Hardware-Aware Neural Architecture Search and Compression Towards Embedded Intelligence

Xiangzhong Luo

School of Computer Science and Engineering

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

2023

Statement of Originality

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

29/11/2023

Date

Xiangzhong Luo

Supervisor Declaration Statement

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

29/11/2023

Date

Liu Weichen

Prof. Weichen Liu

Authorship Attribution Statement

This thesis contains material from four papers published in the following peer-reviewed conferences and journals in which I am listed as an author.

Chapter 3 is published as Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "SurgeNAS: A Comprehensive Surgery on Hardware-Aware Neural Architecture Search." IEEE Transactions on Computers (TC), 2022.

The contributions of the co-authors are as follows:

- I propose the initial idea, design the experiment, write the manuscript draft, and revise the manuscript draft according to the co-authors' comments.
- Prof. Di Liu revises the manuscript draft and provides valuable comments to improve the presentation quality.
- Hao Kong, Shuo Huai, and Hui Chen proofread the revised manuscript and provide valuable comments for further improvement.
- Prof. Weichen Liu helps discuss the feasibility of the initial idea, provides valuable comments to revise the manuscript draft, and provides funding grants and computational resources to support this research project.

Chapter 4 is partially published as Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "You Only Search Once: On Lightweight Differentiable Architecture Search for Resource-Constrained Embedded Platforms." ACM/IEEE Design Automation Conference (DAC), 2022.

The contributions of the co-authors are as follows:

- I propose the initial idea, design the experiment, write the manuscript draft, and revise the manuscript draft according to the co-authors' comments.
- Prof. Di Liu revises the manuscript draft and provides valuable comments to improve the presentation quality.
- Hao Kong, Shuo Huai, and Hui Chen proofread the revised manuscript and provide valuable comments for further improvement.
- Prof. Weichen Liu helps discuss the feasibility of the initial idea, provides valuable comments to revise the manuscript draft, and provides funding grants and computational resources to support this research project.

Chapter 4 is partially published as Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "LightNAS: On Lightweight and Scalable Neural Architecture Search for Embedded Platforms." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.

The contributions of the co-authors are as follows:

- I propose the initial idea, design the experiment, write the manuscript draft, and revise the manuscript draft according to the co-authors' comments.
- Prof. Di Liu revises the manuscript draft and provides valuable comments to improve the presentation quality.
- Hao Kong, Shuo Huai, and Hui Chen proofread the revised manuscript and provide valuable comments for further improvement.
- Prof. Weichen Liu helps discuss the feasibility of the initial idea, provides valuable comments to revise the manuscript draft, and provides funding grants and computational resources to support this research project.

Chapter 5 is published as Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, Shiqing Li, Guochu Xiong, and Weichen Liu, "Pearls Hide Behind Linearity: Simplifying Deep Convolutional Networks for Embedded Hardware Systems via Linearity Grafting." ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.

The contributions of the co-authors are as follows:

- I propose the initial idea, design the experiment, write the manuscript draft, and revise the manuscript draft according to the co-authors' comments.
- Prof. Di Liu revises the manuscript draft and provides valuable comments to improve the presentation quality.
- Hao Kong, Shuo Huai, Hui Chen, Shiqing Li, and Guochu Xiong proofread the revised manuscript and provide valuable comments for further improvement.
- Prof. Weichen Liu helps discuss the feasibility of the initial idea, provides valuable comments to revise the manuscript draft, and provides funding grants and computational resources to support this research project.

29/11/2023

Date

Xiangzhong Luo

Acknowledgements

I would like to express my deepest gratitude to my Ph.D. supervisor, friends, colleagues, and family members, whose invaluable mentorship and unwavering support have been of utmost importance throughout this interesting Ph.D. journey.

Xiangzhong Luo, July 2023

"If I had one hour to save the world, I would spend 55 minutes defining the problem and only five minutes finding the solution."

—Einstein, Albert

To my dear family

Abstract

With the increasing availability of large-scale datasets and powerful computing paradigms, convolutional neural networks (CNNs) have empowered a wide range of intelligent embedded vision tasks, which span from image classification to downstream vision tasks, such as on-device object recognition, detection, and tracking. In the past few years, convolutional networks have been evolving deeper and wider in order to maintain superior accuracy on target task. This rule of thumb, despite its efficacy, leads to an exponential growth in the number of floating-point operations (FLOPs) and parameters. For example, ResNet50, as one of the most representative convolutional networks, consists of over 4 billion FLOPs and 25 million parameters. The prohibitive network complexity, as a result, further enlarges the computational gap between computation-intensive CNNs and resource-constrained embedded platforms, making it challenging to develop hardware-friendly network solutions to accommodate the limited available computational resources in real-world embedded scenarios towards embedded intelligence.

This thesis focuses on alleviating the above computational gap from the perspective of hardware-aware neural architecture search (NAS) and compression.

First of all, we introduce SurgeNAS for efficient architecture search. Specifically, SurgeNAS turns back to one-level optimization for accurate and consistent gradient estimation, which also features an effective identity mapping scheme in order to avoid the search collapse. In addition, we introduce an efficient ordered differentiable sampling approach to reduce the memory consumption to the single-path level, while at the same time maintaining strict search fairness. An efficient graph neural networks (GNNs) based latency predictor is further proposed and integrated into the search engine to avoid tedious on-device latency measurements during the search process. Finally, we introduce the paradigm of *Comfort Zone*, which allows us to scale up the searched architecture candidates to achieve better accuracy on target task without degrading the inference efficiency on target hardware.

Furthermore, we introduce LightNAS for flexible architecture search. The motivation behind LightNAS is that previous relevant NAS methods, including SurgeNAS, simply focus on reducing the *explicit* search cost – the time for one single search, while ignoring the huge *implicit* search cost – the time for manual hyper-parameter tuning to derive the required architecture candidate. In practice, previous relevant NAS methods have to perform manual hyper-parameter tuning in order to navigate the required architecture candidate that satisfies the specified latency constraint, which empirically involves 10 trial-and-errors and thus significantly increases the total search cost by 10 times. In contrast, LightNAS only requires one single search for any specified latency constraint (i.e., *you only search once*). In addition, we introduce an efficient yet reliable proxy, namely batchwise training estimation (BTE), which can be seamlessly integrated into LightNAS to enable channel-level explorations at low computational cost. This further boosts the attainable accuracy on target task without degrading the efficiency on target hardware.

Finally, we introduce *Domino* for efficient network compression, in which we pioneer to revisit the trade-off dilemma between accuracy and efficiency from a fresh perspective of linearity and non-linearity. Specifically, *Domino* focuses on trading the less important network non-linearity for better network efficiency. To this end, *Domino* leverages two efficient performance predictors, including one vanilla latency predictor and one meta-accuracy predictor, to explore less important non-linear building blocks, which are then grafted with their linear counterparts. The resulting grafted network is further trained on target task to achieve decent accuracy. Finally, we reparameterize each grafted linear building block that consists of multiple consecutive linear layers, including multiple convolutional, batch normalization (BN), and grafted linear activation layers, into one single convolutional layer to aggressively boost the efficiency on target hardware, and more importantly, without sacrificing the accuracy on target task since the network maintains the same output regardless of linear reparameterization.

In summary, this thesis focuses on hardware-aware neural architecture search and compression to deliver efficient network solutions for resource-constrained embedded platforms to empower embedded intelligence. Future research will continue to explore more general search spaces and more advanced search/compression techniques to develop more efficient networks for intelligent embedded applications.

Contents

[1]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "SurgeNAS: A Comprehensive Surgery on Hardware-Aware Neural Architecture Search." IEEE Transactions on Computers (TC), 2022.

[2]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "You Only Search Once: On Lightweight Differentiable Architecture Search for Resource-Constrained Embedded Platforms." ACM/IEEE Design Automation Conference (DAC), 2022. and Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "LightNAS: On Lightweight and Scalable Neural Architecture Search for Embedded Platforms." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.

[3]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, Shiqing Li, Guochu Xiong, and Weichen Liu, "Pearls Hide Behind Linearity: Simplifying Deep Convolutional Networks for Embedded Hardware Systems via Linearity Grafting." ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Computational Gap in the Deep Learning Era

Convolutional neural networks (CNNs)[1], thanks to their strong learning capability, have demonstrated tremendous success in a myriad of real-world vision applications, spanning from image classification to many downstream vision tasks, such as object segmentation, detection, and tracking [26–30]. The origin of CNNs' booming success can be traced back to the early 2010s with the emergence of AlexNet [31], which for the first time reveals that CNNs, when properly engineered, can achieve human-like recognition performance. In the subsequent years, a huge amount of research effort has been dedicated to developing more and more powerful deep convolutional networks, such as VGGNet [32], ResNet [19], and DenseNet [33], and their tailored advanced training techniques, such as knowledge distillation [34] and deep mutual learning [35], which have been continuing to boost the attainable accuracy across various vision tasks in the deep learning era.

The tremendous success of CNNs has also garnered increasing interest from both academia and industry to integrate these powerful deep convolutional networks into real-world embedded platforms for the purpose of fostering embedded intelligence [26, 27, 36]. This has the potential to significantly enhance the user experience since

[1] In this thesis, we interchangeably use some technical terms like CNNs, DNNs, convolutional neural networks, convolutional networks, deep neural networks, and deep convolutional networks.

everything can be locally processed without being uploaded to the remote server, which thus protects the data privacy and security. However, these powerful deep convolutional networks, despite being able to exhibit competitive accuracy, often suffer from prohibitive network complexity, which typically involve a huge amount of floating-point operations (FLOPs) and parameters [26]. For example, ResNet-50 [19], as one of the most representative deep convolutional networks, consists of over 4 billion FLOPs and 25 million parameters. More recently, vision transformer (ViT) [28] and its variants [37–39] have demonstrated even stronger accuracy than their convolutional counterparts, which, unfortunately, also comes at the cost of more prohibitive network complexity [40]. And even worse, embedded platforms typically feature limited available computational resources in order to optimize the power consumption as shown in [36]. These limitations lead to and also continue to enlarge the computational gap between computation-intensive deep convolutional networks and resource-constrained embedded platforms [26].

To alleviate the above computational gap, substantial research has been conducted to develop computation-efficient lightweight convolutional networks, which strive to achieve superior accuracy with much lower computational complexity [26, 27, 36]. Among them, MobileNets [6, 41] and ShuffleNets [42, 43], as the most representative lightweight convolutional networks, have taken the very early steps and achieved promising progress. For example, MobileNetV1 [41] features depth-wise separable convolutional layers to trim down the network complexity to accommodate the limited available computational resources in real-world embedded scenarios. Furthermore, MobileNetV2 [6] borrows the residual learning paradigm [19] and introduces an efficient inverted residual structure, which can exhibit better accuracy-efficiency trade-offs than MobileNetV1. However, these lightweight convolutional networks, despite being able to deliver superior accuracy-efficiency trade-offs, involve a huge amount of engineering effort from human experts to explore the optimal network configuration, such as the optimal network width (i.e., the number of channels) and depth (i.e., the number of layers), through trial and error [26]. And even worse, prohibitive computational resources are also required since there exist numerous possible network configurations, each of which must be trained on target task in order to interpret its attainable accuracy [26].

To overcome such limitations, we, in this thesis, focus on hardware-aware neural architecture search (NAS) and compression to design more hardware-efficient

network solutions to further alleviate the computational gap between computation-intensive networks and resource-constrained embedded platforms towards boosted embedded intelligence, both of which can automatically explore top-performing network configurations and variants.

1.1.2 Neural Architecture Search

To explore novel network structures, recent network design practices have shifted from *manual* to *automated*, also referred to as neural architecture search (NAS) or automated machine learning (AutoML), which strives to automate the design of powerful deep convolutional networks. To this end, [44], as the seminal NAS work, proposes to train an effective recurrent neural networks (RNNs) based controller with reinforcement learning (RL), which, once well trained, can automatically explore the pre-defined search space to navigate the optimal network configuration. Experimental results on CIFAR-10 show that [44] can generate superior network structures that compete with early manual convolutional networks in terms of the accuracy. Subsequently, NASNet [2], following [44], leverages the same RNNs-based RL controller as the search engine to explore the pre-defined cell-based transferable search space, which further boosts the attainable accuracy on target task. However, these early RL-based NAS methods, despite being able to deliver promising results, suffer from prohibitive search cost, which may require thousands of GPU days for one single search experiment as demonstrated in [45].

To tackle the prohibitive search cost, ENAS [45] for the first time introduces the weight-sharing paradigm, which, thanks to its strong search efficiency, has been widely employed in subsequent NAS methods [3, 14, 46–48]. Specifically, ENAS proposes to initialize an over-parameterized supernet, which consists of all the possible architecture candidates in the pre-defined search space. During each search iteration, ENAS also features the same RL-based controller [44] to optimize different architecture candidates sampled from the over-parameterized supernet, in which all the architecture candidates share the same network weights from the over-parameterized supernet. As a result, in contrast to [2, 44] that have to train numerous stand-alone architecture candidates, ENAS only needs to train the over-parameterized supernet, which significantly reduces the search cost from thousands

of GPU days to less than one GPU day while at the same time achieving comparable accuracy to [2, 44]. The weight-sharing NAS paradigm can also be leveraged to perform evolutionary architecture search [14, 22, 48–50], in which an evolutionary search engine can be established to explore the pre-trained over-parameterized supernet to navigate the optimal architecture candidate. Furthermore, [5, 14, 22, 48] leverage reinforcement learning and evolutionary algorithm to perform hardware-aware search, which strive to navigate hardware-friendly architecture candidates that can exhibit superior accuracy-efficiency trade-offs.

More recently, the differentiable NAS paradigm dubbed DARTS [3] has emerged, which has dominated recent success in the field of NAS thanks to its strong search efficiency and reliable search performance [51]. Specifically, DARTS introduces a set of architecture parameters to relax the discrete search space to become continuous, which therefore allows us to optimize both the network weights and the architecture parameters using stochastic gradient descent (SGD). Once the differentiable optimization converges, we can interpret the optimal architecture candidate according to the learned architecture parameters [3]. The success of DARTS further inspires a plethora of subsequent differentiable NAS works [20, 21, 46, 47, 52–56], which continue to boost the search performance. In parallel, several popular NAS benchmarks have also been established, such as NAS-Bench-101 [57], NAS-Bench-201 [16], and NAS-Bench-x11 [58], to facilitate the NAS community. Despite the promising performance, the above differentiable NAS works [3, 20, 21, 46, 47, 52–56] simply focus on searching for top-performing architecture candidates in terms of the accuracy, while ignoring other important performance constraints, such as the on-device latency and energy. As a result, they end up with top-performing architecture candidates with competitive accuracy, which, however, comes at prohibitive computational complexity and thus cannot be deployed in real-world embedded scenarios due to the limited available computation resources.

Furthermore, several hardware-aware differentiable NAS works [9, 13, 59–61] are proposed, which focus on searching for hardware-efficient architecture candidates that can exhibit superior accuracy-efficiency trade-offs. These hardware-aware differentiable NAS works, despite being able to deliver superior accuracy-efficiency trade-offs, often suffer from considerable memory consumption and significant search unfairness. And even worse, unlike RL-based and evolutionary NAS works [7, 48, 62], these hardware-aware differentiable NAS works have to perform manual

hyper-parameter tuning in order to navigate the required architecture candidate that satisfies the specified latency constraint, which empirically involves 10 trial-and-errors and thus significantly increases the total search cost by 10 times.

1.1.3 Neural Architecture Compression

In parallel to neural architecture search that strives to explore novel network structures, neural architecture compression is another effective technique, which focuses on delivering efficient network variants towards boosted accuracy-efficiency trade-offs [26, 27]. In practice, network compression techniques can be roughly divided into three main categories, including pruning, quantization, and knowledge distillation [26, 27], which can be optionally plugged into typical NAS algorithms to jointly explore more efficient network solutions for resource-constrained embedded platforms [63]. Note that, in this thesis, we only consider structured pruning, including channel-wise and layer-wise pruning, since other common network compression techniques, such as quantization [64, 65] and knowledge distillation [34, 35], cannot end up with simplified network structures. For example, quantization seeks to reduce the network precision from high bits to lower bits (e.g., from 32 bits to 8 bits or even 1 bit), whereas the network structure still remains to be the same. In addition, we also exclude unstructured pruning (i.e., weight pruning) [66], which highly relies on specialized hardware accelerators for realistic on-device speedups and thus cannot benefit mainstream off-the-shelf embedded platforms due to the irregular network sparsity and memory access patterns [67].

To explore simplified network structures, previous representative channel-wise and layer-wise pruning methods remove the less important channels and layers to trim down the prohibitive network complexity. For example, [68] and [69] leverage L1-norm and Imprint to interpret the importance of different channels and layers, respectively, after which the less important channels and layers are removed to accommodate the specified performance requirement. Note that layer-wise pruning is an aggressive special case of channel-wise pruning and channel-wise pruning can be degraded to layer-wise pruning when all the channels in the same layer are pruned. However, channel-wise pruning, despite its efficacy to maintain competitive accuracy, often suffers from significant resource underutilization, which thus exhibits

inferior hardware efficiency [70, 71]. We conjecture that the resource underutilization here is caused by the layer-wise execution behaviors of convolutional networks, where the output of one specific layer is the input of its subsequent layer. In parallel, layer-wise pruning can indeed deliver superior hardware efficiency as shown in [69, 70, 72], which, unfortunately, comes at the cost of severe accuracy loss on target task since the attainable accuracy highly relies on sufficient network depth [19]. These demonstrate that channel-wise and layer-wise pruning can only achieve either competitive accuracy or hardware efficiency, and as a result, cannot achieve *Pareto-optimal* accuracy-efficiency trade-offs. This further calls for pruning-free alternatives to simplify deep convolutional networks towards boosted accuracy-efficiency trade-off to maximize the attainable accuracy on target task using the limited available computational resources in real-world embedded scenarios.

1.2 Major Contributions

In this thesis, we mainly focus on exploring hardware-aware neural architecture search and compression to deliver hardware-friendly network solutions for resource-constrained embedded platforms towards embedded intelligence. The main contributions of this thesis are three-fold, which are summarized as follows:

- We propose SurgeNAS for efficient architecture search. Specifically, SurgeNAS turns back to one-level optimization for more accurate and consistent gradient estimation during the search process, which also leverages a simple yet effective identity mapping scheme in order to tackle the search collapse. To alleviate the memory bottleneck, we introduce an efficient ordered differentiable sampling approach to aggressively reduce the required memory consumption to the single-path level, which can also maintain strict search fairness. Furthermore, we introduce an efficient graph neural networks (GNNs) based latency predictor, which is integrated into the search process to avoid tedious on-device latency measurements. Finally, we demonstrate that there exists *Comfort Zone*, which allows us to scale up the searched architecture candidates to achieve better accuracy without efficiency degradation.

- We propose LightNAS for flexible architecture search. The motivation behind LightNAS is that previous relevant NAS methods, including SurgeNAS, only

focus on reducing the explicit search cost for one single search, while ignoring the prohibitive implicit search cost for manual hyper-parameter tuning in order to navigate the required architecture candidate. That is, previous relevant NAS methods have to repeat multiple search experiments through trial and error to navigate the required architecture candidate that satisfies the specified latency constraint, which empirically leads to 10 trial-and-errors and thus significantly increases the total search cost by 10 times. In contrast, LightNAS only involves one single search for any specified latency constraint (i.e., *you only search once*), which thus exhibits significant search efficiency and flexibility. In addition, we introduce a reliable yet computationally cheap proxy, namely batchwise training estimation (BTE), to enable channel-level explorations for further accuracy improvement at low computational cost.

- We propose *Domino* for efficient network compression, in which we strive to revisit the accuracy-efficiency trade-off from a fresh perspective of linearity and non-linearity. Specifically, in contrast to previous relevant methods that trade the less important channels or layers, *Domino* trades the less important network non-linearity for better hardware efficiency. To this end, *Domino* features two efficient predictors, including one vanilla latency predictor and one meta-accuracy predictor, to explore the less important non-linear building blocks, which are then grafted with their linear counterparts. The resulting grafted network is further trained on target task to achieve superior accuracy. Finally, we reparameterize each grafted linear building block that consists of multiple consecutive linear layers into one single linear layer to win aggressive hardware efficiency without accuracy loss since the network output remains strictly the same regardless of linear reparameterization.

1.3 Thesis Organization

The structure of this thesis is as follows, which can also be found in Figure 1.1. Chapter 1 introduces the research background and presents an overview of this thesis. Chapter 2 introduces the relevant literature. Chapter 3 introduces SurgeNAS for efficient architecture search. Chapter 4 introduces LightNAS for flexible architecture search. Chapter 5 introduces *Domino* for efficient network compression.

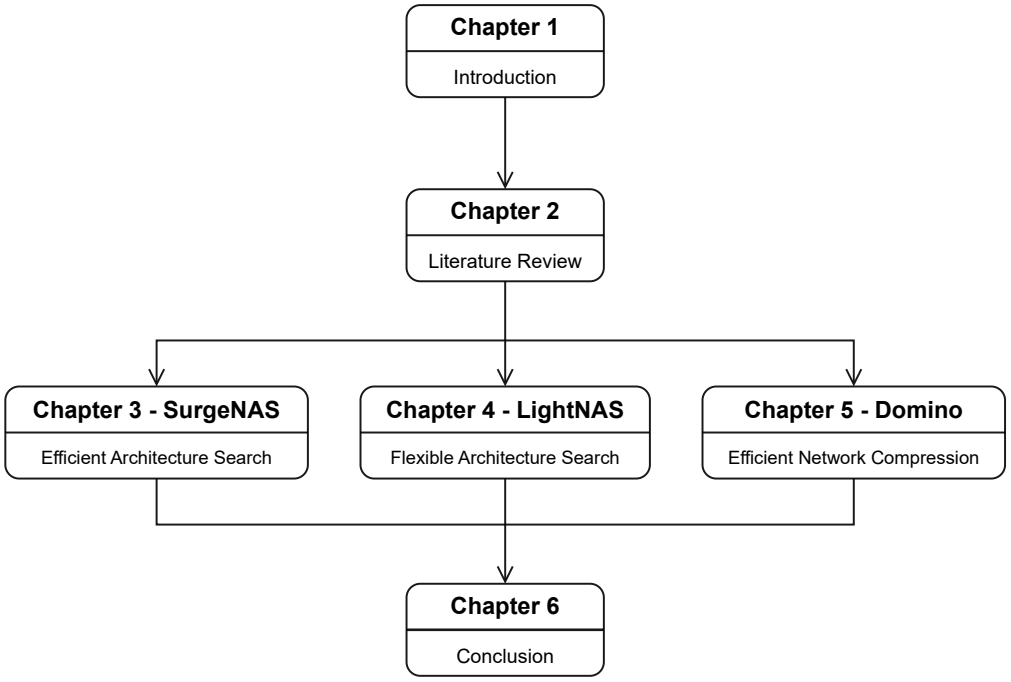

FIGURE 1.1: An intuitive overview of the systematic structure of this thesis.

Finally, Chapter 6 concludes this thesis and discusses possible future directions towards more efficient networks and more advanced embedded intelligence.

Chapter 2

Literature Review

Convolutional neural networks have achieved tremendous success across a myriad of real-world intelligent embedded scenarios, such as AR/VR and on-device object detection/tracking [27, 36]. In the past decade, "*deeper + wider = better*" has been widely deemed as the rule of thumb to design networks that can exhibit superior accuracy [19, 33]. This rule, despite its efficacy to maintain superior accuracy, continues to push forward the network complexity over time, which, as a result, leads to the computational gap between computation-intensive networks and resource-constrained embedded platforms [26]. To this end, we, in this section, present the background and discuss recent advances from the perspective of addressing the aforementioned computational gap to deliver efficient network solutions towards embedded intelligence, including efficient convolutional network design in Section 2.1 and efficient convolutional network compression in Section 2.2.

2.1 Efficient Convolutional Network Design

2.1.1 Manual Convolutional Network Design

Early representative convolutional networks, such as AlexNet [31], VGGNet [32], GoogleNet [73], ResNet [19], and DenseNet [74], despite being able to push forward the attainable accuracy on ImageNet [1] from 57.2% [75] to 77.9% [33], often exhibit prohibitive network complexity. Note that convolutional networks typically consist of convolutional layers, pooling layers, and fully-connected layers, where

most of the network complexity comes from convolutional layers as demonstrated in [27]. In sight of this convention, below we first introduce five representative efficient convolutional layers, including pointwise convolution, groupwise convolution, depthwise convolution, dilated convolution, and Ghost convolution:

- **Pointwise Convolution.** Pointwise convolution is a type of convolutional layer with the fixed kernel size of 1×1, which performs element-wise multiplications and additions along the depth dimension. Compared with standard $K \times K$ convolution, 1×1 pointwise convolution is able to reduce the number of FLOPs and parameters by K^2 times, which thus exhibits significant computational efficiency. In addition, the output from pointwise convolution typically maintains the same spatial dimension as the input except for the number of channels. As such, pointwise convolution can be used to adjust the intermediate feature maps in terms of the number of channels, making it a popular technique to compress or expand convolutional networks [6, 19].

- **Groupwise Convolution.** Groupwise convolution is a type of convolutional layer that (1) divides the input feature maps into G groups along the depth dimension, (2) convolves each group of input feature maps, and (3) concatenates all the outputs along the depth dimension to produce the final output. For example, given an input feature map of $B \times C \times H \times W$, each kernel within $K \times K$ groupwise convolution is with the size of $(C/G) \times K \times K$, which is employed to convolve the above G groups of input feature maps, respectively. Therefore, compared with standard $K \times K$ convolution, groupwise convolution can reduce the number of FLOPs and parameters by G times.

- **Depthwise Convolution.** Depthwise convolution is a type of convolutional layer that has gained increasing popularity, thanks to its strong ability to reduce the number of FLOPs and parameters. In fact, depthwise convolution is a special case of groupwise convolution, in which the number of groups G is equivalent to the number of input channels. As such, each input channel is convolved with a unique kernel of $1 \times K \times K$, after which the outputs from all the input channels are concatenated along the depth dimension to derive the final output. In practice, depthwise convolution has the potential to achieve significant reduction in the number of FLOPs and parameters because the intermediate feature maps may consist of thousands of channels as shown in previous state-of-the-art convolutional networks [19, 33].

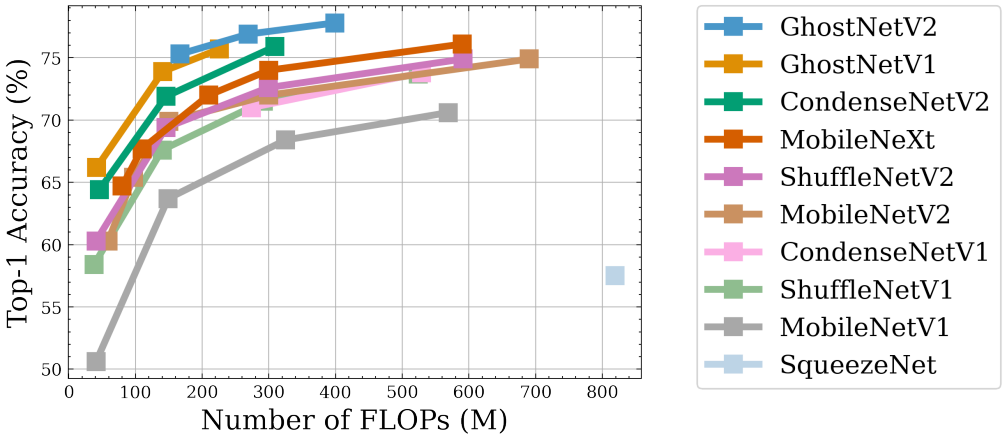

FIGURE 2.1: Comparisons of efficient manual convolutional networks, in which the accuracy is evaluated on ImageNet [1] and is taken from the respective paper.

- **Dilated Convolution.** Dilated convolution, also known as atrous convolution, is a type of convolutional layer, which strives to increase the receptive field without introducing additional computational overheads. To this end, dilated convolution introduces an adjustable hyper-parameter, namely dilation rate, which controls the spacing between different elements and can be varied to adjust the size of the receptive field. For example, 3×3 dilated convolution with an input dilation rate of 1 maintains the same receptive field as standard 5×5 convolution. This allows us to increase the receptive field size to obtain better accuracy without increasing the network complexity.

- **Ghost Convolution.** Ghost convolution is a type of convolutional layer, which seeks to generate richer features using cheaper computational resources. Specifically, Ghost convolution consists of two stages. The first stage corresponds to standard convolution, in which the number of output channels is rigorously controlled [76]. In the second stage, a series of simple linear operations are applied to the output features from the first stage. As a result, the number of output features still remains the same as standard convolution, but the total required computational resources are significantly reduced.

Built on top of the aforementioned efficient convolutional layers, there have been a plethora of representative families of efficient manual convolutional networks, including SqueezeNet [75], MobileNets [6, 41, 77], ShuffleNets [42, 43], CondenseNets [78, 79], and Ghostnets [76, 80]. For example, MobileNetV1 [41], as the first of its kind, introduces depthwise convolution to capture rich features with significantly

reduced computational resources. Furthermore, MobileNetV2 [6] introduces the inverted residual block, which features pointwise convolution and depthwise convolution, to deliver better accuracy-efficiency trade-offs. In parallel, ShuffleNets [42, 43] are another representative family of efficient convolutional networks, which leverage depthwise convolution and also an effective channel shuffle module to achieve competitive accuracy and promising computational efficiency. These efficient manual convolutional networks, thanks to their strong accuracy-efficiency trade-offs as shown in Figure 2.1, have been widely deployed in a wide range of real-world resource-constrained scenarios in pursuit of embedded intelligence [26, 27].

2.1.2 Automated Convolutional Network Design

The aforementioned efficient manual convolutional networks [6, 41–43, 75–80], despite their promising accuracy-efficiency trade-offs, necessitate a huge amount of engineering effort from human experts to determine the optimal network configuration, such as the network depth and width, through trial and error [27]. The tedious trial-and-error process also involves non-trivial computational resources since there are prohibitive network configurations and we have to train every possible network configuration from scratch to interpret the attainable accuracy [26]. To overcome such limitations, recent efficient network design practices have been shifting from *manual* to *automated*, also referred to as neural architecture search (NAS) or automated machine learning (AutoML), which seeks to automatically explore the pre-defined search space to navigate the optimal network configuration [44]. In practice, NAS typically consists of two main components, including the search space and the search strategy, both of which are of utmost importance in order to obtain network solutions with superior accuracy-efficiency trade-offs [44]. Below we discuss previous well-established search spaces and search strategies, followed by efficient search techniques to speed up the search process.

2.1.2.1 Architecture Search Spaces

The search space plays a prominent role in the success of NAS since the search engine of NAS strives to search for top-performing architecture candidates within the pre-defined search space. This demonstrates that the search space determines the upper-performance limit of NAS algorithms. However, designing efficient and

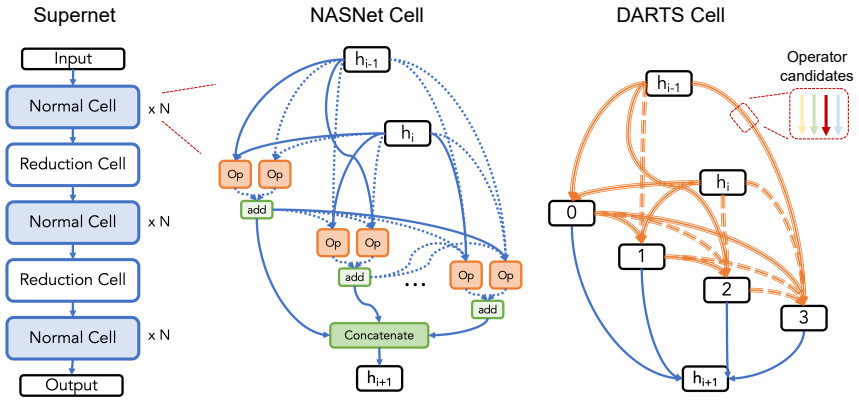

FIGURE 2.2: Illustration of two popular cell-based search spaces in NASNet [2] and DARTS [3], in which NASNet assigns operator candidates to nodes and DARTS assigns operator candidates to directed edges. (figure from [4])

effective search spaces may pose significant challenges since there have numerous possible operators (e.g., 1×1, 3×3, 5×5, and 7×7 convolutional layers) and network structures (e.g., the channel layout) [2, 4]. To overcome such limitations, previous state-of-the-art NAS methods [2, 3, 9, 13, 44] instead restrict the search space to enable efficient search, which typically leverage modular search spaces, including cell-based search space and block-based search space.

Cell-Based Search Space. The cell-based search space has dominated the early success in the field of NAS [51]. Specifically, the cell-based search space is first introduced in NASNet [2] and subsequently generalized in DARTS [3]. As defined in NASNet, the cell-based search space consists of two types of cell structures, which are denoted as the normal and reduction cells. In practice, both types of cells are encoded into directed acyclic graphs (DAGs) as illustrated in Figure 2.2 and maintain the same cell structure with one exception. That is, the reduction cell starts with one convolutional layer with the stride of 2 in order to reduce the input spatial dimension. Once the cell structure is determined at the end of search, it will be repeatedly stacked to derive the final architecture candidate. Furthermore, DARTS simplifies and generalizes the search space in NASNet, which has since been widely followed in subsequent NAS methods, such as PC-DARTS [46], P-DARTS [47], DARTS+ [52], and EdgeNAS [81]. Similar to NASNet, the cell-based search space in DARTS consists of two types of cells, including the normal and reduction cells. As shown in Figure 2.2, each cell has an ordered sequence of nodes, where each node is a latent representation (e.g., a feature map in convolutional networks)

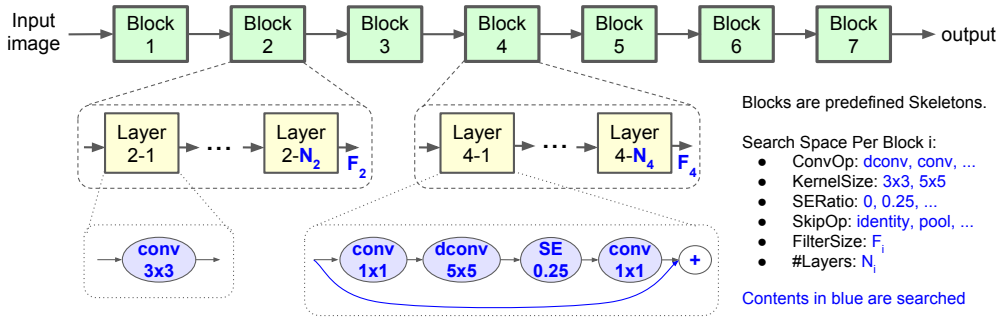

FIGURE 2.3: Illustration of the block-based search space in MnasNet [5], which is built upon MobileNetV2 [6] and MobileNetV3 [7]. (figure from [5])

and each directed edge has a set of possible operators that transform the input. It is worth noting that, different from NASNet, the cell in DARTS is assumed to have two different input nodes and one single output node.

Block-Based Search Space. The block-based search space features simple and diverse network topologies as illustrated in Figure 2.3, in which each architecture candidate consists of multiple sequential operator candidates. As shown in previous NAS practices, the operator candidates in the block-based search space are typically taken from state-of-the-art manual convolutional networks, such as MobileNets [6, 41] and ShuffleNets [42, 43]. For example, the block-based search space in ProxylessNAS [13] is built on top of MobileNetV2, whereas the block-based search space in HSCoNAS [82] is built on top of ShuffleNetV2. In parallel, HURRICANE [83] demonstrates that different hardware platforms favor different search spaces, based on which HURRICANE introduces a hybrid block-based search space that combines both MobileNetV2 and ShuffleNetV2 to deliver superior architecture solutions. Different from the cell-based search space, the block-based search space is hardware-friendly, due to which the block-based search space has been widely adopted in previous hardware-aware NAS methods, such as MnasNet [5], ProxylessNAS [13], OFA [14], HSCoNAS [82], SurgeNAS [84], and LightNAS [85]. The intuition behind this is that the architecture candidate in the cell-based search space consists of multiple parallel branches as shown in Figure 2.2, which introduce additional overheads in terms of the memory access, and as a result, deteriorate the inference efficiency on target hardware according to the roofline analysis [8]. In addition, different from the cell-based search space that repeatedly stacks the same cell structure across the entire network, the block-based search space allows operator diversity within different blocks, enabling to explore architecture candidates

that can exhibit better accuracy-efficiency trade-offs as shown in [5].

Remarks. As shown in recent representative NAS works [9, 13], the search space of NAS typically has a large number of architecture candidates, making it difficult to provide an effective theoretical model to show how the search space is explored. As a workaround, we may instead evaluate different search algorithms, including reinforcement learning-based search, evolutionary algorithm-based search, and gradient-based search, on some popular NAS benchmarks like NAS-Bench-201 [16]. Specifically, NAS-Bench-201 features a small search space with 15,625 different architecture candidates, all of which are trained from scratch using the same training recipe. This allows us to make fair comparisons between different search algorithms and also analyze how the searched networks differ from the optimal ones.

2.1.2.2 Architecture Search Strategies

In this section, we discuss previous representative search strategies, which can be divided into three main categories, including reinforcement learning-based search, evolutionary algorithm-based search, and gradient-based search.

Reinforcement Learning-Based Search. In the field of NAS, [44] is the first NAS work[1] that opens up the possibility of automating the design of top-performing deep networks. Specifically, [44] features reinforcement learning (RL) [87] and leverages an effective recurrent neural networks (RNNs) based controller to generate possible architecture candidates. The generated architecture candidate is then trained from scratch on target task to evaluate its attainable accuracy. The accuracy of the generated architecture candidate is further fed back into the above RNNs-based controller, which optimizes the RNNs-based controller to generate better architecture candidates in the next iteration. Once the search process terminates, the well-optimized RNNs-based controller is able to generate top-performing architecture candidates that exhibit superior accuracy on target task. The promising performance of [44] marks an important milestone in the field of NAS, which, for the first time, demonstrates the possibility of automatically exploring top-performing architecture candidates. Furthermore, NASNet [2] leverages the same search engine as

[1]MetaQNN [86] is another seminal NAS work in parallel to [44], both of which feature reinforcement learning as the search engine to automate the design of top-performing networks.

[44] and introduces the flexible cell-based search space, which further pushes forward the attainable accuracy on target task. For example, NASNet achieves 97.60% top-1 accuracy on CIFAR-10, which is +1.25% higher than [44] while at the same time involving fewer parameters (i.e., 37.4 M parameters in [44] vs. 27.6 M parameters in NASNet). Despite the promising performance, both [44] and NASNet have to train a large number of stand-alone architecture candidates (e.g., 12,800 in [44]) from scratch and thus necessitate considerable computational resources. To overcome such limitations, ENAS [45] introduces an effective weight sharing NAS paradigm, also known as parameter sharing NAS. Specifically, ENAS allows all the possible architecture candidates to share network weights from the supernet, which thus avoids training every possible architecture candidate from scratch. This weight sharing paradigm leads to significant reduction in terms of the search cost, while at the same time still maintaining strong accuracy on target task. Thanks to its significant search efficiency, the weight sharing NAS paradigm has been widely adopted in subsequent NAS works [3, 9, 13, 14, 48, 50, 59].

The early RL-based NAS works [2, 44, 45] have achieved tremendous success, which, however, focus on accuracy-only optimization and ignore other important performance metrics, such as the on-device latency and energy. To this end, MnasNet [5] formulates the RL-based search process as multi-objective optimization, which simultaneously optimizes both accuracy and mobile latency as shown in Figure 2.4. Specifically, MnasNet features the hardware-friendly block-based search space $\mathcal{A}$ as shown in Figure 2.3 and introduces an effective multi-objective RL reward function to optimize the RL-based controller, which explores *Pareto-optimal* architecture candidates that can maximize the following multi-objective RL reward function:

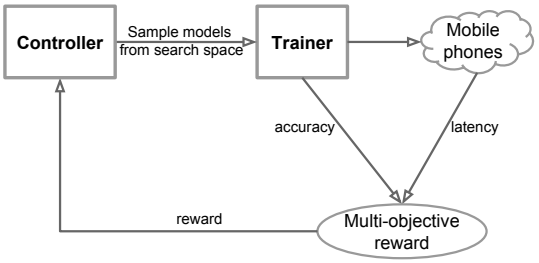

FIGURE 2.4: An intuitive overview of MnasNet [5]. (figure from [5])

$$\underset{arch \in \mathcal{A}}{\text{maximize}} \quad Accuracy(arch) \times \left[\frac{Latency(arch)}{T}\right]^{w} \tag{2.1}$$

where $Accuracy(\cdot)$ and $Latency(\cdot)$ denote the accuracy on target task and the

latency on target hardware, respectively. Besides, T is the specified latency constraint. Apart from these terms, w is the trade-off coefficient to control the trade-off magnitude between accuracy and latency, which is defined as follows:

$$w = \begin{cases} \alpha, & \text{if } Latency(arch) \leq T \\ \beta, & \text{otherwise} \end{cases} \tag{2.2}$$

where α and β are application-specific hyper-parameters to control the trade-off magnitude between accuracy and latency. In practice, according to the empirical observation that doubling the latency typically brings ∼5% relative accuracy improvement, MnasNet empirically fixes both α and β to -0.07 [5].

Evolutionary Algorithm-Based Search. Evolutionary algorithm-based search is another popular NAS branch in the field of NAS, which can be traced back to as early as the 1980s [88–91]. In practice, evolutionary algorithm-based search has gained increasing popularity in the NAS community, thanks to its strong flexibility, conceptual simplicity, and competitive search performance [49]. Specifically, evolutionary algorithm-based search typically consists of four key steps, including (1) initializing a set of possible architecture candidates from the search space as the child population, (2) evaluating possible architecture candidates in the child population to interpret their performance like accuracy and efficiency, (3) reserving the top-k architecture candidates in the latest child population to form the parent population and discarding the remaining architecture candidates in the latest child population, and (4) manipulating the architecture candidates in the latest parent population to produce new architecture candidates to form the next-generation child population. The above four steps are repeated until the evolutionary process converges or the specified performance requirement is satisfied.

Evolutionary algorithm-based NAS works have recently flourished after the weight-sharing NAS paradigm is established [45, 50]. Specifically, [50] demonstrates the possibility of training an over-parameterized supernet, which, once well trained, can be leveraged to reliably estimate the accuracy of possible architecture candidates using minimal computational resources[2]. This makes it possible to employ evolutionary algorithm to explore the prohibitive search space, which also significantly reduces the required computational resources since we do not have to train

[2]The training process involves prohibitive computational resources since it requires a large number of iterations to converge, where the evaluation process is typically computationally cheap.

every possible architecture candidate [50]. Furthermore, inspired by the promising results of [50], SPOS [48] introduces a simple yet effective single-path one-shot NAS paradigm to train the over-parameterized supernet, which discretizes single-path sub-networks from the supernet and thus exhibits significant memory efficiency during the training process. The success of SPOS paves the way for subsequent evolutionary algorithm-based NAS works [14, 22, 83, 92–94]. Note that the above follow-up NAS works mainly focus on training an effective and reliable supernet, which can serve as an accurate performance predictor to quickly interpret the accuracy of different architecture candidates. For example, FairNAS [22] demonstrates that the uniform sampling scheme in SPOS only implies soft search fairness, which may suffer from degraded search performance. To overcome such limitations, FairNAS instead samples multiple sub-networks during each training iteration to enforce strict search fairness. In parallel, OFA [14] introduces the once-for-all paradigm, which strives to train an over-parameterized supernet and then specialize it for diverse deployment scenarios with minimal computational resources. Despite the promising success, OFA still involves several fine-tuning epochs to recover the attainable accuracy. To this end, BigNAS [93] focuses on unleashing the promise of the over-parameterized supernet, making it possible to specialize sub-networks from the well-trained supernet for direct deployment on target hardware without being re-trained or fine-tuned on target task.

Thanks to its strong search flexibility, evolutionary algorithm-based NAS can be easily extended to search for hardware-efficient architecture candidates, which maximize the accuracy on target task while satisfying various real-world performance constraints [83], such as latency, energy, memory, etc. Without loss of generality, we may consider the following multi-objective optimization:

$$\underset{arch \in \mathcal{A}}{\text{maximize}} \;\; Acc(arch) \;\; s.t., \;\; Constraint_1(arch) \leq C_1, ..., Constraint_n(arch) \leq C_n \tag{2.3}$$

where $\{C_i\}_{i=1}^{n}$ correspond to a set of real-world performance constraints.

Gradient-Based Search. In addition to reinforcement learning-based search and evolutionary algorithm-based search, gradient-based search [3], also known as differentiable search, is another representative branch of NAS, which has dominated recent advances in the field of NAS, thanks to its strong search efficiency [4, 51, 95]. Specifically, DARTS [3], as the first-of-its-kind differentiable NAS, can provide competitive architecture candidates in $\sim$1 day on one single Nvidia GTX 1080 Ti GPU.

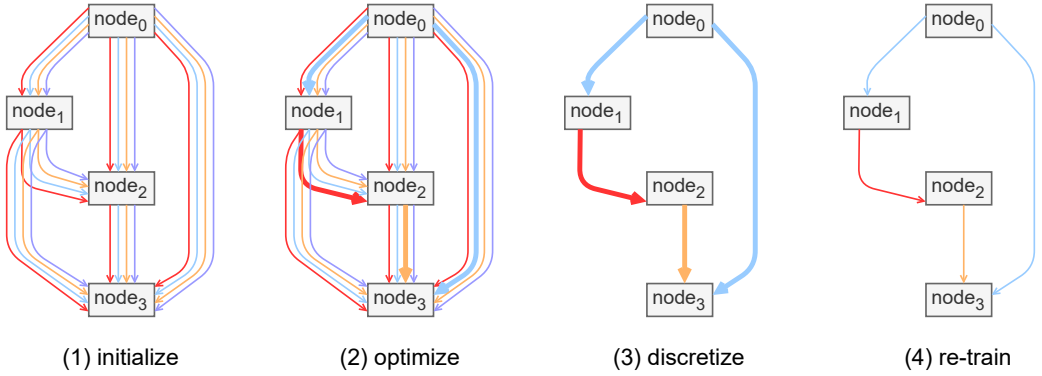

FIGURE 2.5: Overview of DARTS [3], which has four stages, including (1) initializing w and α in the supernet, (2) optimizing w and α via alternating gradient descent, (3) discretizing the optimal architecture candidate from the supernet, and (4) re-training the optimal architecture candidate to recover the accuracy.

