in Table 2.5. In all the three transfer scenarios, the learned kernel in the source setting (a) can help find reasonably good architectures in the target setting without actually training the kernel in the target setting, whose performance is better than the architecture found by applying N2N directly to the target setting. These results proves that the learned architecture embedding space or the learned kernel is able to generalize to new settings for architecture search without any additional training.

### 2.4.5 Choice of the Objective Function

We discuss the possible choices of the objective function for learning the embedding space in this section. In our experiments, we learn the LSTM weights $\theta$ by maximizing the predictive posterior probability, *i.e.*, minimizing the negative log posterior probability as defined in Eq. 2.6. There are two other alternative choices for the objective function as suggested by the reviewers. We discuss the two choices and compare them with our choice in the following text.

Intuitively, a meaningful embedding space should be predictive of the function value, *i.e*, the performance of the architecture. Therefore, a reasonable choice of the objective function is to train the LSTM by regressing the function value with a Euclidean loss. Technically, this is done by adding a fully connected layer $FC(\cdot;\theta')$ after the embedding, whose output is the predicted performance of the input architecture. However, directly training the LSTM by regressing the function value does not let us directly evaluate how accurate the posterior distribution characterizes the statistical structure of the function. As mentioned before, the posterior distribution guides the search process by influencing the choice of architectures for evaluation at each step. Therefore, we believe maximizing the predictive posterior probability is a more suitable training objective for our search algorithm than regressing the function value. To validate this, we have tried changing the objective function from Eq. 2.6 to the squared Euclidean distance between the predicted function value and

Table 2.6: Comparison of different objective functions. 'Euclidean' refers to regressing the function value with a Euclidean loss. 'Marginal' refers to maximizing the log marginal likelihood. 'Posterior' is our choice and refers to maximizing the predictive posterior probability.

| CIFAR-100 | | Accuracy | #Params | Ratio | Times | $f(x)$ |
| --- | --- | --- | --- | --- | --- | --- |
| VGG-19 | Euclidean | 70.95% | 2.47M | 0.8771 | 9.62× | 0.9453 |
| | | ±1.07% | ±1.26M | ±0.0627 | ±4.55× | ±0.0092 |
| | Marginal | 69.90% | 1.50M | 0.9254 | 16.14× | 0.9422 |
| | | ±0.69% | ±0.68M | ±0.3382 | ±9.22× | ±0.0071 |
| | Posterior | 71.41% | 2.61M | 0.8699 | 7.99× | **0.9518** |
| | | ±0.75% | ±0.61M | ±0.0306 | ±1.99× | ±**0.0158** |
| ResNet-18 | Euclidean | 71.67% | 1.62M | 0.856 | 7.07× | 0.8917 |
| | | ±0.67% | ±0.27M | ±0.0243 | ±1.09× | ±0.0137 |
| | Marginal | 72.80% | 1.72M | 0.8467 | 6.57× | 0.9033 |
| | | ±1.11% | ±0.18M | ±0.0160 | ±0.67× | ±0.0094 |
| | Posterior | 73.83% | 1.87M | 0.8335 | 6.01× | **0.9123** |
| | | ±1.11% | ±0.08M | ±0.0073 | ±0.26× | ±**0.0151** |
| ResNet-34 | Euclidean | 72.87% | 2.49M | 0.8834 | 8.90× | 0.9127 |
| | | ±1.11% | ±0.60M | ±0.2814 | ±2.04× | ±0.0103 |
| | Marginal | 73.11% | 3.34M | 0.8435 | 6.47× | 0.9059 |
| | | ±0.57% | ±0.48M | ±0.0224 | ±0.89× | ±0.0134 |
| | Posterior | 73.68% | 2.36M | 0.8895 | 9.08× | **0.9246** |
| | | ±0.57% | ±0.15M | ±0.0069 | ±0.59× | ±**0.0076** |

the true function value: $\frac{1}{|D|}\sum_{i:x_i\in D}(FC(h(x_i;\theta);\theta') - f(x_i))^2$. The results are summarized in Table 2.6. We observe that maximizing the predictive posterior probability consistently yields better results than the Euclidean loss.

Another possible choice of the objective function is to maximize the log marginal likelihood $\log p\left(f(D) \mid D;\theta\right)$, which is the conventional objective function for kernel learning [159,160]. We do not choose to maximize log marginal likelihood because we empirically find that maximizing the log marginal likelihood yields worse results than maximizing the predictive GP posterior as shown in Table 2.6. When using the log marginal likelihood, we observe that the loss is numerically unstable due to the log determinant of the covariance matrix in the log marginal likelihood. The training objective usually goes to infinity when the dimension of the covariance matrix is larger than 50, even with smaller learning rates, which may harm the search performance. Therefore, we learn the embedding space by maximizing the

Table 2.7: Ablation study of the number of kernels $K$.

| CIFAR-100 | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|
| $K=1$ | 73.42% | 2.68M | 0.8745 | 7.97x | 0.9181 |
| $K=2$ | 72.51% | 2.14M | 0.8996 | 9.96x | 0.9119 |
| $K=4$ | 73.70% | 2.47M | 0.8842 | 8.64x | 0.9238 |
| $K=8$ | 73.45% | 2.12M | 0.9006 | 10.06x | 0.9240 |
| $K=16$ | 73.38% | 1.81M | 0.9153 | 11.80x | 0.9256 |

predictive GP posterior instead of the log marginal likelihood.

### 2.4.6 Ablation Study

**Impact of number of kernels $K$.** We study the impact of the number of kernels $K$. We conduct experiments on CIFAR-100 and use ResNet-34 as the teacher network. We vary the value of $K$ and fix the number of evaluated architectures to 160. The results are summarized in Table 2.7. We can see that $K=4,8,16$ yield much better results than $K=1$. Also the performance is not sensitive to $K$ as $K=4,8,16$ yield similar results. In our main experiments, we fix $K=8$.

**Impact of adding skip connections.** Our search space is defined based on three operations: layer removal, layer shrinkage and adding skip connections. A key difference between our search space and N2N [3] is that they only support layer removal and shrinkage do not support adding skip connections. To validate the effectiveness of adding skip connections, we conduct experiments on CIFAR-100 and on three architectures. In Table 2.8, 'Ours - removal + shrink' refers to the search space without considering adding skip connections and 'Ours' refers to using the full search space. We can see that 'Ours' consistently outperforms 'Ours - removal + shrink' across different teacher networks, proving the effectiveness of adding skip connections.

**Impact of the maximization of the acquisition function.** As mentioned in Section 2.3.2, we have two choices to maximize the acquisition function $\mathrm{EI}_t(x)$: randomly sampling (RS) and evolutionary algorithm (EA). We conduct the experiments to compare RS and ES to compress ResNet-34 on CIFAR-100. We find that although EA is empirically better than RS in terms of maximizing $\mathrm{EI}_t(x)$, EA is slightly worse than RS in terms of the final search performance as shown in Table 2.9. For any $\mathrm{EI}_t(x)$, the solution found by EA $x_{\mathrm{EA}}$ may be better than the solution found by RS $x_{\mathrm{RS}}$, *i.e.*, $\mathrm{EI}_t(x_{\mathrm{EA}}) > \mathrm{EI}_t(x_{\mathrm{RS}})$. However, we observe that $f(x_{\mathrm{EA}})$ and $f(x_{\mathrm{RS}})$ are

Table 2.8: Ablation study of adding skip connections.

| CIFAR-100 | | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|---|
| ResNet-18 | Ours - removal + shrink | 72.57%<br>±0.58% | 1.42M<br>±0.52M | 0.8733<br>±0.0461 | 8.85×<br>±3.97× | 0.9062<br>±0.0081 |
| | Ours | 73.83%<br>±1.11% | 1.87M<br>±0.08M | 0.8335<br>±0.0073 | 6.01×<br>±0.26× | 0.9123<br>±0.0151 |
| ResNet-34 | Ours - removal + shrink | 73.72%<br>±1.33% | 2.75M<br>±0.55M | 0.8711<br>±0.0257 | 8.01×<br>±1.70× | 0.9205<br>±0.0117 |
| | Ours | 73.68%<br>±0.57% | 2.36M<br>±0.15M | 0.8895<br>±0.0069 | 9.08×<br>±0.59× | 0.9246<br>±0.0076 |

Table 2.9: Ablation study of the maximization of the acquisition function.

| CIFAR-100 | Accuracy | #Params | Ratio | Times | $f(x)$ |
|---|---|---|---|---|---|
| RS, $K = 1$ | 73.42% | 2.68M | 0.8745 | 7.97x | 0.9181 |
| RS, $K = 8$ | 73.45% | 2.12M | 0.9006 | 10.06x | 0.9240 |
| EA, $K = 1$ | 71.52% | 1.24M | 0.9420 | 17.23x | 0.9056 |
| EA, $K = 8$ | 72.40% | 2.15M | 0.8990 | 9.90x | 0.9104 |

usually similar. We also plot the values of $f(x)$ for the evaluated architectures when using RS and EA to maximize the acquisition function respectively in Figure 2.3. We can see that the function value of the evaluated architectures grows slightly more stable when using RS to maximize the acquisition function then using EA. Therefore, we choose RS in the following experiments for the sake of simplicity.

### 2.4.7 Analysis of Random Search Baseline

We observe that the random search (RS) baseline which maximizes $f(x)$ with random sampling can achieve very good performance. To analyze RS in more detail, we show the value of $f(x)$ for the 160 architectures evaluated in the search process in Figure 2.4. The specific setting we choose is ResNet-34 on CIFAR-100. We can see that although RS can sometimes sample good architectures with high $f(x)$ value, it is much more unstable than our method. The function value of the evaluated architectures selected by our method has a strong tendency to grow as we search more steps while RS does not show such trend. Also, from the histogram of values of $f(x)$, we can see that RS has a much lower chance to get architectures with high function values than our method. This is expected since our method leverages the

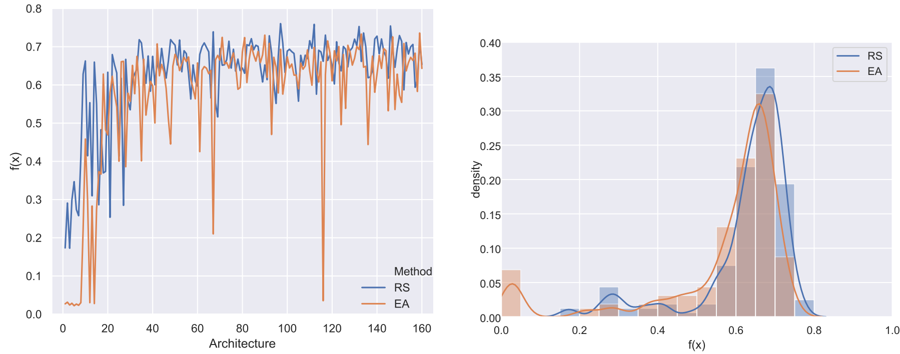



Figure 2.3: Comparison between random sampling (RS) and evolutionary algorithm (EA) for maximizing the acquisition function. Left: Value of $f(x)$ vs. Index of evaluated architecture. Right: Histogram of values of $f(x)$.

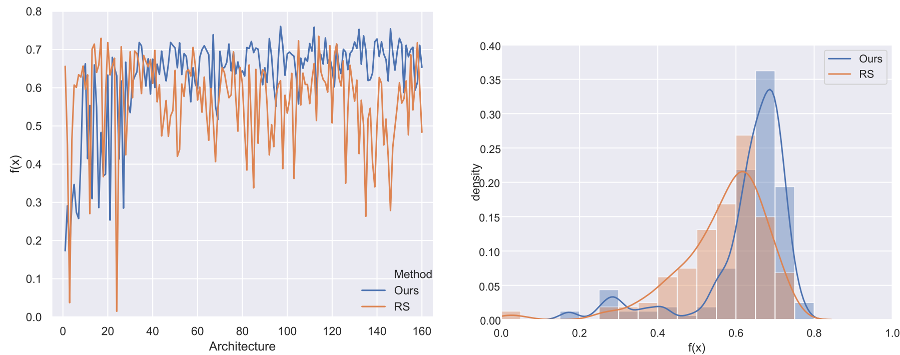



Figure 2.4: Comparison between our method and random search (RS) baseline. Left: Value of $f(x)$ vs. Index of evaluated architecture. Right: Histogram of values of $f(x)$.

learned architecture embedding or the kernel function to carefully select the architecture for evaluation while RS just randomly samples from the search space. We can conclude that our method is much more efficient than RS.

### 2.4.8 Implementation Details

We need to randomly sample architectures from the search space when optimizing the acquisition function. As mentioned in Section 2.3.2, we sample the architectures by sampling the operations to manipulate the architecture of the teacher network. During the process, we need to make sure the layers in the network are still compatible with each other in terms of the dimension of the feature map. Therefore, We impose some conditions when we sample the operations in order to maintain the

consistency between between layers.

For layer removal, only layers whose input dimension and output dimension are the same are allowed to be removed. For layer shrinkage, we divide layers into groups and for layers in the same group, the number of channels are always shrunken with the same ratio. The layers are grouped according to their input and output dimension. For adding skip connections, only when the output dimension of one layer is the same as the input dimension of another layer, the two layers can be connected. When there are multiple incoming edges for one layer, the outputs of source layers are added up to form the input for that layer.

When compressing ShuffleNet, we also slightly modify the original architecture before compression. We insert a $1 \times 1$ convolutional layer before each average pooling layer. This modification increases parameters by about $10\%$ and does not significantly influence the performance of ShuffleNet. Note that the modification only happens when we need to compress ShuffleNet and does not influence the performance of the original ShuffleNet shown in Table 2.3.

## 2.5 Conclusion

We address the task of searching for a compressed network architecture by using BO. Our proposed method can find more efficient architectures than all the baselines on CIFAR-10 and CIFAR-100. Our key contribution is the proposed method to learn an embedding space over the domain of network architectures. We also demonstrate that the learned embedding space can be transferred to new settings for architecture search without any training. Possible future directions include extending our method to the general NAS problem to search for desired architectures from the scratch and combining our proposed embedding space with [55] to identify the Pareto set of the architectures that are both small and accurate.

# Chapter 3

# Neighborhood-Aware NAS

## 3.1 Introduction

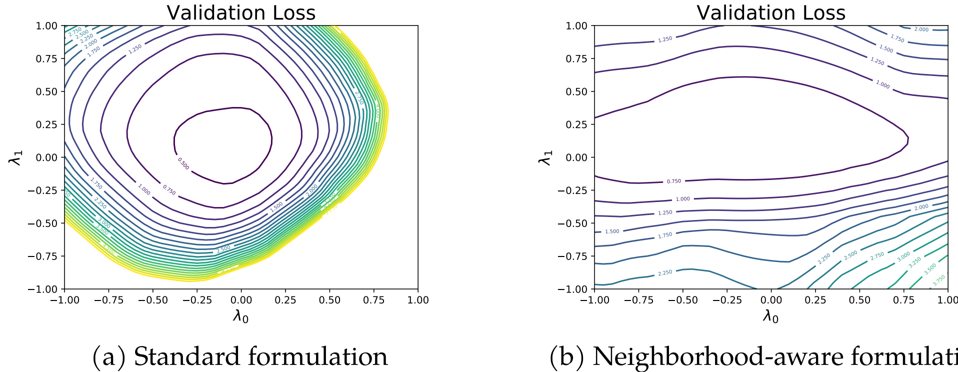
The process of automatic neural architecture design, also called neural architecture search (NAS), is a promising technology to improve performance and efficiency for deep learning applications [91,189,190]. NAS methods typically minimize the validation loss to find the optimal architecture. However, directly optimizing such an objective may cause the search algorithm to overfit to the search setting, *i.e.*, finding a solution architecture with good search performance but generalizes poorly to the test setting. This type of overfitting is a result of the differences between the search and test settings, such as the length of training schedules [189, 190], cross-architecture weight sharing [91,107], and the usage of proxy datasets during search [91,189,190].

To achieve better generalization, we propose a novel NAS formulation that aims to find "flat-minima architectures", which we define as architectures that perform well under small perturbations of the architecture (Figure 3.1). One example of architectural perturbations is to replace a convolutional operator with a skip connection (identity mapping). Our work takes inspiration from prior work on neural network training, which shows that flat minima of the loss function correspond to network weights with better generalization than sharp ones [57]. We show that flat minima in the architecture space also generalize better to a new data distribution than sharp minima (Section 3.3.3).

Unlike the standard NAS formulation that directly optimizes single architecture performance, *i.e.*, $\alpha^* = \arg\min_{\alpha \in \mathcal{A}} f(\alpha)$, we optimize the aggregated performance over the neighborhood of an architecture:

$$\alpha^* = \arg\min_{\alpha \in \mathcal{A}} g\left(f(\mathcal{N}(\alpha))\right), \tag{3.1}$$

where $f(\cdot)$ is a task-specific error metric, $\alpha$ denotes an architecture in the search space $\mathcal{A}$, $\mathcal{N}(\alpha)$ denotes the neighborhood of architecture $\alpha$, and $g(\cdot)$ is an aggregation function (*e.g.*, the $\mathrm{mean}$ function). Note that we overload the notation of the




(a) Standard formulation (b) Neighborhood-aware formulation

Figure 3.1: Loss landscape visualization of the found architecture. We project architectures (instead of the network weights) onto a 2D plane. The architectures are sampled along two prominent directions (the two axes, $\lambda_0$ and $\lambda_1$), with $(0, 0)$ denotes the found architecture. We see that our found architecture (right) is a much flatter minimum than that found with the standard formulation (left). We provide visualization details in Section 3.5.4.

error metric $f(\cdot)$ and define $f(\cdot)$ to return a set of errors when the input is a set of architectures in the neighborhood: $f(\mathcal{N}(\alpha)) = \{f(\alpha') \mid \alpha' \in \mathcal{N}(\alpha)\}$. Common choices for $f(\cdot)$ are validation loss and negative validation accuracy. We will discuss more details of neighborhood $\mathcal{N}(\alpha)$ and aggregation function $g(\cdot)$ in the following text.

To implement our formulation, one must define the neighborhood $\mathcal{N}(\alpha)$ and specify an aggregation function $g(\cdot)$. How to define the neighborhood of an architecture is an open question. One possible method to obtain neighboring architectures is to perturb one or more operations in the architecture and the degree of perturbation defines the scope of the neighborhood. This method can be applied to sampling-based search algorithms, *e.g.*, random search and reinforcement learning. However, it cannot be directly used to generate neighboring architectures for gradient-based search algorithms (*a.k.a*, differentiable NAS), where the neighboring architectures themselves also need to be differentiable with respect to the architecture being learned. To address this issue, we propose a differentiable representation for the neighborhood of architectures, which makes the objective function differentiable and allows us to apply our formulation to gradient-based algorithms, *e.g.*, DARTS [91]. Properly choosing the aggregation function $g(\cdot)$ can help the search algorithm identify flat minima in the search space. Our choice of $g(\cdot)$ (*e.g.*, $\mathrm{mean}$) is inspired by the definition of the flatness/sharpness of local minima in previous work [21,33,70].

We summarize our contributions as follows:

1. We propose a neighborhood-aware NAS formulation based on the flat minima assumption, and demonstrate a principled way to apply our formulation to ex-

isting search algorithms, including sampling-based algorithms and gradient-based algorithms. We empirically validate our assumption and show that flat-minima architectures generalize better than sharp ones.

2. We propose a neighborhood-aware random search (NA-RS) algorithm and demonstrate its superiority over the standard random search. On NAS-Bench-201 [36], NA-RS outperforms the standard random search by $1.48\%$ on CIFAR-100 and $1.58\%$ on ImageNet-16-120.

3. We propose a differentiable neighborhood representation so that we can apply our formulation to gradient-based NAS methods. We augment DARTS [91] with our formulation and name the proposed method NA-DARTS. Our NA-DARTS outperforms DARTS by $1.18\%$ on CIFAR-100 and $1.2\%$ on ImageNet, and also performs better than or on par with state-of-the-art NAS methods.

## 3.2 Related Work

**Flat Minima.** Hochreiter and Schmidhube [57] shows that flat minima of the loss function of neural networks generalize better than sharp minima. Flat minima are used to explain the poor generalization of large-batch methods [70, 176], where large-batch methods are shown to be more likely to converge to sharp minima. Chaudhari et al. [21] propose an objective function for training neural networks so that flat minima are preferred during optimization. Their objective can be interpreted as a weighted average of the (transformed) function values of data points around the local minima, which inspires us to consider $\mathrm{mean}$ as one of the aggregation functions. Previous work mentioned above focus on flat minima in the network weight space. However, we study flat minima in the architecture space, which is discrete and fundamentally different from the continuous weights studied in previous work. This makes it non-trivial to apply the flat minima idea to NAS.

Zela et al. [181] observes a strong correlation between the generalization error of the architecture found by DARTS [91] and the flatness of the loss function at the found architecture. They propose several regularization strategies to improve DARTS, such as early stopping before the loss curvature becomes too high. Our flat minima assumption is motivated by their observation and our method can be combined with their regularization strategies.

**NAS - Search Algorithm.** Various search algorithms have been applied to solve NAS, including sampling-based and gradient-based algorithms. Representative sampling-based algorithms include random search [81], reinforcement learning [4, 184,189,190], Bayesian optimization [17,69], evolutionary algorithms [115,117,165], and sequential model-based optimization [89]. To make NAS more computationally efficient, weight sharing across architectures is proposed to amortize the train-

ing cost of candidate architectures [8, 107]. Based on weight sharing, gradient-based algorithms are proposed to directly learn the architecture with gradient descent [91, 168]. Our focus is not proposing novel search algorithms but revisiting the standard NAS formulation. Our proposed formulation can be applied to both sampling-based algorithms and gradient-based algorithms.

**NAS - Search Space.** Search space is crucial for the performance of NAS. One of the most widely used search spaces is the cell search space [190], which searches for a cell that can be stacked multiple times to form the entire network. Our proposed neighborhood-aware formulation is agnostic to the search space, and we specifically showcase our formulation on the cell search space.

## 3.3 Neighborhood-Aware Formulation

We propose a neighborhood-aware NAS formulation (Eq. 3.1) to identify flat minima in the search space. Our formulation builds upon the assumption that flat-minima architectures usually generalize better than sharp ones. In this formulation, the optimal architecture is selected according to the aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$ of neighbors of an architecture, instead of the standard criterion, *i.e.*, single architecture performance $f(\alpha)$ only. We now introduce the neighborhood definition of an architecture $\mathcal{N}(\alpha)$ and the aggregation function $g(\cdot)$.

### 3.3.1 Neighborhood Definition and Cell Search Space

Formally defining the neighborhood requires a distance metric between architectures, which largely depends on how an architecture is represented and how the search space is constructed. We adopt the cell search space [190] as it has been widely used in recent NAS methods [89, 91]. Instead of the entire architecture, we search for a cell that can be stacked multiple times to form the entire architecture. The number of times the cell is stacked and the output layer are manually defined prior to the search.

A cell is defined as a directed acyclic graph (DAG) consisting of $n$ nodes. Each node represents a feature map. Each directed edge $(i, j)(1 \leq i < j \leq n)$ is associated with an operation used to transform the feature map at node $i$, and passes the transformed feature map to node $j$. The feature map at one node is the sum of all the feature maps on the incoming edges to this node: $x^{(j)} = \sum_{(i,j)\in E} \sum_{k=1}^{m} \alpha_k^{(i,j)} o_k(x^{(i)})$, where $E$ denotes the set of edges in the cell, $x^{(i)}$ is the feature map at node $i$, and $o_k$ is the $k^{th}$ operation among the $m$ available operations. $\alpha^{(i,j)}$ is a $m$-dim one-hot vector, indicating the operation choice for edge $(i, j)$. A cell is then represented by a set of variables $\alpha = \{\alpha^{(i,j)}\}$. Note that $\alpha^{(i,j)}$ being a one-hot vector means that only one operation is chosen for edge $(i, j)$. On a side note, the one-hot constraint on $\alpha^{(i,j)}$ can be relaxed in differentiable NAS methods [91, 168].

We define the distance between two cells $\alpha$ and $\alpha'$ as:

$$\text{dist}(\alpha, \alpha') = \sum_{(i,j)\in E} \delta(\alpha^{(i,j)}, \alpha'^{(i,j)}), \tag{3.2}$$

where $\delta(\cdot,\cdot)$ is the total variation distance between two probability distributions: $\delta(p,q) = \frac{1}{2}||p-q||_1 = \frac{1}{2}\sum_{k=1}^{m} |p_k - q_k|$. Here $p$ and $q$ are both $m$-dim probability distributions. The total variation distance is symmetric and bounded between $0$ and $1$. It also offers the following property: $\delta(\alpha^{(i,j)}, \alpha'^{(i,j)}) = 0$ implies that the two cells have the same operation at edge $(i,j)$ and $\delta(\alpha^{(i,j)}, \alpha'^{(i,j)}) = 1$ implies that they have different operations at edge $(i,j)$. Note that instead of directly counting the edge differences, we adopt total variation distance to accommodate relaxed $\alpha$ that is later used in differentiable NAS methods [91,168].

The neighborhood of a cell $\alpha$ is defined as:

$$\mathcal{N}(\alpha) = \{\alpha' \mid \text{dist}(\alpha, \alpha') \leq d\}, \tag{3.3}$$

where $d$ is a distance threshold. Due to the property of the total variation distance, when $d$ is an integer, the neighborhood contains all the cells that have at most $d$ edges associated with different operations from $\alpha$. For clarification, our definition of neighborhood includes the reference architecture $\alpha$ itself.

### 3.3.2 Aggregation Function

Our formulation aims to identify flat minima in the search space based on the aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$ over the neighborhood. The aggregation function $g(\cdot)$ needs to be properly set such that minimzing $g\left(f(\mathcal{N}(\alpha))\right)$ results in an architecture $\alpha$ that is a local minimum and at the same time has a flat neighborhood.

Given an architecture $\alpha$, the flatness of its neighborhood is determined by how much the performance (*e.g.*, validation loss) of its neighboring architectures varies compared to $\alpha$ itself. Intuitively, when $\alpha$ is a flat minimum, its neighboring architectures should perform similarly to $\alpha$. However, when $\alpha$ is a sharp minimum, the loss of architectures around $\alpha$ increases drastically compared to $\alpha$. Previous work [21,33,57,70,176] all shares this intuition.

Based on this intuition, we discuss possible choices for $g(\cdot)$:

- $\text{mean}$, $\text{median}$ or $\text{max}$.

  The architectures around a sharp minimum tend to high much higher loss compared to this minimum. Therefore, the mean validation loss of architectures around a flat minimum is expected to be lower than those around a sharp minimum. Minimizing $\text{mean}\left(f(\mathcal{N}(\alpha))\right)$ encourages the convergence to an architecture $\alpha$ whose neighbors in $\mathcal{N}(\alpha)$ all have a low loss, which implies that $\alpha$ is a flat minima. This makes $\text{mean}$ a valid choice. For a similar reason, $\text{median}$ and $\text{max}$ are also valid choices for $g(\cdot)$ to differentiate between flat minima and sharp minima.

Setting $g(\cdot)$ as $\text{mean}$ or $\max$ also aligns well with previous work on flat minima. Chaudhari et al. [21] propose an objective function for training neural networks so that flat minima are preferred during optimization. Their objective can be interpreted as a weighted average of the (transformed) function values of data points around the local minima, which inspires us to consider $\text{mean}$ as one of the choices for $g(\cdot)$. Keskar et al. [70] use the largest function value that can be attained in the neighborhood of a local minimum to characterize how sharp the minimum is, which leads us to set $g(\cdot)$ as $\max$.

- Variance.

  For an architecture $\alpha$, we can measure its flatness with the variance (standard deviation) of the performance of its neighbors in $\mathcal{N}(\alpha)$. Let $\sigma(f(\mathcal{N}(\alpha)))$ denote the standard deviation of the performance (*e.g.*, validation loss) of architectures in $\mathcal{N}(\alpha)$. But simply minimizing $\sigma(f(\mathcal{N}(\alpha)))$ can only result in an $\alpha$ with a flat neighborhood, but cannot guarantee that $\alpha$ is a local minimum (*e.g.*, have a low validation loss). So we propose the following variance-based aggregation function $g\left(f(\mathcal{N}(\alpha))\right) = f(\alpha) + \lambda\sigma(f(\mathcal{N}(\alpha)))$ that takes both the performance of $\alpha$ and the flatness of its neighborhood into account, where $\lambda$ is a hyper-parameter to balance the performance $f(\alpha)$ and the flatness $\sigma(f(\mathcal{N}(\alpha)))$.

### 3.3.3 Justification of Flat Minima Assumption

#### 3.3.3.1 Flat Minima Generalize Better

Flat minima in the network weight space are shown to generalize better than sharp ones [57]. However, we focus on flat minima in the architecture space, which is discrete and fundamentally different from the continuous weights studied in previous work. So we conduct experiments on NAS-Bench-201 [36] to verify that flat minima in the architecture space also generalize better.

NAS-Bench-201 provides a simulated environment for NAS experiments. Using NAS-Bench-201, we search on CIFAR-10 and evaluate the found architectures not only on CIFAR-10, but also on CIFAR-100 and ImageNet-16-120 to better assess the generalization performance of architectures. We select 100 architectures from NAS-Bench-201 that have the lowest validation error on CIFAR-10 to represent local minima in the search space. Next, we show that among these local-minima architectures, flat minima outperform sharp ones, especially on CIFAR-100 and ImageNet-16-120. We include the detailed experimental setup at the end of this subsection.

We measure the flatness of each local-minimum architecture with its neighborhood variance: the variance of the search-time validation error of its neighboring architectures on CIFAR-10. Based on their neighborhood variance, we divide the 100 architectures into 2 groups: (1) flat minima, which are the 50 architectures with a flat neighborhood (low neighborhood variance), and (2) sharp minima, which

Table 3.1: Average error of flat-minima architectures and sharp-minima architectures. The validation error ($f(\alpha)$) is defined as the CIFAR-10 validation error after the $90^{th}$ epoch. "CIFAR-10 Validation" refers to the average validation error on CIFAR-10 used in search. CIFAR-10, CIFAR-100 and ImageNet-16-120 refer to the average test error on each dataset. Flat minima and sharp minima obtain a similar validation error on CIFAR-10. However, flat minima consistently achieves lower test error than sharp minima on all three datasets.

| | CIFAR-10 Validation | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| Flat minima | 14.55 | **6.23** | **28.90** | **55.17** |
| Sharp minima | 14.57 | 6.66 | 30.00 | 56.41 |

are the other 50 architectures with a sharp neighborhood (high neighborhood variance).

We observe that the average search-time validation error of flat minima and sharp minima are almost the same (14.55% and 14.57%). But, as shown in Table 3.1, the average test error of flat minima is lower than sharp minima on all three datasets, especially on CIFAR-100 (1.10%) and ImageNet-16-120 (1.24%). This verifies that flat minima generalize better.

In the above results, we define $f(\alpha)$, *i.e.*, the search-time validation error, as the CIFAR-10 validation error after the $90^{th}$ epoch. Table 3.2 provides results where $f(\alpha)$ is defined as the validation error after other epochs (*e.g.*, $30^{th}$, $60^{th}$, $120^{th}$). We conduct the same experiments as Table 3.1 with the CIFAR-10 validation error after the $30^{th}$, $60^{th}$ or $120^{th}$ epoch. Results for all epochs ($30^{th}$, $60^{th}$, $90^{th}$, $120^{th}$) demonstrate the same pattern: the average validation error on CIFAR-10 of flat minima and sharp minima are similar; however, the average test error of flat minima is consistently lower than sharp minima on all three datasets, especially on CIFAR-100 and ImageNet-16-120.

#### 3.3.3.2 Aggregated Performance Gives a Better Ranking of Architectures

Based on the flat minima assumption, our neighborhood-aware formulation suggests using the aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$ as the criterion to select optimal architectures, instead of the standard criterion $f(\alpha)$. The selection criterion determines whether we can obtain an accurate ranking of candidate architectures during search, and further determines the performance of found architectures. We show that our criterion $g\left(f(\mathcal{N}(\alpha))\right)$ ranks architectures more accurately than $f(\alpha)$.

We evaluate the ranking estimated by our criterion $g\left(f(\mathcal{N}(\alpha))\right)$ or the standard criterion $f(\alpha)$ on NAS-Bench-201, where $f(\cdot)$ is the validation error on CIFAR-10. Specifically, we randomly sample 100 architectures from NAS-Bench-201 and rank these architectures according to $g\left(f(\mathcal{N}(\alpha))\right)$ or $f(\alpha)$. Then following Yu et al. [179], we evaluate the estimated ranking with the Kendall's Tau metric (the higher the bet-

Table 3.2: Additional results for average error of flat-minima architectures and sharp-minima architectures. Please refer to the caption of Table 3.1 for the meaning of each column.

