Title
Towards Ultra-Efficient Machine Learning for Edge Inference

Author
Lu, Bingqian

Publication Date
2022

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Towards Ultra-Efficient Machine Learning for Edge Inference

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Bingqian Lu

June 2022

Dissertation Committee:

Dr. Shaolei Ren, Chairperson
Dr. Daniel Wong
Dr. Jiasi Chen

The Dissertation of Bingqian Lu is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The past four years at UC Riverside have been a fruitful adventure for me, not only in my academic path, but also my personal life. During the pursuit of Ph.D. and preparation of this thesis, I have received a lot of invaluable help and tremendous supports from my advisor, committee members, labmates, friends, and family. I would like to take this opportunity to express my heartfelt gratitude to these resourceful, sincere, and brilliant people.

First and foremost, I would like to give my most sincere and deepest gratitude to my advisor Professor Shaolei Ren. It has been an honor to have such a caring mentor, dedicated researcher, and knowledgeable instructor to lead me over the years. Dr. Ren provides me continuous guidance and support on my research topic, as well as the learning methodology to be a precise, profound, and creative researcher. Without his painstaking teaching and insightful advice, the completion of this thesis would not have been possible.

Also, I would like to thank my dissertation committee members: Professor Daniel Wong and Professor Jiasi Chen, for their helpful advice and feedback in my oral qualify and final defense exams.

I have been extremely fortunate to work with a group of wonderful labmates at the Nonlinear Computing Lab: Jianyi Yang, Luting Yang, Shixiong Qi, Zhihui Shao, Fangfang Yang, Pengfei Li, and Yejia Liu, who have made my time in Riverside fun and memorable. It is they who make it possible for me to enjoy research and life at the same time. Special thanks to Jianyi Yang for his help in my early research projects. In particular, I would like to thank Dr. Mohammad Atiqul Islam for providing me with initial guidance when I first joined the lab.

I would like to thank my fellow researchers and collaborators: Dr. Lydia Y Chen, Dr. Jie Xu, Dr. Yiyu Shi, Dr. Weiwen Jiang, and Zheyu Yan for their great work, thoughts and ideas.

Last but most importantly, I am so grateful to my parents for their persistent support and encouragement, who have always been my strongest backing with unconditional love in my entire life. My days as a graduate student were made enjoyable in large part due to the many friends and groups that became a part of my life: Ajia, Michelle, Ye, and the DL family. This dissertation is dedicated to them.

Funding Acknowledgment. I appreciate the funding support from National Science Foundation under grants CNS-1910208 and CNS-2007115, UC Riverside Dean's Distinguished Fellowship, and UC Riverside TAShip.

Publication Acknowledgment. The text of this dissertation, in part or in full, is a reprint of the material as it appears in the list of publications below. The co-author Dr. Shaolei Ren listed in that publication directed and supervised the research which forms the basis for this dissertation.

1. Bingqian Lu, Jianyi Yang, Shaolei Ren, *Scaling Up Deep Neural Network Optimization for Edge Inference*, ACM/IEEE Symposium on Edge Computing (SEC, poster), Nov. 2020 [78].

   Added to the dissertation as Chapter 1. Bingqian Lu is the major contributor of the publication. Jianyi Yang assisted in the theoretical formulation and analysis.

2. Bingqian Lu, Jianyi Yang, Jie Xu, Shaolei Ren, *Improving QoE of Deep Neural Network Inference on Edge Devices: A Bandit Approach*, IEEE Internet of Things Journal.

Added to the dissertation as Chapter 2. Bingqian Lu is the major contributor of the publication. Jianyi Yang assisted in the theoretical formulation and analysis. Dr. Jie Xu assisted in the research idea proposal. Besides, a preliminary work of Chapter 2 is introduced in [75].

3. Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, Shaolei Ren, *One Proxy Device Is Enough for Hardware-Aware Neural Architecture Search*, ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), Jun. 2022 [77].

   Added to the dissertation as Chapter 3. Bingqian Lu is the major contributor of the publication. Jianyi Yang assisted in the theoretical formulation. Dr. Weiwen Jiang and Dr. Yiyu Shi assisted in the explanation of latency monotonicity.

4. Bingqian Lu, Zheyu Yan, Yiyu Shi, Shaolei Ren, *A Semi-Decoupled Approach to Fast and Optimal Hardware-Software Co-Design of Neural Accelerators*, tinyML Research Symposium (tinyML), Feb. 2022 [76].

   Added to the dissertation as Chapter 4. Bingqian Lu is the major contributor of the publication. Zheyu Yan and Dr. Yiyu Shi assisted in the background of hardware design of neural accelerators.

To my parents.

ABSTRACT OF THE DISSERTATION

Towards Ultra-Efficient Machine Learning for Edge Inference

by

Bingqian Lu

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, June 2022
Dr. Shaolei Ren, Chairperson

Deep neural networks (DNNs) have been increasingly deployed on and integrated with edge devices, such as mobile phones, drones, robots and wearables. To run DNN inference directly on edge devices (a.k.a. edge inference) with a satisfactory performance, optimizing the DNN design (e.g., neural architecture and quantization policy) is crucial. However, designing an optimal DNN for even a single edge device often needs repeated design iterations and is non-trivial. Worse yet, DNN model developers commonly need to serve extremely diverse edge devices. Therefore, it has become crucially important to scale up the optimization of DNNs for edge inference using automated approaches. In this dissertation, we come up with several solutions to scalably and efficiently optimize the DNN design for *diverse* edge devices, with increasingly flexible design consideration. Firstly, consider the fact that a large number of diverse DNN models can be generated by navigating through the design space in terms of different neural architectures and compression techniques, we look into the problem that how to select the best DNN model out of many choices for each individual edge device. We propose a novel automated and user-centric DNN selection en-

gine, called Aquaman, which leverages users' Quality of Experience (QoE) feedback to guide DNN selection decisions. The core of Aquaman is a machine learning-based QoE predictor which is continuously updated online, and neural bandit learning to balance exploitation and exploration. However, the assumption of a pre-existing DNN model pool in Aquaman is essentially limited and may not suit any given edge device's best interest. Therefore, we take into consideration the design freedom of neural architectures by resorting to hardware-aware neural architecture search (NAS) for optimizing the DNN design for a given target device. NAS can thoroughly explore the model architecture search space, and automatically discover the optimal combination of building blocks, namely a model, for any target device. A key requirement of efficient hardware-aware NAS is the fast evaluation of inference latencies in order to rank different architectures. While building a latency predictor for each target device has been commonly used in state of the art, this is a very time-consuming process, lacking scalability in the presence of extremely diverse devices. We address the scalability challenge by exploiting latency monotonicity – the architecture latency rankings on different devices are often correlated. When strong latency monotonicity exists, we can re-use architectures searched for **one proxy** device on new target devices, without losing optimality. In the absence of strong latency monotonicity, we also propose an efficient proxy adaptation technique to significantly boost the latency monotonicity. Our results highlight that, by using just one proxy device, we can find almost the same Pareto-optimal architectures as the existing per-device NAS, while avoiding the prohibitive cost of building a latency predictor for each device, reducing the cost of hardware-aware NAS from $O(N)$ to $O(1)$. Further, besides the design flexibility of neural architectures brought by NAS (i.e. software design), exploring the hardware design space such as optimizing hardware accelerators built on FPGA or ASIC, as well as the corresponding dataflows (e.g., scheduling DNN computations and mapping them on hardware), is also critical for speeding up DNN execution. While hardware-software co-design can further optimize DNN performance, it also exponentially enlarges the search space to practical infinity, presenting significant challenges. By settling in-between the fully-decoupled approach and the fully-coupled hardware-software co-design approach, we propose a new **semi-decoupled** approach to reduce the size of the total co-search space by orders of magnitude, yet without losing design optimality. Our approach again builds on the latency and energy monotonicity – neural architectures' ranking orders in terms of inference latency and energy consumption on different accelerators are highly correlated. Our results confirm that strong latency and energy monotonicity exist among different accelerator designs. More importantly, by using one candidate accelerator as the proxy and obtaining its small set of optimal architectures, we can reuse the same architecture set for other accelerator candidates during the hardware search stage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we first introduce the background and motivation of optimizing DNN for edge inference, and state-of-the-art works and their limitations. Then we mathematically formulate the DNN optimization problem in Section 1.3, and propose two general approaches or frameworks to solve the problem in Sections 1.4 and 1.5, where **Chapter 3** and **Chapter 4** in this dissertation fall into category of the first framework. At the end of the chapter, we summarize main contribution of this thesis and briefly introduce the following chapters.

1.1 Background and Motivation

Deep neural networks (DNNs) have been increasingly deployed on and integrated with edge devices, such as mobile phones, drones, robots and wearables. Compared to cloud-based inference, running DNN inference directly on edge devices (a.k.a. *edge inference*) has several major advantages, including being free from the network connection requirement, saving bandwidths and better protecting user privacy as a result of local data processing. For example, it is very common to include one or multiple DNNs in today's mobile apps [121].

To achieve a satisfactory user experience for edge inference, an appropriate DNN design is needed to optimize a multi-objective performance metric, e.g., good accuracy while keeping the latency and energy consumption low. A complex DNN model involves multi-layer perception with up to billions of parameters, imposing a stringent computational and memory requirement that is often too prohibitive for edge devices. Thus, the DNN models running on an edge device must be judiciously optimized using, e.g., neural architecture search (NAS) and model compression [20, 23, 24, 71, 79, 106, 115].

The DNN design choices we focus on in this chapter mainly refer to the neural architecture and compression scheme (e.g., pruning and quantization policy), which constitute an exponentially large space. Note that other DNN design parameters, such as learning rate and choice of optimizer for DNN training, can also be included into the proposed framework. For example, if we want to consider learning rate and DNN architecture optimization, the accuracy predictor can take the learning rate and architecture as the input and be trained by using different DNN samples with distinct architectures and learning rates.

Given different design choices, DNN models can exhibit dramatically different performance tradeoffs in terms of various important performance metrics (e.g., accuracy, latency, energy and robustness). In general, there is not a single DNN model that performs Pareto optimally on all edge devices. For example, with the same DNN model in Facebook's app, the resulting latencies on different devices can vary significantly [121]. Thus, device-aware DNN optimization is mandated [79, 81, 112, 121].

Designing an optimal DNN for even a single edge device often needs repeated design iterations and is non-trivial [28, 120]. Worse yet, DNN model developers often need to serve extremely diverse edge devices. For example, the DNN-powered voice assistant application developed by a third party can be used by many different edge device vendors, and Facebook's DNN model for style transfer is run on billions of mobile devices, more than half of which still use CPUs designed in 2012 or before (shown in Fig. 1.1) [121]. In the mobile market alone, there are thousands of system-on-chips (SoCs) available. Only top 30 SoCs can each take up more than 1% of the share, and they collectively account for 51% of the whole market [121]. Thus, the practice of repeatedly optimizing DNN models, once for each edge device, can no longer meet the demand in view of the extremely diverse edge devices.

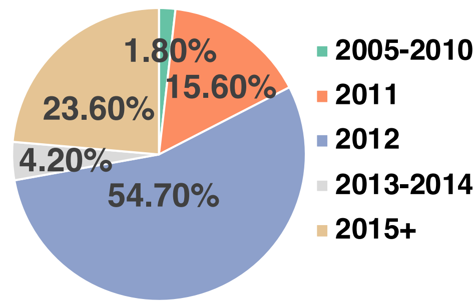


Figure 1.1: Statistics of the year mobile CPUs are designed as of late 2018 [121].

Therefore, it has become crucially important to scale up the optimization of DNNs for edge inference using automated approaches.

1.2 State of the Art and Limitations

Network architecture is a key design choice that affects the resulting performance of DNN models on edge devices. Due to the huge space for network architectures, traditional hand-tuned architecture designs can take months or even longer to train a DNN with a satisfactory performance [42, 133]. Thus, they have become obsolete and been replaced with automated approaches [106]. Nonetheless, the early NAS approaches often require training each DNN candidate (albeit usually on a small proxy dataset), which hence still results in a high complexity and search time. To address this issue, DNN optimization and training need to be decoupled. For example, the current "once-for-all" technique can generate nearly unlimited ($> 10^{19}$) DNN models of different architectures all at once [23]. Consequently, DNN model developers can now focus on the optimization of network architecture, without having to train a DNN for each candidate architecture. Thus, instead of DNN training, we consider scalability of optimizing DNN designs with a focus on the neural architecture.

NAS on a single target device cannot result in the optimal DNN model for all other devices, motivating device-aware NAS. In general, the device-aware NAS process is guided by an objective function, e.g., $accuracy\_loss + weight_1 * energy + weight_2 * latency$. Thus, it is crucial to efficiently evaluate the resulting inference accuracy/latency/energy performance given a DNN candidate [80, 87, 95, 101, 114]. Towards this end, proxy models have been leveraged to calculate latency/energy for each candidate, but they are not very accurate on all devices [120]. Alternatively, actual latency measurement on real devices for each candidate is also considered, but it is time-consuming [106].

More recently, performance predictors or lookup tables have been utilized to assist with NAS (and model compression) [20, 79, 80, 87, 95, 101, 104, 114, 115]: train a machine learning model or build a lookup table to estimate the resulting accuracy/latency/energy performance for a candidate DNN design on the target device. Therefore, by using search techniques aided by performance predictors or lookup tables, an optimal DNN can be identified out of numerous candidates for a target edge device without actually deploying or

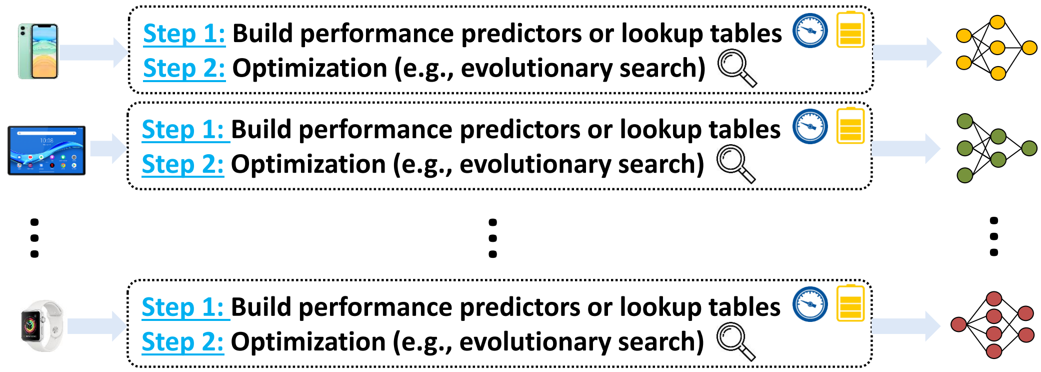


Figure 1.2: The existing device-aware DNN optimization (i.e., once for a single device) [23, 34, 115].

running each candidate DNN on the device [23, 115].

Nonetheless, as illustrated in Fig. 1.2, the existing latency/energy predictors or lookup tables [23, 24, 34, 87, 101, 115] are *device-specific* and only take the DNN features as input to predict the inference latency/energy performance on a particular target device. For example, according to [24], the average inference latencies of 4k randomly selected DNNs are measured on a mobile device and then used to train an average latency predictor for that specific device (plus additional 1k samples for testing). Assuming that each measurement takes 30 seconds, it takes a total of 40+ hours to just collect training and testing samples in order to build the latency predictor for one single device, let alone the additional time spent for latency predictor training and other performance predictors. Likewise, to estimate the inference latency, 350K operator-level latency records are profiled to construct a lookup table in [34], which is inevitably time-consuming. Clearly, building performance predictors or lookup tables incurs a significant overhead by itself [23, 24, 34, 87, 101, 115].

More crucially, without taking into account the device features, the resulting performance predictors or lookup tables only provide good predictions for the individual device on which the performance is measured. For example, as shown in Fig. 4 in [34], the same convolution operator can result in dramatically different latencies on two different devices — Samsung S8 with Snapdragon 835 mobile CPU and Hexagon v62 DSP with 800 MHz frequency.

In addition, the optimizer (e.g., a simple evolutionary search-based algorithm or more advanced exploration strategies [80, 87, 95, 101]) to identify an optimal architecture for each device also takes non-negligible time or CPU-hours. For example, even with limited rounds of evolutionary search, 30 minutes to several hours are needed by the DNN optimization process for each device [23, 57, 115]. In [34], the search time may reduce to a few minutes by only searching for similar architectures compared to an already well-designed baseline DNN model, and hence this comes at the expense of very limited search space and possibly missing better DNN designs. Therefore, combined together, the total search cost for edge devices is still non-negligible, especially given the extremely diverse edge devices for which scalability is very important.