In contrast to previous non-differentiable NAS practices [2, 44, 45, 50] that rely on discrete search spaces, DARTS leverages a set of architecture parameters α to relax the discrete search space to be continuous. Thanks to the above continuous relaxation, both network weights w and architecture parameters α can be optimized with alternating gradient descent. Once the differentiable optimization converges or terminates, we can interpret the optimal architecture candidate from the architecture parameters α. The supernet of DARTS is initialized with multiple sequential over-parameterized cells as shown in Figure 2.5, in which each over-parameterized cell contains all the possible cell structures in the cell-based search space $\mathcal{A}$. As shown in Figure 2.5, each over-parameterized cell is represented using the directed acyclic graph (DAG) that consists of N nodes $\{x_i\}_{i=1}^{N}$. Note that the nodes here correspond to the intermediate feature maps. Besides, the directed edges between x_i and x_j correspond to a list of possible operator candidates $\{o|o \in \mathcal{O}\}$, which are also assigned with a list of architecture parameters $\{\alpha_o^{(i,j)}|o \in \mathcal{O}\}$. With the above in mind, we can formulate the intermediate output x_j as follows:

$$x_j = \sum_{o \in \mathcal{O}} \frac{\exp \alpha_o^{(i,j)}}{\sum_{o' \in \mathcal{O}} \exp \alpha_{o'}^{(i,j)}} o(x_i) \tag{2.4}$$

Note that the output x_j is continuous with respect to x_i, α, and w [3]. To this end, DARTS optimizes α and w using the following bi-level differentiable optimization:

$$\underset{\alpha}{\text{minimize}} \quad \mathcal{L}_{val}(w^*(\alpha), \alpha) \quad s.t., \quad w^*(\alpha) = \arg\min_{w} \mathcal{L}_{train}(w, \alpha) \tag{2.5}$$

where $\mathcal{L}_{train}(\cdot)$ and $\mathcal{L}_{val}(\cdot)$ are the loss functions accumulated on the training and

validation datasets, respectively. Once the above bi-level differentiable optimization converges, DARTS determines the optimal architecture candidate by reserving the strongest operator $\alpha_o^{(i,j)}$ and removing the remaining operators between x_i and x_j, in which the operator strength is defined as $\exp \alpha_o^{(i,j)} / \sum_{o' \in \mathcal{O}} \exp \alpha_{o'}^{(i,j)}$. Finally, the searched architecture candidate is re-trained on target task to recover its attainable accuracy for further deployment on target hardware.

Inspired by the promising performance of DARTS, a plethora of follow-up NAS works [20, 21, 46, 47, 52, 54, 56, 96] have recently emerged, which strive to unleash the power of differentiable NAS so as to deliver superior architecture candidates that can exhibit better accuracy on target task. For example, unlike DARTS that simultaneously optimizes all the operator candidates during the search process, PC-DARTS [46] introduces partial channel connections to alleviate the excessive memory consumption and thus maintains promising search efficiency. Besides, DARTS+ [52] investigates the performance collapse of DARTS and proposes a simple yet effective early stopping strategy to tackle the performance collapse. In parallel, DARTS- [54] also observes the performance collapse of differentiable NAS, which further tailors an auxiliary skip connection to mitigate the performance collapse and also stabilizes the search process. Furthermore, Single-DARTS [96] and Gold-NAS [56] demonstrate that the bi-level differentiable optimization may introduce considerable search bias due to the inaccurate gradient estimation. To this end, Single-DARTS and Gold-NAS turn back to the one-level differentiable optimization for more accurate gradient estimation. To boost the search efficiency, GDAS [21] introduces an efficient Gumbel-Softmax [10] based differentiable sampling approach to reduce the memory consumption to the single-path level.

The aforementioned early differentiable NAS methods [20, 21, 46, 47, 52, 54, 56, 96], however, focus on accuracy-only optimization, which indeed show competitive accuracy but fail to accommodate the limited available computational resources in real-world embedded scenarios [26, 27]. To overcome such limitations, the paradigm of hardware-aware differentiable NAS has recently flourished, which focuses on finding hardware-efficient architecture candidates that can exhibit superior accuracy and hardware efficiency [9, 13, 25, 59, 97, 98]. To this end, these hardware-aware differentiable NAS methods typically integrate the on-device latency into the search process as soft constraints to penalize the architecture candidate with high on-device latency, which thus end up with efficient architecture candidates

that maintain both superior accuracy and hardware efficiency [95]. Specifically, the hardware-aware differentiable optimization can be formulated as follows:

$$\underset{\alpha}{\text{minimize}} \;\; \mathcal{L}_{val}(w^*(\alpha), \alpha) + \lambda \cdot Latency(\alpha) \;\; s.t., \;\; w^*(\alpha) = \arg\min_{w} \mathcal{L}_{train}(w, \alpha) \tag{2.6}$$

where λ is the constant coefficient to control the trade-off magnitude between accuracy and latency. In practice, a larger λ end up with the architecture candidate that exhibits low accuracy and low latency, whereas a smaller λ leads to the architecture candidate with high accuracy and high latency [85]. The optimization objective in Eq (2.6) can also be easily generalized to optimize other hardware constraints, such as energy and memory consumption. For example, we can simply re-formulate the optimization objective in Eq (2.6) as follows to search for efficient architecture candidates in terms of the on-device latency, energy, and memory consumption:

$$\underset{\alpha}{\text{minimize}} \;\; \mathcal{L}_{val}(w^*(\alpha), \alpha) + \lambda_1 \cdot Latency(\alpha) + \lambda_2 \cdot Energy(\alpha) + \lambda_3 \cdot Memory(\alpha) \tag{2.7}$$

where λ_1, λ_2, and λ_3 are the constant coefficients to control the trade-off magnitudes between accuracy and latency, energy, and memory consumption, respectively.

2.1.2.3 Speedup Search Techniques

In this section, we discuss popular speedup search techniques, including latency prediction, accuracy prediction, low-cost proxies, and zero-cost proxies, which have the potential to significantly accelerate the search process.

Latency Prediction. As shown in MnasNet [5], the on-device latency is directly measured on target hardware, which is further integrated into the multi-objective RL reward function to penalize possible architecture candidates with high latency. The direct on-device latency measurement is indeed accurate, which, however, is time-consuming and cannot scale to large search spaces [13, 95]. To overcome such limitations, several efficient latency prediction strategies have been recently established. For example, [9, 13, 14, 61, 82, 99] leverage the latency lookup table to approximate the on-device latency for different architecture candidates. In parallel, [12, 24, 81, 85, 92, 100, 101] turn back to learning-based regression approaches for the latency prediction purpose, which typically train an accurate latency predictor

using fixed number of latency measurements, which then can generalize to predict the on-device latency for unseen architecture candidates.

Accuracy Prediction. In parallel to latency prediction, accuracy prediction has also gained increasing popularity in the NAS community [11, 102–104], which strives to predict the accuracy of different architecture candidates in the search space. Among them, [11] introduces a simple yet effective graph convolutional networks (GCNs) based accuracy predictor, which delivers promising accuracy prediction performance thanks to GCNs' strong capability to interpret graph-structured data. In addition, NASLib [103] investigates a plethora of accuracy predictors on multiple NAS benchmarks, which demonstrates that different accuracy predictors, when properly engineered, can achieve much stronger accuracy prediction performance than any single accuracy predictor. Furthermore, DONNA [104] proposes to build an efficient accuracy predictor, which only involves minimal computational resources, and more importantly, can scale to diverse search spaces.

Low-Cost Proxies. Instead of training the given architecture candidate for hundreds of epochs, a plethora of low-cost proxies [15, 58, 105–108] have recently emerged, which strive to interpret the attainable accuracy of the given architecture candidate only using its early training statistics, such as the training loss in the first few training epochs. For example, in contrast to previous well-established accuracy predictors [11, 103] that only use the network configuration as input to predict the accuracy, [107] proposes to combine the network configuration and its validation accuracy in the first few training epochs, which are used to train an effective regression model to predict the accuracy for unseen architecture candidates. Similarly, [15] introduces training speed estimation (TSE) that simply accumulates the early training statistics, which can achieve reliable yet computationally cheap ranking performance among different architecture candidates.

Zero-Cost Proxies. In addition to the above low-cost proxies, zero-cost proxies have recently flourished [17, 18, 109–118], which typically focus on interpreting the performance of the given architecture candidate in training-free manners. Specifically, zero-cost proxies, such as EPE [109], Fisher [110], GradNorm [17], Grasp [111], Jacov [18], Snip [112], Synflow [113], ZenScore [114], LRC [117], and NTK [118], can provide reliable performance estimation using few or even one single mini-batch of data, which thus only necessitate near-zero computational cost as

shown in [17, 116]. Thanks to their reliable performance estimation and low computational cost, these zero-cost proxies have been widely adopted in recent NAS works to accelerate the search process [18, 115, 116]. In practice, we can also feature multiple zero-cost proxies, which can deliver more reliable performance estimation than any single zero-cost proxy as demonstrated in [18, 116].

2.2 Efficient Convolutional Network Compression

In this section, we discuss recent efficient network compression techniques, including channel-wise pruning and layer-wise pruning, which strive to deliver simplified network variants with less computational complexity. Note that we do not consider unstructured pruning (i.e., weight pruning), which relies on specialized hardware accelerators [67] and thus cannot benefit mainstream embedded platforms due to the irregular network sparsity and memory access patterns [26, 27, 119].

2.2.1 Structured Channel-Wise Pruning

Channel-wise pruning removes the less important channels to reduce the computational redundancy towards better hardware efficiency. In practice, channel-wise pruning can be roughly divided into four categories, including weight magnitude-based [120–122], activation-based [123, 124], statistics-based [68, 125], and search-based pruning [126–128]. Specifically, weight magnitude-based pruning, as one of the early pruning practices, focuses on exploring the redundant channels based on their corresponding weight magnitudes [120–122]. For example, [120] observes that the channels with smaller L_1-norm contribute less to the final network output and thus can be considered less important. Besides, [121] shows that L_2-norm can achieve more reliable pruning performance than L_1-norm. Furthermore, [122] demonstrates that the channels with smaller L_1-norm and L_2-norm are not necessarily less important. To this end, [122] further turns back to the channel correlation, which shows that the channels around the geometric median capture similar features and thus can be pruned to reduce the network redundancy with minimal accuracy loss. In parallel, inspired by the tremendous success of NAS, [126–128] leverage previous well-established NAS algorithms to automatically search for the

less important channels, which feature evolutionary algorithm-based search [127], reinforcement learning-based search [126], and gradient-based search [128].

2.2.2 Structured Layer-Wise Pruning

Layer-wise pruning removes the less important layers to explore simplified network structures. In practice, layer-wise pruning is an aggressive special case of channel-wise pruning, which removes all the channels in the same layer. And under similar compression ratios, layer-wise pruning can typically exhibit better hardware efficiency than channel-wise pruning as demonstrated in [129], which indicates that hardware-efficient networks should be shallow (i.e., with fewer layers) rather than narrow (i.e., with fewer channels). To this end, a plethora of representative layer-wise pruning works have recently flourished [70, 72, 129–132]. For example, [129] introduces several importance criteria from the lens of channel-wise pruning, such as weight magnitudes, activation maps, and batch normalization statistics. These importance criteria can be further combined to reliably navigate the less important layers. In parallel, [132] investigates the lottery ticket hypothesis (LTH) [133] from a fresh perspective of layer-wise pruning, which confirms that there also exist winning tickets at initialization in terms of layer-wise pruning.

Chapter 3

Efficient Hardware-Aware Neural Architecture Search[1]

Differentiable neural architecture search (NAS) is an emerging paradigm to automate the design of top-performing convolutional neural networks (CNNs). Despite the promising progress, previous differentiable NAS methods suffer from several crucial weaknesses, such as inaccurate gradient estimation, high memory consumption, search fairness, etc. More importantly, previous differentiable NAS works are mostly hardware-agnostic since they only search for CNNs in terms of the accuracy, which ignore other critical performance metrics, such as the on-device latency and energy. To overcome such limitations, we, in this work, introduce a novel hardware-aware differentiable NAS framework, namely SurgeNAS, in which we leverage the one-level optimization to avoid inaccurate gradient estimation. To this end, we tailor an effective identity mapping regularization to alleviate the over-selecting issue. In addition, to resolve the memory bottleneck, we propose an ordered differentiable sampling approach, which significantly reduces the search memory consumption to the single-path level and thus allows to directly search on target task instead of small proxy task and also guarantees strict search fairness. Furthermore, we introduce an efficient graph neural networks (GNNs) based predictor to approximate the on-device latency, which is then integrated into SurgeNAS to enable the latency-aware architecture search. Finally, we analyze the resource underutilization issue,

[1]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "SurgeNAS: A Comprehensive Surgery on Hardware-Aware Neural Architecture Search." IEEE Transactions on Computers (TC), 2022.

in which we propose to scale up the searched SurgeNets within *Comfort Zone* to balance the computation and memory accuracy, which leads to considerable accuracy improvement without deteriorating the on-device efficiency. Extensive experiments are conducted on ImageNet with seven hardware platforms, which clearly show the efficacy of SurgeNAS in terms of accuracy, latency, and search efficiency.

The remainder of this chapter is organized as follows. Section 3.1 introduces the research background. Section 3.2 introduces the preliminaries and discusses the motivations. Section 3.3 elaborates on the proposed SurgeNAS. Section 3.4 presents the experimental settings and results. Finally, Section 3.5 concludes this chapter.

3.1 Introduction

Convolution neural networks (CNNs) have been deemed as the fundamental engine of artificial intelligence (AI), achieving remarkable success among a wide range of real-world scenarios like object detection [19] and image classification [33, 134]. Subsequently, the significant success of CNNs attracts both academia and industry to deploy state-of-the-art CNNs on diverse hardware systems to embrace the hardware intelligence [13, 61, 135]. Nevertheless, to achieve competitive accuracy, CNNs have been evolving deeper with more layers as well as wider with more channels [19], thereby inevitably leading to the increasing computational gap between resource-limited hardware systems and computation-intensive CNNs [36].

To bridge the increasing computational gap, significant efforts have been dedicated to designing resource-efficient CNNs. Among them, one alternative is to manually design efficient lightweight CNNs, such as SqueezeNet [75], MobileNets [6, 7, 41], and ShuffleNets [42, 43]. However, the above manual process in practice requires considerable computational overheads since a significant amount of trial-and-errors are required in order to finalize the network configurations like the network depth and the network width. Apart from this, another alternative is the model compression technique, i.e., quantize or prune redundant network weights [66] such that the model complexity can be reduced while causing negligible accuracy loss. Unfortunately, model compression is only applicable to existing complicated CNNs like ResNet [19] and DenseNet [33]. As a result, model compression fails to explore the

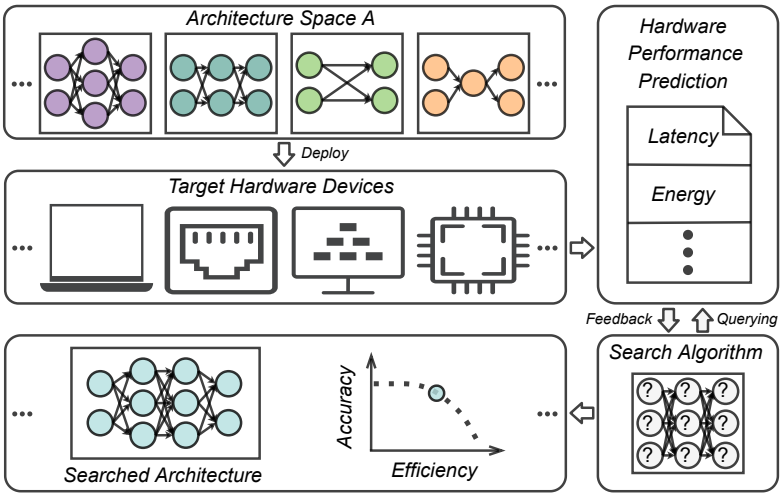

FIGURE 3.1: An overview of SurgeNAS, in which we focus on searching for top-performing architectures in terms of both accuracy and hardware efficiency.

novel CNN architectures which might bring significant performance improvement, and thus cannot provide the best architecture solution for target hardware.

To tackle the above-mentioned issues, neural architecture search (NAS) [44] has recently evolved as an emerging technique to automate the design of top-performing CNNs, which achieves impressive performance yet still suffers from prohibitive computational overheads for the search process, e.g., 2,000 GPU-days for one single search [44]. To this end, [3], for the first time, introduces the differentiable NAS paradigm, namely DARTS, which significantly reduces the search cost to several GPU-days. Afterwards, the variants of DARTS [20, 46, 47, 52, 136] are proposed to mitigate the crucial weaknesses in DARTS, e.g., the depth gap between search and evaluation [47] and large architectural eigenvalues [136]. Nonetheless, the above NAS methods only search for competitive CNNs in terms of the accuracy, regardless of other critical performance metrics like latency and energy, which are important for real-world applications [36]. Among them, [20] optimizes the searched CNNs according to the FLOPs count, but the FLOPs count cannot accurately reflect the on-device performance as seen in Figure 3.2, where we observe CNNs with the same FLOPs count could differ a lot in terms of both latency and energy.

In contrast to the above hardware-agnostic NAS works, several hardware-aware NAS methods [5, 9, 13, 61, 137] have been recently proposed, which primarily leverage the on-device latency to regularize the architecture search process. In particular, [5] adopts reinforcement learning (RL) as the search engine and integrates

the latency into the RL reward function where the latency is directly measured on target hardware, inevitably causing substantial computational overheads [13]. Besides, [9, 13, 61] take DARTS as the baseline, in which [9, 61, 137] and [13] apply the latency lookup table (LUT) and the layerwise latency predictor, respectively, to avoid the tedious measurements on target hardware. Nevertheless, similar to DARTS, the above hardware-aware differentiable NAS methods suffer from considerable memory overheads since multiple paths have to be optimized at the same time [13]. Meanwhile, the above multi-path mechanism dramatically violates the equality principle [22], i.e., the supernet is trained in a multi-path manner while the searched sub-network is re-trained in a single-path manner, making the search process biased. Moreover, the search process in [13] involves significant unfairness since only part of operator candidates get optimized at each search iteration [22]. Furthermore, similar to DARTS, all of the above-mentioned differentiable NAS approaches employ the bi-level optimization, which introduces non-negligible inaccuracy in gradient estimation [55, 56].

To address the above issues, we introduce an efficient hardware-aware differentiable NAS framework, namely SurgeNAS (see Figure 3.1), to automatically search for efficient CNNs upon target hardware. In particular, we turn back to the one-level optimization to avoid the inaccurate gradient estimation. However, given that the convergence rate of the supernet depends on the skip-connect operator heavier than other operators [136], the supernet over-selects the skip-connect operator for fast convergence, dramatically deteriorating the accuracy of the searched CNNs [52]. Please note that the above over-selecting issue also exists in the bi-level optimization where the naive early stopping heuristic [52, 136, 137] is usually integrated to avoid it. Finally, we summarize the main contributions as follows:

(1) We propose an efficient one-level differentiable NAS framework, namely SurgeNAS, in which we integrate a simple yet effective identity mapping scheme in order to avoid over-selecting the skip-connect operator.

(2) We propose an ordered differentiable sampling approach, which decreases the optimization complexity to the single-path level without degrading the search performance, while at the same time maintaining strict search fairness.

(3) We propose an efficient graph neural networks (GNNs) based predictor to accurately approximate the on-device latency with negligible computational

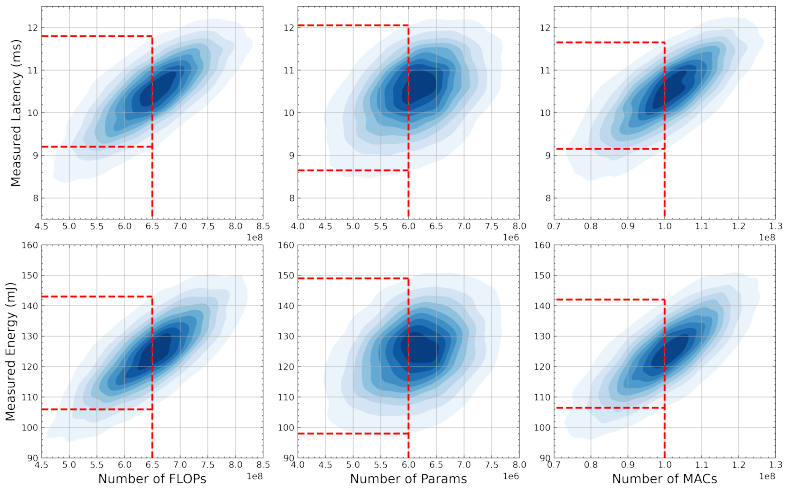

FIGURE 3.2: Relationships between the number of floating-point operations (FLOPs), parameters (Params), and memory accesses (MACs) and the runtime latency and energy on Nvidia Jetson AGX Xavier with an input batch size of 1.

overheads, which is further seamlessly integrated into SurgeNAS to achieve hardware-aware architecture search.

(4) We propose to scale up SurgeNets within *Comfort Zone* to balance the computation and memory access so as to avoid resource underutilization, which can improve the accuracy without degrading the on-device efficiency.

(5) We conduct extensive experiments to evaluate SurgeNAS on ImageNet [1] with seven hardware platforms, which explicitly demonstrate the superiority of SurgeNAS over previous state-of-the-art NAS methods. For example, SurgeNet-C1-Base obtains a top-1 accuracy of 75.5% on ImageNet, which is +2.7%/+0.3% higher than MobileNetV2/V3 [6, 7] while achieving ×1.2/×1.9 speedup on hardware devices lied in Hardware-C1. Then, after scaling up within *Comfort Zone*, SurgeNet-C1-C obtains a top-1 accuracy of 78.7%, whereas the latency is on par with SurgeNet-C1-Base on Xavier and Z8GPU.

3.2 Preliminaries and Motivations

In this section, we first introduce the preliminaries on differentiable neural architecture search (NAS) [3] and then present several assumptions and motivations.

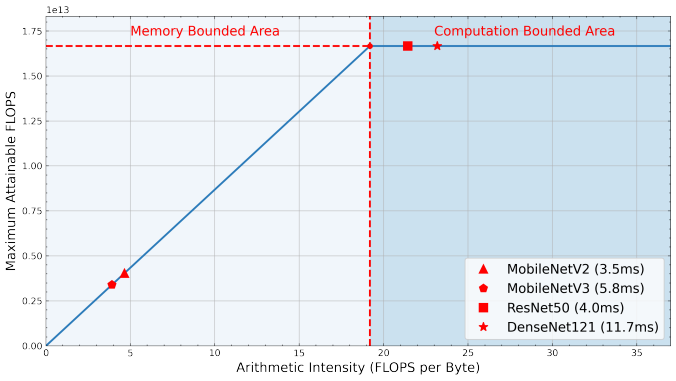

FIGURE 3.3: Illustration of the roofline analysis [8] of different CNNs on Z8GPU[2], where FLOPS refers to the number of processed FLOPs per second.

3.2.1 Preliminaries on Differentiable NAS

We begin with the operator space $\mathcal{O}$, where each element $\mathcal{O}_k$ denotes an operator candidate. Following the weight-sharing NAS paradigm (also known as one-shot NAS) [45], an over-parameterized network named supernet is constructed, in which each searchable layer contains $|\mathcal{O}|$ operators. Then, to relax the discrete search space to be continuous, operators in each searchable layer are assigned with a set of architecture parameters $\alpha \in \mathbb{R}^{L\times|\mathcal{O}|}$, where L is the number of searchable layers. As such, we are able to formulate the output of the supernet as follows:

$$\mathcal{Y}(\mathcal{X}) = \sum_{l=1}^{L}\sum_{k=1}^{|\mathcal{O}|}\frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})}\cdot\mathcal{O}_k(\mathcal{X}_l) \tag{3.1}$$

where $\mathcal{X}_l$ is the input of the l-th layer and $\mathcal{X}$ is the initial input. Besides, $\mathcal{O}_k$ is the k-th operator in $\mathcal{O}$. Apart from these terms, $\alpha_{l,k}$ represents the architecture parameter assigned to the k-th operator candidate in the l-th searchable layer of the supernet. Subsequently, in order to find the optimal architecture, an effective bi-level optimization method is introduced to jointly optimize the network weights w and the architecture parameters α, or more specifically, a training phase to optimize w and a validation phase to optimize α:

$$\underset{\alpha}{\text{minimize}}\ \mathcal{L}_{valid}(w^*(\alpha),\alpha)\ s.t.,\ w^*(\alpha) = \underset{w}{\text{minimize}}\ \mathcal{L}_{train}(w,\alpha) \tag{3.2}$$

where $\mathcal{L}_{train}(\cdot)$ and $\mathcal{L}_{valid}(\cdot)$ represent the loss functions on the training and validation datasets, respectively. In particular, we observe that Eq (3.1) and Eq (3.2)

[2]It is worth noting that we here use hardware abbreviations for the sake of simplicity and the detailed hardware specifications can be found in Section 3.4.

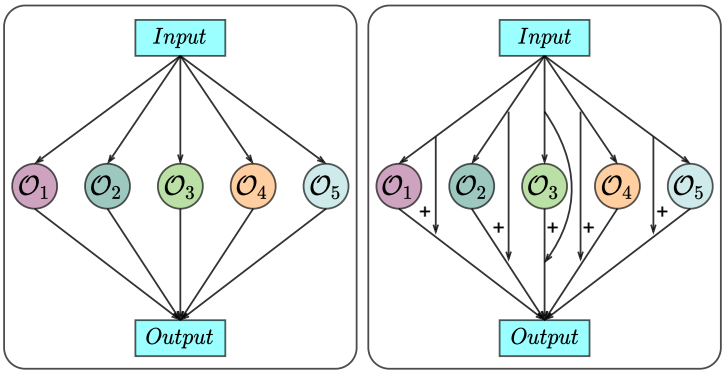

FIGURE 3.4: Comparisons between the conventional supernet structure [3, 9] (*left*) and the proposed supernet structure that features identity mapping (*right*).

only involve continuous transformations with respect to w and α [3]. As such, both w and α can be optimized using stochastic gradient descent (SGD) as follows:

$$\alpha^* = \alpha - \eta_\alpha \cdot \frac{\partial \mathcal{L}_{valid}(w^*(\alpha), \alpha)}{\partial \alpha} \; s.t., \; w^*(\alpha) = w - \eta_w \cdot \frac{\partial \mathcal{L}_{train}(w, \alpha)}{\partial w} \tag{3.3}$$

where η_α and η_w correspond to the learning rates of α and w, respectively. Once the above optimization process terminates, we are able to determine the searched architecture by reserving the strongest operator for each searchable layer while removing other operators, where the operator strength is defined as $\exp(\alpha_{l,k}) / \sum_{k'=1}^{|\mathcal{O}|} \exp(\alpha_{l,k'})$. Despite achieving promising results, the above bi-level optimization in practice introduces non-negligible computation burdens, and more importantly, considerable inaccuracy in gradient estimation that dramatically deteriorates the search process as seen in [55, 56]. Apart from this, the above differentiable NAS paradigm suffers from the search memory bottleneck [13, 46], where the memory consumption linearly increases with respect to the operator space size $|\mathcal{O}|$ as indicated by the summation $\sum_{k=1}^{|\mathcal{O}|}$ in Eq (3.1). Due to the above memory bottleneck, [3, 3, 20, 47] can only search on small proxy tasks like CIFAR10, and thus cannot provide the optimal architecture solutions upon target tasks like ImageNet. Note that the image size in CIFAR10 is 32×32, whereas the image size in ImageNet is 224×224. For more technical details about differentiable NAS, you may refer to DARTS [3].

3.2.2 Assumptions and Motivations

In literature, existing approaches that are related to designing efficient CNNs such as [3, 6, 7, 20, 134] highly reply on the following three assumptions.

Assumption 1: The theoretic complexity metrics like FLOPs or Params are effective to reflect the on-device performance. In fact, [20, 134] focus on designing efficient CNNs with less FLOPs. Nevertheless, the FLOPs count is an indirect proxy metric that cannot directly reflect the on-device performance like latency and energy as illustrated in Figure 3.2, where we observe that CNNs with the same FLOPs/Params/MACs count could demonstrate a big difference in terms of both latency and energy on target hardware.

Assumption 2: The optimal neural architecture searched on the small proxy task is also optimal upon the target task. Due to the search memory bottleneck, DARTS [3] and its variants [20, 47, 52, 136] first conduct the architecture search on the small proxy task (i.e., CIFAR10) and then transfer the searched architecture to the target task (i.e., ImageNet). However, as seen in [13], the above search pattern only generates sub-optimal architectures upon target tasks. For example, in [46], the architecture searched on CIFAR10 obtains a top-1 accuracy of 74.9% on ImageNet, whereas the one directly searched on ImageNet achieves a higher accuracy of 75.8%.

Assumption 3: Lightweight neural networks are always better than non-lightweight ones in terms of the runtime efficiency. Lightweight CNNs like MobileNets [6, 7, 41] have been widely deployed in real-world AI scenarios with rigid latency requirements [36]. However, lightweight CNNs like MobileNetV2/V3 cannot guarantee to run faster than those non-lightweight ones like ResNet [19] as shown in Figure 3.3, where we observe MobileNetV2/V3 suffer from significant resource underutilization (i.e., low attainable FLOPS), which is caused by the low arithmetic intensity [8].

Therefore, the above assumptions are not always valid. To this end, in SurgeNAS, we take direct performance metrics like latency as guidelines to search for efficient CNNs, which is conducted directly upon target tasks like ImageNet. Moreover, we scale up the searched CNNs within *Comfort Zone* to increase the arithmetic intensity so as to avoid resource underutilization, which is able to improve the accuracy without deteriorating the runtime efficiency on target hardware.

3.3 Methodology

In this section, we introduce the proposed SurgeNAS (see Figure 3.1). We begin with the problem formulation, in which we turn back to the one-level optimization

TABLE 3.1: Illustration of the supernet structure under the MobileNetV2-based architecture search space. Specifically, TBS indicates the block type to be searched and Conv means the convolutional layer. Besides, AvgPool and FC are the average pooling layer and the fully-connected layer, respectively.

Input shape	Block	Filters	Repeat	Stride
$224 \times 224 \times 3$	3×3 Conv	40	1	2
$112 \times 112 \times 40$	MBK3E1	24	1	1
$112 \times 112 \times 24$	TBS	32	4	2
$56 \times 56 \times 32$	TBS	56	4	2
$28 \times 28 \times 56$	TBS	112	4	2
$14 \times 14 \times 112$	TBS	128	4	1
$14 \times 14 \times 128$	TBS	256	4	2
$7 \times 7 \times 256$	TBS	432	1	1
$7 \times 7 \times 432$	1×1 Conv	1728	1	1
$7 \times 7 \times 1728$	7×7 AvgPool	-	1	1
1728	FC	1000	1	1

to avoid the inaccuracy in gradient estimation [55, 56]. Meanwhile, we propose an effective identity mapping scheme, which is integrated into SurgeNAS to alleviate the over-selecting issue in terms of the skip-connect operator. Next, we introduce the search space of SurgeNAS, which is built upon MobileNetV2 [6]. Then, we elaborate on the proposed ordered differentiable sampling approach, which brings considerable search efficiency when compared with existing differentiable NAS approaches [9, 13, 61, 137] while guaranteeing the strict search fairness [22]. We emphasize that the above benefits are of great significance for efficient and stable architecture search that existing differentiable NAS approaches cannot provide. Furthermore, to enable the latency-aware architecture search, we propose an efficient graph neural networks (GNNs) based latency predictor, which is incorporated into SurgeNAS to avoid the tedious on-device measurements. Finally, to alleviate the resource underutilization issue, we propose to scale up the searched SurgeNets, followed by insight theoretical analysis from the roofline model's perspective [8]. Please note that the above scaling-up strategy can greatly improve the accuracy of SurgeNets while not increasing the latency on target hardware.

3.3.1 Problem Formulation

Recall that previous differentiable NAS approaches [3, 9, 13, 20, 46, 47, 61] explicitly leverage the bi-level optimization scheme, in which w and α are optimized

in two separate phases as illustrated in Eq (3.2), or more specifically, α is frozen when optimizing w and w is frozen when optimizing α. Nevertheless, the bi-level optimization in practice introduces considerable inaccuracy in gradient estimation as demonstrated in [55, 56], thereby inevitably leading to sub-optimal architecture solutions. Therefore, to bridge the above optimization gap, we focus on the one-level optimization, in which w and α are simultaneously optimized as follows:

$$\underset{w,\alpha}{\text{minimize}} \ \ \mathcal{L}_{train}(w, \alpha) \tag{3.4}$$

where w and α will be simultaneously updated using the stochastic gradient descent (SGD) scheme as follows:

$$\alpha^* = \alpha - \eta_\alpha \cdot \frac{\partial \mathcal{L}_{train}(w, \alpha)}{\partial \alpha}, \ \ w^* = w - \eta_w \cdot \frac{\partial \mathcal{L}_{train}(w, \alpha)}{\partial w} \tag{3.5}$$

where η_w and η_α correspond to the learning rates of w and α. Unfortunately, directly employing the above one-level optimization leads to the over-selecting of the skip-connect operator during the search process, which significantly degrades the accuracy of searched CNNs [52, 136]. According to [136], the over-selecting phenomenon comes from the fact that the convergence rate of the supernet is more dependent on the skip-connect operator than other operators, and thus the supernet prefers to include more skip-connect operators for fast convergence. To this end, unlike [55, 56], we propose an effective identity mapping regularization method (see Figure 3.4)[3], i.e., we equally assign one identity mapping operator to each operator candidate in $\mathcal{O}$. Therefore, we can re-formulate Eq (3.1) as follows:

$$\mathcal{Y}(\mathcal{X}) = \sum\nolimits_{l=1}^{L} \sum\nolimits_{k=1}^{|\mathcal{O}|} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|} \exp(\alpha_{l,k'})} \cdot (\mathcal{O}_k(\mathcal{X}_l) + \varphi \cdot \mathcal{X}_l) \tag{3.6}$$

where φ is the weight assigned to the identity mapping operator, which is initialized as 2 and then linearly decays to zero at the end of search. Thus, the above regularization effect will be removed at the end of search because $\varphi = 0$. In practice, the identity mapping operator accelerates the convergence of each operator [19]. As a result, the convergence difference among different operators is alleviated, and thus the supernet concentrates on selecting the operator with the highest contribution to the accuracy rather than the convergence. More importantly, the proposed identity mapping scheme does not undermine the search fairness since it is equally imposed across all the operators [22]. Meanwhile, we also provide a theoretical

[3]We try both options [55, 56] in SurgeNAS, but neither works well.

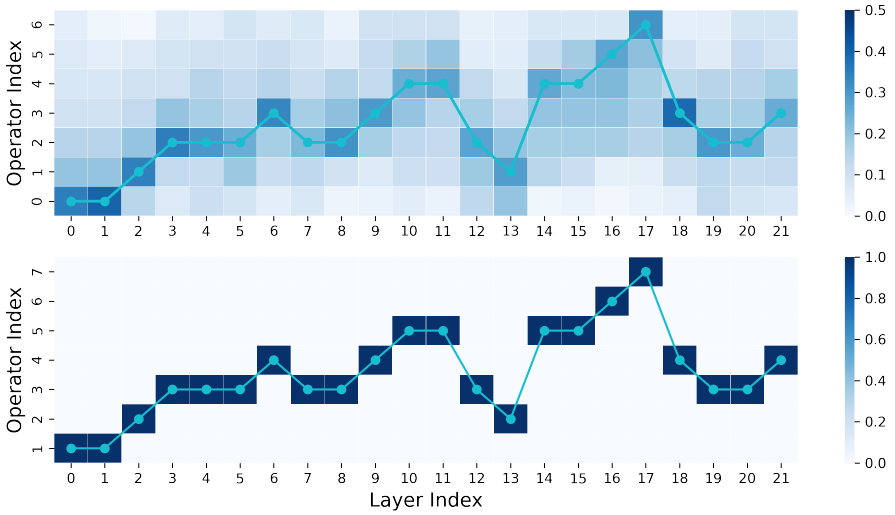

FIGURE 3.5: Illustration of Gumbel-Softmax reparameterization [10] under a large temperature (*top*) and a small temperature (*bottom*).

analysis to show the effectiveness of the proposed identity mapping scheme (see Section 1 in supplementary materials).

3.3.2 Architecture Search Space Design

In literature, differentiable NAS approaches like DARTS [3] and its variants [20, 21, 46, 47, 52] advocate for the cell-level architecture search, i.e., once a cell structure is determined, it will be repeatedly stacked across all the layers to construct the stand-alone network. However, the same cell could demonstrate a big difference in terms of both accuracy [134] and efficiency [13] when being at different layers. Apart from this, as demonstrated in [9], enabling the layerwise diversity is able to strike a better trade-off between accuracy and efficiency. To this end, in SurgeNAS, we follow the heuristic of the layerwise architecture space design as widely adopted in recent hardware-aware NAS frameworks [5, 9, 13, 61], where each searchable layer is able to choose one operator from the operator space $\mathcal{O}$. In particular, the operator space $\mathcal{O}$ of SurgeNAS is mainly inspired by the inverted bottleneck convolution (MBConv) of MobileNetV2 [6], in which we allow a set of MBConv layers with kernel sizes of $\{3, 5, 7\}$ and expansion ratios of $\{3, 6\}$ [13, 61, 74]. Apart from these, to achieve flexible architecture search in terms of the network depth, we include the skip-connect operator, which is computation-free [3, 9, 13]. Here, the channel layout used in SurgeNAS is similar to [9, 13], which is summarized in

Table 3.1. Therefore, in SurgeNAS, we derive that $|\mathcal{O}| = 7$, and given that the supernet consists of 22 layers where the first one is fixed [13, 74], the architecture space size of SurgeNAS is calculated as $|\mathcal{A}| = 7^{21} \approx 5.6 \times 10^{17}$.

3.3.3 Ordered Differentiable Sampling

In this section, we begin with the stand-alone architecture candidate defined as $arch = \{op_l\}_{l=1}^{L}$. Therefore, during the forward propagation of the supernet, we are able to calculate the probability of $arch$ being selected as follows:

$$\mathcal{P}(arch) = \prod_{l=1}^{L} \mathcal{P}(op_l = \mathcal{O}_k), \ s.t., \ \mathcal{O}_k \in \mathcal{O} \tag{3.7}$$

where $\mathcal{O}_k$ denotes the k-th operator candidate in $\mathcal{O}$ and $\mathcal{P}(op_l = \mathcal{O}_k)$ represents the probability of $\mathcal{O}_k$ being selected in the l-th layer, which is determined by α:

$$\mathcal{P}(op_l = \mathcal{O}_k) = \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|} \exp(\alpha_{l,k'})} \tag{3.8}$$

For the sake of simplicity, we replace $\mathcal{P}(op_l = \mathcal{O}_k)$ with $\mathcal{P}_{l,k}$. Nevertheless, given that the search space $\mathcal{A}$ is discrete, traversing all the possible architectures inevitably leads to non-trivial computation overheads since the search space is prohibitively large [13] (e.g., $|\mathcal{A}| \approx 5.6 \times 10^{17}$ in SurgeNAS). To alleviate this issue, we follow the weight-sharing NAS heuristic [45] and take the Gumbel-Softmax reparameterization [10] to relax the discrete space to be continuous:

$$\widehat{\mathcal{P}}_{l,k} = \frac{\exp((\log \mathcal{P}_{l,k} + \mathcal{G}_{l,k})/T)}{\sum_{k'=1}^{|\mathcal{O}|} \exp((\log \mathcal{P}_{l,k'} + \mathcal{G}_{l,k'})/T)} \tag{3.9}$$

where $\mathcal{G} \in \mathbb{R}^{L \times |\mathcal{O}|}$ is the random variable sampled from the $Gumbel(0, 1)$ distribution [10]. T denotes the softmax temperature, which is initialized as 1 and then gradually decayed to be close to 0 in SurgeNAS. The intuition behind this is simply that a larger T pushes the relaxation process to become more smooth (see Figure 3.5 (*top*)), and thus allows more architecture probabilities (so-called the architecture search warm-up phase), whereas a smaller T makes the relaxation become more deterministic (see Figure 3.5 (*bottom*)). Besides, it is worth noting that $\lim_{T \to 0} \widehat{\mathcal{P}}_{l,k} = \mathcal{P}_{l,k}$ as proven in [138], which implies that the above relaxation is unbiased once converged [20]. In this way, we are able to re-formulate the output of the supernet $\mathcal{Y}(\mathcal{X})$ in Eq (3.6) as follows:

$$\mathcal{Y}(\mathcal{X}) = \sum_{l=1}^{L} \sum_{k=1}^{|\mathcal{O}|} \left(\overline{\mathcal{P}}_{l,k} \cdot \mathcal{O}_k(\mathcal{X}_l) + \varphi \cdot \mathcal{X}_l\right) \tag{3.10}$$

Algorithm 1 Ordered Differentiable Sampling Strategy

Input: A set of architecture parameters α, softmax temperature T, the predefined operator space $\mathcal{O}$

Output: A set of ordered differentiale parameters $\{\widehat{\mathcal{P}}^i\}_{i=1}^{|\mathcal{O}|}$

1: Calculate the initial probability distribution $\mathcal{P}^1$ ▷ see Eq (3.8)
2: **for** $i = 1$ **to** $|\mathcal{O}|$ **do**
3: Calculate $\widehat{\mathcal{P}}^i$ in terms of $\mathcal{P}^i$ and T ▷ see Eq (3.9)
4: $\overline{\mathcal{P}}^i \leftarrow \text{binarize}(\widehat{\mathcal{P}}^i)$ ▷ see Eq (3.11)
5: $\mathcal{P}^{i+1} \leftarrow \left[\prod_{i'=1}^{i}(1 - \overline{\mathcal{P}}^{i'})\right] \cdot \mathcal{P}^i$
6: Re-normalize the probability distribution $\mathcal{P}^{i+1}$
7: **end for**
8: Return the ordered differentiable parameters $\{\widehat{\mathcal{P}}^i\}_{i=1}^{|\mathcal{O}|}$

where $\overline{\mathcal{P}}$ corresponds to the binarized representation of $\widehat{\mathcal{P}}$. In practice, $\overline{\mathcal{P}}$ is mathematically calculated as follows:

$$\overline{\mathcal{P}}_{l,k} = \begin{cases} 1, & \text{if } \widehat{\mathcal{P}}_{l,k} = \max(\widehat{\mathcal{P}}_l) \\ 0, & \text{otherwise} \end{cases} \tag{3.11}$$

where $\widehat{\mathcal{P}}$ is calculated with respect to Eq (3.9).

As a result, during the architecture search process, only one single-path sub-network needs to be activated at each search iteration as indicated by $\overline{\mathcal{P}} \in \{0, 1\}$ because the output $\mathcal{Y}(\mathcal{X})$ only depends on the operators with $\overline{\mathcal{P}}_{l,k} = 1$. In practice, the above proposed strategy brings two main benefits when compared with [9, 13, 20, 61, 137]. On the one hand, the above single-path mechanism leads to considerable efficiency in terms of the memory consumption. Therefore, given that the hardware memory size is fixed, the memory efficiency allows us to take a larger batch size, which greatly accelerates the search process. Meanwhile, the above single-path mechanism guarantees the equality principle [22], in which the supernet is trained in the same way as the searched architecture, i.e., both are trained in the single-path manner. On the other hand, owing to the memory efficiency, we are able to conduct architecture search directly on target tasks (e.g., ImageNet) instead of small proxy tasks [3] (e.g., CIFAR10)[4]. It is worth noting that the above proposed single-path mechanism is different from [22, 48] since they are non-differentiable where the single-path sub-network is randomly sampled from the supernet.

[4]The image size is 224×224 in ImageNet and 32×32 in CIFAR10.

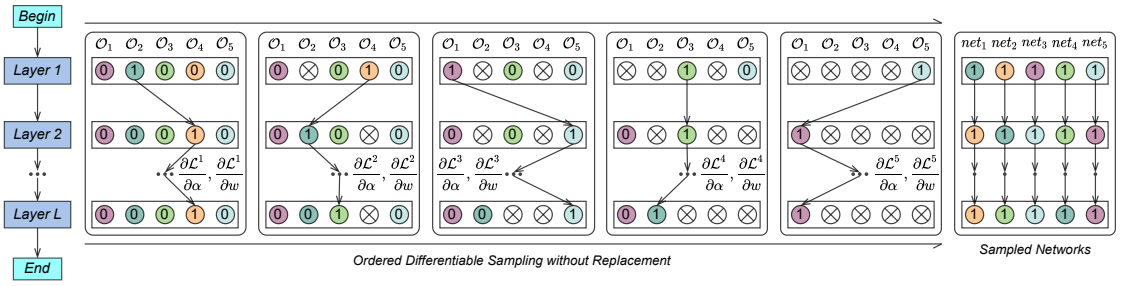

FIGURE 3.6: Illustration of the proposed ordered differentiable sampling approach. Here, for simplicity, we assume the operator space consists of 5 operator candidates (i.e., $|\mathcal{O}| = 5$). To this end, we need to orderly sample 5 single-path sub-networks without replacement during each iteration. Specifically, if one operator candidate is sampled by previous single-path sub-networks, it will be excluded from the subsequent ones. Besides, $\{\frac{\partial \mathcal{L}^i}{\partial \alpha}\}_{i=1}^{|\mathcal{O}|}$ and $\{\frac{\partial \mathcal{L}^i}{\partial w}\}_{i=1}^{|\mathcal{O}|}$ denote the gradients of α and w, respectively. The sampled 5 single-path sub-networks are then visualized in the rightmost sub-figure.