(a) $f(\alpha)$ = CIFAR-10 validation error after the $30^{th}$ epoch.

| | CIFAR-10 Validation | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| Flat minima | 18.39 | **6.33** | **29.15** | **55.52** |
| Sharp minima | 18.45 | 6.67 | 30.10 | 56.18 |

(b) $f(\alpha)$ = CIFAR-10 validation error after the $60^{th}$ epoch.

| | CIFAR-10 Validation | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| Flat minima | 16.15 | **6.28** | **29.15** | **55.51** |
| Sharp minima | 16.43 | 6.91 | 30.56 | 57.31 |

(c) $f(\alpha)$ = CIFAR-10 validation error after the $120^{th}$ epoch.

| | CIFAR-10 Validation | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| Flat minima | 12.67 | **6.13** | **28.59** | **55.11** |
| Sharp minima | 12.81 | 6.33 | 29.28 | 55.53 |

Table 3.3: Kendall's Tau (rank correlation) obtained by the standard criterion $f(\alpha)$ (baseline) and our criterion $g\left(f(\mathcal{N}(\alpha))\right)$ with different choices of $g(\cdot)$.

| | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|
| Baseline | $0.66 \pm 0.03$ | $0.66 \pm 0.02$ | $0.64 \pm 0.03$ |
| Ours - $\mathrm{mean}$ | $\mathbf{0.76 \pm 0.03}$ | $\mathbf{0.77 \pm 0.03}$ | $\mathbf{0.74 \pm 0.03}$ |
| Ours - $\mathrm{median}$ | $0.72 \pm 0.03$ | $0.72 \pm 0.03$ | $0.69 \pm 0.03$ |
| Ours - $\mathrm{max}$ | $0.53 \pm 0.05$ | $0.54 \pm 0.05$ | $0.56 \pm 0.05$ |
| Ours - Variance | $0.72 \pm 0.02$ | $0.73 \pm 0.03$ | $0.71 \pm 0.02$ |

ter), which measures the correlation between the estimated ranking and ground truth ranking of architectures. The ground truth is obtained by sorting these architectures based on their test error. As the ground truth ranking is specific to each dataset, we evaluate the estimated ranking on the three datasets separately.

We repeat the experiments for 10 times and report the mean and standard deviation of the Kendall's Tau value. Table 3.3 shows the ranking estimation results when using different aggregation functions. For the variance-based aggregation function, we set $\lambda$ to $1.0$. As shown in Table 3.3, our criterion $g\left(f(\mathcal{N}(\alpha))\right)$ $(g(\cdot) = \mathrm{mean})$ ranks

Table 3.4: Average neighborhood variance and test error of architectures found by the standard criterion $f(\alpha)$ (baseline) and our criterion $g\left(f(\mathcal{N}(\alpha))\right)$ with different choices of $g(\cdot)$. 'Neighbor-Var' is the average neighborhood variance. Architectures found by the mean validation error ('Ours - $\mathrm{mean}$') have a much smaller neighborhood variance than those found by the baseline criterion, and also achieve lower classification error on all three datasets.

| | Neighbor-Var | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| Baseline | 5.58 | 6.45 | 29.45 | 55.79 |
| Ours - $\mathrm{mean}$ | 2.71 | **6.09** | **28.32** | **54.75** |
| Ours - $\mathrm{median}$ | 4.05 | 6.21 | 28.74 | 55.08 |
| Ours - $\max$ | 1.83 | 6.66 | 29.82 | 56.31 |
| Ours - Variance | 2.47 | 6.35 | 29.06 | 55.52 |

architectures much more accurately than the standard criterion $f(\alpha)$. Other aggregation functions except $\max$ also result in an more accurate ranking estimation of architectures than $f(\alpha)$.

#### 3.3.3.3 Aggregated Performance Finds Flat Minima

We conduct quantitative analysis to show that optimizing the proposed criterion, *i.e.*, the aggregated performance over the neighborhood, finds flat minima. We select 100 architectures from NAS-Bench-201 with the lowest validation error (baseline criterion) on CIFAR-10, and another 100 architectures with the lowest aggregated validation error (proposed criterion).

We measure the flatness of an architecture using neighborhood variance, where smaller variance indicates flatter neighborhood. We summarize the average neighborhood variance and test error of the found architectures in Table 3.4. We observe that optimizing the mean validation error ('Ours - $\mathrm{mean}$') can successfully help us find flat minima, as the found architectures have a much smaller neighborhood variance than those found by the baseline criterion, and also achieve lower classification error on all three datasets.

We also notice that when $g(\cdot) = \max$, the found architectures are not flat minima. Although these architectures have a flat neighborhood (low neighborhood variance), their classification performance is worse than those found by the baseline criterion. We think this is because when using $\max$, the objective $g\left(f(\mathcal{N}(\alpha))\right)$ only considers the flatness of the neighborhood, but fails to characterize how well the architecture $\alpha$ performs.

#### 3.3.3.4 Experimental Setup

NAS-Bench-201 [36] provides a simulated environment for NAS experiments by conducting a thorough evaluation of all the candidate architectures (cells) in a predefined cell search space on three datasets: CIFAR-10 [72], CIFAR-100 [72], and ImageNet-16-120 [36]. It contains the validation error (accuracy) of all the candidate architectures on CIFAR-10 after every training epoch, and the final test error on CIFAR-10, CIFAR-100, and ImageNet-16-120. ImageNet-16-120 is a subset and downsampled version of ImageNet [118] and contains about $158$K images divided into $120$ classes.

In our experiments, we set the distance threshold $d$ to $1$, so each architecture in the NAS-Bench-201 search space has $25$ neighbors including itself. We search on CIFAR-10 and evaluate the found architectures on all three datasets, *i.e.*, $f(\alpha)$ is defined as the validation error on CIFAR-10. It is common in NAS to use early stopping or budgeted training during search [39,83]. So, we use the CIFAR-10 validation error after the $90^{th}$ epoch in the experiments unless otherwise stated.

## 3.4 Neighborhood-Aware Search Algorithms

We propose neighborhood-aware random search and neighborhood-aware DARTS by applying our proposed formulation to random search (sampling-based) and DARTS (gradient-based), respectively.

### 3.4.1 Neighborhood-Aware Random Search

When applying our formulation to random search, we only need to change the criterion of selecting optimal architectures from $f(\alpha)$ to the aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$. At each step, we randomly sample an architecture $\alpha$ and compute its aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$, and choose the one with the best aggregated performance as our solution. We provide a detailed algorithm sketch of neighborhood-aware random search (NA-RS) in Algorithm 2.

In practice, the entire neighborhood may be large. Instead of using all the neighbors, we sample a subset of $n_{\text{nbr}}$ neighboring architectures from the neighborhood. In our implementation, we always include the reference architecture itself in the sampled subset.

Note that since NA-RS evaluates a neighborhood of architectures at each step, for fair comparison, we allow the standard random search (baseline) to run for more steps such that the two methods evaluate the same number of architectures during search. Specifically, if our NA-RS searches for $T$ steps, the standard random searches for $T \cdot n_{\text{nbr}}$ steps.

While we only present the application of our formulation to random search, the formulation is also applicable to other sampling-based search algorithms, such as reinforcement learning (RL) and Bayesian optimization (BO). Similar to NA-RS,

**Algorithm 2** Neighborhood-Aware Random Search

**Input**: Number of steps $T$. Number of neighbors $n_{\text{nbr}}$.
**for** $t = 1, 2, \ldots, T$ **do**
  Randomly sample an architecture from $\mathcal{A}$: $\alpha$.
  Sample $n_{\text{nbr}}$ neighboring architectures of $\alpha$: $\mathcal{N}(\alpha)$.
  Train the $n_{\text{nbr}}$ architectures and compute $g\left(f(\mathcal{N}(\alpha))\right)$.
  Let $\alpha^* = \alpha$ if $g\left(f(\mathcal{N}(\alpha))\right) < g\left(f(\mathcal{N}(\alpha^*))\right)$.
**end for**
Return the optimal architecture $\alpha^*$.

when applying our formulation to RL or BO, we only need to define the reward signal in RL or the objective function in BO as the aggregated performance $g\left(f(\mathcal{N}(\alpha))\right)$. Other components in RL or BO remain unchanged.

### 3.4.2 Neighborhood-Aware Differentiable Search

We now present how to apply our formulation to differentiable NAS methods. The key in these methods [27,91,168] is to make the objective $f(\alpha)$ differentiable with respect to the architecture $\alpha$ such that one can optimize $\alpha$ with gradient descent.

Similar to the case of random search, our formulation changes the objective from $f(\alpha)$ to $g\left(f(\mathcal{N}(\alpha))\right)$. With this change, the differentiability of $g\left(f(\mathcal{N}(\alpha))\right)$ is not guaranteed. Therefore, we propose a differentiable neighborhood representation for $\mathcal{N}(\alpha)$ and set the aggregation function $g(\cdot)$ to be $\mathrm{mean}$ ($g$ can also be other differentiable functions). This makes $g\left(f(\mathcal{N}(\alpha))\right)$ differentiable and allows us to simply adopt prior gradient estimation techniques, *e.g.*, the continuous relaxation in DARTS [91] or Gumbel-Softmax in SNAS [168], to derive the gradient of $g\left(f(\mathcal{N}(\alpha))\right)$. Other parts in the original diffferentiable NAS methods remain the same.

Specifically, we augment DARTS [91] with our formulation and adopt the continuous relaxation in DARTS to estimate the gradient. Therefore, we name our method neighborhood-aware DARTS (NA-DARTS). Note that our formulation is also applicable to other differentiable NAS methods, such as SNAS [168] and P-DARTS [27].

#### 3.4.2.1 Neighborhood-Aware DARTS

We first briefly review DARTS and then introduce the formulation of our proposed NA-DARTS.

**DARTS.** DARTS relaxes the discrete search space to be continuous so that the gradient of the validation loss with respect to the architecture $\alpha$ can be estimated, allowing optimizing $\alpha$ with gradient descent. Concretely, $\alpha^{(i,j)}$ is relaxed from a discrete

one-hot vector to a continuous distribution, and is parameterized as the output of the softmax function: $\alpha_k^{(i,j)} = \frac{\exp(\beta_k^{(i,j)})}{\sum_{k=1}^{m} \exp(\beta_k^{(i,j)})}$, where $m$ is the number of available operations and $\beta = \{\beta_k^{(i,j)}\}$ is the set of continuous logits to be learned. DARTS formulates NAS as the following bilevel optimization problem:

$$\begin{aligned} &\min_{\alpha} \mathcal{L}_{\text{val}}(w^*(\alpha), \alpha) \\ &\text{s.t. } w^*(\alpha) = \arg\min_{w} \mathcal{L}_{\text{train}}(w, \alpha), \end{aligned} \tag{3.4}$$

where $w$ denotes network weights, $w^*(\alpha)$ denotes the weights minimizing the training loss of architecture $\alpha$. $\mathcal{L}_{\text{train}}(w, \alpha)$ and $\mathcal{L}_{\text{val}}(w, \alpha)$ are the training loss and validation loss of architecture $\alpha$ with weights $w$, respectively.

We write the probability distributions $\alpha$ as the variables to be optimized in Eq. 3.4 instead of the logits $\beta$. In practice, we still follow DARTS and learn the logits $\beta$ with gradient descent instead of directly learning $\alpha$. We write DARTS formulation as Eq. 3.4 to make it easier to illustrate our neighborhood-aware formulation for DARTS, as the neighborhood $\mathcal{N}(\alpha)$ is defined in the domain of $\alpha$.

**NA-DARTS.** We augment DARTS with our neighborhood-aware formulation:

$$\begin{aligned} &\min_{\alpha} g(\{\mathcal{L}_{\text{val}}(w^*(\alpha'), \alpha') \mid \alpha' \in \mathcal{N}(\alpha)\}) \\ &\text{s.t. } w^*(\alpha') = \arg\min_{w} \mathcal{L}_{\text{train}}(w, \alpha'), \end{aligned} \tag{3.5}$$

where $\mathcal{N}(\alpha)$ is the neighborhood of architecture $\alpha$ and $g(\cdot)$ is an aggregation function.

To preserve differentiablity of the new objective (aggregated validation loss) in Eq. 3.5, both the neighboring architecture $\alpha'$ and aggregation function $g(\cdot)$ need to be differentiable with respect to $\alpha$. Next, we first describe how to represent the neighboring architecture $\alpha'$ as a differentiable function of $\alpha$ and, then discuss the gradient estimation for specific choices of $g(\cdot)$. An outline of the proposed NA-DARTS algorithm can be found in Algorithm 3.

#### 3.4.2.2 Differentiable Neighborhood Representation

When the one-hot constraint on $\alpha$ is relaxed, the neighborhood contains an infinite number of neighboring architectures. We propose a piratical method to sample a finite number of architectures from the neighborhood. Importantly, our method allows each sampled neighbor $\alpha'$ to be differentiable with respect to the reference architecture $\alpha$.

We generate neighboring architectures of $\alpha$ by perturbing the operations associated with the edges in $\alpha$. We randomly sample $d$ edges to be perturbed from the edge set $E$ of $\alpha$ and leave the operation choice for the remaining edges unchanged.

This implies that the distance between $\alpha$ and the neighboring architecture $\alpha'$ is at most $d$, thus as defined in Eq. 3.3, $\alpha'$ falls into the neighborhood of $\alpha$. Next, we describe two ways to sample and represent $\alpha'$ as a differentiable function of $\alpha$.

**Additive.** Let edge $(i, j)$ be an edge to be perturbed. Let $q^{(i,j)}$ be a $m$-dim real-valued noise vector satisfying the following condition: $|q_k^{(i,j)}| \leq \epsilon (0 < \epsilon < 1)$ and $\alpha_k^{(i,j)} + q_k^{(i,j)} \geq 0$ for all $k (1 \leq k \leq m)$. $\epsilon$ is the threshold of the noise. We randomly sample a noise vector $q^{(i,j)}$ and $\alpha'^{(i,j)}$ is computed as:

$$\alpha_k'^{(i,j)} = \frac{\alpha_k^{(i,j)} + q_k^{(i,j)}}{\sum_{k=1}^{m} (\alpha_k^{(i,j)} + q_k^{(i,j)})}. \tag{3.6}$$

Repeating the process for each edge to be perturbed will result in a neighboring architecture $\alpha'$, which is differentiable with respect to $\alpha$. Different noise vectors are sampled for different edges to be perturbed. We term Eq. 3.6 as the *additive representation* of neighboring architectures.

**Multiplicative.** Let edge $(i, j)$ be an edge to be perturbed and $r^{(i,j)}$ be a $m$-dim one-hot vector with $r_l^{(i,j)} = 1$ and $r_k^{(i,j)} = 0 (1 \leq k \leq m, k \neq l)$. We restrict $l$ to be either the index of the zero operation or skip connection. With the one-hot vector $r^{(i,j)}$, $\alpha'^{(i,j)}$ is computed as:

$$\alpha_k'^{(i,j)} = \frac{r_k^{(i,j)} \alpha_k^{(i,j)}}{\sum r_k^{(i,j)} \alpha_k^{(i,j)}}. \tag{3.7}$$

Under the multiplicative representation, $\alpha'^{(i,j)}$ has the same value as $r^{(i,j)}$, which indicates that the edge $(i, j)$ after perturbation chooses either the zero operation or skip connection. We term Eq. 3.6 as the *multiplicative representation* of neighboring architectures. We specifically develop this representation for when the aggregation function is $\max$ (see Section 3.4.2.3 for more details).

With the proposed additive and multiplicative representation, we can sample a set of neighboring architectures of $\alpha$ and the sampled architectures are differentiable with respect to $\alpha$. In practice, we uniformly sample $n_{\text{nbr}}$ neighboring architectures from the neighborhood and always include $\alpha$ itself in the sampled set.

#### 3.4.2.3 Gradient Estimation

After sampling a finite set of neighboring architectures, we compute the validation loss of each individual architecture $\alpha'$, where we use the current network weights $w$ as an approximation of $w^*(\alpha')$. Then we pass the set of the validation losses to the aggregation function $g(\cdot)$.

**Algorithm 3** Neighborhood-Aware DARTS

**Input**: Number of steps $T$. Number of neighbors $n_{\text{nbr}}$. Initial architecture $\alpha$ and weights $w$.
**for** $t = 1, 2, \ldots, T$ **do**
  Sample a batch of training data $X_{\text{train}}$ and a batch of validation data $X_{\text{val}}$.
  Sample $n_{\text{nbr}}$ neighboring architectures of $\alpha$: $\mathcal{N}(\alpha)$.
  **if** $g(\cdot) == \max$ **then**
    Compute $\bar{\alpha} = \arg\max_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w, \alpha')$ on $X_{\text{val}}$.
    Compute $\nabla_\alpha \mathcal{L}_{\text{val}}(w, \bar{\alpha})$ on $X_{\text{val}}$
    Update $\alpha$ by descending $\nabla_\alpha \mathcal{L}_{\text{val}}(w, \bar{\alpha})$.
  **else if** $g(\cdot) == \text{mean}$ **then**
    Compute $\nabla_\alpha \frac{\sum_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w,\alpha')}{|\mathcal{N}(\alpha)|}$ on $X_{\text{val}}$
    Update $\alpha$ by descending $\nabla_\alpha \frac{\sum_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w,\alpha')}{|\mathcal{N}(\alpha)|}$.
  **end if**
  Compute $\nabla_w \mathcal{L}_{\text{train}}(w, \alpha)$ on $X_{\text{train}}$
  Update $w$ by descending $\nabla_w \mathcal{L}_{\text{train}}(w, \alpha)$.
**end for**
Derive the final architecture based on the learned $\alpha$.

Our default choice of the aggregation function $g(\cdot)$ is $\text{mean}$. As discussed before, $g(\cdot)$ needs to be differentiable, which immediately rules out $\text{median}$. Both $\text{mean}$ and the variance-based aggregation function are differentiable. We prefer $\text{mean}$ because it requires fewer GPU memory. Theoretically, when computing $\nabla_\alpha g\left(f(\mathcal{N}(\alpha))\right)$, we need to keep all architectures in $\mathcal{N}(\alpha)$ in GPU. But when $g(\cdot) = \text{mean}$, we can compute $\nabla_\alpha f(\alpha')$ separately for each neighbor $\alpha' \in \mathcal{N}(\alpha)$. Since PyTorch [105] automatically accumulates the gradient in multiple backward passes, computing $\nabla_\alpha f(\alpha')$ separately is equivalent as computing $\nabla_\alpha \text{mean}\left(f(\mathcal{N}(\alpha))\right)$. Therefore, when using $\text{mean}$, we only need to keep one architecture in GPU. This requires much fewer GPU memory than the variance-based aggregation function and makes the algorithm more practical.

We choose $\text{mean}$ over $\max$ due to its superior empirical performance. When using $\max$, Eq. 3.5 becomes a minimax optimization problem and one can approximate the gradient of the objective using Danskin's Theorem [30]. For completeness, we include the details of using $\max$ in NA-DARTS here.

**Using** $\max$ **in NA-DARTS.** According to Danskin's Theorem [30], we approximate the gradient $\nabla_\alpha \max_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w^*(\alpha'), \alpha')$ with $\nabla_\alpha \mathcal{L}_{\text{val}}(w^*(\bar{\alpha}), \bar{\alpha})$, where $\bar{\alpha}$ is the maximizer of the inner maximization problem $\max_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w^*(\alpha'), \alpha')$. In practice, $w^*(\alpha')$ is approximated by the current network weights $w$. To compute the maximizer $\bar{\alpha}$, we simply compute the validation loss of each sampled neighboring

Table 3.5: Test error of NA-RS and the standard random search (RS). NA-RS consistently outperforms RS on all three datasets under the same computational budget.

| | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|
| Random Search (RS) | $6.39 \pm 0.32$ | $29.81 \pm 0.44$ | $56.30 \pm 1.08$ |
| NA-RS (Ours) | $\mathbf{6.20 \pm 0.35}$ | $\mathbf{28.33 \pm 1.22}$ | $\mathbf{54.72 \pm 0.96}$ |

architecture and choose the maximum one. As can seen from Algorithm 3, when using $\max$, we only need to keep one architecture ($\bar{\alpha}$) in GPU during the gradient computation.

Solving the inner maximization problem $\max_{\alpha' \in \mathcal{N}(\alpha)} \mathcal{L}_{\text{val}}(w^*(\alpha'), \alpha')$ is the process of finding the worst-performing neighbor of $\alpha$ in its neighborhood. Sampling neighbors with the additive representation of neighbors Eq. 3.6 might not always result in a neighbor $\alpha'$ that performs worse than $\alpha$. So, we specifically develop the multiplicative representation in Eq. 3.7. The multiplicative representation allows us to sample $\alpha'$ by changing a subset of operations in $\alpha$ to the zero operation or skip connection such that $\alpha'$ has a higher probability to perform worse than $\alpha$.

## 3.5 Experiments

### 3.5.1 Neighborhood-Aware Random Search

**Experimental setup.** We validate our NA-RS on NAS-Bench-201 [36]. Same as the experimental setup in Section 3.3.3, we search on CIFAR-10 and evaluate on CIFAR-10 [72], CIFAR-100 [72], and ImageNet-16-120 [36]. The number of search steps $T$ in NA-RS is set to $100$. For fair comparison, the standard random search (baseline; denoted as 'RS') is run for $T \cdot n_{\text{nbr}}$ steps, so that RS and NA-RS train and evaluate the same number of architectures. We set the distance threshold $d$ to $1$, so the neighborhood contains $25$ architectures including the reference architecture itself. We set $n_{\text{nbr}}$ to $10$ and use $\text{mean}$ as the aggregation function unless otherwise stated.

**Results.** As shown in Table 3.5, NA-RS consistently outperform RS on all three datasets, which validates our neighborhood-aware formulation. Notably, NA-RS outperforms RS by $1.48\%$ on CIFAR-100 and $1.58\%$ ImageNet-16-120. Note that the cell search space typically has a narrow performance range [171], so the improvement brought by our NA-RS is non-trivial.

**Ablation study.** We provide an ablation study of the aggregation function in NA-RS in Table 3.6 and an ablation study of $n_{\text{nbr}}$ in Table 3.7. We see from Table 3.6 that $\text{mean}$ and $\text{median}$ achieve the best performance among all the choices for $g(\cdot)$.

Table 3.6: Ablation study on the aggregation function in NA-RS. $\mathrm{mean}$ and $\mathrm{median}$ yield the lower test error among all the choices for $g(\cdot)$.

| | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|
| NA-RS - $\mathrm{mean}$ | $6.39 \pm 0.71$ | $28.68 \pm 1.75$ | $55.02 \pm 1.71$ |
| NA-RS - $\mathrm{median}$ | $6.20 \pm 0.35$ | $28.33 \pm 1.22$ | $54.72 \pm 0.96$ |
| NA-RS - $\mathrm{max}$ | $6.73 \pm 0.71$ | $29.70 \pm 1.61$ | $56.96 \pm 2.09$ |
| NA-RS - Variance | $6.65 \pm 0.97$ | $29.06 \pm 1.97$ | $55.48 \pm 2.41$ |

Table 3.7: Ablation study on $n_{\text{nbr}}$ in NA-RS. Sampling a subset of neighbors ($n_{\text{nbr}} = 10$) is a good approximation for the entire neighborhood ($n_{\text{nbr}} = 25$).

| | | CIFAR-10 | CIFAR-100 | ImageNet-16-120 |
|---|---|---|---|---|
| NA-RS - $\mathrm{mean}$ | $n_{\text{nbr}} = 10$ | $6.39 \pm 0.71$ | $28.68 \pm 1.75$ | $55.02 \pm 1.71$ |
| | $n_{\text{nbr}} = 25$ | $6.24 \pm 0.39$ | $28.24 \pm 1.25$ | $54.74 \pm 1.73$ |
| NA-RS - $\mathrm{median}$ | $n_{\text{nbr}} = 10$ | $6.20 \pm 0.35$ | $28.33 \pm 1.22$ | $54.72 \pm 0.96$ |
| | $n_{\text{nbr}} = 25$ | $6.18 \pm 0.38$ | $28.20 \pm 1.27$ | $54.40 \pm 0.98$ |

$\mathrm{max}$ performs the worst, which is consistent with the conclusion in Table 3.3. As shown in Table 3.7, performance obtained by $n_{\text{nbr}} = 10$ is close to $n_{\text{nbr}} = 25$, which indicates that sampling a subset of neighbors is a good approximation for the entire neighborhood.

### 3.5.2 Neighborhood-Aware DARTS

Following the experimental setup in DARTS [91], we search on CIFAR-10 [72] and evaluate on three datasets: CIFAR-10 [72], CIFAR-100 [72] and ImageNet [118]. The performance on CIFAR-100 and ImageNet are more important, which reflects how well the found architecture can generalize to new datasets. For our NA-DARTS, we sample 10 neighboring architectures at each step, *i.e.*, $n_{\text{nbr}} = 10$. Complete experimental details and ablation results are included at then end of this subsection.

We first compare our NA-DARTS with DARTS. This comparison directly verifies the effectiveness of our neighborhood-aware formulation. As shown in Table 3.8, NA-DARTS consistently outperforms DARTS on all three datasets. Notably, NA-DARTS outperforms DARTS by 1.18% on CIFAR-100 and 1.2% on ImageNet. Note that the cell search space used in DARTS has a narrow performance range [171]. For example, the top-1 error on CIFAR-100 mostly fall around 17%. So the performance gap between our NA-DARTS and DARTS is non-trivial.

NA-DARTS also outperforms or performs on par with other state-of-the-art NAS methods (Table 3.9 & 3.10). NA-DARTS obtains the lowest test error on CIFAR-100

Table 3.8: Test error of NA-DARTS and DARTS on CIFAR-10, CIFAR-100 and ImageNet. Our NA-DARTS consistently outperforms DARTS on all three datasets.

| Method | Top-1 Test Error (%) CIFAR-10 | CIFAR-100 | ImageNet | Params (M) CIFAR | ImageNet |
|---|---|---|---|---|---|
| DARTS 1st [91] | $2.90 \pm 0.25$ | $17.66 \pm 0.83$ | - | 2.9 | - |
| DARTS 2nd [91] | $2.70 \pm 0.08$ | $17.72 \pm 0.61$ | 26.7 | 2.9 | 4.7 |
| NA-DARTS (Ours) | $\mathbf{2.63 \pm 0.12}$ | $\mathbf{16.48 \pm 0.13}$ | **25.5** | 3.2 | 4.8 |

Table 3.9: Comparison with state-of-the-art NAS methods on CIFAR-10 and CIFAR-100. Our NA-DARTS achieves the lowest test error on CIFAR-100. As all the architectures are searched on CIFAR-10, this shows that architectures found by NA-DARTS generalize better.

| Method | Test Error (%) CIFAR-10 | CIFAR-100 | Params (M) | Search Cost (GPU days) | Search Method |
|---|---|---|---|---|---|
| NASNet-A [190] | 2.65 | 17.10* | 3.3 | 1800 | RL |
| AmoebaNet-A [115] | 2.84* | 17.16* | 3.2 | 3150 | Evolution |
| PNAS [89] | 2.95* | 17.29* | 3.2 | 225 | SMBO |
| ENAS [107] | 2.54* | 17.18* | 3.9 | 0.5 | RL |
| SNAS [168] | $2.85 \pm 0.02$ | 18.25* | 2.8 | 1.5 | Gradient |
| P-DARTS [27] | 2.50 | 16.55 | 3.4 | 0.3 | Gradient |
| PC-DARTS [169] | $2.57 \pm 0.07$ | 16.74* | 3.6 | 0.1 | Gradient |
| DARTS+ [85] | 2.72* | 16.85* | 4.3 | 0.6 | Gradient |
| DARTS 1st [91] | $2.90 \pm 0.25$ | $17.66 \pm 0.83$ | 2.9 | 0.3 | Gradient |
| DARTS 2nd [91] | $2.70 \pm 0.08$ | $17.72 \pm 0.61$ | 2.9 | 1.0 | Gradient |
| NA-DARTS (Ours) | $2.63 \pm 0.12$ | $\mathbf{16.48 \pm 0.13}$ | 3.2 | 1.1 | Gradient |

* We train the reported architecture following the training setup in DARTS [91].

and the second lowest on ImageNet among state-of-the-art NAS methods. Note that P-DARTS, PC-DARTS and DARTS+ are all extensions of DARTS and the proposed neighborhood-aware formulation is also applicable to them. Their ideas to improve DARTS, *e.g.*, gradually increasing search depth in P-DARTS and the partial-channel connection idea in PC-DARTS, can all be combined with our method for better performance. Therefore, our improvement is complementary to theirs in reference to DARTS.

#### 3.5.2.1 Experimental Setup

Following DARTS [91], we search on CIFAR-10 [72] and evaluate on CIFAR-10 [72], CIFAR-100 [72] and ImageNet [118]. We use exactly the same setup as DARTS [91], including the cell search space, hyper-parameters, such as the learning rate and

Table 3.10: Comparison with state-of-the-art NAS methods on ImageNet. Our NA-DARTS obtains the second lowest test error on ImageNet. We expect further improvement since our contribution is orthogonal to other extensions of DARTS (*e.g.*, P-DARTS, PC-DARTS and DARTS+).

| Method | Test Error (%) Top-1 | Top-5 | Params (M) | +× (M) | Method | Test Error (%) Top-1 | Top-5 | Params (M) | +× (M) |
|---|---|---|---|---|---|---|---|---|---|
| DARTS [91] | 26.7 | 8.7 | 4.7 | 574 | AmoebaNet-A [115]* | 27.0 | 8.9 | 5.0 | 584 |
| P-DARTS [27]* | 25.3 | 8.1 | 4.9 | 557 | NASNet-A [190] | 26.0 | 8.4 | 5.3 | 564 |
| PC-DARTS [169]* | 25.7 | 8.3 | 5.3 | 586 | ENAS [107]* | 26.1 | 8.6 | 5.2 | 576 |
| DARTS+ [85]* | 26.4 | 8.5 | 5.0 | 586 | PNAS [89] | 25.8 | 8.1 | 5.1 | 588 |
| NA-DARTS (Ours) | 25.5 | 8.2 | 4.8 | 557 | SNAS [168] | 27.3 | 9.2 | 4.3 | 522 |

* We train the reported architecture following the training setup in DARTS [91].

weight decay factor, and other experimental details. We split the training images in CIFAR-10 into two subsets of equal size, which are used as the training and validation images during search. We construct a network of $8$ cells with an initial channel number as $16$ and train the network for $50$ epochs to learn $\alpha$.

After the search is done, we derive the final architecture from the learned $\alpha$ using exactly the same procedure as DARTS. When evaluating the found architecture on CIFAR-10 and CIFAR-100, we build a network of $20$ cells and train it for $600$ epochs with batch size 96 and cutout [32]. For our NA-DARTS, We set the initial number of channels of the network such that it has a similar network size with DRATS and contains around 3M parameters.

When evaluating on ImageNet, we build a network of $14$ cells. Same as DARTS, the network is trained for $250$ epochs with batch size $128$. We set the initial number of channels such that the number of multiply-add operations in the network is fewer than 600M when the input is $224 \times 224$. Some NAS methods use a different training setup to train the found architecture on ImageNet. For example, DARTS+ [85] trains for 800 epochs and P-DARTS [27] uses a large batch size 1024 (need 8 V100 GPUs, infeasible to us). For fair comparison, we retrain the found architecture reported by the authors in their paper using the same training setup as DARTS.

For our NA-DARTS, we sample a subset of $10$ neighbors in each step, *i.e.*, $n_{\text{nbr}} = 10$. The distance threshold $d$ for neighborhood can be interpreted as the number of edges to be perturbed. As each cell in the DARTS search space has $14$ edges, we set $d$ to $6$. The noise threshold $\epsilon$ in the additive representation is set to $0.1$. All experiments are performed on a NVIDIA GeForce RTX 2080 Ti GPU.

#### 3.5.2.2 Ablation Study

**Aggregation function.** We report the performance of NA-DARTS when the aggregation function is $\text{mean}$ or $\max$ in Table 3.11a. We observe that $\text{mean}$ outperforms $\max$, which is consistent with the conclusion in Table 3.3. We also notice that $\text{mean}$ consumes a longer search time than $\max$. This is because when using $\text{mean}$, we

Table 3.11: Ablation study of NA-DARTS.

(a) Impact of aggregation function.

| | Test Error (%) | | Param | Search Cost |
|---|---|---|---|---|
| | CIFAR-10 | CIFAR-100 | (M) | (GPU days) |
| max | $2.80 \pm 0.10$ | $16.89 \pm 0.31$ | 3.1 | 0.5 |
| mean | $2.63 \pm 0.12$ | $16.48 \pm 0.13$ | 3.2 | 1.1 |

(b) Impact of $d$.

| | Test Error (%) | | Param |
|---|---|---|---|
| | CIFAR-10 | CIFAR-100 | (M) |
| $d = 2$ | $2.62 \pm 0.08$ | $16.90 \pm 0.45$ | 3.2 |
| $d = 4$ | $2.65 \pm 0.19$ | $16.56 \pm 0.36$ | 3.1 |
| $d = 6$ | $2.63 \pm 0.12$ | $16.48 \pm 0.13$ | 3.2 |

need to back-propagate through every sampled neighboring architecture $\alpha'$, while we only need to back-propagate through one neighboring architecture $\bar{\alpha}$ when using $\max$.

**Distance threshold.** We study the impact of the distance threshold of $d$ in Table 3.11b, where we observe $d = 6$ achieves the best performance and $d = 4$ performs similarly with $d = 6$. Recall that the distance threshold $d$ can be interpreted as the number of edges to be perturbed and the cell in the DARTS search space has $14$ edges. We empirically find that when $d$ becomes larger that $6$, the neighborhood becomes too large and the performance drops.

### 3.5.3 Generalization Test on A New Search Space

Zela et al. [181] finds that on a wide range of search spaces, while DARTS [91] can successfully minimizes the validation loss during search, the found architectures are usually degenerated and generalize poorly to the test setting. To further validate our NA-DARTS, we conduct experiments on the search space suggested by [181] and show that in this new search space, NA-DARTS can still find architectures that generalize much better than DARTS.

The new search space is a subset of the original DARTS search space. The new search space is similar to the original search space, except that it only considers three candidate operations, including $3 \times 3$ separable convolution, skip connection, and the zero operation. Following [181], we refer to the new search space as 'S3 search space'.