There have also been many prior studies on DNN model compression, such as pruning and quantization [7, 29, 32, 37, 51, 52, 71, 73, 83, 93], matrix factorization [**?**, 36], and knowledge distillation [100]. Like the current practice of NAS, the existing optimizer for compression techniques are typically targeting a single device (e.g., optimally deciding the quantization and pruning policy for an individual target device), making the overall optimization cost linearly increase with the number of target devices and lacking scalability [115].

In summary, the state-of-the-art device-aware DNN optimization still takes a large amount of time and efforts for even a single device [23, 24, 34, 115], and cannot scale to extremely diverse edge devices. Therefore, the device-aware DNN optimization process itself needs to scale up, in view of extremely diverse edge devices.

1.3 Problem Formulation

A common goal of optimizing DNN designs is to maximize the inference accuracy subject to latency and/or energy constraints on edge devices. Mathematically, this problem can be formulated as

$$\min_{\mathbf{x}\in\mathcal{X}} -accuracy(\mathbf{x}) \tag{1.1}$$

$$s.t., \quad latency(\mathbf{x};\mathbf{d}) \leq \overline{L}_{\mathbf{d}}, \tag{1.2}$$

$$energy(\mathbf{x};\mathbf{d}) \leq \overline{E}_{\mathbf{d}}, \tag{1.3}$$

where $\mathbf{x}$ is the representation of the DNN design choice (e.g., a combination of DNN architecture, quantization, and pruning scheme), $\mathcal{X}$ is the design space under consideration, and $\mathbf{d}$ is the representation of an edge device (e.g., CPU/RAM/GPU/OS configuration). Our problem formulation is not restricted to energy and latency constraints; additional constraints, such as robustness to adversarial samples, can also be added. Note that we use "$-accuracy(\mathbf{x})$" as the objective function to be consistent with the standard "min" operator in optimization problems.

The constrained optimization problem in Eqns. (1.1)–(1.3) is called *primal* problem in the optimization literature [17]. It can also be alternatively formulated as a relaxed problem parameterized by $\lambda = (\lambda_1, \lambda_2)$:

$$\min_{\mathbf{x} \in \mathcal{X}} -accuracy(\mathbf{x}) + \lambda_1 \cdot energy(\mathbf{x}; \mathbf{d}) + \lambda_2 \cdot latency(\mathbf{x}; \mathbf{d}), \tag{1.4}$$

where $\lambda = (\lambda_1, \lambda_2)$ are non-negative weight parameters (i.e., equivalent to Lagrangian multipliers) corresponding to the energy and latency constraints, respectively. By increasing a weight (say, λ_2 for latency), the optimal design $\mathbf{x}^*(\mathbf{d}, \lambda)$ by solving (1.4) will result in better performance corresponding to that weight. If the performance constraint is very loose, then $\lambda = (\lambda_1, \lambda_2)$ can approach zero; on the other hand, if the constraint is very stringent, $\lambda = (\lambda_1, \lambda_2)$ will be large. Thus, given a set of latency and energy constraints $\overline{L}_{\mathbf{d}}$ and $\overline{E}_{\mathbf{d}}$, we can choose a set of weight parameters λ_1 and λ_2 such that the constraints in (1.2)(1.3) are satisfied and the accuracy is maximized.

Strictly speaking, some technical conditions (e.g., convexity) need to be satisfied such that the optimal solution to the relaxed problem in (1.4) is also the optimal solution to the constrained problem in (1.1)–(1.3). Nonetheless, the goal in practice is to obtain a sufficiently good DNN design rather than the truly global optimum, because of the usage of a (non-convex) performance predictor as a substitute of the objective function [23, 24, 34, 79, 115]. Thus, with proper weight parameters λ, the relaxed version in (1.4) can be seen as a substitute of the constrained optimization problem (1.1)–(1.3).

While the constrained problem in (1.1) –(1.3) is intuitive to understand, it may not be straightforward to optimize when using search-based algorithms. On the other hand, when using the relaxed formulation in (1.4), one needs to find an appropriate set of weight parameters $\lambda = (\lambda_1, \lambda_2)$ to meet the performance constraints in (1.2)(1.3). In the literature, both constrained and relaxed problems are widely considered to guide optimal DNN designs [34, 115].

In this chapter, we choose to solve the relaxed problem in (1.4) while using efficient searches to identify an optimal $\lambda = (\lambda_1, \lambda_2)$ such that the performance constraints in (1.2)(1.3) are satisfied and the resulting optimal DNN design $\mathbf{x}$ minimizes the accuracy loss (i.e., maximize the accuracy).

1.4 Approach 1: Reusing Performance Predictors for Many Devices

A key bottleneck that slows down the DNN optimization process is the high cost of building performance predictors for each device. In our first approach, we propose to reuse the performance predictors built on a *proxy* device denoted as $\mathbf{d}_0$. While the predictor cannot accurately estimate the performance on a different device, it maintains performance *monotonicity* (e.g., if DNN design $\mathbf{x}_A$ has a lower latency than $\mathbf{x}_B$ on the proxy device, $\mathbf{x}_A$ should still be faster than $\mathbf{x}_B$ on a new device) in many cases. We leverage the performance monotonicity to scale up the DNN optimization without re-building performance predictors for each different device.

1.4.1 Stage 1: Training Performance Predictors on a Proxy Device

To speed up the DNN optimization process, we need to quickly evaluate objective function given different DNN designs. Instead of actually measuring the performance for each DNN design candidate (which is time-consuming), we utilize performance predictors. In our example, we have accuracy/latency/energy predictors. Concretely, the accuracy predictor can be a simple Gaussian process model as used in [34] or a neural network, whose input is the DNN design choice represented by $\mathbf{x}$, and it does not depend on the edge device feature $\mathbf{d}$. We denote the trained accuracy predictor by $Acc_{\Theta_A}(\mathbf{x})$, where Θ_A is learnt parameter for the predictor.

On the other hand, the latency/energy predictors depend on devices. Here, we train the latency/energy predictors on a proxy device following the existing studies [34,115]. For example, to build the latency predictor offline, we can measure the latency for each operator in a DNN candidate and then sum up all the involved operators to obtain the total latency. We denote the latency and energy predictors as $\overline{latency}_{\mathbf{d}_0}(\mathbf{x})$ and $\overline{energy}_{\mathbf{d}_0}(\mathbf{x})$, where the subscript $\mathbf{d}_0$ is to stress that the performance predictors are only accurate (in terms of the absolute performance prediction) for the proxy device $\mathbf{d}_0$.

Given the latency/energy predictor for an edge device, one can easily follow [34,115] and adopt an evolutionary search process to obtain the optimal DNN design. Nonetheless, in [34], the performance predictor cannot transfer to a different device, because the latency/energy performance on one device can change dramatically on a different device: [34] directly uses the absolute performance constraints $\overline{L}_{\mathbf{d}}$ and $\overline{E}_{\mathbf{d}}$ in its (modified) objective function and hence needs accurate performance prediction for each individual device. In [23,115], the weight parameters $\lambda = (\lambda_1, \lambda_2)$ are simply treated as hyperparameters. How to tune $\lambda = (\lambda_1, \lambda_2)$ to meet the performance constraints for a target device is not specified. Since it aims at making weighted objective function in (1.4) as close to the true value as possible on a target device, it needs accurate performance prediction for that target device.

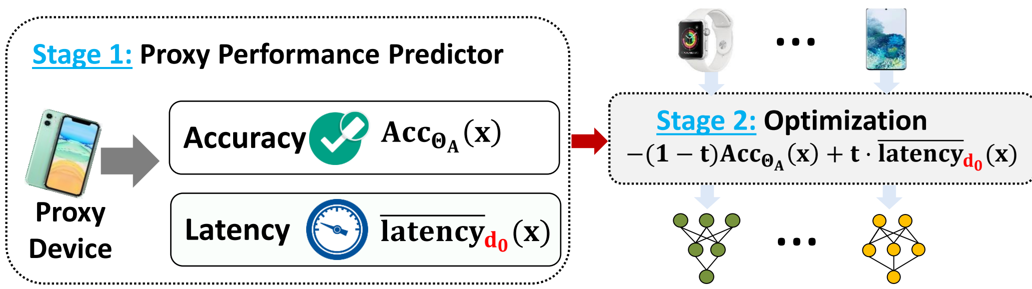
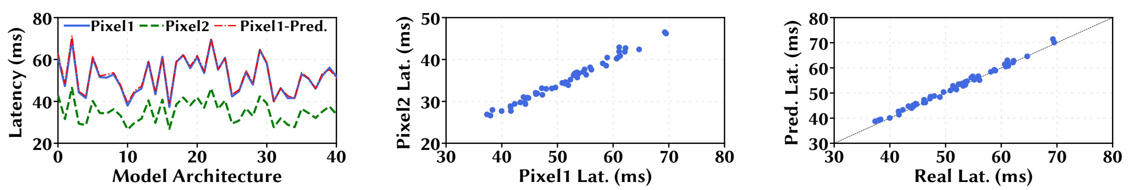


Figure 1.3: Overview of “reusing performance predictors” to scale up DNN optimization.



(a) Latencies of 40 DNN models (b) Latency on Pixel 2 vs. Pixel 1 (c) Pred. vs. real latency (Pixel 1)

Figure 1.4: The measured and predicted average latencies of a set of 40 DNN models with different architectures on Google Pixel 1 and Pixel 2. The latency predictor is built based on Google Pixel 1. The latency values are released accompanying the publication [24].

Thus, performance predictors are needed for each individual device in [23,115]. In our work, instead of building a latency/energy predictor for each device, we will reuse the predictor for other devices as described in the next subsection.

1.4.2 Stage 2: Optimizing DNN Designs on New Devices

In this framework, we avoid the cost of building performance predictors for each individual device by leveraging the performance monotonicity of DNNs on different devices. To better explain our idea, we only consider the latency constraint and illustrate our approach in Fig. 1.3.

In many cases, DNNs’ latency performances are monotone on two different devices, which we formally state as follows.

Algorithm 1 DNN Optimization on a New Device

Input: Accuracy predictor $Acc_{\Theta_A}(\mathbf{x})$, proxy device's latency predictor $\overline{latency}_{\mathbf{d}_0}(\mathbf{x})$, latency constraint on the target device $\overline{L}_{\mathbf{d}}$, already considered $\mathcal{T}$ and corresponding optimal DNN designs $\mathcal{X}^* = \{\mathbf{x}^*(t), \forall t \in \mathcal{T}\}$, small $\delta > 0$ for checking latency constraint satisfaction, and maximum iteration $Max\_Iterate$
Output: Optimal DNN design $\mathbf{x}^*$
Initialize: Set $t_{\min} = 0$ and $t_{\max} = 1$;
for i= 1 to $Max\_Iterate$ **do**
  $t = \frac{t_{\min}+t_{\max}}{2}$;
  if $t \notin \mathcal{T}$ **then**
    Solve (1.6) and obtain $\mathbf{x}^*(t)$;
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{t\}$ and $\mathcal{X}^* \leftarrow \mathcal{X}^* \cup \{\mathbf{x}^*(t)\}$
  end if
  Measure latency $latency(\mathbf{x}^*(t^*); \mathbf{d})$;
  if $latency(\mathbf{x}^*(t^*); \mathbf{d}) \geq \overline{L}_{\mathbf{d}} + \delta$ **then**
    $t_{\min} = t$;
  else if $latency(\mathbf{x}^*(t^*); \mathbf{d}) \leq \overline{L}_{\mathbf{d}} - \delta$ **then**
    $t_{\max} = t$;
  else
    Break;
  end if
end for

return $\mathbf{x}^*(t)$;

Performance monotonicity. Given two different devices $\mathbf{d}_0 \neq \mathbf{d}$ and two different DNN designs $\mathbf{x}_A \neq \mathbf{x}_B$, if $latency(\mathbf{x}_A; \mathbf{d}_0) \geq latency(\mathbf{x}_B; \mathbf{d}_0)$, then $latency(\mathbf{x}_A; \mathbf{d}) \geq latency(\mathbf{x}_B; \mathbf{d})$ also holds. We say that the two DNN designs $\mathbf{x}_A$ and $\mathbf{x}_B$ are performance monotonic on the two devices $\mathbf{d}_0$ and $\mathbf{d}$.

With performance monotonicity, the relative ranking of different DNNs' latency performances is preserved between the two devices. For example, as shown in Fig. 4 in [34], for different convolution operators, latency performance monotonicity is observed between Samsung S8 with Snapdragon 835 mobile CPU and Hexagon v62 DSP with 800 MHz frequency, although the absolute performances are very different. We also show in Fig. 1.4 the performance monotonicity of a set of 40 DNN models with different architectures on

Google Pixel 1 and Pixel 2. These two devices have major differences in terms of several specifications, such as operating systems (Android 7.1 vs. Android 8.0), chipset (Qualcomm MSM8996 Snapdragon 821 with 14 nm vs. Qualcomm MSM8998 Snapdragon 835 with 10 nm), CPU (Quad-core 2x2.15 GHz Kryo & 2x1.6 GHz Kryo vs. Octa-core 4x2.35 GHz Kryo & 4x1.9 GHz Kryo) and GPU (Adreno 530 vs Adreno 540), which can affect the latencies. As a result, the absolute latency values on these two devices are very different and not following a simple scaling relation. Nonetheless, on these two devices, many of the DNNs preserve performance monotonicity very well. Moreover, we see that the latency predictor built on Google Pixel 1 is quite accurate compared to the true value. This demonstrates that the latency predictor on Google Pixel 1 can also be reused for Pixel 2, although the authors build another latency predictor for Pixel 2 in their released files [24]. More extensive discussion and measurements regarding performance monotonicity are introduced in Chapters 3 and 4.

As a result, the latency constraint $latency(\mathbf{x}; \mathbf{d}) \leq \overline{L}_{\mathbf{d}}$ can be transformed into $latency(\mathbf{x}; \mathbf{d}_0) \leq \overline{L}'_{\mathbf{d}}$. That is, there exists another latency constraint $\overline{L}'_{\mathbf{d}}$ such that if the latency of a DNN design $\mathbf{x}$ on the proxy device $\mathbf{d}_0$ satisfies $latency(\mathbf{x}; \mathbf{d}_0) \leq \overline{L}'_{\mathbf{d}}$, then the latency of the same DNN design $\mathbf{x}$ on our target device $\mathbf{d}$ will meet is actual latency constraint, i.e., $latency(\mathbf{x}; \mathbf{d}) \leq \overline{L}_{\mathbf{d}}$.

Consequently, we convert the original latency constraint $latency(\mathbf{x}; \mathbf{d}) \leq \overline{L}_{\mathbf{d}}$ into an equivalent latency constraint expressed on the proxy device $latency(\mathbf{x}; \mathbf{d}_0) \leq \overline{L}'_{\mathbf{d}}$, which we can reuse the proxy device's latency predictor to approximate (i.e., $\overline{latency}_{\mathbf{d}_0}(\mathbf{x}) \leq \overline{L}'_{\mathbf{d}}$). Therefore, based on proxy device's predictor, the DNN design problem for our new target device can be re-written as

$$\min_{\mathbf{x}\in\mathcal{X}} -Acc_{\Theta_A}(\mathbf{x}), \qquad s.t., \;\; \overline{latency}_{\mathbf{d}_0}(\mathbf{x}) \leq \overline{L}'_{\mathbf{d}}. \tag{1.5}$$

Nonetheless, without knowing $\overline{L}'_{\mathbf{d}}$ a priori, we cannot directly solve the constrained optimization problem (4.5). Thus, we reformulate the problem (4.5) as

$$\min_{\mathbf{x}\in\mathcal{X}} -(1-t)\cdot Acc_{\Theta_A}(\mathbf{x}) + t\cdot \overline{latency}_{\mathbf{d}_0}(\mathbf{x}), \tag{1.6}$$

where $t \in [0,1]$ plays an equivalent role as λ_2 in the original relaxed problem in (1.4). With a larger value of t, the resulting latency will be smaller (predicted for the proxy device), and vice versa. Importantly, because of performance monotonicity, a larger t will also result in a smaller latency on the new target device. Given each value of t, the problem (1.6) can be quickly solved (e.g., using search-based algorithms), because the objective function can be efficiently evaluated based on accuracy/latency predictors built on the proxy device. For each t, there exists a corresponding optimal $\mathbf{x}^*(t)$.