Next, we generalize the reparameterization method in Eq (3.9) and (3.10) to achieve strict search fairness [22] without sacrificing the differentiability. To this end, during each search iteration, we orderly sample $|\mathcal{O}|$ single-path sub-networks without replacement, or in other words, if one operator is sampled by previous single-path sub-networks, it will not appear in the subsequent ones. We denote the above-sampled $|\mathcal{O}|$ single-path sub-networks as $\{\widehat{\mathcal{P}}^i\}_{i=1}^{|\mathcal{O}|}$. In practice, after sampling one sub-network, we re-normalize the probability distribution $\mathcal{P}$ so as to exclude the operators sampled by the previous sub-networks, and then use the re-normalized probability distribution to sample the subsequent ones. The above sampling process is visualized in Figure 3.6. To be more specific, we take the i-th sampled single-path sub-network as an example, i.e., we first encode the i-th sampled single-path sub-network with a sparse matrix $\overline{\mathcal{P}}^i \in \{0,1\}^{L\times|\mathcal{O}|}$ (see Eq (3.11)), in which 1 means that the corresponding operator is sampled whereas 0 means not (see Figure 3.6). As such, we are able to effectively exclude the previously sampled single-path sub-networks using $\prod_{i'=1}^{i}(1-\overline{\mathcal{P}}^{i'})$, in which we multiply the initial possibility distribution $\mathcal{P}$ with $\prod_{i'=1}^{i}(1-\overline{\mathcal{P}}^{i'})$ and substitute the newly obtained possibility distribution into Eq (3.9) to sample the next single-path sub-network. In this way, all the operators in the supernet will get sampled and optimized only once during each search iteration, which satisfies the strict search fairness [22]. We provide a detailed demonstration of the above sampling process in Algorithm 1. Next, we substitute $\{\widehat{\mathcal{P}}^i\}_{i=1}^{|\mathcal{O}|}$ into Eq (3.10) and compute the output of each

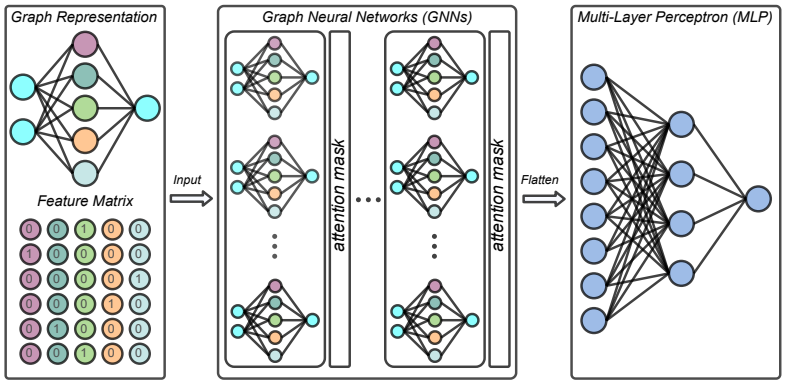

FIGURE 3.7: Illustration of the proposed GNN-based predictor.

sub-network, which will be used to calculate the training loss:

$$\mathcal{L}_{train}(w, \alpha) = \sum_{i=1}^{|\mathcal{O}|} \mathcal{L}_{train}^{i}(w, \widehat{\mathcal{P}}^{i}) \tag{3.12}$$

To summarize, the proposed ordered differentiable sampling approach brings twofold benefits in practice. On the one hand, it is able to reduce the optimization complexity during the search process from $O(|\mathcal{O}|^2)$ [3, 9, 20, 47, 61, 137] to $O(|\mathcal{O}|)$. Here, the optimization complexity is calculated in terms of both inference time and memory consumption. Meanwhile, due to the single-path mechanism, the network weights are not coupled, which alleviates the optimization gap between search and evaluation [22, 48]. On the other hand, it guarantees the strict search fairness [22] in a differentiable manner since all the operators get equally optimized during each search iteration (see Figure 3.6), which stabilizes the search process and brings considerable performance improvement [22]. These benefits are of great importance in order to achieve efficient, direct, fair, and stable architecture search that existing differentiable hardware-aware NAS methods [9, 13, 61, 137] cannot provide.

Remarks. Similar to the proposed SurgeNAS, RelativeNAS [139] also introduces an efficient differentiable sampling strategy to enable differentiable architecture search. Specifically, during each search iteration, RelativeNAS first samples a set of random architecture candidates and then divides them into two splits according to their loss values. In addition, RelativeNAS specifies those with smaller loss values as fast learners and those with larger loss values as slow learners. During each search iteration, RelativeNAS iteratively generates pseudogradients from those slow learners and updates the architecture parameters using the generated pseudogradients. Finally, all the slow and fast learners are re-merged for subsequent

differentiable optimization. However, the generated pseudogradients may be inaccurate since the generated pseudogradients only consider those slow learners while ignoring those fast learners. This inevitably suffers from significant search bias and thus may mislead the search process towards sub-optimal network solutions. In contrast to RelativeNAS, the search process of SurgeNAS is fully differentiable, which allows us to directly optimize the architecture parameters using standard gradient descent. In parallel, we can also consider the search space to be sampled or population-based in non-differentiable manners. However, this may introduce considerable search overheads and also suffer from sub-optimal search results due to the coupled weights among different architecture candidates as discussed in [48]. More recently, RADARS [140] also introduces an efficient differentiable search algorithm to mitigate the search memory bottleneck. To this end, RADARS leverages reinforcement learning techniques to gradually remove the less important operator candidates from the supernet. However, despite its memory efficiency, RADARS still suffers from considerable memory consumption, especially in the early search process. In contrast to RADARS, the proposed SurgeNAS can maintain strong search memory efficiency throughout the entire search process.

3.3.4 Efficient Latency Prediction

However, given that the architecture search space of NAS is prohibitively large, directly measuring the runtime latency on target hardware for $arch \in \mathcal{A}$ inevitably leads to non-trivial computation overheads as seen in [5]. Meanwhile, during the search process, measuring the latency across different hardware platforms is quite inconvenient, which significantly slows down the search process. To this end, we propose an efficient graph neural networks (GNNs) [141] based latency predictor (see Figure 3.7), which is motivated by the fact that GNNs are capable to learn discriminative representations for graph-structured data [141] like the neural architecture in the NAS scenario [11, 12].

To begin with, we represent $arch$ with an adjacency matrix $M \in \mathbb{R}^{L\times L}$ and a feature matrix $\overline{\alpha} \in \mathbb{R}^{L\times|\mathcal{O}|}$, in which M and $\overline{\alpha}$ jointly determine $arch$, or more specifically, M indicates the global node connectivity of $arch$ and $\overline{\alpha}$ suggests which operator is selected in each node. Here, $\overline{\alpha}$ is the binarized representation of α, and thus only consists of $\{1, 0\}$, where 1 indicates the corresponding operator is

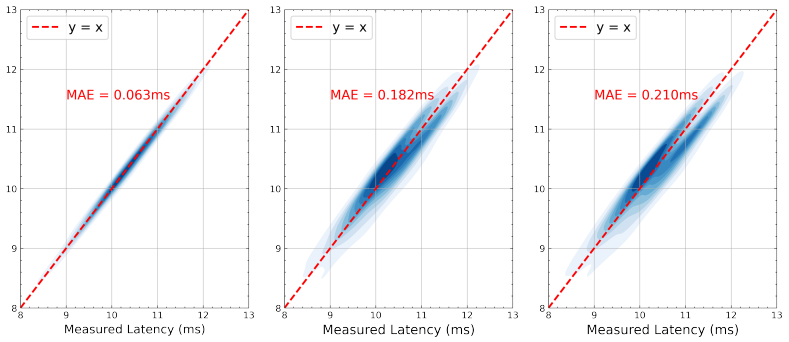

FIGURE 3.8: Relationships between the predicted latency and the measured latency on Xavier. Specifically, the *left*, *middle*, and *right* sub-figures correspond to the proposed GNN-based latency predictor, [11], and BRP-NAS [12].

selected while 0 means not. In this way, we are able to formulate the GNN layer [141] as follows:

$$GNN(M, \overline{\alpha}) = F(M\overline{\alpha}W_g), \ s.t., \ arch \sim (M, \overline{\alpha}) \tag{3.13}$$

where $F(\cdot)$ is the activation function and W_g is the learnable weight in the GNN layer. However, the original GNNs [141] assume graphs are undirected, whereas graphs are directed in the NAS scenario [3]. Therefore, directly employing the original GNNs in the NAS scenario inevitably leads to degraded prediction performance. To alleviate this issue, we closely follow [11] and take the average of two GNN layers, or more specifically, we use one GNN layer M to propagate the information in the forward propagation and one GNN layer M^T to reverse the propagation direction:

$$GNN(M, \overline{\alpha}) = \frac{1}{2}\left[F(M\overline{\alpha}W_g^+) + F(M^T\overline{\alpha}W_g^-)\right] \tag{3.14}$$

where W_g^+ and W_g^- represent the learnable weights in the above two GNN layers. Unfortunately, the above formulation does not perform well since it treats all the operators in the same way while the inference latency highly depends on the selected operators. To this end, we introduce the following attention mechanism:

$$GNN(M, \overline{\alpha}) = \frac{1}{2}\left[H_g^+ F(M\overline{\alpha}W_g^+) + H_g^- F(M\overline{\alpha}W_g^-)\right] \tag{3.15}$$

where H_g^+ and H_g^- are the attention masks. Besides, for the regression purpose, we include the multi-layer perceptron (MLP) after the GNN layers as seen in Figure 3.7, and then we can derive the latency of *arch* as $MLP(GNN(M, \overline{\alpha}))$, which is further denoted as $LAT(\overline{\alpha})$ for simplicity.

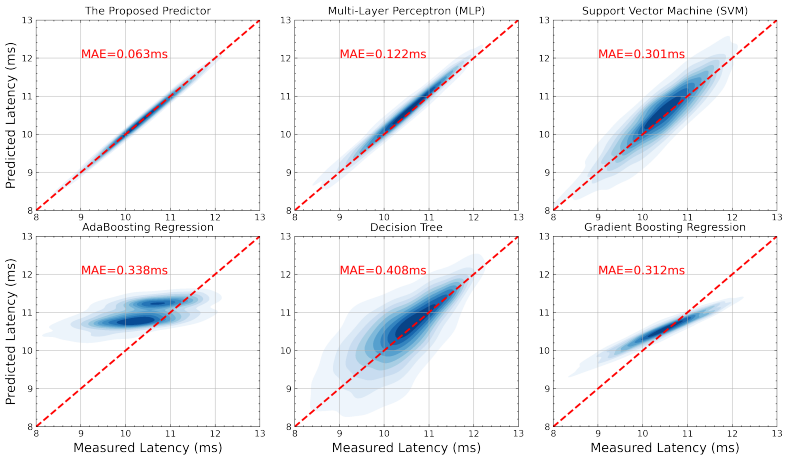

FIGURE 3.9: Comparisons of the proposed GNN-based latency predictor with different regression models where we leverage Xavier as target hardware.

In practice, we construct the GNN-based latency predictor using three GNN layers with 128 neurons in each layer. For the MLP part, we employ three fully-connected layers with 64, 64, 1 neurons, respectively. Besides, ReLU is applied as the default activation function. For data preparations, we collect 10,000 architecture candidates in the architecture space $\mathcal{A}$, which follows the sampling strategy in Figure 3.6. Then, we split the collected architectures and the corresponding latency measurements into two parts, i.e., 9,000 as the training data and 1,000 as the validation data. Afterwards, we train the proposed latency predictor with the Adam optimizer, where we use a batch size of 8 and an initial learning rate of 0.001 (annealed down to zero following the cosine schedule). The above training process takes less than five minutes on Z4GPU, which is negligible compared to the prohibitive search cost. We note that the proposed GNN-based latency predictor is only applied during the search process, whereas the latency results in Table 3.2 are directly measured on target hardware for fair comparisons.

To show the effectiveness of the proposed GNN-based latency predictor, we conduct extensive experiments on seven hardware platforms. Due to space constraints, we take the experimental results on Nvidia Jetson Xavier as an example. The experimental results and comparisons with [11, 12] are illustrated in Figure 3.8. In particular, the proposed GNN-based latency predictor equipped with the proposed attention mechanism achieves an extremely low mean-absolute-error (MAE) of 0.063ms as illustrated in Figure 3.8 (*left*), which significantly outperforms the relevant counterparts [11, 12]. Besides, we compare the prediction performance

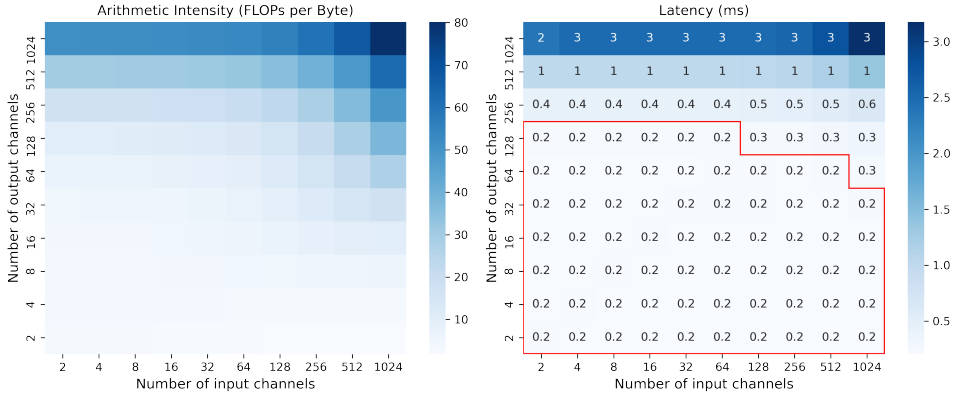

FIGURE 3.10: Illustration of the relationships between the arithmetic intensity (*left*) / the runtime latency measured on Z8GPU (*right*) and the number of input/output channels in the convolutional layer. In particular, we define the area bounded with red lines as *Comfort Zone*.

with other regression methods such as multi-layer perceptron (MLP) and support vector machine (SVM). The experimental results are illustrated in Figure 3.9, which clearly shows the GNN-based predictor is able to capture the latency more accurately. More importantly, once the latency predictor is well trained, it is able to approximate the on-device latency for $arch \in \mathcal{A}$ with only one-time inference, which takes less than one millisecond, i.e., accurate yet still efficient. Apart from this, the proposed GNN-based latency predictor only consists of continuous transformations in terms of the architecture parameters α as seen in Eq (3.15), which in turn enables to calculate the gradient of α during the backward propagation of the supernet whereas the direct on-device measurement achieves neither. Later, we integrate the proposed latency predictor into SurgeNAS to accomplish latency-aware architecture search.

3.3.5 Efficient Hardware-Aware Architecture Search

Nevertheless, the optimization objective defined in Eq (3.12) only optimizes the search process in terms of the accuracy while leaving other critical performance metrics like the runtime latency unconsidered. As a result, it derives CNNs with competitive accuracy, but, unfortunately, such competitive accuracy comes at the cost of extremely high computational complexity as shown in Table 3.2 (see SurgeNet-Base), thereby greatly hindering the real-world deployments especially on resource-constrained hardware platforms like CPUs and edge devices [36]. To this end, we

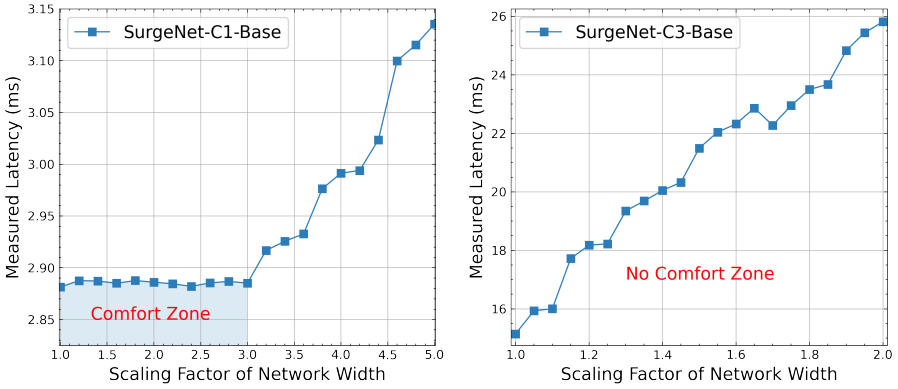

FIGURE 3.11: Illustration of *Comfort Zone* on Z8GPU (*left*) and Z8CPU (*right*). Unlike GPUs with high parallelism, CPUs are of low parallelism and thus do not have *Comfort Zone*.

further incorporate the proposed GNN-based latency predictor (see Section 3.3.4) into SurgeNAS to accomplish the latency-aware architecture search. Therefore, we are able to re-formulate the optimization objective in Eq (3.12) as follows:

$$\mathcal{L}_{train}(w,\alpha) = \sum\nolimits_{i=1}^{|\mathcal{O}|} \left[\mathcal{L}^{i}_{train}(w,\widehat{\mathcal{P}}^{i}) + \beta \cdot LAT(\overline{\mathcal{P}}^{i})\right] \tag{3.16}$$

where β is the coefficient to control the trade-off magnitude between accuracy and latency. In particular, we can obtain the architecture solution of low latency by imposing a larger β or of high accuracy with a smaller β (see Figure 3.15), which in turn brings considerable search flexibility. Besides, $\mathcal{L}^{i}_{train}(w,\widehat{\mathcal{P}}^{i})$ and $LAT(\overline{\mathcal{P}}^{i})$ correspond to the training loss and the predicted latency of the i-th sampled sub-network. Here, $\overline{\mathcal{P}}$ is equivalent to $\overline{\alpha}$ since they both denote the binarized representation of the architecture candidate, in which 1 indicates the operator is selected while 0 means not (see Figure 3.6). In this way, we can directly feed $\overline{\mathcal{P}}^{i}$ into the well-trained latency predictor to approximate the latency of the i-th sub-network as seen in Eq (3.16). Considering that $\mathcal{L}(w,\alpha)$ is apparently differentiable with respect to the network weights w [3], we only provide the differentiability analysis with respect to the architecture parameters α:

$$\begin{aligned}\frac{\partial \mathcal{L}_{train}(w,\alpha)}{\partial \alpha} = &\sum\nolimits_{i=1}^{|\mathcal{O}|} \frac{\partial L_{train}(w,\widehat{\mathcal{P}}^{i})}{\partial \overline{\mathcal{P}}^{i}} \cdot \frac{\partial \overline{\mathcal{P}}^{i}}{\partial \widehat{\mathcal{P}}^{i}} \cdot \frac{\partial \widehat{\mathcal{P}}^{i}}{\partial \mathcal{P}^{i}} \cdot \frac{\partial \mathcal{P}^{i}}{\partial \mathcal{P}^{1}} \cdot \frac{\partial \mathcal{P}^{1}}{\partial \alpha} \\ &+ \beta \cdot \sum\nolimits_{i=1}^{|\mathcal{O}|} \frac{\partial LAT(\overline{\mathcal{P}}^{i})}{\partial \overline{\mathcal{P}}^{i}} \cdot \frac{\partial \overline{\mathcal{P}}^{i}}{\partial \widehat{\mathcal{P}}^{i}} \cdot \frac{\partial \widehat{\mathcal{P}}^{i}}{\partial \mathcal{P}^{i}} \cdot \frac{\partial \mathcal{P}^{i}}{\partial \mathcal{P}^{1}} \cdot \frac{\partial \mathcal{P}^{1}}{\partial \alpha}\end{aligned} \tag{3.17}$$

where $\frac{\partial \overline{\mathcal{P}}^{i}}{\partial \widehat{\mathcal{P}}^{i}} = 1$ has already been proven in [142] since $\overline{\mathcal{P}}^{i}$ is the binarization of $\widehat{\mathcal{P}}^{i}$. Besides, $\frac{\partial LAT(\overline{\mathcal{P}}^{i})}{\partial \overline{\mathcal{P}}^{i}}$ is determined by the network weights of the GNN-based

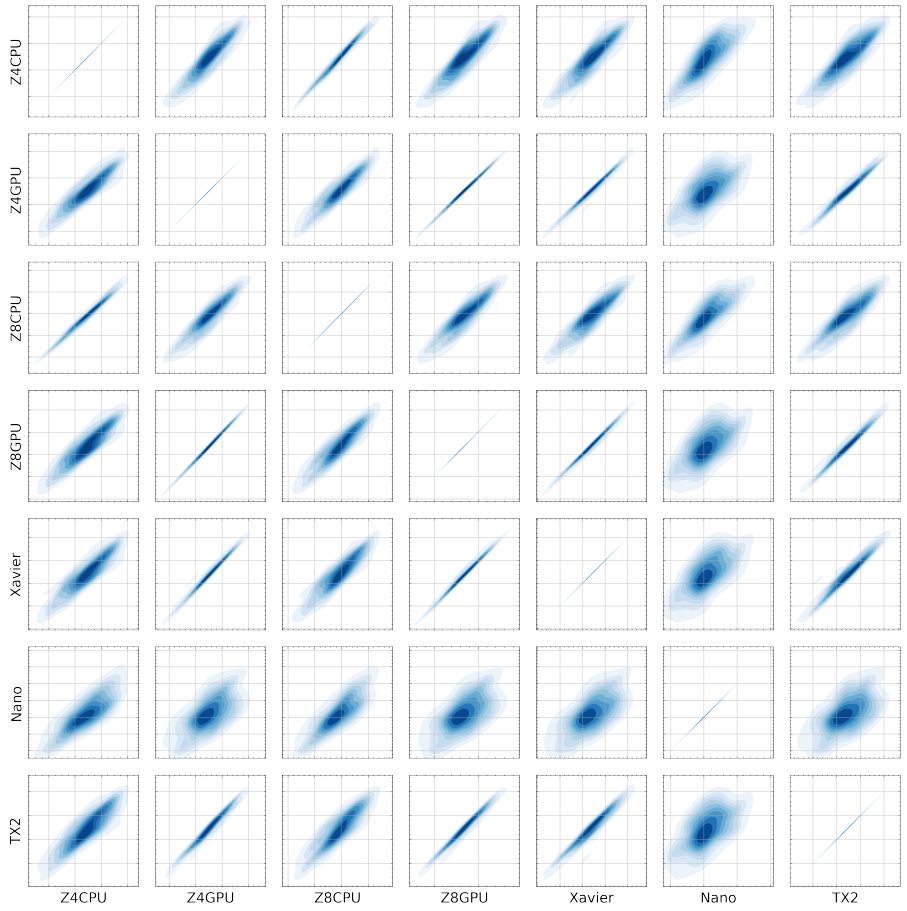

FIGURE 3.12: Illustration of the performance-monotonicity among different hardware platforms, where the latency here is measured on target hardware.

latency predictor, and can be obtained through one-time backward propagation which is quite efficient. Except for the above two terms, other terms in Eq (3.17) are apparently differentiable since only continuous transformations are involved. In this way, we are able to optimize both w and α by using the stochastic gradient descent (SGD) scheme as illustrated in Eq (3.5).

3.3.6 Scaling up SurgeNets

As illustrated in Figure 3.3, lightweight CNNs like MobileNetV2 [6] and MobileNetV3 [7] in practice fail to provide noticeable speedup when compared with non-lightweight ones like ResNet [19]. Here, we emphasize that this is caused by the low arithmetic intensity of MobileNetV2/V3, which in turn leads to the memory bottleneck during the runtime execution [8, 14], thereby suffering from

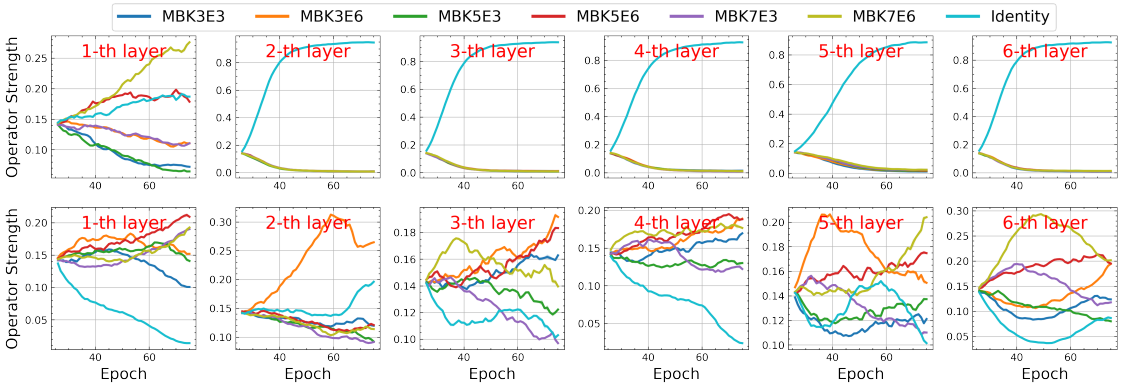

FIGURE 3.13: Illustration of the effectiveness of the proposed identity mapping regularization. Here, due to space constraints, we only visualize the architecture search process with respect to the first six layers, where *bottom* visualizes the architecture search process integrated with the proposed identity mapping regularization while *top* not.

significant resource underutilization [8]. However, if the arithmetic intensity is too high (see DenseNet [33] in Figure 3.3), the computation bottleneck arises, exponentially deteriorating the execution efficiency. The above observation motivates us to increase the arithmetic intensity[5] to balance the computation (FLOPs) and memory access (MACs), thereby being able to make full use of the available computation resources [8]. To this end, we introduce a simple but effective method, i.e., scale up the searched SurgeNets in terms of the network width. To show this, we take the convolutional layer as an example, where we use H, W, K_H, and K_W to represent the input dimensions and the filter dimensions. For simplicity, we assume the convolution stride to be 1, and thus the output dimensions are the same as the input ones. Besides, we use I and O to denote the number of input and output channels. Therefore, we can theoretically formulate the FLOPs count and the MACs count [43] as follows:

$$\text{FLOPs} = HWIOK_HK_W,\ \text{MACs} = HW(I+O) + IOK_HK_W \tag{3.18}$$

Then, following the definition of the arithmetic intensity [8], we can compute the arithmetic intensity as follows:

$$Intensity = \frac{\text{FLOPs}}{\text{MACs}} = \frac{HWIOK_HK_W}{HW(I+O) + IOK_HK_W} \tag{3.19}$$

[5]CNNs with higher arithmetic intensity are less memory bounded, and thus are easier to parallelize upon target hardware [14].

Besides, we use $\phi > 1$ to represent the scaling factor, and then substitute $I' = \phi I$ and $O' = \phi O$ into Eq (3.19):

$$Intensity = \frac{HWIOK_HK_W}{HW(I+O)/\phi + IOK_HK_W} \tag{3.20}$$

As seen in Eq (3.20), the arithmetic intensity monotonically increases with respect to the scaling factor ϕ. As such, we can increase the arithmetic intensity by scaling up CNNs in terms of the network width ϕ. Please note that, although FLOPs and MACs formulated in Eq (3.18) are not hardware-related, putting them together (i.e., $\frac{\text{FLOPs}}{\text{MACs}}$) can reflect the resource utilization on target hardware according to the roofline analysis [8]. To show this, we conduct a simple experiment on Z8GPU, in which we gradually increase the number of input/output channels in the convolutional layer, and then calculate the arithmetic intensity according to [8] and measure the runtime latency, respectively. The experimental results are shown in Figure 3.10, in which we observe that there exists *Comfort Zone* where we are allowed to scale up the searched architecture without increasing the on-device latency (see Figure 3.10 (*right*)). The intuition behind this is that, although the computational complexity increases as we scale up the neural architecture, the resource utilization on target hardware increases as well, thereby increasing the attainable computation resources on target hardware. Finally, the on-device latency remains unchanged (see Figure 3.10 (*right*)). More importantly, scaling up the neural architecture is able to improve the accuracy on target task [134]. However, as shown in Figure 3.3, a low arithmetic intensity makes target hardware memory-bounded, whereas a high arithmetic intensity makes target hardware computation-bounded. Therefore, it is of great significance to find the right balance between FLOPs and MACs. To derive the optimal scaling factor within *Comfort Zone*, we take a straightforward approach, i.e., we gradually increase the scaling factor until the runtime latency increases. As seen in Section 3.4, the optimal scaling factor could be device-specific since the computational capability of different hardware devices may vary. Besides, as illustrated in Figure 3.11, *Comfort Zone* is only applicable to hardware platforms with high parallelism such as GPUs, whereas hardware platforms with low parallelism such as CPUs do not have *Comfort Zone*.

Meanwhile, we find that the performance-monotonicity also exists among different hardware platforms:

TABLE 3.2: Comparisons with state-of-the-art CNNs on ImageNet. † indicates the architecture search is conducted on CIFAR10 while others are directly on ImageNet. ‡ indicates SE and Swish are used to further improve the accuracy. More specifically, SurgeNet-C1-A and SurgeNet-C1-B are for Z4GPU and TX2, and SurgeNet-C1-C is for Z8GPU and Xavier. SurgeNet-C2-A is for Nano, and SurgeNet-C3-Base is for Z4PCU and Z8CPU. Besides, the on-device latency with the blue color is measured after fusing the convolutional layer and the subsequent batch normalization layer into one single convolutional layer.

Architecture	Method	Accuracy (%)		Search Cost	Latency (ms)						
		Top-1	Top5	(GPU-hours)	Z4CPU	Z4GPU	Z8CPU	Z8GPU	Xavier	Nano	TX2
ResNet [19]	Manual	75.3	92.2	-	32.1	4.7	46.8	4.0	18.5	32.5	69.5
ShuffleNetV1 [42]	Manual	73.6	89.8	-	19.4	4.4	23.8	4.7	17.9	36.2	70.2
ShuffleNetV2 [43]	Manual	74.9	-	-	19.3	4.4	23.9	4.7	17.6	36.6	69.4
MobileNetV2 [6]	Manual	72.8	91.1	-	15.5	3.5	16.7	3.5	14.0	26.8	57.6
MobileNetV3‡ [7]	Manual	75.2	-	-	15.3	5.6	17.2	5.8	22.5	46.5	83.8
EfficientNet‡ [134]	Manual	76.3	93.2	-	23.3	7.5	29.3	7.6	29.3	57.3	113.7
DARTS [3]	Differentiable	73.3	91.3	96†	45.8	16.5	70.6	16.2	76.1	128.0	285.9
SNAS-Mild [20]	Differentiable	72.7	90.8	36†	24.6	9.8	71.0	10.9	46.0	76.1	167.2
SurgeNet-Base (ours)	Differentiable	77.6	93.7	30	24.1 / 17.2	4.8 / 3.0	29.3 / 20.2	5.7 / 3.3	19.0 / 11.5	37.7 / 24.1	79.6 / 46.3
MnasNet‡ [5]	RL	75.2	92.5	40,000	15.1	3.5	16.7	3.4	13.1	28.8	58.6
SPNAS [59]	Differentiable	75.0	92.2	30	18.2	5.2	18.9	5.2	19.9	40.7	92.3
FBNet-A [9]	Differentiable	73.0	-	216	14.9	4.8	15.8	4.6	17.9	36.9	88.3
FBNet-B [9]	Differentiable	74.1	-	216	17.1	4.9	18.6	5.0	19.0	39.9	91.7
FBNet-C [9]	Differentiable	74.9	-	216	18.4	5.4	20.7	5.2	20.5	42.6	99.6
ProxylessNAS-CPU [13]	Differentiable	75.3	-	200	20.5	5.0	22.8	4.5	16.2	36.4	82.0
ProxylessNAS-GPU [13]	Differentiable	75.1	92.5	200	14.1	3.4	16.7	3.3	11.7	28.1	58.8
ProxylessNAS-Mobile [13]	Differentiable	74.6	92.2	200	16.3	4.7	17.8	4.5	16.4	38.1	80.7
OFA-CPU [14]	Evolution	74.6	92.0	1,225	14.1	3.1	15.2	2.9	12.4	23.2	51.8
OFA-GPU [14]	Evolution	75.3	92.4	1,225	15.2	3.7	16.6	3.3	13.7	26.6	59.1
OFA-TX2 [14]	Evolution	75.4	92.4	1,225	15.3	3.5	17.3	3.2	13.4	26.5	58.7
SurgeNet-C1-Base (ours)	Differentiable	75.5	92.5	30	17.7 / 10.8	3.0 / 2.0	21.1 / 14.0	2.9 / 1.7	12.2 / 7.2	24.0 / 15.8	52.0 / 30.3
SurgeNet-C1-A (ours)	Differentiable	76.8	93.1	30	24.3 / 15.2	3.0 / 2.0	28.3 / 19.8	2.9 / 1.7	12.3 / 8.2	24.3 / 14.9	52.1 / 30.9
SurgeNet-C1-B (ours)	Differentiable	76.9	93.1	30	27.8 / 17.5	3.1 / 2.0	32.5 / 22.3	2.9 / 1.7	12.2 / 7.5	24.3 / 15.5	52.3 / 31.2
SurgeNet-C1-C (ours)	Differentiable	78.7	93.9	30	52.4 / 32.8	3.2 / 2.0	84.2 / 54.9	2.9 / 1.7	12.3 / 7.5	24.1 / 14.3	53.8 / 31.3
SurgeNet-C2-Base (ours)	Differentiable	75.4	92.5	30	16.5 / 11.3	3.2 / 2.1	17.3 / 11.9	3.1 / 1.7	13.6 / 7.6	22.7 / 14.2	54.6 / 32.5
SurgeNet-C2-A (ours)	Differentiable	77.4	93.4	30	23.5 / 14.8	3.2 / 2.2	31.7 / 23.8	3.1 / 1.7	13.3 / 7.4	22.7 /14.0	55.1 / 32.3
SurgeNet-C3-Base (ours)	Differentiable	75.3	92.5	30	14.6 / 10.0	3.3 / 2.1	15.1 / 11.5	3.3 / 2.1	13.3 / 7.6	25.1 / 17.1	55.9 / 33.2

Definition 3.1. Given two different hardware devices d_1 and d_2 and the architecture space $\mathcal{A}$, if $\forall arch \in \mathcal{A}$, $LAT_{d_1}(arch) = \gamma LAT_{d_2}(arch)$ always holds, where γ is a constant (e.g., 0.5). We say that d_1 and d_2 are performance-monotonic in $\mathcal{A}$.

Here, $LAT_d(arch)$ denotes the latency of $arch$ on target hardware d. As shown in Figure 3.12, the latency measured on Z4CPU is monotonic to the latency measured on Z8CPU. Thus, we consider that Z4CPU is performance-monotonic with Z8CPU, whereas such performance-monotonicity does not hold between Nano and Z8CPU as seen in Figure 3.12. More importantly, given two performance-monotonic hardware devices, the optimal architecture searched on one hardware can easily generalize to the other. Therefore, motivated by the above performance-monotonicity, we divide the experimental devices into three categories: (1) Z4GPU, Z8GPU, Xavier, and TX2 denoted as Hardware-C1; (2) Nano denoted as Hardware-C2; and (3) Z4CPU and Z8CPU denoted as Hardware-C3. The devices lied in the same category are performance-monotonic with each other. Therefore, we first search for the base model for each hardware category and then scale up the base model towards each target hardware within *Comfort Zone*. Note that we do not apply the above scaling strategy for Z4CPU and Z8CPU in Hardware-C3 since CPUs are of low parallelism and do not have *Comfort Zone* (see Figure 3.11 (*right*)).

3.4 Experiments

In this section, we employ SurgeNAS to search on seven hardware platforms, i.e., CPUs (Z4CPU and Z8CPU), GPUs (Z4GPU and Z8GPU), and edge accelerators (Xavier, Nano, and TX2). Here, Z4CPU and Z8CPU are Intel(R) Core(TM) i7-7820X and Intel(R) Xeon(R) E5-1650. Z4GPU and Z8GPU denote Nvidia Quadro P620 (2GB) and Nvidia Quadro GV100 (32GB). Xavier, Nano, and TX2 represent Nvidia Jetson AGX Xavier (16GB), Nvidia Jetson Nano (4GB), and Nvidia Jetson TX2 (8GB). We turn on the MAXN power mode in edge devices to maximize the computational capability [14]. For the sake of simplicity, we use the above hardware abbreviations across this work. For fair comparisons with existing works, all the latency measurements are reported under the same settings and with a batch size of 1 throughout our experiments, which are directly measured on target hardware. It is worth noting that all of our experiments are implemented using PyTorch.

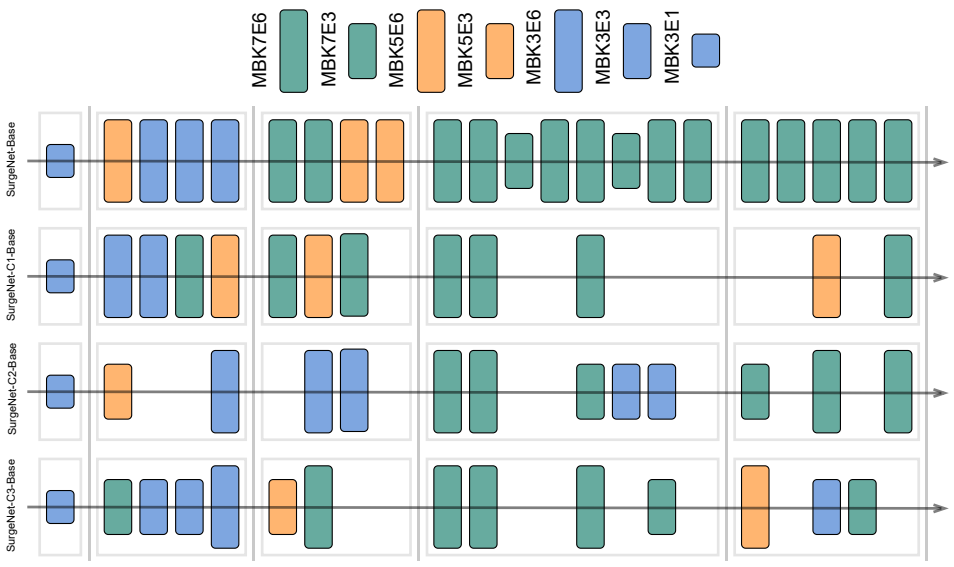

FIGURE 3.14: Visualization of SurgeNets for target hardware. Here, MBKxEy represents the operator with a kernel size of x and an expansion ratio of y.

3.4.1 Datasets and Experimental Settings

As mentioned in Section 3.3, conventional differentiable NAS approaches [3, 20, 21, 47] can only conduct the architecture search on small proxy tasks like CIFAR10 and CIFAR100 due to the memory consumption bottleneck, which in turn can only provide sub-optimal architecture solutions for target tasks like ImageNet as pointed out in [13]. In this work, different from the above-mentioned NAS works, the proposed SurgeNAS can directly search on the target task (i.e., ImageNet) to discover the optimal architecture solution for target hardware. More specifically, ImageNet consists of 1,000 object categories, where there are 1.28M training images and 50K validation images, all of which are high-resolution and equally distributed over all object categories. Besides, similar to [5, 9, 13, 61], we set the input resolution of the network to 224×224.

For the architecture search part, we follow [9, 46, 61, 74], where we randomly sample 100 out of 1,000 classes from the training dataset, which are used to optimize both the architecture parameters α and the network weights w during the search process. More specifically, to optimize the network weights, we employ the SGD optimizer with a momentum of 0.9, a weight decay of 3×10^{-5}, a norm gradient clipping of 5, and a learning rate of 0.1 which is annealed down to zero following the cosine schedule. Besides, to optimize the architecture parameters, we adopt the Adam

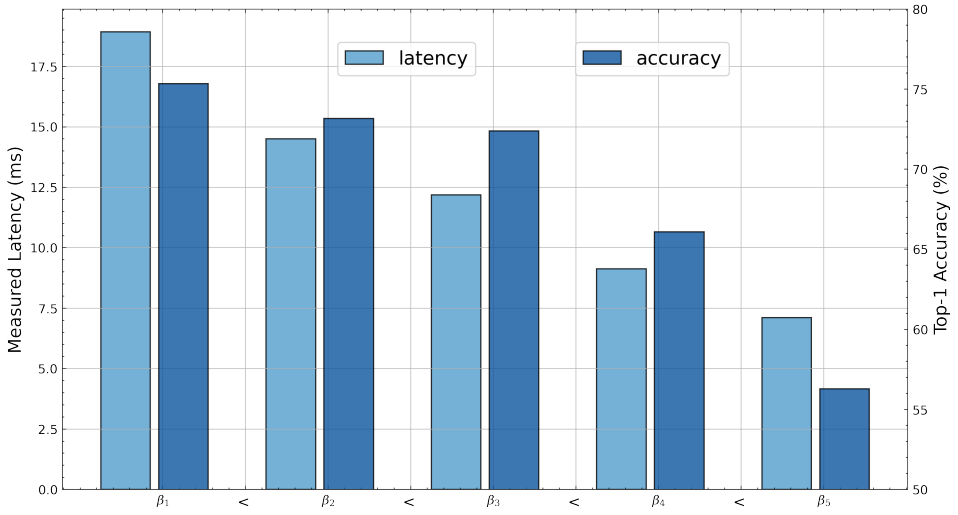

FIGURE 3.15: Illustration of the trade-off between accuracy and latency under different β. For simplicity, the above SurgeNets are trained for 50 epochs, where the SurgeNets in Table 3.2 are trained for 400 epochs for fair comparisons.

optimizer with a fixed learning rate of 7.5×10^{-4} and a weight decay of 1×10^{-3} [3]. In SurgeNAS, we train the supernet with a batch size of 232 for a total of 75 epochs, in which the first 25 epochs are referred as the architecture search warm-up stage where only the network weights get updated while the architecture parameters are frozen [46, 74, 137]. Apart from these, the standard data augmentations [13, 19] are applied throughout this work. Here, it is worth noting that all the architecture search experiments are conducted on one single Tesla V100 GPU. Furthermore, we name the hardware-aware CNNs searched by SurgeNAS as the SurgeNet series.

For the architecture evaluation part, we use the original validation dataset to report the final accuracy and compare with existing state-of-the-art approaches [5, 9, 13, 14]. To accomplish this goal, we re-train the searched neural architectures on the training dataset of ImageNet from scratch to derive the accuracy. In particular, all the searched neural architectures are trained for a total of 400 epochs with a batch size of 1024 on 8 Tesla V100 GPUs. Here, the optimizer is SGD with a momentum of 0.9 and a weight decay of 3×10^{-5}. Besides, the learning rate is initialized as 0.5, which is annealed down to zero following the cosine schedule. Apart from these, we apply the learning rate warm-up strategy for the first 5 epochs [3, 74].

3.4.2 Experimental Results

3.4.2.1 Architecture Search Results

As discussed in Section 3.3.6, we divide the hardware devices into three categories, which is according to the defined performance-monotonicity (see Definition 3.1 and Fig. 3.12), i.e., Z4GPU, Z8GPU, Xavier, and TX2 denoted as Hardware-C1, Nano denoted as Hardware-C2, and Z4CPU and Z8CPU denoted as Hardware-C3. To this end, in SurgeNAS, we first search for the base model corresponding to each hardware category, and then scale up the base model within *Comfort Zone* for further specializing to each target hardware device, respectively. It is worth noting that CPUs are of low parallelism and low computational capability, which do not suffer from the resource underutilization issue in practice. Thus, there does not exist *Comfort Zone* within CPUs as illustrated in Fig. 3.11 (*right*). As such, we only deploy the searched base model corresponding to Hardware-C3 for Z4CPU and Z8CPU. The searched SurgeNets are visualized in Fig. 3.14, in which SurgeNet-Base represents the one searched without latency constraint (i.e., $\beta = 0$ in Eq (3.16)). Besides, SurgeNet-C1-Base, SurgeNet-C2-Base, and SurgeNet-C3-Base correspond to the searched base models for Hardware-C1, Hardware-C2, and Hardware-C3. Subsequently, we scale up the searched base models within *Comfort Zone* for hardware devices in Hardware-C1 and Hardware-C2. To summarize, we have SurgeNet-C1-A for Z4GPU, SurgeNet-C1-B for TX2, SurgeNet-C1-C for Z8GPU and Xavier[6], SurgeNet-C2-A for Nano, and SurgeNet-C3-Base for Z4CPU and Z8CPU. In particular, we observe that the searched architectures show different preferences when targeting different hardware, e.g., Hardware-C1 prefers the wider architecture while Hardware-C3 favors the narrower one. This observation demonstrates there are no once-for-all architecture solutions, which necessitates deploying device-specific neural architectures upon target hardware.

3.4.2.2 Architecture Evaluation Results

The architecture evaluation results are summarized in Table 3.2. In particular, when without latency constraint, SurgeNet-Base achieves a top-1/5 accuracy of

[6]This is mainly because Z8GPU and Xavier in practice have similar scaling factors within each *Comfort Zone*, i.e., both are around 3.0.

TABLE 3.3: Comparisons with state-of-the-art NAS approaches on NAS-Bench-201 [16]. Among them, SNAS [20], GDAS [21], and FairNAS [22] are the most relevant methods to the proposed SurgeNAS. Following NAS-Bench-201, we report the average accuracy by repeating three architecture search experiments.