Table 3.12: Test error of architectures found from the S3 search space on CIFAR-10 and CIFAR-100. **Top**: Our NA-DARTS outperforms both DARTS and DARTS-ES. Also, our NA-DARTS-ES easily outperform DARTS-ES, which shows that our formulation is applicable to DARTS-ES and yields further improvement. **Bottom**: Applying our formulation to PC-DARTS also yields further improvement. Our NA-PC-DARTS outperforms PC-DARTS by $0.72\%$ on CIFAR-100.

| | CIFAR-10 | CIFAR-100 |
|---|---|---|
| DARTS [91] | $4.13 \pm 0.98$ | $22.49 \pm 2.62$ |
| DARTS-ES [181] | $3.71 \pm 1.14$ | $19.21 \pm 0.65$ |
| NA-DARTS (Ours) | $2.97 \pm 0.18$ | $18.86 \pm 0.49$ |
| NA-DARTS-ES (Ours) | $\mathbf{2.49 \pm 0.02}$ | $\mathbf{17.03 \pm 0.41}$ |
| PC-DARTS [169] | $\mathbf{2.66 \pm 0.14}$ | $17.38 \pm 0.45$ |
| NA-PC-DARTS (Ours) | $2.69 \pm 0.08$ | $\mathbf{16.66 \pm 0.39}$ |

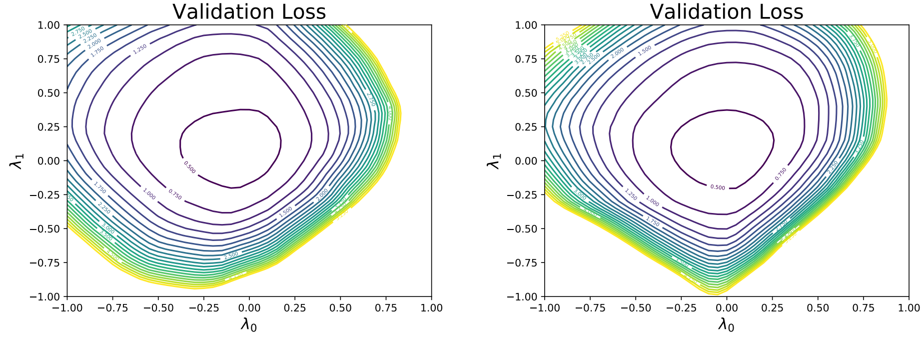
We search architectures from the S3 search space on CIFAR-10 using DARTS, DARTS-ES [181] or our NA-DARTS, and then evaluate the found architecture on both CIFAR-10 and CIFAR-100. DARTS-ES is DARTS with an early stopping criterion based on the dominant eigenvalue of the Hessian of the validation loss. We summarize the performance in the top half in Table 3.12. We see that our NA-DARTS easily outperforms both DARTS and DARTS-ES on CIFAR-10 and CIFAR-100. Notably, NA-DARTS outperforms DARTS by $3.63\%$ on CIFAR-100.

In the above, we mention that our contribution is orthogonal to other extensions of DARTS and we expect better performance after applying our formulation to them. To empirically verify this claim, we propose NA-DARTS-ES and NA-PC-DARTS by applying our neighborhood-aware formulation to DARTS-ES [181] and PC-DARTS [169], respectively. As shown in Table 3.12, NA-DARTS-ES outperforms DARTS-ES by $1.22\%$ on CIFAR-10 and $2.18\%$ on CIFAR-100. The improvement on CIFAR-100 demonstrates that architectures found by our NA-DARTS-ES generalize much better than those found by DARTS-ES.

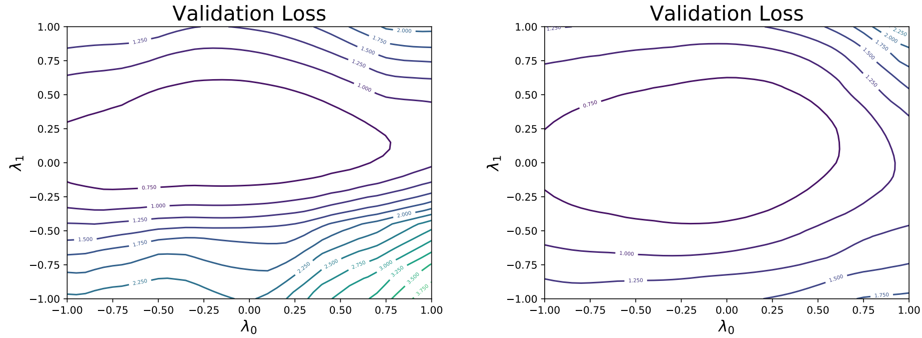
We notice that NA-PC-DARTS performs similarly to PC-DARTS on CIFAR-10. We would like to emphasize that the DARTS search space (a superset of the S3 search space) has a narrow performance range [171] and the test error of most NAS methods on CIFAR-10 are within $[2.5\%, 3.0\%]$. So we focus more on the performance on CIFAR-100. Our NA-PC-DARTS outperforms PC-DARTS by $0.72\%$ on CIFAR-100. As all the architectures are searched on CIFAR-10, this shows that architectures found by our NA-PC-DARTS generalize better than those found by PC-DARTS. We obtain further improvement after applying our formulation to PC-DARTS.

### 3.5.4 Loss Landscape Visualization

To qualitatively examine whether our NA-DARTS has found a flat minima, we plot the loss landscape of DARTS and NA-DARTS with the visualization strategy from




(a) DARTS (standard formulation $\min f(\alpha)$).

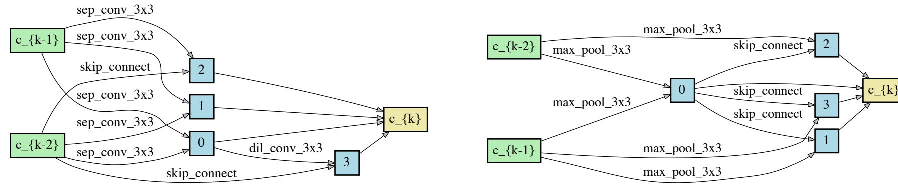



(b) NA-DARTS (neighborhood-aware formulation $\min g\left(f(\mathcal{N}(\alpha))\right)$).

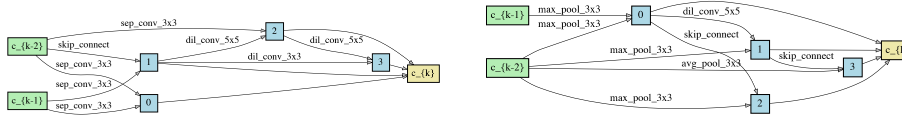
Figure 3.2: Loss landscape visualization of the found architecture (more runs). The two plots in Figure 3.2a (Figure 3.2b) are generated from two independent runs of DARTS (NA-DARTS). The left plot in Figure 3.2a and Figure 3.2b are the same as the plots in Figure 3.1 in the main text. For the architecture found by DARTS (Figure 3.2a), we observe that the loss of its neighbors increase drastically as the magnitude of $\lambda_0$ or $\lambda_1$ increases. However, for the architecture found by our NA-DARTS (Figure 3.2b), the loss of its neighbors increases much slower. This shows that the architecture found by our NA-DARTS is a much flatter minimum than that found by DARTS.

[78]. Let $\alpha$ denote the architecture found by DARTS or NA-DARTS. We compute the Hessian of the validation loss with respect to $\alpha$, and $v_0$ and $v_1$, which are the eigenvectors corresponding to the two largest eigenvalues of the Hessian matrix. Then we visualize the validation loss of the neighbors of $\alpha$ over the plane spanned by $v_0$ and $v_1$. Specifically, we compute the validation loss of the architecture $\alpha + \lambda_0 v_0 + \lambda_1 v_0$, where $\lambda_0$ and $\lambda_1$ are uniformly sampled from $[-1.0, 1.0]$. The loss values are visualized by the contour plots in Figure 3.2. We observe that the curvature of NA-DARTS at $(0, 0)$ (the found architecture $\alpha$) is much flatter than that of DARTS.

We provide details of the neighboring architecture $\alpha' = \alpha + \lambda_0 v_0 + \lambda_1 v_0$. Here, we




(a) DARTS normal cell (left) and reduction cell (right).




(b) NA-DARTS normal cell (left) and reduction cell (right).

Figure 3.3: Cell Visualization.

overload the plus sign $(+)$ with the additive representation. Recall that $\alpha$ contains a set of variables representing the operation choice for each edge $(i, j)$: $\alpha = \{\alpha^{(i,j)}\}$. The eigenvectors $v_0$ and $v_1$ have the same dimension as $\alpha$ and then can be represented as $v_0 = \{v_0^{(i,j)}\}$ and $v_1 = \{v_1^{(i,j)}\}$. Let $q^{(i,j)} = \lambda_0 v_0^{(i,j)} + \lambda_1 v_1^{(i,j)}$. $\alpha'^{(i,j)}$ is then computed using the additive representation in Eq. 3.6 ($\alpha_k'^{(i,j)} = \frac{\alpha_k^{(i,j)} + q_k^{(i,j)}}{\sum_{k=1}^{n}(\alpha_k^{(i,j)} + q_k^{(i,j)})}$). The eigenvectors $v_0$ and $v_1$ are normalized so that the scale of the noise vector $q^{(i,j)}$ is controlled only by $\lambda_0$ and $\lambda_1$. We use the weights of $\alpha$ obtained in the search as an approximation for the weights of the neighbors $\alpha'$.

### 3.5.5 Cell Visualization

We visualize the normal cell and reduction cell found by DARTS and our NA-DARTS in Figure 3.3. We observe that the normal cell found by our method NA-DARTS tend to be deeper than that found by DARTS. Normal cells found by our NA-DARTS from different runs have a depth of 3 at most of the time, while normal cells found by DARTS mostly have a depth of 1 or 2. We also observe that the normal cell found by NA-DARTS contains more $5 \times 5$ convolution operations. Both of the reduction cells found by DARTS and NA-DARTS contain very few convolution operations. Most operations in the reduction cell do not have parameters, *e.g.*, pooling and skip-connection.

## 3.6 Conclusion

To achieve better generalization, we propose a novel neighborhood-aware NAS formulation, based on the assumption that flat-minima architectures generalize better than sharp ones. Our formulation provides a new perspective for NAS that one should use the aggregated performance over the neighboorhood as the criterion to select optimal architectures. We also demonstrate a principled way to apply our formulation to existing search algorithms and propose two practical search algorithms NA-RS and NA-DARTS. Extensive experiments on CIFAR-10, CIFAR-100 and ImageNet validate the flat minima assumption, and demonstrate the significance of our formulation and algorithms.

# Part II

# Search Spaces

# Chapter 4

# AttentionNAS: Spatiotemporal Attention Cell Search

## 4.1 Introduction

One major contributing factor to the success of neural networks in computer vision is the novel design of network architectures. In early work, most network architectures [54, 73, 138] were manually designed by human experts based on their knowledge and intuition of specific tasks. Recent work on neural architecture search (NAS) [89, 91, 115, 189, 190] propose to directly learn the architecture for a specific task from data and discovered architectures have been shown to outperform human-designed ones.

Convolutional Neural Networks (CNNs) have been the *de facto* choice for network architectures. Most work in computer vision uses convolutional operations as the primary building block to construct the network. However, convolutional operations still have their limitations. It has been shown that attention is complementary to convolutional operations, and they can be combined to further improve performance on vision tasks [6, 154, 161].

While being complementary to convolution, many design choices remain to be determined to use attention. The design becomes more complex when applying attention to videos, where the following questions arise: *What is the right dimension to apply an attention operation to videos? Should an operation be applied to the temporal, spatial, or spatiotemporal dimension? How to compose multiple attention operations applied to different dimensions?*

Towards a principled way of applying attention to videos, we address the task of spatiotemporal attention cell search, i.e., the automatic discovery of cells that use attention operations as the primary building block. The discovered attention cells can be seamlessly inserted into a wide range of backbone networks, e.g., I3D [19] or S3D [167], to improve the performance on video understanding tasks.

Specifically, we propose a search space for spatiotemporal attention cells, which

allows the search algorithm to flexibly explore all of the aforementioned design choices in the cell. The attention cell is constrcuted by composing several primitive attention operations. Importantly, we consider two types of primitive attention operations: (1) map-based attention [104, 161] and (2) dot-product attention (a.k.a., self-attention) [6, 145, 154]. Map-based attention explicitly models where to focus in videos, compensating for the fact that convolutional operations apply the same filter to all the positions in videos. Dot-product attention enables the explicit modeling of long-range dependencies between distant positions in videos, accommodating the fact that convolutional operations only operate on a small and local neighborhood.

We aim to find an attention cell from the proposed search space such that the video classification accuracy is maximized when adding that attention cell into the backbone network. But the search process can be extremely costly. One significant bottleneck of the search is the need to constantly evaluate different attention cells. Evaluating the performance of an attention cell typically requires training the selected attention cell as well as the backbone network from scratch, which can take days on large-scale video datasets, e.g., Kinetics-600 [18].

To alleviate this bottleneck, we consider two search algorithms: (1) Gaussian Process Bandit (GPB) [130, 132], which judiciously selects the next attention cell for evaluation based on the attention cells having been evaluated so far, allowing us to find high-performing attention cells within a limited number of trials; (2) differentiable architecture search [91], where we develop a differentiable formulation of the proposed search space, making it possible to jointly learn the attention cell design and network weights through back-propagation, without explicitly sampling and evaluating different cells. The entire differentiable search process only consumes a computational cost similar to fully training one network on the training videos. This formulation also allows us to learn position-specific attention cell designs with zero extra computational cost (see Section 4.4.2 for details).

We conduct extensive experiments on two datasets: Kinetics-600 [18] and Moments in Time (MiT) [102]. Our discovered attention cells can improve the performance of two backbone networks I3D [19] and S3D [167] by more than 2% on both datasets, and also outperforms non-local blocks – the state-of-the-art manually designed attention cells for videos. Inserting our attention cells into I3D-R50 [154] yields state-of-the-art performance on both datasets. Notably, our discovered attention cells can also generalize well across modalities (RGB to optical flow), backbones (e.g., I3D to S3D or I3D to I3D-R50), and datasets (MiT to Kinetics-600 or Kinetics-600 to MiT).

**Contributions.** (1) This is the first attempt to extend NAS beyond discovering convolutional cells to attention cells. (2) We propose a novel search space for spatiotemporal attention cells that use attention operations as the primary building block, which can be seamlessly inserted into existing backbone networks to improve their performance on video classification. (3) We develop a differentiable formulation of the proposed search space, making it possible to learn the attention cell de-

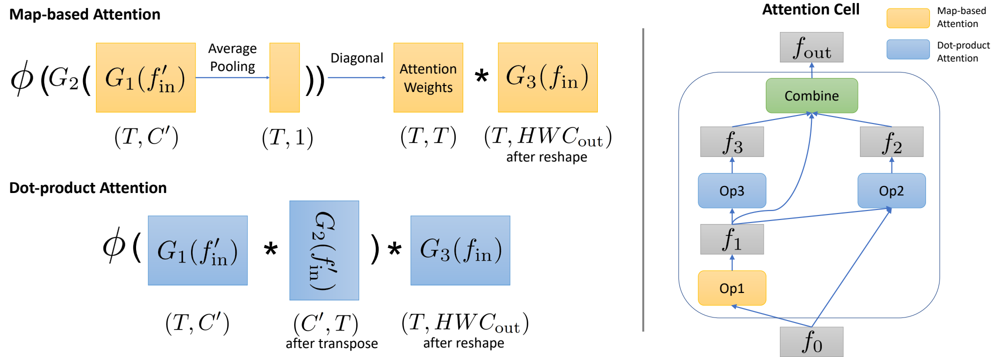



Figure 4.1: Illustration of the operation-level search space (left) and cell-level search space (right). The example attention operations use temporal as the attention dimension and the tuple under each feature map denotes its shape.

sign with back-propagation and learn position-specific attention cell designs with zero extra cost. (4) Our discovered attention cells outperform non-local blocks, on both the Kinetics-600 and MiT dataset. We achieve state-of-the-art performance on both datasets by inserting our discovered attention cells into I3D-R50. Our attention cells also demonstrate strong generalization capability when being applied to different modalities, backbones, or datasets.

## 4.2 Related Work

**Video Classification.** Early work on video classification extends image classification CNNs with recurrent networks [34, 180] or two-stream architectures [43, 128] that take both RGB frames and optical flow frames as inputs. Recent work on video classification are mainly based on 3D convolution [144] or its variants to directly learn video representations from RGB frames. I3D [19] proposes to inflate the filters and pooling kernels of a 2D CNN into 3D to leverage successful 2D CNN architecture designs and their ImageNet pretrained weights. S3D [167] improves upon I3D by decomposing a 3D convolution into a 2D spatial convolution and a 1D temporal convolution. A similar idea is also explored in P3D [113]. CPNet [92] learns video representations by aggregating information from potential correspondences. SlowFast [42] proposes an architecture operating at two different frame rates, where spatial semantics are learned on low frame rates, and temporal dynamics are learned on high frame rates. Different from them, we do not focus on proposing novel CNN architecture designs for video classification. Instead, we focus on discovering attention cells using attention operations as the primary building block, which are complementary to CNNs.

**Attention in Vision.** Both map-based attention and dot-product attention are useful for computer vision tasks. Map-based attention [104, 161] has been used to improve the performance of CNNs on image recognition, where spatial attention maps are learned to scale the features given by convolutional layers. Dot-product attention [145] is successfully used in sequence modeling and transduction tasks, e.g., machine translation, and is recently used to augment CNNs and enhances their performance on image recognition [6]. Non-local blocks [154] are proposed to capture long-range dependencies in videos and can significantly improve the video classification accuracy of CNNs. Non-local blocks can be viewed as applying one single dot-product attention operation to the spatiotemporal dimension. In contrast, our attention cells can contain multiple attention operations applied to different dimensions of videos. Non-local blocks are a particular case in our proposed search space, and our attention cells are discovered automatically in a data-driven way instead of being manually designed.

**NAS - Search Space.** Search space is crucial for NAS. Randwire [166] shows that one random architecture from a carefully designed search space can achieve competitive performance on image recognition. NASNet [190] proposes to search for convolutional cells that can be stacked multiple times to form the entire architecture. Auto-DeepLab [88] proposes a two-level hierarchical architecture search space for semantic image segmentation. AssembleNet [119] proposes to search for the connectivity between multi-stream convolutional blocks for video classification. They all focus on finding convolutional cells or networks for the end task. Different from them, our proposed search space uses attention as the primary building component instead of convolution.

**NAS - Search Algorithm.** Various search algorithms have been explored in NAS, such as random search [81, 179], reinforcement learning [4, 184, 189, 190], evolutionary algorithms [115, 117, 165], Bayesian optimization (BO) [17, 69], and differentiable methods [91]. We have tried using GPB (belonging to the category of BO) to search for desired attention cells. We also develop a differentiable formulation of our proposed search space. This makes it possible to conduct the search using differentiable methods and greatly improves the search speed.

## 4.3 Attention Cell Search Space

We aim to search for spatiotemporal attention cells, which can be seamlessly inserted into a wide range of backbone networks, e.g., I3D [19] or S3D [167], to improve the performance on video understanding tasks.

Formally, an attention cell takes a 4D feature map of shape $(T, H, W, C)$ as input and outputs a feature map of the same shape. $T$, $H$, and $W$ are the temporal dimension, height, and width of the feature map, respectively. $C$ denotes the number of

channels. The output of an attention cell is enforced to have the same shape as its input by design, so that the discovered attention cells can be easily inserted after any layers in any existing backbone networks.

An attention cell is composed of $K$ primitive attention operations. The proposed attention cell search space consists of an operation level search space and a cell level search space (see Fig. 4.1). The operation level search space contains different choices to instantiate an individual attention operation. The cell level search space consists of different choices to compose the $K$ operations to form a cell, i.e., the connectivity between the $K$ operations within a cell. We first introduce the operation level search space and then the cell level search space.

### 4.3.1 Operation Level Search Space

An attention operation takes a feature map of shape $(T, H, W, C_{\text{in}})$ as input and outputs an attended featured map of shape $(T, H, W, C_{\text{out}})$. For an attention operation, $C_{\text{in}}$ and $C_{\text{out}}$ can be different. To construct an attention operation, we need to make two fundamental choices: the dimension to compute the attention weights and the type of the attention operation.

#### 4.3.1.1 Attention Dimension

For brevity, we term the dimension to compute the attention weights as *attention dimension*. In CNNs for video classification, previous work [42,113,167] has studied when to use temporal convolution (e.g., $3\times1\times1$), spatial convolution (e.g., $1\times3\times3$), and spatiotemporal convolution (e.g., $3 \times 3 \times 3$). It is also a valid question to ask for attention what is the right dimension to apply an attention operation to videos: temporal, spatial or spatiotemporal (temporal and spatial together). The choice of the attention dimension is important as computing attention weights for different dimensions represents focusing on different aspects of the video.

#### 4.3.1.2 Attention Operation Type

We consider two types of attention operations, each of which helps address a specific limitation of convolutional operations, as mentioned in the introduction:

- **Map-based attention** [104,161]: Map-based attention learns a weighting factor for each position in the attention dimension and scales the feature map with the learned attention weights. Map-based attention explicitly models what positions in the attention dimension to attend to in videos.

- **Dot-product attention** [6,145,154]: A dot-product attention operation computes the feature response at a position as a weighted sum of features of all the positions in the attention dimension, where the weights are determined by a similarity function between features of all the positions [6,154]. Dot-product

attention explicitly models the long-range interactions among distant positions in the attention dimension.

We now describe the details of the two types of attention operations. Let $f_{\text{in}}$ denote the input feature map to an attention operation and denote its shape as $(T, H, W, C_{\text{in}})$. Applying an attention operation consists of three steps, including reshaping the input feature map $f_{\text{in}}$, computing the attention weights, and applying the attention weights.

**Reshape $f_{\text{in}}$.** We reshape $f_{\text{in}}$ into a 2D feature map $f'_{\text{in}}$ before computing the attention weights. The first dimension of $f'_{\text{in}}$ is the attention dimension and the second dimension contains the remaining dimensions. For example, $f'_{\text{in}}$ has the shape of $(T, HWC_{\text{in}})$ when temporal is the attention dimension and has the shape of $(THW, C_{\text{in}})$ when spatiotemporal is the attention dimension. We denote this procedure as a function `ReshapeTo2D`, i.e., $f'_{\text{in}} = \texttt{ReshapeTo2D}(f_{\text{in}})$.

Spatial attention requires extra handling. As video content changes over time, when applying attention to the spatial dimension, each frame $f^t_{\text{in}}$ should have its own spatial attention weights, where $f^t_{\text{in}}$ is the $t^{th}$ frame in $f_{\text{in}}$ and has the shape of $(H, W, C_{\text{in}})$. Therefore, when spatial is the attention dimension, instead of reshaping the entire 4D feature map $f_{\text{in}}$, we reshape $f^t_{\text{in}}$ into a 2D feature map $f'^t_{\text{in}}$ of shape $(HW, C_{\text{in}})$ for every $t$, i.e., $f'^t_{\text{in}} = \texttt{ReshapeTo2D}(f^t_{\text{in}})(1 \leq t \leq T)$.

**Map-based attention.** Assuming temporal is the attention dimension, map-based attention generates $T$ attention weights to scale the feature map of each temporal frame. The attention weights are computed as follows:

$$W_{\text{map}} = \texttt{Diag}(\phi(G_2(\texttt{AvgPool}(G_1(f'_{\text{in}}))))). \tag{4.1}$$

$G_1$ is a 1D convolutional layer with kernel size as 1, which reduces the dimension of the feature response of each temporal frame from $HWC_{\text{in}}$ to $C'$ and gives a feature map of shape $(T, C')$. `AvgPool` denotes an average pooling operation applied to each temporal dimension and outputs a $T$-dim vector. The multilayer perceptron $G_2$ and the activation function $\phi$ (e.g., the sigmoid function) further transform the $T$-dim vector to $T$ attention weights. More details about the activation function are discussed later. `Diag` rearranges the $T$ attention weights into a $T \times T$ matrix, where the $T$ attention weights are placed on the diagonal of the matrix. The obtained attention weight matrix $W$ is a diagonal matrix.

Similarly, when spatiotemporal is the attention dimension, map-based attention gives a $THW \times THW$ diagonal matrix containing the attention weights. When spatial is the attention dimension, we generate one $HW \times HW$ diagonal matrix for every $f'^t_{\text{in}}$ $(1 \leq t \leq T)$ separately, using the above described procedure. Note that while different frames have separate spatial attention weights, $G_1$ and $G_2$ are shared among different frames when computing attention weights.

**Dot-product attention.** When applying dot-product attention to the temporal dimension, a $T \times T$ attention weight matrix is generated as follows:

$$W_{\text{dot-prod}} = \phi(G_1(f'_{\text{in}})G_2(f'_{\text{in}})^T). \tag{4.2}$$

Here, $G_1$ and $G_2$ are both a 1D convolutional layer with kernel size as 1 and they both output a feature map of shape $(T, C')$. Let $Q = G_1(f'_{\text{in}})$ and $K = G_2(f'_{\text{in}})$. $QK^T$ computes an similarity matrix between the features of all the temporal frames. We then use $\phi$, an activation function of our choice, e.g., the softmax function, to convert the similarity matrix into attention weights. Note that different from $W_{\text{map}}$, $W_{\text{dot-prod}}$ is a full matrix instead of a diagonal matrix.

When being applied to the spatiotemporal dimension, dot-product attention generates a $THW \times THW$ attention weight matrix. When applying dot-product attention to the spatial dimension, each frame has its own attention weights (a $HW \times HW$ matrix), where $G_1$ and $G_2$ are shared among different frames.

**Apply the attention weights.** We apply the attention weight matrix to the input feature map through matrix multiplication to obtain the attended feature map:

$$f_{\text{out}} = \texttt{ReshapeTo2D}^{-1}(W\texttt{ReshapeTo2D}(G_3(f_{\text{in}}))). \tag{4.3}$$

$W$ is the weight matrix generated by map-based attention ($W_{\text{map}}$) or dot-product attention ($W_{\text{dot-prod}}$). $G_3$ is a $1 \times 1 \times 1$ convolutional layer to reduce the number of channels of $f_{\text{in}}$ from $C_{\text{in}}$ to $C_{\text{out}}$. If temporal is the attention dimension, $W$ has the shape of $(T, T)$ and $\texttt{ReshapeTo2D}(G_3(f_{\text{in}}))$ has the shape $(T, HWC_{\text{out}})$. $\texttt{ReshapeTo2D}^{-1}$ is the inverse function of `ReshapeTo2D`, reshaping the attended feature map back to the shape of $(T, H, W, C_{\text{out}})$.

For spatial attention, the attention weights are applied to each frame independently, i.e., $f^t_{\text{out}} = \texttt{ReshapeTo2D}^{-1}(W^t\texttt{ReshapeTo2D}(G_3(f^t_{\text{in}})))$, where $W^t$ is the spatial attention weights for frame $t$ and $f^t_{\text{out}}$ has the shape of $(H, W, C_{\text{out}})$. We stack $\{f^t_{\text{out}} \mid 1 \leq t \leq T\}$ along the temporal dimension to form the attended feature map $f_{\text{out}}$ of shape $(T, H, W, C_{\text{out}})$. Similar to $G_1$ and $G_2$ used for computing attention weights, $G_3$ is also shared among different frames.

Note that by design $G_3$ only changes number of channels, i.e., transforms the features at each spatiotemporal position. The spatiotemporal structure of the input $f_{\text{in}}$ is preserved. This ensures that after the application of attention weights, $f_{\text{out}}$ still follows the original spatiotemporal structure of the input $f_{\text{in}}$.

**Activation function.** We empirically find that the activation function $\phi$ (see Eq. 4.1 and Eq. 4.2) used in the attention operation can influence the performance. So, we also include the choice of the activation function in the operation level search space and rely on the search algorithm to choose the right one for each attention operation. We consider the following four choices for the activation function: (1) no activation function, (2) ReLU, (3) sigmoid, and (4) softmax.

**Channel attention.** While our search space mainly focuses on spatiotemporal attention, we include channel attention as an additional choice in the search space. Concretely, when building an attention operation, the search algorithm can choose whether to apply a feature gating layer [167] to the attended feature map $f_{\text{out}}$. The feature gating layer is a typical channel attention mechanism. It first applies average pooling to a 4D feature map across space and time, then learns a weighting factor for each channel, and finally multiplies features at each channel of the original feature map with the learned weighting factor. Note that channel attention does not replace the attention operation described above and is only an additional layer choice within the attention operation.

**Keys and values in dot-product attention.** We introduce an additional design choice in dot-product attention. In the above, a dot-product attention operation is defined as:

$$\begin{aligned} f'_{\text{in}} &= \texttt{ReshapeTo2D}(f_{\text{in}}), \\ W_{\text{dot-prod}} &= \phi(G_1(f'_{\text{in}})G_2(f'_{\text{in}})^T), \\ f_{\text{out}} &= \texttt{ReshapeTo2D}^{-1}(W\texttt{ReshapeTo2D}(G_3(f_{\text{in}}))), \end{aligned} \tag{4.4}$$

where $f_{\text{in}}$ and $f_{\text{out}}$ are the input and output feature map of the attention operation respectively, $W_{\text{dot-prod}}$ is the attention weight matrix, and $G_1$, $G_2$ and $G_3$ are all $1 \times 1 \times 1$ convolutional layers.

Let $Q = G_1(f'_{\text{in}})$, $K = G_2(f'_{\text{in}})$ and $V = \texttt{ReshapeTo2D}(G_3(f_{\text{in}}))$. $Q$, $K$ and $V$ are termed as query, keys and values in dot-product attention [145]. In Eq 4.4, the query, keys and values are computed based on the same feature map, i.e., the operation input $f_{\text{in}}$. It is also common practice in dot-product attention to compute the keys and values based on feature maps other than $f_{\text{in}}$. For example, dot-product attention has been used in Transformer [145] in the following way: the query comes from the decoder while the keys and values come from the encoder, so that every position in the decoded sequence can attend to positions in the input sequence.

In our search space, for a dot-product attention operation, we also allow computing its keys and values based on the cell input $f_0$. This allows positions in the operation input $f_{\text{in}}$ to attend to positions in the cell input $f_0$. When computing keys and values based on $f_0$, the dot-product attention becomes:

$$\begin{aligned} f'_0 &= \texttt{ReshapeTo2D}(f_0), \\ W_{\text{dot-prod}} &= \phi(G_1(f'_{\text{in}})G_2(\boldsymbol{f'_0})^T), \\ f_{\text{out}} &= \texttt{ReshapeTo2D}^{-1}(W\texttt{ReshapeTo2D}(G_3(\boldsymbol{f_0}))). \end{aligned} \tag{4.5}$$

The differences between Eq. 4.4 and Eq. 4.5 are highlighted in boldface and red.

In summary, for dot-product attention operations in the attention cell, we can choose to compute its keys and values based on the operation input $f_{\text{in}}$ or the cell

input $f_0$. We enforce all the dot-product attention operations in a cell to make the same choice, either computing the keys and values based on their own operation input or the cell input $f_0$.

### 4.3.2 Cell Level Search Space

We define an attention cell as a cell composed of $K$ attention operations. Let $f_0$ denote the input feature map to the entire attention cell and $(T, H, W, C)$ be the shape of $f_0$. $f_0$ is usually the output of a stack of convolutional layers. An attention cell takes $f_0$ as input and outputs a feature map of the same shape.

The connectivity between convolutional layers is essential to the performance of CNNs, no matter if the network is manually designed, e.g., ResNet [54] and Inception [138], or automatically discovered [166, 189, 190]. Similarly, to build an attention cell, another critical design choice is how the $K$ attention operations are connected inside the cell, apart from the design of these attention operations.

As shown in Fig. 4.1, in an attention cell, the first attention operation always takes $f_0$ as input and outputs feature map $f_1$. The $k^{th}(2 \leq k \leq K)$ attention operation chooses its input from $\{f_0, f_1, \ldots, f_{k-1}\}$ and gives feature map $f_k$ based on the selected input. We allow the $k^{th}$ operation to choose multiple feature maps from $\{f_0, f_1, \ldots, f_{k-1}\}$ and compute a weighted sum of selected feature maps as its input, where the weights are learnable parameters. This process is repeated for all $k$ and allows us to explore all possible connectivities between the $K$ attention operations in the cell.

We combine $\{f_1, f_2, \ldots, f_K\}$ to obtain the output feature map of the entire attention cell. For all attention operations inside the cell, we set their output shape to be $(T, H, W, C_{\text{op}})$, i.e., $f_k$ has the shape of $(T, H, W, C_{\text{op}})$ for all $k(1 \leq k \leq K)$. $C_{\text{op}}$ is usually smaller than $C$ to limit the computation in an attention cell with multiple attention operations. We concatenate $\{f_1, f_2, \ldots, f_K\}$ along the channel dimension and then employ a $1 \times 1 \times 1$ convolution to transform the concatenated feature map back to the same shape as the input $f_0$. We denote the feature map after transformation as $f_{\text{comb}}$. Similar to non-local blocks [154], we add a residual connection between the input and output of the attention cell. So the final output of the attention cell is the sum of $f_0$ and $f_{\text{comb}}$. The combination procedure is the same for all attention cells.

## 4.4 Attention Cell Search Algorithm

### 4.4.1 Gaussian Process Bandit (GPB)

Given $K$, i.e., the number of attention operations inside the attention cell, the attention cell design can be parameterized by a fixed number of hyper-parameters, including the attention dimension, the type and the activation function of each attention operation, and the input to each attention operation.

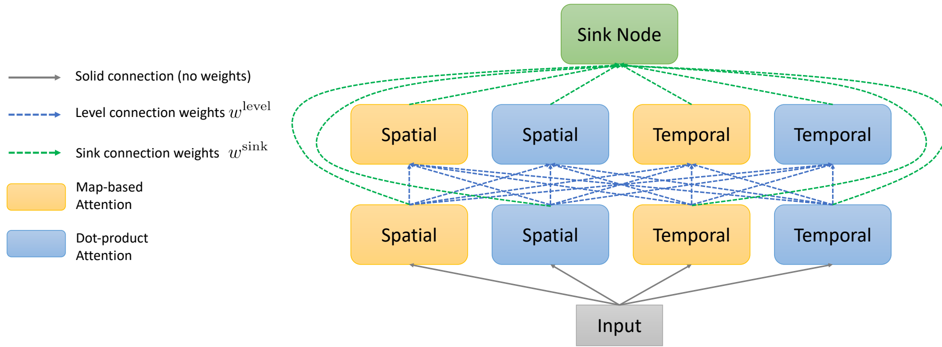



Figure 4.2: Illustration of the supergraph used by the differentiable method.