Now, the problem reduces to finding an optimal t^* such that the actual latency constraint $latency(\mathbf{x};\mathbf{d}) \approx \overline{L}_{\mathbf{d}}$ is satisfied[1] and the accuracy is also maximized (i.e., minimizing $-Acc_{\Theta_A}(\mathbf{x})$). Then, given t^*, we can obtain $\mathbf{x}^*(t^*)$. Specifically, for each t, we measure the actual latency $latency(\mathbf{x}^*(t^*);\mathbf{d})$ and check if it just meets the actual latency constraint $\overline{L}_{\mathbf{d}}$. Since t is a scalar, we can efficiently search for the optimal t^* using bi-section methods. For example, even with a granularity of 0.001 (i.e., 1001 possible values of $t \in [0,1]$), we only

[1] If the latency constraint is very loose (i.e., $\overline{L}_{\mathbf{d}}$ is sufficiently large), then the actual latency $latency(\mathbf{x};\mathbf{d})$ will always be smaller than $\overline{L}_{\mathbf{d}}$. In this case, we have $t^* \to 0$.

need at most $10 = \lceil \log_2(1001) \rceil$ searches and latency measurements on the target device. This can reduce the significant cost of building a latency predictor for the target device. The algorithm is described in Algorithm 6.

1.4.3 Remarks

We offer the following remarks on our first approach.

Proxy latency with monotonicity. Essentially, the proxy device's latency predictor $\overline{latency}_{\mathbf{d}_0}(\mathbf{x})$ serves as a proxy latency for the actual target device. Nonetheless, a key novelty and difference from the FLOP-based proxy latency function is that $\overline{latency}_{\mathbf{d}_0}(\mathbf{x})$ can preserve performance monotonicity for a large group of devices (i.e., a larger $\overline{latency}_{\mathbf{d}_0}(\mathbf{x})$ also means a large actual latency on the target device), whereas FLOP-based proxy latency does not have this desired property and a higher FLOP can commonly have a smaller latency on a target device.

When performance monotonicity does not hold. The core idea of our first approach is to leverage the performance monotonicity of DNNs on different devices. But, this may not hold for all devices: a DNN model with the lowest latency on one device may not always have the best latency performance on another device [81]. The violation of performance monotonicity can be found when the actual latency of a new DNN design becomes significantly higher while it is expected to be lower. If the performance monotonicity does not hold between the proxy device and the new target device, then we will train a new performance predictor for the new target device and treat it as a new proxy device (for possible future reuse); when another device arrives, we will match it with the best suitable proxy devices based on their similarities, and if performance monotonicity does not hold between the new target device and any of the existing proxy devices, we will train a new performance predictor for this new device.

Note that performance monotonicity is not required to strictly hold for all DNNs, as long as it *approximately* holds for optimal DNN designs $\mathbf{x}^*(t)$ for a sufficiently large set of t. The reason is that the DNN design problem is non-convex and we only expect to find a reasonably good DNN design, rather than the truly global optimal design. We expect performance monotonicity at least among a group of devices that are *not* significantly different from each other (e.g., see Fig. 1.4 for latencies on Google Pixel 1 and Pixel 2, which have different operating systems, chipsets, CPUs and GPUs). In any case, our approach will not be slower than the existing predictor-aided DNN optimization that requires performance predictors for each different device [34], since our approach can always roll back to the existing approaches by treating each target device as a new proxy device.

Energy constraint. If we also want to factor energy into the objective function, we need to consider a new objective function parameterized by $\mathbf{t} = (t_1, t_2)$ where $t_1 \geq 0$, $t_2 \geq 0$, and $t_1 + t_2 \leq 1$:

$$\min_{\mathbf{x} \in \mathcal{X}} -(1 - t_1 - t_2) \cdot Acc_{\Theta_A}(\mathbf{x}) + t_1 \cdot \overline{latency}_{\mathbf{d}_0}(\mathbf{x}) + t_2 \cdot \overline{energy}_{\mathbf{d}_0}(\mathbf{x}), \tag{1.7}$$

where $\overline{energy}_{\mathbf{d}_0}(\mathbf{x})$ is the proxy device's energy predictor. Accordingly, we need to extend Algorithm 6 to consider a search process over t_1 and t_2. While this is more complicated than bi-section on a scalar value, there exist efficient search methods over a multi-dimension space [44]. Regardless, searching over a low-dimensional parameter space (t_1, t_2) is much easier than searching over the DNN design space (e.g., architecture space).

1.5 Approach 2: Learning to Optimize

In this section, we introduce our second framework to scale up DNN design optimization.

1.5.1 Overview

While our first approach aims at avoiding training performance predictors for each individual device, we still need to take a small number of actual latency/energy measurements on each target device, because the proxy device's performance predictor can only provide a relative/ordered performance instead of the absolute performance. To scale up the optimization of DNNs for edge inference and generate an optimal DNN design instantly for each target device, we now present our second approach.

Our key idea is *learning to optimize*: instead of performing DNN design optimization repeatedly (once for an individual device), we first learn a DNN *optimizer* from DNN optimization on sample devices, and then apply the learnt DNN optimizer to new unseen devices and directly obtain the optimal DNN design.

More specifically, we take a departure from the existing practice by: (1) leveraging new performance predictors that can estimate the resulting inference latency/energy performance given a DNN-device pair; and (2) using an automated optimizer which takes the device features and optimization parameters as input, and then directly outputs the optimal DNN design. This is illustrated in Fig. 2.3. Our latency/energy performance predictors take as explicit input both the DNN features and device features, and hence they can output the resulting performance for new unseen devices. Note that appropriate embedding of DNN

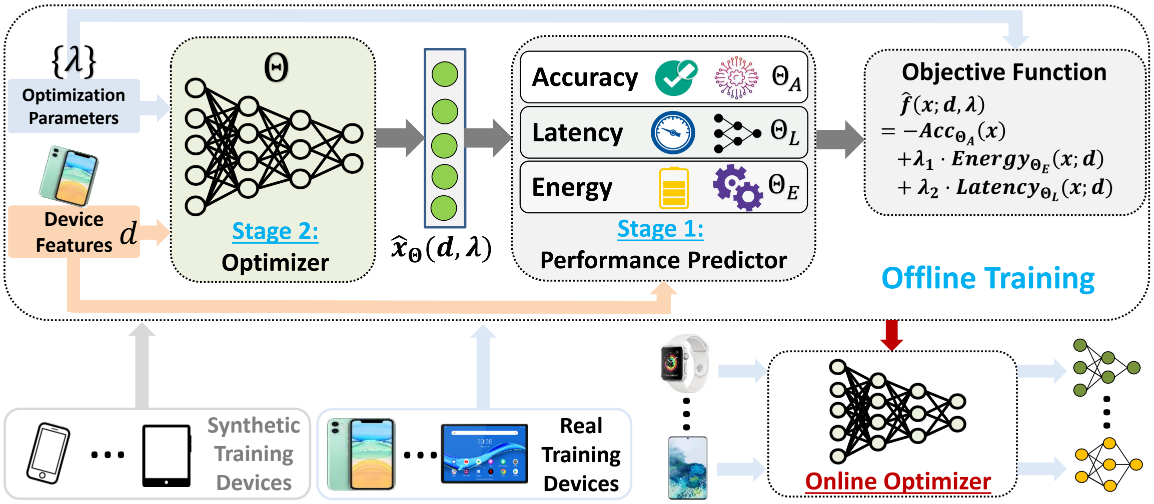


Figure 1.5: Overview of "learning to optimize" to scale up DNN optimization for edge inference. Once the optimizer is trained, the optimal DNN design for a new device is done almost instantly (i.e., only one inference time).

and device features will be very helpful to facilitate training the performance predictors and DNN optimizer.

Our automated optimizer utilizes a neural network to approximate the optimal DNN design function, and is intended to cut the search time that would otherwise be incurred for each device. The initial overhead of training our performance predictors and optimizer is admittedly higher than the current practice of only training device-specific predictors, but the overall overhead is expected to be significantly lower, considering the extreme diversity of edge devices.

1.5.2 Training Performance Predictors and Optimizer

Our proposed design builds on top of two-stage training as described below.

Stage 1: Training performance predictors. The accuracy predictor is the same as the one used in our first approach, since it is measured on a reference dataset without dependence on devices. On the other hand, the latency/energy predictor neural network will use both device feature $\mathbf{d}$ and DNN design representation $\mathbf{x}$ as input, and output the respective performance. They are each trained by running DNNs with sampled designs on training devices and using mean squared error (i.e., the error between the predicted performance and the true measured value) as the loss function. The key difference between our design and [34,115] is that our latency/energy performance predictors use device features as part of the input and hence can apply to new unseen devices without training new performance predictors.

We denote the set of training edge device features as $\mathcal{D}'_T$, where each element $\mathbf{d} \in \mathcal{D}'_T$ corresponds to the feature of one available training device. To generate training samples, we can randomly sample some DNN designs (e.g., randomly select some architectures) plus existing DNN designs if available, and then measure their corresponding performances on training devices as the labels. We denote the trained accuracy/energy/latency predictor neural network by $Acc_{\Theta_A}(\mathbf{x})$, $Energy_{\Theta_E}(\mathbf{x};\mathbf{d})$, and $Latency_{\Theta_L}(\mathbf{x};\mathbf{d})$, respectively, where Θ_A, Θ_E, and Θ_L are learnt parameters for the three respective networks. Thus, the predicted objective function $\hat{f}(\mathbf{x};\mathbf{d},\lambda)$ can be expressed as

$$\hat{f}(\mathbf{x};\mathbf{d},\lambda) = -Acc_{\Theta_A}(\mathbf{x}) + \lambda_1 \cdot Energy_{\Theta_E}(\mathbf{x};\mathbf{d}) + \lambda_2 \cdot Latency_{\Theta_L}(\mathbf{x};\mathbf{d}). \tag{1.8}$$

The accuracy/energy/latency predictor neural networks are called *performance networks*, to be distinguished from the optimizer network we introduce below.

Since collecting energy/latency performances on real training devices is time-consuming, we can use iterative training to achieve better sample efficiency. Specifically, we can first choose a small training set of DNN designs at the beginning, and then iteratively include an exploration set of new DNN designs $\mathcal{X}_{explore}$ to update the performance networks. This is described in Algorithm 2. The crux is how to choose the exploration set $\mathcal{X}_{explore}$. Some prior studies have considered Bayesian optimization to balance exploration vs. exploitation [95, 101], and we leave the choice of $\mathcal{X}_{explore}$ in each iteration as our future work.

Stage 2: Training the automated optimizer. Given an edge device represented by feature $\mathbf{d}$ and optimization parameter λ, the representation of the corresponding optimal DNN design can be expressed as a function $\mathbf{x}^*(\mathbf{d}, \lambda)$. The current practice of DNN optimization is to repeatedly run an optimizer (e.g., search-based algorithm), once for a single device, to minimize the predicted objective function [34, 115]. Nonetheless, obtaining $\mathbf{x}^*(\mathbf{d}, \lambda)$ is non-trivial for each device and not scalable to extremely diverse edge devices. Thus, we address the scalability issue by leveraging the strong prediction power of another fully-connected neural network parameterized by Θ to approximate the optimal DNN design function $\mathbf{x}^*(\mathbf{d}, \lambda)$. We call this neural network *optimizer network*, whose output is denoted by $\hat{\mathbf{x}}_\Theta(\mathbf{d}, \lambda)$ where Θ is the network parameter that needs to be learnt. Once Θ is learnt, when a new device arrives, we can directly predict the corresponding optimal DNN design choice $\hat{\mathbf{x}}_\Theta(\mathbf{d}, \lambda)$.

For training purposes, in addition to features of real available training devices $\mathcal{D}'_T$, we can also generate a set of additional *synthetic* device features $\mathcal{D}_S$ to augment the training samples. We denote the combined set of devices for training as $\mathcal{D}_T = \mathcal{D}'_T \cup \mathcal{D}_S$, and the training set of optimization parameters as Λ_T which is chosen according to practical needs (e.g., latency may be more important than energy or vice versa). Next, we discuss two

Algorithm 2 Training Performance and Optimizer Networks

Input: Real training devices $\mathcal{D}'_T$, synthetic training devices $\mathcal{D}_S$, training set of optimization parameters Λ_T, trained DNN models and their corresponding design space $\mathcal{X}$, initial exploration set of $\mathcal{X}_{explore}$, initial training sets of sampled DNN designs $\mathcal{X}_T \subset \mathcal{X}$ and the corresponding accuracy/energy/latency labels measured on real training devices, and maximum iteration rounds $Max\_Iterate$

Output: Performance network parameters $\Theta_A, \Theta_E, \Theta_L$, and optimizer network parameter Θ

Initialize: Randomize $\Theta_A, \Theta_E, \Theta_L$, and Θ;

for $i = 1$ to $Max\_Iterate$ **do**

  for $\mathbf{x} \in \mathcal{X}_{explore} \subset \mathcal{X}$ and $\mathbf{d} \in \mathcal{D}'_T$ **do**

    $\mathcal{X}_T \leftarrow \mathcal{X}_T \cup \{\mathbf{x}\}$;

    Measure $accuracy(\mathbf{x})$ for a new accuracy label;

    Measure $energy(\mathbf{x}; \mathbf{d})$ and $latency(\mathbf{x}; \mathbf{d})$ for new energy and latency labels, respectively;

    Update Θ_A, Θ_E, and Θ_L by training performance networks as described in **Stage 1**;

  end for

  Choose a new $\mathcal{X}_{explore}$;

end for

if Training method 1 is used **then**

  Fix $\Theta_A, \Theta_E, \Theta_L$, and obtain $\mathbf{x}^*(\mathbf{d}, \lambda) = \arg\min_{\mathbf{x}} \hat{f}(\mathbf{x}; \mathbf{d}, \lambda)$, $\forall(\mathbf{d}, \lambda) \in (\mathcal{D}_T, \Lambda_T)$;

  Update Θ by training the optimizer network using Method 1;

else

  Fix $\Theta_A, \Theta_E, \Theta_L$, and update Θ by training the optimizer network using Method 2;

end if

return $\Theta_A, \Theta_E, \Theta_L$, and Θ;

different methods to train the optimizer network.

Training Method 1: A straightforward method of training the optimizer network is to use the optimal DNN design $\mathbf{x}^*(\mathbf{d}, \lambda)$ as the ground-truth label for input sample $(\mathbf{d}, \lambda) \in (\mathcal{D}_T, \Lambda_T)$. Specifically, we can use the mean squared error loss

$$\min_{\Theta} \frac{1}{N} \sum_{(\mathbf{d},\lambda)\in(\mathcal{D}_T,\Lambda_T)} |\hat{\mathbf{x}}_{\Theta}(\mathbf{d}, \lambda) - \mathbf{x}^*(\mathbf{d}, \lambda)|^2 + \mu\|\Theta\|, \tag{1.9}$$

where N is the total number of training samples, $\mu\|\Theta\|$ is the regularizer to avoid overfitting, and the ground-truth optimal DNN design $\mathbf{x}^*(\mathbf{d}, \lambda)$ is obtained by using an existing optimization algorithm (e.g., evolutionary search in [34, 115]) based on the predicted objective function. Concretely, the optimal DNN design used as the ground truth is $\mathbf{x}^*(\mathbf{d}, \lambda) = \arg\min_{\mathbf{x}} \hat{f}(\mathbf{x}; \mathbf{d}, \lambda)$, where $\hat{f}(\mathbf{x}; \mathbf{d}, \lambda)$ is the predicted objective function with parameters Θ_A, Θ_E, and Θ_L learnt in Stage 1.

Training Method 2: While Method 1 is intuitive, generating many training samples by obtaining the optimal DNN design $\mathbf{x}^*(\mathbf{d}, \lambda)$, even based on the predicted objective function, can be slow [34, 115]. To reduce the cost of generating training samples, we can directly minimize the predicted objective function $\hat{f}(\mathbf{x}; \mathbf{d}, \lambda) = -Acc_{\Theta_A}(\mathbf{x}) + \lambda_1 \cdot Energy_{\Theta_E}(\mathbf{x}; \mathbf{d}) + \lambda_2 \cdot Latency_{\Theta_L}(\mathbf{x}; \mathbf{d})$ in an unsupervised manner, without using the optimal DNN design choice $\mathbf{x}^*(\mathbf{d}, \lambda)$ as the ground-truth label. Specifically, given the input samples $(\mathbf{d}, \lambda) \in (\mathcal{D}, \Lambda)$ including both real and synthetic device features, we optimize the optimizer network parameter Θ to directly minimize the following loss:

$$\min_{\Theta} \frac{1}{N} \sum_{(\mathbf{d},\lambda)\in(\mathcal{D}_T,\Lambda_T)} \hat{f}(\hat{\mathbf{x}}_{\Theta}(\mathbf{d}, \lambda); \mathbf{d}, \lambda) + \mu\|\Theta\|. \tag{1.10}$$

The output of the optimizer network directly minimizes the predicted objective function, and hence represents the optimal DNN design. Thus, our training of the optimizer network in Method 2 is guided by the predicted objective function only and unsupervised. When updating the optimizer network parameter Θ, the parameters for performance predictors Θ_A, Θ_E, and Θ_L learnt in Stage 1 are fixed without updating. In other words, by viewing the concatenation of optimizer network and performance predictor networks as a single neural network (illustrated in Fig. 2.3), we update the parameters (Θ) in the first few layers while freezing the parameters $(\Theta_A, \Theta_E, \Theta_L)$ in the last few layers to minimize the loss expressed in Eqn. (1.10).