Approach	CIFAR10		CIFAR100		ImageNet	
	valid	test	valid	test	valid	test
ENAS [45]	39.8	54.3	15.0	15.6	16.4	16.3
DARTS-V1 [3]	39.8	54.3	15.0	15.6	16.4	16.3
DARTS-V2 [3]	39.8	54.3	15.0	15.6	16.4	16.3
SNAS [20]	90.1	92.8	69.7	69.3	42.8	43.2
GDAS [21]	90.0	93.5	71.1	70.6	41.7	41.8
FairNAS [22]	90.1	93.2	70.9	71.0	41.9	42.2
DSNAS [143]	89.7	93.1	30.9	31.0	40.6	41.1
PC-DARTS [46]	90.0	93.4	67.1	67.5	40.8	41.3
SurgeNAS (ours)	90.2	93.7	71.2	71.6	44.5	45.2

77.6%/93.7%, which is +2.3%/+1.5% higher than ResNet [19] and +1.3%/+0.5% higher than EfficientNet [134]. After enabling the latency-aware architecture search, the searched SurgeNets outperform existing lightweight CNNs like MobileNets [6, 7] and ShuffleNets [42, 43] in terms of both accuracy and efficiency. For example, SurgeNet-C1-Base obtains a top-1 accuracy of 75.5%, which is +2.7% higher than MobileNetV2 [6] and +0.3% higher than MobileNetV3 [7] while being ×1.2 and ×1.9 faster on Z4GPU and ×1.1 and ×1.8 faster on Xavier, respectively. Besides, when targeting the same hardware like TX2, SurgeNet-C1-Base achieves a +0.1% higher top-1 accuracy than OFA-TX2 [14] with ×1.1 speedup on TX2. Then, after scaling up within *Comfort Zone*, SurgeNet-C1-B obtains significant top-1/5 accuracy improvement (+1.4%/0.6%), whereas the inference latency remains the same level as SurgeNet-C1-Base on TX2. Notably, SurgeNet-C1-C attains the highest top-1/5 accuracy of 78.7%/93.9%, which is +3.6%/+1.4% higher than ProxylessNAS-GPU [14] while bringing ×1.1 speedup on Z8GPU and Xavier. Moreover, SurgeNet-C2-A accomplishes a top-1 accuracy of 77.4%, which is +2.5% higher than FBNet-C [9] while achieving ×1.9 speedup on Nano. Furthermore, SurgeNet-C3-Base reports a comparable top-1 accuracy with ProxylessNAS-CPU, while being ×1.4 and ×1.5 faster on Z4CPU and Z8CPU, respectively. More importantly, all the above SurgeNets are searched with a search cost of 30 GPU-hours, which is much lower when compared with other NAS approaches like 200 GPU-hours in ProxylessNAS [13], 216 GPU-hours in FBNet [9], and 1,225 GPU-hours in OFA [14]. Furthermore, after fusing the convolutional layer and the subsequent batch normalization layer into one single convolutional layer, the searched

SurgeNets achieve up to $\times 1.7$ speedup on target hardware.

3.4.3 Diagnostic Experiments

3.4.3.1 Effectiveness of the identity mapping regularization

In this section, we visualize the architecture search process of SurgeNAS with respect to the first six layers in Fig. 3.13. In particular, when without the proposed identity mapping regularization, the search process prefers to over-select the skip-connect operator in order to quickly converge as demonstrated in [136] (see Fig. 3.13 (*top*)). Then, as seen in Fig. 3.13 (*bottom*), the above over-selecting issue is well addressed after integrating the proposed identity mapping regularization into the proposed SurgeNAS, which clearly shows the effectiveness of the proposed regularization approach. More importantly, as seen in Fig. 3.4, the proposed regularization does not deteriorate the search process since it is equally imposed across all the operators [22]. Besides, it is worth noting that the search process in SurgeNAS will recover as the normal search [3] since we gradually decay φ as seen in Eq (3.6) to zero at the end of architecture search.

3.4.3.2 Effectiveness of the trade-off factor β

In this section, we conduct multiple architecture search experiments to show the effectiveness of the proposed latency-aware architecture search, or more specifically, we examine the effects of the trade-off factor β as shown in Eq (3.16). Here, we take Xavier as an example. More specifically, we gradually increase β from β_1 to β_5, whereas other experimental settings remain the same. In this way, we can obtain one architecture solution under each β setting. Afterwards, we re-train the searched architectures from scratch for 50 epochs, respectively. The experimental results are illustrated in Fig. 3.15, where we observe that a larger β encourages to search for architectures with lower latency while a smaller β prefers architectures with higher accuracy. This means that β can effectively trade off between the accuracy and the runtime latency, which shows the effectiveness of the proposed latency-aware architecture search. In practice, this brings considerable flexibility since we can dynamically control the search process towards accuracy or efficiency by imposing different β according to the requirements of target tasks. Particularly, β_5 leads to

the architecture which only consists of the skip-connect operator, thereby resulting in extremely poor accuracy.

3.4.3.3 Evaluations on NAS-Bench-201

In this section, we further compare the proposed SurgeNAS against previous relevant NAS methods on NAS-Bench-201 [16]. Following NAS-Bench-201, we focus on searching for the best architecture candidate in terms of the accuracy. Specifically, NAS-Bench-201 introduces a small search space with 15,625 different architecture candidates, all of which are trained from scratch using the same training recipe. In order to make the proposed SurgeNAS compatible with NAS-Bench-201, we exclude the latency regularization term by setting $\beta = 0$ in Eq (3.16) and conduct the architecture search experiments in the cell-based search space. For fair comparisons, we repeat three architecture search experiments under different settings of initialization seeds and report the average accuracy [16]. The experimental results, as well as the comparisons with previous relevant NAS methods, are summarized in Table 3.3. Specifically, we observe that SurgeNAS clearly outperforms previous state-of-the-art NAS approaches in terms of the accuracy on three datasets, especially on ImageNet, which demonstrates the superiority of SurgeNAS.

3.5 Conclusion

In this work, we introduce SurgeNAS, a hardware-aware differentiable neural architecture search (NAS) framework, to automatically search for efficient CNNs upon target hardware. In particular, we adopt the one-level optimization to avoid the inaccurate gradient estimation, in which an effective identity mapping regularization method is integrated to address the over-selecting issue. Besides, to mitigate the search memory bottleneck, we propose a novel ordered differentiable sampling approach, which greatly reduces the search memory consumption to the single-path level, thereby allowing to directly search on target tasks instead of small proxy tasks. Meanwhile, it guarantees the strict search fairness. Moreover, we introduce an efficient graph neural networks (GNNs) based latency predictor, which is further incorporated into SurgeNAS to achieve the latency-aware architecture search. Finally, we propose to scale up the searched SurgeNets within *Comfort Zone* to

improve the accuracy without worsening the inference efficiency. Extensive experiments are conducted on ImageNet with diverse hardware devices, which clearly demonstrate the superiority of SurgeNAS over existing state-of-the-art works in terms of accuracy, latency, and search efficiency.

Chapter 4

Flexible Hardware-Aware Neural Architecture Search[1]

Differentiable neural architecture search (NAS) is an emerging paradigm to automate the design of competitive deep neural networks (DNNs) using minimal computational resources. In practice, DNNs are subject to strict latency constraints and any violation may lead to catastrophic consequences (e.g., autonomous vehicles). However, to obtain the architecture that strictly satisfies the required latency constraint, previous hardware-aware differentiable NAS methods have to repeat a plethora of search runs to tune relevant hyper-parameters by trial and error, and as a result, the total design cost increases proportionally (empirically by 10 times). To overcome such limitations, we, in this chapter, introduce a lightweight and scalable hardware-aware NAS framework named LightNAS, which consists of two separate stages. In the first stage, we strive to search for the architecture that strictly satisfies the required latency constraint at the macro level in a differentiable manner, and more importantly, through a one-time search (i.e., *you only search once*). The architectures searched in the first stage are denoted as LightNets. After that, in the second stage, we introduce an efficient evolutionary scheme to further explore the micro-level channel configuration of each LightNet at low cost. To achieve

[1]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "You Only Search Once: On Lightweight Differentiable Architecture Search for Resource-Constrained Embedded Platforms." ACM/IEEE Design Automation Conference (DAC), 2022. and Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "LightNAS: On Lightweight and Scalable Neural Architecture Search for Embedded Platforms." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.

this, we propose an effective yet computationally cheap proxy, namely batchwise training estimation (BTE), as a plug-in complement to enable the channel-level exploration of LightNets on the fly such that the accuracy of LightNets can be improved without degrading the runtime latency on target hardware. Finally, extensive experiments are conducted on ImageNet with one popular embedded platform (i.e., Nvidia Jetson AGX Xavier), which explicitly demonstrate the efficacy of the proposed approach over previous state-of-the-art counterparts.

The remainder of this chapter is organized as follows. Section 4.1 introduces the research background. Section 4.2 presents the preliminaries and discusses the motivations. Section 4.3 elaborates on the proposed LightNAS. Section 4.4 presents the experimental settings and results. Finally, Section 4.5 concludes this chapter.

4.1 Introduction

Deep neural networks (DNNs) have been gaining increasing popularity in a wide variety of intelligent embedded scenarios, such as virtual reality (VR), object detection and tracking, etc., delivering impressive performance and enabling entirely new on-device experiences [7, 26]. Nonetheless, given that the network design space is prohibitively large [3, 9, 13], manually designing competitive DNNs requires a huge amount of engineering efforts to determine the optimal network configuration like the network depth and width. To alleviate this issue, neural architecture search (NAS) [44] has recently flourished, which strives to automate the design of efficient DNNs that can exhibit superior accuracy. In the literature, NAS studies can be mainly divided into reinforcement learning [44], evolution [49], and gradient-based [3] (a.k.a., differentiable) categories. However, both reinforcement learning and evolution-based NAS approaches suffer from substantial search overheads as shown in [3] (e.g., over 2,000 GPU-days [44] and 3,150 GPU-days [49]), whereas the differentiable counterpart is of significant search efficiency that can reduce the search cost by multiple orders of magnitude (e.g., 1 GPU-day[2] [3]). Thanks to the promising search efficiency, the differentiable NAS paradigm has recently emerged as the most dominant alternative in the NAS community [51].

[2]The search cost of DARTS [3] is approximated on a small proxy dataset, i.e., CIFAR10.

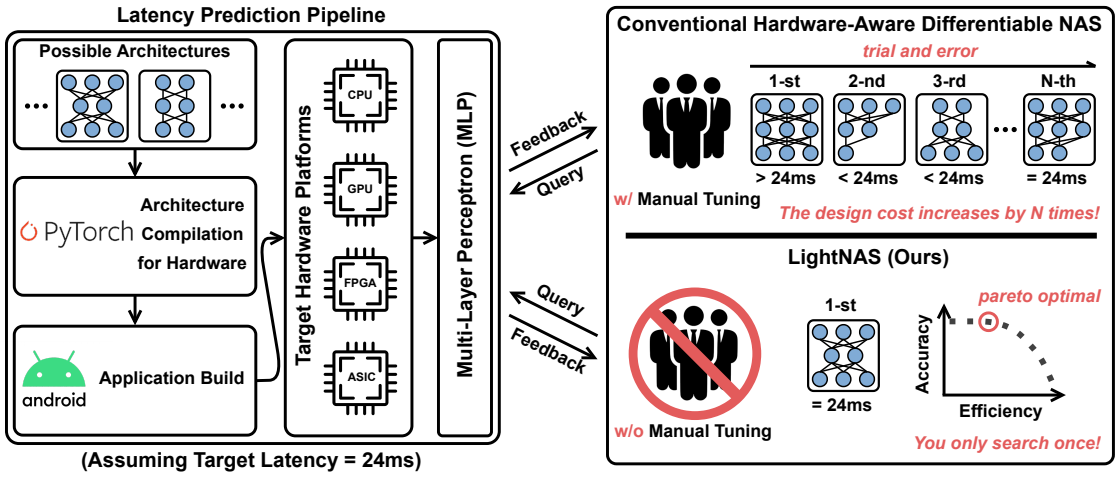

FIGURE 4.1: An intuitive overview of LightNAS. In contrast to previous relevant NAS methods that involve multiple search runs, LightNAS focuses on finding the required architecture in one single search (i.e., *you only search once*).

Despite the significant progress, early differentiable NAS works like DARTS [3] and its variants [20, 21] are hardware-agnostic, or more specifically, they merely focus on searching for competitive architectures in terms of the accuracy, regardless of other critical performance constraints like latency and energy [26]. As such, they end up with the architecture with promising accuracy on target task, which, however, comes at the cost of high computational complexity on target hardware, thereby impeding the practical deployment of DNNs, especially on resource-constrained embedded platforms [26]. Among them, [20] strives to search for resource-efficient DNNs according to the number of floating-point operations (FLOPs), but the number of FLOPs is an inaccurate proxy [9], which cannot translate to the actual latency and energy on target hardware as illustrated in Figure 4.2. Meanwhile, under the same number of FLOPs or parameters, the accuracy on target task may differ a lot as shown in NAS-Bench-201 [16]. To resolve this problem, several hardware-aware differentiable NAS methods are proposed [9, 13, 59, 60, 144], which incorporate the latency into the search objective to penalize the architecture with high latency, thereby leading to hardware-friendly architectures with high accuracy on target task but low latency on target hardware.

But even so, we should consider not only the *explicit* search cost – the time required for one single search, but also the *implicit* search cost – the time required for manual hyper-parameter tuning to find the required architecture. The rationale here is that, in real-world embedded scenarios like autonomous vehicles,

DNNs are subject to strict latency requirements, where any violation may cause catastrophic consequences. However, to obtain the required architecture with competitive accuracy while strictly satisfying the given latency constraint, previous hardware-aware differentiable NAS methods [9, 13, 59, 60, 144] have to perform multiple search runs to manually tune the hyper-parameters (see λ in Eq (4.3)) by trial and error. As a result, the total design cost increases proportionally (empirically by $\sim$ 10 times). We emphasize that previous NAS methods only report the explicit search cost for one single search while the huge implicit search cost for hyper-parameter tuning is excluded. Meanwhile, during the search process [9, 13, 60, 144], multiple sub-networks (paths) need to be optimized simultaneously (see Table 4.1), thereby leading to non-trivial memory overheads. And even worse, the above multi-path paradigm introduces considerable inconsistency between search and evaluation since the stand-alone architecture at the evaluation stage is a single-path sub-network [22, 48]. To overcome such limitations, we turn back to the following question:

> *Can we find the architecture that satisfies the specified performance constraint through a one-time search in both differentiable and lightweight manners??*

In addition to the macro-level exploration, previous methods like [6] have also delved into the micro-level exploration in terms of the channel configuration for macro-level building operators, striving for better performance in terms of both accuracy and hardware efficiency[3] [145]. However, previous methods like [6] employ the most brute-force approach to achieve the channel-level exploration, i.e., manually determining the optimal channel configuration by trial and error, which significantly increases the design cost. Besides, another line of research is channel pruning, which aims to prune redundant channels to improve the efficiency with negligible accuracy loss [26, 146], which, however, fails to guarantee the specified performance constraint as pointed out in [147]. To overcome these challenges, a recent work [145] introduces a simple yet effective solution, which consists of two separate stages. In the first stage, a slimmable network [148, 149] that can run at arbitrary channel configurations is trained from scratch, which is then applied in the second stage to approximate the accuracy of different channel configurations. Similar to [145], [127, 150] also focus on automatic channel-level explorations, in

[3]The macro level typically refers to the network structure (i.e., the operator type), whereas the micro level refers to the operator hyper-parameter (i.e., the channel configuration).

which a one-shot supernet is leveraged to evaluate the accuracy of different channel configurations. In practice, training a slimmable network or supernet is indeed much faster than the brute-force counterpart, but still requires substantial computational resources. For example, as shown in [151], training a slimmable ResNet [149] on 8 GeForce RTX 2080Ti GPUs takes ∼17 days (equivalent to 3,264 GPU hours). In view of this limitation, we focus on the following question:

> *Given an architecture candidate, can we further improve the accuracy under the specified performance constraint through channel-level explorations at low cost?*

To investigate the above two questions, we, in this chapter, introduce a lightweight and scalable hardware-aware NAS framework named LightNAS[4], which consists of two separate stages. In the first stage, we strive to search for the architecture that strictly satisfies the required latency constraint at the macro level in a differentiable manner, and more importantly, through a one-time search (i.e., *you only search once*). An intuitive overview of LightNAS is illustrated in Figure 4.1. The architectures searched in the first stage are denoted as LightNets. After that, in the second stage, we propose an efficient evolutionary scheme to explore the micro-level channel configuration of each searched LightNet at low cost. To achieve this, we introduce an effective and computationally cheap proxy, namely batchwise training estimation (BTE), which serves as a plug-in complement to enable the channel-level exploration of LightNets on the fly such that the accuracy of LightNets can be improved without degrading the runtime latency on target hardware. The architectures searched in the second stage are further denoted as LightNets++. Finally, we summarize the main contributions of this chapter as follows:

- **Fundamentals.** In contrast to previous well-established hardware-aware differentiable NAS works, LightNAS strives to explore the required architecture candidate that strictly satisfies the specified latency constraint through one single search (i.e., *you only search once*), which thus eliminates the prohibitive implicit search cost due to the manual hyper-parameter tuning.
- **Framework.** LightNAS features a simple yet effective MLP-based latency predictor to accurately estimate the on-device latency for different architecture candidates in the search space, which is then integrated into an efficient

[4]The preliminary version of LightNAS has been published in [85].

single-path differentiable search algorithm to enable hardware-aware architecture search. Meanwhile, instead of manually tuning hyper-parameters to guarantee the specified latency requirement, LightNAS automatically learns the optimal hyper-parameter configuration during the search process, thus being able to navigate the required architecture candidate in one single search. Furthermore, we introduce an effective proxy, namely batchwise training estimation (BTE), which is able to provide reliable yet computationally efficient estimation of the performance ranking among different channel configurations and thus allows us to enable channel-level explorations to further boost the attainable accuracy of LightNets at low computational cost.

- **Evaluations.** We conduct extensive experiments to show the superiority of the proposed LightNAS. In particular, LightNAS can effectively and efficiently find the architecture with competitive accuracy while satisfying the specified latency constraint through a one-time search, which outperforms previous relevant NAS methods in terms of both accuracy and search efficiency. Meanwhile, the accuracy of LightNets can be further improved after enabling the channel-level exploration using BTE.

4.2 Preliminaries and Motivations

In this section, we first introduce the background preliminaries on differentiable NAS (i.e., DARTS [3]). After that, we discuss the motivations behind this work.

4.2.1 Preliminaries on Differentiable NAS

We begin with the operator space denoted as $\mathcal{O} = \{o_k\}_{k=1}^{K}$, in which each element o_k represents an operator candidate. Following the weight-sharing NAS paradigm (a.k.a., one-shot NAS) [51], an over-parameterized network is constructed, namely supernet, where each searchable layer is composed of K operator candidates in the operator space $\mathcal{O}$. To relax the discrete network design space to become continuous, operators in the supernet are assigned with a set of architecture parameters $\alpha \in \mathbb{R}^{L \times K}$, where L corresponds to the number of searchable layers. As such, we

can derive that the search space $\mathcal{A}$ consists of $|\mathcal{O}|^L$ architecture candidates. Specifically, as shown in Figure 4.4, the supernet consists of all possible architecture candidates in the search space $\mathcal{A}$ so that the architecture candidates can inherit weights from the supernet to avoid training a huge amount of architecture candidates from scratch during the search phase. With the above in mind, we can mathematically formulate the output of the supernet $F(x)$ as follows:

$$F(x) = \sum_{l=1}^{L} \sum_{k=1}^{K} \left(\frac{\exp(\alpha_l^k)}{\sum_{k'=1}^{K} \exp(\alpha_l^{k'})} \cdot o_k(x_l) \right), \ s.t., \ o_k \in \mathcal{O} \tag{4.1}$$

where x_l is the input of l-th layer and x is the initial input. Thanks to the above continuous relaxation, both network weights w and architecture parameters α are allowed to be optimized using stochastic gradient descent (SGD) [3], or more specifically, an effective bi-level optimization strategy is applied, including one training phase to optimize w and one validation phase to optimize α:

$$\begin{aligned} &\underset{\alpha}{\text{minimize}} \ \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) \\ &s.t., \ \ w^*(\alpha) = \arg\min_w \mathcal{L}_{train}(w, \alpha) \end{aligned} \tag{4.2}$$

where $\mathcal{L}_{train}(\cdot)$ and $\mathcal{L}_{valid}(\cdot)$ denote the loss functions on the training and validation datasets, respectively. Once the search process terminates, we can determine the searched architecture by reserving the strongest operator candidate for each searchable layer while other operator candidates are removed, where the operator strength is defined as $\exp(\alpha_l^k)/\sum_{k'=1}^{K} \exp(\alpha_l^{k'})$ [3]. Besides, following recent differentiable NAS conventions [3, 9, 13, 59], the searched architecture needs to be retrained from scratch on target task to obtain decent accuracy for further deployment on target hardware. For more technical details about differentiable NAS, interested readers may refer to DARTS [3].

4.2.2 Observation and Motivations

The objective in Eq (4.2), however, focuses on the accuracy-only optimization, regardless of other important performance metrics like latency and energy [9]. As a result, the objective in Eq (4.2) derives the architecture with competitive accuracy, which, unfortunately, comes at the cost of extremely high computational complexity [26]. To tackle this issue, previous NAS methods [20, 134] exploit the multi-objective optimization scheme to achieve trade-offs between accuracy and

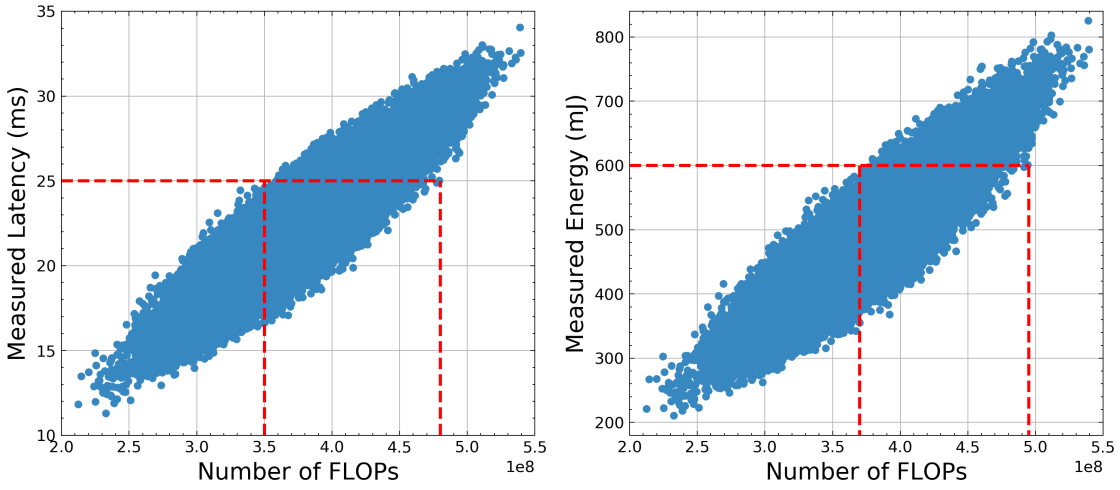

FIGURE 4.2: Relationships between the number of FLOPs and the latency (*Left*) and energy (*Right*) on Nvidia Jetson AGX Xavier with an input batch size 8.

efficiency, where they use hardware-agnostic metrics like the number of FLOPs or parameters to denote the network efficiency. Nonetheless, the number of FLOPs or parameters cannot reflect the latency and energy on target hardware as illustrated in Figure 4.2. Meanwhile, the number of FLOPs or parameters does not correlate with the accuracy on target task as shown in NAS-Bench-201 [16], either.

Subsequently, several hardware-aware differentiable NAS works are proposed [9, 13, 60, 144] to search for hardware-friendly architectures. Specifically, they incorporate the latency into the optimization objective as soft constraints to penalize the architecture with high latency, which is as follows:

$$\underset{\alpha}{\text{minimize}} \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) + \lambda \cdot LAT(\alpha) \tag{4.3}$$

where $LAT(\alpha)$ denotes the latency of the architecture encoded by α. Besides, $\lambda \geq 0$ is a constant coefficient to control the trade-off magnitude between accuracy and latency. In practice, the optimization objective defined in Eq (4.3) is able to derive hardware-friendly architecture solutions with both high accuracy and low latency, but only if a suitable λ is applied. The intuition behind this is simply that, if λ is too small, the latency penalty term $\lambda \cdot LAT(\alpha)$ will be effectively ignored. In contrast, if λ is too large, we will end up with the architecture that achieves extremely low latency on target hardware but sub-optimal accuracy on target task. Meanwhile, in real-world embedded scenarios like autonomous vehicles, DNNs must be executed under strict latency constraints, and any violation may

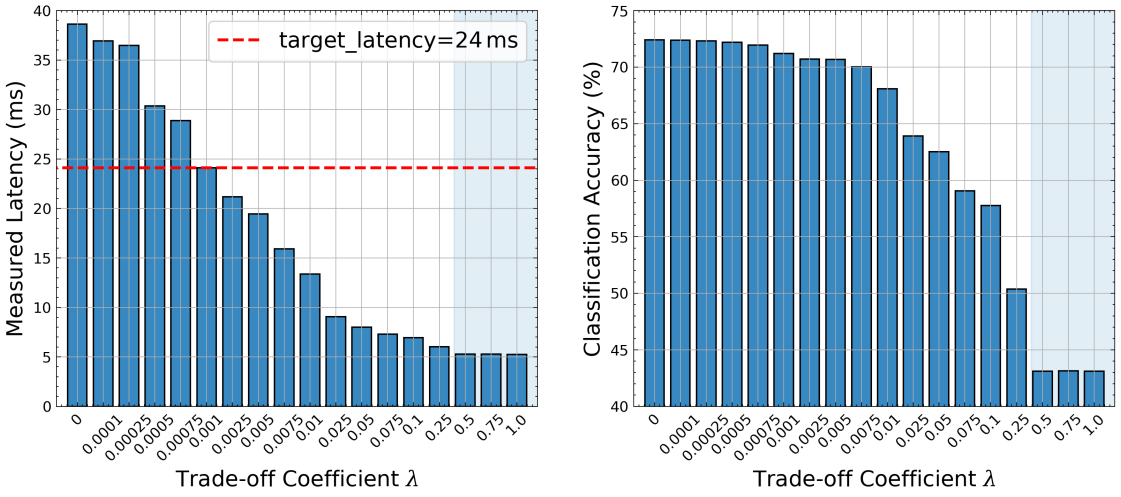

FIGURE 4.3: Illustration of the architecture search results under $\lambda \in [0, 1]$ in terms of latency on Nvidia Jetson AGX Xavier (*Left*) and accuracy on ImageNet (*Right*), where the searched architectures are trained from scratch for 50 epochs.

lead to catastrophic consequences. As a result, to find the architecture that satisfies the specified latency constraint, previous relevant NAS methods have to perform a tedious hyper-parameter sweep to manually tune λ by trial and error.

Nonetheless, the above manual hyper-parameter sweep requires us to repeat multiple search experiments (empirically 10 runs), and as a result, increases the total search cost by 10 times. To demonstrate this, we present a motivational experiment, in which we employ FBNet [9] as the search engine and conduct a series of architecture search experiments under different settings of $\lambda \in [0, 1]$. Once the search process terminates, we train the searched architecture from scratch on ImageNet [1] for 50 epochs to quickly evaluate the accuracy. Meanwhile, we measure the latency on Nvidia Jetson AGX Xavier with an input batch size of 8. As shown in Figure 4.3 (*Right*), the accuracy of the searched architectures varies from 43.1% to 72.4%. Among them, $\lambda = 0$ corresponds to the architecture with the highest accuracy of 72.4%, in which most of the operators are with the expansion ratio of 6 (i.e., computationally intensive). And $\lambda > 0.25$ leads to the architecture with the lowest accuracy of 43.1%, which only consists of *SkipConnect*. In particular, we observe that λ can effectively determine the trade-off magnitude between accuracy and latency. That is, a larger λ leads to the architecture with low latency and low accuracy whereas a smaller λ generates the architecture with high latency and high accuracy. However, λ is quite sensitive and difficult to control in order to satisfy the specified latency constraint. For example, to obtain the architecture with the

required latency of 24 ms, we should set λ to 0.001. But, next time if we require an architecture with the latency of 26 ms, we need to further manually tune λ within the range of $[0.00075, 0.001]$ as shown in Figure 4.3, which necessitates a plethora of trial-and-errors. We emphasize that previous relevant NAS methods [9, 13, 60, 144] only report the explicit search cost required for one single search, whereas the huge implicit search cost required for hyper-parameter tuning is ignored. Therefore, to address this issue, we focus on finding the architecture with competitive accuracy around the specified latency constraint through a one-time search (i.e., *you only search once*), which thus exhibits significant search efficiency and flexibility.

4.3 Methodology

In this section, we first elaborate on each component of the proposed LightNAS and then discuss the relationships with previous methods in the relevant literature to further distinguish the technical contributions of this work.

4.3.1 Architecture Search Space Design

Previous differentiable NAS methods like DARTS [3] and its variants [20, 60] explicitly advocate for the cell-level architecture search. That is, once a cell structure is determined, it will be repeatedly stacked to construct the stand-alone architecture. However, as pointed out in [5], enabling the layer diversity helps to strike the right balance between accuracy and latency. To this end, we follow the layer-wise architecture space design as widely used in recent hardware-aware differentiable NAS methods [9, 13, 14]. The layer-wise architecture search space is illustrated in Figure 4.4. Specifically, the operator space $\mathcal{O}$ is built upon MobileNetV2 [6], in which we allow a set of *MBConv* operators with diverse kernel sizes of $\{3, 5, 7\}$ and expansion ratios of $\{3, 6\}$ [13]. Please note that *MBConv* denotes the inverted residual block in MobileNetV2. Meanwhile, we include *SkipConnect*, which is computation-free, to achieve flexible search in terms of the network depth [9, 13]. Here, *SkipConnect* denotes the skip-connect operation, in which the output is typically equivalent to the input. Therefore, we have $|\mathcal{O}| = 7$ in LightNAS, and given that the supernet consists of $L = 22$ searchable operators where the first one is fixed [13], the architecture space size of LightNAS can be easily calculated as

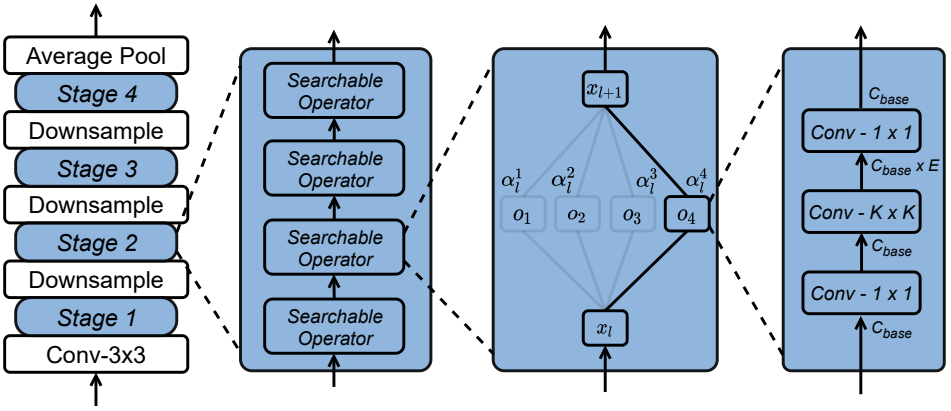

FIGURE 4.4: Illustration of the supernet structure over the search space, in which K and E denote the kernel size and the expansion ratio, respectively.

$|\mathcal{A}| = 7^{21} \approx 5.6 \times 10^{17}$. Unless explicitly stated, we do not apply extra techniques like Squeeze-and-Excitation (SE) [23] and Swish activation [7] in order to ensure ensure fair comparisons with previous relevant NAS methods [9, 13, 14].

4.3.2 Efficient Latency Prediction

Nonetheless, given that the architecture search space of NAS is prohibitively large (e.g., $|\mathcal{A}| \approx 5.6 \times 10^{17}$ in LightNAS), directly measuring the latency on target hardware for every possible architecture *arch* is quite tedious and also involves significant computation resources [13]. To overcome such limitations, we introduce an accurate yet efficient latency predictor to approximate the on-device latency for $arch \in \mathcal{A}$ with negligible computation overheads. With this goal in mind, we first encode *arch* using a sparse matrix $\overline{\alpha} \in \{0,1\}^{L \times K}$, where $\overline{\alpha}_l^k = 1$ indicates that the k-th operator is reserved for the l-th layer of *arch* while other operators are discarded. As such, we are able to formulate $\overline{\alpha}$ as follows:

$$\overline{\alpha}_l^k = \begin{cases} 1, & \text{if } \alpha_l^k = \arg\max ||\alpha_l|| \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

Therefore, since the supernet is composed of L searchable layers, we can simply interpret that the architecture encoding matrix $\overline{\alpha}$ contains L entries with values of 1, whereas the remaining entries are with values of 0.

Subsequently, we leverage a multi-layer perceptron (MLP) model for the prediction purpose, which consists of three fully-connected layers with 128, 64, and 1

neurons. Furthermore, we sample 10,000 random architectures from the search space $\mathcal{A}$, which are then measured on Nvidia Jetson AGX Xavier. After that, we divide the above sampled architectures and the latency measurements into two folds with 80% as the training set and 20% as the validation set. Next, we flatten $\overline{\alpha}$ corresponding to each sampled architecture and feed it into the MLP-based latency predictor for the training purpose. The rationale behind picking up an MLP-based latency predictor rather than analytic latency predictors is simply that MLP models can maintain strong differentiability with respect to their input and output. This is necessary in the context of differentiable NAS since we have to calculate the gradient of the architecture parameters for differentiable optimization during the search process. In contrast, analytic latency predictors have to consider various factors for reliable analytic latency prediction and may not guarantee strong differentiability with respect to their input and output. The latency prediction results on the validation set are illustrated in Figure 4.5 (*Left*), where we find that the proposed latency predictor achieves an extremely low root-mean-square error (RMSE) of 0.04 ms. More importantly, once the latency predictor is well trained, it is able to approximate the on-device latency for $arch \in \mathcal{A}$ through a one-time forward inference, which takes less than one millisecond, and thus introduces negligible computation overheads. In the meantime, given a new hardware platform or a new search space, we need $\sim$ 1 hour to re-build the latency predictor, which is trivial compared with the search cost in the context of NAS (see Table 4.2). In practice, the design cost of the latency predictor may vary. That is, given a hardware platform that has less computational resources or requires additional cost for compilation, we may need more than $\sim$ 1 hour to re-build the latency predictor.

Next, we compare the proposed latency predictor with the latency lookup table (LUT) as widely used in recent hardware-aware NAS methods [9, 13, 14]. The predicted results of LUT are shown in Figure 4.5 (*Right*), where we observe that there exists a consistent gap (about 11.48 ms) between the predicted latency and the measured ground truth. We conjecture that this prediction gap is caused by the inter-layer data movements and the parallelization inconsistency between the layer-wise and network-wise executions. And even though the above prediction gap is eliminated, the RMSE of LUT still remains 0.41 ms, which is much worse than the proposed MLP-based latency predictor as seen in Figure 4.5 (*Left*). We emphasize that the goal of this work is to search for the architecture around the specified latency, and thus an accurate latency predictor is of great necessity.

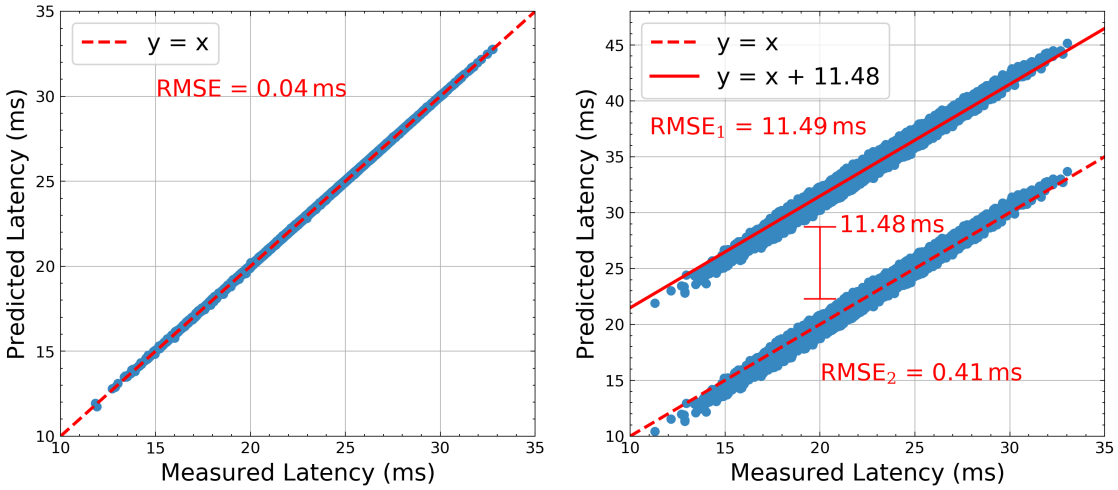

FIGURE 4.5: Illustration of the latency prediction results with the proposed latency predictor (*Left*) and the latency lookup table (LUT) [9, 13, 14] (*Right*).

Furthermore, we elaborate on the necessity of the proposed latency predictor in the context of differentiable NAS. On the one hand, we note that the architecture search experiments must be conducted on GPUs since the optimization of the supernet is computationally expensive [3, 60]. However, target hardware in practice may be different from GPUs (e.g., Nvidia Jetson AGX Xavier in this work). As such, the cross-hardware measurements significantly slow down the search process since the search process on GPUs cannot continue until the latency measurement on target hardware is obtained. On the other hand and more importantly, measuring the latency on target hardware for possible architectures, despite being accurate, is discrete with respect to the architecture parameters α [25]. The intuition behind this is that the stand-alone architecture to be measured on target hardware is discretized from the supernet and does not contain the architecture parameters α. As a result, we cannot derive the gradient of α (i.e., $\frac{\partial LAT(\alpha)}{\partial \alpha}$) from the on-device latency measurement. In contrast, the proposed latency predictor only consists of continuous transformations with respect to α, which not only accurately approximates the latency for possible architectures during the forward propagation but also enables to calculate the gradient of α during the backward propagation. We emphasize that differentiable NAS [3] heavily relies on the differentiability of α. That is, without $\frac{\partial LAT(\alpha)}{\partial \alpha}$, α cannot be updated using gradient descent during the search process as shown in Eq (4.11).

4.3.3 Lightweight Differentiable Architecture Search

As shown in previous differentiable NAS approaches [3, 9, 20, 60], multiple sub-networks (paths) must be simultaneously optimized during the search process, inevitably suffering from the memory bottleneck [13] and also violating the equality principle [22]. Specifically, the equality principle defines that the supernet during the search process should be trained in the same way as the stand-alone architecture, and violating the equality principle may lead to degraded performance in terms of the accuracy [22]. To this end, we introduce a lightweight differentiable architecture search approach to reduce the optimization complexity of the supernet to the single-path level, thereby effectively alleviating the memory bottleneck during the search process [48]. Here, we use $arch = \{op_l\}_{l=1}^{L}$ to denote the stand-alone architecture candidate. Therefore, once the search process terminates, we are able to calculate the probability that $arch$ is selected as follows:

$$P(arch) = \prod_{l=1}^{L} P(op_l = o_k), \ s.t., \ o_k \in \mathcal{O} \tag{4.5}$$

where $P(op_l = o_k)$ is the probability of o_k being selected at the l-th layer of $arch$. Following [3, 9], we can mathematically formulate $P(op_l = o_k)$ as follows:

$$P(op_l = o_k) = \frac{\exp(\alpha_l^k)}{\sum_{k'=1}^{K} \exp(\alpha_l^{k'})} \tag{4.6}$$

For the sake of simplicity, we replace $P(op_l = o_k)$ with P_l^k in the remainder of this paper. In practice, to determine the stand-alone architecture $arch$, the most intuitive and straightforward strategy is to optimize the architecture parameters α over the architecture search space $\mathcal{A}$ in a discrete manner. However, given that the architecture search space of LightNAS is prohibitively large as illustrated in Section 4.3.1, iterating possible architectures over the search space $\mathcal{A}$ one by one inevitably requires a huge amount of computation resources [3, 9]. Therefore, to resolve this issue, we closely follow [9, 20, 21] and leverage the Gumbel Softmax reparameterization trick [10] to relax the discrete architecture space to become continuous, which, consequently, allows the architecture parameters α to be optimized using gradient descent. Furthermore, we reparameterize P_l^k as follows:

$$\widehat{P_l^k} = \frac{\exp[(P_l^k + G_l^k)/\tau]}{\sum_{k'=1}^{K} \exp[(P_l^{k'} + G_l^{k'})/\tau]} \tag{4.7}$$

where $G \in \mathbb{R}^{L \times K}$ represents the random variable drawn from the Gumbel distribution (i.e., Gumbel$(0, 1)$[5] [10]). Apart from this, τ denotes the softmax temperature

[5] As shown in [10], G is equivalent to $-\log(-\log(U))$, where U is the uniform distribution.

to control the sampling process, which is initialized as 5 and then gradually decays to zero at the end of the search process. It is worth noting that, once converged, the above relaxation is unbiased as proved in [10] that $\lim_{\tau \to 0} \widehat{P}_l^k = P_l^k$. Finally, we are able to re-formulate the output of the supernet $F(x)$ as follows:

$$F(x) = \sum_{l=1}^{L} \sum_{k=1}^{K} \left(\overline{P}_l^k \cdot o_k(x_l) \right), \ s.t., \ o_k \in \mathcal{O} \tag{4.8}$$

where $\overline{P} \in \{0,1\}^{L \times K}$ denotes the binarization of $\widehat{P}$. Specifically, $\overline{P}$ is obtained by binarizing the largest entry in each row to 1 and the remaining entries to 0:

$$\overline{P}_l^k = \begin{cases} 1, & \text{if } \widehat{P}_l^k = \arg\max ||\widehat{P}_l|| \\ 0, & \text{otherwise} \end{cases} \tag{4.9}$$

As a result, only one single-path sub-network, instead of the entire supernet, requires to be activated and optimized during the forward propagation since $\overline{P} \in \{0,1\}^{L \times K}$. The intuition behind this is simply that the output of the supernet $F(x)$ as illustrated in Eq (4.8) only depends on the operator with $\overline{P}_l^k = 1$, whereas the output of the operator with $\overline{P}_l^k = 0$ is eliminated due to the multiplication property of zero. Please note that, different from Softmax, Gumbel Softmax is not deterministic with respect to the input probability P, which is due to the random Gumbel variable G and the temperature τ. As pointed out in [10], a larger τ enforces smoother output, while a smaller τ makes the output closer to the categorical variant of P. Given the above, during the early search process, the search engine randomly (uniformly) discretizes the single-path sub-networks to optimize. As a result, the operator candidates in the supernet are roughly optimized in an equal fashion in the early search process, after which the search engine focuses on finding the operator candidate that leads to better accuracy. The above aligns with the expectation fairness in [22].

To summarize, the above single-path mechanism achieves two main benefits. On the one hand, it brings significant memory efficiency because the optimization complexity during the search process has been reduced to the single-path level, and considering that the GPU memory is constant, we are allowed to apply a larger batch size to speed up the search process. On the other hand, the above single-path mechanism forces the search process to strictly satisfy the equality principle [22], i.e., the supernet and the searched sub-network should be trained in the same single-path manner. Furthermore, we note that the single-path mechanism

in LightNAS is fundamentally different from ProxylessNAS [13]. In fact, ProxylessNAS randomly discretizes the single-path sub-networks from the supernet using binary gates, which suffers from significant bias [138]. To alleviate this, LightNAS continuously reparameterizes the probability distribution P using Gumbel Softmax, which is then employed to discretize the single-path sub-networks from the supernet. Apart from this, during each search iteration, ProxylessNAS randomly samples two different single-path sub-networks to optimize the architecture parameters α, in which one path is enhanced with increased α and the other path is attenuated with decreased α. Therefore, the optimization complexity in ProxylessNAS is $\mathcal{O}(2^2)$ in terms of both inference time and memory consumption. In contrast, the optimization complexity in LightNAS is $\mathcal{O}(1)$.

4.3.4 Flexible Hardware-Aware Architecture Search

The search algorithm in Section 4.3.3 focuses on optimizing the search process in terms of the accuracy, which, however, ignores other important hardware performance constraints in real-world embedded scenarios, for example, the on-device latency as the most dominant one [5, 9, 13]. Therefore, to obtain hardware-friendly architecture solutions, we further integrate the proposed latency predictor into LightNAS so as to achieve hardware-aware architecture search. The optimization objective of LightNAS is then given as follows:

$$\underset{\alpha}{\text{minimize}} \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) + \lambda \cdot \left(\frac{LAT(\overline{\alpha})}{T} - 1 \right) \tag{4.10}$$

where T is the specified latency constraint on target hardware and λ is the trade-off coefficient to control the trade-off magnitude between accuracy and latency. Besides, $LAT(\overline{\alpha})$ denotes the latency of the architecture encoded by α. In practice, λ is first converted into $\overline{\alpha}$ according to Eq (4.4), which is then taken as the input of the latency predictor. Note that, different from previous relevant NAS methods [9, 13, 59, 60], λ in Eq (4.10) is not a fixed constant but a learnable hyper-parameter, which is optimized during the search process. As such, instead of manual hyper-parameter tuning, LightNAS can automatically explore the optimal hyper-parameter configuration during the search process, which not only maximizes the accuracy but also guarantees the specified latency constraint (i.e., $LAT(\alpha) = T$). For the sake of simplicity, we use $\mathcal{L}(w, \alpha, \lambda)$ to denote the optimization objective defined in Eq (4.10). Finally, w and α are optimized using gradient

descent like [3], whereas λ is optimized using gradient ascent as follows:

$$\begin{aligned} w^* &= w - \eta_w \cdot \frac{\partial \mathcal{L}(w, \alpha, \lambda)}{\partial w}, \ \alpha^* = \alpha - \eta_\alpha \cdot \frac{\partial \mathcal{L}(w, \alpha, \lambda)}{\partial \alpha} \\ \lambda^* &= \lambda + \eta_\lambda \cdot \frac{\partial \mathcal{L}(w, \alpha, \lambda)}{\partial \lambda} = \lambda + \eta_\lambda \cdot \left(\frac{LAT(\alpha)}{T} - 1 \right) \end{aligned} \tag{4.11}$$

where η_w, η_α, and η_λ are the learning rates of w, α, and λ. Below we analyze the efficacy of LightNAS to guarantee $LAT(\alpha) = T$ in one single search. As shown in Figure 4.3, λ is able to effectively control the trade-off magnitude between accuracy and latency. For example, $\lambda = 0.001$ corresponds to the architecture with the accuracy of 71.2% and the latency of 24.1 ms. And $\lambda = 0.0001$ ends up with the architecture with the accuracy of 72.4% and the latency of 37.0 ms. These demonstrate that a larger λ leads to the architecture with low accuracy and low latency, whereas a smaller λ generates the architecture with high accuracy and high latency. With the above in mind, during the search process, if $LAT(\alpha) > T$, the gradient ascent scheme increases λ to reinforce the latency regularization magnitude, and as a result, $LAT(\alpha)$ decreases towards T in the next search iteration. Likewise, if $LAT(\alpha) < T$, the gradient ascent scheme then decreases λ to diminish the latency regularization magnitude, and the search engine therefore increases $LAT(\alpha)$ towards T in the next search iteration. Consequently, the search engine ends up with the architecture that strictly satisfies the required latency constraint $LAT(\alpha) = T$. To demonstrate this, we visualize the search process under various latency constraints in Figure 4.12. Specifically, we observe that LightNAS explicitly focuses on finding the architecture around the specified latency constraint T during the search process, which is consistent with the analysis presented above. More importantly, the above differentiable optimization can be automatically performed during the search process, which does not involve any manual hyper-parameter tuning over either the trade-off coefficient or its learning rate. Meanwhile, the above differentiable optimization does not sacrifice the attainable accuracy during the search process. The rationale here is that LightNAS focuses on exploring possible architecture candidates around the specified latency constraint. This optimizes the attainable accuracy during the search process and also satisfies the specified latency constraint at the end of the search process.