We employ GPB [130, 132], a popular hyper-parameter optimization algorithm, to optimize all the hyper-parameters for the attention cell design jointly. Intuitively, GPB can predict the performance of an attention cell at a modest computational cost without actually training the entire network, based on those already evaluated attention cells. Such prediction helps GPB to select promising attention cells to evaluate in the following step and makes it possible to discover high-performing attention cells within a limited number of search steps.

Concretely, in GPB, the performance of an attention cell is modeled as a sample from a Gaussian process. At each search step, GPB selects the attention cell for evaluation by optimizing the Gaussian process upper confidence conditioned on those already evaluated attention cells.

### 4.4.2 Differentiable Architecture Search

Inspired by recent progress on differentiable architecture search [91], we develop a differentiable formulation of our proposed search space. The formulation makes it possible to jointly learn the attention cell design and network weights with back-propagation, without explicitly sampling and evaluating different cells.

#### 4.4.2.1 Differentiable Formulation of Search Space

We propose to represent the attention cell search space as a supergraph, where all the possible attention cells are different subgraphs of this supergraph. The supergraph representation allows us to parameterize the design of an attention cell with a set of continuous and differentiable connection weights between the nodes in the supergraph.

To be more specific, we define the supergraph to have $m$ levels, where each level has $n$ nodes. Each node is an attention operation of a pre-defined type (map-based or dot-product attention) and a pre-defined attention dimension. Fig. 4.2 shows an example supergraph with 2 levels, where each level has 4 nodes. The input feature

map to the entire attention cell is passed to all the nodes at the first level. Starting from the second level, the input feature map to a node is a weighted sum of the output feature maps of all the nodes at its previous level:

$$f_{i,j}^{\text{in}} = \sum_{k=1}^{n} w_{i,j,k}^{\text{level}} \cdot f_{i-1,k}^{\text{out}}, \tag{4.6}$$

where $2 \leq i \leq m$, $1 \leq j \leq n$, $f_{i,j}^{\text{in}}$ is the input to the $j^{th}$ node at $i^{th}$ level, $f_{i-1,k}^{\text{out}}$ is the output of the $k^{th}$ node at $(i-1)^{th}$ level, and $w_{i,j}^{\text{level}}$ are the connection weights between the $j^{th}$ node at $i^{th}$ level and all the nodes at $(i-1)^{th}$ level. In practice, $w_{i,j}^{\text{level}}$ is a probability distribution obtained by softmax.

For each node in the supergraph, we also learn a probability distribution over the possible choices of activation functions. The output of a node is a weighted sum of the attended feature map under different activation functions:

$$f_{i,j}^{\text{out}} = \sum_{k=1}^{|\mathcal{A}|} w_{i,j,k}^{\text{activation}} \cdot f_{i,j}^{\text{out},\phi_k}, \tag{4.7}$$

where $\mathcal{A}$ is the set of available activation functions, $\phi_k$ is the $k^{th}$ activation function in $\mathcal{A}$, $w_{i,j,k}^{\text{activation}}$ is the weighting factor to be learned for $\phi_k$, and $f_{i,j}^{\text{out},\phi_k}$ is the attended feature map under the activation function $\phi_k$. The only difference among these attended feature maps $\{f_{i,j}^{\text{out},\phi_k}\}$ is the activation function $\phi$ used in Eq. 4.1 or Eq. 4.2. The layers $G_1$, $G_2$ and $G_3$ are shared by different activation functions within one node. We also learn a 2-dim probability distribution $w_{i,j}^{\text{gating}}$ for each node, indicating whether to include a feature gating layer [167] in the attention operation represented by the node.

The supergraph has a sink node, receiving the output feature maps of all the nodes. The sink node is defined as follows:

$$f_{\text{sink}}^{\text{out}} = \sum_{1 \leq i \leq m, 1 \leq j \leq n} w_{i,j}^{\text{sink}} \cdot G_{i,j}(f_{i,j}^{\text{out}}), \tag{4.8}$$

where $f_{\text{sink}}^{\text{out}}$ is the output of the sink node, $f_{i,j}^{\text{out}}$ is the output of the $j^{th}$ node at $i^{th}$ level, $G_{i,j}$ is a $1\times1\times1$ convolutional layer changing the number of channels in $f_{i,j}^{\text{out}}$ to $C$, and $w_{i,j}^{\text{sink}}$ is the weighting factor to be learned. We enforce $f_{\text{sink}}^{\text{out}}$ to have the same shape as the input to the supergraph, so that the supergraph can be inserted into any position of the backbone network. Same as attention cells, a residual connection is added between the input and output of the supergraph.

#### 4.4.2.2 Attention Cell Design Learning

Both the network weights, e.g., weights of convolutional layers in the network, and the connection weights in the supergraph ($\{w^{\text{level}}, w^{\text{sink}}, w^{\text{activation}}, w^{\text{gating}}\}$) are differentiable. During the search, we insert supergraphs into the backbone network

and jointly optimize the network weights and connection weights by minimizing the training loss using gradient descent. The entire search process only consumes a computational cost similar to fully training one network on the training videos. Once the training is completed, we can derive the attention cell design from the learned connection weights.

Note that we insert the supergraphs at positions where the final attention cells will be inserted. In practice, usually multiple supergraphs or attention cells (e.g., 5) are inserted into the backbone network. If we enforce the inserted supergraphs to share the same set of connection weights, we will obtain one single attention cell design, dubbed as the *position-agnostic* attention cell.

One significant advantage of the differentiable method is that we can also learn separate connection weights for supergraphs inserted at different positions, which will give *position-specific* attention cells (see Table 4.2). Searching for separate attention cells for different positions results in an exponentially larger search space than searching for one single attention cell. But thanks to the differentiable method, we can learn position-specific attention cells with zero extra cost compared to learning one position-agnostic attention cell.

#### 4.4.2.3 Attention Cell Design Derivation

We derive the attention cell design from the learned continuous connection weights. We first choose the top $\alpha$ nodes with the highest weights in $w^{\text{sink}}$ and add them to the set $S$. Then for each node in $S$, we add its top $\beta$ predecessors in its previous level to $S$, based on the corresponding connection weights in $w^{\text{level}}$. This process is conducted recursively for every node in $S$ until we reach the first level. $\alpha$ and $\beta$ are two hyper-parameters.

Recall that each node is an attention operation of a pre-defined type and attention dimension. So, $S$ contains a set of selected attention operations. The construction process of $S$ also determines how these attention operations are connected. For all the selected attention operations, we decide its activation function based on the corresponding weighting factors in $w^{\text{activation}}$, and whether to include a feature gating layer in the operation based on $w^{\text{gating}}$.

## 4.5 Experiments

### 4.5.1 Experimental Setup

**Datasets.** We conduct experiments on two benchmark datasets: Kinetics-600 [18] and Moments in Time (MiT) [102]. Top-1 and top-5 classification accuracy are used as the evaluation metric for both datasets.

**Backbones.** We conduct the attention cell search on two backbones: I3D [19] and S3D [167]. Both I3D and S3D are constructed based on the Inception [138] net-

work. When examining the generalization of the found cells, we also consider the backbone I3D-R50 [154], which is constructed based on ResNet-50 [54].

**Baselines.** Non-local blocks [154] are the state-of-the-art manually designed attention cell for video classification and are the most direct competitor of our automatically searched attention cells. We mainly focus on the relative improvement brought by our attention cells after being inserted into backbones. Besides non-local blocks, we also compare with other state-of-the-art methods for video classification, such as TSN [149], TRN [185], and SlowFast [42].

### 4.5.2 Search Results

Table 4.1: Search results on Kinetics-600 and MiT using GPB. Our attention cells improve the classification accuracy for both backbones and on both datasets.

| | | **Kinetics** | | | **MiT** | | |
|---|---|---|---|---|---|---|---|
| | Model | Top-1 | Top-5 | ΔTop-1 | Top-1 | Top-5 | ΔTop-1 |
| I3D | Backbone [19] | 75.58 | 92.93 | - | 27.38 | 54.29 | - |
| | Non-local [154] | 76.87 | 93.44 | 1.29 | **28.54** | 55.35 | **1.16** |
| | Ours - GPB | **77.39** | **93.63** | **1.81** | 28.41 | **55.49** | 1.03 |
| S3D | Backbone [167] | 76.15 | 93.22 | - | 27.69 | 54.68 | - |
| | Non-local [154] | 77.56 | 93.68 | 1.41 | **29.52** | **56.91** | **1.83** |
| | Ours - GPB | **78.28** | **94.04** | **2.13** | 29.23 | 56.22 | 1.54 |

Table 4.2: Search results on Kinetics-600 and MiT using the differentiable method. Our attention cells consistently outperform non-local blocks on all the combinations of backbones and datasets. Position-specific attention cells ('Pos-Specific') consistently outperform position-agnostic attention cells ('Pos-Agnostic').

| | | **Kinetics** | | | **MiT** | | |
|---|---|---|---|---|---|---|---|
| | Model | Top-1 | Top-5 | ΔTop-1 | Top-1 | Top-5 | ΔTop-1 |
| I3D | Backbone [19] | 75.58 | 92.93 | - | 27.38 | 54.29 | - |
| | Non-local [154] | 76.87 | 93.44 | 1.29 | 28.54 | 55.35 | 1.16 |
| | Ours - Pos-Agnostic | 77.56 | 93.63 | 1.98 | 28.18 | 55.01 | 0.80 |
| | Ours - Pos-Specific | **77.86** | **93.75** | **2.28** | **29.58** | **56.62** | **2.20** |
| S3D | Backbone [167] | 76.15 | 93.22 | - | 27.69 | 54.68 | - |
| | Non-local [154] | 77.56 | 93.68 | 1.41 | 29.52 | 56.91 | 1.83 |
| | Ours - Pos-Agnostic | 77.82 | 93.72 | 1.67 | 29.19 | 55.96 | 1.50 |
| | Ours - Pos-Specific | **78.51** | **93.88** | **2.36** | **29.82** | **57.02** | **2.13** |

Table 4.1 shows the search results of GPB and Table 4.2 summarizes the search

results using the differentiable method. Notably, attention cells found by the differentiable method can improve the accuracy of both backbones by more than 2% on both datasets, and consistently outperform non-local blocks on all the combinations of backbones and datasets.

In Table 4.2, 'Pos-Agnostic' refers to that one attention design is learned for all the positions where the cells are inserted. 'Pos-Specific' means that we learn a separate attention cell design for each position where a cell is inserted, i.e., the cells inserted at different positions can be different. We observe that position-specific attention cells consistently outperform position-agnostic attention cells.

### 4.5.3 Generalization of Discovered Cells

We examine how well the discovered attention cells can generalize to new settings. We do not perform any search in the following experiments, but directly apply attention cells searched for one setting to a different setting and see if the attention cells can improve the classification performance. Concretely, we evaluate whether our discovered attentions can generalize across different modalities, different backbones, and different datasets.

**Modality.** We insert the attention cells discovered on RGB frames into the backbone and train the network on optical flow only. The results are summarized in Table 4.3. 'GPB' refers to cells discovered by GPB and 'Differentiable' refers to cells discovered by the differentiable method. Our attention cells significantly improve the classification accuracy when being applied on optical flow and consistently outperform non-local blocks for both backbones and on both datasets. For example, our attention cells improve the accuracy of I3D by 5.67% on Kinetics-600. Note that the cells are discovered by maximizing its performance on RGB frames and no optical flow is involved during search. This demonstrates that our cells discovered on RGB frames can generalize well to optical flow.

**Backbone.** Table 4.4 summarizes the results of inserting cells discovered for one backbone to another backbone. The second row shows that cells discovered for S3D can still improve the classification accuracy of I3D by about 2% on both datasets, even though these cells are never optimized to improve the performance of I3D. We observe similar improvement when inserting cells found for I3D to S3D (third row), or cells found for I3D/S3D to I3D-R50 (last row). Notably, our attention cells can still outperform non-local blocks even after being inserted into a different backbone. For example, cells found for S3D achieve 77.81% accuracy on Kinetics-600 after being inserted to I3D, which outperforms non-local blocks (76.87%) and performs similar to cells specifically discovered for I3D (77.86%).

Table 4.3: Generalization across different modalities (RGB to Optical flow).

| | Model | Kinetics Top-1 | Kinetics Top-5 | Kinetics ΔTop-1 | MiT Top-1 | MiT Top-5 | MiT ΔTop-1 |
|---|---|---|---|---|---|---|---|
| I3D | Backbone [19] | 61.14 | 82.77 | - | 20.01 | 42.42 | - |
| | Non-local [154] | 64.88 | 85.77 | 3.74 | 21.86 | **46.59** | 1.85 |
| | Ours - GPB | 65.81 | 87.04 | 4.67 | 21.83 | 45.45 | 1.82 |
| | Ours - Differentiable | **66.81** | **87.85** | **5.67** | **21.94** | 45.57 | **1.93** |
| S3D | Backbone [167] | 62.46 | 84.59 | - | 20.50 | 42.86 | **-** |
| | Non-local [154] | 65.79 | 86.85 | 3.33 | 22.13 | **46.48** | 1.63 |
| | Ours - GPB | **67.02** | **87.72** | **4.56** | 22.29 | 46.16 | 1.79 |
| | Ours - Differentiable | 66.29 | 86.97 | 3.83 | **22.52** | 46.30 | **2.02** |

Table 4.4: Generalization across different backbones.

| | Model | Kinetics Top-1 | Kinetics Top-5 | Kinetics ΔTop-1 | MiT Top-1 | MiT Top-5 | MiT ΔTop-1 |
|---|---|---|---|---|---|---|---|
| I3D | Backbone [19] | 75.58 | 92.93 | - | 27.38 | 54.29 | - |
| | S3D - GPB | 77.47 | 93.67 | 1.89 | 28.92 | 56.09 | 1.54 |
| | S3D - Differentiable | **77.81** | **93.74** | **2.23** | **29.26** | **56.61** | **1.88** |
| S3D | Backbone [167] | 76.15 | 93.22 | - | 27.69 | 54.68 | - |
| | I3D - GPB | 78.23 | 94.07 | 2.08 | 29.45 | 56.50 | 1.76 |
| | I3D - Differentiable | **78.46** | **94.05** | **2.31** | **29.67** | **57.05** | **1.98** |
| I3D-R50 | Backbone [154] | 78.10 | 93.79 | - | 30.63 | 58.15 | - |
| | I3D - Differentiable | **79.83** | **94.37** | **1.73** | **32.48** | **60.31** | **1.85** |
| | S3D - Differentiable | 79.71 | 94.28 | 1.61 | 31.91 | 59.87 | 1.28 |

Table 4.5: Generalization across different datasets.

| | Model | MiT to Kinetics Top-1 | MiT to Kinetics Top-5 | MiT to Kinetics ΔTop-1 | Kinetics to MiT Top-1 | Kinetics to MiT Top-5 | Kinetics to MiT ΔTop-1 |
|---|---|---|---|---|---|---|---|
| I3D | Backbone [19] | 75.58 | 92.93 | - | 27.38 | 54.29 | - |
| | GPB | 77.34 | 93.47 | 1.76 | 27.62 | 56.70 | 0.24 |
| | Differentiable | **77.85** | **93.89** | **2.27** | **29.45** | **56.83** | **2.07** |
| S3D | Backbone [167] | 76.15 | 93.22 | - | 27.69 | 54.68 | - |
| | GPB | 77.54 | 93.62 | 1.39 | 28.80 | 56.16 | 1.11 |
| | Differentiable | **78.19** | **93.98** | **2.04** | **29.33** | **56.33** | **1.64** |

**Dataset.** We insert attention cells discovered on MiT to the corresponding backbone, fully train the network on Kinetics-600 and report its accuracy on Kinetics-600 in the middle column ('MiT to Kinetics') of Table 4.5. We observe that cells

Table 4.6: Comparison with the state-of-the-art methods. Our method ('I3D-R50+Cell') obtains similar or higher performance with the state-of-the-art methods on both Kinetics-600 and MiT.

(a) Kinetics-600.

| Model | Top-1 | Top-5 | GFLOPs |
|---|---|---|---|
| I3D [19] | 75.58 | 92.93 | 1136 |
| S3D [167] | 76.15 | 93.22 | 656 |
| I3D-R50 [154] | 78.10 | 93.79 | 938 |
| D3D [134] | 77.90 | - | - |
| I3D+NL [154] | 76.87 | 93.44 | 1305 |
| S3D+NL [154] | 77.56 | 93.68 | 825 |
| TSN-IRv2 [149] | 76.22 | - | 411 |
| StNet-IRv2 [53] | 78.99 | - | 440 |
| SlowFast-R50 [42] | **79.9** | **94.5** | 1971 |
| **I3D-R50+Cell** | 79.83 | 94.37 | 1034 |

(b) MiT.

| Model | Top-1 | Top-5 | Modality |
|---|---|---|---|
| I3D [19] | 27.38 | 54.29 | RGB |
| S3D [167] | 27.69 | 54.68 | RGB |
| I3D+NL [154] | 28.54 | 55.35 | RGB |
| S3D+NL [154] | 29.52 | 56.91 | RGB |
| R50-ImageNet [102] | 27.16 | 51.68 | RGB |
| TSN-Spatial [149] | 24.11 | 49.10 | RGB |
| I3D-R50 [154] | 30.63 | 58.15 | RGB |
| **I3D-R50+Cell** | **32.48** | **60.31** | RGB |
| TSN-2stream [149] | 25.32 | 50.10 | R+F |
| TRN-Multiscale [185] | 28.27 | 53.87 | R+F |
| AssembleNet-50 [119] | 31.41 | 58.33 | R+F |

discovered on MiT can improve the accuracy on Kinetics-600 by more than 2%, although they are never optimized to improve the Kinetics-600 performance during the search. Similarly, the right column ('Kinetics to MiT') demonstrates that the cells searched on Kinetics-600 can also generalize gracefully to MiT. We conclude that our attention cells generalize well across datasets.

Table 4.7: Comparison between different supergraph designs.

| Model | Top-1 | Top-5 |
|---|---|---|
| I3D [19] | 75.58 | 92.93 |
| I3D+NL [154] | 76.87 | 93.44 |
| I3D+SG-1 Cell | 77.86 | 93.75 |
| I3D+SG-2 Cell | 77.82 | 93.75 |
| I3D+SG-3 Cell | 77.71 | 93.87 |

### 4.5.4 Comparison with State-of-the-art

We insert our attention cells found on I3D into I3D-R50 ('I3D-R50+Cell') and compare with the state-of-the-art methods in Table 4.6. On Kinetics-600, we obtain similar performance with SlowFast-R50 [42] with fewer inference FLOPs. On MiT, we achieve 32.48% top-1 accuracy and 60.31% top-5 accuracy only using the RGB frames. This significantly outperforms the previous state-of-the-art method on MiT, AssembleNet-50 [119], which uses both RGB frames and optical flow.

### 4.5.5 Ablation Study of Supergraph Designs

In the differentiable method, we represent the attention cell search space as a supergraph. Using different supergraph designs allows us to analyze what desgin choice is important for the performance of the discovered attention cells. Specifically, we compare the following three supergraph designs:

**SG-1.** SG-1 is our default choice as described in Section 4.5.7.3. It contains 2 levels, where each level has 6 nodes. SG-1 only contains dot-product attention and the 6 nodes at each level are 2 temporal dot-product, 2 spatial dot-product, and 2 spatiotemporal dot-product attention operations. In SG-1, the keys and values of dot-product attention are computed based on the cell input (see Eq. 4.5).

**SG-2.** Same SG-1, SG-2 also contains 2 levels and each level has 6 nodes. SG-2 include both map-based attention and dot-product attention. The 6 nodes at each level are 1 temporal dot-product, 1 spatial dot-product, 1 spatiotemporal dot-product, 1 temporal map-based, 1 spatial map-based, and 1 spatiotemporal map-based attention operation. In SG-2, the keys and values of dot-product attention are also computed based on the cell input (see Eq. 4.5).

**SG-3.** SG-3 is the same as SG-1 except that the keys and values of dot-product attention are computed based on the input to each attention operation (see Eq. 4.4), instead of the cell input.

Comparing SG-1 and SG-2 tells us which attention type (map-based or dot-product) is more important. As shown in Table 4.7, SG-1 and SG-2 achieve a very close top-1 accuracy and the same top-5 accuracy on Kinetics-600. However, we observe that most operations (20 of out 28) in the 5 position-specific cells discov-

ered from SG-2 are dot-product attention. This shows that dot-product attention is more important than map-based attention, and explains why SG-1 can achieve high accuracy with only dot-product attention.

SG-3 achieves similar performance with SG-1 and also outperforms non-local blocks. This shows that our search space is not sensitive to whether to compute the keys and values based on the input to each dot-product operation or based on the cell input.

### 4.5.6 Attention Cell Visualization

We visualize the position-agnostic attention cell found by GPB and the differentiable method in Fig. 4.3. The position-specific cells found by the differentiable method are shown in Fig. 4.4. These cells are found for I3D and on Kinetics-600. We show the attention dimension and type of each operation, as well as the connectivity between the operations.

The cell found by GPB contains both map-based attention and dot-product attention and contains one path that first applies spatial attention and then temporal attention. Cells found by the differentiable method only contain dot-product attention as we only include dot-product attention in the supergraph (SG-1). We observe that all the cells found by the differentiable method prefer decomposing spatiotemporal attention into temporal and spatial attention, as they all contain paths that first apply temporal attention and then spatial attention. This shares a similar spirit to S3D [167] that decomposes a 3D convolution into a 2D spatial convolution and a 1D temporal convolution. As a side note, our cells choose to first apply temporal and then spatial attention, while S3D first applies spatial convolution and then temporal convolution.

### 4.5.7 Experimental Details

#### 4.5.7.1 Training and Inference

We conduct experiments on two benchmark datasets: Kinetics-600 [18] and Moments in Time (MiT) [102]. Kinetics-600 contains about 392K training videos and 30K validation videos from 600 classes. MiT consists of about 800K training videos and 34K validation videos from 339 classes.

After obtaining the attention cells found by our method, we fully train the backbone networks and cells on training videos and report their performance on validation videos. Following non-local blocks [154], we insert 5 cells or non-local blocks into the backbone. For I3D or S3D, they are inserted 5 inception modules (4a to 4e, see Table 1 in [138]). For I3D-R50, we insert them after 5 residual blocks, where 2 cells are inserted after every other residual block in $res_3$ and 3 cells are inserted after every other residual block in $res_4$.

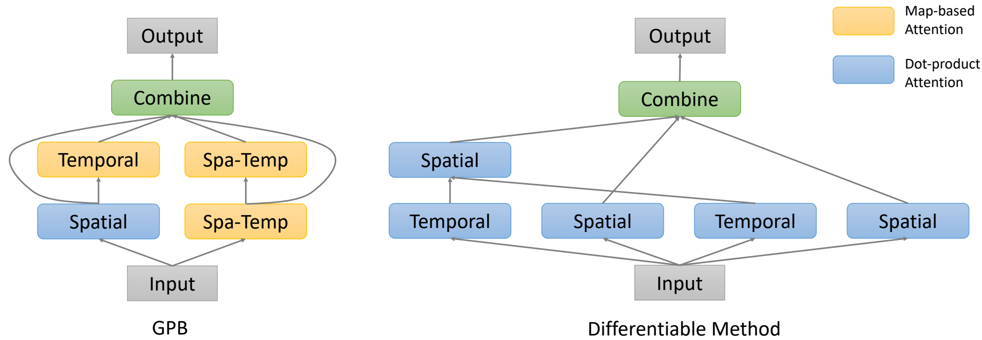



Figure 4.3: Visualization of the position-agnostic cell discovered by GPB and the differentiable method for I3D and on Kinetics-600. 'Spa-Temp' stands for the spatiotemporal attention dimension.

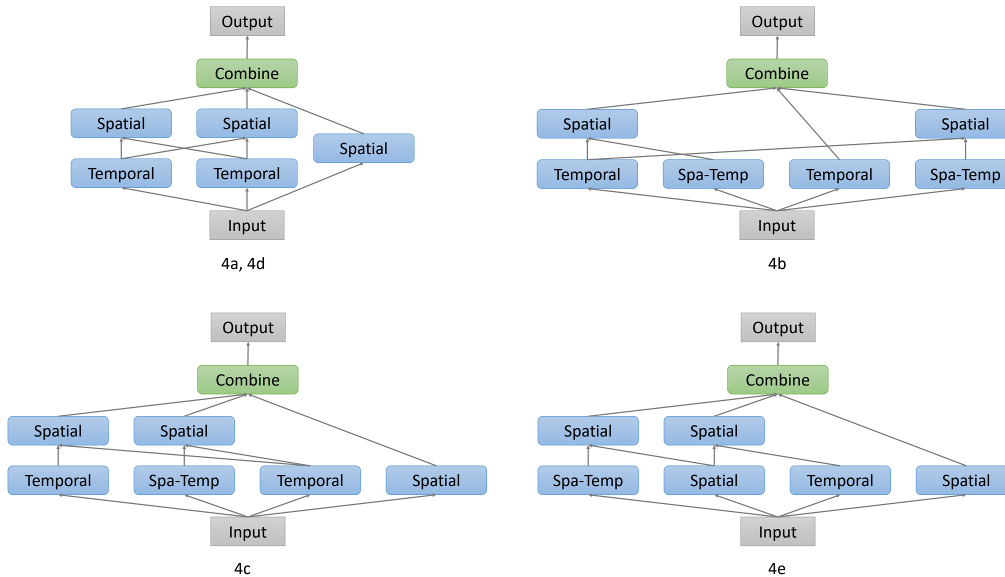



Figure 4.4: Visualization of the position-specific cells discovered by the differentiable method for I3D and on Kinetics-600. 'Spa-Temp' stands for the spatiotemporal attention dimension. The text under each cell indicates the inception module after which the cell is inserted (4a to 4e, see Table 1 in [138]) in the Inception network. The learned attention cell for 4a and 4d are the same.

During training, we resize the spatial resolution of videos to $256 \times 256$ and randomly crop $224 \times 224$ pixels or its horizontal flip from videos, for both Kinetics-600 [18] and MiT [102]. For I3D or S3D, we randomly crop 64 consecutive frames from the full-length video as the input clip during training. For I3D-R50, we ran-

domly crop 16 frames with stride 4 from the full-length video.

During inference, we perform fully-convolutional inference both spatially and temporally. We resize the spatial resolution to $256 \times 256$, pass the full-length video to the network, and predict the class based on the softmax scores. Our inference procedure does not require the sampling of multiple temporal clips and spatial crops in previous works [42]. The input clip to I3D or I3D has 250 frames for Kinetics-600 and has 75 frames for MiT. The input clip to I3D-50 has 64 frames for Kinetics-600 and has 18 frames for MiT, which is obtained by temporally downsampling the full-length video with stride 4.

We initialize the backbone in all the models (backbone, backbone + non-local blocks or our attention cells) with its ImageNet pre-trained weights. I3D or S3D based models are trained for 135 epochs, and I3D-R50 based models are trained for 150 epochs on Kinetics-600. All the models are trained for 45 epochs on MiT. We adopt a cosine learning rate schedule with a linear warm-up. The initial learning rate is 0.1 for I3D or S3D and 0.4 for I3D-R50. All the models are trained on 50 GPUs with synchronized SGD. The momentum is 0.9. The batch size per GPU is 6 for I3D or S3D and 4 for I3D-R50.

#### 4.5.7.2 Attention Cell Implementation

We have three pre-processing steps for the input to the entire attention cell: (1) channel reduction, (2) spatial resize, and (3) temporal grouping. These steps can not only reduce the computation consumed by the cell, but also allow the cell to process feature maps of different temporal and spatial resolutions.

Let $(B, T, H, W, C)$ be the shape of the input to the entire cell. We explicitly write out the batch size dimension $B$ for better explanation. We first reduce the number of channels from $C$ to $C_{\text{reduction}}$ with a $1 \times 1 \times 1$ convolutional layer. After channel reduction, the shape becomes $(B, T, H, W, C_{\text{reduction}})$. Then, we resize the spatial resolution of the feature map with bilinear interpolation from $(H, W)$ to $(H_{\text{resize}}, W_{\text{resize}})$, so the shape becomes $(B, T, H_{\text{resize}}, W_{\text{resize}}, C_{\text{reduction}})$. Finally, we divide the feature map into multiple groups of $T_{\text{group}}$ frames and obtain a feature map of shape $(nB, T_{\text{group}}, H_{\text{resize}}, W_{\text{resize}}, C_{\text{reduction}})$, where $T = n \cdot T_{\text{group}}$ and zero padding frames are added when necessary. After temporal grouping, the feature map of shape $(nB, T_{\text{group}}, H_{\text{resize}}, W_{\text{resize}}, C_{\text{reduction}})$ is then passed to attention operations in the cell. During the combination procedure, we resize the spatial resolution back to $(H, W)$ and merge temporal groups back to $T$ frames.

It is not difficult to see that these steps can reduce the computation. We elaborate on the second advantage. Note that the temporal and spatial resolution of test videos can vary (e.g., $250 \times 256 \times 256$) and be different from sampled training clips (e.g., $64 \times 224 \times 224$). This causes the shape of the feature map output by each layer to be different between training and test. However, temporal attention requires the spatial resolution of the feature map to be fixed and spatial attention requires the number of frames to be fixed. To address this issue, we adopt these pre-

processing steps so that the input to attention operations always has a fixed shape of $(T_{\text{group}}, H_{\text{resize}}, W_{\text{resize}}, C_{\text{reduction}})$.

#### 4.5.7.3 Search Algorithm

**GPB.** We sample training videos from the original dataset as the search-train and search-validation split. No validation videos are used during the search. We maximize the validation performance using GPB. We set the number of trails of GPB to 50, i.e., 50 attention cells are sampled by GPB and evaluated on the search-validation split after trained on the search-train split. Both the search-train split for Kinetics-600 and MiT contain about 360K videos. We train for 60 epochs for Kinetics-600 and 20 epochs for MiT during the search on their corresponding search-train split. We set $K = 4$ and search for an attention cell consisting of 4 attention operations. We use GPB to find one position-agnostic attention cell and insert the same cell architecture at different positions in the backbone network. To simplify the search space explored by GPB, we restrict the $k^{th}$ operation to select only one feature from $\{f_0, f_1, \ldots, f_{k-1}\}$ as its input.

**Differentiable Method.** When using the differentiable search method, we consider a supergraph consisting of 2 levels. Each level in the supergraph has 6 nodes. We do not include more nodes in one level due to the GPU memory constraint. At eacl level, we repeat each attention dimension twice and only include dot-product attention. So the 6 nodes are 2 temporal dot-product, 2 spatial dot-product, and 2 spatiotemporal dot-product attention operations. We also fix that the keys and values of dot-product attention are computed based on the attention cell input (see Eq. 4.5). This is the default supergraph design. We compare different supergraph designs in Section 4.5.5.

The connection weights and the network weights are learned jointly on training videos. The entire search process of the differentiable method consumes a computational cost similar to fully training one network on the training videos. For example, training I3D with the found attention cells on Kinetics-600 takes about 2.5 days. Searching attention cells for I3D, i.e., training I3D with supergraphs, takes about 3.5 days on Kinetics-600. The increase in the time is due to that supergraphs consume more computation than the final attention cells.

When deriving the attention cell design from the learned connection weights, the hyper-parameters $\alpha$ and $\beta$ are set to $\alpha = 3, \beta = 2$. Attention cells found by the differentiable method do not have a fixed number of operations, which are determined by the connection weights and $\alpha$ and $\beta$. Each operation may receive up to $\beta$ feature maps and computes a weighted sum of these feature maps as its input. We slightly revisit the combination procedure for cells found by the differentiable method. Instead of combining all the operation output feature maps, we only combine the output of the top $\alpha$ nodes (operations) with the highest weights in $w^{\text{sink}}$.

Table 4.8: Inference FLOPs on Kinetics-600.

| Model | Top-1 | Top-5 | GFLOPs |
|---|---|---|---|
| I3D [19] | 75.58 | 92.93 | 1136 |
| I3D+NL [154] | 76.87 | 93.44 | 1305 |
| **I3D+Cell** | 77.86 | 93.75 | 1170 |
| S3D [167] | 76.15 | 93.22 | 656 |
| S3D+NL [154] | 77.56 | 93.68 | 825 |
| **S3D+Cell** | 78.51 | 93.88 | 692 |
| I3D-R50 [154] | 78.10 | 93.79 | 938 |
| **I3D-R50+Cell** | 79.83 | 94.37 | 1034 |

#### 4.5.7.4 Comparison of FLOPs

We compare the inference FLOPs of all the models on Kinetics-600 in Table 4.8. Note that although our cells contain multiple operations, the aforementioned preprocessing applied on the cell input can effectively reduce the FLOPs consumed by attention operations. As shown in Table 4.8, our cells only add a small amount of computation to the backbone network and use fewer FLOPs than non-local blocks. The FLOPs are computed when the input clip has 250 frames with spatial resolution $256 \times 256$.

## 4.6 Conclusion

We propose a novel search space for spatiotemporal attention cells for the application of video classification. We also propose a differentiable formulation of the search space, allowing us to learn position-specific attention cell designs with zero extra cost compared to learning a single position-agnostic attention cell. We show the significance of our discovered attention cells on two large-scale video classifications benchmarks. The discovered attention cells also outperform non-local blocks and demonstrate strong generalization performance when being applied to different modalities, backbones, or datasets.