Finally, we can search for appropriate weight parameters λ to obtain the optimal DNN design subject to performance requirement. The key difference between our second approach and the first one is that in the second approach, there is no need to measure the performance for each candidate DNN design on the target device. Note that in our first approach, for each target device, there are only a few candidate DNN designs due to the high efficiency bisection methods.

1.5.3 Remarks

In this section, we propose a new approach to scaling up DNN optimization for edge inference and present an example of training the optimizer. The key point we would like to highlight in this chapter is that performing DNN optimization for each individual device as considered in the existing research is not scalable in view of extremely diverse edge devices. We now offer the following remarks (mostly regarding our second approach — learning to optimize).

• **DNN update.** When a new training dataset is available and the DNN models need to be updated for edge devices, we only need to build a new accuracy predictor on (a subset of) the new dataset and re-train the optimizer network. The average energy/latency predictors remain unchanged, since they are not much affected by training datasets. Thus, the time-consuming part of building energy/latency predictors in our proposed approach is a one-time effort and can be re-used for future tasks.

• **Generating optimal DNN design.** Once the optimizer network is trained, we can directly generate the optimal DNN design represented by $\hat{\mathbf{x}}_\Theta(\mathbf{d}, \lambda)$ given a newly arrived edge device $\mathbf{d}$ and optimization parameter λ. Then, the representation $\hat{\mathbf{x}}_\Theta(\mathbf{d}, \lambda)$

is mapped to the actual DNN design choice using the learnt decoder. Even though the optimizer network may not always result in the optimal DNN designs for all edge devices, it can at least help us narrow down the DNN design to a much smaller space, over which fine tuning the DNN design becomes much easier than over a large design space.

• **Empirical effectiveness.** Using performance predictors to guide the optimizer is relevant to *optimization from samples* [10,11]. While in theory optimization from samples may result in bad outcomes because the predictors may output values with significant errors, the existing NAS and compression approaches using performance predictors [23,34,80,87,115] have empirically shown that such optimization from samples work very well and are able to significantly improve DNN designs in the context of DNN optimization. This is partly due to the fact that the predicted objective function only serves as a guide and hence does not need to achieve close to 100% prediction accuracy.

• **Relationship to the existing approaches.** Our proposed design advances the existing prediction-assisted DNN optimization approaches [34, 115] by making the DNN optimization process scalable to numerous diverse edge devices. If our approach is applied to only *one* edge device, then it actually reduces to the methods in [34, 115]. Specifically, since the device feature $\mathbf{d}$ is fixed given only one device, we can remove it from our design illustrated in Fig. 2.3. As a result, our performance predictors are the same as those in [34, 115]. Additionally, our optimizer network can be eliminated, or reduced to a trivial network that has a constant input neuron directly connected to the output layers without any hidden layers. Thus, when there is only one edge device, our approach is essentially identical to those in [34, 115]. Therefore, even in the worst event that the optimizer network or performance predictor network does not generalize well to some new unseen edge devices (due to, e.g., poor training and/or lack of edge device samples), we can always optimize the DNN design for each individual device, one at a time, and roll back to state of the art [34, 115] without additional penalties.

• **When scalability is not needed.** It has been widely recognized that a single DNN model cannot perform the best on many devices, and device-aware DNN optimization is crucial [23, 34, 112, 115, 121]. Thus, we focus on the *scalability* of DNN optimization for extremely diverse edge devices. On the other hand, if there are only a few target devices (e.g., a vendor develops its own specialized DNN model for only a few products), our second approach does not apply while our first appraoch (i.e., re-using proxy device's performance predictors is more suitable).

• **GAN-based DNN design.** There have been recent attempts to reduce the DNN design space by training generative adversarial networks [60]. Nonetheless, they only produce DNN design candidates that are more likely to satisfy the accuracy requirement, and do not perform energy or latency optimization for DNN designs. Thus, a scalable performance evaluator is still needed to identify an optimal DNN design for diverse edge devices. By contrast, our second approach is inspired by "learning to optimize" [9]: our optimizer network takes almost no time (i.e., only one optimizer network inference) to directly produce an *optimal* DNN design, and can also produce multiple optimal DNN designs by varying the optimization parameter λ to achieve different performance tradeoffs.

• **Ensemble.** To mitigate potentially bad predictions produced by our optimizer or performance networks, we can use an ensemble in our second approach. For example, an ensemble of latency predictors can be used to smooth the latency prediction, while an ensemble of the optimizer network can be used to generate multiple optimal DNN designs, out of which we select the best one based on (an ensemble of) performance predictors.

- **Learning to optimize.** Our proposed optimizer network is relevant to the concept of learning to optimize [9], but employs a different loss function in Method 2 which does not utilize ground-truth optimal DNN designs as labels. The recent study [66] considers related unsupervised learning to find optimal power allocation in an orthogonal problem context of multi-user wireless networks, but the performance is evaluated based on theoretical formulas. By contrast, we leverage performance predictors to guide the training of our optimizer network and use iterative training.

- **Public datasets for future research.** Finally, the lack of access to many diverse edge devices is a practical challenge that prohibits many researchers from studying or experimenting scalable DNN optimization for edge inference. While there are large datasets available on $(architecture, accuracy)$ [103], to our knowledge, there do not exist similar publicly-available datasets containing $(architecture, energy, latency, device)$ for a wide variety of devices. If such datasets can be made available, they will tremendously help researchers build novel automated optimizers to scale up the DNN optimization for heterogeneous edge devices, benefiting every stakeholder in edge inference be it a gigantic player or a small start-up.

1.6 Thesis Contribution

In this dissertation, we explore automated approaches to scalably and efficiently optimize DNN design for diverse edge devices, towards achieving ultra efficient machine learning for edge inference. Our proposed approaches include an automated DNN model selection engine to maximize edge users' Quality of Experience (QoE) utilizing neural bandit learning, efficient hardware-aware NAS and software-hardware co-design of neural accelerators based on the "Resuing Performance Predictors for Many Devices" framework introduced in Section 1.4.

In **Chapter 2**, we target the *automated and user-centric DNN selection* problem considering three facts: (1) given an inference task, a large number of diverse DNN models can be generated by navigating through (even a small part of) the huge design space in terms of neural architectures and compression techniques (e.g., pruning, quantization, and knowledge distillation), and different DNN models can exhibit dramatically different tradeoffs among performance metrics, such as accuracy, inference latency and energy consumption; (2) edge devices are extremely diverse, ranging from high-end devices with state-of-the-art CPUs/GPUs and dedicated accelerators to low-end devices powered by CPUs of several years old; (3) optimizing standard performance metrics (e.g., accuracy and latency) may not translate into improvement of users' actual subjective QoE. Speicifically, we propose a novel automated and user-centric DNN selection engine, called Aquaman, which keeps users into a closed loop and leverages their QoE feedback to guide DNN selection decisions. The core of Aquaman is a neural network-based QoE predictor, which is continuously updated online. Additionally, we use neural bandit learning to balance exploitation and exploration, with a provably-efficient QoE performance. Our evaluation on a 15-user experimental study as well as synthetic simulations demonstrates that Aquaman outperforms the static DNN selection while approaching the optimal oracle in terms of users' QoE.

Complementary to DNN selection from a pre-existing model pool for given edge devices, we relax this assumption by studying hardware-aware NAS in **Chapter 3**. Instead of simply selecting a pre-trained DNN from a model pool, NAS brings flexibility to model design by fully exploring the neural architecture search space and discovering the optimal combination of building blocks for a target device. Specifically, we focus on reducing the total latency evaluation cost for scalable hardware-aware NAS in the presence of diverse target devices across different platforms (e.g., mobile platform, FPGA platform, desktop/server GPU, etc.). Concretely, we show that latency monotonicity commonly exists among different devices, especially devices of the same platform. Informally, latency monotonicity means that the ranking orders of different architectures' latencies are correlated on two or more devices. Thus, with latency monotonicity, building a latency predictor for just one device that serves as a proxy – rather than for each individual target device as in state of the art – is enough. In the absence of strong latency monotonicity, we propose an efficient proxy adaptation technique to significantly boost the latency monotonicity. Finally, we validate our approach and conduct experiments with devices of different platforms on multiple mainstream search spaces, including MobileNet-V2, MobileNet-V3, NAS-Bench-201, ProxylessNAS and FBNet. Our results highlight that, by using just one proxy device, we can find almost the same Pareto-optimal architectures as the existing per-device NAS, while avoiding the prohibitive cost of building a latency predictor for each device.

Besides the automatically-designed DNN models generated by NAS (i.e. software design), optimizing hardware accelerators built on FPGA or ASIC, as well as the corresponding dataflows (e.g., scheduling DNN computations and mapping them on hardware), is also critical for speeding up DNN execution (i.e. hardware design) and has been increasingly studied. Therefore, in addition to the freedom brought to model design by NAS, we further explore the hardware design space in **Chapter 4**. That is, instead of running hardware-aware NAS to obtain Pareto-optimal models for a given target device, we consider the hardware design space together with model search space, in an effort to discover the ultimately efficient accelerator-architecture pairs, which is commly known as hardware-software co-design. While hardware-software co-design can further optimize DNN performance, it also exponentially enlarges the total search space to practically infinity and presents substantial challenges. By settling in-between the fully-decoupled approach and the fully-coupled hardware-software co-design approach, we propose a new semi-decoupled approach to reduce the size of the total co-search space by orders of magnitude, yet without losing design optimality. Particularly, we first perform neural architecture search to obtain a small set of optimal architectures for one accelerator candidate. Importantly, this is also the set of (close-to-)optimal architectures for other accelerator designs based on the property that neural architectures' ranking orders in terms of inference latency and energy consumption on different accelerator designs are highly similar. Then, instead of considering all the possible architectures, we optimize the accelerator design only in combination with this small set of architectures, thus significantly reducing the total search cost. We validate our approach by conducting experiments on various architecture spaces for accelerator designs with different dataflows. Our results highlight that strong latency and energy monotonicity exist among different accelerator designs. More importantly, by using one candidate accelerator as the proxy and obtaining its small set of optimal architectures, we can reuse the same architecture set for other accelerator candidates during the hardware search stage.

Chapter 2

Improving QoE of Deep Neural Network Inference on Edge Devices: A Bandit Approach

2.1 Introduction

Edge devices, such as smart phones and tablets, are gaining a strong momentum and emerging as a crucial platform for deep neural network (DNN) inference [102, 122]. For example, DNN models have been commonly integrated with mobile apps for various functions (e.g., real-time style transfer in Facebook app [121]). Compared to cloud-based inference, running DNN inference directly on these devices (referred to as *edge inference*) is much less reliant on network connection and also better protects user privacy because of local data processing, thus driving the quick adoption of DNN-powered mobile inference.

To enable a satisfactory user experience of edge inference, numerous DNN optimization techniques for neural architecture search as well as model compression have been recently proposed [23,50,106]. Thus, given an inference task, a large number of diverse DNN models can be generated by navigating through (even a small part of) the huge design space in terms of different neural architectures and compression techniques (e.g., pruning, quantization, and knowledge distillation) [23, 51, 71, 73, 115]. Generally, different DNN models can exhibit dramatically different tradeoffs among performance metrics, such as accuracy, inference latency and energy consumption [74, 115].

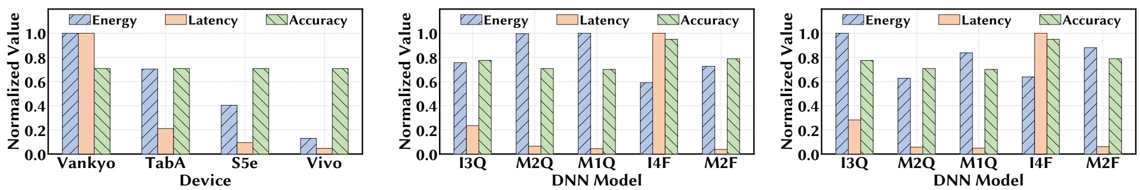
On the other hand, edge devices are extremely diverse, ranging from high-end devices with state-of-the-art CPUs/GPUs and dedicated accelerators to low-end devices powered by CPUs of several years old. For example, the existence of thousands of unique system-on-chips (SoCs) running on over ten thousand different types of mobile phones and tablets further explains the extreme heterogeneity of mobile devices. Importantly, not a single SoC dominates the market: only top 30 SoCs can each have more than 1% of the market share and, when combined, cover 51% of the whole market [121].

Consequently, *how to select the best DNN model out of many choices for each edge device* arises a significant challenge. The DNN selection/optimization for edge inference must be: (1) automated for scalability; and (2) optimizing user's quality of experience (QoE) of edge inference.

Many studies have considered optimally selecting the neural architecture, compression scheme, and/or compiler design for running DNN on a target device [34, 72, 106, 115]. The key idea is to formulate the DNN design as an optimization problem with a pre-determined objective function, such as minimizing a weighted sum of the accuracy loss, inference energy and latency. Nonetheless, the pre-determined objective function is essentially a *proxy* for subjective QoE, without necessarily reflecting the users' true QoE. While it is true that users are generally more satisfied if the latency/energy is lower and the accuracy is higher, using a weighted sum of the accuracy loss, inference energy and latency can be over-simplified for modelling QoE, because users' satisfaction may not be linear in these metrics (as supported by our experimental results in Section 2.5). Even when the actual QoE is indeed a linear function of inference accuracy, energy and latency, the relative weights for these three metrics may not be known in advance and needs online learning based on users' QoE feedback.

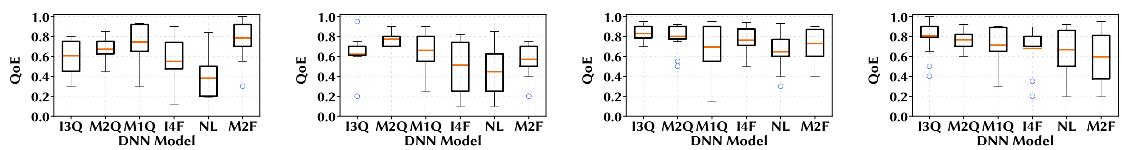
More generally, the QoE can be a very complicated function of the DNN performance metrics. Importantly, the QoE function may not be known or accurately estimated in advance without sufficiently collecting the users' QoE feedback. Thus, properly selecting an objective function in advance to identify the QoE-optimal DNN model for a device is very challenging. Additionally, the same DNN model running on different devices can lead to dramatically different performance metrics. As a result, QoE-optimal DNN models can be very different for different devices, thus mandating an automated DNN selection algorithm rather than manual designs for each device.

In this chapter, we advocate a different approach — *automated and user-centric DNN selection* — which keeps users into a closed loop and leverages their QoE feedback to guide DNN selections. Specifically, we focus on mobile devices as a concrete platform for edge inference and propose a novel provably-efficient DNN selection engine, called Aquaman, to



(a) MobileNet_V2_224_quant (b) Vankyo Matrixpad z1 (c) Samsung Galaxy Tab A

Figure 2.1: (a) Performances of MobileNet_V2_1.0_224_quant on four devices: Vankyo Matrixpad z1 (Vankyo), Samsung Galaxy Tab A (TabA), Samsung Galaxy Tab S5e (S5e), and Vivo V1838A (Vivo). Energy and average latency are normalized to the maximum values on these four devices. (b) and (c) Performances of five DNN models on two devices: Inception_V3_quant (I3Q), MobileNet_V2_1.0_224_quant (M2Q), MobileNet_V1_0.75_192_quant (M1Q), Inception_V4 (I4F), and MobileNet_V2_1.0_224 (M2F). Energy and average latency are normalized to their respective maximum values of the five models on each device. In our experiment, we run image classification 2000 times in our lab for each DNN model, and obtain the percentage of correct classification as the accuracy here.



(a) Vankyo Matrixpad z1 (b) Samsung Galaxy TabA (c) Samsun Galaxy S5e (d) Vivo V1838A

Figure 2.2: Distribution of user ratings regarding six DNN models on four devices. Each rating is normalized to 0-1. "NL" refers to NASNet_large in [47], and the abbreviations for the other five models are shown in Fig. 2.1.

achieve QoE-optimal mobile inference. To address the key challenge of the *a priori* unknown QoE, we leverage a machine learning model to approximate the QoE function based on users' QoE feedback that serves as "labels" for training the model.