Finally, since $\mathcal{L}(w, \alpha, \lambda)$ is differentiable with respect to w and λ as shown in [3] and Eq (4.11), below we provide the differentiable analysis with respect to α:

$$\begin{aligned}\frac{\partial \mathcal{L}(w,\alpha,\lambda)}{\partial \alpha} &= \frac{\partial \mathcal{L}_{valid}}{\partial \alpha} + \frac{\lambda}{T} \cdot \frac{\partial LAT(\alpha)}{\partial \alpha} \\ &= \frac{\partial \mathcal{L}_{valid}}{\partial \overline{P}} \cdot \frac{\partial \overline{P}}{\partial \widehat{P}} \cdot \frac{\partial \widehat{P}}{\partial P} \cdot \frac{\partial P}{\partial \alpha} + \frac{\lambda}{T} \cdot \frac{\partial LAT(\alpha)}{\partial \overline{P}} \cdot \frac{\partial \overline{P}}{\partial \widehat{P}} \cdot \frac{\partial \widehat{P}}{\partial P} \cdot \frac{\partial P}{\partial \alpha}\end{aligned} \tag{4.12}$$

where $\frac{\partial \overline{P}}{\partial \widehat{P}} \approx 1$ because $\overline{P}$ is the binarization of $\widehat{P}$ [142]. Besides, with the equivalence of $\overline{\alpha}$ and $\overline{P}$ as illustrated in Eq (4.4) and Eq (4.9), $\frac{\partial LAT(\alpha)}{\partial \overline{P}}$ is determined by the network weights of the latency predictor $LAT(\cdot)$, which can be easily obtained through a one-time backward propagation. That is, for one architecture candidate, we need to back-propagate the latency predictor once, which takes less than one millisecond. Apart from these terms, other gradient terms in Eq (4.12) are apparently differentiable since only continuous transformations are involved [3, 10]. Finally, the differentiable analysis of λ is given in Eq (4.11).

Remarks. We note that the above search optimization objective can optimize not only the inference efficiency (e.g., latency and energy consumption) but also the training efficiency. To achieve this, similar to latency and energy prediction, we can first build an accurate performance predictor to reliably estimate the training overheads of different architecture candidates. After that, we can integrate the above performance predictor into the above search optimization objective to further explore efficient networks with optimized training efficiency, which can also end up with the optimal network around the specified training overhead.

4.3.5 Channel-Level Explorations

Apart from the macro-level exploration across the entire network, previous methods like [6] have also digged into the fine-grained exploration in terms of the channel configuration at the micro level, which further improves both accuracy and inference efficiency [145]. Nonetheless, the aforementioned methods take the most brute-force strategy to determine the channel configuration in each building block by trial and error, thereby significantly increasing the total design cost. Besides, another alternative is the channel pruning technique [146], which primarily resorts to pruning redundant channels to reduce the network redundancy [26]. However, previous differentiable channel pruning techniques mainly focus on reducing the computational complexity of the given architecture according to hardware-agnostic

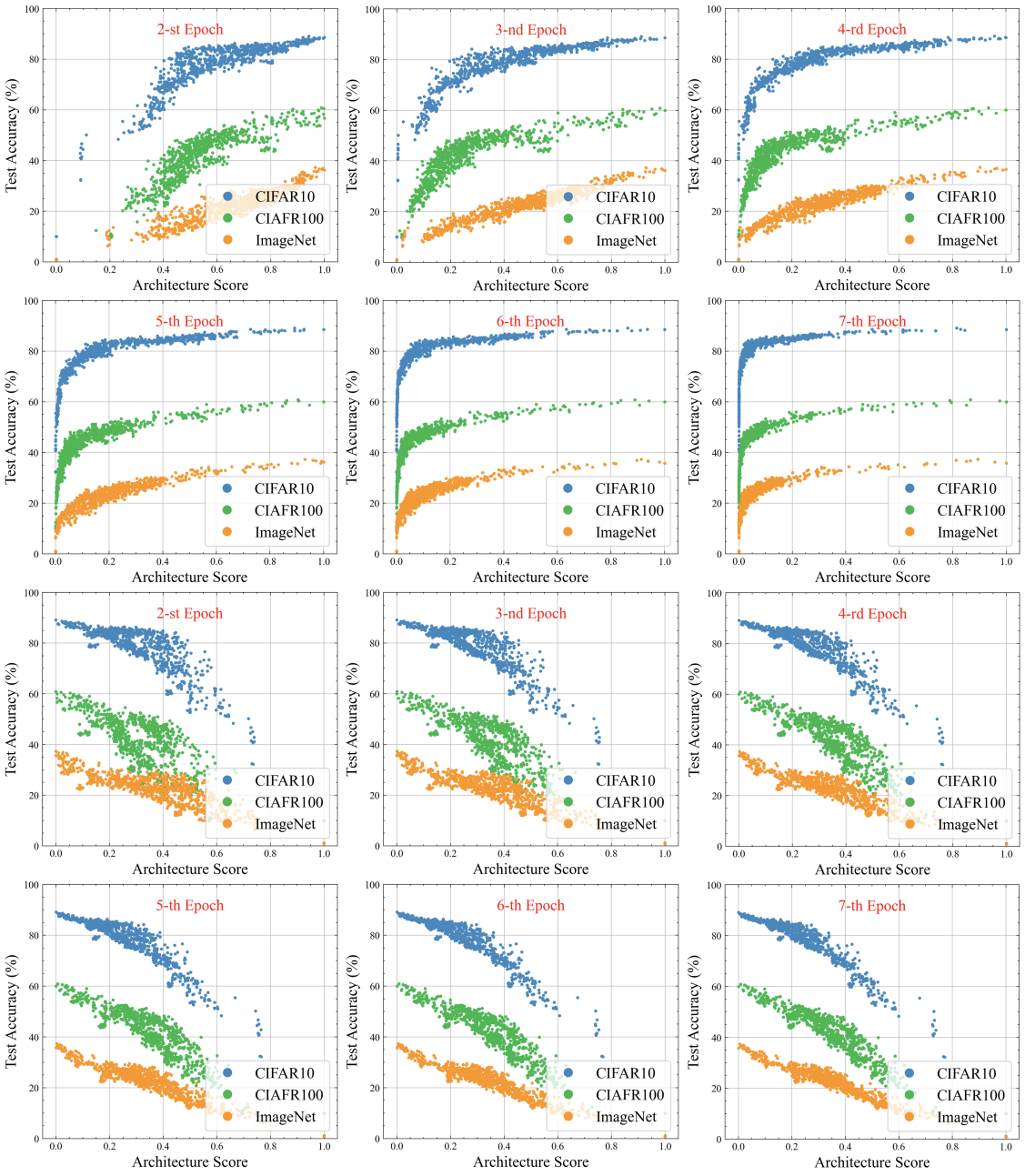

FIGURE 4.6: Comparisons of the proposed batchwise training estimation (BTE) (*Top* six sub-figures) and another relevant proxy [15] (*Bottom* six sub-figures), in which 1,004 random architectures are sampled from NAS-Bench-201 [16].

metrics (e.g., the number of FLOPs and parameters) [147], which consequently fail to generate hardware-aware architecture solutions with optimized latency and energy. To address these, [145] introduces a simple yet effective solution to automatically search for the optimal channel configuration, which is composed of two

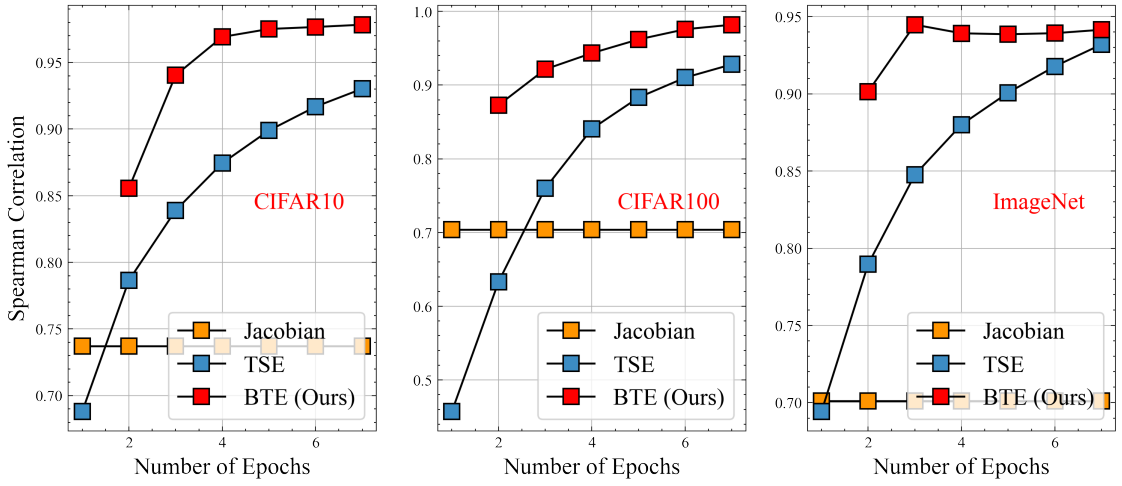

FIGURE 4.7: Comparisons of the correlation performance of BTE with TSE [15] and Jacobian [17, 18], where the accuracy is queried from NAS-Bench-201 [16].

separate stages. In the first stage, a slimmable network [148, 149] that can execute arbitrary channel configurations is trained from scratch, which then serves as the accuracy estimator in the second stage to approximate the accuracy of different channel configurations on the fly. Similar to [145], [127, 150] also focus on automatic channel-level explorations, in which a one-shot supernet is leveraged to evaluate the accuracy of different channel configurations. Despite the promising performance, these methods [127, 145, 150], in practice, suffer from substantial computation overheads, especially training a slimmable network or supernet [17, 18].

To alleviate the above-mentioned computation overheads, instead of training a slimmable network or supernet, another recent line of research is explicitly dedicated to approximating the accuracy of different architectures at initialization using zero-cost proxies [17, 18]. Specifically, following [17, 18], zero-cost proxies, in a more precise way, represent the estimators that can reliably reflect the performance of the given architecture candidate at initialization, in which zero training budgets are required. For example, Jacobian [17, 18], also known as Jacobian covariance, is to capture the covariance of input gradients across different images within the same input batch. In practice, Jacobian can be applied to those randomly initialized architecture candidates without being trained from scratch, thereby necessitating zero training budgets. Nonetheless, as shown in [15], the zero-cost proxies described above suffer from sub-optimal rank correlation performance when compared with the state of the art, are inconsistent across different datasets, and more importantly, cannot be further improved with additional training budgets. To

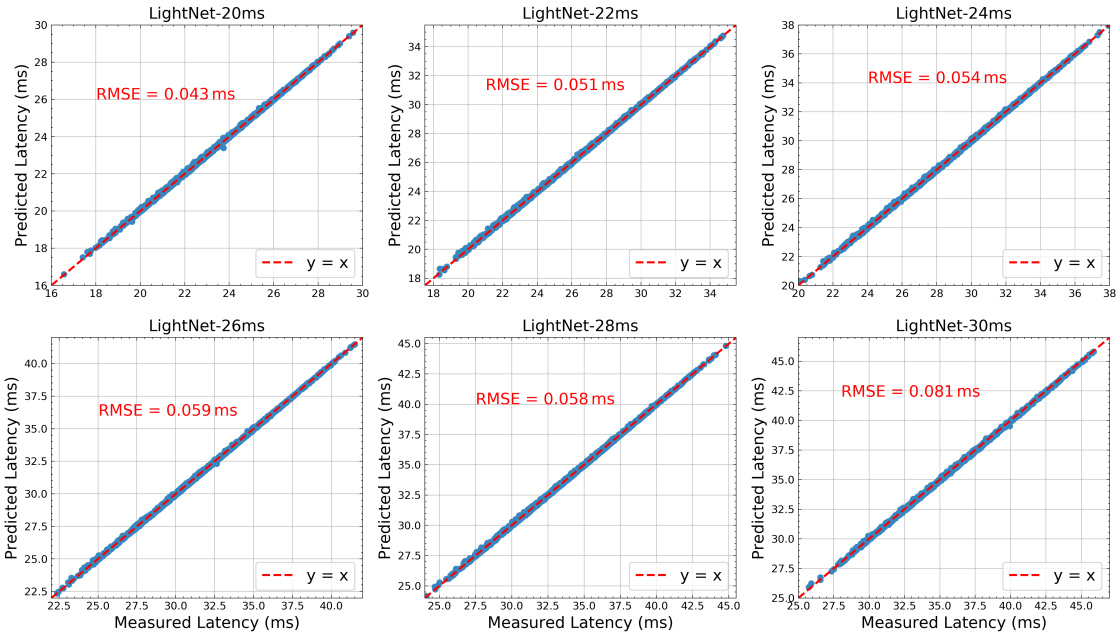

FIGURE 4.8: Illustration of the latency prediction results under different channel configurations using the MLP-based latency predictor in Section 4.3.2.

this end, [15] introduces a reliable yet computationally efficient proxy named TSE, which empirically investigates the quality of different architectures by summing the losses during the training phase. In particular, the rank correlation performance of TSE can be further improved when additional training budgets are allocated, which, however, ignores the training dynamics in the very early epochs due to the random initialization [152]. To resolve this, we propose an effective proxy at near-zero cost, namely Batchwise Training Estimation (BTE), which can achieve better rank correlation performance on multiple datasets (i.e., CIFAR10, CIFAR100, and ImageNet) than [15] with even less training budgets. Please note that we follow TSE [15] to call the cost in BTE as near-zero cost, which is compared against the full training scheme. For example, DARTS [3] trains the searched architecture from scratch on CIFAR10 for 600 epochs. In contrast, BTE only requires to train the given architecture candidate from scratch for several epochs, which leads to significantly reduced training budgets. Finally, the definition of BTE is given as follows:

Definition 4.1 (BTE). *Let* $\mathcal{D}_{train} = \{(x_1, y_1), \cdots, (x_N, y_N)\}$ *denote the training dataset that consists of* N *input batches. Besides,* $\mathcal{L}(\cdot)$ *is the training loss function and* $F(\cdot)$ *is the output of a possible architecture. After training the architecture for* $E > 1$ *epochs, we accumulate the batchwise training losses to quantify the*

Algorithm 2 Channel-Level Explorations with BTE

Input: latency constraint T, training dataset $\mathcal{D}_{train}$, population size P, number of iterations Q, number of training epochs E;
Output: channel configuration with highest BTE score;

$P_0 \leftarrow Initialize\_Population(P, T)$ and $Topk \leftarrow \emptyset$;
$r \leftarrow 0.5$; ▷ mutation and crossover rate
for $i = 1$ **to** Q **do**
    $Score_{i-1} \leftarrow BTE(P_{i-1}, \mathcal{D}_{train}, E)$;
    $Topk \leftarrow Update\_Topk(Topk, P_{i-1}, Score_{i-1})$;
    $P_{crossover} \leftarrow Crossover(Topk, T, r)$;
    $P_{mutation} \leftarrow Mutation(Topk, T, r)$;
    $P_i \leftarrow P_{crossover} \cup P_{mutation} \cup Topk$;
end for
Derive the solution c with highest score in $Topk$;
Return the channel configuration c;

architecture using Batchwise Training Estimation (BTE) as follows:

$$BTE = \sum_{n=1}^{N} \left[\prod_{e=1}^{E-1} \left(\frac{\mathcal{L}(F_e(x_n), y_n) - \mathcal{L}(F_{e+1}(x_n), y_n)}{\mathcal{L}(F_e(x_n), y_n)} \right) \right]$$

where $F_e(\cdot)$ is the architecture at the training epoch of e.

Note that, different from the conventional training paradigm, we do not shuffle or re-shuffle the training dataset throughout the training process in order to collect the batchwise training statistics. In addition, the random data augmentations are excluded as well. The intuition behind the above is to ensure that the input batches are consistent across different training epochs in terms of both input order and input content. Otherwise, the training statistics will be accumulated according to different input batches, which further introduces random noises in the context of BTE, and consequently, degrades the correlation performance of BTE.

Subsequently, we conduct a series of experiments on NAS-Bench-201 [16] to show the efficacy of BTE. To collect batchwise training statistics, we sample 1,004 random architectures from NAS-Bench-201 [16], which are equally trained from scratch on CIFAR10 for 7 epochs. The 7 epochs here are for demonstration only, in which we show that the correlation performance of BTE can be improved with additional training budgets. In this work, the default training budgets are 3 epochs. After that, we calculate the architecture score using BTE. As shown in Figure 4.6 (*Top*), the normalized architecture score strongly correlates to the accuracy on CIFAR10, CIFAR100, and ImageNet, which indicates that BTE is a reliable proxy

to reflect the accuracy of possible architectures across different datasets, and more importantly, at near-zero cost. Meanwhile, the rank correlation performance of BTE consistently outperforms TSE [15] (see Figure 4.6 (*Bottom*)) under the same training budgets. Besides, we compare BTE with zero-cost proxies, for example, Jacobian [17, 18]. As shown in Figure 4.7, Jacobian, despite its cost efficiency, suffers from degraded rank correlation performance compared with BTE and TSE. And even worse, unlike BTE and TSE, the rank correlation performance of Jacobian cannot be improved when additional training budgets are allocated. These experimental results clearly demonstrate the superiority of BTE over TSE and Jacobian.

Furthermore, we leverage BTE to enable the channel-level explorations at low cost, in which we aim to find the channel configuration that leads to the highest architecture score in terms of BTE. The objective is described as follows:

$$\text{maximize } BTE(\alpha_c),\ s.t.,\ LAT(\alpha_c) = T \text{ and } c \in \mathcal{C} \tag{4.13}$$

where $\mathcal{C}$ denotes the channel space. α_c is the searched LightNet α under the channel configuration c. In this work, we allow a set of channel scales, varying from 0.5 to 2.0 with an interval of 0.25. As such, we derive $|\mathcal{C}| = 7^{21} \approx 5.6 \times 10^{17}$. To this end, we further construct a unique latency predictor for each LightNet to avoid the tedious on-device measurements, in which the implementation details are the same as the latency predictor described in Section 4.3.2. Specifically, for each LightNet, we first generate 10,000 architectures under random channel configurations. Next, we measure the latency of the above-sampled architectures on Nvidia Jetson AGX Xavier. After that, we train the latency predictor using the above architectures and the corresponding latency measurements. It is worth noting that the design cost of each latency predictor is about one hour as mentioned in Section 4.3.2. The latency prediction results are illustrated in Figure 4.8, which indicates that the proposed latency predictor can accurately approximate the latency of different architecture candidates not only at the macro level but also at the micro level.

We next introduce an evolutionary algorithm (EA) to solve the optimization problem described in Eq (4.13). In practice, we set the number of training epochs E (see Definition 4.1) to 3 by default, which strikes a balance between the correlation performance and the training budgets. Meanwhile, as illustrated in Figure 4.7, BTE is able to reliably reflect the performance of the given architecture candidate in 3

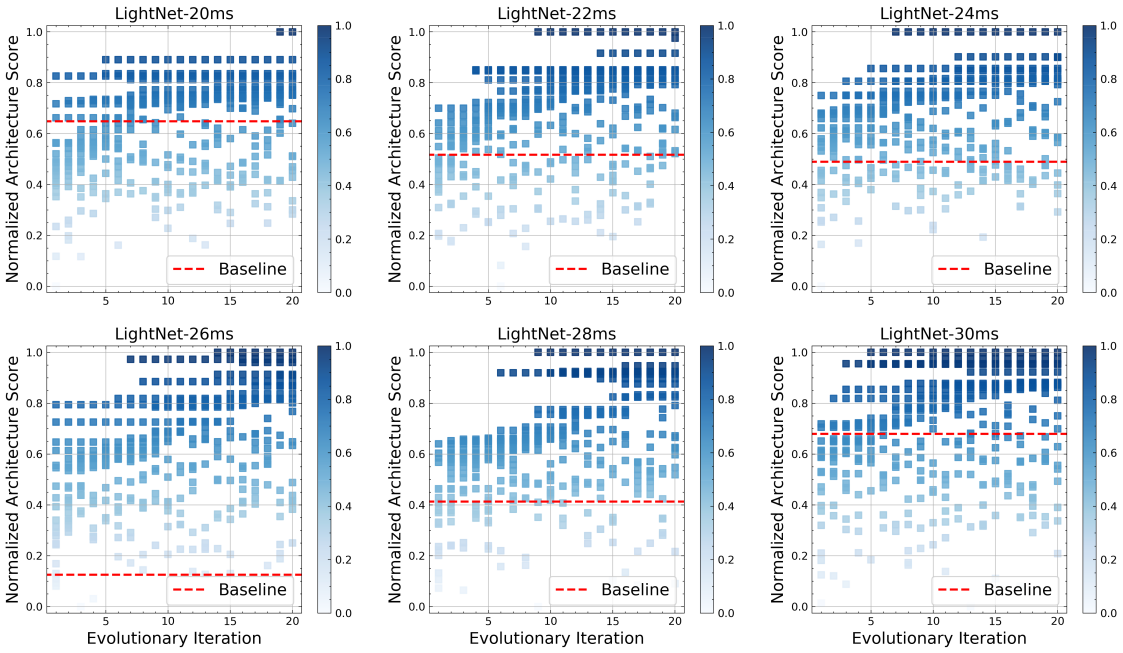

FIGURE 4.9: Visualization of the evolutionary exploration of different LightNets under different channel configurations using BTE as described in Definition 4.1.

epochs. The evolutionary algorithm is summarized in Algorithm 2. Specifically, we set the number of iterations to 20 and the population size to 20 by default. During each search iteration, we reserve the top 10 candidates, and then repeatedly apply crossover and mutation to produce enough new candidates under the specified latency constraint. As visualized in Figure 4.9, the evolutionary process can effectively end up with the channel configuration that brings a higher architecture score than the baseline counterpart, while still satisfying the required latency constraint. Recall that BTE is able to evaluate one architecture candidate at near-zero cost. During the evolutionary process, there are hundreds of architecture candidates to be evaluated using BTE. But even so, the required computation resources are still much less than previous methods [127, 145, 150], i.e., at low cost. In particular, we only need to train the given architecture for several epochs using BTE, which thus exhibits significant computational efficiency compared with [127, 145, 150].

4.3.6 Relationships with Previous Methods

First and foremost, to obtain the architecture that strictly satisfies the specified latency constraint, previous hardware-aware differentiable NAS methods [9, 13, 59,

TABLE 4.1: Comparisons with previous state-of-the-art NAS methods. ‡ The reported search cost here is directly taken from the respective paper, which is also converted into GPU hours for fair comparisons.

	[3]	[5]	[14]	[9]	[13]	Ours
Differentiable	✓	✗	✗	✓	✓	✓
Latency Optimization	✗	✓	✓	✓	✓	✓
Specified Latency	✗	✓	✓	✗	✗	✓
Proxyless Search	✗	✓	✓	✓	✓	✓
Search Complexity	$\mathcal{O}(K^2)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(K^2)$	$\mathcal{O}(2^2)$	$\mathcal{O}(1)$
Search Cost†	24	40,000	1,275	216	200	10

60, 144] require to perform a hyper-parameter sweep to manually tune the trade-off coefficient λ by trial and error as discussed in Section 4.2.2. Therefore, the total design cost increases proportionally (empirically by $\times 10$ times). We emphasize that previous relevant NAS methods only report the explicit search cost such as the time required for one single search, whereas the huge implicit search cost like the time required for manual hyper-parameter tuning is excluded [51]. In contrast, we, in this paper, focus on finding the required architecture through a one-time search such that the huge implicit search cost is eliminated, thereby bringing significant search efficiency and flexibility as well. More importantly, the proposed LightNAS is able to find the required architecture in a fully differentiable manner, which is fundamentally different from [97, 153] that include human heuristics to satisfy the given latency constraint. For instance, [97] manually shrinks and extends the network width during the search process and [153] manually projects the searched architecture to a discrete space at the end of search. Apart from these, reinforcement learning and evolution-based NAS approaches [5, 14, 154] can achieve the same goal as LightNAS, but suffer from prohibitive search overheads (e.g., 40,000 GPU hours in [5] and 10 GPU hours in LightNAS). Meanwhile, in the NAS literature, [101] strives to train an accurate latency predictor with only few latency measurements via meta learning, which, however, does not introduce new materials from the NAS perspective.

Furthermore, due to the multi-path optimization paradigm, previous differentiable NAS methods [3, 9, 20] suffer from the memory bottleneck during the search process, which severely violate the equality principle as defined in [22]. To alleviate the memory bottleneck, we introduce a lightweight differentiable architecture search approach based on the Gumbel Softmax reparameterization strategy [9, 20, 21],

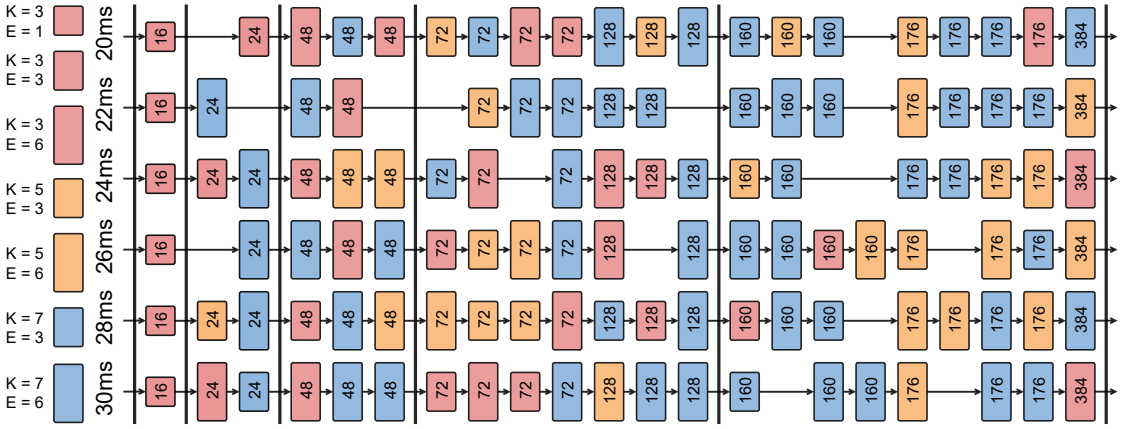

FIGURE 4.10: Visualization of the searched LightNets under different latency requirements ranging from 20 ms to 30 ms with an interval of 2 ms. Note that the number in each operator is the number of base channels.

which effectively reduces the optimization complexity to the single-path level. Note that, the single-path mechanism in LightNAS is fundamentally different from [59], which formulates different operator candidates into a single-path superkernel. More importantly, owing to the single-path mechanism, the optimization gap between the supernet and the searched sub-network is bridged, thereby improving the accuracy of the searched architecture [22]. Besides, different from the latency lookup table (LUT) [9, 14], the predictor proposed in Section 4.3.2 can accurately approximate not only the on-device latency but also other hardware performance metrics like the on-device energy as shown in Figure 4.15. As such, without loss of generality, LightNAS can be effortlessly plugged into various real-world scenarios, in which we only need to replace the latency predictor with the predictor of the target scenario. To highlight the technical merits, we further compare the proposed LightNAS against previous relevant NAS methods in Table 4.1.

Finally, we discuss BTE, which enables the channel-level exploration of LightNets at low cost. In the literature, several relevant methods [127, 145, 150] also focus on automatically finding the channel configuration that maximizes the accuracy without sacrificing the inference efficiency. Nonetheless, these methods [127, 145, 150] require a huge amount of computation resources, especially to train a slimmable network or supernet that can run under arbitrary channel configurations [148, 149]. To avoid the above-mentioned computation overheads, instead of training a slimmable network or supernet, another recent research direction is devoted to approximating the accuracy of different architectures at initialization using

zero-cost proxies [17, 18], which, however, suffer from sub-optimal rank correlation performance when compared with the state of the art [15], and more importantly, cannot be improved with additional training budgets. To tackle this issue, [15] introduces an effective proxy dubbed TSE to reliably estimate the training accuracy at near-zero cost, which empirically accumulates the training loss statistics in the early training epochs. Despite its promising empirical results, TSE ignores the training dynamics in the very early training epochs and thus suffers from sub-optimal rank correlation performance. In contrast, the proposed BTE proxy considers the training dynamics and turns back to the relative training improvement of the same training batches across different training epochs. Extensive empirical results on NAS-Bench-201 also clearly demonstrate that BTE can achieve more reliable rank correlation performance on three different datasets than the aforementioned proxies. Apart from the above, several recent mask-based NAS methods [155–159] instead focus on searching for the optimal channel configuration in a fully differentiable manner using binary masks, which are of low computational cost since the search process can be optimized with gradient descent. However, the above mask-based NAS methods use simple cost metrics (e.g., the number of FLOPs and parameters) as guidelines, which cannot reflect the on-device performance on target hardware. Meanwhile, to satisfy the specified latency constraint, the above mask-based NAS methods have to repeat a plethora of search experiments to tune the trade-off coefficient λ by trial and error, thereby significantly increasing the total design cost.

Remarks. In this work, we mainly focus on exploring computation-efficient convolutional networks for one popular embedded platform. Furthermore, the proposed LightNAS can also be easily generalized to cover evolving neural network structures and changing hardware platforms. To cover evolving neural network structures (e.g., transformers [160] and vision transformers [161]), we only need to re-design the search space to replace convolutional operators with novel encoder/decoder/attention structures of transformers and vision transformers. To cover changing hardware platforms, we only need to re-build an accurate hardware-aware performance predictor for reliable performance prediction on target hardware.

TABLE 4.2: Comparisons with previous state-of-the-art architectures on ImageNet [1]. † corresponds to architectures utilizing extra techniques like Swish activation and Squeeze-and-Excitation (SE) module [7, 23]. ‡ FBNet-Xavier is the architecture searched on Nvidia Jetson AGX Xavier using FBNet [9].

Architecture	Method	Search Cost (GPU hours)	Accuracy (%)		Latency (ms)
			Top-1	Top-5	
MobileNetV2 [6]	Manual	-	72.0	91.0	20.2
ProxylessNAS [13]	Differentiable	200	74.6	92.2	21.2
FBNet-A [9]	Differentiable	216	73.0	90.9	21.7
OFA-S [14]	Evolution	1,275	72.9	91.1	21.4
MnasNet-B1 [5]	Reinforcement	40,000	74.5	92.1	20.1
LightNet-20ms	Differentiable	10	75.0	92.2	20.0
LightNet++-20ms	Evolution	20	75.4	92.6	20.1
FBNet-B [9]	Differentiable	216	74.1	91.8	23.0
MobileNetV3† [7]	Reinforcement	40,000	75.2	-	23.0
MnasNet-A1† [5]	Reinforcement	40,000	75.2	92.5	22.9
SPNet [59]	Differentiable	~ 30	75.0	92.2	23.6
LightNet-22ms	Differentiable	10	75.2	92.2	22.1
LightNet++-22ms	Evolution	22	75.6	92.7	22.0
ProxylessNAS [13]	Differentiable	200	75.1	92.5	24.5
UNAS [144]	Differentiable	103	75.3	92.4	24.2
FBNet-Xavier‡ [9]	Differentiable	186	74.6	92.1	24.1
LightNet-24ms	Differentiable	10	75.5	92.3	23.9
LightNet++-24ms	Evolution	23	76.2	93.1	24.1
FBNet-C [9]	Differentiable	216	74.9	92.3	26.4
OFA-M [14]	Evolution	1,275	75.4	92.4	26.3
LightNet-26ms	Differentiable	10	75.9	92.6	26.1
LightNet++-26ms	Evolution	25	76.4	92.9	26.0
OFA-L [14]	Evolution	1,275	75.8	92.7	29.3
ProxylessNAS [13]	Differentiable	200	75.3	-	29.9
LightNet-28ms	Differentiable	10	76.1	92.7	28.2
LightNet++-28ms	Evolution	27	76.5	93.1	28.1
TFNAS [97]	Differentiable	43	76.0	-	33.6
EfficientNet-B0† [134]	Reinforcement	40,000	76.3	-	37.2
LightNet-30ms	Differentiable	10	76.4	92.9	30.1
LightNet++-30ms	Evolution	27	76.7	93.3	30.2

4.4 Experiments

In this section, we conduct extensive experiments to evaluate the proposed approach on a cutting-edge embedded platform, namely Nvidia Jetson AGX Xavier. The MAXN power mode is applied by default to maximize the hardware performance. Following [13], all the measurements are reported with an input batch size of 8 [13] to avoid resource underutilization. Please note that all the experiments in this work are conducted on GeForce RTX 3090 GPUs using Python 3.7.10, PyTorch 1.7.0, CUDA 11.2, and cuDNN 8003.

TABLE 4.3: Illustration of the experimental results on NAS-Bench-201 [16] and HW-NAS-Bench [24]. Please note that, except for LightNAS, other experimental results in this table are directly taken from EH-DNAS [25].

Approach		Raspberry Pi 4			
Search Algorithm	Hardware Feedback	Accuracy↑ (%)	Latency↓ (ms)	FLOPs↓ (M)	Params↓ (M)
DARTS [3]	None	54.30	3.84	7.78	0.073
	FBNet LUT [9]	70.92	2.96	7.78	0.073
	EH-DNAS [25]	70.92	2.96	7.78	0.073
	LightNAS (1-st)	86.66	2.55	7.78	0.073
	LightNAS (2-nd)	84.19	2.89	7.78	0.073
	LightNAS (3-rd)	84.16	2.84	7.78	0.073
GDAS [21]	None	93.37	69.19	153.27	1.073
	FBNet LUT [9]	10.00	0.01	47.10	0.344
	EH-DNAS [25]	93.58	56.89	121.82	0.858
	LightNAS (1-st)	93.53	44.52	90.36	0.643
	LightNAS (2-nd)	93.49	49.45	90.36	0.643
	LightNAS (3-rd)	93.53	44.52	90.36	0.643

4.4.1 Dataset and Implementation Details

Dataset. All the experiments in this work are directly conducted on a popular large-scale dataset named ImageNet [1]. Specifically, ImageNet consists of 1,000 categories that have 1.28M training images and 50K validation images, all of which are roughly equally distributed across all categories. Following recent network design conventions [5, 6], we use the mobile setting throughout this work, where the image size is set to 224×224 and the number of multi-add operations is strictly under 600M during the runtime inference.

Architecture Search Settings. In this work, the architecture search settings closely follow FBNet [9]. Specifically, we randomly sample 100 categories from ImageNet, which are used to optimize both network weights w and architecture parameters α. Meanwhile, we train the stochastic supernet for 90 epochs with an input batch size of 128. In the first 10 epochs, only w is optimized, whereas α is frozen [9]. Subsequently, the optimization steps of w and α alternate in each epoch. To optimize w, we apply the SGD optimizer with a learning rate of 0.1 (annealed down to zero following the cosine schedule), a momentum of 0.9, and a weight decay of 3×10^{-5}. To optimize α, we employ the Adam optimizer [3] with a learning rate of 0.001 and a weight decay of 1×10^{-3}. Besides, as discussed

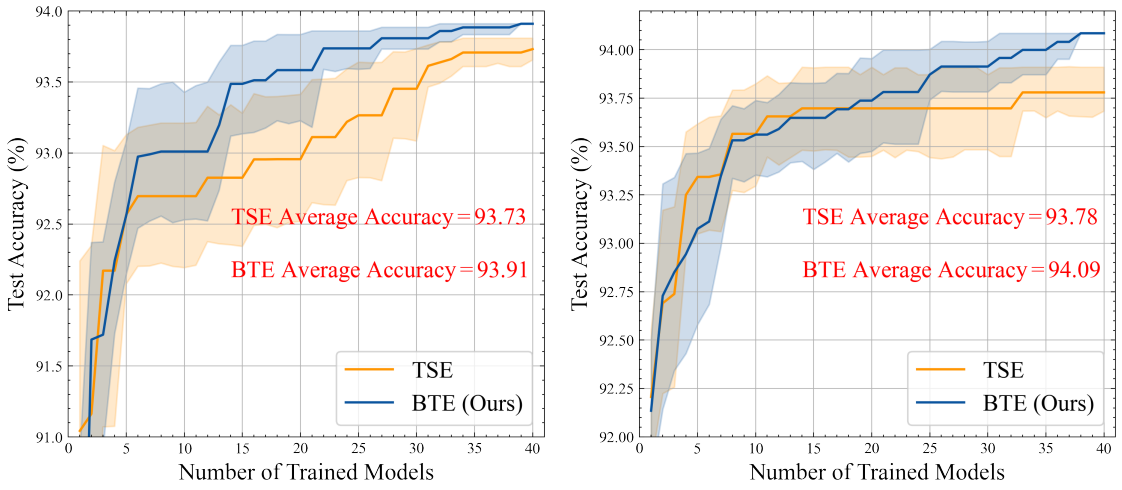

FIGURE 4.11: Visualization of the search process with TSE [15] abd BTE under the same training budget of thee training epochs, in which the required latency constraints are set to 35 ms (*Left*) and 45 ms (*Right*), respectively.

in Section 4.3.4, the trade-off coefficient λ in LightNAS is not a constant but a learnable parameter to be optimized during the search process. To this end, we initialize λ as zero and optimize λ with the gradient ascent scheme as illustrated in Eq (4.11), where the learning rate is set to 0.0005. We denote the architectures searched by LightNAS as LightNets. Please note that all the architecture search experiments are conducted on one single GeForce RTX 3090 GPU. Finally, as described in Section 4.3.5, we further perform channel-level explorations in terms of the searched LightNets, which are denoted as LightNets++.

Architecture Evaluation Settings. In this work, we simply follow the training protocols as widely used in previous NAS methods [9, 13] to evaluate the searched LightNets and LightNets++ on ImageNet [1]. Specifically, we retrain LightNets and LightNets++ from scratch for 360 epochs with a batch size of 1024 on 4 GeForce RTX 3090 GPUs, in which standard data augmentations are used by default [9, 13]. The optimizer is SGD with a momentum of 0.9 and a weight decay of 4×10^{-5}. Besides, the learning rate is initialized as 0.5, which gradually decays to zero following the cosine schedule. In the meantime, following [3, 9, 13], we linearly warm up the initial learning rate from 0.1 to 0.5 in the first 5 epochs. Apart from the above, we insert the Dropout module before the final classification layer, where the dropout ratio is fixed to 0.2 [9]. We note that the above settings closely follow recent practices in the field of NAS. And to avoid unfair comparisons with previous relevant methods, we do not apply any advanced training techniques,

either stronger data augmentations or better training protocols, which have the potential to further boost the attainable accuracy as shown in [134, 153].

4.4.2 Experimental Results

Architecture Search Results. We visualize the searched LightNets under different latency constraints in Figure 4.10. Different from MobileNetV2 [6] that simply stacks the same operator across the entire network, LightNAS is able to enable the layer diversity to strike the right balance between accuracy and latency. In particular, given a larger latency constraint, the search engine encourages to search for the required architecture that goes deeper and wider.

Architecture Evaluation Results. Results and comparisons with previous state-of-the-art architectures are summarized in Table 4.2[6]. We observe that the searched LightNets strictly satisfy the given latency constraints, while at the same time coming at an extremely low search cost of 10 GPU hours. Please note that the search cost in Table 4.2 only represents the cost of the search process on GPUs. Following previous relevant NAS methods [9, 13], the design cost of the latency predictor (i.e., about 1 hour) is not included into the search cost due to the unit inconsistency. However, even though the design cost of the latency predictor is directly included into the search cost, the search cost of LightNets just slightly increases from 10 GPU-hours to 11 GPU-hours and vice versa for LightNets++. More importantly, LightNets are obtained through a one-time search, and thus the manual hyper-parameter tuning process over the trade-off coefficient λ is eliminated. Meanwhile, under the same latency constraints, LightNets outperform previous state-of-the-art architectures in terms of the accuracy on ImageNet [1]. Furthermore, after the channel-level exploration, LightNets++ can achieve better accuracy on ImageNet than LightNets, while maintaining similar latency. Besides, considering that FBNet [9] as the baseline of this work is the most relevant NAS method in the literature, we further implement FBNet and exploit FBNet to perform the search experiment on Nvidia Jetson AGX Xavier, in which the searched architecture is denoted as FBNet-Xavier. After that, we train FBNet-Xavier from

[6]Following the conventions in the field of NAS, the search cost and the accuracy in Table 4.2 are taken from the corresponding papers, whereas the latency measurements are measured on Nvidia Jetson AGX Xavier under the same settings as LightNAS. Note that the reported search cost in Table 4.2 may be measured on different GPUs.

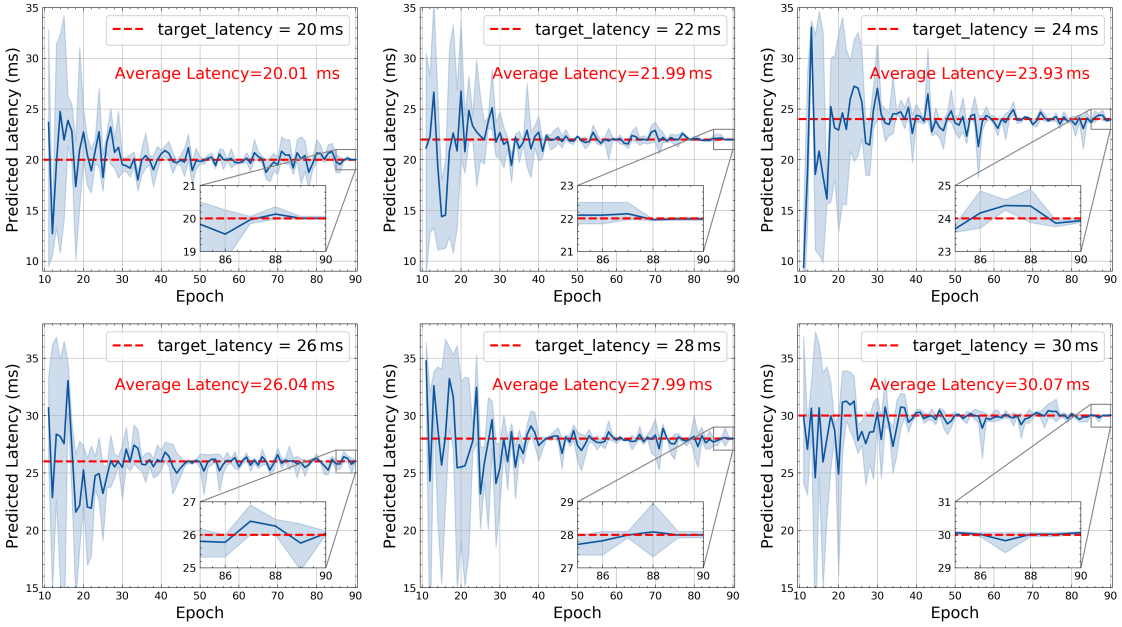

FIGURE 4.12: Visualization of the search process of LightNets under different latency constraints ranging from 20 ms to 30 ms with an interval of 2 ms.

scratch under the same settings as LightNets. As shown in Table 4.2, LightNet-24ms and LightNet++-24ms achieve +0.9% and 1.6% higher top-1 accuracy on ImageNet than FBNet-Xavier under the latency constraint of 24 ms.