# Chapter 5

# Wisdom of Committees: An Overlooked Approach to Faster and More Accurate Models

## 5.1 Introduction

Optimizing the efficiency of neural networks is important for real-world applications as they can only use limited computational resources and often have requirements on response time. There has been considerable work in this direction [59, 140, 183], but they mostly focus on designing novel network architectures that can achieve a favorable speed-accuracy trade-off. Here, we do not present any novel method or architecture design. Instead, we focus on analyzing the accuracy and efficiency of a simple paradigm: committee-based models. We use the term "committee" to refer to model ensembles or cascades, which indicates that they are built using *multiple* independent models.

Committee-based models have been extensively studied and used before deep learning [14, 45, 121, 147]. However, when comparing the efficiency of deep models, committee-based models are rarely considered in recent work [59, 140, 183]. There still lacks a systematic understanding of their efficiency in comparison with single models – models that only use one network. Such an understanding is informative for both researchers to push the frontier of efficient models and practitioners to select model designs in real-world applications.

To fill this knowledge gap, we conduct a comprehensive analysis of the efficiency of committee-based models. To highlight the practical benefit of committee-based models, we intentionally choose the simplest possible method, which directly uses off-the-shelf, independently pre-trained models to build ensembles or cascades. We ensemble multiple pre-trained models via a simple average over their predictions (Sec. 5.3). For cascades, we sequentially apply each model and use a simple heuristic (*e.g.*, maximum probability in the prediction) to determine when to exit from

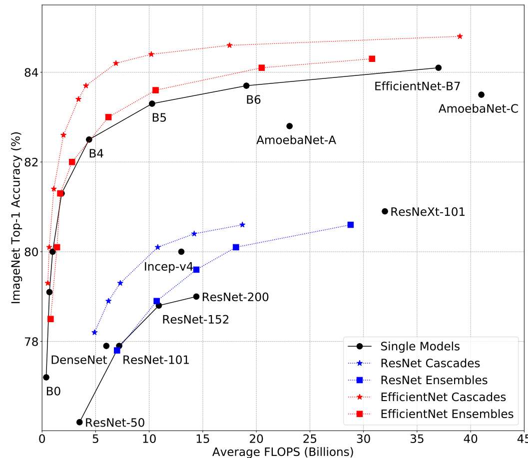



Figure 5.1: Committee-based models achieve a higher accuracy than single models on ImageNet while using fewer FLOPs. For example, although Inception-v4 ('Incep-v4') outperforms all single ResNet models, a ResNet cascade can still outperform Incep-v4 with fewer FLOPs.

the cascade (Sec. 5.4).

We show that even this method already outperforms state-of-the-art architectures found by costly neural architecture search (NAS) methods. Note that this method works with off-the-shelf models and does not use specialized techniques. For example, it differs from Boosting [121] where each new model is conditioned on previous ones, and does not require the weight generation mechanism in previous efficient ensemble methods [156]. This method does not require the training of an early exit policy [13, 49] or the specially designed multi-scale architecture [61] in previous work on building cascades.

To be clear, the contribution of this paper is not in the invention of model ensembles and cascades, as they have been known for decades, and is not in a new proposed method to build them. Instead, it is in the thorough evaluation and comparison of committee-based models with commonly used model architectures. Our analysis shows that committee-based models provide a simple complementary paradigm to achieve superior efficiency without tuning the architecture. One can often improve accuracy while reducing inference and training cost by building committees out of existing networks.

Our findings generalize to a wide variety of tasks, including image classification, video classification, and semantic segmentation, and hold true for various architecture families: ViT [37], EfficientNet [140], ResNet [54], MobileNetV2 [120], and X3D [41]. We summarize our findings as follows:

- Ensembles are more cost-effective than a single model in the large computation regime (Sec. 5.3). For example, an ensemble of two separately trained

EfficientNet-B5 models matches B7 accuracy, a state-of-the-art ImageNet model, while having almost 50% less FLOPs (20.5B vs. 37B).

- Cascades outperform single models in *all* computation regimes (Sec. 5.4&5.5). Our cascade matches B7 accuracy while using on average 5.4x fewer FLOPs. Cascades can also achieve a 2.3x speedup over ViT-L-384, a Transformer architecture, while matching its accuracy on ImageNet.

- We further show that (1) the efficiency of cascades is evident in both FLOPs and on-device latency and throughput (Sec. 5.5.1); (2) cascades can provide a guarantee on worst-case FLOPs (Sec. 5.5.2); (3) one can build self-cascades using a single model with multiple inference resolutions to achieve a significant speedup (Sec. 5.7).

- Committee-based models are applicable beyond image classification (Sec. 5.8) and outperform single models on the task of video classification and semantic segmentation. Our cascade outperforms X3D-XL by 1.2% on Kinetics-600 [18] while using fewer FLOPs.

## 5.2 Related Work

**Efficient Neural Networks.** There has been significant progress in designing efficient neural networks. In early work, most efficient networks, such as MobileNet [59, 120] and ShuffleNet [58], were manually designed. Recent work started to use neural architectures search (NAS) to automatically learn efficient network designs [17, 22, 139, 140, 190]. They mostly fcous on improving the efficiency of single models by designing better architectures, while we explore committee-based models without tuning the architecture.

**Ensembles.** Ensemble learning has been well studied in machine learning and there have been many seminal works, such as Bagging [14], Boosting [121], and AdaBoost [45]. Ensembles of neural networks have been used for many tasks, such as image classification [62, 138], machine translation [156], active learning [7], and out-of-distribution robustness [44, 75, 157]. But the efficiency of model ensembles has rarely been systematically investigated. Recent work indicated that ensembles can be more efficient than single models for image classification [71, 94]. Our work further substantiates this claim through the analysis of modern architectures on large-scale benchmarks.

**Cascades.** A large family of works have explored using cascades to speed up certain tasks. For example, the seminal work from [147] built a cascade of increasingly complex classifiers to speed up face detection. Cascades have also been explored in the context of deep neural networks. [13] reduced the average test-time cost by

learning a policy to allow easy examples to early exit from a network. A similar idea was also explored by [49]. [61] proposed a specially designed architecture Multi-Scale DenseNet to better incorporate early exits into neural networks. Given a pool of models, [133] presented an approximation algorithm to produce a cascade that can preserve accuracy while reducing FLOPs and demonstrated improvement over state-of-the-art NAS-based models on ImageNet. Different from previous work that primarily focuses on developing new methods to build cascades, we show that even the most straightforward method can already provide a significant speedup without training an early exit policy [13, 49] or designing a specialized multi-scale architecture [61].

**Dynamic Neural Networks.** Dynamic neural networks allocate computational resources based on the input example, *i.e.*, spending more computation on hard examples and less on easy ones [52]. For example, [122] trained a gating network to determine what parts in a high-capacity model should be used for each example. Recent work [146, 155, 163] explored learning a policy to dynamically select layers or blocks to execute in ResNet based on the input image. Our analysis shows that cascades of pre-trained models are actually a strong baseline for dynamic neural networks.

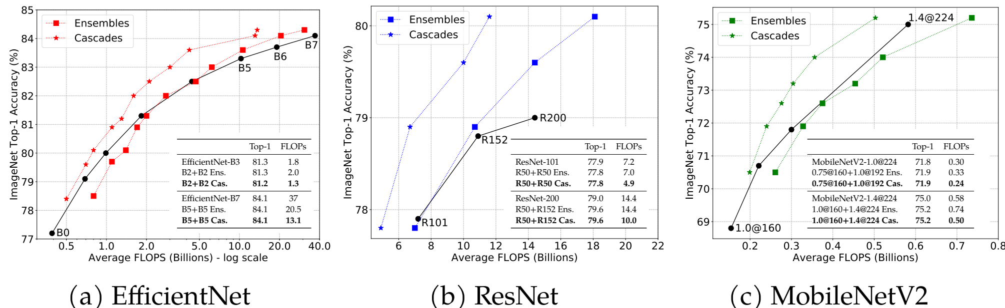
## 5.3 Ensembles are Accurate, Efficient, and Fast to Train

Model ensembles are useful for improving accuracy, but the usage of multiple models also introduces extra computational cost. When the total computation is fixed, which one will give a higher accuracy: single models or ensembles? The answer is important for real-world applications but this question has rarely been systematically studied on modern architectures and large-scale benchmarks.

We investigate this question on ImageNet [118] with three architecture families: EfficientNet [140], ResNet [54], and MobileNetV2 [120]. Each architecture family contains a series of networks with different levels of accuracy and computational cost. Within each family, we train a pool of models, compute the ensemble of different combinations of models, and compare these ensembles with the single models in the family.

For EfficientNet, we consider ensembles of two to four models of either the same or different architectures. Note that we only try different combinations of architectures used in the ensemble, but not the combinations of models. For example, when an ensemble contains an EfficientNet-B5, while we have multiple B5 models available, we just randomly pick one but do not try all possible choices. The FLOPs range of ResNet or MobileNetV2 models is relatively narrow compared with EfficientNet, so we only consider ensembles of two models for ResNet and MobileNetV2.

We denote an ensemble of $n$ image classification models by $\{M_1, \ldots, M_n\}$, where $M_i$ is the $i^{th}$ model. Given an image $x$, $\alpha_i = M_i(x)$ is a vector representing the




(a) EfficientNet (b) ResNet (c) MobileNetV2

Figure 5.2: Ensembles work well in the *large* computation regime and cascades show benefits in *all* computation regimes. These cascades are directly converted from ensembles without optimizing the choice of models (see Sec. 5.4). Black dots represent single models. **Ensembles:** Ensembles are more cost-effective than large single models, *e.g.*, EfficientNet-B5/B6/B7 and ResNet-152/200. **Cascades:** Converting ensembles to cascades significantly reduces the FLOPs without hurting the full ensemble accuracy (each star is on the left of a square).

logits for each class. To ensemble the $n$ models, we compute the mean of logits[1] $\alpha^{\text{ens}} = \frac{1}{n}\sum_i \alpha_i$ and predicts the class for image $x$ by applying argmax to $\alpha^{\text{ens}}$. The total computation of the ensemble is $\text{FLOPs}^{\text{ens}} = \sum_i \text{FLOPs}(M_i)$, where $\text{FLOPs}(\cdot)$ gives the FLOPs of a model.

We show the top-1 accuracy on ImageNet and FLOPs of single models and ensembles in Figure 5.2. Since there are many possible combinations of models to ensemble, we only show those Pareto optimal ensembles in the figure.

We see that ensembles are more cost-effective than large single models, *e.g.*, EfficientNet-B5/B6/B7 and ResNet-152/200. But in the small computation regime, single models outperform ensembles. For example, the ensemble of 2 B5 matches B7 accuracy while using about 50% less FLOPs. However, ensembles use more FLOPs than MobileNetV2 when they have a similar accuracy.

A possible explanation of why model ensembles are more powerful at large computation than at small computation comes from the perspective of bias-variance tradeoff. Large models usually have small bias but large variance, where the variance term dominates the test error. Therefore, ensembles are beneficial at large computation as they can reduce the variance in prediction [14]. For small models, the bias term dominates the test error. Ensembles can reduce the variance, but this cannot compensate the fact that the bias of small models is large. Therefore, ensembles are less powerful at small computation.

Our analysis indicates that instead of using a large model, one should use an

[1]We note that the mean of probabilities is a more general choice since logits can be arbitrarily scaled. In our experiments, we observe that they yield similar performance with the mean of logits being marginally better. The findings in our work hold true no matter which choice is used.

Table 5.1: Training time (TPUv3 days) of EfficientNet.

| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 |
|---|---|---|---|---|---|---|---|
| 9 | 12 | 15 | 24 | 32 | 48 | 128 | 160 |

Table 5.2: Training time (TPU v3 days) of ensembles. We use the '+' notation to indicate the models in enmsebles. Ensembles are faster than single models in both training and inference while achieve a similar accuracy.

| | Top-1 (%) | FLOPs (B) | Training |
|---|---|---|---|
| B6 | 83.7 | 19.1 | 128 |
| B3+B4+B4 | 83.6 | 10.6 | 88 |
| B7 | 84.1 | 37 | 160 |
| B5+B5 | 84.1 | 20.5 | 96 |
| B5+B5+B5 | 84.3 | 30.8 | 144 |

ensemble of multiple relatively smaller models, which would give similar performance but with fewer FLOPs. In practice, model ensembles can be easily parallelized (*e.g.*, using multiple accelerators), which may provide further speedup for inference. Moreover, often the total training cost of an ensemble is much lower than that of an equally accurate single model as shown below.

### 5.3.1 Training Time of Ensembles

We now show that the total training cost of an ensemble if often lower than an equally accurate single model. We show the training time of single EfficinetNet models in Table 5.1. We use 32 TPUv3 cores to train B0 to B5, and 128 TPUv3 cores to train B6 or B7. All the models are trained with the public official implementation of EfficientNet. We choose the ensemble that matches the accuracy of B6 or B7 and compute the total training time of the ensemble based on Table 5.1. As shown in Table 5.2, the ensemble of 2 B5 can match the accuracy of B7 while being faster in both training and inference.

## 5.4 From Ensembles to Cascades

In the above we have identified the scenarios where ensembles outperform or underperform single models. Specifically, ensembles are not an ideal choice when only a small amount of computation is allowed. In this section, we show that by simply converting an ensemble to a cascade, one can significantly reduce the computation and outperform single models in all computation regimes.

**Algorithm 4** Cascades

**Input**: Models $\{M_i\}$. Thresholds $\{t_i\}$. Test image $x$.
**for** $k = 1, 2, \ldots, n$ **do**
  $\alpha^{\text{cas}} = \frac{1}{k}\sum_{i=1}^{k} \alpha_i = \frac{1}{k}\sum_{i=1}^{k} M_i(x)$
  $\text{FLOPs}^{\text{cas}} = \sum_{i=1}^{k} \text{FLOPs}(M_i)$
  Early exit if confidence score $g(\alpha^{\text{cas}}) \geq t_k$
**end for**
Return $\alpha^{\text{cas}}$ and $\text{FLOPs}^{\text{cas}}$

Applying an ensemble is wasteful for easy examples where a subset of models will give the correct answer. Cascades save computation via early exit - potentially stopping and outputting an answer before all models are used. The total computation can be substantially reduced if we accurately determine when to exit from cascades. For this purpose, we need a function to measure how likely a prediction is correct. This function is termed *confidence* (more details in Sec. 5.4.1). A formal procedure of cascades is provided in Algorithm 4. Note that our cascades also average the predictions of the models having been used so far. So for examples where all models are used, the cascade effectively becomes an ensemble.

### 5.4.1 Confidence Function

Let $g(\cdot)\colon \mathbb{R}^N \rightarrow \mathbb{R}$ be the confidence function, which maps maps predicted logits $\alpha$ to a confidence score. The higher $g(\alpha)$ is, the more likely the prediction $\alpha$ is correct. Previous work [64, 133] tried several simple metrics to indicate the prediction confidence, such as the the maximum probability in the predicted distribution, the gap between the top-2 logits or probabilities, and the (negative) entropy of the distribution. As shown in Figure 5.3, all the metrics demonstrate reasonably good performance on measuring the confidence of a prediction, *i.e.*, estimating how likely a prediction is correct. In the following experiments, we adopt the maximum probability metric, *i.e.*, $g(\alpha) = \max(\text{softmax}(\alpha))$[2].

For a cascade of $n$ models $\{M_i\}$, we also need $(n-1)$ thresholds $\{t_i\}$ on the confidence score, where we use $t_i$ to decide whether a prediction is confident enough to exit after applying model $M_i$ (see Algorithm 4). As we define $g(\cdot)$ as the maximum probability, $t_i$ is in $[0, 1]$. A smaller $t_i$ indicates more images will be passed to the next model $M_{i+1}$. A cascade will reduce to an ensemble if all the thresholds $\{t_i\}$ are set to 1. $t_n$ is unneeded, since the cascade will stop after applying the last model $M_n$, no matter how confident the prediction is.

We can flexibly control the trade-off between the computation and accuracy of

[2]As a side observation, when analyzing the confidence function, we notice that models in our experiments are often slightly underconfident. This contradicts the common belief that deep neural networks tend to be overconfident [51].

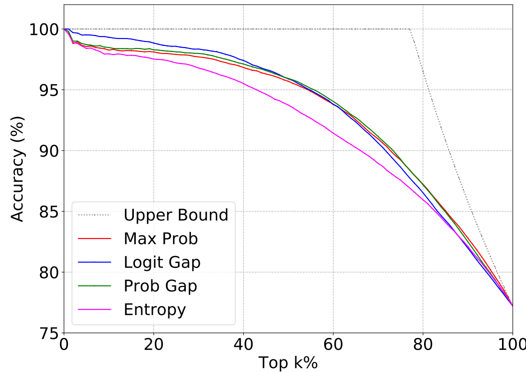



Figure 5.3: Different metrics for the confidence function. For a EfficientNet-B0 model, we select the top-$k\%$ validation images with highest confidence scores and compute the classification accuracy within the selected images. The higher the accuracy is at a certain $k$, the better the confidence metric is. All the metrics perform similarly in estimating how likely a prediction is correct.

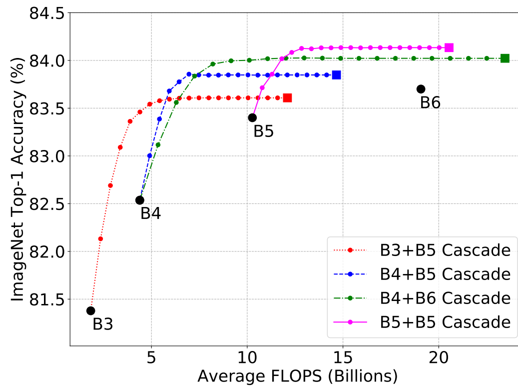



Figure 5.4: Cascades with different confidence thresholds. Each black dot is a single model and each square is an ensemble of models. Each colored dot represents a cascade with a specific $t_1 (0 \leq t_1 \leq 1)$. As $t_1$ increases from 0 to 1, the cascade uses more and more computation and changes from a single model (first model in the cascade; $t_1 = 0$) to the ensemble ($t_1 = 1$). The plateau in each curve indicates that the cascade can achieve a similar accuracy to the ensemble with much less computation.

a cascade through thresholds $\{t_i\}$. To understand how the thresholds influence a cascade, we visualize several 2-model cascades in Figure 5.4. For each cascade, we sweep $t_1$ from 0 and 1 and plot the results. Note that all the curves in Figure 5.4 have a plateau, indicating that we can significantly reduce the average FLOPs without hurting the accuracy if $t_1$ is properly chosen. We select the thresholds $\{t_i\}$ on

held-out validation images according to the target FLOPs or validation accuracy. In practice, we find such thresholds via grid search. Note that the thresholds are determined after all models are trained. We only need the logits of validation images to determine $\{t_i\}$, so computing the cascade performance for a specific choice of thresholds is fast, which makes grid search computationally possible.

### 5.4.2 Converting Ensembles to Cascades

For each ensemble in Figure 5.2, we convert it to a cascade that uses the same set of models. During conversion, we set the confidence thresholds such that the cascade performs similar to the ensemble while the FLOPs are minimized. By design in cascades some inputs incur more FLOPs than others. So we report the average FLOPs computed over all images in the test set.

We see that cascades consistently use less computation than the original ensembles and outperform single models in all computation regimes and for all architecture families. Taking 2 EfficientNet-B2 as an example (see Figure 5.2a), the ensemble initially obtains a similar accuracy to B3 but uses more FLOPs. After converting this ensemble to a cascade, we successfully reduce the average FLOPs to 1.3B (1.4x speedup over B3) and still achieve B3 accuracy. Cascades also outperform small MobileNetV2 models in Figure 5.2c.

## 5.5 Model Selection for Building Cascades

The cascades in Figure 5.2 do not optimize the choice of models and directly use the set of models in the original ensembles. For best performance, we show that one can design cascades to match a specific target FLOPs or accuracy by selecting models to be used in the cascade.

Let $\mathcal{M}$ be the set of available models, *e.g.*, models in the EfficientNet family. Given a target FLOPs $\beta$, we select $n$ models $M = \{M_i \in \mathcal{M}\}$ and confidence thresholds $T = \{t_i\}$ by solving the following problem:

$$\begin{aligned} \max_{\{M_i \in \mathcal{M}\}, \{t_i\}} \quad & \text{Accuracy}\left(\mathcal{C}\left(M, T\right)\right) \\ s.t. \quad & \text{FLOPs}\left(\mathcal{C}\left(M, T\right)\right) \leq \beta, \end{aligned} \tag{5.1}$$

where $\mathcal{C}\left(M, T\right)$ is the cascade of models $\{M_i\}$ with thresholds $\{t_i\}$, $\text{Accuracy}(\cdot)$ gives the validation accuracy of a cascade, and $\text{FLOPs}(\cdot)$ gives the average FLOPs. Similarly, we can also build a cascade to match a target validation accuracy $\gamma$ by minimizing $\text{FLOPs}\left(\mathcal{C}\left(M, T\right)\right)$ with the constraint $\text{Accuracy}\left(\mathcal{C}\left(M, T\right)\right) \geq \gamma$.

Note that this optimization is done after all models in $\mathcal{M}$ were independently trained. The optimization complexity is exponential in $|\mathcal{M}|$ and $n$, and the problem will be challenging if $|\mathcal{M}|$ and $n$ are large. In our experiments, $|\mathcal{M}|$ and $n$ are not

prohibitive. Therefore, we solve the optimization problem with exhaustive search. One can also use more efficient procedures such as the algorithm described in [133].

Same as our analysis of ensembles, we do not search over different models of the same architecture, but only search combination of architectures. Therefore, for EfficientNet, $|\mathcal{M}| = 8$ and $n \leq 4$ and we have in total $4672 = (8^4 + 8^3 + 8^2)$ possible combinations of models. Note that the search is cheap to do as it is conducted after all the models are independently trained. No GPU training is involved in the search. We pre-compute the predictions of each model on a held-out validation set before search. During the search, we try possible models combinations by loading their predictions. We can usually find optimal model combinations within a few CPU hours.

In practice, we first train each EfficientNet model separately for 4 times and pre-compute their predicted logits. Then for each possible combination of models, we load the logits of models and determine the thresholds according to the target FLOPs or accuracy. Finally, we choose the best cascade among all possible combinations. Similar as above, we choose models and thresholds on held-out training images for ImageNet experiments. No images from the ImageNet validation set are used when we select models for a cascade.

For ResNet and MobileNetV2, we only tried 2-model cascades due to their relatively narrow FLOPs range. Therefore, the number of possible model combinations is very small ($< 20$). For ViT, we only tried 2 cascades: ViT-B-224 + ViT-L-224 and ViT-B-384 + ViT-L-384.

### 5.5.1 Targeting for a Specific FLOPs or Accuracy

For each single EfficientNet, ResNet or MobileNetV2, we search for a cascade to match its FLOPs (red squares in Figure 5.5a-5.5d) or its accuracy (green squares in Figure 5.5a-5.5d). Notably, in addition to convolutional networks, we also consider a Transformer architecture – ViT [37]. We build a cascade of ViT-Base and ViT-Large to match the cost or accuracy of ViT-Large (Figure 5.5e). For ViT, we measure the speedup in throughput (more details on throughput below).

When building cascades, we consider all networks in the same family as the set of available models. The same model type is allowed to be used for multiple times in a cascade but they will be different models trained separately. For ImageNet experiments, the search is conducted on a small set of held-out training images and cascades are evaluated on the original validation set.

Results in Figure 5.5 further substantiate our finding that cascades are more efficient than single models in all computation regimes. For small models, we can outperform MobileNetV2-1.0@224 by 1.4% using equivalent FLOPs. For large models, we can obtain 2.3x speedup over ViT-L-384 and 5.4x over EfficientNet-B7 while matching their accuracy.

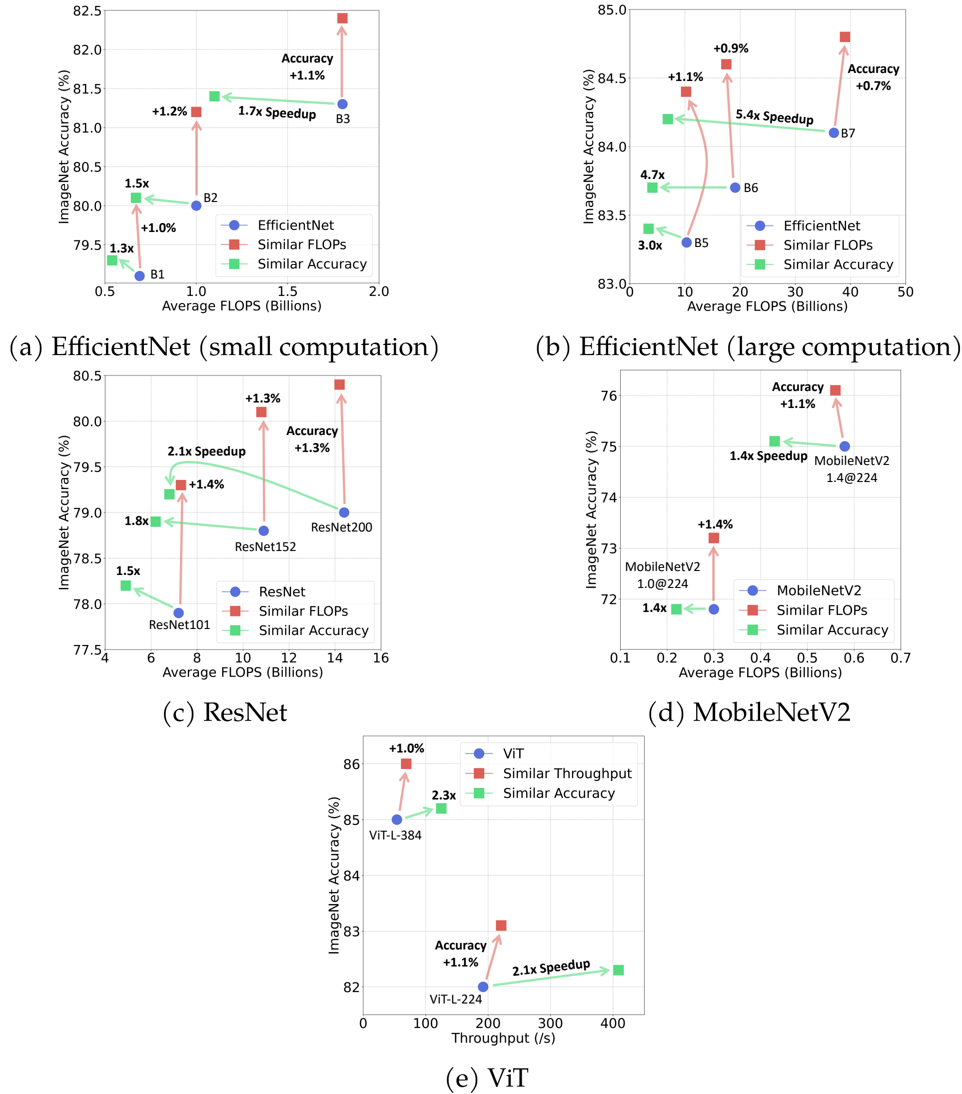



Figure 5.5: Cascades of EfficientNet, ResNet, MobileNetV2 or ViT models on ImageNet. Compared with single models, cascades can obtain a higher accuracy with similar cost (red squares) or achieve a significant speedup while being equally accurate (green squares; *e.g.*, 5.4x speedup for B7). **The benefit of cascades generalizes to all four architecture families and all computation regimes.** Numerical results are also available in Table 5.3&5.4.

#### 5.5.1.1 On-device Latency and Throughput

In the above, we mostly use average FLOPs to measure the computational cost. We now report the latency and throughput of cascades on TPUv3 in Table 5.5&5.6 to

Table 5.3: Cascades of EfficientNet, ResNet or MobileNetV2 models on ImageNet. This table contains the numerical results for Figure 5.5a-5.5d. **Middle**: Cascades obtain a higher accuracy than single models when using similar FLOPs. **Right**: Cascades achieve a similar accuracy to single models with significantly fewer FLOPs (*e.g.*, 5.4x fewer for B7). **The benefit of cascades generalizes to all three convolutional architecture families and all computation regimes.**

| | Single Models | | Cascades - Similar FLOPs | | | Cascades - Similar Accuracy | | |
|---|---|---|---|---|---|---|---|---|
| | Top-1 (%) | FLOPs (B) | Top-1 (%) | FLOPs (B) | ΔTop-1 | Top-1 (%) | FLOPs (B) | Speedup |
| **EfficientNet** | | | | | | | | |
| B1 | 79.1 | 0.69 | **80.1** | 0.67 | **1.0** | 79.3 | **0.54** | **1.3x** |
| B2 | 80.0 | 1.0 | **81.2** | 1.0 | **1.2** | 80.1 | **0.67** | **1.5x** |
| B3 | 81.3 | 1.8 | **82.4** | 1.8 | **1.1** | 81.4 | **1.1** | **1.7x** |
| B4 | 82.5 | 4.4 | **83.7** | 4.1 | **1.2** | 82.6 | **2.0** | **2.2x** |
| B5 | 83.3 | 10.3 | **84.4** | 10.2 | **1.1** | 83.4 | **3.4** | **3.0x** |
| B6 | 83.7 | 19.1 | **84.6** | 17.5 | **0.9** | 83.7 | **4.1** | **4.7x** |
| B7 | 84.1 | 37 | **84.8** | 39.0 | **0.7** | 84.2 | **6.9** | **5.4x** |
| **ResNet** | | | | | | | | |
| R101 | 77.9 | 7.2 | **79.3** | 7.3 | **1.4** | 78.2 | **4.9** | **1.5x** |
| R152 | 78.8 | 10.9 | **80.1** | 10.8 | **1.3** | 78.9 | **6.2** | **1.8x** |
| R200 | 79.0 | 14.4 | **80.4** | 14.2 | **1.3** | 79.2 | **6.8** | **2.1x** |
| **MobileNetV2** | | | | | | | | |
| 1.0@160 | 68.8 | 0.154 | **69.5** | 0.153 | **0.6** | 69.1 | **0.146** | **1.1x** |
| 1.0@192 | 70.7 | 0.22 | **71.8** | 0.22 | **1.1** | 70.8 | **0.18** | **1.2x** |
| 1.0@224 | 71.8 | 0.30 | **73.2** | 0.30 | **1.4** | 71.8 | **0.22** | **1.4x** |
| 1.4@224 | 75.0 | 0.58 | **76.1** | 0.56 | **1.1** | 75.1 | **0.43** | **1.4x** |

Table 5.4: Cascades of ViT models on ImageNet. This table contains the numerical results for Figure 5.5e. 224 or 384 indicates the image resolution the model is trained on. Throughput is measured on NVIDIA RTX 3090. Our cascades can achieve a 1.0% higher accuracy than ViT-L-384 with a similar throughput or achieve a 2.3x speedup over it while matching its accuracy. **The benefit of cascades generalizes to Transformer architectures.**

| | Single Models | | Cascades - Similar Throughput | | | Cascades - Similar Accuracy | | |
|---|---|---|---|---|---|---|---|---|
| | Top-1 (%) | Throughput (/s) | Top-1 (%) | Throughput (/s) | ΔTop-1 | Top-1 (%) | Throughput (/s) | Speedup |
| ViT-L-224 | 82.0 | 192 | **83.1** | 221 | **1.1** | 82.3 | **409** | **2.1x** |
| ViT-L-384 | 85.0 | 54 | **86.0** | 69 | **1.0** | 85.2 | **125** | **2.3x** |

confirm that the reduction in FLOPs can translate to the real speedup on hardware. The latency or throughput of a model is highly dependent on the batch size. So we consider two scenarios: (1) online processing, where we use a fixed batch size 1, and (2) offline processing, where we can batch the examples.

**Online Processing.** Cascades are useful for online processing with a fixed batch size 1. Using batch size 1 is sub-optimal for the utilization of accelerators like GPU or TPU, but it still happens in some real-world applications, *e.g.*, mobile phone cam-

Table 5.5: Average latency on TPUv3 for the case of online processing with batch size 1. Cascades are much faster than single models in terms of the average latency while being similarly accurate.

| | Top-1 (%) | Latency (ms) | Speedup |
|---|---|---|---|
| B1 | 79.1 | 3.7 | |
| Cascade* | 79.3 | **3.0** | **1.2x** |
| B2 | 80.0 | 5.2 | |
| Cascade* | 80.1 | **3.7** | **1.4x** |
| B3 | 81.3 | 9.7 | |
| Cascade* | 81.4 | **5.9** | **1.7x** |
| B4 | 82.5 | 16.6 | |
| Cascade* | 82.6 | **9.6** | **1.7x** |
| B5 | 83.3 | 27.2 | |
| Cascade* | 83.4 | **14.3** | **1.9x** |
| B6 | 83.7 | 57.1 | |
| Cascade* | 83.7 | **15.1** | **3.8x** |
| B7 | 84.1 | 126.6 | |
| Cascade* | 84.2 | **23.2** | **5.5x** |

* The cascade that matches the accuracy of EfficientNet-B1 to B7 in Figure 5.5 or the right column of Table 5.3.

eras processing a single image [148] or servers that need to rapidly return the result without waiting for enough queries to form a batch. We report the average latency of cascades on TPUv3 with batch size 1 in Table 5.5. Cascades are up to 5.5x faster than single models with comparable accuracy.