The core of Aquaman is to exploit the universal approximation capability of a neural network-based QoE predictor, which estimates the expected QoE for each DNN selection decision and is updated online using users' QoE feedback. While Aquaman can predict the QoE function with good accuracy after collecting a sufficiently large number of users' feedback/labels, its prediction errors can be large at the beginning (i.e., a cold-start problem). To avoid being trapped in a local optimum due to initially large prediction errors and continuously selecting sub-optimal DNN models, Aquaman employs the bandit-based reinforcement learning framework to balance exploitation and exploration. In particular, we can view each DNN model as an *arm* (or action), while the QoE is our reward. The key idea of bandit algorithms is to give higher priorities to those under-explored arms such that they can be explored more and potentially produce higher rewards in the long run. Concretely, we formulate the DNN selection problem into neural bandit learning with delayed feedback, which is also a novel setting in the emerging context of neural bandit [131]. We also provide rigorous analysis for Aquaman, proving that Aquaman can asymptotically maximize the users' average QoE even compared to an optimal oracle.

Finally, we evaluate Aquaman by using an image classification app built on top of the official example of TensorFlow Lite [47]. We consider a 15-user experimental study as well as synthetic simulations, demonstrating that Aquaman outperforms the static DNN selection while approaching the optimal oracle in terms of the users' QoE.

Our main contributions can be summarized as follows:

- We advocate the motivation and necessity of QoE optimal edge inference and user-centric DNN selection approaches.

- We formulate automated user-centric DNN model selection as a multi-armed bandit problem with delayed feedback, and propose an efficient neural bandit approach to balance exploration and exploitation.

- We provide rigorous theoretical analysis, proving that Aquaman can asymptotically maximize the users' average QoE even compared to an optimal oracle.

- We experimentally evaluate our algorithm on a dataset collected via a user study as well as synthetic simulations, demonstrating its effectiveness over the static DNN selection while approaching the optimal oracle in terms of the users' QoE.

2.2 Motivation for User-Centric DNN Selection

We present two existing approaches to DNN selection and explain the necessity of a user-centric approach for QoE-optimal mobile inference.

Device-unaware DNN selection. A straightforward approach to DNN selection is to test on a number of mobile devices and then conservatively select a single model that can meet the performance requirement for most devices. Although simple, its drawbacks are also obvious: *first*, the selected DNN model is optimal only for certain devices and can perform poorly on others; and *second*, there is not a single DNN model that performs the best on all mobile devices.

We take the model of MobileNet_V2_1.0_224_quant as an example and deploy it on four mobile devices for image classification (the details of our experimental setup are provided in Section 2.5). We show in Fig. 2.1(a) the three widely-considered performance metrics for each inference execution: average energy consumption, average latency, and average inference accuracy. The "accuracy" metric in this work is measured in terms of how often the model correctly classifies an image. In our experiment, we run image classification 2000 times in our lab for each DNN model, and obtain the percentage of correct classification as our accuracy. While the model performs well on the high-end device Vivo V1838A, its performance is not satisfactory on low-end devices such as Vankyo Matrixpad z1 (whose average latency is around 1.2s). Our observation is also corroborated by a Facebook study [121], which finds that the same DNN model can result in a large performance variation by a factor of 10x on different devices.

We further test five different DNN models hosted on the official TensorFlow website and show their resulting performances on two mobile devices in Figs. 2.1(b) and 2.1(c), respectively. We notice that different models exhibit different performance tradeoffs on different devices. Among the five tested models on Vankyo Matrixpad z1, the MobileNet_V2_1.0_224 model is the most appealing one, because of its low energy and latency while achieving a reasonably high accuracy. We notice that the latency of I4F is much larger than other models in Figs. 2.1(b) and 2.1(c). The reasons are: first, I4F is a floating-point model compared to other quantized models; second, I4F has a more complicated model structure than the other floating-point model M2F. This aligns with the official latency measurements released in [48].

Device-aware DNN selection. To overcome the drawbacks of the one-for-all approach, many prior studies have considered device-aware DNN optimization/selection [3, 24, 34, 51, 71, 72, 79, 82, 85, 106, 115]. While the existing approaches can produce an optimal DNN model for a given device, it lacks scalability when facing a large number of heterogeneous mobile devices. Specifically, even using prediction-assisted optimization, the often lengthy process of building an offline performance predictor (e.g., profiling the performance of many sample models on the target device) is required for *each* individual target device [34, 115, 129].

More importantly, optimizing a pre-determined objective function (such as a weighted sum of accuracy loss, inference energy and latency [72,115]) does *not* necessarily lead to improvement in users' QoE. For example, for mobile devices with limited battery capacities, users may generally prefer more energy-efficient DNN models while willing to accept a lower accuracy and/or increased latency. In fact, the QoE can be a very complicated function that may not be accurately known in advance without sufficiently collecting the users' feedback.

A survey of users' QoE. To see how satisfied users feel when using DNN-based mobile inference, we recruit 15 participants into our survey to test six different DNN models on four different mobile devices (i.e., a total of 24 DNN-device pairs for each participant) and provide QoE feedback in the form of numeric ratings in 1-10. Each participant is asked to use each of the DNN models for a while and then provide a numerical rating for its QoE with each device-model pair on a scale of 1-10 (more details are in Section 2.5). In addition to the five DNN models shown in Fig. 2.1, we include another model — NASNet_large (NL) — to verify our intuition that it should score the worst QoE, especially on low- and mid-end devices since it is an overly large and slow model for these devices. While our survey size is small due to limited resources to admit more participants, the key point is that users' QoE is subjective and not a simple linear combination of the three widely-considered metrics (accuracy, energy and latency) in the existing DNN optimization approaches. Fig. 2.2 shows the survey results (normalized to 0-1), including the average rating, 25/75 percentile, minimum/maximum rating as well as outliers. Based on the average rating, each device has its own QoE-optimal DNN model: MobileNet_V2_1.0_224 on Vankyo Matrixpad z1, MobileNet_V2_1.0_224_quant on Samsung Galaxy Tab A, and Inception_V3_quant on

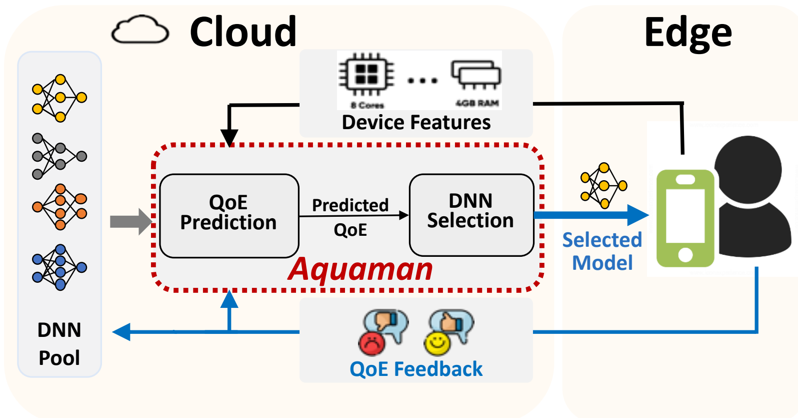


Figure 2.3: Overview of **Aquaman**. Aquaman is implemented on the server/cloud side and works as a middleman between a pool of available DNN models and usersâĂŹ devices. Aquaman focuses on the selection of already pre-trained DNN models (each having different architectures and/or compression schemes) for mobile inference. Aquaman mainly consists of QoE prediction and DNN model selection: for a given device, the QoE predictor first outputs the QoE of each device-model pair; then, Aquaman selects the candidate model that generates the highest QoE upper confidence bound for the device.

Samsung Galaxy S5e and Vivo V1838A. Moreover, a common observation is that the overly large model NASNet_large yields almost the worst QoE on all the devices.

To truly optimize the users' QoE of mobile inference, we advocate a different approach — automated and user-centric DNN selection — which keeps users into a closed loop and leverages their QoE feedback to refine future DNN selections.

2.3 Overview of Aquaman

As illustrated in Fig. 2.3, Aquaman is implemented on the server/cloud side and works as a middleman between a pool of available DNN models and users' devices. Aquaman focuses on the selection of already pre-trained DNN models (each having different architectures and/or compression schemes) for mobile inference, and hence model training is orthogonal. In practice, the selected model is integrated with a DNN-powered mobile app. The workflow of Aquaman is summarized as follows.

Initialization. The QoE function may not be accurately known in advance without enough feedback from the users. Thus, before running online, Aquaman first initializes a QoE prediction model that takes both DNN model and device features as input and outputs the estimated average QoE. Here, we leverage a fully-connected neural network-based QoE predictor due to its universal capability to approximate any functional relationship (i.e., QoE in our study). In our work, Aquaman focuses on optimizing the users' average QoE. Nonetheless, Aquaman can be extended to optimize user-specific QoE by including user features (e.g., preferences and usage behaviors, if available) into the QoE predictor.

Online DNN selection. During the online stage, Aquaman selects DNN models for mobile devices and gradually updates its QoE predictor based on users' feedback.

QoE prediction. Whenever a mobile user requests a DNN model, Aquaman takes the DNN feature and user's device feature (e.g., CPU and RAM) as input into its QoE predictor and then estimates the resulting average QoE for each DNN in the model pool.

DNN selection. After predicting the QoE for each DNN on the given device, Aquaman selects the DNN that has the highest QoE upper confidence bound (UCB), balancing exploration and exploitation.

QoE predictor update. In our work, QoE is a subjective score modeled as an *unknown* function of the device and model features. To learn the QoE model/predictor, users are kindly requested to provide feedback regarding their experiences with the selected DNN model in the mobile app after using the models for a while. This can be achieved by prompting a simple interface to solicit users' rating in a numeric scale or simply "like/not like", as commonly used by many of today's mobile apps. If users choose to provide QoE feedback, their QoE values will be leveraged by Aquaman to update the QoE predictor and improve future DNN selections, thus keeping users into a closed loop. Note that Aquaman aims at maximizing the average QoE, while individual QoE values are highly subjective. Thus, we can take the average of a few QoE values (from different users that have the same selected DNN and device) as one *average QoE* sample to update the QoE predictor.

2.4 Formulation, Algorithm, and Analysis

2.4.1 Preliminaries on Multi-armed Bandits

The multi-armed bandit (MAB) problem models a process of sequential decision making with the tradeoff between exploration and exploitation. In each round t, an action is chosen from a pool of candidate actions (called arms), each of which corresponds to an *a priori* unknown reward. The actual reward is not known until the arm is chosen and played. The goal is to maximize the accumulated expected rewards over T rounds. With more rounds being played, some arms' rewards become better known to us. The tradeoff is a balance between sticking to the arm with the currently known maximal reward (exploitation), or exploring new arms which might give higher rewards in the long run (exploration). MAB models have been widely applied to practical problems, including as ad placement, source routing, computer game-playing, among others [19].

2.4.2 Problem Formulation

Consider that Aquaman needs to perform online DNN model selection over T rounds, each corresponding to a mobile device. Let $\mathcal{A}_t$ be the set of pre-trained DNN candidate models to be selected at round t and the number of candidate DNNs at round t is $|\mathcal{A}_t|$. Note that available DNN model set $\mathcal{A}_t$ can be volatile from round to round, because new DNN models may be added and/or only a subset of DNN models can be deployed on the given mobile device due to QoS constraints. The context/feature with respect to a DNN model $a \in \mathcal{A}_t$ at round t is denoted as $x_{t,a} \in \mathbb{R}^d$, which can be obtained by concatenating the mobile device features at round t (e.g. device's CPU and RAM capacity) and the features of DNN model a (e.g., DNN model size, million MACs, million parameters [34, 115]).

Considering the QoE $y_{t,a}$ is a random variable depending on context $x_{t,a}$, we model it as

$$y_{t,a} = h(x_{t,a}) + \eta_t \tag{2.1}$$

where the average QoE function $h(x_{t,a})$ is a deterministic yet *unknown* function of $x_{t,a}$ with a normalized range $[0, 1]$, and η_t is a ν-sub-Gaussian noise with zero mean conditioned on filtration $\mathcal{F}_{t-1} = \{x_\tau, \eta_{\tau-1}, a_\tau, \tau = 1, \cdots, t-1\}$, i.e. $\forall \varsigma \in \mathbb{R}, \mathbb{E}\left[e^{\varsigma \eta_t} \mid \mathcal{F}_{t-1}\right] \leq \exp\left(\varsigma^2 \nu^2 / 2\right)$.

There is usually a random delay in QoE feedback since users cannot give useful feedback until their models are used for a while. Formally, assuming that if a DNN model is selected for a target mobile device at round s, its corresponding QoE feedback is received after d_s rounds. Then, the set of rounds with QoE fed back at the beginning of round t is expressed as $\mathcal{T}_t^{\mathrm{r}} = \{s \mid s = t - d_s\}$. Also, we assume that the QoE feedback for any $s = 1, \cdots, T$ satisfies $d_s \leq d_{\mathrm{m}}$, i.e., the largest delay is d_{m} after which users are no longer requested for QoE feedback. A larger QoE feedback delay will cause less QoE data collected, whose impact will be analyzed in Theorem 2.4.1. In practice, some users may not provide any QoE feedback (i.e., missing feedback). This will slow down the QoE predictor updating in Aquaman but does not affect the asymptotic optimality of Aquaman.

The goal of Aquaman is to select DNN models to optimize the average QoE. This is also equivalent to minimizing the *regret*, which is the difference between the optimal average QoE and the average QoE achieved by Aquaman. More formally, we consider the cumulative *regret* as the performance metric for DNN model selection, which is expressed as

$$\begin{aligned} R_T &= \mathbb{E}\left[\sum_{t=1}^{T}\left(y_{t,a_t^*} - y_{t,a_t}\right)\right] \\ &= \mathbb{E}\left[\sum_{t=1}^{T} y_{t,a_t^*}\right] - \mathbb{E}\left[\sum_{t=1}^{T} y_{t,a_t}\right], \end{aligned} \tag{2.2}$$

where $a_t^* = \arg\max_{a \in \mathcal{A}_t} \mathbb{E}\left[y_{t,a}\right] = \arg\max_{a \in \mathcal{A}_t} h(x_{t,a})$ is the optimal DNN model with respect to the expected QoE chosen by the oracle and a_t is the DNN model selected by Aquaman at round t. If the cumulative regret R_T increases sub-linearly with T, the resulting average QoE achieved by Aquaman will be asymptotically optimal as $T \to \infty$ [16].

In our problem, the QoE predictor predicts the extent of user satisfaction towards DNN models at *device* level. That is, we optimize the average QoE among each group of users using the same type of device, while individual users' personal preferences are modeled as noise in Eqn. (2.1). Nonetheless, we can extend our QoE predictor by including user-level features (e.g., gender, hobby, etc.) if available. This will make the DNN model selection more personalized to individual users (not only devices), although this requires user-level features and might raise user privacy concerns that are beyond the scope of our study.

2.4.3 Algorithm for Online DNN Selection

At the beginning of each round, the QoE predictor is updated when at least one new QoE feedback is received, i.e., the set of rounds with QoE feedback at the beginning of round t is $\mathcal{T}_t^{\mathrm{r}} = \{s \mid s = t - d_s\} \neq \emptyset$. Considering the powerful representation capacity of neural networks, a fully-connected neural network is used as the QoE predictor to approximate the *unknown* QoE function $h(x_{t,a})$. Let $f(x, \theta)$ be the QoE neural network with respect to input x and network parameter $\boldsymbol{\theta}$. The network can be trained by Algorithm 4.

The input of Algorithm 4 is prepared as follows. Denote the set of rounds whose delayed QoE feedback is received before round t as $\mathcal{T}_t = \bigcup_{i=0}^{t} \mathcal{T}_i^{\mathrm{r}}$. The training data of the QoE predictor neural network is an input matrix $\mathbf{X}_t$ of size $|\mathcal{T}_t| \times d$, with each row being the context vector x_{s,a_s} for $s \in \mathcal{T}_t$ and a $|\mathcal{T}_t|$-dimensional output vector $\mathbf{y}_t$, with each element being the corresponding received QoE feedback y_{s,a_s}, for $s \in \mathcal{T}_t$. With input $\mathbf{X}_t$ and $\mathbf{y}_t$, the QoE predictor is trained by, e.g., gradient descent with a loss function

$$L(\boldsymbol{\theta}_t) = \frac{1}{2} \sum_{s \in \mathcal{T}_t} (f(x_{s,a_s}; \boldsymbol{\theta}_t) - y_{s,a_s})^2 + \frac{1}{2} \lambda ||\boldsymbol{\theta}_t - \boldsymbol{\theta}_0||_2^2, \tag{2.3}$$

where $\boldsymbol{\theta}_0$ is the initialized neural network parameters.