4.4.3 Ablation Studies and Discussions

Ablation of LightNAS on NAS-Bench-201. In this section, we closely follow EH-DNAS [25] to evaluate the proposed LightNAS on CIFAR10 in the cell-based search space as defined in NAS-Bench-201 [16]. To achieve this, we incorporate LightNAS into two popular cell-based differentiable NAS algorithms, namely DARTS [3] and GDAS [21]. Specifically, we first collect the latency measurements on Raspberry Pi 4 using HW-NAS-Bench [24], which are then used to construct a precise latency predictor as described in Section 4.3.2. After that, following EH-DNAS, we integrate the above latency predictor into DARTS and GDAS. For example, after integrating the latency predictor into DARTS, the accuracy objective in Eq (4.10) corresponds to the search algorithm in DARTS, while the latency objective in Eq (4.10) represents the hardware-aware architecture search method in LightNAS. To compare with EH-DNAS, we set the target latency to 3 ms and 47 ms for DARTS and GDAS, respectively. The quantitative results on NAS-Bench-201

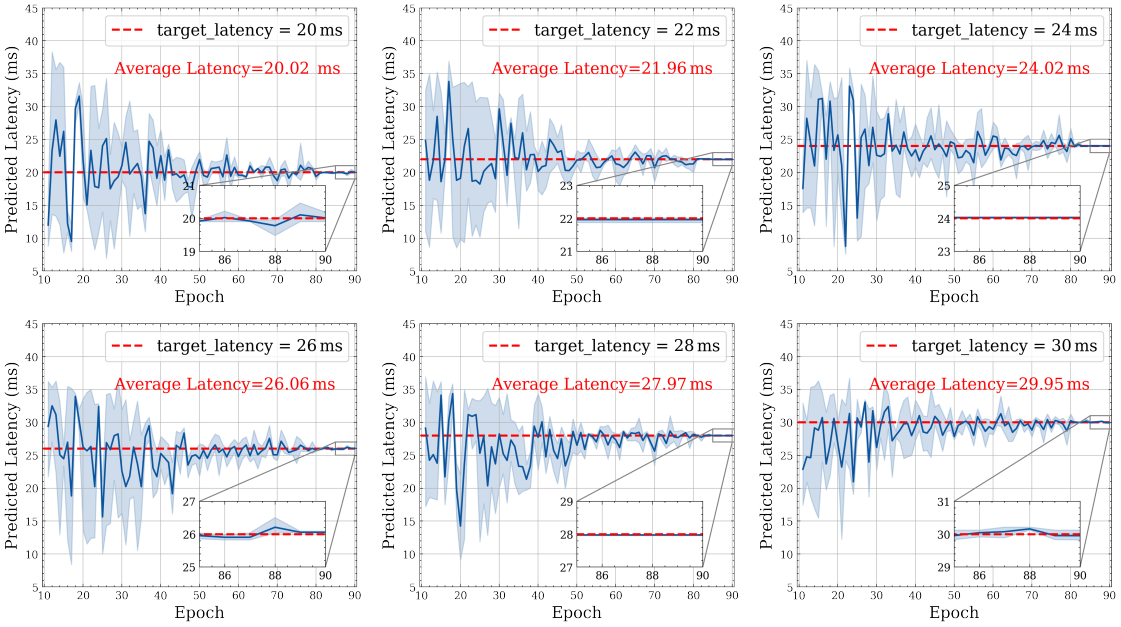

FIGURE 4.13: Visualization of the search process of LightNets-Worse under various latency constraints ranging from 20 ms to 30 ms with an interval of 2 ms.

and HW-NAS-Bench are summarized in Table 4.3, in which three search runs are reported [16]. In particular, LightNAS is able to achieve +15.74% higher accuracy than EH-DNAS, while at the same time being ×1.16 faster on Raspberry Pi 4. Among them, GDAS + FBNet-LUT achieves extremely low top-1 accuracy on CIFAR10, which is caused by the over-selection of *SkipConnect*.

Ablation of BTE on NAS-Bench-201. In this section, we benchmark BTE against TSE [15] on NAS-Bench-201 [16] and HW-NAS-Bench [24], in which we take a popular query-based NAS algorithm [162] as the search engine. And we choose Raspberry Pi 4 as the default embedded hardware. To ensure fair comparisons, we set the training budgets to 3 epochs for both BTE and TSE. In practice, we first sample 40 random architectures from NAS-Bench-201 under the specified latency constraint on HW-NAS-Bench. After that, we exploit BTE to evaluate the above-sampled architectures and reserve the architecture with the highest BTE score during each search iteration. To ensure fair comparisons, we here repeat ten different search experiments under the latency constraint of 35 ms and 45 ms, respectively. As illustrated in Figure 4.11, under the same latency constraint, the search engine, integrated with BTE, can consistently end up with the architecture that achieves higher accuracy on CIFAR10 than TSE. Besides, we also observe that the accuracy gap between BTE and TSE is smaller when the required latency

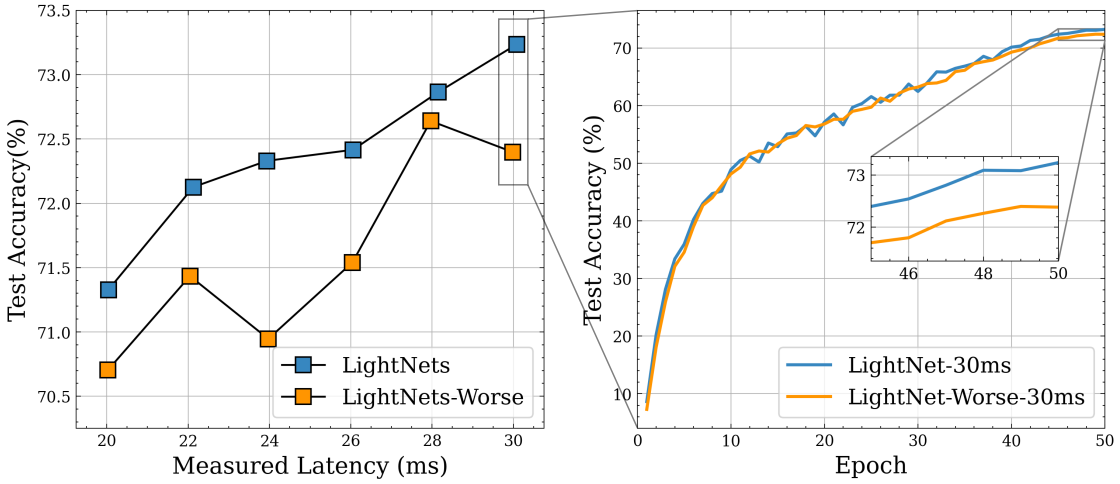

FIGURE 4.14: Comparisons between LightNets and LightNets-Worse under different latency constraints ranging from 20 ms to 30 ms with an interval of 2 ms. Note that all the architectures here are trained from scratch for 50 epochs.

constraint is tight at 35 ms and larger when the required latency constraint is loose at 45 ms. The rationale here is that the search space of NAS-Bench-201 consists of more possible architecture candidates around 45 ± 1 ms than 35 ± 1 ms. This may further enlarge the accuracy gap between BTE and TSE since BTE can deliver more reliable rank correlation performance than TSE. The above experimental results clearly demonstrate the superiority of BTE over TSE.

Architecture Search Stability. We demonstrate the search process of LightNAS under different latency constraints in Figure 4.12, which range from 20 ms to 30 ms with an interval of 2 ms. Specifically, each figure in Figure 4.12 is visualized by averaging three different search runs in order to show the search stability. As shown in Figure 4.12, the proposed LightNAS consistently ends up with the required architecture that satisfies the specified latency constraint, which explicitly aligns with the analysis presented in Section 4.3.4. Furthermore, we also observe that LightNAS focuses on exploring possible architecture candidates around the specified latency constraint during the search process, which correspond to a smaller subset of the entire search space. This makes it easier to navigate better architecture candidates towards enhanced accuracy-efficiency trade-offs.

Ablation of the Accuracy Objective in Eq (4.10). In this section, we investigate the accuracy objective in Eq (4.10), in which we remove the accuracy objective from Eq (4.10). Apart from this, other settings remain the same as

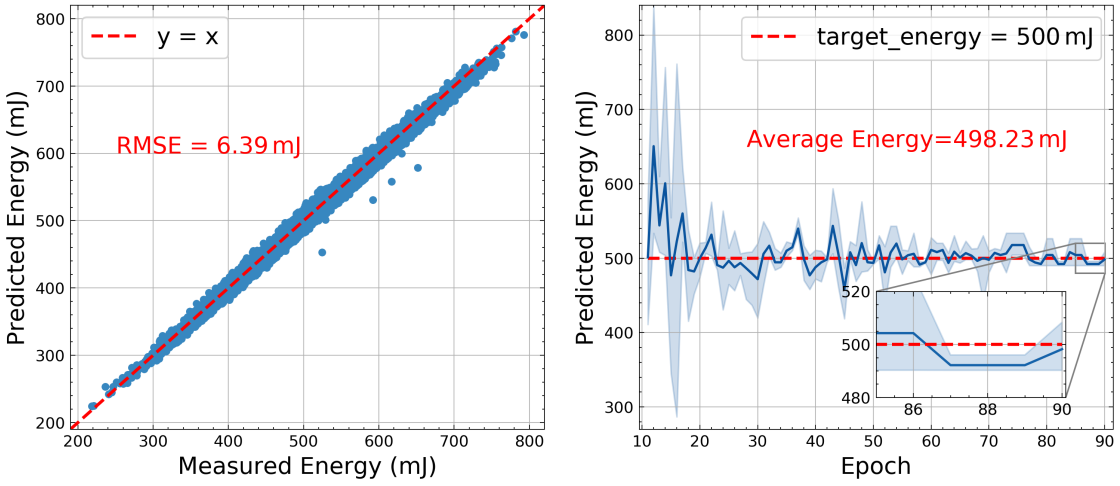

FIGURE 4.15: Illustration of the generality to energy-critical scenarios.

those described in Section 4.4.1. Likewise, we conduct a series of architecture search experiments under various latency constraints ranging from 20 ms to 30 ms. The searched architectures are denoted as LightNets-Worse. After that, we train LightNets-Worse from scratch for 50 epochs under the same settings as LightNets. As illustrated in Figure 4.13, LightNets-Worse still satisfy the specified latency constraints, which again shows the effectiveness of the latency objective in Eq (4.10). Meanwhile, as illustrated in Figure 4.14, LightNets achieve better accuracy on ImageNet than LightNets-Worse, which shows the effectiveness of the accuracy objective in Eq (4.10). Given the above, we conclude that the two objectives in Eq (4.10) are both necessary to search for the architecture with competitive accuracy while strictly satisfying the required latency constraint.

Generality to Energy-Critical Tasks. As mentioned in Section 4.3.6, the proposed LightNAS can be easily extended to various real-world scenarios, in which we only need to replace the latency predictor with the predictor of the target scenario. To further demonstrate this, we take the energy consumption as an example and generalize LightNAS to the energy-critical scenario. To this end, we first exploit the MLP-based predictor proposed in Section 4.3.2 to approximate the energy consumption for architectures in the search space. The energy prediction results are illustrated in Figure 4.15 (*Left*). Please note that, different from the latency measurement, the energy measurement suffers from inevitable noises caused by the hardware temperature. Furthermore, we integrate the above energy predictor

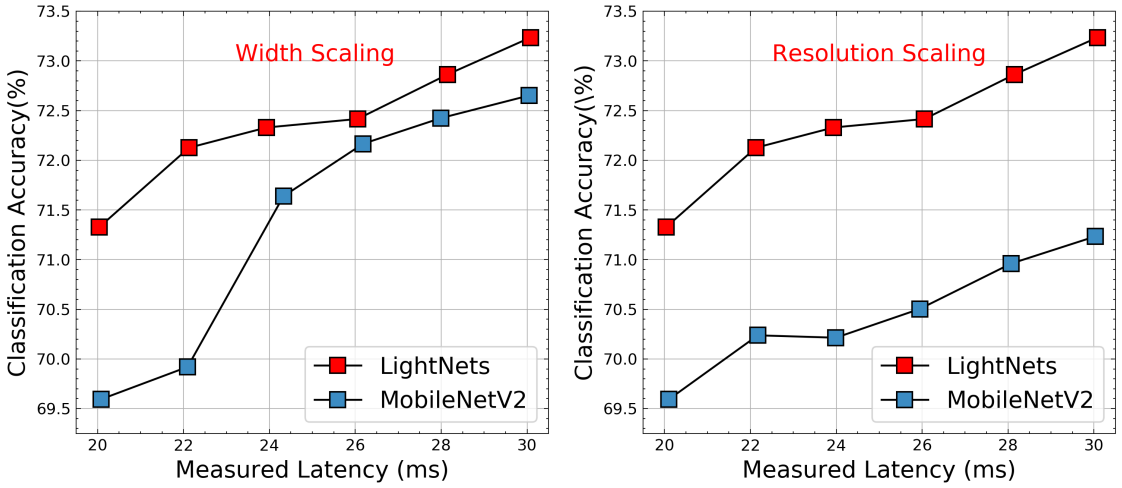

FIGURE 4.16: Comparisons with width (*Left*) and resolution scaling (*Right*) [5]. Note that all the networks here are trained from scratch for 50 epochs.

$Energy(\cdot)$ into the search optimization objective as follows:

$$\underset{\alpha}{\text{minimize}} \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) + \lambda \cdot \left(\frac{Energy(\overline{\alpha})}{E_{target}} - 1 \right) \tag{4.14}$$

where λ is the accuracy-energy trade-off coefficient and E_{target} is the specified energy constraint. Note that other search settings remain the same as Eq (4.10). Finally, we employ the above search optimization objective to search for the optimal architecture around the required energy constraint of 500 mJ. The resulting search process is visualized in Figure 4.15 (*Right*), in which we observe that LightNAS is able to effectively generalize to energy-critical embedded scenarios.

Transferability to Object Detection. We further evaluate the transferability of LightNets on object detection, which is much more challenging than image classification. To achieve this, we take a widely used object detection framework named SSDLite [163], in which we treat different architectures as drop-in replacements of backbone feature extractors [6]. For fair comparisons, we train all the architectures from scratch (i.e., without loading the pretrained weights on ImageNet) under the same experimental settings on COCO2017. As shown in Table 4.4, LightNets, despite being optimized in the context of image classification, can be transferred to object detection with better detection performance and execution efficiency than previous relevant architectures. Meanwhile, we note that the architecture that leads to better classification performance does not guarantee better detection performance. For example, as shown in Table 4.2, MnasNet-A1 achieves +0.3% higher top-1 accuracy on ImageNet than FBNet-C. However, when transferred to object

TABLE 4.4: Comparisons with previous relevant object detection backbones.

Backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L	Latency (ms)
ProxylessNAS [13]	20.3	34.6	20.3	2.2	19.3	39.6	70.1
MobileNetV2 [6]	20.4	34.3	20.5	1.6	19.5	40.2	72.6
MnasNet-A1 [5]	21.2	36.0	21.4	2.5	20.6	41.5	74.2
FBNet-C [9]	21.5	36.2	21.9	2.5	20.9	41.5	76.5
OFA-M [14]	21.6	36.7	21.9	2.2	21.4	41.3	75.4
LightNet-20ms	20.8	35.2	21.2	1.9	19.9	41.0	67.1
LightNet-24ms	21.5	36.3	21.7	2.5	21.2	42.2	68.6
LightNet-28ms	21.9	36.9	22.0	2.4	21.9	41.8	69.7

TABLE 4.5: Comparisons of LightNets and LightNets-SE on ImageNet.

	Architecture	Accuracy (%)		FLOPs (M)	Latency (ms)
		Top-1	Top-5		
With SE [23]	LightNet-20ms-SE	75.4 (+0.4)	92.3 (+0.1)	356 (+2)	20.9 (+0.9)
	LightNet-22ms-SE	76.1 (+0.9)	92.5 (+0.3)	352 (+3)	23.2 (+1.1)
	LightNet-24ms-SE	75.9 (+0.4)	92.6 (+0.3)	385 (+2)	25.5 (+1.6)
	LightNet-26ms-SE	76.3 (+0.4)	92.8 (+0.2)	435 (+3)	27.7 (+1.6)
	LightNet-28ms-SE	76.5 (+0.4)	92.8 (+0.1)	464 (+4)	30.3 (+2.1)
	LightNet-30ms-SE	77.0 (+0.6)	93.1 (+0.2)	493 (+4)	31.9 (+1.8)

detection, SSDLite + MnasNet-A1 even achieves -0.3% lower detection AP on COCO2017 than SSDLite + FBNet-C as illustrated in Table 4.4.

Comparisons with Scaling Techniques. As shown in previous relevant literature, another effective alternative to guarantee the specified latency requirement is the model scaling technique [5]. For example, we are allowed to gradually scale up or down the architecture until the specified latency requirement is fulfilled. Given that the search space of LightNAS is built upon MobileNetV2 [6], we further scale MobileNetV2 with respect to the network width and the input resolution, respectively, to accommodate different latency requirements. Note that the default input resolution is set to 224×224 when scaling up/down the network width and the default network width is set to 1.0 when scaling up/down the input resolution. The experimental results are illustrated in Figure 4.16. We observe that, under the same latency constraints, LightNets can consistently achieve better accuracy on ImageNet than the scaled MobileNetV2 variants.

Ablation of Squeeze-and-Excitation Module. Previous methods [5, 7, 134] use extra techniques like Squeeze-and-Excitation (SE) [23] to improve the performance as shown in Table 4.2. Therefore, for fair comparisons, we further conduct experiments to investigate the searched LightNets with the SE module, in which we

apply the SE module to the last nine layers of LightNets. The searched LightNets with the SE module are then denoted as LightNets-SE. As shown in Table 4.5, the SE module can greatly boost the attainable accuracy of LightNets while slightly sacrificing the inference efficiency on target hardware.

4.5 Conclusion

In this chapter, we propose, implement, and validate a novel lightweight and scalable hardware-aware NAS framework named LightNAS, which consists of two separate stages. In the first stage, LightNAS is able to find the architecture with competitive accuracy while strictly satisfying the required latency constraint in a differentiable manner, and more importantly, through a one-time search. After that, in the second stage, we introduce a reliable yet computationally cheap proxy named BTE, with which we explore the channel-level configurations of LightNets searched in the first stage at low cost. As a result, the accuracy of LightNets can be further improved without degrading the runtime latency on target hardware. Extensive experiments are conducted to demonstrate the superiority of LightNAS over previous relevant NAS approaches.

Chapter 5

Efficient Hardware-Aware Neural Architecture Compression[1]

The evolving complexity of convolutional neural networks (CNNs) has fueled an increasing demand for compression. However, pruning, as one of the most effective knobs, fails to deliver *Pareto-optimal* network solutions. To tackle this, we propose a first-of-its-kind pruning-free compression framework dubbed *Domino*, striving to revisit the trade-off dilemma between accuracy and efficiency from a fresh perspective of linearity and non-linearity. Specifically, *Domino* leverages two efficient predictors, including one vanilla latency predictor and one meta-accuracy predictor, to explore the less important non-linear building blocks, which are further grafted with the linear counterparts. The grafted network is then trained on target task to obtain superior accuracy, after which the grafted linear building block that consists of multiple consecutive linear layers is reparameterized into one single convolutional layer to boost the efficiency on target hardware without sacrificing the accuracy on target task. Extensive experiments on two popular NVIDIA Jetson embedded platforms (i.e., Xavier and Nano) and two representative deep convolutional networks (i.e., MobileNetV2 and ResNet50) explicitly demonstrate the efficacy of *Domino*. For example, *Domino*-Aggressive is able to achieve +10.6%/+8.8% higher top-1/top-5 accuracy on ImageNet than MobileNetV2×0.2, while at the same time maintaining ×1.9 and ×1.3 speedup on Xavier and Nano, respectively.

[1]This chapter has been published in: Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, Shiqing Li, Guochu Xiong, and Weichen Liu, "Pearls Hide Behind Linearity: Simplifying Deep Convolutional Networks for Embedded Hardware Systems via Linearity Grafting." ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.

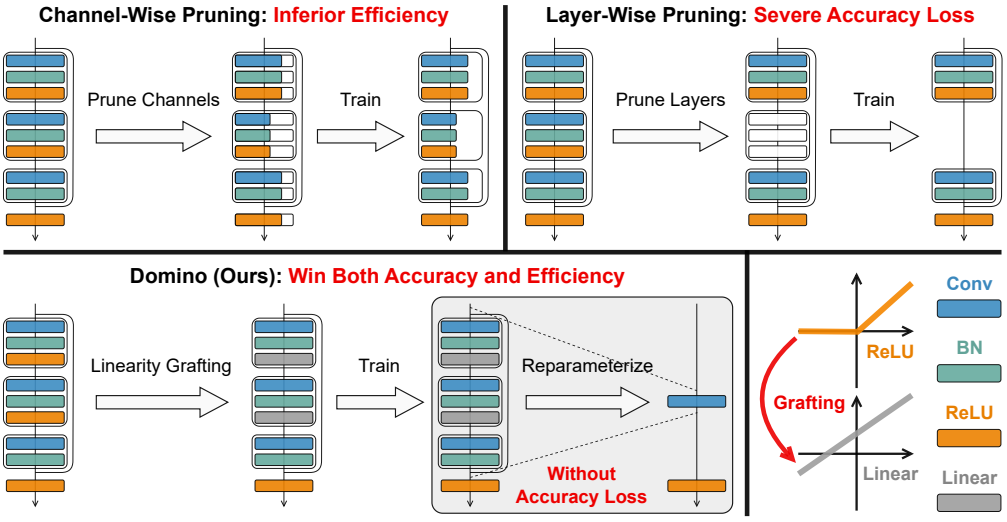

FIGURE 5.1: Comparisons between previous well-established pruning techniques (*top*) and *Domino* (*bottom*), where we consider the most representative residual building block in ResNet50 [19] as an example. Note that, in this figure, Conv, BN, and Linear are linear layers, whereas ReLU is non-linear.

The remainder of this chapter is organized as follows. Section 5.1 introduces the research background. Section 5.2 presents the observations and discusses the motivations. Section 5.3 elaborates on the proposed *Domino*. Section 5.4 presents the experimental settings and results. Finally, Section 5.5 concludes this chapter.

5.1 Introduction

With the increasing availability of large-scale datasets and advanced computing paradigms, convolutional neural networks (CNNs) have been empowering a myriad of intelligent embedded tasks, such as on-device object detection, recognition, and tracking [27, 36]. Over the past decade, "*deeper + wider = better*" has been deemed as the rule of thumb to design networks with competitive accuracy [6, 19]. This trend, despite its efficacy, leads to an exponential growth in the number of floating-point operations (a.k.a., FLOPs) and parameters as the network evolves deeper and wider [26]. For example, ResNet50 [19], as one of the most representative convolutional networks, involves over 4 billion FLOPs and 25 million parameters. The evolving network complexity further enlarges the computational gap between computation-intensive CNNs and resource-constrained embedded platforms [26].

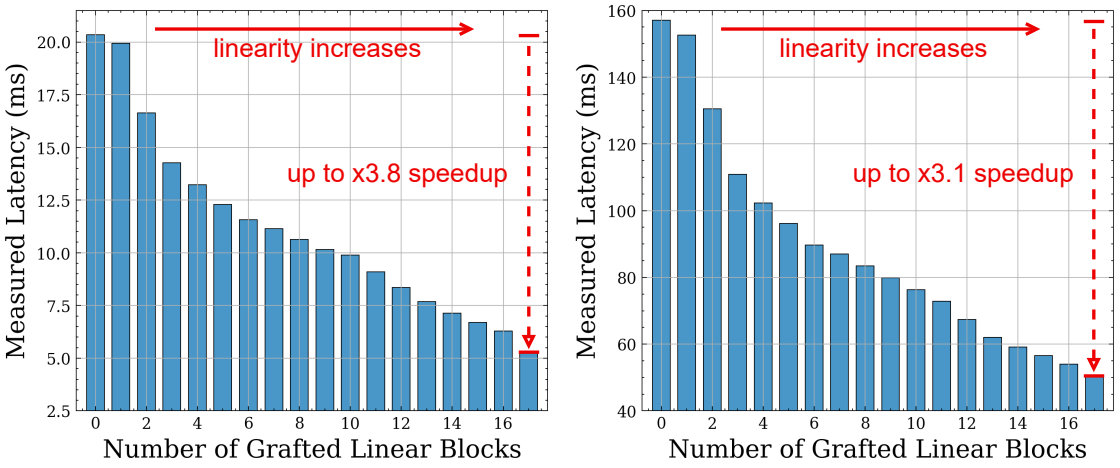

FIGURE 5.2: Relationships between the number of grafted linear building blocks G and the on-device latency measured on Xavier (*left*) and Nano (*right*).

To bridge the above computational gap, considerable effort has been dedicated to designing resource-efficient networks. Among them, (structured) pruning[2], including channel-wise and layer-wise pruning, has been deemed as the most dominant solution [27], which removes redundant channels and layers to trim down the network complexity [69, 70, 72, 126–128, 164–168]. However, previous channel-wise pruning methods [126–128, 164–168], despite being able to maintain superior accuracy on target task, suffer from inferior speedup on target hardware due to significant resource underutilization [70, 71]. In contrast, previous layer-wise pruning methods [69, 70, 72] indeed deliver promising speedup on target hardware, which, however, comes at the cost of severe accuracy loss on target task since the attainable accuracy relies on sufficient network depth [19]. These reveal that pruning, regardless of channel-wise and layer-wise pruning, cannot win both accuracy and efficiency, and as a result, fails to achieve *Pareto-optimal* accuracy-efficiency trade-offs. However, this is critical in those environments with rigorous resource constraints, such as resource-constrained embedded scenarios [27, 92, 100]. To tackle this issue, we turn back to the following counterintuitive question:

> *Can we simplify deep convolutional networks to enable Pareto-optimal accuracy-efficiency trade-offs without pruning any channels or layers?*

[2]In this work, we do not consider unstructured pruning (i.e., weight pruning), which highly relies on specialized hardware accelerators and thus cannot benefit mainstream embedded platforms due to the unstructured network sparsity and irregular memory access patterns [26, 27, 119].

To investigate the above question, we introduce a first-of-its-kind pruning-free compression framework dubbed *Domino*, striving to simplify deep convolutional networks without pruning any channels or layers. As illustrated in Figure 5.1, *Domino* revisits the accuracy-efficiency trade-off from a fresh perspective of linearity and non-linearity across different basic building blocks, which explicitly distinguishes itself from previous relevant methods [69, 70, 72, 126–128, 164–168] in the rich pruning literature. Note that, in this work, we compare *Domino* against pruning because both share the same goal of deriving simplified network structures as shown in Figure 5.1. Finally, we summarize our main contributions as follows:

- **Fundamentals.** *Domino* is the first to revisit the trade-off dilemma between accuracy and efficiency from the perspective of linearity and non-linearity, which pioneers a novel pruning-free alternative to simplify deep convolutional networks towards *Pareto-optimal* accuracy-efficiency trade-offs.
- **Framework.** *Domino* features two efficient predictors, including one vanilla latency predictor and one meta-accuracy predictor, to explore the less important non-linear building blocks, which are then grafted with the linear counterparts. The resulting grafted network is further trained on target task to obtain superior accuracy. Finally, we reparameterize each grafted linear building block that contains multiple consecutive linear layers into one single convolutional layer (see Figure 5.1 (*bottom*)) to largely boost the efficiency on target hardware without sacrificing the accuracy on target task.
- **Evaluations.** Extensive experiments are conducted to evaluate *Domino* on ImageNet with two popular NVIDIA Jetson embedded platforms (i.e., Xavier and Nano) and two representative convolutional networks (i.e., MobileNetV2 [6] and ResNet50 [19]), which clearly show that *Domino* can outperform uniform baselines and previous relevant methods in terms of both accuracy and hardware efficiency, especially under a large compression ratio. Taking MobileNetV2 as an example, *Domino*-Aggressive achieves +10.6%/+8.8% higher top-1/top-5 accuracy on ImageNet than MobileNetV2×0.2, while delivering ×1.9 and ×1.3 speedup on Xavier and Nano, respectively.

5.2 Observations and Motivations

Observation ① *Multiple consecutive linear layers can be effectively reparameterized into one single linear layer without changing the output.* Without loss of generality, we consider two consecutive linear layers as follows:

$$Y = W_1 X + B_1 \text{ and } Z = W_2 Y + B_2 \tag{5.1}$$

where X is the input data. Besides, W_1, W_2, B_1, and B_2 are the weight and bias parameters of the above two consecutive linear layers. Taken together, we can re-formulate the above two consecutive linear layers in Eq (5.1) as follows:

$$Z = W_2(W_1 X + B_1) + B_2 = (W_1 W_2)X + (W_2 B_1 + B_2) \tag{5.2}$$

This explicitly indicates that the above two consecutive linear layers can be reparameterized into one single linear layer with $W^* = W_1 W_2$ and $B^* = W_2 B_1 + B_2$, both of which maintain the same output Z. Note that Observation ① can be generalized to all common linear layers, including fully-connected (FC), batch normalization (BN), and convolutional (Conv) layers. For example, one convolutional layer and one subsequent BN or convolutional layer can be reparameterized into one single convolutional layer, leaving the output unchanged.

TABLE 5.1: Illustration of the layer-wise latency profiling results of MobileNetV2 [6] on NVIDIA Jetson AGX Xavier and NVIDIA Jetson Nano.

	Linear Layers			Non-Linear Layers
	Conv	BN	FC	ReLU6
Xavier	15.9 ms	2.9 ms	0.3 ms	1.3 ms (**6.4%**)
Nano	110.9 ms	35.6 ms	2.1 ms	8.4 ms (**5.4%**)

Observation ② *The linear layers dominate the latency on target hardware.* To demonstrate this, we profile MobileNetV2 on Xavier and Nano with an input batch size of 8, in which the inter-layer communication overheads and data movements are ignored for the sake of simplicity. The experimental results are shown in Table 5.1, in which we observe that the linear layers, including fully-connected (FC), batch normalization (BN), and convolutional (Conv) layers, account for more than 93% of the total latency on both Xavier and Nano. This also demonstrates that

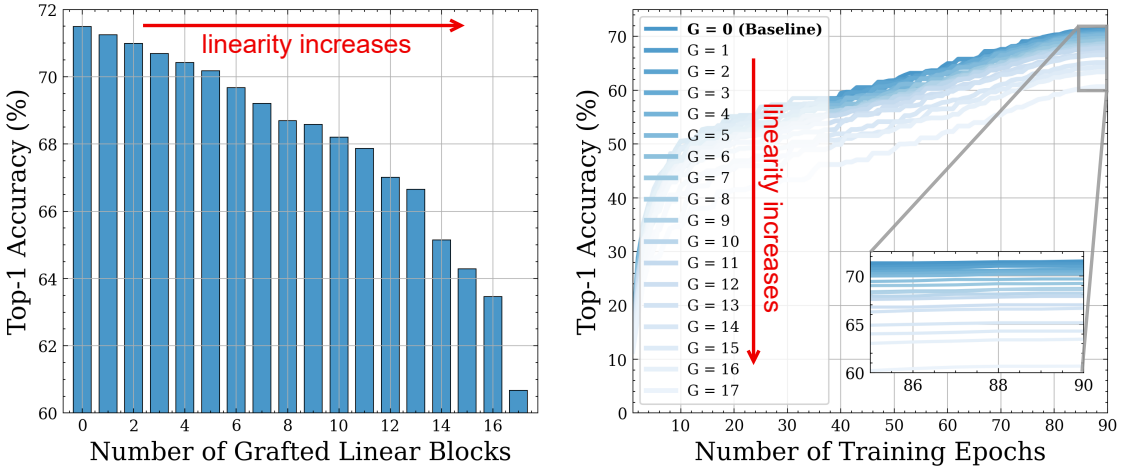

FIGURE 5.3: Relationships between the number of grafted linear building blocks G and the accuracy on ImageNet (*left*) and the training curves (*right*).

linear reparameterization, when properly engineered, has the potential to deliver significant speedup on target hardware without accuracy loss.

Observation ③ *The network linearity benefits the efficiency on target hardware.* To illustrate this, we take MobileNetV2 as an example. Specifically, we first progressively graft the non-linear building blocks with the linear counterparts (see Figure 5.1 (*bottom*)). Subsequently, to derive the simplified network, we, following Observation ①, reparameterize the grafted linear building block that consists of three convolutional layers, three BN layers, and two grafted linear activation layers into one single convolutional layer. Finally, we measure the latency of the resulting simplified network on Xavier and Nano with an input batch size of 8. As shown in Figure 5.2, the network linearity boosts the efficiency on target hardware, which delivers up to $\times$3.8 speedup on Xavier and $\times$3.1 speedup on Nano.

Observation ④ *The network non-linearity benefits the accuracy on target task.* Similar to Observation ③, we also leverage MobileNetV2 as the baseline network. Specifically, we (1) progressively graft the non-linear building blocks with the linear counterparts and (2) train the resulting grafted network from scratch on ImageNet for 90 epochs. Finally, we, following Observation ①, reparameterize the well-trained grafted network to derive the simplified network, where both networks exhibit the same accuracy on ImageNet since reparameterization does not change the network output according to Observation ①. As illustrated in Figure 5.3, the network non-linearity is critical to maintaining competitive accuracy on ImageNet,

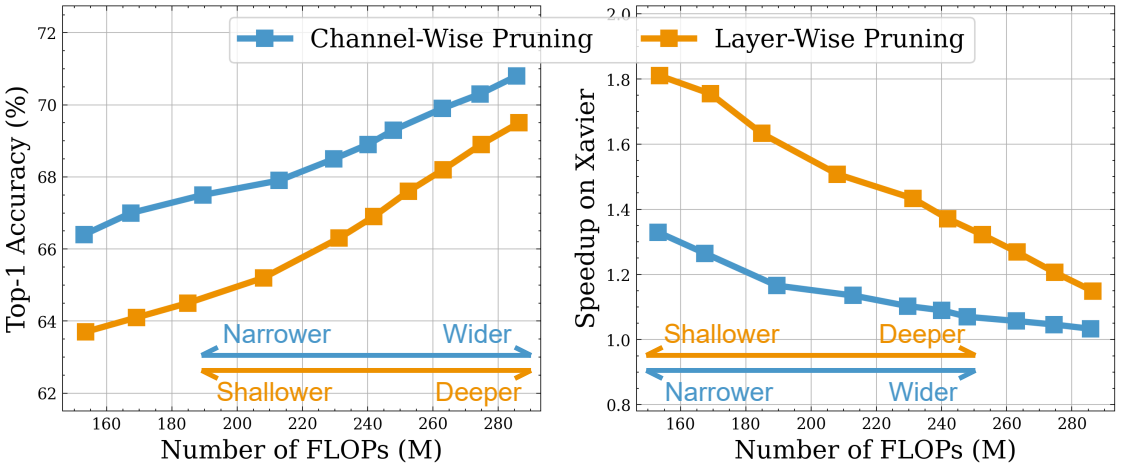

FIGURE 5.4: Comparisons between the accuracy on ImageNet (*left*) and the speedup on Xavier (*right*) under channel-wise and layer-wise pruning.

which also reveals that the non-linearity in different building blocks may exhibit higher importance for the attainable accuracy than others as discussed in [169].

Observation ⑤ *Pruning fails to achieve Pareto-optimal accuracy-efficiency trade-offs.* To show this, we first define a set of complexity constraints in the number of FLOPs and then gradually prune MobileNetV2 in terms of the network width (i.e., channel) and depth (i.e., layer), respectively, to accommodate the pre-defined constraints. After that, we (1) train the pruned network from scratch on ImageNet for 90 epochs to derive the accuracy and (2) measure the latency of the pruned network on Xavier with an input batch size of 8 to calculate the speedup. The experimental results are shown in Figure 5.4. We observe that channel-wise pruning, despite being able to deliver superior accuracy on ImageNet, suffers from inferior speedup on Xavier[3]. In contrast, layer-wise pruning can achieve promising speedup on Xavier, which, however, comes at the cost of severe accuracy loss on ImageNet. These reveal that pruning cannot win both accuracy and efficiency, and thus fails to enable *Pareto-optimal* accuracy-efficiency trade-offs. These also indicate that hardware-efficient networks should be shallow rather than narrow.

Motivations. As discussed in Observation ⑤, both channel-wise and layer-wise pruning cannot achieve *Pareto-optimal* accuracy-efficiency trade-offs, which motivates us to innovate pruning-free alternatives for network compression towards

[3]Unlike layer-wise pruning that can make full use of available resources, channel-wise pruning suffers from significant resource underutilization on Xavier, especially under a large pruning ratio.

Pareto-optimal accuracy-efficiency trade-offs. An alternative, inspired by Observation ① and ②, is to reparameterize all the linear layers into one single linear layer to trim down the network complexity for free, which, however, is infeasible because linear layers and non-linear layers alternate in order to maintain superior accuracy on target task [6, 19] (see Figure 5.1). In addition, as shown in [169], some non-linear activation layers may be less important than others and thus can be removed with minimal accuracy loss, which has also been applied to graph neural networks with consecutive FC layers [170]. These further motivate us to graft the intermediate non-linear activation layers with the linear counterparts, after which we are allowed to reparameterize multiple consecutive linear layers into one single linear layer. In other words, different from state-of-the-art pruning methods [69, 70, 72, 126–128, 164–168] that trade the less important channels or layers for better network efficiency, we can instead trade the less important network non-linearity for better network efficiency, which is well-supported by Observation ③ and ④. Motivated by these observations, we introduce a novel pruning-free network compression alternative, namely *Domino*, striving to revisit the accuracy-efficiency trade-off from a fresh perspective of linearity and non-linearity. More importantly, different from state-of-the-art channel-wise and layer-wise pruning methods [69, 70, 72, 126–128, 164–168] that can only win either accuracy or efficiency, *Domino* wins both without pruning any channels or layers.

5.3 Methodology

In this section, we first present the problem formulation in Section 5.3.1, after which we introduce several key components of *Domino*, including vanilla latency predictor in Section 5.3.2, meta-accuracy predictor in Section 5.3.3, evolutionary exploration in Section 5.3.4, and linear reparameterization in Section 5.3.5. Finally, we discuss the relationships with previous relevant methods in Section 5.3.6.

5.3.1 Problem Formulation

Let $\alpha \in \{0, 1\}^{1\times L}$ represent the grafted network, in which 0 and 1 correspond to the grafted linear building block and the non-linear building block, respectively. Here,

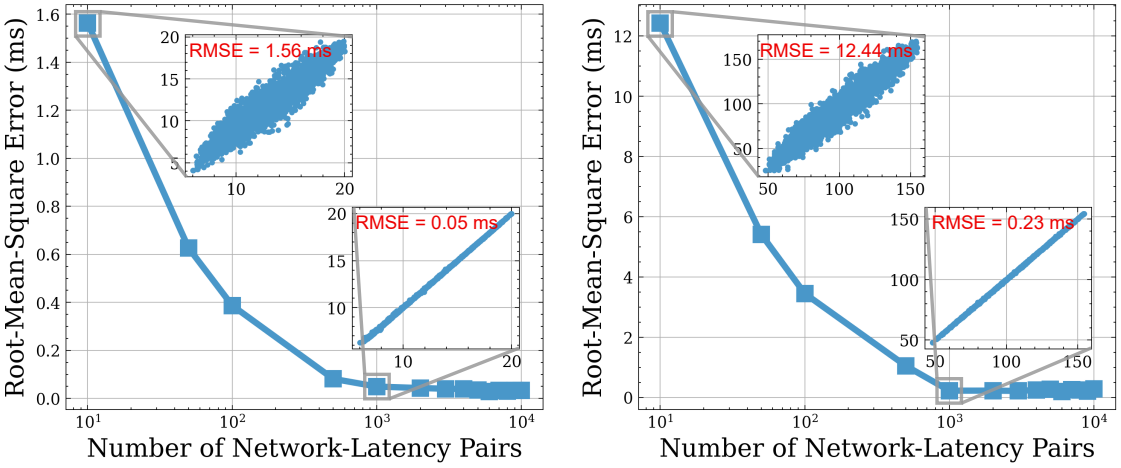

FIGURE 5.5: Illustration of the latency prediction results on Xavier (*left*) and Nano (*right*) under different numbers of network-latency pairs ranging from 10 to 10,000, where zoom-in figures visualize the predicted and measured latency.

L is the number of total building blocks. With the above in mind, we introduce the following multi-objective optimization to derive *Pareto-optimal* grafted networks:

$$\underset{\alpha}{\text{maximize}} \quad ACC(\alpha) \text{ s.t., } \quad LAT(\alpha) \leq T \tag{5.3}$$

where $ACC(\cdot)$ and $LAT(\cdot)$ are the accuracy on target task and the latency on target hardware, respectively. Besides, T is the specified latency constraint. Unless explicitly stated otherwise, $LAT(\cdot)$ corresponds to the on-device latency after the linear reparameterization is applied (see Section 5.3.5). Note that the linear reparameterization does not change the network output as discussed in Observation ①. As such, $ACC(\cdot)$ remains the same, regardless of linear reparameterization.

Remarks. On the one hand, we note that the derived grafted networks might not be globally *Pareto-optimal* in the strict sense, especially when involving approximate strategies, such as evolutionary algorithm [92]. In contrast, *Pareto-optimality*, in the context of network compression, implies that the derived networks are able to outperform state-of-the-art counterparts in the relevant literature [171]. On the other hand, in this work, we, following recent conventions [27], consider the on-device latency as the default constraint, which can be easily generalized to other hardware performance constraints, such as energy. For example, as pointed out in [24], there exists an extremely strong correlation between the latency and energy measurements on resource-constrained embedded platforms, which demonstrates that latency-efficient networks are also energy-efficient and vice versa.

5.3.2 Vanilla Latency Prediction

The predominant consensus in the network compression community is to trim down the network complexity in terms of the number of FLOPs [27, 36]. Nonetheless, the number of FLOPs only represents the theoretical network complexity and the reduction in the number of FLOPs does not necessarily translate into the expected speedup on target hardware as shown in [100]. As such, in pursuit of hardware-efficient networks, we here leverage the direct on-device latency as the optimization objective. However, different from the number of FLOPs that is easy to calculate, we have to first compile and then deploy the possible network on target hardware in order to interpret the latency, which inevitably becomes computationally infeasible when the design space exponentially evolves [24].

To avoid the tedious on-device measurements, we, following [100], construct a simple yet effective vanilla latency predictor $LAT(\cdot)$, which takes the network encoding $\alpha \in \{0,1\}^{1\times L}$ as the input to predict the on-device latency $LAT(\alpha)$. Specifically, we adopt a simple multi-layer perceptron (MLP) model, which consists of three fully-connected layers with 128, 64, and 1 neuron(s). To train the vanilla latency predictor, we first generate 1,000 random grafted networks, which are then reparameterized to derive the corresponding simplified networks. Next, we deploy the resulting simplified networks on Xavier and Nano to establish the latency measurements. Finally, we divide the established network-latency pairs into two splits with 80% as the training set and 20% as the validation set, respectively. Note that the design cost of the vanilla latency predictor, including (1) collecting 1,000 network-latency pairs and (2) training the vanilla latency predictor, is negligible, which only requires less than one hour on both Xavier and Nano.

Results. Taking MobileNetV2 as an example, we show the latency prediction results on the validation set in Figure 5.5 to demonstrate the efficacy of the vanilla latency predictor. Specifically, we observe that the vanilla latency predictor $LAT(\cdot)$ is able to accurately predict the latency on both Xavier and Nano using 1,000 network-latency pairs, which achieves an extremely low root-mean-square error (RMSE) of 0.05 ms on Xavier and 0.23 ms on Nano, respectively. In parallel, we also observe that the latency prediction performance can be further improved with an increasing number of network-latency pairs, among which 1,000 network-latency pairs are sufficient to train an accurate vanilla latency predictor.

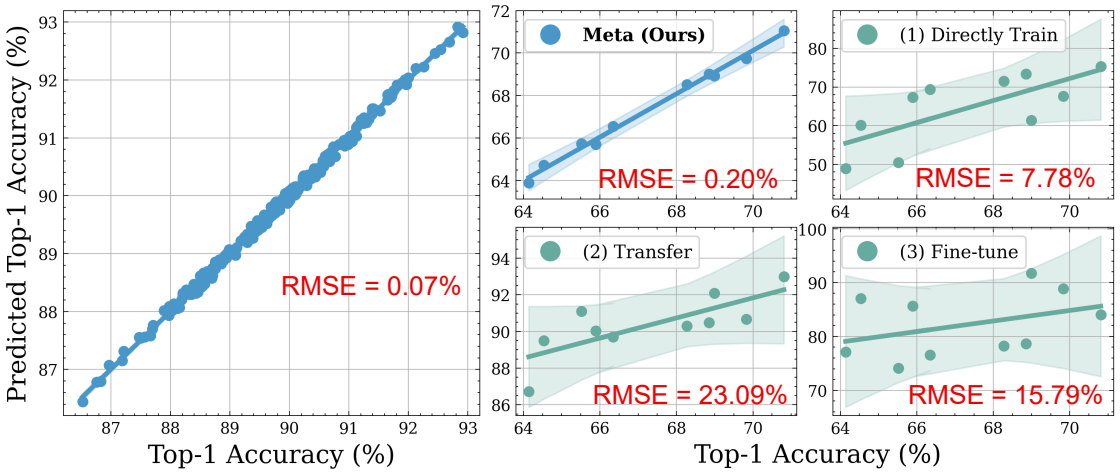

FIGURE 5.6: Illustration of the accuracy prediction results on ImageNet-tiny (*left*) and ImageNet (*right*) using the proposed meta-accuracy predictor and another three natural counterparts under similar computational resources.

5.3.3 Meta-Accuracy Prediction

As shown in Section 5.3.2, collecting network-latency pairs on target hardware is computationally efficient (e.g., less than one hour for 1,000 network-latency pairs on both Xavier and Nano). In contrast, to establish one network-accuracy pair, we have to train the given network on target task for hundreds of epochs, which typically involves days on GPUs. For example, training MobileNetV2 [6] on ImageNet for 180 epochs takes $\sim$1 day on 4 GeForce RTX 3090 GPUs. This further makes it computationally unaffordable to collect sufficient network-accuracy pairs on target task (e.g., thousands of days for 1,000 network-accuracy pairs on ImageNet), which, unfortunately, are essential to training an accurate performance predictor as shown in Figure 5.5. The intuition behind this is simply that insufficient training data significantly over-fits the performance predictor, which leads to superior prediction performance on the training set but fails to generalize to the unseen validation set.

To tackle this dilemma, unlike [172, 173] that collect a large number of network-accuracy pairs on target task (i.e., 1,000), we propose a novel meta-learning [174] based accuracy predictor, which strives to predict the accuracy on target task using minimal computational resources. Specifically, we focus on meta-learning a reliable accuracy predictor on small proxy task (i.e., ImageNet-tiny) using sufficient network-accuracy pairs, which can then adapt to target task (i.e., ImageNet) using as few as 10 network-accuracy pairs on target task. To achieve this, we first

randomly sample 10 out of 1,000 categories from ImageNet to establish ImageNet-tiny. After that, we train 1,000 random grafted networks on ImageNet-tiny, where the required computational resources are equivalent to training 10 random grafted networks on ImageNet. As shown in Figure 5.6 (*left*), 1,000 network-accuracy pairs are sufficient to yield reliable accuracy prediction on ImageNet-tiny.