**Offline Processing.** Cascades are also useful for offline processing of large-scale data. For example, when processing all frames in a large video dataset, we can first apply the first model in the cascade to all frames, and then select a subset of frames based on the prediction confidence to apply following models in the cascade. In this way all the processing can be batched to fully utilize the accelerators. We report the throughput of cascades on TPUv3 in Table 5.6, which is measured as the number of images processed per second. We use batch size 16 when running models on TPUv3 for the case of offline processing. As shown in Table 5.6, cascades achieve a much higher throughput than single models while being equally accurate. For clarification, only the throughput of ViT in Table 5.4 is measured on RTX 3090 while the throughput for other models is measured on TPUv3.

Table 5.6: Throughput on TPUv3 for the case of offline processing. Throughput is measured as the number of images processed per second. Cascades achieve a much larger throughput than single models while being equally accurate.

| | Top-1 (%) | Throughput (/s) | Speedup |
|---|---|---|---|
| B1 | 79.1 | 1436 | |
| Cascade* | 79.3 | **1798** | **1.3x** |
| B2 | 80.0 | 1156 | |
| Cascade* | 80.1 | **1509** | **1.3x** |
| B3 | 81.3 | 767 | |
| Cascade* | 81.4 | **1111** | **1.4x** |
| B4 | 82.5 | 408 | |
| Cascade* | 82.6 | **656** | **1.6x** |
| B5 | 83.3 | 220 | |
| Cascade* | 83.4 | **453** | **2.1x** |
| B6 | 83.7 | 138 | |
| Cascade* | 83.7 | **415** | **3.0x** |
| B7 | 84.1 | 81 | |
| Cascade* | 84.2 | **280** | **3.5x** |

* The cascade that matches the accuracy of EfficientNet-B1 to B7 in Figure 5.5 or the right column of Table 5.3.

Table 5.7: Comparison with SOTA NAS methods. Cascades outperform novel architectures found by costly NAS methods.

| | Top-1 (%) | FLOPs (B) |
|---|---|---|
| BigNASModel-L [178] | 79.5 | 0.59 |
| $OFA_{Large}$ [15] | 80.0 | 0.60 |
| Cream-L [106] | 80.0 | 0.60 |
| Cascade* | **80.1** | 0.67 |
| BigNASModel-XL [178] | 80.9 | 1.0 |
| Cascade* | **81.2** | 1.0 |

* The cascade that matches B1 or B2 FLOPs in Figure 5.5a.

#### 5.5.1.2 Comparison with NAS

We also compare with state-of-the-art NAS methods, *e.g.*, BigNAS [178], OFA [15] and Cream [106], which can find architectures better than EfficientNet. But as

Table 5.8: Cascades can be built with a guarantee on worst-case FLOPs. We use 'with' or 'w/o' to indicate whether a cascade can provide such a guarantee or not. Cascades with such a guarantee are assured to use fewer FLOPs than single models in the worst-case scenario, and also achieve a considerable speedup in average-case FLOPs.

| | Top-1 (%) | Average-case FLOPs (B) | Worst-case FLOPs (B) | Average-case Speedup | | Top-1 (%) | Average-case FLOPs (B) | Worst-case FLOPs (B) | Average-case Speedup |
|---|---|---|---|---|---|---|---|---|---|
| B5 | 83.3 | 10.3 | 10.3 | | B6 | 83.7 | 19.1 | 19.1 | |
| w/o* | 83.4 | 3.4 | 14.2 | 3.0x | w/o* | 83.7 | 4.1 | 25.9 | 4.7x |
| with | 83.3 | **3.6** | **9.8** | **2.9x** | with | 83.7 | **4.2** | **15.0** | **4.5x** |

* Cascades from Figure 5.5b.

shown in Table 5.7, a simple cascade of EfficientNet without tuning the architecture already outperforms these sophisticated NAS methods. The strong performance and simplicity of cascades should motivate future research to include them as a strong baseline when proposing novel architectures.

### 5.5.2 Guarantee on Worst-case FLOPs

Up until now we have been measuring the computation of a cascade using the average FLOPs across all images. But for some images, it is possible that all the models in the cascade need to be applied. In this case, the average FLOPs cannot fully indicate the computational cost of a cascade. For example, the cascade that matches B5 or B6 accuracy in Figure 5.5b has higher worst-case FLOPs than the comparable single models (see 'w/o' in Table 5.8). Therefore, we now consider worst-case FLOPs of a cascade, the sum of FLOPs of all models in the cascade.

We can easily find cascades with a guarantee on worst-case FLOPs by adding one more constraint: $\sum_i \mathrm{FLOPs}(M_i) \leq \beta^{\mathrm{worst}}$, where $\beta^{\mathrm{worst}}$ is the upper bound on the worst-case FLOPs of the cascade. With the new condition, we re-select models in the cascades to match of accuracy of B5 or B6. As shown in Table 5.8, compared with single models, the new cascades achieve a significant speedup in average-case FLOPs and also ensure its worst-case FLOPs are smaller. The new cascades with the guarantee on worst-case FLOPs are useful for applications with strict requirement on response time.

### 5.5.3 Exit Ratios

To better understand how a cascade works, we compute the exit ratio of the cascade, *i.e.*, the percentage of images that exit from the cascade at each stage. Specifically, we choose the cascades in Table 5.3&5.8 that match the accuracy of B1 to B7 and report their exit ratios in Table 5.9. For all the cascades in Table 5.9, most images only consume the cost of the first model in the cascade and only a few images have to use all the models. For example, the cascade above that matches B7 accuracy

Table 5.9: Exit ratios of cascades. We use the '+' notation to indicate the models in cascades.

| | Top-1 (%) | FLOPs (B) | Exit Ratio (%) at Each Stage | | | |
|---|---|---|---|---|---|---|
| | | | Model 1 | Model 2 | Model 3 | Model 4 |
| B1 | 79.1 | 0.69 | | | | |
| B0+B1 | 79.3 | 0.54 | 78.7 | 21.3 | | |
| B2 | 80.0 | 1.0 | | | | |
| B0+B1+B3 | 80.1 | 0.67 | 73.2 | 21.4 | 5.4 | |
| B3 | 81.3 | 1.8 | | | | |
| B0+B3+B3 | 81.4 | 1.1 | 68.0 | 26.4 | 5.7 | |
| B4 | 82.5 | 4.4 | | | | |
| B1+B3+B4 | 82.6 | 2.0 | 67.9 | 15.3 | 16.8 | |
| B5 | 83.3 | 10.3 | | | | |
| B2+B4+B4+B4 | 83.4 | 3.4 | 67.6 | 21.2 | 0.0 | 11.2 |
| B2+B4+B4* | 83.3 | 3.6 | 57.7 | 26.0 | 16.3 | |
| B6 | 83.7 | 19.1 | | | | |
| B2+B4+B5+B5 | 83.7 | 4.1 | 67.6 | 21.2 | 5.9 | 5.3 |
| B3+B4+B4+B4* | 83.7 | 4.2 | 67.3 | 16.2 | 10.9 | 5.6 |
| B7 | 84.1 | 37 | | | | |
| B3+B5+B5+B5 | 84.2 | 6.9 | 67.3 | 21.6 | 5.6 | 5.5 |

* Cascades from Table 5.8 with a guarantee on worst-case FLOPs.

contains four models: [B3, B5, B5, B5]. In this cascade, 67.3% images only consume the cost of B3 and only 5.5% images use all four models. This saves a large amount of computation compared with using B7 for all the images. This shows that cascades are able to allocate fewer resources to easy images and explains the speedup of cascades over single models.

### 5.5.4 Cascades can be Scaled Up

One appealing property of single models is that they can be easily scaled up or down based on the available computational resources one has. We show that such property is also applicable to cascades, *i.e.*, we can scale up a base cascade to respect different FLOPs constraints. This avoids the model selection procedure when designing cascades for different FLOPs, which is required for cascades in Table 5.3.

Specifically, we build a 3-model cascade to match the FLOPs of EfficientNet-B0. We call this cascade C0 (see below for details of building C0). Then, simply by scaling up the architectures in C0, we obtain a family of cascades C0 to C7 that have increasing FLOPs and accuracy. The models in C0 are from the EfficientNet family.

Table 5.10: A Family of Cascades C0 to C7. C0 to C7 significantly outperform single EfficientNet models in all computation regimes. C1 and C2 also compare favorably with state-of-the-art NAS methods, such as BigNAS [178], OFA [15] and Cream [106]. This shows that the cascades can also be scaled up or down to respect different FLOPs constraints as single models do. This is helpful for avoiding the model selection procedure when designing cascades for different FLOPs.

| Model | Top-1 (%) | FLOPs (B) | ΔTop-1 | Model | Top-1 (%) | FLOPs (B) | ΔTop-1 |
|---|---|---|---|---|---|---|---|
| **C0** | **78.1** | **0.41** | | **C3** | **82.2** | **1.8** | |
| EfficientNet-B0 | 77.1 | 0.39 | 1.0 | EfficientNet-B3 | 81.3 | 1.8 | 0.9 |
| **C1** | **80.3** | **0.71** | | **C4** | **83.7** | **4.2** | |
| EfficientNet-B1 | 79.1 | 0.69 | 1.2 | EfficientNet-B4 | 82.5 | 4.4 | 1.2 |
| BigNASModel-L | 79.5 | 0.59 | 0.8 | **C5** | **84.3** | **10.2** | |
| $\text{OFA}_{\text{Large}}$ | 80.0 | 0.60 | 0.3 | EfficientNet-B5 | 83.3 | 10.3 | 1.0 |
| Cream-L | 80.0 | 0.60 | 0.3 | **C6** | **84.6** | **18.7** | |
| **C2** | **81.2** | **1.0** | | EfficientNet-B6 | 83.7 | 19.1 | 0.9 |
| EfficientNet-B2 | 80.0 | 1.0 | 1.2 | **C7** | **84.8** | **32.6** | |
| BigNASModel-XL | 80.9 | 1.0 | 0.3 | EfficientNet-B7 | 84.1 | 37 | 0.7 |

The results of C0 to C7 in Table 5.10 show that simply scaling up C0 gives us a family of cascades that consistently outperform single models in all computation regimes. This finding enhances the practical usefulness of cascades as one can select cascades from this family based on available resources, without worrying about what models should be used in the cascade.

**Details of building C0.** The networks in EfficientNet family are obtained by scaling up the depth, width and resolution of B0. The scaling factors for depth, width and resolution are defined as $d = \alpha^\phi$, $w = \beta^\phi$ and $r = \gamma^\phi$, respectively, where $\alpha = 1.2$, $\beta = 1.1$ and $\gamma = 1.15$, as suggested in Tan et al. [140]. One can control the network size by changing $\phi$. For example, $\phi = 0$ gives B0, $\phi = 1$ gives B2, and $\phi = 7$ gives B7.

We build a 3-model cascade C0 to match the FLOPs of EfficientNet-B0 by solving Eq. 5.1 on held-out training images from ImageNet. When building C0, we consider 13 networks from EfficientNet family. As we want C0 to use similar FLOPs to B0, we make sure the 13 networks include both networks smaller than B0 and networks larger than B0. Their $\phi$ are set to -4.0, -3.0, -2.0, -1.0, 0.0, 0.25, 0.5, 0.75, 1.0, 1.25, 1.50, 1.75, 2.0, respectively.

The $\phi$ of the three models in C0 are -2.0, 0.0 and 0.75. Then simply scaling up the architectures in C0, *i.e.*, increasing the $\phi$ of each model in C0, gives us a family of cascades C0 to C7 that have increasing FLOPs and accuracy. The thresholds in C0 to C7 are determined such that their FLOPs are similar to B0 to B7.

## 5.6 Model Pool Details and Analysis

### 5.6.1 Details of ImageNet Models

In the previous analysis of the efficiency of ensembles or cascades on ImageNet, we consider four architecture families: EfficientNet [140], ResNet [54], MobileNetV2 [120], and ViT [37]. All the single models are independently trained with their original training procedure. We do not change the training schedule or any other hyper-parameters.

- The EfficientNet family contains 8 architectures (EfficientNet-B0 to B7). We train each architecture separately for 4 times with the official open-source implementation[3] provided by the authors. So, in total there are 32 EfficientNet models.
- For ResNet, we consider 4 architectures (ResNet-50/101/152/200) and train each architecture for 2 times using an open-source TPU implementation[4]. There are 8 ResNet models in total.
- For MobileNetV2, we directly download the pre-trained checkpoints from its official open-source implementation[5]. We use 5 MobileNetV2 models: MobileNetV2-0.75@160, 1.0@160, 1.0@192, 1.0@224, and 1.4@224. Each model is represented in the form of $w@r$, where $w$ is the width multiplier and $r$ is the image resolution.
- For ViT, we directly use the pre-trained checkpoints provided by the Hugging Face Team[6]. We use 4 ViT models: ViT-B-224, ViT-L-224, ViT-B-384, and ViT-L-384.

Training each EfficientNet architecture for 4 times (in total 32 models) may sound computationally expensive. We note that it is unnecessary to train each architecture for 4 times to find a well-performing ensemble or cascade. We train a large pool of EfficientNet models mainly for the purpose of analysis so that we can try a diverse range of model combinations, *e.g.*, the cascade of 4 EfficientNet-B5. We analyze the influence of the size and diversity of the model pool in the following text.

### 5.6.2 Number of Models in Cascades

We study the influence of the number of models in cascades on the performance. Concretely, we sweep the target FLOPs from 1 to 40 and find cascades of 2, 3 or 4 models from the EfficientNet family. As shown in Figure 5.6, the performance of

[3] https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet
[4] https://github.com/tensorflow/tpu/tree/master/models/official/resnet
[5] https://github.com/tensorflow/models/tree/master/research/slim/nets/mobilenet
[6] For example, ViT-B-224: https://huggingface.co/google/vit-base-patch16-224

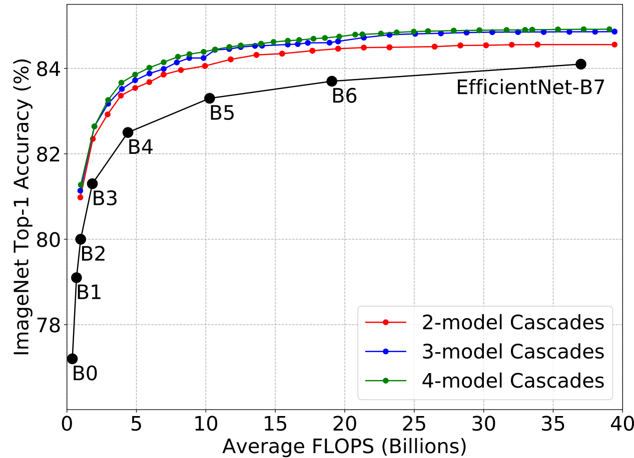



Figure 5.6: Impact of the number of models in cascades.

cascades keeps improving as the number of models increases. We see a big gap between 2-model cascades and 3-model cascades, but increasing the number of models from 3 to 4 demonstrates a diminishing return.

As mentioned above, for EfficientNet cascades, we tried in total $4672 = (8^4 + 8^3 + 8^2)$ possible combinations of models. Since 3-model cascades can obtain very close performance to 4-model cascades, one could try much fewer combinations to obtain similar results.

### 5.6.3 Size of the Model Pool

As mentioned above, we train each EfficientNet architecture for 4 times so that we can try a diverse range of model combinations. We now empirically show that naively adding more models of the same architecture to the pool only has a small influence on the performance of ensembles or cascades.

We train 8 EfficientNet-B5 models separately and build 2-B5 ensembles or cascades using any two of these models. The FLOPs of these 2-B5 ensembles are the same (20.5B). For each cascade, we tune the confidence threshold such that the cascade achieves a similar accuracy to the full ensemble. We show the max, min, mean, and standard deviation of the performance of these different ensembles or cascades in Table 5.11 and observe that the performance variation is small. Therefore, we conclude that adding more models of the same architecture only has modest influence on the performance.

### 5.6.4 Diversity of the Model Pool

We study the influence of the diversity of architectures in the model pool on the performance. We compare cascades of models of same architectures and cascades

Table 5.11: Max, min, mean, and standard deviation of the performance of 8 single B5 models, 28 possible 2-B5 ensembles, and 56 possible 2-B5 cascades.

| | max | min | mean | std |
|---|---|---|---|---|
| **Single Model** | | | | |
| Accuracy (%) | 83.40 | 83.29 | 83.34 | 0.04 |
| **2-B5 Ensembles** | | | | |
| Accuracy (%) | 84.18 | 83.97 | 84.10 | 0.05 |
| **2-B5 Cascades** | | | | |
| Accuracy (%) | 84.17 | 83.96 | 84.09 | 0.05 |
| FLOPs (B) | 13.35 | 12.32 | 12.62 | 0.29 |

Table 5.12: Cascades of models of same architectures vs. Cascades of models of different architectures. The '+' notation indicates the models used in cascades.

| | Top-1 (%) | FLOPs (B) | Speedup |
|---|---|---|---|
| B4 | 82.5 | 4.4 | |
| B3+B3+B3 | 82.6 | 2.7 | 1.6x |
| B1+B3+B4 | 82.6 | **2.0** | **2.2x** |
| B5 | 83.3 | 10.3 | |
| B4+B4 | 83.3 | 5.1 | 2.1x |
| B2+B4+B4+B4 | 83.4 | **3.4** | **3.0x** |
| B6 | 83.7 | 19.1 | |
| B4+B4+B4 | 83.8 | 6.0 | 3.2x |
| B2+B4+B5+B5 | 83.7 | **4.1** | **4.7x** |
| B7 | 84.1 | 37 | |
| B5+B5 | 84.1 | 13.1 | 2.8x |
| B3+B5+B5+B5 | 84.2 | **6.9** | **5.4x** |

of models of different architectures in Tables 5.12. As shown in Table 5.12, while cascades of same-architecture models can already significantly reduce the FLOPs compared with a similarly accurate single model, adding more variations in the architecture can significantly improve the performance of cascades.

## 5.7 Self-cascades

Cascades typically contain multiple models. This requires training multiple models and combining them after training. What about when only one model is available?

Table 5.13: Self-cascades. In the column of self-cascades, the two numbers represent the two resolutions $r_1$ and $r_2$ used in the cascade. Self-cascades use fewer FLOPs than comparable single models.

| EfficientNet | Top-1 (%) | FLOPs (B) | Self-cascades | Top-1 (%) | FLOPs (B) | Speedup |
|---|---|---|---|---|---|---|
| B2 | 80.0 | 1.0 | B1-240-300 | 80.1 | **0.85** | **1.2x** |
| B3 | 81.3 | 1.8 | B2-260-380 | 81.3 | **1.6** | **1.2x** |
| B4 | 82.5 | 4.4 | B3-300-456 | 82.5 | **2.7** | **1.7x** |
| B5 | 83.3 | 10.3 | B4-380-600 | 83.4 | **6.0** | **1.7x** |
| B6 | 83.7 | 19.1 | B5-456-600 | 83.8 | **12.0** | **1.6x** |
| B7 | 84.1 | 37 | B6-528-600 | 84.1 | **22.8** | **1.6x** |

We demonstrate that one can convert a single model into a cascade by passing the same input image at different resolutions to the model. Here, we leverage the fact that resizing an image to a higher resolution than the model is trained on often yields a higher accuracy [143] at the cost of more computation. We call such cascades as "self-cascades" since these cascade only contain the model itself.

Given a model $M$, we build a 2-model cascade, where the first model is applying $M$ at resolution $r_1$ and the second model is applying $M$ at a higher resolution $r_2 (r_2 > r_1)$. We build self-cascades using EfficientNet models. Since each EfficientNet is defined with a specific resolution (*e.g.*, 240 for B1), we set $r_1$ to its original resolution and set $r_2$ to a higher resolution. We set the confidence threshold such that the self-cascade matches the accuracy of a single model.

Table 5.13 shows that self-cascades easily outperform single models, *i.e.*, obtaining a similar accuracy with fewer FLOPs. Table 5.13 also suggests that if we want to obtain B7 accuracy, we can train a B6 model and then build a self-cascade, which not only uses much fewer FLOPs during inference, but also takes much shorter time to train.

Self-cascades provide a way to convert one single model to a cascade which will be more efficient than the original single model. The conversion is almost free and does not require training any additional models. They are useful when one does not have resources to train additional models or the training data is unavailable (*e.g.*, the model is downloaded).

## 5.8 Applicability beyond Image Classification

We now demonstrate that the benefit of cascades generalizes beyond image classification.

Table 5.14: Cascades of X3D models on Kinetics-600. We outperform X3D-XL by 1.2%.

| | **Single Models** | | **Cascades - Similar FLOPs** | | | **Cascades - Similar Accuracy** | | |
|---|---|---|---|---|---|---|---|---|
| | Top-1 (%) | FLOPs (B) | Top-1 (%) | FLOPs (B) | ΔTop-1 | Top-1 (%) | FLOPs (B) | Speedup |
| X3D-M | 78.8 | $6.2 \times 30$ | **80.3** | $5.7 \times 30$ | **1.5** | 79.1 | $\mathbf{3.8 \times 30}$ | **1.6x** |
| X3D-L | 80.6 | $24.8 \times 30$ | **82.7** | $24.6 \times 30$ | **2.1** | 80.8 | $\mathbf{7.9 \times 30}$ | **3.2x** |
| X3D-XL | 81.9 | $48.4 \times 30$ | **83.1** | $38.1 \times 30$ | **1.2** | 81.9 | $\mathbf{13.0 \times 30}$ | **3.7x** |

Table 5.15: Cascades of DeepLabv3 models on Cityscapes.

| | mIoU | FLOPs (B) | Speedup |
|---|---|---|---|
| ResNet-50 | 77.1 | 348 | - |
| ResNet-101 | 78.1 | 507 | - |
| Cascade - full | 78.4 | 568 | 0.9x |
| Cascade - $s = 512$ | 78.1 | 439 | 1.2x |
| Cascade - $s = 128$ | 78.2 | **398** | **1.3x** |

### 5.8.1 Video Classification

Similar to image classification, a video classification model outputs a vector of logits over possible classes. We use the same procedure as above to build cascades of video classification models.

We consider the X3D [41] architecture family for video classification, which is the state-of-the-art in terms of both the accuracy and efficiency. The X3D family contains a series of models of different sizes. Specifically, we build cascades of X3D models to match the FLOPs or accuracy of X3D-M, X3D-L or X3D-XL on Kinetics-600 [18].

The results are summarized in Table 5.14, where cascades significantly outperform the original X3D models. Following X3D [41], we sample 30 clips from each input video when evaluating X3D models on Kinetics-600. The 30 clips are the combination of 10 uniformly sampled temporal crops and 3 spatial crops. The final prediction is the mean of all individual predictions. Therefore, we include '$\times$30' in Table 5.14. Our cascade outperforms X3D-XL, a state-of-the-art video classification model, by 1.2% while using fewer average FLOPs. Our cascade can also match the accuracy of X3D-XL with 3.7x fewer average FLOPs.

### 5.8.2 Semantic Segmentation

In semantic segmentation, models predict a vector of logits for each pixel in the image. This differs from image classification, where the model makes a single prediction for the entire image. We therefore revisit the confidence function definition

to handle such dense prediction tasks.

Similar to before, we use the maximum probability to measure the confidence of the prediction for a single pixel $p$, *i.e.*, $g(\alpha_p) = \max(\text{softmax}(\alpha_p))$, where $\alpha_p$ is the predicted logits for pixel $p$. Next, we need a function $g^{\text{dense}}(\cdot)$ to rate the confidence of the dense prediction for an image, so that we can decide whether to apply the next model to this image based on this confidence score. For this purpose, we define $g^{\text{dense}}(\cdot)$ as the average confidence score of all the pixels in the image: $g^{\text{dense}}(R) = \frac{1}{|R|}\sum_{p \in R} g(\alpha_p)$, where $R$ represents the input image.

In a cascade of segmentation models, we decide whether to pass an image $R$ to the next model based on $g^{\text{dense}}(\cdot)$. Since the difficulty to label different parts in one image varies significantly, *e.g.*, roads are easier to segment than traffic lights, making a single decision for the entire image can be inaccurate and leads to a waste of computation. Therefore, in practice, we divide an image into grids and decide whether to pass each grid to the next model separately.

We conduct experiments on Cityscapes [29] and use mean IoU (mIoU) as the metric. We build a cascade of DeepLabv3-ResNet-50 and DeepLabv3-ResNet-101 [25] and report the reults in Table 5.15. $s$ is the size of the grid. The full image resolution is 1024×2048, so $s = 512$ means the image is divided into 8 grids. If we operate on the full image level ('full'), the cascade will use more FLOPs than ResNet-101. But if operating on the grid level, the cascade can successfully reduce the computation without hurting the performance. For example, the smaller grid size ('$s = 128$') yields 1.3x reduction in FLOPs while matching the mIoU of ResNet-101.

## 5.9 Conclusion

We show that committee-based models, *i.e.*, model ensembles or cascades, provide a simple complementary paradigm to obtain efficient models without tuning the architecture. Notably, cascades can match or exceed the accuracy of state-of-the-art models on a variety of tasks while being drastically more efficient. Moreover, the speedup of model cascades is evident in both FLOPs and on-device latency and throughput. The fact that these simple committee-based models outperform sophisticated NAS methods, as well as manually designed architectures, should motivate future research to include them as strong baselines whenever presenting a new architecture. For practitioners, committee-based models outline a simple procedure to improve accuracy while maintaining efficiency that only needs off-the-shelf models.

# Chapter 6

# Cost-Aware Evaluation and Model Scaling for LiDAR-Based 3D Object Detection

## 6.1 Introduction

LiDAR-based 3D object detection is essential for autonomous driving. Existing research efforts have proposed a diverse range of 3D detectors, where point clouds are organized in various formats (*e.g.*, point-based [110, 125], grid-based [76, 170, 177], range view [12, 40], or hybrid [31, 123, 124] representation) and processed by different architecture components (*e.g.*, PointNet [111, 112], 3D sparse convolution [48], or 2D convolution). We observe a significant boost in the detection performance. For example, the Average precision (AP) for vehicle detection on the Waymo Open Dataset [1] has been improved from 56.62% (PointPillars [76]) to 79.25% (PV-RCNN++ [124]) in just three years!

However, despite the promising empirical performance, we make a somewhat worrisome observation on the current state of 3D detection research: the computational cost (*e.g.*, inference latency) is usually not controlled during the comparison of different detectors. Recent 3D detection methods tend to emphasize and attribute the performance gain to their novel architecture design. But it is unclear whether the proposed detectors are faster or slower than the baselines they are comparing to.

Why are we worried about this observation? We note that when developing architectures on ImageNet [118], it is common practice to compare them under the same cost [91, 93, 114, 140, 150, 152]. But this has yet to be the case for 3D object detection. Since simply scaling up an architecture can already significantly boost the accuracy [5, 140], it is unfair to compare different architectures without controlling the cost. Such an unfair comparison makes it unclear whether the performance gain in recent 3D detection methods is actually brought by their proposed architectural

changes or simply due to the usage of more computation. This can cause misleading conclusions on the contribution of different architecture components.

Addressing the unfair comparison issue is the basis for us to know the true contribution of the diverse architecture components used in existing methods. This is also important for future research to further push the frontier of 3D detection. Fully addressing this issue surely requires a community-wide effort. We take a step forward by analyzing the performance of a simple grid-based one-stage detector, *i.e.*, SECOND [170], under different costs by scaling its original architecture.

We choose SECOND for the following reasons: (1) SECOND is a widely-used baseline and generally believed to have been significantly outperformed; (2) SECOND has nice open-source implementation[1] available; and (3) most importantly, SECOND is the common part of several high-performing two-stage detectors (*e.g.*, PV-RCNN++ [124] and Voxel R-CNN [31]), where SECOND is adopted as their first stage to generate region proposals. Studying the performance of SECOND can immediately inform us about whether these sophisticated second stage detectors are necessary to achieve competitive detection performance.

In order to analyze the performance of SECOND under different costs, we study different choices to scale its backbone. We show that increasing the pre-head resolution, *i.e.*, the spatial dimension of the feature map being passed to the detection head, is usually better than just increasing the network depth or width.

Then we compare the family of scaled SECOND against recent 3D detection methods. The results are surprising. We find that, SECOND can easily outperform most recent 3D detection methods after being scaled up. Notably, scaled SECOND can match the performance of PV-RCNN++, the current state-of-the-art on the Waymo Open Dataset, if allowed to use a similar inference latency. Scaled SECOND also easily outperforms many recent 3D detection methods published during the past year. Our results indicate that the gain brought by the architectural innovation in many recent methods is not as significant as what was shown in their papers.

We summarize our contributions as follows: (1) We point out the vast importance of comparing different 3D detectors under the same cost, which may sound obvious but was overlooked in the recent literature. (2) We provide an extensive analysis on how to scale up the backbone of grid-based 3D detectors, *e.g.*, SECOND, and find that increasing the pre-head resolution is a reliable source for better performance. (3) We introduce the family of scaled SECOND by scaling up the original backbone of SECOND and conduct a cost-aware comparison of scaled SECOND against recent 3D detection methods. Our comparison leads to a surprising observation: simply scaling the backbone in SECOND can already match the state-of-the-art performance on the Waymo Open Dataset.

[1] We refer to the open-source implementation in OpenPCDet [142].

## 6.2 Related Work

### 6.2.1 LiDAR-Based 3D Object Detection

Point clouds captured by LiDAR sensor are irregular. This makes it difficult to directly apply traditional convolutional architectures to point clouds, which have been successful for images and videos but require the input data to be organized in the format of regular grids [84]. Several different ways have been proposed to address this issue. Following the categorization of 3D detectors in PV-RCNN++ [124], we briefly review existing 3D detection methods based on how they represent and process the point cloud.

**Point-based Representation.** This line of work treats point clouds as unordered point sets and directly process the raw point cloud. Most of them [108–110, 125, 173, 174] adopted PointNet or its variant [111, 112] as the backbone. For example, PointRCNN [125] proposed to first generate 3D proposals based on the foreground point segmentation given by a PointNet++ [112] network, and then further refine the 3D bounding boxes via point cloud region pooling. STD [174] also used PointNet++ to extract point features to generate proposals in a bottom-up fashion. LiDAR R-CNN [84] assumed that a set of 3D proposals had been generated and proposed a second-stage detector based on PointNet to refine the proposals. Point-GNN [127] explored using graph neural networks to encode the point cloud by constructing a fixed radius near-neighbors graph. Pan et al. [103] proposed Point-Former, a Transformer architecture for 3D point clouds, to serve as the backbone in point-based detectors. Point-based representation can fully reserve the 3D structure and fine details of the point cloud. But the nearest neighbor search operation used in PointNet or PointNet++ variant is computationally prohibitive as the number of points increases. While the efficiency issue can be partially mitigated by downsampling the point cloud (*e.g.*, only keeping 16384 points [125]), the downsampling inevitably brings a significant performance drop. This limits the application of point-based detectors to large-scale scenes [84].

**Grid-based Representation.** To deal with the irregularity of point clouds, previous work proposed to divide point clouds into regular grids, *e.g.*, voxels, pillars, and bird's-eye view (BEV), to make it possible to apply convolutional operations. Voxel-Net [188] partitioned the space into equally spaced voxels, applied PointNet [111] in each voxel to generate voxel features, and then used dense 3D convolution to further aggregate the spatial context. SECOND [170] improved upon VoxleNet by using 3D sparse convolution [48] and removing the PointNet. PointPillars [76] proposed to organize the point cloud as pillars (vertical columns) to improve the voxel backbone efficiency. Voxel features or pillar features are often projected onto the ground plane, *i.e.*, BEV, before being passed to the detection head, where 2D convolution can be readily applied. PIXOR [172] directly represented the input as a set of

2D BEV maps without using 3D voxel grids and used 2D convolutional networks as the backbone. Given BEV feature maps, CenterPoint [177] proposed a center-based detection head for 3D object detection without pre-defining axis-aligned anchors. The grid-based representation makes it easy to apply convolutional operations, but suffers from the quantization error caused by dividing the space into regular grids, which can limit the detection performance, especially for distant objects with a few points.

**Range View Representation.** Range view is the native representation of LiDAR data and can be efficiently processed by existing 2D convolutional architectures [100]. VeloFCN [77] is the pioneering work in this line, where they designed a fully convolutional network to detect 3D objects from range images. LaserNet [100] also used a fully convolutional network as the backbone. RCD [12] proposed a novel range-conditioned dilation layer to account for the scale variation of objects in range images. RangeDet [40] proposed several strategies to improve pure rage-view-based object detection and achieved comparable performance with state-of-the-art methods. Challenges of using range view representation include dealing with scale variation and occlusion [100].

**Hybrid Representation.** Since different representations have their own pros and cons, previous methods [26, 50, 74, 86, 123, 124, 136, 187] also explored combining multiple representations of the point cloud. For example, MVF [26] proposed a novel multi-view fusion algorithm to effectively use BEV and range view. M3DETR [50] explored fusing multi-representation features from points, voxels and BEV with Transformer. Shi et al. [123] proposed PV-RCNN, a two-stage detection framework that takes the advantage of both the voxel-based and point-based methods. Its extension PV-RCNN++ [124] achieved state-of-the-art performance on the Waymo Open Dataset.

### 6.2.2 Characterizing Architectures

Pursing the ImageNet [118] challenge probably motivated the most amount of efforts on designing novel architectures [37,54,59,73,138,140,183] in computer vision. ImageNet classification architectures also generalized well to other tasks, such as detection [65] and segmentation [24]. We note that when intending to show a novel architecture is more accurate than previous ones on ImageNet, the common practice is to ensure that the proposed architecture use similar computational cost with the baselines[2] [91, 93, 114, 140, 150, 152]. For example, RegNet [114] considered a wide range of computation regimes and conducted the comparison of architectures within each regime, *e.g.*, the mobile regime where all the architectures are at about

[2]Equivalently, it is also common in practice to show the proposed architecture can achieve comparable accuracy with less cost.