With $\boldsymbol{\theta}_t$ being the updated QoE neural network parameter, the QoE at round t is predicted as $f(x_{t,a}, \boldsymbol{\theta}_t)$ where $x_{t,a}$ is the context which concatenates device features with DNN model features. To balance the exploitation and exploration without being trapped in a local optimum, Aquaman selects the DNN model to maximize the corresponding QoE

Algorithm 3 Aquaman: Online DNN Model Selection with Delayed QoE Feedback

Initialize Neural network parameter is initialized as $\boldsymbol{\theta}_0$. Let $\mathbf{Z}_0 = \lambda\mathbf{I}$. $\mathbf{X}_0$ and $\mathbf{y}_0$ are empty matrix and empty vectors.
for $t = 1, ..., T$ **do**
  if the set of feedback rounds at round t is $\mathcal{T}_t^{\mathbf{r}} \neq$ **then**
    $\forall s \in \mathcal{T}_t^{\mathbf{r}}$, append x_{s,a_s} to $\mathbf{X}_t$ and append y_{s,a_s} to $\mathbf{y}_t$.
    Update $\boldsymbol{\theta}_t$ by Algorithm 4;
    Update $\mathbf{Z}_t = \mathbf{Z}_{t-1} + \sum_{s \in \mathcal{T}_t^{\mathbf{r}}} \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)^T \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)$
  else
    $\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1}$; $\mathbf{Z}_t = \mathbf{Z}_{t-1}$
  end if
  if the set of feedback rounds before round t is $\mathcal{X}_t = \emptyset$ **then**
    Randomly choose a DNN model a_t;
  else
    Compute UCB $p_{t,a}$ for $a \in \mathcal{A}_t$ by Eqn. (2.9)
    Select DNN model $a_t = \arg\max_{a \in \mathcal{A}_t} p_{t,a}$;
  end if
end for

UCB, which is approximated as [131]:

$$p_{t,a} = f(x_{t,a}, \boldsymbol{\theta}_t) + \gamma_{t-1} \|\mathbf{g}(x_{t,a}, \boldsymbol{\theta}_t)\|_{\mathbf{Z}_t^{-1}} \tag{2.4}$$

where $\mathbf{g}(x_{t,a}, \boldsymbol{\theta}_t)$ is gradient of the neural network of the QoE predictor, $\mathbf{Z}_t = \lambda\mathbf{I} + \sum_{s \in \mathcal{T}_t} \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)^T \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)$, and γ_{t-1} is a hyperparameter to balance exploration (the first term) and exploitation (the second term): the larger γ_{t-1}, the more emphasis on exploration. Then, the DNN model is selected as:

$$a_t = \arg\max_{a \in \mathcal{A}_t} p_{t,a}, \tag{2.5}$$

where a_t is the selected model with the largest QoE UCB, and $\mathcal{A}_t$ is the candidate DNN

Algorithm 4 QoE Predictor Neural network Updating

1: **Input** Step size ρ, number of gradient descent steps G, current context matrix $\mathbf{X}_t$, current reward matrix $\mathbf{y}_t$, initialized network parameter $\boldsymbol{\theta}_0$
2: Compute the gradient $\nabla L(\boldsymbol{\theta}_t)$ of loss function $L(\boldsymbol{\theta}_t)$ in Eqn. (2.3);
3: **for** $j = 0, 1, 2, ..., G-1$ **do**
4: $\quad \boldsymbol{\theta}_t^{j+1} = \boldsymbol{\theta}_t^j - \rho \nabla L(\boldsymbol{\theta}_t^j)$
5: **end for**
6: **return** $\boldsymbol{\theta}_t = \boldsymbol{\theta}_t^G$.

model pool at time t.

2.4.4 Theoretical Analysis

We now prove an upper bound of the cumulative regret R_T achieved by Aquaman in Theorem 2.4.1, whose proof is available in the appendix.

Theorem 2.4.1 *Assuming that in neural bandit with delayed QoE feedback, the feedback delay satisfies $d_t \leq d_{\mathrm{m}}$, and neural network satisfying the assumptions in the appendix is used as the QoE predictor. Then, with probability $1-\delta, \delta \in (0,1)$, the cumulative regret of* Aquaman *satisfies*

$$R_T \leq 2\gamma_T \sqrt{T d_{\mathrm{m}} \left(2\tilde{d} \log(1 + \lfloor T/d_{\mathrm{m}} \rfloor K/\lambda) + 3\right)} + 2d_{\mathrm{m}}, \tag{2.6}$$

where $\lfloor \cdot \rfloor$ is the floor function, $\tilde{d}$ is the effective dimension of the neural tangent kernel matrix which is defined in Definition 4.3 of [131] and there exists a constant C such that $\gamma_T \leq C\nu\sqrt{\tilde{d}\log(1+TK/\lambda) + 2 - 2\log\delta}$ where ν is the parameter of sub-Gaussian noise.□

Theorem 2.4.1 extends the performance analysis of the standard neural bandit setting in [131] where no feedback delay is considered. The key point is that, despite the QoE feedback delay, the cumulative regret is sub-linear in T, meaning that Aquaman is asymptotically optimal in terms of the average QoE as $T \to \infty$. Like in the existing bandit learning literature [61, 131], the regret upper bound quantifies the *worst-case* QoE gap between Aquaman and the optimal oracle, and is mostly accurate in asymptotic cases when $T \to \infty$. Even when T is finite, however, we can empirically observe the shrinking gap between online learning-based Aquaman and the optimal oracle (i.e., the average QoE achieved by Aquaman is approaching that of the oracle).

Now, let us discuss how Aquaman handles delayed QoE feedback. For each round, if there is delayed feedback arriving, Aquaman accumulates it to the previous feedback history and uses the entire history as training data to update the QoE predictor. For the training data, input or feature is the context vectors for each previous rounds, while the labels are the corresponding (potentially delayed) QoE feedback. For the initial rounds when no feedback has arrived yet, Aquaman faces cold-start issues like any other recommendation systems and hence focuses more on exploration. When QoE feedback arrives, we can use it to supervise the learning of the QoE predictor. Naturally, the larger the feedback delay, the potentially worse the QoE (mostly at the beginning). Nonetheless, as time goes on, the impact of feedback delays also becomes less significant, because enough QoE feedback, albeit with delays, has been collected.

2.4.5 Practical Considerations

While Aquaman is provably-efficient with a sub-linear cumulative regret, it is established based on a set of simplifying assumptions for the sake of theoretical analysis. We now discuss some practical considerations. that can be addressed by extending Aquaman accordingly.

QoS constraint. In practice, a quality-of-service (QoS) constraint may be imposed in terms of, e.g., latency constraint for mobile inference. To handle the QoS constraint, we first estimate the resulting QoS-related performance for DNN models on an incoming device. Unlike QoE, the QoS-related performance does not depend on users' subjective evaluation, and hence can be measured offline based on the DNN developer's own device pool or estimated using performance models [34, 77, 115]. Then, Aquaman will only select those DNNs that meet the QoS constraint.

Updating QoE predictor online. For the ease of analysis, Algorithm 3 states that the QoE predictor is updated whenever new QoE feedback from an individual user is received. In practice, as described in Section 2.3, we can calculate the average QoE feedback from multiple users using the same device and DNN model and consider the average value as one QoE sample to reduce the variation due to individual users' subjective QoE values. Moreover, we can update the QoE predictor neural network using stochastic gradient decent whenever receiving a batch of average QoE values.

Selection of multiple DNNs. In practice, a mobile app may include multiple DNN models, each for one function in the app. This can be addressed by either viewing multiple DNN models as a "super-DNN" ensemble using our current design of Aquaman, or implementing multiple Aquaman in parallel each for one DNN selection. To reduce annoyance to users, the users' QoE feedback is shared for all the selected DNNs.

Malicious user feedback. For any online services, malicious user feedback is an unavoidable problem [86]. In practice, Aquaman can take the average of multiple individual QoE values as one data sample. Thus, the QoE predictor is expected to work well, provided that malicious users are not dominant. Additionally, this issue can be mitigated by using robust online learning [61] and/or removing likely malicious feedback via anomaly detection, which is beyond our focus.

Computational complexity and overhead. Aquaman serves as a middleman selection engine at the server or cloud side. There are many large-scale recommendation systems (e.g., YouTube recommender, Amazon recommender) running in the cloud and serving billions of user requests each day. These systems are also frequently updated based on users' feedback (e.g., click rates). Compared to them, DNN model selection in Aquaman occurs much less frequently, and training our QoS predictor has a much lower complexity. Thus, the computational complexity of Aquaman is affordable and less of a concern. The runtime overhead of Aquaman involves communications (for collecting device features) and QoE prediction. While the communications overheads are addressed by some bandit studies considering rate-limited channels [91], they are not a major concern for our problem. This is because we only need to collect device features (e.g., CPU speed, number of cores, RAM frequency and battery size) for each incoming request. Compared to downloading the DNN model itself which is typically in the order of megabytes or more, transmitting the device features to Aquaman takes negligible bandwidth. Additionally, upon receiving the device features, running the QoE predictor only takes one forward inference, which is in the order of milliseconds. Therefore, the total runtime overhead is negligible for Aquaman.

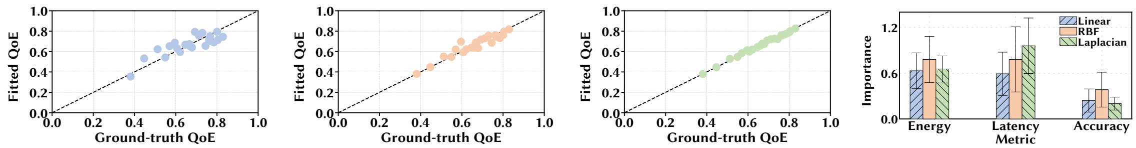
2.5 Evaluation Methodology

2.5.1 Experimental Setup

Experiment Platform. We set up our experiment platform by building an image classification app for Android based on the official example provided by Tensorflow Lite [47]. We can modify the app by deploying different pre-trained DNN models. We use Android Studio Profiler [46] to measure the energy and latency performance of a DNN running on a mobile device. For each DNN-device pair, we run more than 2,000 inferences to obtain average performance of energy consumption, inference latency, and inference accuracy. Some examples of performance profiling results for five DNN models and four mobile devices are shown in Fig. 2.1.

Devices and DNN Models. To profile DNN model runtime metrics and collect QoE data, we deploy six DNN models on four different devices. The models are: MobileNet_V2_1.0_224_quant, Inception_V3_quant, MobileNet_V2_1.0_224_quant, MobileNet_V1_0.75_192_quant, Inception_V4, and MobileNet_V2_1.0_224. The devices are Vankyo Matrixpad z1, Samsung Galaxy Tab A, Samsung Galaxy Tab S5e, and Vivo V1838A.

QoE Predictor. Our QoE predictor consists of four fully-connected layers. It takes features of DNN-device pair as input, and outputs the predicted QoE. The input features include device features (i.e., CPU speed, number of cores, RAM size, RAM frequency and battery size) and DNN model features (i.e., million MACs, million parameters, model size, number of nodes and number of layers). After predicting the QoE, the QoE UCB can be computed to select the DNN model according to Eqn. 2.5. We train the QoE predictor



(a) Linear regression (b) Kernel ridge regression (RBF) (c) Kernel ridge regression (Laplacian) (d) Metric importance

Figure 2.4: (a)(b)(c) Average normalized QoE fitting results for three regression methods, compared to the average QoE of 15 users. RBF: $\alpha = 0.001$ and $\gamma = 4$. Laplacian: $\alpha = 0.01$ and $\gamma = 1$. (d) Importance of different performance metrics.

for 100 epochs for every 100 QoE feedback collected, and our learning rate is set as 0.001.

Datasets. We conduct two types of experiments: experiment on the actually collected user QoE data, and experiment on our synthetic data. For our collected QoE dataset, we run Aquaman for T=20000 rounds. In each round, one of four mobile devices arrives randomly. More details of conducting our user survey, and utilizing the survey results to build the training dataset and gain QoE labels are introduced in Sections 2.5.2 and 2.6.1, respectively. Furthermore, our methodology to synthesize the dataset is presented in Section 2.5.3.

2.5.2 User Study

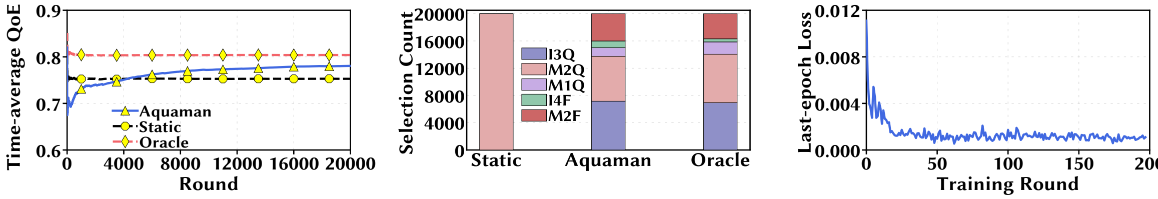
As stated in Section 2.2, we recruit 15 participants into our survey to test six different DNN models on four different mobile devices (i.e., a total of 24 DNN-device pairs for each participant) and provide QoE feedback in the form of numeric ratings in 1-10. Each participant is asked to use each of the DNN models for a while and then provide a numerical rating for its QoE with each device-model pair on a scale of 1-10. The survey result is shown in Fig. 2.2.

2.5.3 Generation of Synthetic Data

Because of limited resources, it is practically impossible for us to evaluate Aquaman on a large scale, which requires interaction with many more mobile users. To circumvent this hurdle, we resort to synthetic data generated as follows.

The idea for generating synthetic QoE values is to utilize our user study to build a synthetic QoE generator that outputs user's average QoE given a DNN-device pair. This synthetic QoE generator is unknown to Aquaman during the evaluation. Specifically, we first build a synthetic performance generator based on our profiling results, and then feed the synthetic performance values to our synthetic QoE generator for producing QoE. The reason we use this indirect approach to generating synthetic QoE instead of directly utilizing DNN-device features as input is that the QoE data we have is not adequate. We consider three widely-considered performance metrics — average energy consumption, inference latency, and inference accuracy — in our synthetic data generation. That is, we mainly consider that the users' QoE depends on these three runtime metrics of a DNN model on a device, which further depend on features of the device-model pair. Thus, the QoE is a function of the DNN and device features. While other factors such as latency variability can affect a user's QoE, they are essentially captured by the noise term in our QoE function in Eqn. (2.1).

To build our synthetic performance generator, we augment the energy/latency performance predictor used the existing research [34,115] by including device features as additional input. Here, the device features include CPU speed, number of cores, RAM size, RAM frequency and battery size, while additional features such as GPU and operating system version can also be added. To represent DNN models, one can use high-level features such as million

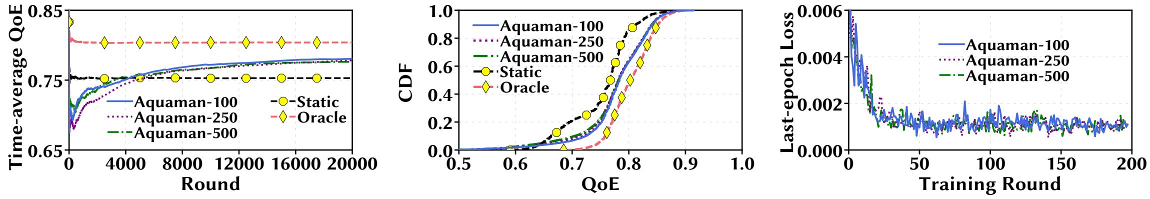


(a) Time-average QoE (b) Selection count of each DNN. (c) Training Loss

Figure 2.5: Experiment on collected data. Feedback delay is uniformly distributed on 1-100 rounds.

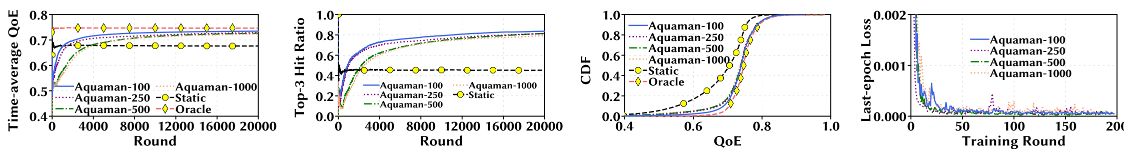
MACs, million parameters, model size, number of nodes and number of layers. Alternatively, a finer-grained DNN embedding representation on a block basis can also be used as input for performance generator.