We next elaborate on the meta-learning process. Let $ACC(\cdot)$ denote the meta-accuracy predictor that shares the same layer structure as the vanilla latency predictor $LAT(\cdot)$ as discussed in Section 5.3.2, in which θ is the learnable parameter of the meta-accuracy predictor. During the meta-learning process, θ is automatically optimized in order to achieve reliable accuracy prediction. For the simplicity of notation, below we use $f_\theta(\cdot)$ to denote the meta-accuracy predictor. Besides, $P = \{p_i|(\alpha_i, y_i)\}_{i=1}^{1,000}$ represents a set of 1,000 network-accuracy pairs on ImageNet-tiny, where y_i is the accuracy of each sampled grafted network α_i on ImageNet-tiny. After one gradient descent, θ is updated as follows:

$$\theta' = \theta - \eta \nabla_\theta \mathcal{L}(f_\theta(\alpha_i), y_i) \tag{5.4}$$

where η represents the inner-loop learning rate. Besides, $\mathcal{L}(\cdot)$ denotes the mean squared error (MSE) loss function. With the above in mind, we, following [174], mathematically formulate the meta-optimization objective as follows:

$$\min_\theta \sum_{p_i \in P} \mathcal{L}(f_{\theta'}(\alpha_i), y_i) = \sum_{p_i \in P} \mathcal{L}(f_{\theta - \eta \nabla_\theta \mathcal{L}(f_\theta(\alpha_i), y_i)}(\alpha_i), y_i) \tag{5.5}$$

Note that $f_{\theta'}(\alpha_i)$ is still continuous with respect to the input α_i as demonstrated in [174]. In sight of this, we further leverage the standard stochastic gradient descent (SGD) scheme to optimize θ as follows:

$$\theta \leftarrow \theta - \lambda \nabla_\theta \sum_{p_i \in P} \mathcal{L}(f_{\theta'}(\alpha_i), y_i) \tag{5.6}$$

where λ represents the outer-loop learning rate. Once the meta-learning process in Eq (5.5) and Eq (5.6) converges, we adapt the predictor meta-learned on ImageNet-tiny to ImageNet using as few as 10 network-accuracy pairs on ImageNet.

Results. Taking MobileNetV2 as an example, we compare the proposed meta-accuracy predictor with another three first-in-mind counterparts under similar computational resources, including (1) directly trained on ImageNet, (2) first trained on

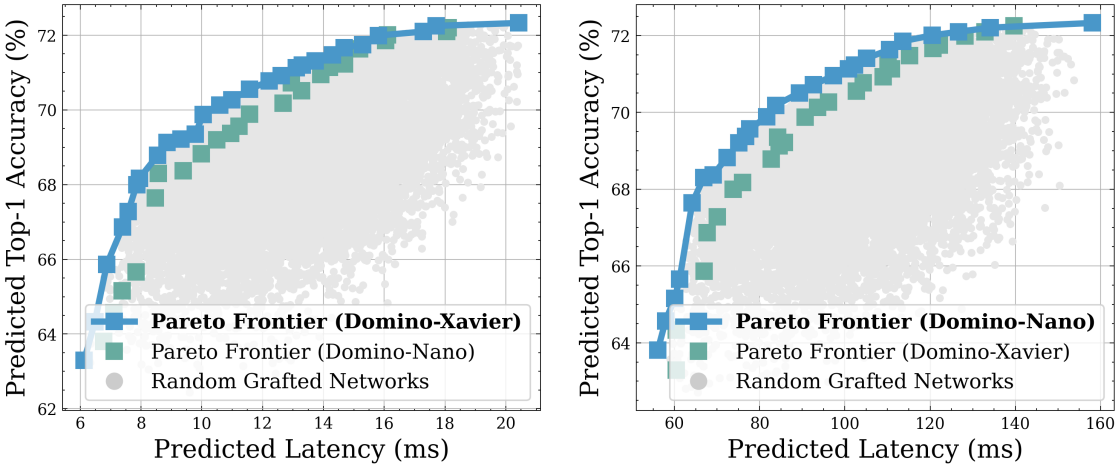

FIGURE 5.7: Visualization of evolutionary exploration on Xavier (*left*) and Nano (*right*), in which gray data points correspond to random grafted networks.

ImageNet-tiny and then transferred to ImageNet, and (3) first trained on ImageNet-tiny and then fine-tuned on ImageNet. The experimental results are illustrated in Figure 5.6 (*right*). We clearly observe that, compared with the above three counterparts, the proposed meta-accuracy predictor $ACC(\cdot)$ is able to deliver significantly better accuracy prediction performance on ImageNet, which achieves a much lower root-mean-square error (RMSE) of 0.20%.

Cost Analysis. Below we break down the design cost, which is two-fold. On the one hand, we train 1,000 grafted networks on ImageNet-tiny, resembling roughly the same computational resources as training 10 grafted networks on ImageNet. Besides, we also train 10 grafted networks on ImageNet for adaptation. Taken together, the total computational resources are roughly equivalent to training 20 grafted networks on ImageNet, which are $\times 50$ less than previous relevant methods [172, 173]. On the other hand, the cost of training the meta-accuracy predictor using the above network-accuracy pairs is negligible, which only requires minutes on one single GeForce RTX 3090 GPU. Note that the above is a one-time cost that can be amortized across diverse deployment scenarios (i.e., once for all).

5.3.4 Evolutionary Exploration

After establishing the vanilla latency predictor $LAT(\cdot)$ and the meta-accuracy predictor $ACC(\cdot)$, we introduce an efficient evolutionary-based search algorithm [14]

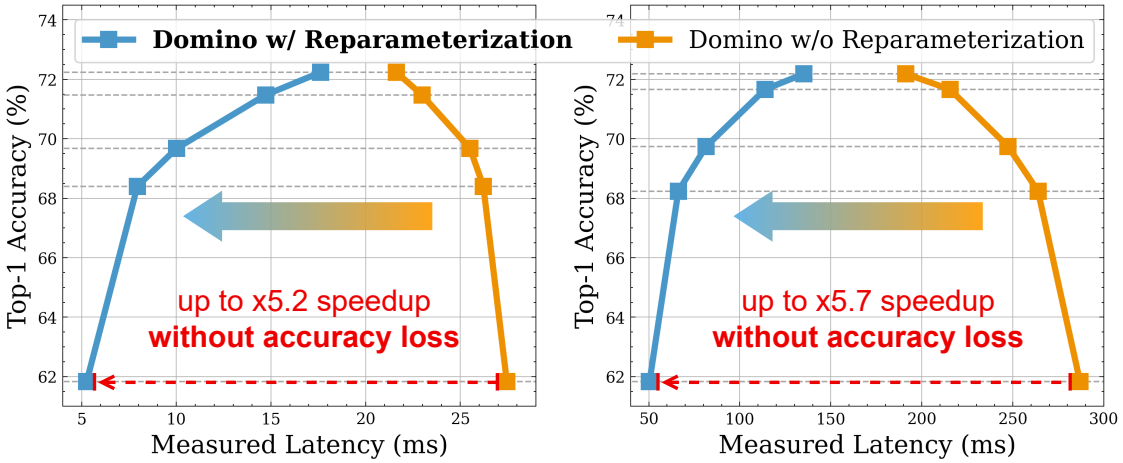

FIGURE 5.8: Linear reparameterization on Xavier (*left*) and Nano (*right*).

to navigate *Pareto-optimal* grafted networks α, in which we focus on maximizing the accuracy under the specified latency constraint T as shown in Eq (5.3). To this end, during the evolutionary exploration, we leverage mutation and crossover to jointly generate sufficient grafted network candidates, which are further fed into the above two predictors $LAT(\cdot)$ and $ACC(\cdot)$ to approximate the latency on target hardware and the accuracy on target task. After that, we reserve those grafted network configurations with top accuracy results for subsequent evolutionary exploration. Finally, the evolutionary exploration can converge upon the optimal grafted network configuration α. Note that the evolutionary engine with $LAT(\cdot)$ and $ACC(\cdot)$ can explore the design space on the fly, which necessitates only minutes even on one single GeForce RTX 3090 GPU.

Results. Taking MobileNetV2 as an example, we visualize the evolutionary exploration in Figure 5.7, which demonstrates that the evolutionary engine integrated with the above two predictors is able to derive *Pareto-optimal* grafted networks with minimal computational resources. In addition, we observe that the *Pareto-optimal* grafted networks on Xavier are not necessarily *Pareto-optimal* on Nano and vice versa. This empirical observation aligns with the findings in [14], which again demonstrates the necessity to design and deploy specialized network solutions towards different hardware platforms in order to embrace the best accuracy-efficiency trade-off. Finally, we denote the resulting *Pareto-optimal* grafted networks on Xavier and Nano as *Domino*-Xavier and *Domino*-Nano, respectively.

5.3.5 Linear Reparameterization

Subsequently, we train *Pareto-optimal* grafted networks on target task to interpret the attainable accuracy[4]. Finally, to derive the simplified network for further direct deployment on target hardware, we reparameterize the grafted linear building block that consists of multiple consecutive linear layers, including three convolutional, three BN, and two grafted linear activation layers into one single convolutional layer. An example is shown in Figure 5.1 (*bottom*). Note that reparameterizing one convolutional layer and one subsequent BN or grafted linear activation layer into one single convolutional layer is straightforward, which is equivalent to linearly transforming the convolutional layer. As such, below we discuss the linear reparameterization in terms of multiple consecutive convolutional layers.

Without loss of generality, we here consider two consecutive convolutional layers with an input feature $X \in \mathbb{R}^{C_1 \times H_1 \times W_1}$, an intermediate feature $Y \in \mathbb{R}^{C_2 \times H_2 \times W_2}$, and an output feature $Z \in \mathbb{R}^{C_3 \times H_3 \times W_3}$. Besides, $W^1 \in \mathbb{R}^{C_1 \times C_2 \times K_1 \times K_1}$ and $W^2 \in \mathbb{R}^{C_2 \times C_3 \times K_2 \times K_2}$ are the weights of the above two convolutional layers. Taken together, we can formulate the above two consecutive convolutional layers as follows:

$$Z_{c,h,w} = \sum_{c_i=0}^{C_1-1} \sum_{k_h^*=0}^{K^*-1} \sum_{k_w^*=0}^{K^*-1} W^*_{c_i,c,k_h^*,k_w^*} X_{c_i,h+\Delta k_h^*,w+\Delta k_w^*} \tag{5.7}$$

where $\Delta k_h^* = k_h^* - [\frac{K^*-1}{2}]$ and $\Delta k_w^* = k_w^* - [\frac{K^*-1}{2}]$. Besides, W^* is the weight of the reparameterized convolutional layer, which has the kernel size of $K^* = K_1 + K_2 - 1$ [175]. Furthermore, we are able to calculate W^* as follows:

$$W^*_{c_i,c_o,h,w} = \sum_{c_j=0}^{C_2-1} \sum_{k_h=0}^{K_2-1} \sum_{k_w=0}^{K_2-1} W^1_{c_i,c_j,k_h,k_w} W^2_{c_j,c_o,h-\Delta k_h,w-\Delta k_w} \tag{5.8}$$

where $\Delta k_h = k_h - [\frac{K_2-1}{2}]$ and $\Delta k_w = k_w - [\frac{K_2-1}{2}]$. Finally, the above two convolutional layers are reparameterized into one single convolutional layer with $W^* \in \mathbb{R}^{C_1 \times C_3 \times K^* \times K^*}$, both of which maintain the same output Z. As such, the attainable accuracy remains the same, regardless of linear reparameterization.

[4]In this work, we do not directly train the simplified network, which suffers from non-negligible accuracy loss on target task as illustrated in Figure 5.15.

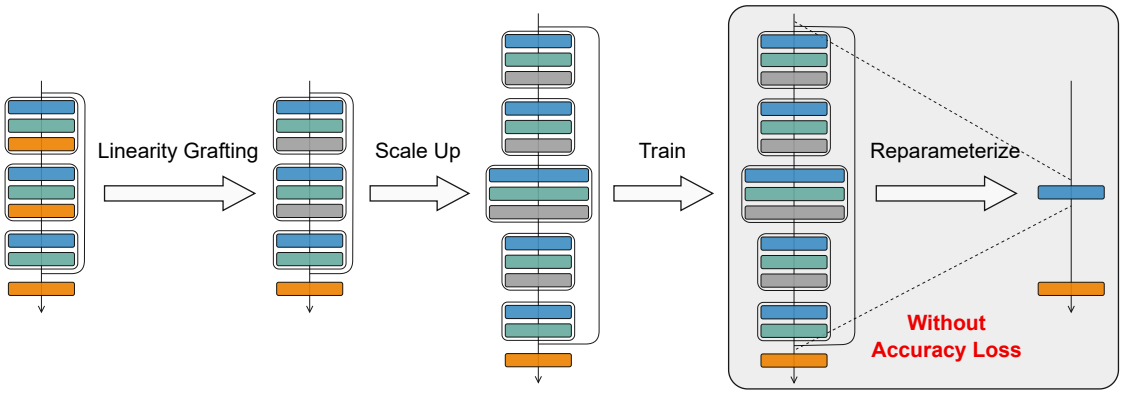

FIGURE 5.9: Overview of *Domino-Pro*, in which each rectangle represents one specific network layer and the definition can be found in Figure 5.1.

Results. Taking MobileNetV2 as an example, we show the effectiveness of linear reparameterization in Figure 5.8, which clearly demonstrates that linear reparameterization is able to deliver up to ×5.2 and ×5.7 speedup on Xavier and Nano[5], and more importantly, without compromising the accuracy on ImageNet since the network output, regardless of linear reparameterization, remains the same as discussed in Observation ①. Finally, we visualize the simplified networks in Figure 5.10, which target Xavier and take MobileNetV2 as the baseline network.

Insights and *Domino-Pro*. As shown in previous representative networks [6, 19], the basic building block typically consists of three convolutional layers with diverse kernel sizes of 1, 3, and 1. In sight of this convention, we consider the grafted linear building block that has three convolutional layers with $W^1 \in \mathbb{R}^{C_1 \times C_2 \times 1 \times 1}$, $W^2 \in \mathbb{R}^{C_2 \times C_3 \times 3 \times 3}$, and $W^3 \in \mathbb{R}^{C_3 \times C_4 \times 1 \times 1}$. According to Eq (5.8), one 1×1 and one 3×3 convolutional layer, regardless of the layer order, can always be reparameterized into one 3×3 convolutional layer. With this in mind, we, following Eq (5.8), progressively reparameterize the above three convolutional layers into one single convolutional layer with $W^* \in \mathbb{R}^{C_1 \times C_4 \times 3 \times 3}$. This reveals that, for the grafted linear building block, we can (1) scale up the intermediate convolutional layer in terms of the number of input and output channels (e.g., C_2 and $C_3 \to +\infty$) and (2) insert an arbitrary number of intermediate 1×1 convolutional layers. In practice, the above two enhancements increase the complexity of the grafted linear building

[5]The latency of the grafted network temporarily increases since grafted linear activation layers involve more computational resources than non-linear activation layers. Fortunately, such latency increase can be fully eliminated after linear reparameterization is applied as shown in Figure 5.1.

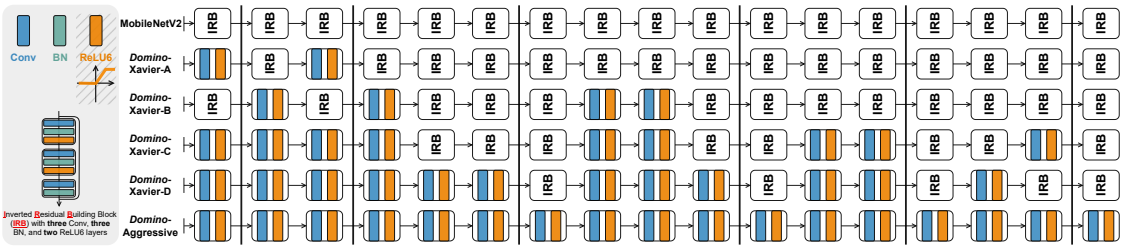

FIGURE 5.10: Illustration of the simplified networks targeting Xavier with MobileNetV2 as the baseline network. Different from ResNet50 (see Figure 5.1), MobileNetV2 does not include non-linear activation layer at the end of each building block. To this end, we insert one non-linear activation layer (i.e., ReLU6) at the end of each grafted linear building block, which causes negligible computational cost. The related ablation experiment can also be found in Figure 5.16.

block during the training process to achieve better accuracy, after which the enhanced grafted linear building block can still be reparameterized back into the same convolutional structure with $W^* \in \mathbb{R}^{C_1 \times C_4 \times 3 \times 3}$ for further direct deployment. It is worth noting that we only apply the above two enhancements to the grafted linear building blocks rather than the non-linear building blocks. The rationale here is that, for the non-linear building blocks, the stacked convolutional layers are not linearly consecutive since there are non-linear activation layers between these stacked convolutional layers. This makes it impossible to equivalently reparameterize these stacked convolutional layers into one single convolutional layer according to Observation ①. Finally, we denote *Domino* with such enhancements as *Domino-Pro* (see Figure 5.9), which boosts the attainable accuracy on target task while maintaining the same efficiency on target hardware as shown in Figure 5.17.

5.3.6 Relationships with Previous Methods

The pruning paradigm, which is the most relevant technique to *Domino* since both aim to derive simplified network structures as shown in Figure 5.1, has been dominating recent advances in the field of network compression [27]. Specifically, previous state-of-the-art pruning methods remove the redundant channels [126–128, 164–168] or layers [69, 70, 72] to reduce the network complexity. Despite the impressive success to date, previous channel-wise and layer-wise pruning methods suffer from inferior speedup on target hardware due to significant resource underutilization [70, 71] and non-negligible accuracy loss on target task due to

TABLE 5.2: Comparisons with the uniform baselines on ImageNet, where MobileNetV2 is the baseline network. Note that the numbers in the brackets denote the accuracy loss and the on-device speedup, respectively.

Network	Accuracy (%)		Latency (ms)	
	Top-1	Top-5	Xavier	Nano
MobileNetV2×1.0	72.3 (−0.0)	90.8 (−0.0)	20.4 (×1.0)	157.0 (×1.0)
MobileNetV2×0.8	70.6 (−1.7)	89.6 (−1.2)	18.0 (×1.1)	143.5 (×1.1)
***Domino*-Xavier-A**	72.2 (−0.1)	90.8 (−0.0)	17.6 (×1.2)	139.7 (×1.1)
***Domino*-Nano-A**	72.2 (−0.1)	90.8 (−0.0)	18.1 (×1.1)	135.1 (×1.2)
MobileNetV2×0.6	67.7 (−4.6)	88.0 (−2.8)	15.1 (×1.4)	119.5 (×1.3)
***Domino*-Xavier-B**	71.5 (−0.8)	90.5 (−0.3)	14.7 (×1.4)	120.6 (×1.3)
***Domino*-Nano-B**	71.7 (−0.6)	90.5 (−0.3)	16.0 (×1.3)	113.8 (×1.4)
MobileNetV2×0.4	62.1 (−10.2)	83.7 (−7.1)	10.9 (×1.9)	82.4 (×1.9)
***Domino*-Xavier-C**	69.7 (−2.6)	89.2 (−1.6)	10.0 (×2.0)	90.7 (×1.7)
***Domino*-Nano-C**	69.7 (−2.6)	89.0 (−1.8)	11.6 (×1.8)	81.4 (×1.9)
MobileNetV2×0.2	51.2 (−21.1)	74.9 (−15.9)	9.9 (×2.1)	65.2 (×2.4)
***Domino*-Xavier-D**	68.4 (−3.9)	88.2 (−2.6)	7.9 (×2.6)	75.9 (×2.1)
***Domino*-Nano-D**	68.2 (−4.1)	88.0 (−2.8)	8.6 (×2.4)	66.2 (×2.4)
***Domino*-Aggressive**	61.8 (−10.5)	83.7 (−7.1)	5.3 (×3.8)	50.4 (×3.1)

insufficient network depth [19, 27], respectively, and as a result, fail to achieve *Pareto-optimal* accuracy-efficiency trade-offs. In contrast, *Domino* revisits the trade-off dilemma between accuracy and efficiency from a fresh perspective of linearity and non-linearity, pioneering a first-of-its-kind pruning-free alternative to simplify deep convolutional networks without pruning any channels or layers as visualized in Figure 5.1. This explicitly distinguishes *Domino* from previous relevant methods in the rich pruning literature. More importantly, unlike previous pruning methods [69, 70, 72, 126–128, 164–168] that can only win either accuracy or efficiency, *Domino* is able to win both, especially under a large compression ratio. Note that *Domino* is fundamentally different from [176] although both feature linearity grafting. This is because [176] focuses on grafting the non-linearity of network channels for enhanced certifiable robustness, whereas *Domino* focuses on grafting the non-linearity of network layers for efficient network compression.

Remarks. We, in this work, compare *Domino* with pruning because other compression techniques (i.e., quantization [66, 177–180] and knowledge distillation [34, 35, 181]) cannot deliver simplified network structures [27]. For example, quantization reduces the precision of network weights to lower bits (e.g., from 32 bits to 8 bits), whereas the network structure still remains the same. This indicates that quantization and knowledge distillation are complementary to *Domino* and

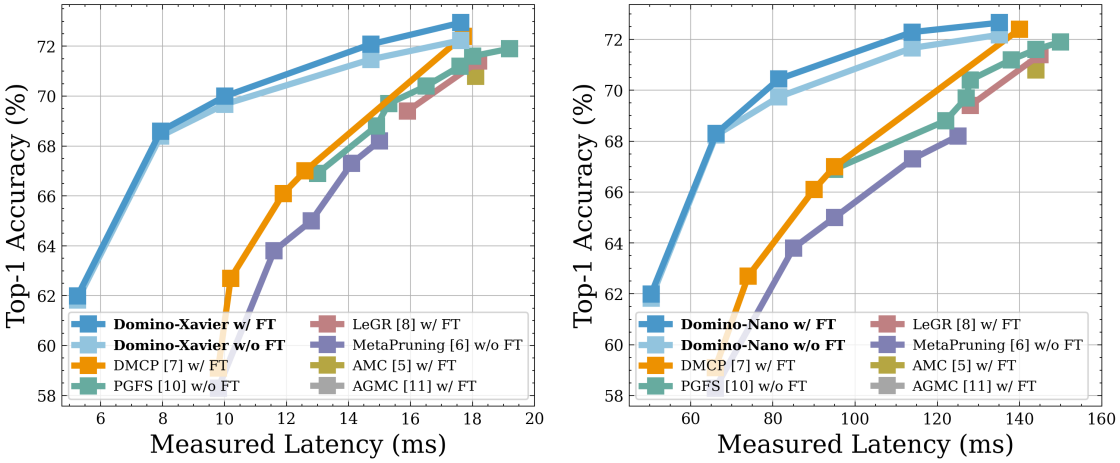

FIGURE 5.11: Comparisons with state-of-the-art pruning works on Xavier (*left*) and Nano (*right*) using MobileNetV2 [6] as the baseline network. Here, `w/ FT` is the fine-tuned accuracy, whereas the accuracy `w/o FT` is trained from scratch.

can be integrated into *Domino* to further boost the accuracy-efficiency trade-off. An ablation experiment is also provided in Section 5.4.3 (see Figure 5.13). Besides, we, in this work, compare *Domino* with pruning on two representative manually designed networks [6, 19], which can also be seamlessly generalized to compress efficient networks searched by popular NAS techniques. Taking LightNAS [85] as an example, below we demonstrate how to employ the proposed compression technique to further compress the searched LightNets. Similar to *Domino*, we can first build two efficient latency and accuracy predictors for each searched LightNet to achieve reliable latency and accuracy prediction. After that, we can employ the same evolutionary algorithm to explore the less important non-linear building blocks of each searched LightNet, which are then grafted with their linear counterparts with minimal accuracy loss on target task. Furthermore, we train the resulting grafted network on target task to achieve superior accuracy. Finally, for the well-trained grafted network, we can equivalently reparameterize each of its linear building blocks with multiple consecutive linear layers into one single linear layer to derive its simplified network for direct on-device deployments.

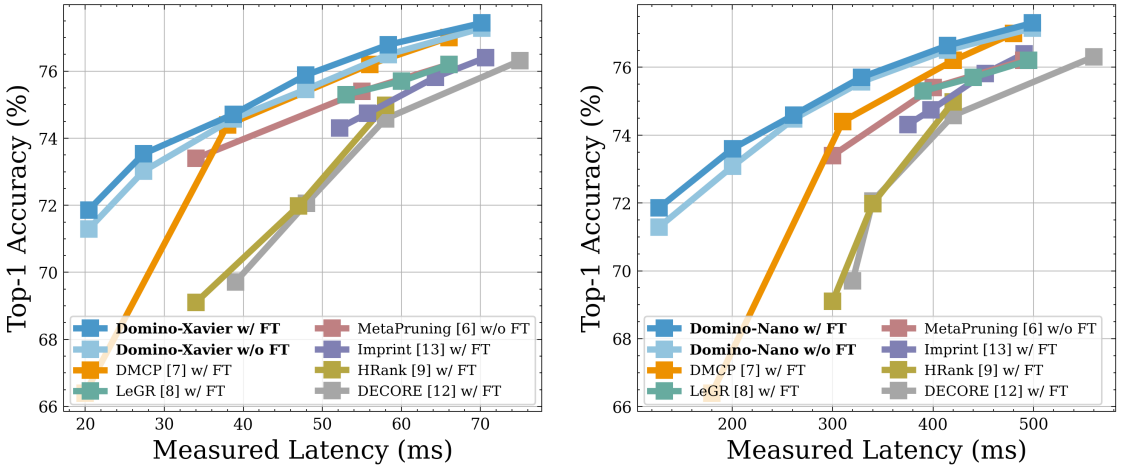

FIGURE 5.12: Comparisons with state-of-the-art pruning works on Xavier (*left*) and Nano (*right*) using ResNet50 [19] as the baseline network. Here, `w/ FT` is the fine-tuned accuracy, whereas the accuracy `w/o FT` is trained from scratch.

5.4 Experiments

In this section, we conduct extensive experiments, including thorough comparisons with previous relevant methods and insightful ablation studies, to show the superiority of *Domino* over previous state-of-the-art pruning methods.

5.4.1 Experimental Settings

Dataset. We conduct all the experiments on ImageNet [1], which is a popular large-scale dataset. Specifically, ImageNet consists of 1,281,167 training images and 50,000 validation images, which are roughly distributed across 1,000 categories. Besides, following previous conventions [126, 127], we employ the standard resolution setting, in which the input image resolution is fixed to 224×224.

Networks. We target two representative deep convolutional networks, including ResNet50 [19] and MobileNetV2 [6]. We note that MobileNetV2 is a lightweight compact network with less network redundancy than ResNet50 [27], thus being more challenging to compress as demonstrated in [126, 127].

Hardware Platforms. We target two mainstream embedded devices, including NVIDIA Jetson AGX Xavier and NVIDIA Jetson Nano, which are abbreviated as Xavier and Nano for the sake of simplicity. Specifically, Xavier is integrated with

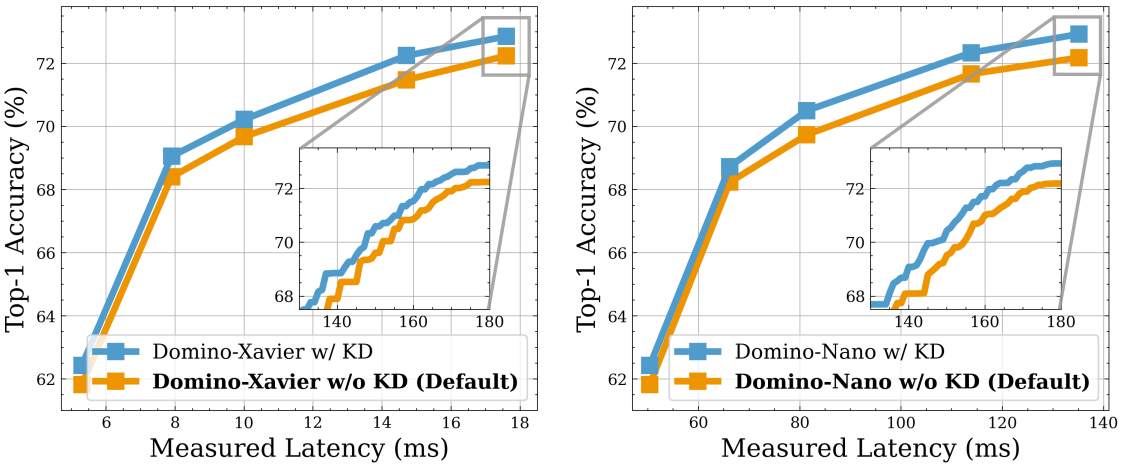

FIGURE 5.13: Ablation of knowledge distillation (KD) on Xavier (*left*) and Nano (*right*), where the zoom-in figure visualizes the training curve.

512-core Volta GPU and 32 GB LPDDR4x memory, whereas Nano is integrated with 128-core Maxwell GPU and 4 GB LPDDR4 memory. For both Xavier and Nano, we turn on the MAXN power mode to maximize the hardware performance.

Implementation Details. We implement all the experiments using PyTorch [182]. Specifically, all the *Domino*-related networks are trained from scratch for 180 epochs with a batch size of 1,024 on 4 GeForce RTX 3090 GPUs, in which the standard data augmentations [19] are applied across this work. The default optimizer is SGD with a momentum of 0.9 and a weight decay of 4×10^{-5}. Besides, the learning rate is initialized as 0.5, which first linearly increases from 0.1 to 0.5 in the first 5 epochs and then decays to zero following the cosine schedule [183]. To ensure fair comparisons with previous relevant methods, we do not apply extra training enhancements, such as stronger data augmentations (e.g., AutoAugment [184]) and more advanced training techniques (e.g., deep mutual training [35]).

Additional Remarks. On the one hand, to ensure fair comparisons with previous relevant methods in terms of latency, we report the latency measured on Xavier and Nano with the same input batch size of 8. On the other hand, we note that previous relevant methods may differ in reporting the final accuracy. For example, the accuracy reported in [127] is trained from scratch, whereas [126] reports the fine-tuned accuracy. In practice, the fine-tuned network typically exhibits better accuracy on target task than the same network trained from scratch. Therefore, to ensure fair comparisons with previous relevant methods in terms of accuracy, we

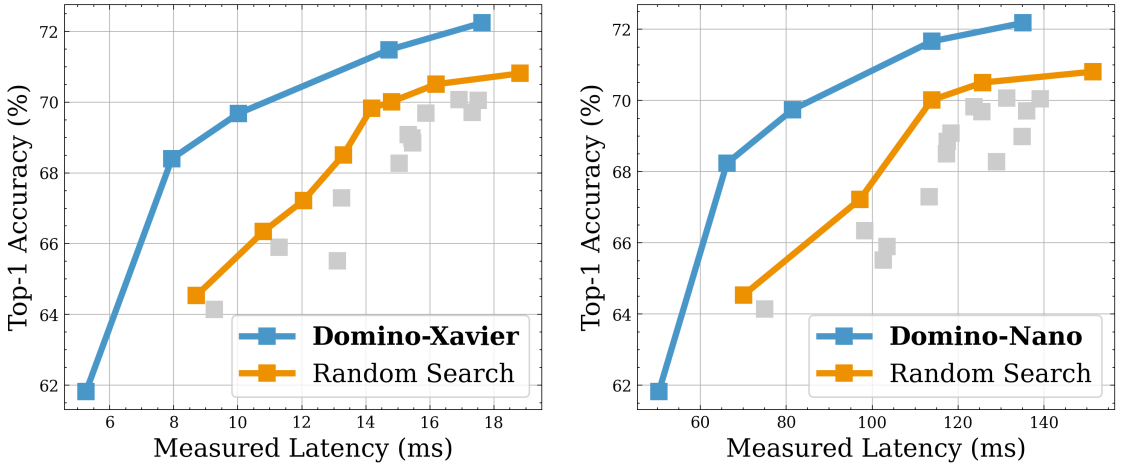

FIGURE 5.14: Comparisons with the random search scheme using *Domino*-Xavier on Xavier (*left*) and *Domino*-Nano on Nano (*right*).

train all the *Domino*-related networks with both schemes. Unless explicitly stated otherwise, we take training from scratch as the default training option.

5.4.2 Experimental Results

Comparisons with Uniform Baselines. Following previous conventions [27, 126, 127], we first compare *Domino* with the uniform baselines in Table 5.2, where we leverage MobileNetV2 as the baseline network. Note that *Domino*-Aggressive represents the grafted network, in which all the non-linear building blocks are grafted with the linear counterparts (see Figure 5.10). Without bells and whistles, *Domino* outperforms all the uniform baselines in terms of both accuracy and efficiency. For example, compared with MobileNetV2×0.2, *Domino*-Aggressive can achieve +10.6%/+8.8% higher top-1/top-5 accuracy on ImageNet, while maintaining ×1.9 and ×1.3 speedup on Xavier and Nano, respectively. Besides, we observe that networks targeting Xavier do not necessarily run fast on Nano and vice versa, which further demonstrates the necessity to design specialized networks towards different hardware platforms to embrace the best accuracy-efficiency trade-off.

Comparisons with State-of-the-Art Works. We next compare *Domino* with previous state-of-the-art pruning methods, including AMC [126], MetaPruning [127], DMCP [128], LeGR [164], HRank [165], PGFS [166], AGMC [167], DECORE [168], and Imprint [69]. Experimental results on MobileNetV2 and ResNet50 are

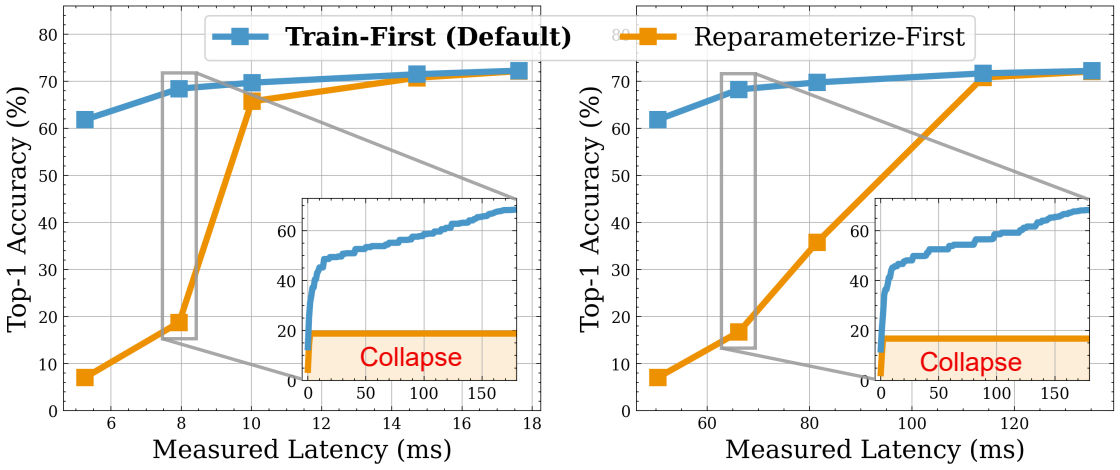

FIGURE 5.15: Train-First vs. Reparameterize-First on Xavier (*left*) and Nano (*right*), where the zoom-in figure visualizes the training curve.

shown in Figure 5.11 and Figure 5.12, in which we observe that *Domino* consistently achieves better accuracy-efficiency trade-offs on MobileNetV2 and ResNet50, especially under a large compression ratio. We note that MobileNetV2 is a lightweight compact network with less redundancy than ResNet50, due to which previous relevant methods may achieve satisfactory performance on ResNet50 but fail (or even do not conduct experiments) on MobileNetV2. In contrast, *Domino* can exhibit strong performance on both MobileNetV2 and ResNet50, which shows the superiority of *Domino* over previous well-established pruning methods.

5.4.3 Ablation Studies and Discussions

In this section, we further present rich and in-depth ablation studies to investigate *Domino* and draw more insights to benefit future research, in which we leverage MobileNetV2 as the default baseline network for all ablation studies.

***Domino* vs. Random Search.** We first compare *Domino* with another strong search baseline (i.e., random search), in which the non-linear building blocks are randomly grafted with the linear counterparts. To this end, we generate 20 random grafted networks to accommodate different latency constraints, which are then trained using the same training recipe as *Domino*. Finally, the well-trained random grafted networks are reparameterized into the simplified networks for further direct

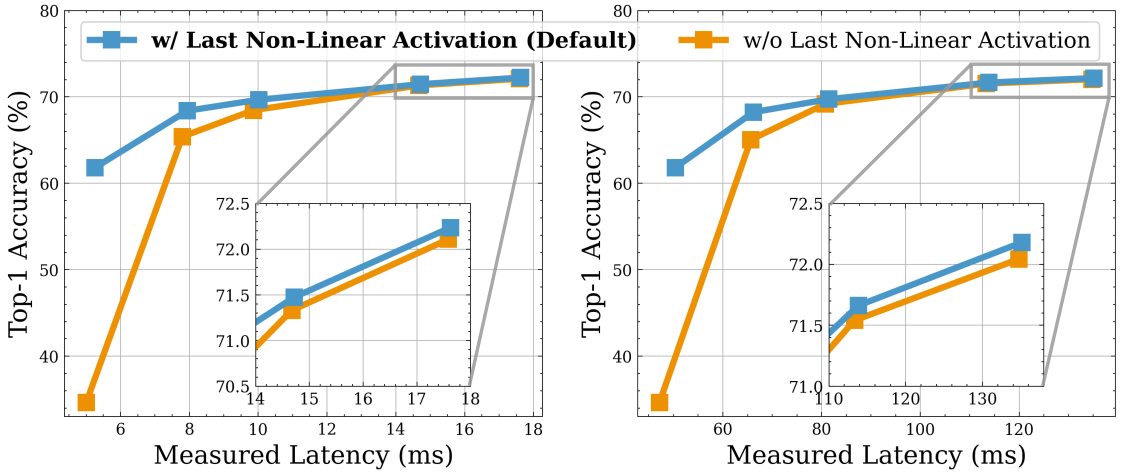

FIGURE 5.16: Ablation of last non-linear activation layer (i.e., ReLU6) using *Domino*-Xavier on Xavier (*left*) and *Domino*-Nano on Nano (*right*).

deployment on Xavier and Nano. As illustrated in Figure 5.14, *Domino* consistently achieves better accuracy-efficiency trade-offs than random search.

Ablation of Knowledge Distillation. As discussed in Section 5.3.6, other compression techniques (i.e., quantization and knowledge distillation) are complementary to *Domino* and can be seamlessly integrated into *Domino* to further enable better accuracy-efficiency trade-offs. To this end, we take knowledge distillation [34] as an example. The experimental results are shown in Figure 5.13, where we observe that *Domino* with knowledge distillation exhibits +0.49%∼+0.77% higher accuracy on ImageNet without degrading the efficiency on both Xavier and Nano. Unless explicitly stated otherwise, we do not integrate knowledge distillation into *Domino* to ensure fair comparisons with previous relevant methods.

Train-First vs. Reparameterize-First. As shown in Figure 5.1 (*bottom*), we first train the grafted network, which is then reparameterized to derive the simplified network. Apart from this, another training option is to first reparameterize the grafted network to derive the simplified network and then train the simplified network instead. In practice, both training options end up with the same simplified network structure that exhibits the same on-device efficiency. The latter, however, may suffer from severe accuracy loss on ImageNet as shown in Figure 5.15, which reveals that (1) the network redundancy benefits and stabilizes the training process to unlock better accuracy and (2) directly training the simplified network may collapse at the early training stage as shown in Figure 5.15's zoom-in figures.

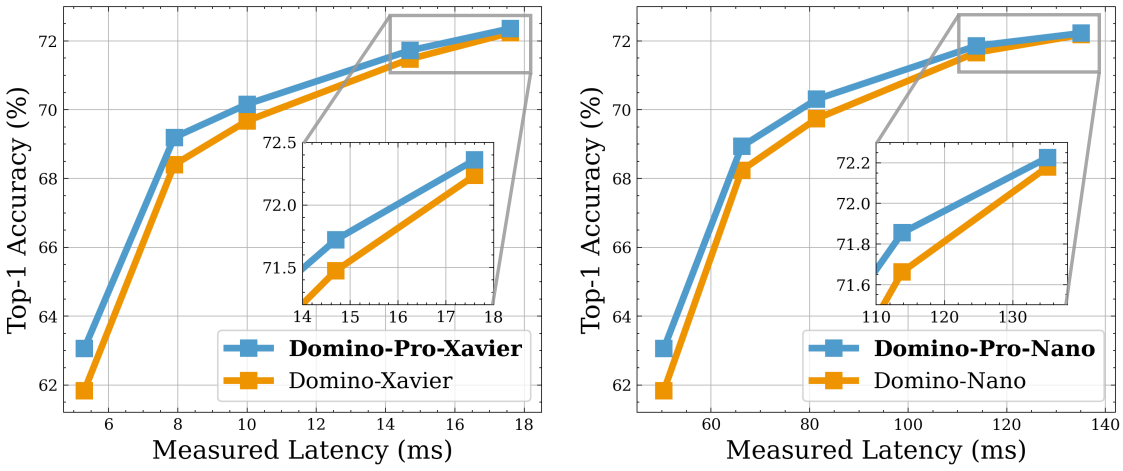

FIGURE 5.17: *Domino* vs. *Domino-Pro* on Xavier (*left*) and Nano (*right*).

Ablation of Last Non-Linear Activation Layer. As shown in Figure 5.1 (*bottom*), we graft intermediate non-linear activation layers and leave the last one ungrafted. The reason is that non-linear activation layers are computationally cheap as shown in Table 5.1. As such, inserting one non-linear activation layer at the end of each grafted linear building block only introduces negligible computational cost since multiple consecutive linear layers in the grafted linear building block can still be reparameterized into one single convolutional layer, but delivers strong accuracy improvement as illustrated in Figure 5.16.

***Domino* vs. *Domino-Pro*.** As discussed in Section 5.3.5, we can enhance the grafted network to temporarily increase the network redundancy during the training process, after which the enhanced grafted network can be reparameterized back into the same simplified network structure (see Figure 5.9) and thus exhibits the same on-device efficiency. We note *Domino* with such enhancements as *Domino-Pro*, which opens up new possibilities to further boost the attainable accuracy. As illustrated in Figure 5.17, *Domino-Pro* consistently exhibits better accuracy on ImageNet than *Domino* while strictly maintaining the same inference efficiency on both Xavier and Nano, which again demonstrates that the network redundancy may benefit the training process to boost the attainable accuracy.

5.5 Conclusion

In this chapter, we introduce a novel pruning-free compression technique dubbed *Domino*, which strives to simplify deep convolutional networks without pruning any channels or layers. In contrast to previous state-of-the-art pruning methods, *Domino*, as the first of its kind, revisits the trade-off dilemma between accuracy and efficiency from a fresh perspective of linearity and non-linearity. Extensive experiments on ImageNet with two popular NVIDIA Jetson embedded platforms and two representative deep convolutional networks demonstrate the superiority of *Domino* over previous state-of-the-art pruning approaches.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Convolutional neural networks (CNNs), thanks to their strong learning capability, have empowered a myriad of intelligent embedded scenarios, such as on-device object recognition, detection, and tracking. In the past decade, "*deeper + wider = better*" has been widely deemed as the rule of thumb to manually design networks with competitive accuracy. This rule, despite its efficacy, necessitates a huge amount of engineering effort from human experts, which also leads to an exponential growth in the number of operations (a.k.a., FLOPs) and parameters as the network evolves deeper and wider. The evolving network complexity further enlarges the computational gap between computation-intensive CNNs and resource-constrained embedded platforms, making it challenging to deploy powerful CNNs on less capable embedded platforms towards embedded intelligence.

In this thesis, we focus on alleviating the above computational gap using hardware-aware neural architecture search (NAS) and compression, in which we strive to establish hardware-friendly network solutions to accommodate the limited available computational resources in real-world embedded scenarios. Finally, we summarize the main contributions of this thesis as follows:

- In Chapter 3, we introduce SurgeNAS for efficient architecture search. Specifically, SurgeNAS turns back to one-level optimization for accurate and consistent gradient estimation, in which an effective identity mapping scheme is

integrated to avoid the search collapse. Besides, we introduce an efficient ordered differentiable sampling approach, which can decrease the optimization complexity to the single-path level while at the same time strictly satisfying the search fairness. Furthermore, to enable hardware-aware search, an efficient graph neural networks (GNNs) based latency predictor is plugged into the search process to avoid tedious on-device latency measurements. Finally, we introduce the paradigm of *Comfort Zone*, which allows us to scale up the searched architecture candidates for further accuracy improvement without degrading the inference efficiency on target hardware.

- In Chapter 4, we introduce LightNAS for flexible architecture search. The main motivation behind LightNAS is that previous relevant NAS methods, including SurgeNAS, have to perform manual hyper-parameter tuning through trial and error in order to navigate the required architecture candidate around the specified latency constraint, which empirically involves 10 trial-and-errors and thus significantly increases the total search cost by 10 times. In contrast, LightNAS only requires one single search for any specified latency constraint (i.e., *you only search once*), which thus exhibits significant search efficiency and flexibility. Furthermore, we also introduce an efficient and reliable proxy, namely batchwise training estimation (BTE), which can be seamlessly integrated into LightNAS to enable channel-level explorations at low computational cost. This further pushes forward the attainable accuracy on target task without degrading the efficiency on target hardware.

- In Chapter 5, we introduce *Domino* for efficient network compression, which strives to revisit the accuracy-efficiency trade-off from a fresh perspective of linearity and non-linearity. Specifically, *Domino* features two efficient predictors, including one vanilla latency predictor and one meta-accuracy predictor, to explore less important non-linear building blocks, which are then grafted with their linear counterparts. The resulting grafted network is further trained on target task to achieve superior accuracy. Finally, the well-trained grafted network is reparameterized to derive the simplified network, in which the grafted linear building block that consists of multiple consecutive linear layers, including multiple convolutional, batch normalization (BN), and grafted linear activation layers, is reparameterized into one single convolutional layer to aggressively boost the efficiency on target hardware, and more importantly, without sacrificing the accuracy on target task.

6.2 Future Work

In the future, we will continue to shrink the computational gap to design more efficient network solutions for resource-constrained embedded platforms. Below we discuss several promising future directions that may further deliver better accuracy-efficiency trade-offs towards boosted embedded intelligence.