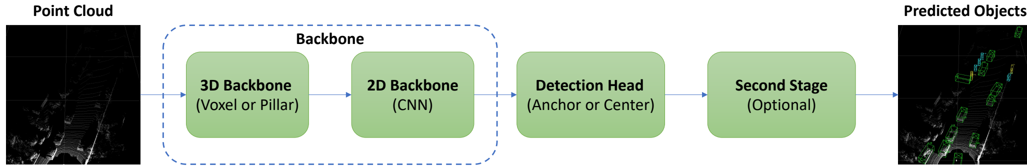



Figure 6.1: Main components in a LiDAR-based 3D object detector. We list examples choices for a component in the bracket but there could be other choices.

Table 6.1: Component Overview of Relevant Detectors.

| | 3D Backbone | Detection Head | Second Stage |
|---|---|---|---|
| †SECOND-Anchor [170] | Voxel | Anchor | ✗ |
| ‡SECOND-Center | Voxel | Center | ✗ |
| PointPillars [76] | Pillar | Anchor | ✗ |
| CenterPoint [177] | Pillar / Voxel | Center | ✓ |
| PV-RCNN [123] | Voxel | Anchor / Center | ✓ |
| PV-RCNN++ [124] | Voxel | Anchor / Center | ✓ |

† SECOND-Anchor is the original SECOND method that uses the anchor head.
‡ SECOND-Center is equivalent to the first stage of CenterPoint.

600M FLOPs. The very recent Swin Transformer [93] reported model parameters, FLOPs, and throughput in their empirical evaluation.

Previous results [5,140] have demonstrated that scaling up an architecture, *e.g.*, increasing the number of layers or channels, can significantly improve the accuracy. Therefore, comparing architectures of different costs is unfair and cannot justify that the performance gain is due to the novel architecture design. Unfortunately, the aforementioned good practice has not yet been ubiquitously adopted in 3D object detection. Our work is inspired by this good practice and aims to quantify how much performance gain in state-of-the-art 3D detection methods is due to the architectural innovation.

## 6.3 Architecture Overview

This section reviews architecture details of relevant detection methods to provide background for our analysis. We illustrate a detector as the combination of a backbone, a detection head, and optionally a second stage in Fig. 6.1. The backbone is further divided into a 3D one and a 2D one. Table 6.1 gives a component overview of relevant detectors and we describe more details as follows.

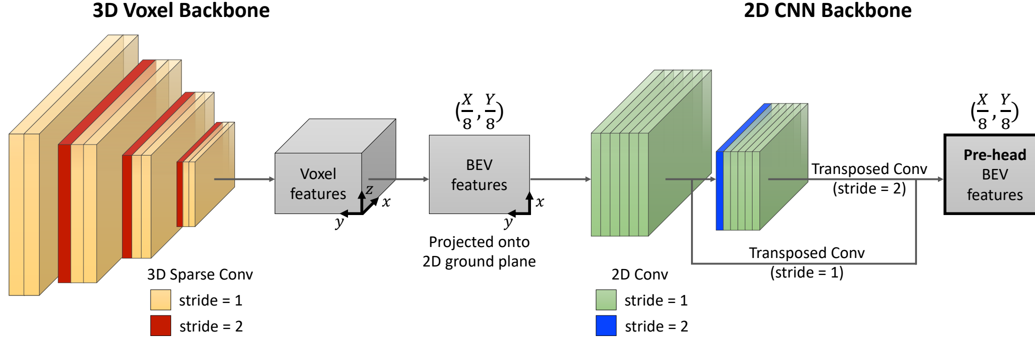



Figure 6.2: Backbone Architecture of SECOND (A0 in Table 6.2). Assuming the input point cloud is initially grouped into $X \times Y \times Z$ voxels, the pre-head resolution of the shown A0 backbone, *i.e.*, the spatial dimension of the obtained pre-head BEV features, will be $(\frac{X}{8}, \frac{Y}{8})$.

Table 6.2: List of architectures used in our analysis. A0 is the original backbone of SECOND implemented in OpenPCDet.

| | 3D Depth | 3D Width | 2D Depth | 2D Width | Pre-Head |
|---|---|---|---|---|---|
| A0 | 2, 3, 3, 3 | 16, 32, 64, 64 | 6, 6 | 128, 256 | $(X/8, Y/8)$ |
| A0-deep | 8, 12, 12, 12 | 16, 32, 64, 64 | 24, 24 | 128, 256 | $(X/8, Y/8)$ |
| A0-wide | 2, 3, 3, 3 | 32, 64, 128, 128 | 6, 6 | 256, 512 | $(X/8, Y/8)$ |
| A0-d&w | 3, 5, 5, 5 | 28, 56, 112, 112 | 9, 9 | 224, 448 | $(X/8, Y/8)$ |
| A1 | 2, 4 ,4 | 32, 64, 64 | 6, 6 | 128, 256 | $(X/4, Y/4)$ |
| A2 | 3, 6, 6 | 48, 96, 144 | 12, 12 | 128, 256 | $(X/4, Y/4)$ |

### 6.3.1 SECOND, PointPillars & CenterPoint

**SECOND.** Most of our analysis focuses on SECOND [170], one of the earliest 3D detection methods and a widely-used baseline in the literature. SECOND first groups the point cloud into voxels and then extracts 3D voxel-wise features using the 3D backbone. The sparse 3D voxel-wise features are then projected onto the ground plane ($x$ and $y$-axis) to obtain dense 2D BEV features, which is done by channel concatenation across the height dimension ($z$-axis). The obtained BEV features are then processed by the 2D backbone and passed to the detection head to generate a set of region proposals, including their location, size, orientation and class. Finally, non-maximum suppression (NMS) is applied on the region proposals to remove redundant object predictions.

Fig. 6.2 shows the original backbone architecture of SECOND implemented in OpenPCDet [142] (A0 in Table 6.2). The 3D backbone is formed by stacking 3D sparse convolutional layers [48] and consists of four stages. The first layer at each stage, except the first stage, has a stride of 2 to reduce the spatial dimension in 3D.

The 2D backbone consists of 2D convolution layers and has two stages. All the layers have a stride of 1 except that the first layer at the second stage has a stride of 2. The output of each stage is transformed or upsampled via transposed convolution and then concatenated across the channel dimension to obtain the final BEV features, which we will refer to as "pre-head features" since the features will be directly passed to the detection head to predict region proposals. Relatedly, the spatial dimension of the pre-head features given by A0 will be $(\frac{X}{8}, \frac{Y}{8})$, assuming the input point cloud is initially grouped into $X \times Y \times Z$ voxels. We discuss the details of detection head after PointPillars.

**PointPillars.** For completeness we also include PointPillars [76] in our analysis, another popular baseline for 3D detection. PointPillars [76] is similar to SECOND except for the 3D backbone, where the point cloud is organized as pillars (vertical columns) instead of voxels. They obtain pillar features by aggregating point features inside each pillar and then convert pillar features into a pseudo-image, *i.e.*, BEV features, which are later passed to the 2D backbone and detection head.

**Detection Head.** As listed in Table 6.1, the original SECOND and PointPillars method use an anchor-based detection head, which pre-defines axis-aligned anchors of different classes on each location. We refer to the original SECOND method using anchor head as SECOND-Anchor. The anchor head takes BEV features as input and uses a convolutional layer to regress the residuals between the ground truth object boxes and pre-defined anchors, as well as predicting the class probabilities of each anchor.

CenterPoint [177] proposed a center-based detection head, which dose not require pre-defining anchors and achieves superior performance over the anchor head. This center head first detects object centers using a keypoint detector and then regresses other attributes, *e.g.*, object size and orientation, for each detected center. The center head is generic and can be used as a drop-in replacement for anchor head. Therefore, our analysis also considers the center head and uses it within SECOND by replacing the anchor head in SECOND-Anchor as center head, which we list as SECOND-Center in Table 6.1. For clarification, the full method of CenterPoint is a two-stage detector and SECOND-Center is exactly the same as the first stage of CenterPoint.

### 6.3.2 Part-A2-Net, PV-RCNN, PV-RCNN++ & Voxel R-CNN

Part-A2-Net [126], PV-RCNN [123], PV-RCNN++ [124], and Voxel R-CNN [31] are all two-stage 3D detectors achieving competitive performance. They all use SECOND as their first stage to generate initial region proposals, which are then further refined in the second stage. Similar to SECOND, both anchor head and center head

can be used in these two-stage detectors. Notably, PV-RCNN++ achieves the state-of-the-art performance on the Waymo Open Dataset.

## 6.4 Experimental Setup

We conduct experiments with OpenPCDet, an open-source code base for LiDAR-based 3D object detection. OpenPCDet is the official code release of Part-A2-Net [126], PV-RCNN [123], PV-RCNN++ [124], and Voxel R-CNN [31], and also supports many other methods, including SECOND [170] and PointPillars [76].

**Dataset and Metrics.** We use the Waymo Open Dataset [1], the largest public benchmark for LiDAR-based 3D object detection, in our experiments. It contains 798 train sequences (~158k frames) and 202 validation sequences (~40k frames). Following the standard protocol, we adopt average precision (AP) and average precision weighted by heading (APH) as the metrics and evaluate in two difficulty levels (LEVEL_1 and LEVEL_2). Among all the metrics, the LEVEL_2 APH is the most important according to the official Waymo evaluation server.

**Training Setup.** By default, we train on all the train sequences and evaluate on all the validation sequences (100% training setup). To save training time for the analysis in Sec. 6.5, we adopt the 20% training setup provided in OpenPCDet [142]. Under this setup, we train on 20% frames uniformly sampled from the train sequences but still evaluate on all the validation sequences. Since LiDAR frames in one sequence are highly correlated, 20% of data is usually representative enough.

**Inference Latency.** We use batch size 1 when measuring the latency of a detector, following the convention in detection [141]. The latency is measured on a Nvidia GeForce RTX 3090 GPU and a AMD EPYC 7402 24-Core CPU.

## 6.5 Scaling Depth, Width, and Pre-Head Resolution

Scaling the depth (number of layers), width (number of channels), or input image resolution have been widely used to improve the performance of a model [5, 54, 59, 140]. But it is still an open question about what the optimal scaling strategy is. There can be a large performance variation among different scaling configurationseven when then the amount of cost is controlled, especially in the large computation regime [5]. For example, EfficientNet [140] demonstrated that compound scaling, *i.e.*, scaling all three dimensions including depth, width, and resolution, is better than scaling only one of the dimensions. Bello et al. [5] observed diminishing returns in very large image resolutions in EfficientNet and suggested that one

Table 6.3: Performance of SECOND-Anchor with different backbones on the Waymo validation set (20% training). Scaling the pre-head resolution from $(\frac{X}{8}, \frac{Y}{8})$ to $(\frac{X}{4}, \frac{Y}{4})$ significantly improve the performance on all classes. But further increasing the resolution does not help.

| Anchor Head | LEVEL_2 3D APH | | | | Latency | Params | Memory | Pre-Head |
|---|---|---|---|---|---|---|---|---|
| | Vehicle | Pedestrian | Cyclist | mAPH | (ms) | (M) | (GB) | Resolution |
| A0 | 62.02 | 47.49 | 53.53 | 54.35 | 27 | 5.33 | 5.8 | $(X/8, Y/8)$ |
| A0+Upsample | 64.61 | 51.92 | 59.81 | 58.78 | 38 | 5.69 | 9.0 | $(X/4, Y/4)$ |
| A0+Upsample×2 | 64.51 | 50.50 | 59.82 | 58.28 | 52 | 5.69 | 20.0 | $(X/2, Y/2)$ |
| A1 | **65.65** | **59.20** | **64.33** | **63.06** | 65 | 5.56 | 14.4 | $(X/4, Y/4)$ |

Table 6.4: Performance of SECOND-Center with different backbones on the Waymo validation set (20% training). The advantage of scaling pre-head resolution generalizes to the center-based detection head.

| Center Head | LEVEL_2 3D APH | | | | Latency | Params | Memory | Pre-Head |
|---|---|---|---|---|---|---|---|---|
| | Vehicle | Pedestrian | Cyclist | mAPH | (ms) | (M) | (GB) | Resolution |
| A0 | 62.65 | 58.23 | 64.87 | 61.92 | 28 | 5.78 | 5.6 | $(X/8, Y/8)$ |
| A0+Upsample | 64.86 | 61.26 | 65.79 | 63.98 | 43 | 6.14 | 10.0 | $(X/4, Y/4)$ |
| A1 | **64.92** | **65.35** | **67.96** | **66.08** | 67 | 6.01 | 14.7 | $(X/4, Y/4)$ |

should increase the resolution slowly. To form the basis of our analysis, this section analyzes different ways to scale the backbone of SECOND.

### 6.5.1 Pre-Head Resolution

Unlike images, there is no notion of resolution for raw point clouds. Therefore, we consider the pre-head resolution, *i.e.*, the spatial dimension of the pre-head BEV features, as an alternative choice to scale the backbone.

We note that previous work on 2D object detection [46, 87, 141] explored using multi-scale feature maps, whose main motivation is to handle the scale variation of objects in images, *i.e.*, using a higher-resolution feature map for larger anchors, since the size of the same object could vastly vary depending on its distance to the camera. But this is not the case for LiDAR point clouds as the size of a specific object in point clouds is fixed no matter its distance to the sensor.

The advantage of a larger pre-head resolution for LiDAR-based detection is in the denser sampling of anchors or keypoints. The anchor head places pre-defined anchors on every location of the BEV features. Therefore, a larger pre-head feature map resolution means more anchors are used. For center head [177], which detects object centers via keypoint estimation, a larger resolution means more keypoints are classified.

Now we empirically investigate whether scaling the pre-head resolution leads to performance improvement. Starting from A0, the original backbone in SECOND with a pre-head resolution of $(\frac{X}{8}, \frac{Y}{8})$, we consider the following backbones to increase the pre-head resolution:

- **A0+Upsample:** we change the stride of the transposed convolution at the end of first stage in the 2D backbone to 2, and add another transposed convolutional layer of a stride 2 at the end of the second stage. This yields a pre-head resolution of $(\frac{X}{4}, \frac{Y}{4})$.
- **A0+Upsample×2:** we further upsamples the pre-head feature map given by 'A0+Upsample' by 2x with Bilinear interpolation. We do not use an transposed convolution here due to its large memory footprint. This yields a pre-head resolution of $(\frac{X}{2}, \frac{Y}{2})$.
- **A1:** we remove the last stage in the 3D backbone of A0 to perform less downsampling in the network. The number of layers and channels are slightly adjusted so that A1 has a similar number of parameters to A0. A1 has a pre-head resolution of $(\frac{X}{4}, \frac{Y}{4})$.

We show the performance of different backbones in Table 6.3. We also report the inference latency, number of parameters, and memory footprint at batch size 2 during training for completeness. Here we focus on analyzing whether increasing the pre-head resolution can improve the performance, so we do not control the latency of different backbones to be the same.

As shown in Table 6.3, scaling the resolution to $(\frac{X}{4}, \frac{Y}{4})$ is beneficial as A1 and 'A0+Upsample' outperform A0 by a large margin on all classes. Table 6.4 provides the results of SECOND-Center with different backbones and we see the benefit of a larger pre-head resolution generalizes to center-based head. But we also notice that further increasing the pre-head resolution does not bring any additional gain ('A0+Upsample' vs. 'A0+Upsample×2'). Therefore, we conclude that scaling the pre-head resolution is a reliable source for better performance but one should refrain from increasing the resolution aggressively.

### 6.5.2 Depth vs. Width vs. Pre-Head Resolution

We now compare the following ways to scale the backbone in SECOND: (1) depth only (A0-deep), (2) width only (A0-wide), (3) depth and width at the same time (A0-d&w), and (4) pre-head resolution only (A1). The architecture details are available in Table 6.2 and the results are shown in Table 6.5. We control the inference latency of of different backbones are to be similar for fair comparison.

While other backbones can easily improve the performance, we notice that A0-deep underperforms the original A0 backbone. We conjecture this is due to the lack of residual connections. A0-deep is significantly deeper that other networks

Table 6.5: Depth vs. Width vs. Pre-Head Resolution. We compare the performance of SECOND-Anchor with different scaled backbones under the same latency on the Waymo validation set (20% training). Scaling the pre-head resolution provides the highest overall mAPH with the fewest parameters.

| Anchor Head | LEVEL_2 3D APH Vehicle | Pedestrian | Cyclist | mAPH | Latency (ms) | Params (M) | Memory (GB) |
|---|---|---|---|---|---|---|---|
| A0 | 62.02 | 47.49 | 53.53 | 54.35 | 27 | 5.33 | 5.8 |
| **Without Residual Connections** | | | | | | | |
| A0-deep | 59.98 | 35.53 | 43.15 | 46.22 | 55 | 20.92 | 13.5 |
| A0-wide | 65.10 | 53.01 | 59.61 | 59.24 | 61 | 19.96 | 8.0 |
| A0-d&w | 65.50 | 51.94 | 59.46 | 58.97 | 68 | 23.76 | 9.9 |
| A1 | **65.65** | **59.20** | **64.33** | **63.06** | 65 | 5.56 | 14.4 |
| **With Residual Connections** | | | | | | | |
| A0-deep$_{res}$ | **66.98** | 54.80 | 60.35 | 60.71 | 59 | 21.23 | 15.4 |
| A0-wide$_{res}$ | 65.82 | 55.17 | 60.41 | 60.47 | 64 | 20.19 | 8.0 |
| A0-d&w$_{res}$ | 66.69 | 56.00 | 61.70 | 61.46 | 66 | 21.66 | 10.4 |
| A1$_{res}$ | 65.78 | **59.82** | **64.27** | **63.29** | 67 | 5.65 | 13.7 |

and thus much harder to train. Therefore, we add residual connections to all the backbones (subscripted by 'res' in Table 6.5).

All the backbones benefit from adding residual connections and significantly outperform A0. Among the different choices, the compound scaling of depth and width is better than scaling one dimension only, echoing Tan et al. [140]. Scaling the pre-head resolution achieves the best overall performance, while scaling depth is the best for vehicle detection. Scaling the pre-head resolution also significantly saves the number of parameters compared with other choices.

## 6.6 Comparing Scaled SECOND Against Recent Methods

### 6.6.1 Family of Scaled SECOND

Based on the above analysis on how to scale the backbone in SECOND, we introduce the family of scaled SECOND. In addition to the A0 and A1$_{res}$ backbone, we design a larger backbone A2$_{res}$ to cover the high latency regime. A2$_{res}$ is obtained by adding residual connections in A2, where A2 is obtained by increasing the depth and width of A1 (see Table 6.2 for architecture details).

Then the family of scaled SECOND includes three backbones: A0, A1$_{res}$, and A2$_{res}$. As mentioned in Sec. 6.3, both the anchor head and center head can be used within SECOND. So we have both SECOND-Anchor and SECOND-Center with

the three backbones. To be clear, SECOND-Anchor is the original method of SECOND [170] and SECOND-Center replaces the anchor head in the original SECOND with the center head proposed in CenterPoint [177].

Next, we will compare the family of scaled SECOND against recent LiDAR-based 3D object detection methods. We first compare scaled SECOND against several two-stage detectors in Sec. 6.6.2. These two-stage detectors are specifically selected as they use SECOND as their first stage, including Part-A2-Net [126], PV-RCNN [123], PV-RCNN++ [124], and Voxel R-CNN [31]. Then we extend the comparison to more methods in Sec. 6.6.3.

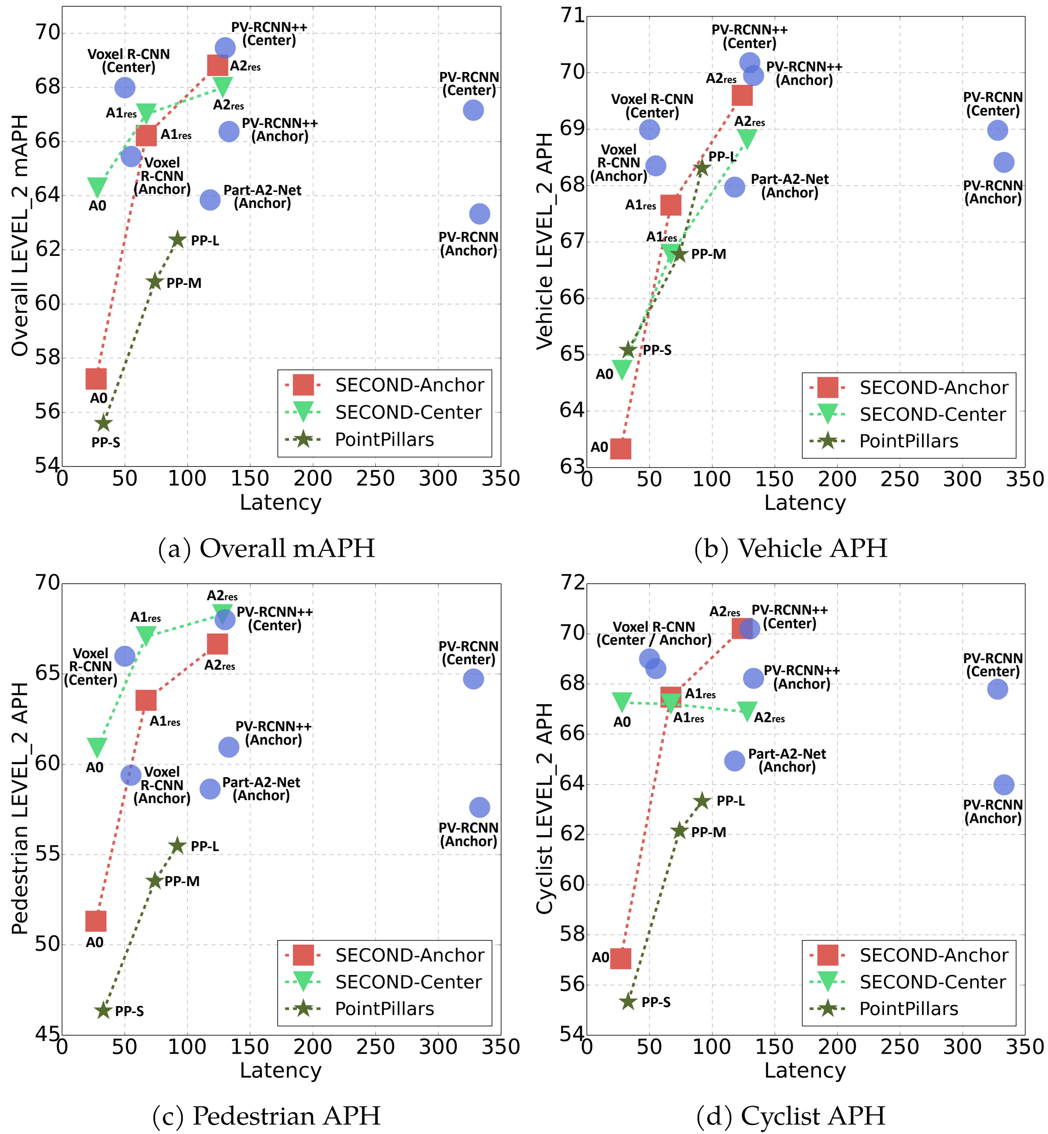
### 6.6.2 Comparison Against Selected Two-Stage Detectors

We compare the family of scaled SECOND against the following two-stage detectors in Fig 6.3: Part-A2-Net [126], PV-RCNN [123], PV-RCNN++ [124], and Voxel R-CNN [31]. As mentioned above, we select them as they all use SECOND as their first stage. We summarize the results in Figure 6.3. All the results in Figure 6.3 are obtained using OpenPCDet [142], the official implementation of the selected two-stage detectors, to ensure reproducibility and fair comparison. Since the detection head can have a big influence on the performance, we consider both the anchor head and center head for PV-RCNN, PV-RCNN++ and Voxel R-CNN in Figure 6.3 to provide a more complete comparison.

**Overall Comparison.** We observe from Figure 6.3a that scaled SECOND significantly outperforms Part-A2-Net and PV-RCNN after controlling the latency in the comparison. Only Voxel R-CNN and PV-RCNN++ can pass the test of scaled SECOND, *i.e.*, more accurate than scaled SECOND when using a similar or smaller latency. But this only happens if they use the center head. Voxel R-CNN (Anchor) and PV-RCNN++ (Anchor) do not pass the test of scaled SECOND either.

**PV-RCNN++.** We take a closer look at PV-RCNN++ as it is the current state-of-the-art on the Waymo Open Dataset. The original SECOND method (SECOND-Anchor-A0) significantly underperforms PV-RCNN++ (Anchor). But simply scaling up its original backbone (A0) to $A1_{res}$ can easily achieve a similar mAPH with PV-RCNN++ (Anchor) while being twice faster during inference. The highest-performing method PV-RCNN++ (Center) is only slightly better than SECOND-Anchor-$A2_{res}$ if SECOND is allowed to use a similar latency. The performance gain brought by PV-RCNN++ is much smaller than what was originally shown in their paper.

**Voxel R-CNN.** Voxel R-CNN (Center) achieves a higher mAPH than SECOND-Center-$A1_{res}$ with a smaller latency. The strong performance and efficiency of Voxel R-CNN is mainly due to their proposed Voxel RoI pooling [31] in the second stage, which does not use the expensive ball query to find nearest neighbors in the 3D space but rather uses an efficient voxel query operation.




(a) Overall mAPH

(b) Vehicle APH

(c) Pedestrian APH

(d) Cyclist APH

Figure 6.3: Scaled SECOND vs. Selected Two-Stage Detectors. In terms of the overall mAPH, only Voxel-RCNN (Center) and PV-RCNN++ (Center) can outperform scaled SECOND with a similar or smaller latency. See Table 6.6a for numerical results.

### 6.6.3 Full Comparison Against Recent Methods

We now extend the comparison to other recent 3D object detection methods in Table 6.6b. While all these methods are proposed after SECOND [170], we observe that most methods fail to outperform SECOND-Anchor-A1$_{res}$, which is exactly

Table 6.6: Scaled SECOND vs. Recent 3D Object Detection Methods. LEVEL_2 AP and APH on Waymo validation set are shown. Only PV-RCNN++ (Center) and Voxel-RCNN (Center) can outperform scaled SECOND when using a similar or smaller latency.

(a) Scaled SECOND vs. Selected Two-Stage Detectors. See Figure 6.3 to visually compare the performance and latency of the listed methods at the same time.

| | Venue | Latency (ms) | Vehicle AP | Vehicle APH | Pedestrian AP | Pedestrian APH | Cyclist AP | Cyclist APH | Overall mAPH |
|---|---|---|---|---|---|---|---|---|---|
| **Selected Two-Stage Detectors** | | | | | | | | | |
| Part-A2-Net (Anchor) | TPAMI 2020 | 118 | 68.47 | 67.97 | 66.18 | 58.62 | 66.13 | 64.93 | 63.84 |
| PV-RCNN (Anchor) | CVPR 2020 | 333 | 68.98 | 68.41 | 66.04 | 57.61 | 65.39 | 63.98 | 63.33 |
| PV-RCNN (Center) | CVPR 2020 | 328 | 69.43 | 68.98 | 70.42 | 64.72 | 68.95 | 67.79 | 67.16 |
| Voxel R-CNN (Anchor) | AAAI 2021 | 55 | 68.86 | 68.35 | 67.15 | 59.39 | 69.78 | 68.61 | 65.45 |
| Voxel R-CNN (Center) | AAAI 2021 | 50 | 69.42 | 68.99 | 71.26 | 65.98 | 70.03 | 69.00 | 67.99 |
| PV-RCNN++ (Anchor) | Arxiv 2021 | 133 | 70.45 | 69.95 | 68.85 | 60.94 | 69.42 | 68.22 | 66.37 |
| PV-RCNN++ (Center) | Arxiv 2021 | 130 | **70.61** | **70.18** | 73.17 | 68.00 | 71.21 | 70.19 | **69.46** |
| **Family of Scaled SECOND** | | | | | | | | | |
| SECOND-Anchor-A0 | Sensors 2018 | 27 | 63.85 | 63.33 | 60.72 | 51.31 | 58.34 | 57.05 | 57.23 |
| SECOND-Anchor-A1$_{res}$ | | 67 | 68.13 | 67.65 | 71.57 | 63.54 | 68.53 | 67.47 | 66.22 |
| SECOND-Anchor-A2$_{res}$ | | 124 | 70.06 | 69.60 | **73.98** | 66.65 | **71.22** | **70.21** | 68.82 |
| SECOND-Center-A0 | | 28 | 65.23 | 64.72 | 66.83 | 60.88 | 68.36 | 67.25 | 64.28 |
| SECOND-Center-A1$_{res}$ | | 71 | 67.23 | 66.78 | 72.98 | 67.06 | 68.29 | 67.20 | 67.01 |
| SECOND-Center-A2$_{res}$ | | 128 | 69.26 | 68.81 | 73.87 | **68.28** | 67.90 | 66.88 | 67.99 |

(b) Scaled SECOND vs. Other Methods.

| | Venue | Latency (ms) | Vehicle AP | Vehicle APH | Pedestrian AP | Pedestrian APH | Cyclist AP | Cyclist APH | Overall mAPH |
|---|---|---|---|---|---|---|---|---|---|
| PPBA [28] | ECCV 2020 | - | - | 53.4 | - | 53.9 | - | - | - |
| LiDAR R-CNN [84] | CVPR 2021 | - | 64.7 | 64.2 | 63.1 | 51.7 | 66.1 | 64.4 | 60.10 |
| 3D-MAN [175] | CVPR 2021 | - | 67.61 | 67.14 | 62.58 | 59.04 | - | - | - |
| PPC [20] | CVPR 2021 | - | - | 56.7 | - | 61.5 | - | - | - |
| MGAF-3DSSD [79] | ACM MM 2021 | $^{\dagger}\sim 60$ | 65.35 | - | - | - | - | - | - |
| RangeDet [40] | ICCV 2021 | $^{\ddagger}\sim 58$ | 64.03 | 63.57 | 67.60 | **63.89** | 63.33 | 62.08 | 63.18 |
| VoTr-TSD [98] | ICCV 2021 | $^{\sharp}> 300$ | 65.91 | 65.29 | - | - | - | - | - |
| Pyramid-PV [97] | ICCV 2021 | $^{\sharp}> 300$ | 67.23 | 66.68 | - | - | - | - | - |
| **SECOND-Anchor-A1$_{res}$** | | 67 | **68.13** | **67.65** | **71.57** | 63.54 | **68.53** | **67.47** | **66.22** |
| Voxel-to-Point [80] | ACM MM 2021 | - | 69.77 | - | - | - | - | - | - |
| **SECOND-Anchor-A2$_{res}$** | | 124 | **70.06** | **69.60** | **73.98** | **66.65** | **71.22** | **70.21** | **68.82** |

$^{\dagger}$ We estimate its latency to be $\sim$ 60 ms on Waymo as it is about twice faster than Part-A2-Net on KITTI [79]
$^{\ddagger}$ Measured as 12 fps on a single 2080Ti GPU [40]. We estimate its latency to be 58 ms on RTX 3090 based on [82].
$^{\sharp}$ We estimate its latency to be larger than 300 ms as it is slightly slower than PV-RCNN on KITTI [97, 98].

the same as the originally proposed SECOND except for using a larger backbone. SECOND-Anchor-A1$_{res}$ is both faster and more accurate than some recent methods, such as VoTR-TSD [98] and Pyramid-PV [97].

The strong performance and simplicity of the family of scaled SECOND should motivate future research to include them as baselines whenever proposing novel 3D object detectors.

### 6.6.4 Comparison Under Multiple Latencies

In the above, we only consider one specific latency when comparing two methods. The family of SECOND contains backbones of different sizes and allows us to have a more complete comparison between methods under multiple latencies. We will show that the conclusion under one latency may not generalize to another and the ranking of two methods could flip.

**Anchor Head vs. Center Head.** The center head was originally proposed in CeterPoint [177] and demonstrated impressive performance gain over the anchor head. But CenterPoint only experimented with a small backbone (similar size to A0). We observe that the benefit of center head shrinks as the backbone size grows.

As shown in Figure 6.3a, the anchor head considerably underperforms the center head when A0 is used in terms of the overall mAPH. But after scaling up A0 to $A2_{res}$, the anchor head obtains a higher mAPH than the center head. We observe a similar trend for vehicle or pedestrian detection. For vehicle detection in Figure 6.3b, the ranking of anchor head and center head flips after the backbone is scaled up to $A1_{res}$. For pedestrian detection in Figure 6.3c, the center head outperforms the anchor head by $\sim 10\%$ in terms of APH under the low latency regime (A0), but the gap decreases to $\sim 1.6\%$ after switching to the high latency regime ($A2_{res}$).

We notice in Figure 6.3d that the cyclist performance of SECOND-Center drops as the backbone becomes larger. We conjecture that this is due to overfitting as the cyclist boxes are very rare ($< 1\%$) in the Waymo Open Dataset. We verify the conjecture with the results in Table 6.7. But the overfitting does not happen for SECOND-Anchor.

**SECOND-Anchor vs. PointPillars.** We include PointPillars in our comparison in Figure 6.3 as PointPillars is also a widely-used baseline for 3D object detection. PointPillars-S is the original PointPillars implementation provided in OpenPCDet. The results of PointPillars-M&L are obtained by scaling up the 2D backbone in PointPillars-S. All the PointPillars models use the anchor head.

We observe that the ranking of SECOND-Anchor and PointPillars changes under different latencies. For example, PointPillars outperforms SECOND by $\sim 2\%$ on vehicle detection under the low latency regime (PointPillars-S vs. SECOND-Anchor-A0 in Figure 6.3b). The low latency regime is where the PointPillars focused on when it was proposed. However, as the backbone size grows, PointPillars-M underperforms SECOND-Anchor-$A1_{res}$ on vehicle detection.

Table 6.7: The performance of SECOND-Center on cyclist detection drops as the backbone size grows under 100% training, but steadily increases under 20% training. This indicates that overfitting happens under the 100% training setup.