We use kernel ridge regression to fit the relationship between DNN-device features and the resulting performance metrics due to limited mobile devices we have. Next, based on the energy/latency/accuracy values, we build the synthetic QoE generator denoted by $QoE = m_{QoE}(energy, latency, accuracy)$. Again, because of limited QoE data we have, we consider regression methods to approximate $QoE = m_{QoE}(energy, latency, accuracy)$: linear regression, and kernel ridge regression with RBF and Laplacian kernels. Fig. 2.4 shows the average QoE fitting results, along with importance of different performance metrics. We see that Laplacian kernel ridge regression can almost fully fit into our limited average QoE data, whereas linear regression performs poorly (which also implies that the existing DNN design maximizing linear combination of accuracy, energy and latency [34, 115] may not lead to optimal QoE). We use permutation importance (a.k.a. mean decrease accuracy) as the indicator of feature importance [18, 41]. Fig. 2.4(d) shows that different regression methods impose different feature importance, but all the three methods gives the top priority to the



(a) Time-average QoE (b) QoE CDF (c) Training loss

Figure 2.6: Experiment on collected data. "Aquaman-x" means the QoE feedback delay is uniformly distributed between 1 and x.



(a) Time-average QoE (b) Ratio of selecting top-3 DNNs. (c) QoE CDF (d) Training Loss

Figure 2.7: Simulation on synthetic data. "Aquaman-x" means the QoE feedback delay is uniformly distributed between 1 and x.

latency metric feature.

Finally, we note that Aquaman is oblivious of the QoE generator we use; instead, it tries to learn it online (and learn the true user QoE if deployed in the real world).

2.5.4 Baseline Approaches

We consider the following representative baselines.

- Static: It selects a single DNN model regardless of the actual mobile devices. The single DNN model is chosen such that the average QoE of all the users is maximized. The assumption is that Static knows a priori which model will result in the maximum average QoE of all the users, and selects that one in all rounds regardless of incoming devices.

- Oracle: The oracle is assumed to know the best DNN that results in the highest expected QoE for each mobile device. In reality, there does not exist such an oracle, and no practical algorithms can outperform the oracle in terms of the expected QoE for mobile inference.

In principle, the existing device-aware DNN optimization/selection approaches [3, 24, 34, 51, 71, 72, 79, 82, 85, 106, 115, 116] could also result in the QoE-optimal DNN model, had the exact QoE function been known for each mobile device in advance and used as the optimization objective. Nonetheless, this is equivalent to assuming an Oracle (one of our baselines). On the other hand, if we fix a proxy objective function in the DNN selection process to select a single DNN, it can be even worse than Static, since the proxy objective function may not reflect the actual QoE whereas Static directly optimize for the average QoE. For these reasons, we do not compare Aquaman against the existing approaches that focuses on optimizing (proxy) objective functions. Our two baselines, Oracle and Static, cover the ideal case and a fairly strong case.

2.6 Evaluation Results

We conduct two types of experiments: on the actually collected user QoE, and on our synthetic dataset.

2.6.1 Experiment on the User Study

We first evaluate Aquaman using the collected QoE data from our user study. We exclude the NASNet_large model from consideration as it is overly slow and has the worst QoE on three of the four devices in our experiment. Thus, for each device, Aquaman selects one out of five pre-trained DNN models. Instead of using average QoE of 15 users as one sample, we utilize bootstrapping to randomly pick 8 samples out of 15 QoE data each time and use their average value as one ground-truth QoE sample for training. We run Aquaman for T=20000 rounds. For each round (with a given device), the QoE for each of the five DNN-device pairs is calculated as the average of eight random samples out of 15 QoE data collected by our user study. The training details and other settings are reported in Section 2.5.

We present the results in Figs. 2.5 and 2.6(a), where Fig. 2.5 is for the case when users' QoE feedback delay is distributed uniformly between 1 and 100, while Fig. 2.6(a) shows more results on different feedback delays. While the absolute value of QoE improvement is subjective, we see clearly that Aquaman gradually improves the time-average QoE, approaching Oracle and outperforming Static. The count breakdown of every single DNN model in Fig. 2.5(b) shows that the distribution of selected DNNs by Aquaman is similar to that of Oracle. As for the training loss of our QoE predictor, we show in Fig. 2.5(c) the last-epoch loss. The total training rounds is about 200 because we train the neural network whenever the number of collected QoE feedback reaches 100 and the total learning rounds is T=20000. Additionally, in Figs. 2.5(a), 2.6(a) and 2.6(b), the time-average QoE is calculated in terms of all devices. As Aquaman takes the device features into account when selecting DNN models, it also achieves higher QoE than Static for each individual type of device.

To see if Aquaman can handle longer QoE feedback delay effectively, we show in Fig. 2.6 the time-average QoE, CDF of QoE, and last-epoch training loss of Aquaman under increasingly larger feedback delay – uniformly distributed delays of 1-100, 1-250 and 1-500

– respectively. It shows that Aquaman is not sensitive to the feedback delay and still better than Static.

2.6.2 Simulation on Synthetic Data

Synthesising DNNs and Mobile Devices

For synthetic evaluation, each DNN is represented by a feature vector, and so is each mobile device. Thus, by randomly generating device features, we simulate the arrival process of different mobile devices. Concretely, besides the five DNN models used in Section 2.6.1, we synthesize another 10 DNN models. We still run Aquaman for T=20000 rounds, in each of which we synthesize a device feature vector to denote a group of mobile devices. Combining the device feature and with DNN feature as input, our synthetic QoE generator produces the average QoE value. To mimic real cases, we will also add noise to the synthetic QoE value.

Results

We show the results in Fig. 2.7, by considering that the synthetic QoE is generated using the RBF kernel ridge regression (Section 2.5.3). The time-average QoE and QoE CDF achieved by Aquaman gradually approaches those of Oracle with an increasingly smaller gap. The reason is that by exploiting the history information, Aquaman can learn the QoE predictor neural network parameter $\boldsymbol{\theta}_t$ with an improved accuracy over time, which is helpful for future DNN model selection. On the contrary, Static constantly yields the lowest QoE, because it does not adapt its DNN model selection to an incoming mobile device's feature. We also show in Fig. 2.7(b) the percentage of time when the selected DNN is among the top-3 models in terms of the average QoE. It can be observed that Aquaman is much more likely to select a top-3 DNN model than Static. Note that Aquaman keeps on exploring in order to avoid be trapped in local optimum and keep up with the potentially changing QoE statistics. Hence, the top-3 DNN hit ratio is increasing by using Aquaman but not 100%. Like in Section 2.6.1, we see that Aquaman is not sensitive to the QoE feedback delays.

We also consider the case when the synthetic QoE values are generated using the Laplacian kernel and a random mixture of two different kernels (Section 2.5.3). The results are similar: because of the strong prediction power of neural networks, Aquaman learn the average QoE for a DNN-device pair over time and hence gradually improve the average QoE for mobile inference. Thus, we omit the results for space limitation.

2.6.3 Complexity Analysis

As discussed in Section 2.4.5, Aquaman is deployed in the cloud with an affordable computational complexity in practice. More concretely, the QoE predictor consists of four fully-connected layers in our experiment, and is updated for 100 epochs whenever a batch of 100 QoE feedback values are collected. The training is very fast and each takes less than 10 seconds on Google Colab platform configured with a regular CPU. For online inference, given each incoming device, we only need to run one forward inference, which is in the order of milliseconds. Even considering large-scale deployment, due to the smaller input features and much less frequent model updating, Aquaman is still much less computationally demanding than industry-level recommendation systems that are frequently updated and serve tens of thousands of users every minute.

2.7 Related Work

To turn DNN-based mobile inference into reality, it is crucial to reduce the size of otherwise overly large and computationally prohibitive DNN models by using efficient model compression techniques, such as pruning and quantization [83], matrix factorization [36], compact convolution filters [54], and knowledge distillation [100], among many others. Furthermore, automated neural architecture search is also necessary to identify an appropriate network architecture for mobile inference [115]. The survey [113] comprehensively covers accelerating the training process for large machine learning models in IoT. These studies are complementary to Aquaman: the lightweight DNN models produced by these studies can be included into the DNN pool and selected by Aquaman for QoE-optimal mobile inference.

For device-aware DNN optimization, latency/energy predictors have been utilized for optimization speed-up [20, 115]. : pre-train a machine learning model to predict the resulting latency/energy for a candidate DNN on the target device. Nonetheless, these average latency/energy performance predictors do not incorporate device features. As a result, for every new device, new latency/energy predictors need to be built, which can be time-consuming. More crucially, these studies as well as other relevant approaches [72] focus on optimizing an objective function, which may not improve the users' actual QoE. By contrast, we advocate a scalable user-centric DNN selection approach which keeps users into a closed loop and leverages their QoE feedback to optimize DNN selections.

There have also been studies on runtime DNN model selection/adaptation in view of time-varying environmental/input conditions [90, 109]. While they focus on dynamic selection/adaptation of already-deployed DNN models on mobile devices, Aquaman is different and focuses on DNN model selection during the deployment stage. Moreover, [92] studies model selection and switching between a subset of the machine learning models from a superset of models for Industrial IoT. The goal is to maximize the level of model trustworthiness, which is orthogonal to our study.

Bandit is a classic online learning setting [16, 25, 65, 99], and our work extends the neural bandit [131] by considering delayed feedback. A recent study [75] briefly addresses DNN selection for mobile inference using a linear function as a toy example. By contrast, we propose a provably-efficient online algorithm, leverage a neural network-based QoE predictor with strong representation power, and conduct both experiments and synthetic simulations for evaluation. In a different context, [127] proposes to build a neural network to predict user QoE for video applications, whereas we not only predict QoE but also propose a bandit algorithm to balance exploration and exploitation.

2.8 Conclusion

In this chapter, we propose an automated and user-centric DNN selection engine, called Aquaman, which leverages QoE feedback to optimize DNN selection decisions. The core of Aquaman is a neural network-based QoE predictor, which is updated online based on QoE feedback. More specifically, Aquaman consists of two integrated parts: QoE prediction and DNN model selection. To balance exploitation and exploration, Aquaman selects

DNN models for diverse devices based on the QoE UCB, resulting in provably-efficient QoE performance compared to the oracle. Finally, we evaluate Aquaman on a user study as well as synthetic simulations. We demonstrate the effectiveness of Aquaman by showing that it outperforms the static DNN selection approach while being close to the oracle in terms of users' QoE.

Appendix

Proof of Theorem 2.4.1

Assumptions. Without loss of generality, we assume that each hidden layer has the same width m, and the parameter matrix of layer l at time t is $\mathbf{W}_{l,t}$, then the dimension of $\mathbf{W}_{l,t}$ is $\mathbf{W}_{1,t} \in \mathbb{R}^{d\times m}, \mathbf{W}_{l,t} \in \mathbb{R}^{m\times m}$ for $l = 2, ..., L-1$, and $\mathbf{W}_{L,t} \in \mathbb{R}^{m\times 1}$. Assume that the width m can be sufficiently large. In the neural network, each hidden layer is followed by activation operations $\sigma(\cdot)$, such as Rectified Linear Unit (ReLU) function defined as $\sigma(x) = \max(0, x)$. Thus the predicted QoE of arm a in round t is

$$f(x_{t,a}; \boldsymbol{\theta}_t) = \sqrt{m}\sigma(\sigma(\sigma(\sigma(x_{t,a}\mathbf{W}_{t,1})\mathbf{W}_{t,2}) \cdots)\mathbf{W}_{t,L-1})\mathbf{W}_{t,L}. \tag{2.7}$$

We vectorize each of $\mathbf{W}_{l,t}$ and get vector $\mathbf{W}'_{l,t} \in \mathbb{R}^{k\times 1}$, where $k = m \times d$ if $l = 1$, $k = m \times m$ if $l = 2, ..., L-1$, and $k = m$ if $l = L$. Thus neural network parameter at round t in Algorithm 4 can be written as $\boldsymbol{\theta}_t = [\mathbf{W}'_{1,t}; \mathbf{W}'_{2,t}; ...; \mathbf{W}'_{L,t}] \in \mathbb{R}^{(m\times d + m\times m + ... + m\times m + m)\times 1}$.

The same initialization method of the neural network as in [131] is adopted. $\mathbf{W}_{l,0}$

is initialized as $\begin{pmatrix} \mathbf{W} & \mathbf{0} \\ \mathbf{0} & \mathbf{W} \end{pmatrix}$ for $0 \leq l \leq L-1$ with each entry of $\mathbf{W}$ sampled from Gaussian distribution $\mathcal{N}(0, 4/m)$. $\mathbf{W}_{L,0}$ is initialized as $\left[\mathbf{w}^T, -\mathbf{w}^T\right]$ with each entry of $\mathbf{w}$ sampled from Gaussian distribution $\mathcal{N}(0, 2/m)$. And thus the initialized neural network parameters $\boldsymbol{\theta}_0$ is acquired.

The neural network is trained by gradient descent with loss function

$$L(\boldsymbol{\theta}_t) = \frac{1}{2}\sum_{s \in \mathcal{T}_t}(f(x_{s,a_s}; \boldsymbol{\theta}_t) - y_{s,a_s})^2 + \frac{1}{2}m\lambda||\boldsymbol{\theta}_t - \boldsymbol{\theta}_0||_2^2, \tag{2.8}$$

where $\boldsymbol{\theta}_0$ is the initialized neural network parameters and m is the width of the neural network. UCB for each arm $p_{t,a}$ in algorithm 3 can be computed as:

$$p_{t,a} = f(x_{t,a}, \boldsymbol{\theta}_t) + \gamma_{t-1}\|\mathbf{g}(x_{t,a}, \boldsymbol{\theta}_t)/\sqrt{m}\|_{\mathbf{Z}_t^{-1}} \tag{2.9}$$

where $\mathbf{g}(x_{t,a}, \boldsymbol{\theta}_t)$ is gradient of the neural network of the QoE predictor,

$\mathbf{Z}_t = \sum_{s \in \mathcal{T}_t} \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)^T \mathbf{g}(x_{s,a}; \boldsymbol{\theta}_t)/m$, and the parameter γ_t is set in the same way as the exploration rate in Algorithm 1 of [131] to get a provable sub-linear regret.

Proof. Since the maximum feedback delay is d_{m}, we consider the cumulative regret of the first d_{m} rounds and the cumulative regret from $d_{\mathrm{m}}+1$ to T rounds separately. That means the regret can be written as:

$$\begin{aligned} R_T &= \sum_{t=1}^{T}[h(x_{t,a_t^*}) - h(x_{t,a_t})] \\ &= \sum_{t=1}^{d_{\mathrm{m}}}[h(x_{t,a_t^*}) - h(x_{t,a_t})] + \sum_{t=d_{\mathrm{m}}+1}^{T}[h(x_{t,a_t^*}) - h(x_{t,a_t})] \end{aligned} \tag{2.10}$$

Since we assume that QoE of any model satisfies $0 \leq h(x_{t,a}) \leq 1$, the starting regret $\sum_{t=1}^{d_{\mathrm{m}}}[h(x_{t,a_t^*}) - h(x_{t,a_t})] \leq d_{\mathrm{m}}$. Denote the continuing regret as $R_T^c = \sum_{t=d_{\mathrm{m}}+1}^{T}[h(x_{t,a_t^*}) - h(x_{t,a_t})]$. According to Lemma 5.3 in [131], with probability $1-\delta, delta \in (0,1)$, we have

$$
\begin{aligned}
R_T^c \leq 2 \sum_{t=d_{\mathrm{m}}+1}^{T} \gamma_t \min\left\{\left\|\mathbf{g}(x_{t,a_t};\boldsymbol{\theta}_t)/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}}, 1\right\} \\
+ C_1\left(Sm^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{7}{6}}\lambda^{-\frac{1}{6}}L^{\frac{7}{2}} + m^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{5}{3}}\lambda^{-\frac{2}{3}}L^{3}\right),
\end{aligned}
\tag{2.11}
$$

where C_1 is a constant and S is a constant about the neural tangent kernel matrix defined in Theorem 4.5 of [131]. So the challenge is to bound $\sum_{t=d_{\mathrm{m}}+1}^{T}\|\mathbf{g}(x_{t,a};\theta_t)/\sqrt{m}\|_{\mathbf{Z}_t^{-1}}$ where $Z_t = \lambda\mathbf{I} + \sum_{s\in\mathcal{T}_t}\mathbf{g}\left(x_{s,a_s},\theta_{s+d_s}\right)\mathbf{g}^T\left(x_{s,a_s},\theta_{s+d_s}\right)/m$ in the case of delayed feedback. For compactness, we denote gradients as $\mathbf{g}_t = \mathbf{g}(x_{t,a_t};\theta_t)$ and delayed gradients as $\bar{\mathbf{g}}_t = \mathbf{g}(x_{t,a_t};\theta_{t+d_t})$.