6.2.1 General Search Space

The tremendous success of NAS highly relies on the well-engineered search spaces, such as the cell-based search space [3, 44] and the block-based search space [5, 9, 13]. The well-engineered search space here is coarse-grained and also a smaller subset of general fine-grained search spaces. These well-engineered search spaces, despite being able to facilitate the search process and improve the search efficiency, significantly limit the search performance, which may exclude more competitive architecture candidates that fall outside the well-engineered search spaces. In light of this, delving deeper into general fine-grained search spaces has the potential to provide better architecture candidates with enhanced accuracy-efficiency trade-offs. However, this may pose significant challenges since general fine-grained search spaces are much larger than current popular coarse-grained search spaces, which may require new search algorithms for efficient and effective exploration over general fine-grained search spaces. To overcome such limitations, we, in the future, will explore more general and more fine-grained search spaces, such as layer-wise or even channel-wise search spaces, to unleash the promise of NAS.

6.2.2 Dynamic Architecture Search

Previous NAS methods [5, 9, 13] focus on searching for static architecture candidates that can only run at fixed computational budgets, which cannot adapt to lower or higher computational budgets. However, the available computational budgets in real-world embedded scenarios are often subject to variations. For example, in some cases, mobile phones may be in low-power or power-saving modes to reduce the power consumption. Therefore, one promising future direction in the field of NAS is to search for efficient dynamic architecture candidates, which can instantly

trade off between accuracy and efficiency to accommodate the rapidly-changing computational budgets in real-world embedded scenarios.

6.2.3 Fully Automated Architecture Search

Previous NAS methods either focus on searching for the optimal architecture candidate [44], the optimal data augmentation [184], the optimal activation function [185], or the optimal training recipe [186, 187]. As shown in [186, 187], different architecture candidates may prefer different training recipes, in which jointly searching for the optimal architecture candidate and its tailored training recipe has the potential to push forward the attainable accuracy. This observation can be easily generalized. For example, different architecture candidates may prefer different data augmentations. As such, one promising future direction in the field of NAS is fully automated search, which jointly searches for the optimal architecture candidate and its tailored data augmentation, activation function, and training recipe in one single search to maximize the attainable accuracy.

6.2.4 Hybrid Architecture Search

Vision transformer [161] and its variants have recently emerged as powerful alternatives to deal with vision tasks, which have shown strong accuracy across various vision tasks, such as image classification, object detection, and semantic segmentation [40]. However, vision transformers, despite being able to exhibit strong accuracy, suffer from prohibitive computational complexity [40]. Therefore, one promising future direction in the field of NAS is to search for top-performing hybrid architecture candidates that can marry the best of both convolutional networks and vision transformers to achieve better accuracy-efficiency trade-offs.

6.2.5 Multi-Task Architecture Search

Previous NAS methods typically focus on searching for task-specific architecture candidates that can exhibit competitive performance in specified tasks, such as image classification [3], object detection [188], and semantic segmentation [137]. This search paradigm, however, may significantly increase the total search cost when

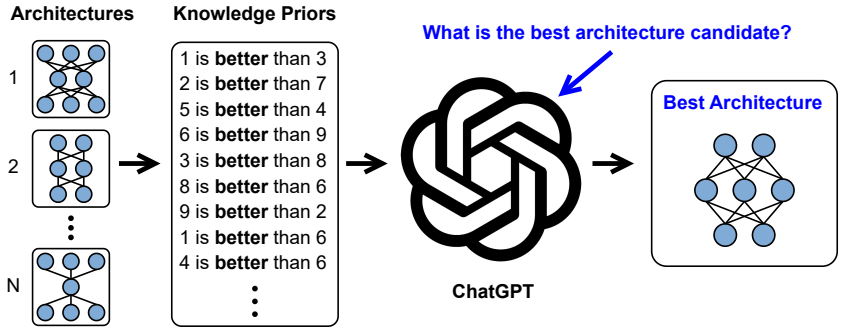

FIGURE 6.1: An intuitive overview of LLMs-enabled architecture search.

the number of tasks exponentially evolves since we have to design search strategies for each task, respectively. In addition, the architecture candidate searched for one task may not generalize to the other and vice versa [189]. As such, one promising future direction in the field of NAS is multi-task search, which automatically explores top-performing architecture candidates across multiple tasks.

6.2.6 LLMs-Enabled Architecture Search

Large language models (LLMs), thanks to their strong learning capability, have recently achieved impressive performance across various vision and language processing tasks. In light of this, one promising future direction is to leverage powerful LLMs to automatically explore the search space to navigate the optimal architecture candidate around the specified performance constraint. As shown in Figure 6.1, we can first construct a set of architecture knowledge priors and then feed these architecture knowledge priors into powerful LLMs (e.g., ChatGPT). Finally, we can simply ask powerful LLMs what is the optimal architecture candidate around the specified performance constraint. The above LLMs-enabled search paradigm may revolutionize current search techniques to innovate better architecture candidates, and more importantly, has the potential to maintain strong search efficiency since the search process can even be finished in seconds.

Appendix A

Proofs for Chapter 3

A.1 Proof of Identity Mapping

Recall that we introduce an effective identity mapping scheme to alleviate the fast convergence in terms of the skip-connect operator. Below we investigate the proposed identity mapping scheme from a theoretical perspective to show its efficacy. Without loss of generality, we take the l-th searchable layer as an example. To begin with, when the proposed identity mapping scheme is not integrated, the output of the l-th searchable layer $\mathcal{X}_{l+1}$ can be mathematically formulated as follows:

$$\begin{aligned}\mathcal{X}_{l+1} &= \sum_{k=1}^{|\mathcal{O}|} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) \\ &= \sum_{k\neq s} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) + \frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_s(\mathcal{X}_l)\end{aligned} \tag{A.1}$$

where $\mathcal{O}_s$ represents the skip-connect operator that identically transforms the input to the output (i.e., $\mathcal{O}_s(\mathcal{X}_l) = \mathcal{X}_l$). Besides, $\alpha_{l,s}$ denotes the architecture parameter assigned to the skip-connect operator $\mathcal{O}_s$. As illustrated in Fig. 8 (see supplementary materials), the supernet over-selects the skip-connect operator $\mathcal{O}_s$, which in turn makes the convergence of the supernet heavily depend on $\alpha_{l,s}$ [136], or in other words, given an input $\mathcal{X}_l$, the output of the l-th searchable layer $\mathcal{X}_{l+1}$ depends more on $\alpha_{l,s}$ than $\{\alpha_{l,k}|k \in [1, |\mathcal{O}|] \wedge k \neq s\}$. Subsequently, we integrate the proposed identity mapping scheme into SurgeNAS, and then re-formulate the output $\mathcal{X}_{l+1}$

in Eq (A.1) as follows:

$$
\begin{aligned}
\mathcal{X}_{l+1} &= \sum_{k=1}^{|\mathcal{O}|} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot (\mathcal{O}_k(\mathcal{X}_l) + \varphi \cdot \mathcal{X}_l) \\
&= \sum_{k=1}^{|\mathcal{O}|} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) + \frac{\sum_{k=1}^{|\mathcal{O}|}\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \varphi \cdot \mathcal{X}_l \\
&= \sum_{k=1}^{|\mathcal{O}|} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) + \varphi \cdot \mathcal{X}_l \\
&= \sum_{k\neq s} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) + \left(\frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} + \varphi\right) \cdot \mathcal{X}_l \\
&= \sum_{k\neq s} \frac{\exp(\alpha_{l,k})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \cdot \mathcal{O}_k(\mathcal{X}_l) + \left(\frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} + \varphi\right) \cdot \mathcal{O}_s(\mathcal{X}_l)
\end{aligned}
\tag{A.2}
$$

where φ is the weight assigned to the identity mapping operator. In SurgeNAS, φ is initialized as 2, which then linearly decays to zero at the end of the search process. In particular, we compare Eq (A.1) and Eq (A.2), and we can easily observe that the operator strength of $\mathcal{O}_s$ changes from $\frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})}$ to $\frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} + \varphi$ after introducing the proposed identity mapping scheme. Therefore, in the first half of the search phase, $\varphi \in [1, 2]$ is obviously much larger than $\frac{\exp(\alpha_{l,s})}{\sum_{k'=1}^{|\mathcal{O}|}\exp(\alpha_{l,k'})} \in [0, 1]$. As such, the output of the l-th searchable layer $\mathcal{X}_{l+1}$ is more dependent on φ than $\alpha_{l,s}$. We note that the above analysis in terms of the l-th searchable layer is able to generalize to the supernet since the supernet structure is layerwise, i.e., the output of the l-th searchable layer is the input of the $(l+1)$-th searchable layer. To summarize, integrated with the proposed identity mapping scheme, the supernet becomes less dependent and less sensitive to $\alpha_{l,s}$, thereby effectively alleviating the fast convergence of the skip-connect operator $\mathcal{O}_s$ (see Fig. 9). More importantly, the proposed identity mapping scheme does not introduce extra computation overheads because the identity mapping operator is computation-free. Besides, the proposed identity mapping scheme does not undermine the search fairness [22] since it is equally imposed on every possible operator candidate as illustrated in Eq (A.2).

List of Author's Awards, Patents, and Publications[1]

Awards

- **Publicity Paper Award**, "You Only Search Once: On Lightweight Differentiable Architecture Search for Resource-Constrained Embedded Platforms", *IEEE/ACM Design Automation Conference (DAC), 2022.*

Book Chapters

- Di Liu, Hao Kong, **Xiangzhong Luo**, Shuo Huai, and Weichen Liu, "Edge Intelligence: From Deep Learning's Perspective", *The Key Elements of Digital Factory, Elsevier, 2022.*

Journal Articles

- Shuo Huai*, Hao Kong*, **Xiangzhong Luo***, Di Liu, Ravi Subramaniam, Christian Makaya, Qian Lin, and Weichen Liu, "On Hardware-Aware Design and Optimization of Edge Intelligence", *IEEE Design & Test (D&T), 2023.*

- Hao Kong, Di Liu, Shuo Huai, **Xiangzhong Luo**, Ravi Subramaniam, Christian Makaya, Qian Lin, and Weichen Liu, "EdgeCompress: Coupling Multi-Dimensional Model Compression and Dynamic Inference for EdgeAI", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2023.*

[1]The superscript * indicates joint first authors

- **Xiangzhong Luo**, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "LightNAS: On Lightweight and Scalable Neural Architecture Search for Embedded Platforms", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.*

- **Xiangzhong Luo**, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "SurgeNAS: A Comprehensive Surgery on Hardware-Aware Differentiable Neural Architecture Search", *IEEE Transactions on Computers (TC), 2022.*

- Hui Chen, Peng Chen, **Xiangzhong Luo**, Shuo Huai, and Weichen Liu, "LAMP: Load-balanced Multipath Parallel Transmission in Point-to-point NoCs", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2022.*

- **Xiangzhong Luo**, Di Liu, Shuo Huai, Hao Kong, Hui Chen, and Weichen Liu, "Designing Efficient DNNs via Hardware-Aware Neural Architecture Search and Beyond", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2021.*

- Hui Chen, Zihao Zhang, Peng Chen, **Xiangzhong Luo**, Shiqing Li, and Weichen Liu, "MARCO: A High-performance Task Mapping and Routing Co-optimization Framework for Point-to-Point NoC-based Heterogeneous Computing Systems", *ACM Transactions on Embedded Computing Systems (TECS), as Proceedings of ACM/IEEE International Conference on Compilers, Architecture, and Synthesis of Embedded Systems (CASES), 2021.*

- Di Liu*, Hao Kong*, **Xiangzhong Luo***, Weichen Liu, and Ravi Subramaniam, "Bringing AI To Edge: From Deep Learning's Perspective", *Elsevier Neurocomputing, 2021.*

Conference Proceedings

- **Xiangzhong Luo**, Di Liu, Hao Kong, Shuo Huai, Hui Chen, Shiqing Li, Guochu Xiong, and Weichen Liu, "Pearls Hide Behind Linearity: Simplifying Deep Convolutional Networks for Embedded Hardware Systems via Linearity Grafting.", *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2024.*

- Hao Kong, Di Liu, **Xiangzhong Luo**, Shuo Huai, Ravi Subramaniam, Christian Makaya, Qian Lin, and Weichen Liu, "Towards Efficient Convolutional Neural Network for Embedded Hardware via Multi-Dimensional Pruning", *ACM/IEEE Design Automation Conference (DAC), 2023.*

- Hao Kong, **Xiangzhong Luo**, Shuo Huai, Di Liu, Ravi Subramaniam, Christian Makaya, Qian Lin, and Weichen Liu, "EMNAPE: Efficient Multi-Dimensional Neural Architecture Pruning for EdgeAI", *IEEE Design, Automation and Test in Europe (DATE), 2023. (Short Paper)*

- Hui Chen, Di Liu, Shiqing Li, Shuo Huai, **Xiangzhong Luo**, and Weichen Liu, "MUGNoC: A Software-configured Multicast-Unicast-Gather NoC for Accelerating CNN Dataflows", *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2023.*

- Shuo Huai, Di Liu, **Xiangzhong Luo**, Hui Chen, Weichen Liu, and Ravi Subramaniam, "Crossbar-Aligned & Integer-Only Neural Network Compression for Efficient In-Memory Acceleration", *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2023.*

- Shuo Huai, Di Liu, Hao Kong, **Xiangzhong Luo**, Weichen Liu, Ravi Subramaniam, Christian Makaya, and Qian Lin, "Collate: Collaborative Neural Network Learning for Latency-Critical Edge Systems", *IEEE International Conference on Computer Design (ICCD), 2022.*

- Hao Kong, Di Liu, Shuo Huai, **Xiangzhong Luo**, Weichen Liu, Ravi Subramaniam, Christian Makaya, and Qian Lin, "Smart Scissor: Coupling Spatial Redundancy Reduction and CNN Compression for Embedded Hardware", *ACM/IEEE International Conference on Computer Aided Design (ICCAD), 2022.*

- **Xiangzhong Luo**, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu, "You Only Search Once: On Lightweight Differentiable Architecture Search for Resource-Constrained Embedded Platforms", *ACM/IEEE Design Automation Conference (DAC), 2022. (Publicity Paper)*

- **Xiangzhong Luo**, Di Liu, Hui Chen, Hao Kong, Shuo Huai, and Weichen Liu, "What to Expect of Early Training Statistics? An Investigation on Hardware-Aware Neural Architecture Search: Work-in-Progress",

ACM/IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2022. (Short Paper)

- Hao Kong, Di Liu, **Xiangzhong Luo**, Weichen Liu, and Ravi Subramaniam, "HACScale: Hardware-Aware Compound Scaling for Resource-Efficient DNNs", *ACM/IEEE Asia and South Pacific Design Automation Conference (ASP-DAC), 2022.*

- **Xiangzhong Luo**, Di Liu, Shuo Huai, and Weichen Liu, "HSCoNAS: Hardware-Software Co-Design of Efficient DNNs via Neural Architecture Search", *IEEE Design, Automation and Test in Europe (DATE), 2021.*

- **Xiangzhong Luo**, Di Liu, Hao Kong, and Weichen Liu, "EdgeNAS: Discovering Efficient Neural Architectures for Edge Systems", *IEEE International Conference on Computer Design (ICCD), 2020.*

- **Xiangzhong Luo**, Luan H. K. Duong, and Weichen Liu, "Person Re-identification via Pose-aware Multi-semantic Learning", *IEEE International Conference on Multimedia and Expo (ICME), 2020.*

Bibliography

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255. Ieee, 2009. xix, xxv, 9, 11, 29, 65, 84, 85, 86, 87, 114

[2] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8697–8710, 2018. xix, 3, 4, 13, 15, 16, 19

[3] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018. xix, 3, 4, 13, 16, 18, 19, 27, 29, 31, 32, 33, 35, 37, 39, 41, 44, 48, 50, 51, 53, 54, 58, 59, 62, 63, 66, 69, 70, 73, 74, 77, 81, 85, 86, 88, 123, 124

[4] Colin White, Mahmoud Safari, Rhea Sukthanker, Binxin Ru, Thomas Elsken, Arber Zela, Debadeepta Dey, and Frank Hutter. Neural architecture search: Insights from 1000 papers. *arXiv preprint arXiv:2301.08727*, 2023. xix, 13, 18

[5] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019. xix, xxi, 4, 14, 15, 16, 17, 21, 27, 35, 40, 48, 50, 51, 66, 72, 81, 84, 85, 92, 93, 123

[6] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. xix, xxii, xxvi, 2, 10, 11, 12, 14, 26, 29, 31, 32, 33, 35, 45, 48, 53, 60, 66, 74, 84, 85, 87, 92, 93, 96, 98, 99, 102, 105, 110, 113, 114

[7] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1314–1324, 2019. xix, xxv, 4, 14, 26, 29, 31, 32, 45, 48, 53, 58, 67, 84, 93

[8] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009. xix, 14, 30, 32, 33, 45, 46, 47

[9] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019. xix, xxi, xxv, 4, 13, 15, 16, 20, 21, 27, 28, 31, 33, 35, 37, 39, 48, 50, 51, 53, 58, 59, 60, 63, 64, 65, 66, 67, 68, 69, 70, 72, 80, 81, 82, 84, 85, 86, 87, 93, 123

[10] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016. xix, 20, 35, 36, 70, 71, 74

[11] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Li, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. In *European Conference on Computer Vision*, pages 660–676. Springer, 2020. xx, 22, 40, 41, 42

[12] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. Brp-nas: Prediction-based nas using gcns. *Advances in Neural Information Processing Systems*, 33:10480–10490, 2020. xx, 21, 40, 41, 42

[13] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018. xxi, 4, 13, 14, 15, 16, 20, 21, 26, 27, 28, 31, 32, 33, 35, 36, 37, 39, 48, 50, 51, 53, 58, 59, 60, 63, 64, 66, 67, 68, 69, 70, 72, 80, 81, 84, 86, 87, 93, 123

[14] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019. xxi, 3, 4, 14, 16, 18, 21, 45, 46, 48, 49, 51, 53, 66, 67, 68, 69, 81, 82, 84, 93, 107, 108

[15] Robin Ru, Clare Lyle, Lisa Schut, Miroslav Fil, Mark van der Wilk, and Yarin Gal. Speedy performance estimation for neural architecture search. *Advances in Neural Information Processing Systems*, 34:4079–4092, 2021. xxi, 22, 75, 76, 77, 79, 83, 86, 89

[16] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020. xxi, xxv, 4, 15, 53, 55, 59, 64, 75, 76, 78, 85, 88, 89

[17] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas D Lane. Zero-cost proxies for lightweight nas. *arXiv preprint arXiv:2101.08134*, 2021. xxi, 22, 23, 76, 79, 83

[18] Joe Mellor, Jack Turner, Amos Storkey, and Elliot J Crowley. Neural architecture search without training. In *International Conference on Machine Learning*, pages 7588–7598. PMLR, 2021. xxi, 22, 23, 76, 79, 83

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. xxii, 1, 2, 6, 9, 10, 26, 32, 34, 45, 48, 51, 53, 96, 97, 98, 102, 110, 112, 113, 114, 115

[20] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018. xxv, 4, 20, 27, 31, 32, 33, 35, 36, 37, 39, 48, 50, 53, 59, 63, 66, 70, 81

[21] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1761–1770, 2019. xxv, 4, 20, 35, 50, 53, 59, 70, 81, 85, 88

[22] Xiangxiang Chu, Bo Zhang, and Ruijun Xu. Fairnas: Rethinking evaluation fairness of weight sharing neural architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12239–12248, 2021. xxv, 4, 18, 28, 33, 34, 37, 38, 39, 53, 54, 60, 70, 71, 81, 82, 128

[23] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. xxv, 67, 84, 93

[24] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan Lin. Hw-nas-bench: Hardware-aware neural architecture search benchmark. *arXiv preprint arXiv:2103.10584*, 2021. xxv, 21, 85, 88, 89, 103, 104

[25] Qian Jiang, Xiaofan Zhang, Deming Chen, Minh N Do, and Raymond A Yeh. Eh-dnas: End-to-end hardware-aware differentiable neural architecture search. *arXiv preprint arXiv:2111.12299*, 2021. xxv, 20, 69, 85, 88

[26] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. Bringing ai to edge: From deep learning's perspective. *Neurocomputing*, 485:297–320, 2022. 1, 2, 5, 9, 12, 20, 23, 58, 59, 60, 63, 74, 96, 97

[27] Han Cai, Ji Lin, Yujun Lin, Zhijian Liu, Haotian Tang, Hanrui Wang, Ligeng Zhu, and Song Han. Enable deep learning on mobile devices: Methods, systems, and applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, pages 1–50, 2022. 1, 2, 5, 9, 10, 12, 20, 23, 96, 97, 103, 104, 111, 112, 114, 116

[28] Shervin Minaee, Yuri Boykov, Fatih Porikli, Antonio Plaza, Nasser Kehtarnavaz, and Demetri Terzopoulos. Image segmentation using deep learning:

A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(7):3523–3542, 2021. 2

[29] Li Liu, Wanli Ouyang, Xiaogang Wang, Paul Fieguth, Jie Chen, Xinwang Liu, and Matti Pietikäinen. Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, 128:261–318, 2020.

[30] Wenhan Luo, Junliang Xing, Anton Milan, Xiaoqin Zhang, Wei Liu, and Tae-Kyun Kim. Multiple object tracking: A literature review. *Artificial Intelligence*, 293:103448, 2021. 1

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 2012. 1, 9

[32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 1, 9

[33] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4700–4708, 2017. 1, 9, 10, 26, 46

[34] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 1, 5, 112, 118

[35] Ying Zhang, Tao Xiang, Timothy M Hospedales, and Huchuan Lu. Deep mutual learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4320–4328, 2018. 1, 5, 112, 115

[36] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019. 1, 2, 9, 26, 27, 32, 43, 96, 104

[37] Daquan Zhou, Bingyi Kang, Xiaojie Jin, Linjie Yang, Xiaochen Lian, Zihang Jiang, Qibin Hou, and Jiashi Feng. Deepvit: Towards deeper vision transformer. *arXiv preprint arXiv:2103.11886*, 2021. 2

[38] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10012–10022, 2021.

[39] Zhengsu Chen, Lingxi Xie, Jianwei Niu, Xuefeng Liu, Longhui Wei, and Qi Tian. Visformer: The vision-friendly transformer. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 589–598, 2021. 2

[40] Kai Han, Yunhe Wang, Hanting Chen, Xinghao Chen, Jianyuan Guo, Zhenhua Liu, Yehui Tang, An Xiao, Chunjing Xu, Yixing Xu, et al. A survey on vision transformer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(1):87–110, 2022. 2, 124

[41] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 2, 11, 12, 14, 26, 32

[42] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018. 2, 11, 12, 14, 26, 48, 53

[43] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 116–131, 2018. 2, 11, 12, 14, 26, 46, 48, 53

[44] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016. 3, 4, 12, 13, 15, 16, 19, 27, 58, 123, 124

[45] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*, pages 4095–4104. PMLR, 2018. 3, 16, 17, 19, 30, 36, 53

[46] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. Pc-darts: Partial channel connections for memory-efficient architecture search. *arXiv preprint arXiv:1907.05737*, 2019. 3, 4, 13, 20, 27, 31, 32, 33, 35, 50, 51, 53

[47] Xin Chen, Lingxi Xie, Jun Wu, and Qi Tian. Progressive differentiable architecture search: Bridging the depth gap between search and evaluation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1294–1303, 2019. 4, 13, 20, 27, 31, 32, 33, 35, 39, 50

[48] Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020. 3, 4, 16, 18, 37, 39, 40, 60, 70

[49] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 4780–4789, 2019. 17, 58

[50] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *International Conference on Machine Learning*, pages 550–559. PMLR, 2018. 4, 16, 17, 18, 19

[51] Lingxi Xie, Xin Chen, Kaifeng Bi, Longhui Wei, Yuhui Xu, Lanfei Wang, Zhengsu Chen, An Xiao, Jianlong Chang, Xiaopeng Zhang, et al. Weight-sharing neural architecture search: A battle to shrink the optimization gap. *ACM Computing Surveys (CSUR)*, 54(9):1–37, 2021. 4, 13, 18, 58, 62, 81

[52] Hanwen Liang, Shifeng Zhang, Jiacheng Sun, Xingqiu He, Weiran Huang, Kechen Zhuang, and Zhenguo Li. Darts+: Improved differentiable architecture search with early stopping. *arXiv preprint arXiv:1909.06035*, 2019. 4, 13, 20, 27, 28, 32, 34, 35

[53] Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. Fair darts: Eliminating unfair advantages in differentiable architecture search. In *European Conference on Computer Vision*, pages 465–480. Springer, 2020.

[54] Xiangxiang Chu, Xiaoxing Wang, Bo Zhang, Shun Lu, Xiaolin Wei, and Junchi Yan. Darts-: robustly stepping out of performance collapse without indicators. *arXiv preprint arXiv:2009.01027*, 2020. 20

[55] Kaifeng Bi, Changping Hu, Lingxi Xie, Xin Chen, Longhui Wei, and Qi Tian. Stabilizing darts with amended gradient estimation on architectural parameters. *arXiv preprint arXiv:1910.11831*, 2019. 28, 31, 33, 34

[56] Kaifeng Bi, Lingxi Xie, Xin Chen, Longhui Wei, and Qi Tian. Gold-nas: Gradual, one-level, differentiable. *arXiv preprint arXiv:2007.03331*, 2020. 4, 20, 28, 31, 33, 34

[57] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pages 7105–7114. PMLR, 2019. 4

[58] Shen Yan, Colin White, Yash Savani, and Frank Hutter. Nas-bench-x11 and the power of learning curves. *Advances in Neural Information Processing Systems*, 34:22534–22549, 2021. 4, 22

[59] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. Single-path nas: Designing hardware-efficient convnets in less than 4 hours. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 481–497. Springer, 2019. 4, 16, 20, 48, 59, 60, 63, 72, 80, 82, 84

[60] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Bowen Shi, Qi Tian, and Hongkai Xiong. Latency-aware differentiable neural architecture search. *arXiv preprint arXiv:2001.06392*, 2020. 59, 60, 64, 66, 69, 70, 72, 81

[61] Yuhong Li, Cong Hao, Xiaofan Zhang, Xinheng Liu, Yao Chen, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020. 4, 21, 26, 27, 28, 33, 35, 37, 39, 50

[62] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan Yuille, and Jianchao Yang. Atomnas: Fine-grained end-to-end neural architecture search. *arXiv preprint arXiv:1912.09640*, 2019. 4

[63] Han Cai, Tianzhe Wang, Zhanghao Wu, Kuan Wang, Ji Lin, and Song Han. On-device image classification with proxyless neural architecture search and quantization-aware fine-tuning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019. 5

[64] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *Advances in Neural Information Processing Systems*, 28, 2015. 5

[65] Fengfu Li, Bin Liu, Xiaoxing Wang, Bo Zhang, and Junchi Yan. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. 5

[66] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 5, 26, 112

[67] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016. 5, 23

[68] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE Cnternational Conference on Computer Vision*, pages 2736–2744, 2017. 5, 23

[69] Sara Elkerdawy, Mostafa Elhoushi, Abhineet Singh, Hong Zhang, and Nilanjan Ray. One-shot layer-wise accuracy approximation for layer pruning. In *2020 IEEE International Conference on Image Processing (ICIP)*, pages 2940–2944. IEEE, 2020. 5, 6, 97, 98, 102, 111, 112, 116

[70] Artur Jordao, Maiko Lie, and William Robson Schwartz. Discriminative layer pruning for convolutional neural networks. *IEEE Journal of Selected Topics in Signal Processing*, pages 828–837, 2020. 6, 24, 97, 98, 102, 111, 112

[71] Shuochao Yao, Yiran Zhao, Huajie Shao, ShengZhong Liu, Dongxin Liu, Lu Su, and Tarek Abdelzaher. Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291, 2018. 6, 97, 111

[72] Ke Zhang and Guangzhe Liu. Layer pruning for obtaining shallower resnets. *IEEE Signal Processing Letters*, 29:1172–1176, 2022. 6, 24, 97, 98, 102, 111, 112

[73] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015. 9

[74] Jiemin Fang, Yuzhu Sun, Qian Zhang, Yuan Li, Wenyu Liu, and Xinggang Wang. Densely connected search space for more flexible neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10628–10637, 2020. 9, 35, 36, 50, 51

[75] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 9, 11, 12, 26

[76] Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. Ghostnet: More features from cheap operations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1580–1589, 2020. 11

[77] Daquan Zhou, Qibin Hou, Yunpeng Chen, Jiashi Feng, and Shuicheng Yan. Rethinking bottleneck structure for efficient mobile network design. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*, pages 680–697. Springer, 2020. 11

[78] Gao Huang, Shichen Liu, Laurens Van der Maaten, and Kilian Q Weinberger. Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2752–2761, 2018. 11

[79] Le Yang, Haojun Jiang, Ruojin Cai, Yulin Wang, Shiji Song, Gao Huang, and Qi Tian. Condensenet v2: Sparse feature reactivation for deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3569–3578, 2021. 11

[80] Yehui Tang, Kai Han, Jianyuan Guo, Chang Xu, Chao Xu, and Yunhe Wang. Ghostnetv2: enhance cheap operation with long-range attention. *Advances in Neural Information Processing Systems*, 35:9969–9982, 2022. 11, 12

[81] Xiangzhong Luo, Di Liu, Hao Kong, and Weichen Liu. Edgenas: Discovering efficient neural architectures for edge systems. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 288–295. IEEE, 2020. 13, 21

[82] Xiangzhong Luo, Di Liu, Shuo Huai, and Weichen Liu. Hsconas: Hardware-software co-design of efficient dnns via neural architecture search. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 418–421. IEEE, 2021. 14, 21

[83] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. Fast hardware-aware neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 692–693, 2020. 14, 18

[84] Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu. Surgenas: a comprehensive surgery on hardware-aware differentiable neural architecture search. *IEEE Transactions on Computers*, 72(4):1081–1094, 2022. 14

[85] Xiangzhong Luo, Di Liu, Hao Kong, Shuo Huai, Hui Chen, and Weichen Liu. You only search once: On lightweight differentiable architecture search for resource-constrained embedded platforms. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 475–480, 2022. 14, 21, 61, 113

[86] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016. 15

[87] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992. 15

[88] Geoffrey F Miller, Peter M Todd, and Shailesh U Hegde. Designing neural networks using genetic algorithms. In *ICGA*, volume 89, pages 379–384, 1989. 17

[89] Peter J Angeline, Gregory M Saunders, and Jordan B Pollack. An evolutionary algorithm that constructs recurrent neural networks. *IEEE transactions on Neural Networks*, 5(1):54–65, 1994.

[90] Dario Floreano, Peter Dürr, and Claudio Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1:47–62, 2008.

[91] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002. 17

[92] Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, and Shaolei Ren. One proxy device is enough for hardware-aware neural architecture search. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–34, 2021. 18, 21, 97, 103

[93] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, Ruoming Pang, and Quoc Le. Bignas: Scaling up neural architecture search with big single-stage models. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VII 16*, pages 702–717. Springer, 2020. 18

[94] Shan You, Tao Huang, Mingmin Yang, Fei Wang, Chen Qian, and Changshui Zhang. Greedynas: Towards fast one-shot nas with greedy supernet. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1999–2008, 2020. 18

[95] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search. *arXiv preprint arXiv:2101.09336*, 2021. 18, 21

[96] Pengfei Hou, Ying Jin, and Yukang Chen. Single-darts: towards stable architecture search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 373–382, 2021. 20

[97] Yibo Hu, Xiang Wu, and Ran He. Tf-nas: Rethinking three search freedoms of latency-constrained differentiable neural architecture search. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XV 16*, pages 123–139. Springer, 2020. 20, 81, 84

[98] Jaeseong Lee, Duseok Kang, and Soonhoi Ha. S3nas: Fast npu-aware neural architecture search methodology. *arXiv preprint arXiv:2009.02009*, 2020. 20

[99] Xiangzhong Luo, Di Liu, Shuo Huai, Hao Kong, Hui Chen, and Weichen Liu. Designing efficient dnns via hardware-aware neural architecture search and beyond. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1799–1812, 2021. 21

[100] Kevin Alexander Laube, Maximus Mutschler, and Andreas Zell. What to expect of hardware metric predictors in nas. In *International Conference on Automated Machine Learning*, pages 13–1. PMLR, 2022. 21, 97, 104

[101] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. Hardware-adaptive efficient latency prediction for nas via meta-learning. *Advances in Neural Information Processing Systems*, 34:27016–27028, 2021. 21, 81

[102] Renqian Luo, Xu Tan, Rui Wang, Tao Qin, Enhong Chen, and Tie-Yan Liu. Accuracy prediction with non-neural model for neural architecture search. *arXiv preprint arXiv:2007.04785*, 2020. 22

[103] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34:28454–28469, 2021. 22

[104] Bert Moons, Parham Noorzad, Andrii Skliar, Giovanni Mariani, Dushyant Mehta, Chris Lott, and Tijmen Blankevoort. Distilling optimal neural networks: Rapid search in diverse spaces. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12229–12238, 2021. 22

[105] Dan Zhao, Nathan C Frey, Vijay Gadepally, and Siddharth Samsi. Loss curve approximations for fast neural architecture ranking & training elasticity estimation. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 715–723. IEEE, 2022. 22

[106] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. Learning curve prediction with bayesian neural networks. In *International Conference on Learning Representations*, 2016.

[107] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017. 22

[108] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-fourth International Joint Conference on Artificial Intelligence*, 2015. 22

[109] Vasco Lopes, Saeid Alirezazadeh, and Luís A Alexandre. Epe-nas: Efficient performance estimation without training for neural architecture search. In *International Conference on Artificial Neural Networks*, pages 552–563. Springer, 2021. 22

[110] Jack Turner, Elliot J Crowley, Michael O'Boyle, Amos Storkey, and Gavin Gray. Blockswap: Fisher-guided block substitution for network compression on a budget. *arXiv preprint arXiv:1906.04113*, 2019. 22

[111] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. *arXiv preprint arXiv:2002.07376*, 2020. 22

[112] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip HS Torr. Snip: Single-shot network pruning based on connection sensitivity. *arXiv preprint arXiv:1810.02340*, 2018. 22

[113] Hidenori Tanaka, Daniel Kunin, Daniel L Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow. *Advances in Neural Information Processing Systems*, 33:6377–6389, 2020. 22

[114] Ming Lin, Pichao Wang, Zhenhong Sun, Hesen Chen, Xiuyu Sun, Qi Qian, Hao Li, and Rong Jin. Zen-nas: A zero-shot nas for high-performance image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 347–356, 2021. 22

[115] Yash Akhauri, Juan Munoz, Nilesh Jain, and Ravishankar Iyer. Eznas: Evolving zero-cost proxies for neural architecture scoring. *Advances in Neural Information Processing Systems*, 35:30459–30470, 2022. 23

[116] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on imagenet in four gpu hours: A theoretically inspired perspective. *arXiv preprint arXiv:2102.11535*, 2021. 23

[117] Huan Xiong, Lei Huang, Mengyang Yu, Li Liu, Fan Zhu, and Ling Shao. On the number of linear regions of convolutional neural networks. In *International Conference on Machine Learning*, pages 10514–10523. PMLR, 2020. 22

[118] Lechao Xiao, Jeffrey Pennington, and Samuel Schoenholz. Disentangling trainability and generalization in deep neural networks. In *International Conference on Machine Learning*, pages 10462–10472. PMLR, 2020. 22

[119] Yang He and Lingao Xiao. Structured pruning for deep convolutional neural networks: A survey. *arXiv preprint arXiv:2303.00566*, 2023. 23, 97

[120] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016. 23

[121] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. *arXiv preprint arXiv:1808.06866*, 2018. 23

[122] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019. 23

[123] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 5058–5066, 2017. 23

[124] Xiaohan Ding, Guiguang Ding, Yuchen Guo, Jungong Han, and Chenggang Yan. Approximated oracle filter pruning for destructive cnn width optimization. In *International Conference on Machine Learning*, pages 1607–1616. PMLR, 2019. 23

[125] Zhonghui You, Kun Yan, Jinmian Ye, Meng Ma, and Ping Wang. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 32, 2019. 23

[126] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018. 23, 24, 97, 98, 102, 111, 112, 114, 115, 116

[127] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Xin Yang, Kwang-Ting Cheng, and Jian Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3296–3305, 2019. 24, 60, 76, 80, 82, 114, 115, 116

[128] Shaopeng Guo, Yujie Wang, Quanquan Li, and Junjie Yan. Dmcp: Differentiable markov channel pruning for neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1539–1547, 2020. 23, 24, 97, 98, 102, 111, 112, 116

[129] Sara Elkerdawy, Mostafa Elhoushi, Abhineet Singh, Hong Zhang, and Nilanjan Ray. To filter prune, or to layer prune, that is the question. In *Proceedings of the Asian Conference on Computer Vision*, 2020. 24

[130] Shi Chen and Qi Zhao. Shallowing deep networks: Layer-wise pruning based on feature representations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 41(12):3048–3056, 2018.

[131] Hui Tang, Yao Lu, and Qi Xuan. Sr-init: An interpretable layer pruning method. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1–5. IEEE, 2023.

[132] Artur Jordao, George Correa de Araujo, Helena de Almeida Maia, and Helio Pedrini. When layers play the lottery, all tickets win at initialization. *arXiv preprint arXiv:2301.10835*, 2023. 24

[133] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018. 24

[134] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019. 26, 31, 32, 35, 47, 48, 53, 63, 84, 87, 93

[135] Weiwen Jiang, Xinyi Zhang, Edwin H-M Sha, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019. 26

[136] Arber Zela, Thomas Elsken, Tonmoy Saikia, Yassine Marrakchi, Thomas Brox, and Frank Hutter. Understanding and robustifying differentiable architecture search. *arXiv preprint arXiv:1909.09656*, 2019. 27, 28, 32, 34, 54, 127

[137] Albert Shaw, Daniel Hunter, Forrest Landola, and Sammy Sidhu. Squeezenas: Fast neural architecture search for faster semantic segmentation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019. 27, 28, 33, 37, 39, 51, 124

[138] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016. 36, 72

[139] Hao Tan, Ran Cheng, Shihua Huang, Cheng He, Changxiao Qiu, Fan Yang, and Ping Luo. Relativenas: Relative neural architecture search via slow-fast learning. *IEEE Transactions on Neural Networks and Learning Systems*, 2021. 39

[140] Zheyu Yan, Weiwen Jiang, Xiaobo Sharon Hu, and Yiyu Shi. Radars: memory efficient reinforcement learning aided differentiable neural architecture search. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 128–133. IEEE, 2022. 40

[141] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. 40, 41

[142] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013. 44, 74

[143] Shoukang Hu, Sirui Xie, Hehui Zheng, Chunxiao Liu, Jianping Shi, Xunying Liu, and Dahua Lin. Dsnas: Direct neural architecture search without parameter retraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12084–12092, 2020. 53

[144] Arash Vahdat, Arun Mallya, Ming-Yu Liu, and Jan Kautz. Unas: Differentiable architecture search meets reinforcement learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11266–11275, 2020. 59, 60, 64, 66, 81, 84

[145] Jiahui Yu and Thomas Huang. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv preprint arXiv:1903.11728*, 2019. 60, 74, 75, 76, 80, 82

[146] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, and Trevor Darrell. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018. 60, 74

[147] Zhengsu Chen, Jianwei Niu, Lingxi Xie, Xuefeng Liu, Longhui Wei, and Qi Tian. Network adjustment: Channel search guided by flops utilization ratio. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10658–10667, 2020. 60, 75

[148] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. *arXiv preprint arXiv:1812.08928*, 2018. 60, 76, 82

[149] Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1803–1811, 2019. 60, 61, 76, 82

[150] Xuanyi Dong and Yi Yang. Network pruning via transformable architecture search. *Advances in Neural Information Processing Systems*, 32, 2019. 60, 76, 80, 82

[151] J. Yu and et al. https://github.com/JiahuiYu/slimmable_networks/issues. 61

[152] Aitor Lewkowycz, Yasaman Bahri, Ethan Dyer, Jascha Sohl-Dickstein, and Guy Gur-Ari. The large learning rate phase of deep learning: the catapult mechanism. *arXiv preprint arXiv:2003.02218*, 2020. 77

[153] Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik. Hardcore-nas: Hard constrained differentiable neural architecture search. In *International Conference on Machine Learning*, pages 7979–7990. PMLR, 2021. 81, 87

[154] Gabriel Bender, Hanxiao Liu, Bo Chen, Grace Chu, Shuyang Cheng, Pieter-Jan Kindermans, and Quoc V Le. Can weight sharing outperform random architecture search? an investigation with tunas. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14323–14332, 2020. 81

[155] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12965–12974, 2020. 83

[156] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1586–1595, 2018.

[157] Matteo Risso, Alessio Burrello, Francesco Conti, Lorenzo Lamberti, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Lightweight neural architecture search for temporal convolutional networks at the edge. *IEEE Transactions on Computers*, pages 744–758, 2022.

[158] Matteo Risso, Alessio Burrello, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. Multi-complexity-loss dnas for energy-efficient and memory-constrained deep neural networks. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 1–6, 2022.

[159] Igor Fedorov, Ramon Matas, Hokchhay Tann, Chuteng Zhou, Matthew Mattina, and Paul Whatmough. Udc: Unified dnas for compressible tinyml models. *arXiv preprint arXiv:2201.05842*, 2022. 83

[160] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 83

[161] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 83, 124

[162] Liam Li and Ameet Talwalkar. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, pages 367–377. PMLR, 2020. 89

[163] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016. 92

[164] Ting-Wu Chin, Ruizhou Ding, Cha Zhang, and Diana Marculescu. Towards efficient model compression via learned global ranking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1518–1528, 2020. 97, 98, 102, 111, 112, 116

[165] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1529–1538, 2020. 116

[166] Mao Ye, Chengyue Gong, Lizhen Nie, Denny Zhou, Adam Klivans, and Qiang Liu. Good subnetworks provably exist: Pruning via greedy forward selection. In *International Conference on Machine Learning*, pages 10820–10830. PMLR, 2020. 116

[167] Sixing Yu, Arya Mazaheri, and Ali Jannesari. Auto graph encoder-decoder for neural network pruning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6362–6372, 2021. 116

[168] Manoj Alwani, Yang Wang, and Vashisht Madhavan. Decore: Deep compression with reinforcement learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12349–12359, 2022. 97, 98, 102, 111, 112, 116

[169] Nandan Kumar Jha, Zahra Ghodsi, Siddharth Garg, and Brandon Reagen. Deepreduce: Relu reduction for fast private inference. In *International Conference on Machine Learning*, pages 4839–4849. PMLR, 2021. 101, 102

[170] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019. 102

[171] Jin-Dong Dong, An-Chieh Cheng, Da-Cheng Juan, Wei Wei, and Min Sun. Dpp-net: Device-aware progressive search for pareto-optimal neural architectures. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 517–531, 2018. 103

[172] Yehui Tang, Yunhe Wang, Yixing Xu, Hanting Chen, Boxin Shi, Chao Xu, Chunjing Xu, Qi Tian, and Chang Xu. A semi-supervised assessor of neural architectures. In *proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1810–1819, 2020. 105, 107

[173] Yuqiao Liu, Yehui Tang, and Yanan Sun. Homogeneous architecture augmentation for neural predictor. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 12249–12258, 2021. 105, 107

[174] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, pages 1126–1135. PMLR, 2017. 105, 106

[175] Mu Hu, Junyi Feng, Jiashen Hua, Baisheng Lai, Jianqiang Huang, Xiaojin Gong, and Xian-Sheng Hua. Online convolutional re-parameterization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 568–577, 2022. 109

[176] Tianlong Chen, Huan Zhang, Zhenyu Zhang, Shiyu Chang, Sijia Liu, Pin-Yu Chen, and Zhangyang Wang. Linearity grafting: Relaxed neuron pruning helps certifiable robustness. In *International Conference on Machine Learning*, pages 3760–3772. PMLR, 2022. 112

[177] Ruizhou Ding, Zeye Liu, Ting-Wu Chin, Diana Marculescu, and Ronald D Blanton. Flightnns: Lightweight quantized deep neural networks for fast and accurate inference. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019. 112

[178] Azat Azamat, Faaiz Asim, and Jongeun Lee. Quarry: Quantization-based adc reduction for reram-based deep neural network accelerators. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–7. IEEE, 2021.

[179] Sitao Huang, Aayush Ankit, Plinio Silveira, Rodrigo Antunes, Sai Rahul Chalamalasetti, Izzat El Hajj, Dong Eun Kim, Glaucimar Aguiar, Pedro

Bruel, Sergey Serebryakov, et al. Mixed precision quantization for reram-based dnn inference accelerators. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pages 372–377, 2021.

[180] Qingcheng Xiao and Yun Liang. Zac: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2019. 112

[181] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129:1789–1819, 2021. 112

[182] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019. 115

[183] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 115

[184] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019. 115, 124

[185] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017. 124

[186] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, et al. Fbnetv3: Joint architecture-recipe search using predictor pretraining. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16276–16285, 2021. 124

[187] Xuanyi Dong, Mingxing Tan, Adams Wei Yu, Daiyi Peng, Bogdan Gabrys, and Quoc V Le. Autohas: Efficient hyperparameter and architecture search. *arXiv preprint arXiv:2006.03656*, 2020. 124

[188] Yunyang Xiong, Hanxiao Liu, Suyog Gupta, Berkin Akin, Gabriel Bender, Yongzhe Wang, Pieter-Jan Kindermans, Mingxing Tan, Vikas Singh, and Bo Chen. Mobiledets: Searching for object detection architectures for mobile accelerators. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3825–3834, 2021. 124

[189] Bichen Wu, Chaojian Li, Hang Zhang, Xiaoliang Dai, Peizhao Zhang, Matthew Yu, Jialiang Wang, Yingyan Lin, and Peter Vajda. Fbnetv5: Neural architecture search for multiple tasks in one run. *arXiv preprint arXiv:2111.10007*, 2021. 125