(a) 100% Training.

| | LEVEL_2 3D APH | | | |
|---|---|---|---|---|
| | Vehicle | Pedestrian | Cyclist | mAPH |
| A0 | 64.72 | 60.88 | 67.25 | 64.28 |
| $A1_{res}$ | 66.78 | 67.06 | 67.20 | 67.01 |
| $A2_{res}$ | 68.81 | 68.28 | 66.88 | 67.99 |

(b) 20% Training.

| | LEVEL_2 3D APH | | | |
|---|---|---|---|---|
| | Vehicle | Pedestrian | Cyclist | mAPH |
| A0 | 62.65 | 58.23 | 64.87 | 61.92 |
| $A1_{res}$ | 65.30 | 66.09 | 68.50 | 66.63 |
| $A2_{res}$ | 67.74 | 67.83 | 69.92 | 68.50 |

## 6.7 Conclusion

To correctly evaluate the architecture design space of 3D detectors, we point out that it is important to compare different architectures under the same cost. Following this philosophy, we conduct an analysis of how to scale the backbone of SECOND, a simple baseline which is generally believed to have been significantly surpassed, and then introduce the family of scaled SECOND. Scaled SECOND sets a strong baseline for future research on 3D object detection: it outperforms most recent methods and can match the performance of the state-of-the-art method PV-RCNN++ on the Waymo Open Dataset, if allowed to use a similar latency. We hope our analysis can encourage future research to adopt the good practice of cost-aware evaluation and include the family of scaled SECOND as a strong baseline when presenting novel 3D detection methods. We also show that the ranking of two methods can flip under different latencies and suggests that one should conduct the comparison under multiple latencies if possible.

# Chapter 7

# Conclusion & Discussion

This thesis has made progress on both search algorithms and search spaces, the two critical components in NAS. First, we have proposed search algorithms to improve the search efficiency (Chapter 2) and to improve the generalization of found architectures (Chapter 3). Second, we have proposed a novel attention cell search space to extend NAS beyond discovering convolutional cells to attention cells (Chapter 4), identified an overlooked design space for efficient models (Chapter 5), and pointed out the vast importance of controlling the cost when comparing different 3D detector architectures (Chapter 6). Next, we first discuss the potential future directions and then give a concluding summary of this thesis.

## 7.1 Future Work

**Hybrid Search Algorithms.** Existing search algorithms can be categorized into sampling-based algorithms and gradient-based algorithms. Sampling-based algorithms explicitly sample candidate architectures from the search space and select the optimal one. Gradient-based algorithms, *a.k.a.*, differentiable search, do not explicitly sample architectures but use gradient descent to find the optimal solution.

We have explored and used both types of search algorithms in Chapter 2& 3, but we only used them separately. Gradient-based algorithms are efficient but they require the search space to be differentiable or can be made differentiable. Sample-based algorithms are flexible enough to handle different search spaces but may require significant computational resources to find satisfying architectures.

So, one future direction is to propose hybrid search algorithms that can combine the advantages of both types of algorithms. Such hybrid algorithms will be useful for scenarios where part of the search space is differentiable while the other part is non-differentiable or difficult to be made differentiable, *e.g.*, the joint search of hyper-parameters and architectures or the joint search of data augmentation policies and architectures.

**Interplay between the architecture design and compression techniques.** We have shown in Chapter 2 that NAS can successfully compress various types of architectures . In practice, post-training pruning and quantization are also widely used to further compress neural networks. But one question is still unanswered: what types of architectures are easier to compress, prune, or quantize?

We conjecture that architectures with a large number of skip connections are difficult to compress since the skip connections enforce many dependencies between different layers. A principled answer to the above question will be informative for us to develop better efficient network architectures or better compression techniques.

**NAS for Transformer.** We have demonstrated in Chapter 4 that inserting the attention cells found by NAS into existing convolution networks can successfully improve the video classification accuracy. But, as suggested by the recent success of Transformer architectures in computer vision [11, 37, 93], only using self-attention without convolution can also achieve impressive performance on various tasks, such as image or video classification. So, one future direction is to apply NAS to find better Transformer architectures, instead of only attention cells.

At the macro level, we can use NAS to find better ways to stack attention operations to form the entire Transformer architecture. For example, when stacking convolution operations, people have explored various design choices, such as adding skip connections between layers, gradually downsampling the spatial resolution, and combining multi-scale feature maps. We can use NAS to determine whether these design choices are helpful when stack self-attention operations.

At the micro level, we can design the architecture for a single attention operation with NAS. For example, Vaswani et al. [145] applied softmax as the activation function to obtain weights on the value features. But our NAS method in Chapter 4 found that removing this activation function would actually give a higher performance. NAS can also be used to determine the dimension of the hidden layers and the normalization function used in an attention operation.

**NAS for Dynamic Neural Networks.** The model cascades we studied in Chapter 5 fall into the category of dynamic neural networks. Different from static neural networks whose computational graph is fixed for different input examples, dynamic neural networks activate different parts in the network based on the input example [52], where they try to spend more computation on hard examples and less on easy ones.

Our model cascades in Chapter 5 only use pre-trained models and these models are separate and arranged as a simple chain. Our model cascades can be improved in the following three aspects: (1) We can use NAS to search the architectures for each model in a cascade instead of using pre-trained models; (2) We can arrange models in a complex structure, *e.g.*, a multi-branched tree, instead of a simple chain; (3) Models in a cascade do not have to be separate and they can share layers or

branches and have more complex interactions. All these design choices can be explored via NAS.

**NAS for LiDAR-based 3D Object Detecion.** We have shown in Chapter 6 that scaled SECOND is a strong baseline for LiDAR-based 3D object detection and can almost match the performance of PV-RCNN++ [124], the state-of-the-art method on Waymo Open Dataset, if allowed to use a similar latency. But this is still not the full potential of SECOND as we only manually scaled up the backbones of SECOND. So, one future direction is to use NAS to search the backbone for SECOND under different latencies. We can also include high-performing second-stage detectors, *e.g.*, Voxel R-CNN [31], in the search space, which should let the algorithm to find better 3D detectors.

## 7.2 Concluding Summary

We summarize the lessons we have learned about NAS, and more broadly about designing architectures, to conclude this thesis.

**Always control the cost when comparing architectures.** This is the basis for us to correctly evaluate different architectures and is necessary no matter we manually design the architecture or use NAS to find one. This might sound obvious but was overlooked in some recent literature. As shown in our analysis of LiDAR-based 3D detectors in Chapter 6, not controlling the cost would cause unfair comparison and misleading conclusions on different architecture components.

**Strengths of NAS.** NAS is good at optimizing the following attributes of an architecture: type of each layer, connections between layers, depth (number of layers), and width (number of channels). For example, if one invents a new type of layer but is unsure about how to stack this layer to form the entire architecture or how to combine this new layer with existing types of layer, NAS can be used here to figure out the various design choices.

NAS is also useful for practical applications that need a set of architectures to satisfy difference resource constraints or run on different hardware platforms, *e.g.*, different mobile devices. Instead of manually tuning the architecture for each scenario, we can apply NAS here to find those architectures.

NAS is not as expensive as one may have thought if the search space can be made differentiable. The aforementioned attributes (layer type, connections, depth, width) can all be made differentiable, which allows us to efficiently search for the optimal architecture via differentiable search methods. The cost of differentiable search usually equals to training a single big network.

**Limitations of NAS.** The main limitation of NAS is that current NAS methods still rely on humans to define the search space. An ideal search space needs to be flexible enough to contain a sufficient number of high-performing architectures, but cannot be too flexible as this will make satisfying architectures sparse in the space and make search cost unaffordable. This subtle trade-off sometimes makes it non-trivial to define a good search space and makes NAS not as "autonomous" as people hope it to be.

For example, as the reader may have noticed, Chapter 5&6 focus more on analyzing the architecture design space rather than inventing new NAS methods. This is because when we tried to apply NAS to find better dynamic neural networks (Chapter 5) or 3D detectors (Chapter 6), we realized that the search space is crucial to the performance of the found architectures. So we switched to analyzing the architecture design space in Chapter 5&6.

A common criticism on NAS is that NAS cannot discover fundamentally novel architectures. Real et al. [116] has shown preliminary success to evolve machine learning algorithms, *e.g.*, two-layer neural networks trained by backpropagation, from a search space that only uses simple mathematical operations as building blocks. This search space is flexible enough and can contain architectures fundamentally better than existing ones. But it still remains a open problem to actually find those architectures since the sheer size of this search space makes it unaffordable to fully explore it. To solve this problem, we will need flexible search spaces with a reasonable size, more intelligent search algorithms that can quickly find good architectures from a large search space, and maybe also more powerful hardware.

**Manually Designing Architectures vs. NAS.** When manually designing architectures, we directly translate our ideas into the final architecture design. In NAS, since the current form of NAS is still not autonomous enough, we need to formalize our insight and intuition into the search space design. In this sense, NAS and manual design are similar as they all rely on the knowledge humans to design good basic building blocks.

But the design process of an architecture is often complicated by the fact that the empirical performance of an architecture is also influenced by many other design choices in addition to the basic blocks. Sometimes it still takes many efforts to figure out the right number of layers, the right number of channels for each layer, or other design choices, especially when we need to respect certain resource constraints. Formulating these design choices into a search space can allow humans to focus on the creative work and leave the "dirty work" to NAS algorithms.

**Architecture is not the only thing.** While this thesis focuses on NAS, it is important to remember that architecture design is just one part in the machine learning pipeline. The training techniques, *e.g.*, learning rate schedule, regularization strategies, and data augmentation strategies, are also very important and sometimes may

have a bigger impact on the performance than the architecture design. In practice, we need to make sure we have a reasonably good training pipeline when developing architectures and make sure we use the training pipeline when comparing different architectures.

## 7.3 Publication List

Here is the publication list of the author during his study at CMU.
* indicates equal contribution.

[1] **Cost-Aware Evaluation and Model Scaling for LiDAR-Based 3D Object Detection**
**Xiaofang Wang**, Kris M. Kitani
*Arxiv 2022*

[2] **Wisdom of Committees: An Overlooked Approach To Faster and More Accurate Models**
**Xiaofang Wang**, Dan Kondratyuk, Eric Christiansen, Kris M. Kitani, Yair Alon, Elad Eban
*International Conference on Learning Representations (ICLR), 2022*

[3] **Neighborhood-Aware Neural Architecture Search**
**Xiaofang Wang**, Shengcao Cao, Mengtian Li, Kris M. Kitani
*British Machine Vision Conference (BMVC), 2021*

[4] **AttentionNAS: Spatiotemporal Attention Cell Search for Video Classification**
**Xiaofang Wang**, Xuehan Xiong, Maxim Neumann, AJ Piergiovanni, Michael S. Ryoo, Anelia Angelova, Kris M. Kitani, Wei Hua
*European Conference on Computer Vision (ECCV), 2020*

[5] **Learnable Embedding Space for Efficient Neural Architecture Compression**
Shengcao Cao*, **Xiaofang Wang***, Kris M. Kitani
*International Conference on Learning Representations (ICLR), 2019*

[6] **Error Correction Maximization for Deep Image Hashing**
Xiang Xu, **Xiaofang Wang**, Kris M. Kitani
*British Machine Vision Conference (BMVC), 2018*

[7] **Deep Supervised Hashing with Triplet Labels**
**Xiaofang Wang**, Yi Shi, Kris M. Kitani
*Asian Conference on Computer Vision (ACCV), 2016*

# Bibliography

[1] Waymo open dataset: An autonomous driving dataset, 2019.

[2] M. Al-Shedivat, A. G. Wilson, Y. Saatchi, Z. Hu, and E. P. Xing. Learning scalable deep kernels with recurrent structure. *JMLR*, 2017.

[3] A. Ashok, N. Rhinehart, F. Beainy, and K. M. Kitani. N2n learning: Network to network compression via policy gradient reinforcement learning. In *ICLR*, 2018.

[4] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017.

[5] I. Bello, W. Fedus, X. Du, E. D. Cubuk, A. Srinivas, T.-Y. Lin, J. Shlens, and B. Zoph. Revisiting resnets: Improved training and scaling strategies. *NeurIPS*, 2021.

[6] I. Bello, B. Zoph, A. Vaswani, J. Shlens, and Q. V. Le. Attention augmented convolutional networks. In *ICCV*, 2019.

[7] W. H. Beluch, T. Genewein, A. Nürnberger, and J. M. Köhler. The power of ensembles for active learning in image classification. In *CVPR*, 2018.

[8] G. M. Bender, P. jan Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

[9] J. Bergstra, D. Yamins, and D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML*, 2013.

[10] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NeurIPS*, 2011.

[11] G. Bertasius, H. Wang, and L. Torresani. Is space-time attention all you need for video understanding? In *ICML*, 2021.

[12] A. Bewley, P. Sun, T. Mensink, D. Anguelov, and C. Sminchisescu. Range conditioned dilated convolutions for scale invariant 3d object detection. In *CoRL*, 2020.

[13] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama. Adaptive neural networks for efficient inference. In *ICML*, 2017.

[14] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[15] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *ICLR*, 2020.

[16] H. Cai, L. Zhu, and S. Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.

[17] S. Cao, X. Wang, and K. M. Kitani. Learnable embedding space for efficient neural architecture compression. In *ICLR*, 2019.

[18] J. Carreira, E. Noland, A. Banki-Horvath, C. Hillier, and A. Zisserman. A short note about kinetics-600. *arxiv:1808.01340*, 2018.

[19] J. Carreira and A. Zisserman. Quo vadis, action recognition? a new model and the kinetics dataset. In *CVPR*, 2017.

[20] Y. Chai, P. Sun, J. Ngiam, W. Wang, B. Caine, V. Vasudevan, X. Zhang, and D. Anguelov. To the point: Efficient 3d object detection in the range image with graph convolution kernels. In *CVPR*, 2021.

[21] P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, and R. Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. In *ICLR*, 2017.

[22] S. R. Chaudhuri, E. Eban, H. Li, M. Moroz, and Y. Movshovitz-Attias. Fine-grained stochastic architecture search. *arXiv:2006.09581*, 2020.

[23] L.-C. Chen, M. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *NeurIPS*, 2018.

[24] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *TPAMI*, 2017.

[25] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv:1706.05587*, 2017.

[26] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia. Multi-view 3d object detection network for autonomous driving. In *CVPR*, 2017.

[27] X. Chen, L. Xie, J. Wu, and Q. Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *ICCV*, 2019.

[28] S. Cheng, Z. Leng, E. D. Cubuk, B. Zoph, C. Bai, J. Ngiam, Y. Song, B. Caine, V. Vasudevan, C. Li, et al. Improving 3d object detection through progressive population based augmentation. In *ECCV*, 2020.

[29] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *CVPR*, 2016.

[30] J. M. Danskin. *The theory of max-min and its application to weapons allocation problems*. Springer, 1967.

[31] J. Deng, S. Shi, P. Li, W. Zhou, Y. Zhang, and H. Li. Voxel r-cnn: Towards high performance voxel-based 3d object detection. In *AAAI*, 2021.

[32] T. DeVries and G. W. Taylor. Improved regularization of convolutional neural networks with cutout. *arxiv:1708.04552*, 2017.

[33] L. Dinh, R. Pascanu, S. Bengio, and Y. Bengio. Sharp minima can generalize for deep nets. In *ICML*, 2017.

[34] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. In *CVPR*, 2015.

[35] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei, and M. Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *ECCV*, 2018.

[36] X. Dong and Y. Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *ICLR*, 2020.

[37] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[38] T. Elsken, J. H. Metzen, and F. Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *ICLR*, 2019.

[39] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *JMLR*, 2019.

[40] L. Fan, X. Xiong, F. Wang, N. Wang, and Z. Zhang. Rangedet: In defense of range view for lidar-based 3d object detection. In *CVPR*, 2021.

[41] C. Feichtenhofer. X3d: Expanding architectures for efficient video recognition. In *CVPR*, 2020.

[42] C. Feichtenhofer, H. Fan, J. Malik, and K. He. Slowfast networks for video recognition. In *ICCV*, 2019.

[43] C. Feichtenhofer, A. Pinz, and A. Zisserman. Convolutional two-stream network fusion for video action recognition. In *CVPR*, 2016.

[44] S. Fort, H. Hu, and B. Lakshminarayanan. Deep ensembles: A loss landscape perspective. *arXiv:1912.02757*, 2019.

[45] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[46] G. Ghiasi, T.-Y. Lin, and Q. V. Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *CVPR*, 2019.

[47] R. Gómez-Bombarelli, J. N. Wei, D. Duvenaud, J. M. Hernández-Lobato, B. Sánchez-Lengeling, D. Sheberla, J. Aguilera-Iparraguirre, T. D. Hirzel, R. P. Adams, and A. Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 2018.

[48] B. Graham, M. Engelcke, and L. Van Der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. In *CVPR*, 2018.

[49] J. Guan, Y. Liu, Q. Liu, and J. Peng. Energy-efficient amortized inference with cascaded deep classifiers. In *IJCAI*, 2018.

[50] T. Guan, J. Wang, S. Lan, R. Chandra, Z. Wu, L. Davis, and D. Manocha. M3detr: Multi-representation, multi-scale, mutual-relation 3d object detection with transformers. In *WACV*, 2022.

[51] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *ICML*, 2017.

[52] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang. Dynamic neural networks: A survey. *arXiv:2102.04906*, 2021.

[53] D. He, Z. Zhou, C. Gan, F. Li, X. Liu, Y. Li, L. Wang, and S. Wen. Stnet: Local and global spatial-temporal modeling for action recognition. In *AAAI*, 2019.

[54] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[55] D. Hernández-Lobato, J. Hernandez-Lobato, A. Shah, and R. Adams. Predictive entropy search for multi-objective bayesian optimization. In *ICML*, 2016.

[56] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arxiv:1503.02531*, 2015.

[57] S. Hochreiter and J. Schmidhuber. Flat minima. *Neural Computation*, 1997.

[58] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. Searching for mobilenetv3. In *ICCV*, 2019.

[59] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[60] C.-H. Hsu, S.-H. Chang, D.-C. Juan, J.-Y. Pan, Y.-T. Chen, W. Wei, and S.-C. Chang. Monas: Multi-objective neural architecture search using reinforcement learning. In *AAAI*, 2019.

[61] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense networks for resource efficient image classification. In *ICLR*, 2018.

[62] G. Huang, Y. Li, G. Pleiss, Z. Liu, J. E. Hopcroft, and K. Q. Weinberger. Snapshot ensembles: Train 1, get m for free. In *ICLR*, 2017.

[63] G. Huang, S. Liu, L. Van der Maaten, and K. Q. Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *CVPR*, 2018.

[64] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017.

[65] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In *CVPR*, 2017.

[66] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 2011.

[67] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arxiv:1602.07360*, 2016.

[68] R. Jenatton, C. Archambeau, J. González, and M. Seeger. Bayesian optimization with tree-structured dependencies. In *ICML*, 2017.

[69] K. Kandasamy, W. Neiswanger, J. Schneider, B. Poczos, and E. P. Xing. Neural architecture search with bayesian optimisation and optimal transport. In *NeurIPS*, 2018.

[70] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

[71] D. Kondratyuk, M. Tan, M. Brown, and B. Gong. When ensembling smaller models is more efficient than single large models. *arXiv:2005.00570*, 2020.

[72] A. Krizhevsky, G. Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[73] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NeurIPS*, 2012.

[74] J. Ku, M. Mozifian, J. Lee, A. Harakeh, and S. L. Waslander. Joint 3d proposal generation and object detection from view aggregation. In *IROS*, 2018.

[75] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *NeurIPS*, 2017.

[76] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom. Pointpillars: Fast encoders for object detection from point clouds. In *CVPR*, 2019.

[77] B. Li, T. Zhang, and T. Xia. Vehicle detection from 3d lidar using fully convolutional network. In *RSS*, 2016.

[78] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein. Visualizing the loss landscape of neural nets. In *NeurIPS*, 2018.

[79] J. Li, H. Dai, L. Shao, and Y. Ding. Anchor-free 3d single stage detector with mask-guided attention for point cloud. In *ACM Multimedia*, 2021.

[80] J. Li, H. Dai, L. Shao, and Y. Ding. From voxel to point: Iou-guided 3d object detection for point cloud with voxel-to-point decoder. In *ACM Multimedia*, 2021.

[81] L. Li and A. Talwalkar. Random search and reproducibility for neural architecture search. In *UAI*, 2019.

[82] M. Li. Deep learning gpu benchmark: A latency-based approach. `https://mtli.github.io/gpubench/`, 2022.

[83] M. Li, E. Yumer, and D. Ramanan. Budgeted training: Rethinking deep neural network training under resource constraints. In *ICLR*, 2020.

[84] Z. Li, F. Wang, and N. Wang. Lidar r-cnn: An efficient and universal 3d object detector. In *CVPR*, 2021.

[85] H. Liang, S. Zhang, J. Sun, X. He, W. Huang, K. Zhuang, and Z. Li. Darts+: Improved differentiable architecture search with early stopping. *arxiv:1909.06035*, 2019.

[86] M. Liang, B. Yang, S. Wang, and R. Urtasun. Deep continuous fusion for multi-sensor 3d object detection. In *ECCV*, 2018.

[87] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017.

[88] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In *CVPR*, 2019.

[89] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive neural architecture search. In *ECCV*, 2018.

[90] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu. Hierarchical representations for efficient architecture search. In *ICLR*, 2018.

[91] H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019.

[92] X. Liu, J.-Y. Lee, and H. Jin. Learning video representations from correspondence proposals. In *CVPR*, 2019.

[93] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *ICCV*, 2021.

[94] E. Lobacheva, N. Chirkova, M. Kodryan, and D. P. Vetrov. On power laws in deep ensembles. In *NeurIPS*, 2020.

[95] X. Lu, J. Gonzalez, Z. Dai, and N. Lawrence. Structured variationally auto-encoded optimization. In *ICML*, 2018.

[96] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. In *NeurIPS*, 2018.

[97] J. Mao, M. Niu, H. Bai, X. Liang, H. Xu, and C. Xu. Pyramid r-cnn: Towards better performance and adaptability for 3d object detection. In *ICCV*, 2021.

[98] J. Mao, Y. Xue, M. Niu, H. Bai, J. Feng, X. Liang, H. Xu, and C. Xu. Voxel transformer for 3d object detection. In *ICCV*, 2021.

[99] H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, 2016.

[100] G. P. Meyer, A. Laddha, E. Kee, C. Vallespi-Gonzalez, and C. K. Wellington. Lasernet: An efficient probabilistic 3d object detector for autonomous driving. In *CVPR*, 2019.

[101] J. Mockus and L. Mockus. Bayesian approach to global optimization and application to multiobjective and constrained problems. *Journal of Optimization Theory and Applications*, 1991.

[102] M. Monfort, A. Andonian, B. Zhou, K. Ramakrishnan, S. A. Bargal, T. Yan, L. Brown, Q. Fan, D. Gutfreund, C. Vondrick, et al. Moments in time dataset: one million videos for event understanding. *TPAMI*, 2019.

[103] X. Pan, Z. Xia, S. Song, L. E. Li, and G. Huang. 3d object detection with pointformer. In *CVPR*, 2021.

[104] J. Park, S. Woo, J.-Y. Lee, and I.-S. Kweon. Bam: Bottleneck attention module. In *BMVC*, 2018.

[105] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[106] H. Peng, H. Du, H. Yu, Q. Li, J. Liao, and J. Fu. Cream of the crop: Distilling prioritized paths for one-shot neural architecture search. In *NeurIPS*, 2020.

[107] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. In *ICML*, 2018.

[108] C. R. Qi, X. Chen, O. Litany, and L. J. Guibas. Imvotenet: Boosting 3d object detection in point clouds with image votes. In *CVPR*, 2020.

[109] C. R. Qi, O. Litany, K. He, and L. J. Guibas. Deep hough voting for 3d object detection in point clouds. In *ICCV*, 2019.

[110] C. R. Qi, W. Liu, C. Wu, H. Su, and L. J. Guibas. Frustum pointnets for 3d object detection from rgb-d data. In *CVPR*, 2018.

[111] C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *CVPR*, 2017.

[112] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *NeurIPS*, 2017.

[113] Z. Qiu, T. Yao, and T. Mei. Learning spatio-temporal representation with pseudo-3d residual networks. In *ICCV*, 2017.

[114] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár. Designing network design spaces. In *CVPR*, 2020.

[115] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

[116] E. Real, C. Liang, D. So, and Q. Le. Automl-zero: Evolving machine learning algorithms from scratch. In *ICML*, 2020.

[117] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-scale evolution of image classifiers. In *ICML*, 2017.

[118] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *IJCV*, 2015.

[119] M. S. Ryoo, A. Piergiovanni, M. Tan, and A. Angelova. Assemblenet: Searching for multi-stream neural connectivity in video architectures. In *ICLR*, 2020.

[120] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[121] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[122] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *ICLR*, 2017.

[123] S. Shi, C. Guo, L. Jiang, Z. Wang, J. Shi, X. Wang, and H. Li. Pv-rcnn: Point-voxel feature set abstraction for 3d object detection. In *CVPR*, 2020.

[124] S. Shi, L. Jiang, J. Deng, Z. Wang, C. Guo, J. Shi, X. Wang, and H. Li. Pv-rcnn++: Point-voxel feature set abstraction with local vector representation for 3d object detection. *arXiv:2102.00463*, 2022.

[125] S. Shi, X. Wang, and H. Li. Pointrcnn: 3d object proposal generation and detection from point cloud. In *CVPR*, 2019.

[126] S. Shi, Z. Wang, J. Shi, X. Wang, and H. Li. From points to parts: 3d object detection from point cloud with part-aware and part-aggregation network. *TPAMI*, 2020.

[127] W. Shi and R. Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. In *CVPR*, 2020.

[128] K. Simonyan and A. Zisserman. Two-stream convolutional networks for action recognition in videos. In *NeurIPS*, 2014.

[129] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[130] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NeurIPS*, 2012.

[131] D. R. So, C. Liang, and Q. V. Le. The evolved transformer. In *ICML*, 2019.

[132] N. Srinivas, A. Krause, S. M. Kakade, and M. W. Seeger. Gaussian process optimization in the bandit setting: No regret and experimental design. In *ICML*, 2009.

[133] M. Streeter. Approximation algorithms for cascading prediction models. In *ICML*, 2018.

[134] J. Stroud, D. Ross, C. Sun, J. Deng, and R. Sukthankar. D3d: Distilled 3d networks for video action recognition. In *WACV*, 2020.

[135] E. Strubell, A. Ganesh, and A. McCallum. Energy and policy considerations for deep learning in nlp. In *ACL*, 2019.

[136] P. Sun, W. Wang, Y. Chai, G. Elsayed, A. Bewley, X. Zhang, C. Sminchisescu, and D. Anguelov. Rsn: Range sparse net for efficient, accurate lidar 3d object detection. In *CVPR*, 2021.

[137] K. Swersky, D. Duvenaud, J. Snoek, F. Hutter, and M. A. Osborne. Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. *arxiv:1409.4011*, 2014.

[138] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[139] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.

[140] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[141] M. Tan, R. Pang, and Q. V. Le. Efficientdet: Scalable and efficient object detection. In *CVPR*, 2020.

[142] O. D. Team. Openpcdet: An open-source toolbox for 3d object detection from point clouds. https://github.com/open-mmlab/OpenPCDet, 2020.

[143] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou. Fixing the train-test resolution discrepancy. In *NeurIPS*, 2019.

[144] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri. Learning spatiotemporal features with 3d convolutional networks. In *ICCV*, 2015.

[145] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, 2017.

[146] A. Veit and S. Belongie. Convolutional networks with adaptive inference graphs. In *ECCV*, 2018.

[147] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR*, 2001.

[148] N. Wadhwa, R. Garg, D. E. Jacobs, B. E. Feldman, N. Kanazawa, R. Carroll, Y. Movshovitz-Attias, J. T. Barron, Y. Pritch, and M. Levoy. Synthetic depth-of-field with a single-camera mobile phone. *ACM Transactions on Graphics (TOG)*, 2018.

[149] L. Wang, Y. Xiong, Z. Wang, Y. Qiao, D. Lin, X. Tang, and L. Van Gool. Temporal segment networks: Towards good practices for deep action recognition. In *ECCV*, 2016.

[150] X. Wang, S. Cao, M. Li, and K. M. Kitani. Neighborhood-aware neural architecture search. In *BMVC*, 2021.

[151] X. Wang and K. M. Kitani. Cost-aware evaluation and model scaling for lidar-based 3d object detection. *arXiv:2205.01142*, 2022.

[152] X. Wang, D. Kondratyuk, E. Christiansen, K. M. Kitani, Y. Alon, and E. Eban. Wisdom of committees: An overlooked approach to faster and more accurate models. In *ICLR*, 2022.

[153] X. Wang, X. Xiong, M. Neumann, A. Piergiovanni, M. S. Ryoo, A. Angelova, K. M. Kitani, and W. Hua. Attentionnas: Spatiotemporal attention cell search for video classification. In *ECCV*, 2020.

[154] X. Wang, R. Girshick, A. Gupta, and K. He. Non-local neural networks. In *CVPR*, 2018.

[155] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *ECCV*, 2018.

[156] Y. Wen, D. Tran, and J. Ba. Batchensemble: an alternative approach to efficient ensemble and lifelong learning. In *ICLR*, 2020.

[157] F. Wenzel, J. Snoek, D. Tran, and R. Jenatton. Hyperparameter ensembles for robustness and uncertainty quantification. In *NeurIPS*, 2020.

[158] C. K. Williams and C. E. Rasmussen. Gaussian processes for machine learning. *the MIT Press*, 2006.

[159] A. G. Wilson, Z. Hu, R. R. Salakhutdinov, and E. P. Xing. Stochastic variational deep kernel learning. In *NeurIPS*, 2016.

[160] A. G. Wilson, Z. Hu, R. Salakhutdinov, and E. P. Xing. Deep kernel learning. In *Artificial Intelligence and Statistics*, 2016.

[161] S. Woo, J. Park, J.-Y. Lee, and I. So Kweon. Cbam: Convolutional block attention module. In *ECCV*, 2018.

[162] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*, 2019.

[163] Z. Wu, T. Nagarajan, A. Kumar, S. Rennie, L. S. Davis, K. Grauman, and R. Feris. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*, 2018.

[164] L. Xie, X. Chen, K. Bi, L. Wei, Y. Xu, Z. Chen, L. Wang, A. Xiao, J. Chang, X. Zhang, et al. Weight-sharing neural architecture search: A battle to shrink the optimization gap. *arXiv:2008.01475*, 2020.

[165] L. Xie and A. Yuille. Genetic cnn. In *ICCV*, 2017.

[166] S. Xie, A. Kirillov, R. Girshick, and K. He. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019.

[167] S. Xie, C. Sun, J. Huang, Z. Tu, and K. Murphy. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *ECCV*, 2018.

[168] S. Xie, H. Zheng, C. Liu, and L. Lin. SNAS: stochastic neural architecture search. In *ICLR*, 2019.

[169] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. In *ICLR*, 2020.

[170] Y. Yan, Y. Mao, and B. Li. Second: Sparsely embedded convolutional detection. *Sensors*, 2018.

[171] A. Yang, P. M. Esperança, and F. M. Carlucci. Nas evaluation is frustratingly hard. In *ICLR*, 2020.

[172] B. Yang, W. Luo, and R. Urtasun. Pixor: Real-time 3d object detection from point clouds. In *CVPR*, 2018.

[173] Z. Yang, Y. Sun, S. Liu, and J. Jia. 3dssd: Point-based 3d single stage object detector. In *CVPR*, 2020.

[174] Z. Yang, Y. Sun, S. Liu, X. Shen, and J. Jia. Std: Sparse-to-dense 3d object detector for point cloud. In *ICCV*, 2019.

[175] Z. Yang, Y. Zhou, Z. Chen, and J. Ngiam. 3d-man: 3d multi-frame attention network for object detection. In *CVPR*, 2021.

[176] Z. Yao, A. Gholami, Q. Lei, K. Keutzer, and M. W. Mahoney. Hessian-based analysis of large batch training and robustness to adversaries. In *NeurIPS*, 2018.

[177] T. Yin, X. Zhou, and P. Krahenbuhl. Center-based 3d object detection and tracking. In *CVPR*, 2021.

[178] J. Yu, P. Jin, H. Liu, G. Bender, P.-J. Kindermans, M. Tan, T. Huang, X. Song, R. Pang, and Q. Le. Bignas: Scaling up neural architecture search with big single-stage models. In *ECCV*, 2020.

[179] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann. Evaluating the search phase of neural architecture search. In *ICLR*, 2020.

[180] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. Beyond short snippets: Deep networks for video classification. In *CVPR*, 2015.

[181] A. Zela, T. Elsken, T. Saikia, Y. Marrakchi, T. Brox, and F. Hutter. Understanding and robustifying differentiable architecture search. In *ICLR*, 2020.

[182] A. Zela, A. Klein, S. Falkner, and F. Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *arxiv:1807.06906*, 2018.

[183] X. Zhang, X. Zhou, M. Lin, and J. Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018.

[184] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu. Practical block-wise neural network architecture generation. In *CVPR*, 2018.

[185] B. Zhou, A. Andonian, A. Oliva, and A. Torralba. Temporal relational reasoning in videos. In *ECCV*, 2018.

[186] Y. Zhou, S. Ebrahimi, S. Ö. Arık, H. Yu, H. Liu, and G. Diamos. Resource-efficient neural architect. *arxiv:1806.07912*, 2018.

[187] Y. Zhou, P. Sun, Y. Zhang, D. Anguelov, J. Gao, T. Ouyang, J. Guo, J. Ngiam, and V. Vasudevan. End-to-end multi-view fusion for 3d object detection in lidar point clouds. In *CoRL*, 2019.

[188] Y. Zhou and O. Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *CVPR*, 2018.

[189] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

[190] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.