Devide the $T - d_{\mathrm{m}}$ rounds into d_{m} groups, each with $I = \frac{T-d_{\mathrm{m}}}{d_{\mathrm{m}}}$ elements. Thus, the nth round set, $n \in \mathbb{Z}^+, n \in [1, d_{\mathrm{m}}]$, is $\Omega^n = \{d_{\mathrm{m}}+n, 2d_{\mathrm{m}}+n, \cdots, Id_{\mathrm{m}}+n\}$. Correspondingly, the delayed gradients are also devided into d_{m} groups, each with I elements. In this way, the nth context group is $\left\{\mathbf{g}_{d_{\mathrm{m}}+n}, \mathbf{g}_{2d_{\mathrm{m}}+n}, \cdots, \mathbf{g}_{Id_{\mathrm{m}}+n}\right\}$. For each group n, define I matrices as

$$
\begin{aligned}
V_i^n = \lambda\mathbf{I} + \sum_{s=1}^{i}\bar{\mathbf{g}}_{sd_{\mathrm{m}}+n}\bar{\mathbf{g}}_{sd_{\mathrm{m}}+n}^T/m, \\
i, n \in \mathbb{Z}^+, \quad n \in [1, d_{\mathrm{m}}], \quad i \in [1, I].
\end{aligned}
\tag{2.12}
$$

By directly using Lemma 11 in [1], there exists a constant C_2 such that

$$\begin{aligned}\sum_{i=1}^{I} \min\left\{\|\bar{\mathbf{g}}_{id_{\mathrm{m}}+n}/\sqrt{m}\|^2_{(V^n_{i-1})^{-1}}, 1\right\} &\leq 2\log\frac{\det(V^n_I)}{\det(\lambda\mathbf{I})} \\ &\leq 2\tilde{d}\log(1+IK/\lambda) + 2 + C_2 m^{-1/6}\sqrt{\log m}L^4T^{5/3}\lambda^{-1/6}\end{aligned} \tag{2.13}$$

where K is the number of candidate arms, the second inequality is from Lemma 5.4 in [131] and $\tilde{d}$ is the effective dimension of the neural tangent kernel matrix which is defined in Definition 4.3 of [131].

Since the feedback delay is not larger than d_{m}, we have $\forall t > d_{\mathrm{m}}, \mathcal{T}_{t-d_{\mathrm{m}}} \subseteq \mathcal{T}_t$. Let $\Omega^n_i = \{d_{\mathrm{m}}+n, 2d_{\mathrm{m}}+n, \cdots, id_{\mathrm{m}}+n\}$ for $i, n \in \mathbb{Z}^+, i \leq I$. If $t = id_{\mathrm{m}}+n$, then $\Omega^n_{i-1} \subset \mathcal{T}_{t-d_{\mathrm{m}}} \subseteq \mathcal{T}_t$. Since Z_t and V^n_i are both positive-definite matrix, for $t = id_{\mathrm{m}}+n$, $i \leq I$, we have

$$\begin{aligned}\|\bar{\mathbf{g}}_t/\sqrt{m}\|_{Z_t^{-1}} &= \bar{\mathbf{g}}_t^T\left(\lambda\mathbf{I} + \sum_{s\in\mathcal{T}_t}\bar{\mathbf{g}}_s\bar{\mathbf{g}}_s^T/m\right)^{-1}\bar{\mathbf{g}}_t/m \\ &\leq \bar{\mathbf{g}}_t^T\left(\lambda\mathbf{I} + \sum_{s\in\Omega^n_{i-1}}\bar{\mathbf{g}}_s\bar{\mathbf{g}}_s^T/m\right)^{-1}\bar{\mathbf{g}}_t/m \\ &= \|\bar{\mathbf{g}}_t/\sqrt{m}\|_{(V^n_{i-1})^{-1}}\end{aligned} \tag{2.14}$$

Therefore, we have the following

$$\begin{aligned}\sum_{t=d_{\mathrm{m}}+1}^{T}||\bar{\mathbf{g}}_t/\sqrt{m}||^2_{Z_t^{-1}} &= \sum_{n=1}^{d_{\mathrm{m}}}\sum_{i=1}^{I}\|\bar{\mathbf{g}}_{id_{\mathrm{m}}+n}/\sqrt{m}\|_{\mathbf{Z}^{-1}_{id_{\mathrm{m}}+n}} \\ &\leq \sum_{n=1}^{d_{\mathrm{m}}}\sum_{i=1}^{I}\|\bar{\mathbf{g}}_{id_{\mathrm{m}}+n}/\sqrt{m}\|_{(V^n_{i-1})^{-1}}.\end{aligned} \tag{2.15}$$

By Eqn. (2.13), we have

$$
\begin{aligned}
&\sum_{t=d_{\mathrm{m}}+1}^{T} \min\left\{ \|\bar{\mathbf{g}}_t/\sqrt{m}\|_{Z_t^{-1}}^2, 1 \right\} \\
\leq & \sum_{n=1}^{d_{\mathrm{m}}} \sum_{i=1}^{I} \min\left\{ \|\bar{\mathbf{g}}_{id_{\mathrm{m}}+n}/\sqrt{m}\|_{\left(V_{i-1}^n\right)^{-1}} \right\} \\
\leq & d_{\mathrm{m}} \left(2\tilde{d}\log(1+IK/\lambda) + 2 + C_2 m^{-1/6}\sqrt{\log m} L^4 I^{5/3} \lambda_{-1/6} \right)
\end{aligned}
$$

Next, since $\|\mathbf{g}_t\|_{\mathbf{Z}_t^{-1}} \leq \|\bar{\mathbf{g}}_t\|_{\mathbf{Z}_t^{-1}} + \|\mathbf{g}_t - \bar{\mathbf{g}}_t\|_{\mathbf{Z}_t^{-1}}$ according to triangle ineuqality, it is necessary to bound $\|\mathbf{g}_t - \bar{\mathbf{g}}_t\|_{\mathbf{Z}_t^{-1}}$.

By triangle inequality, with probability at least $1-\delta$, there exists a constant C_3 such that

$$
\begin{aligned}
\|\mathbf{g}_t - \bar{\mathbf{g}}_t\|_2 &\leq \|\mathbf{g}_t - \mathbf{g}\left(x_{t,a_t}, \theta_0\right)\|_2 + \|\bar{\mathbf{g}}_t - \mathbf{g}\left(x_{t,a_t}, \theta_0\right)\|_2 \\
&\leq C_3\sqrt{\log m}\left(\tau_1^{1/3} + \tau_2^{1/3}\right) L^3 \left\|\mathbf{g}\left(x_{t,a_t}, \theta_0\right)\right\|_2
\end{aligned}
$$

where $\tau_1 = 2\sqrt{t/(m\lambda)}$ and $\tau_2 = 2\sqrt{(t+d_t)/(m\lambda)}$, and the second inequality holds due to Lemma B.5 in [131]. Further, since the maximum eigenvalue of Z_t^{-1} is λ^{-1}, we have probability at least $1-\delta$,

$$
\begin{aligned}
&\left\|\left(\mathbf{g}_t - \bar{\mathbf{g}}_t\right)/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}} \\
\leq & \lambda^{-1/2}\left\|\left(\mathbf{g}_t - \bar{\mathbf{g}}_t\right)/\sqrt{m}\right\|_2 \\
\leq & \lambda^{-1/2} C_2 \sqrt{\log m}\left(\tau_2^{1/3}\right) L^{7/2} \\
\leq & 2C_3 m^{-1/6}\sqrt{\log m}\left(t + d_{\mathrm{m}}\right)^{1/6} \lambda^{-2/3} L^{7/2},
\end{aligned}
\tag{2.16}
$$

where the second inequality comes from Eqn. (2.16), Lemma B.6 in [131] and the fact that

$\tau_2 \geq \tau_1$. Now, with probability at least $1-\delta$, we have

$$\begin{aligned}
&\sum_{t=d_{\mathrm{m}}+1}^{T} \min\left\{\left\|\mathbf{g}_t/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}}^2, 1\right\} \\
\leq &\sum_{t=d_{\mathrm{m}}+1}^{T} \left[\min\left\{\left\|\bar{\mathbf{g}}_t/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}}^2, 1\right\} + \left\|\left(\mathbf{g}_t - \bar{\mathbf{g}}_t\right)/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}}^2\right] \\
\leq &d_{\mathrm{m}}\left(2\tilde{d}\log(1+IK/\lambda) + 2 + C_2 m^{-1/6}\sqrt{\log m} L^4 I^{5/3}\lambda^{-1/6}\right) \\
&+ 2C_3 m^{-1/6}\sqrt{\log m}\left(T+d_{\mathrm{m}}\right)^{7/6}\lambda^{-2/3}L^{7/2}.
\end{aligned}$$

By Eqn. (2.11), we have with probability at least $1-\delta$,

$$\begin{aligned}
R_T^c \leq\ & 2\sqrt{T}\gamma_T\sqrt{\sum_{t=d_{\mathrm{m}}+1}^{T} \min\left\{\left\|\mathbf{g}(x_{t,a_t};\boldsymbol{\theta}_t)/\sqrt{m}\right\|_{\mathbf{Z}_t^{-1}}^2, 1\right\}} \\
&+ C_1\left(Sm^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{7}{6}}\lambda^{-\frac{1}{6}}L^{\frac{7}{2}} + m^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{5}{3}}\lambda^{-\frac{2}{3}}L^3\right) \\
&\leq 2\gamma_T\sqrt{Td_{\mathrm{m}}\left(2\tilde{d}\log(1+IK/\lambda)+2+U\right)} \\
&+ 2\gamma_T\sqrt{2TC_3 m^{-1/6}\sqrt{\log m}\left(T+d_{\mathrm{m}}\right)^{7/6}\lambda^{-2/3}L^{7/2}} \\
&+ C_1\left(Sm^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{7}{6}}\lambda^{-\frac{1}{6}}L^{\frac{7}{2}} + m^{-\frac{1}{6}}\sqrt{\log m}T^{\frac{5}{3}}\lambda^{-\frac{2}{3}}L^3\right) \\
&\leq 2\gamma_T\sqrt{Td_{\mathrm{m}}\left(2\tilde{d}\log(1+IK/\lambda)+3\right)} + 1,
\end{aligned}$$

where $U = C_2 m^{-1/6}\sqrt{\log m}L^4 I^{5/3}\lambda^{-1/6}$ and the third equation holds with a sufficiently large m.

Remind that $I = \lfloor T/d_{\mathrm{m}} \rfloor$, we have with probability at least $1-\delta$,

$$\begin{aligned}
R_T &= \sum_{t=1}^{d_{\mathrm{m}}}[h(x_{t,a_t^*}) - h(x_{t,a_t})] + R_T^c \\
&\leq 2\gamma_T\sqrt{Td_{\mathrm{m}}\left(2\tilde{d}\log\left(1+\lfloor T/d_{\mathrm{m}}\rfloor K/\lambda\right)+3\right)} + 2d_{\mathrm{m}}.
\end{aligned} \tag{2.17}$$

By Lemma 5.4 in [131], there exists a constant C_4 such that $\gamma_T \leq C_4 \nu \sqrt{\tilde{d} \log (1 + TK/\lambda) + 2 - 2 \log \delta}$ where ν is the parameter of sub-Gaussian noise.

Chapter 3

One Proxy Device Is Enough for Hardware-Aware Neural Architecture Search

3.1 Introduction

Convolutional neural networks (CNNs) are a most commonly used class of deep neural networks, offering human-level inference accuracy for numerous real-world applications such as vision-based autonomous driving and video content analysis [45]. Going beyond the contentional server-only platforms, CNNs have been deployed on increasingly diverse devices and platforms, including mobile, ASIC and edge devices [121]. As the foundation of a CNN, the neural architecture can greatly affect the resulting model performance such as accuracy, latency, and energy consumption. Thus, optimizing the architecture through hardware-aware neural architecture search (NAS) is crucial and being actively studied [13, 15, 30, 107, 108, 120].

The exponentially large search space consisting of billions of or even more architectures renders NAS a very challenging task [34, 107, 108, 115, 120, 125]. The key reason is that evaluating and ranking the architectures in terms of metrics of interest (e.g., accuracy and latency) can be extremely time-consuming. As a result, many studies have been focused on reducing the cost[1] of training and evaluating the architecture accuracy, including reinforcement learning-based NAS with accuracy evaluated based on a small proxy dataset [133], differentiable NAS [120], one-shot or few-shot NAS [12, 23, 130], NAS assisted with an accuracy predictor [34, 115], among many others.

In addition to speeding up accuracy evaluation, reducing the cost of assessing the inference latency on a target device is equally important for efficient hardware-aware NAS [23, 40, 79, 107]. The naive method of measuring the latency for each architecture can lead to a total search time exceeding several weeks or even months, whereas using the floating-point operations (FLOPs) as a device-agnostic proxy may not accurately reflect the true inference latency on different devices [107]. As a result, state-of-the-art (SOTA) hardware-aware NAS has mainly relied on device-specific latency lookup tables or predictors [13, 15, 24, 30, 34, 40, 115, 125].

Nonetheless, building a latency predictor for a target device requires significant engineering efforts and can be very slow. For example, [24] measures average inference latencies for 5k sample DNNs on a mobile device and uses the results to build a latency

[1] In this chapter, "cost" also interchangeably refers to computational complexity: a higher complexity requires more computational resources (measured in, e.g., machine hours) and hence a higher monetary cost, too.

lookup table for that specific device. Assuming the ideal scenario of 20 seconds for each measurement (to average out randomness per the TensorFlow guideline [47]) and non-stop measurement, it can take 27+ hours to build the latency predictor for one single device [24]. Similarly, it is reported by [34] that 350k records are collected for building a latency predictor for just one device. Even by measuring latencies on six devices in parallel, the authors of [64] report on OpenReview that they spent *one month* to collect latency data on the small NAS-Bench-201 space and build latency predictors for another two datasets on the FBNet space. More recently, kernel-level latency predictors that capture complex processing flows of different neural execution units are proposed, but it takes up to 4.4 days for just collecting the latency measurements on one edge device [129]. All these facts highlight the crucial point that building a latency predictor for a target device — a key step of SOTA hardware-aware NAS — is costly and cannot be taken for granted as free lunch.

Worse yet, the target devices for CNN deployment are extremely diverse, ranging from mobile CPUs, ASIC, edge devices to GPUs. For example, even for the mobile devices alone, as shown in Fig. 3.1, there are more than two thousand system-on-chips (SoCs) available in the market, and only top 30 SoCs can each have over 1% of the share [121]. Importantly, the diverse set of devices have different latency collection pipelines, programming environment, and/or hardware domain knowledge requirement [64]. Thus, in the presence of extremely diverse tar-

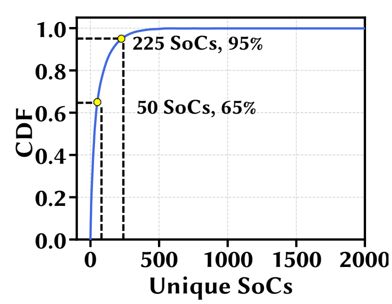


Figure 3.1: Device statistics for Facebook users as of 2018 [121].

get devices, the combined cost of building device-specific latency predictors for hardware-aware NAS is prohibitively high and increasingly becoming a key bottleneck for scalable hardware-aware NAS. In addition, this challenge is further magnified by the fact that building device-specific latency predictors is *not* a one-time cost: varying the input resolution and/or output classes also requires new latency predictors (e.g., two device-specific latency predictors are built, each for one dataset, on the FBNet space [64]). Consequently, how to efficiently scale up hardware-aware NAS for extremely diverse target devices has arisen as a critical challenge.

Contributions. In this chapter, we focus on reducing the total latency evaluation cost for scalable hardware-aware NAS in the presence of diverse target devices across different platforms (e.g., mobile platform, FPGA platform, desktop/server GPU, etc.). Concretely, we show that latency monotonicity commonly exists among different devices, especially devices of the same platform. Informally, latency monotonicity means that the ranking orders of different architectures' latencies are correlated on two or more devices. Thus, with latency monotonicity, building a latency predictor for just one device that serves as a proxy — rather than for each individual target device as in state of the art [23,34,64] — is enough. Even when a target device has a weak monotonicity with the default proxy device (e.g., a mobile phone proxy vs. a target edge TPU), we use an efficient adaptation technique which, by measuring latencies of a small number of architectures on the target device, significantly boosts the latency monotonicity between the adapted proxy device and the target device.

We validate our approach by considering various search spaces and running experiments with devices of different platforms, including mobile, desktop GPU, desktop CPU, edge devices and FPGA. Our results show that, using just one proxy device, there is almost no Pareto optimality loss compared to architectures specifically optimized for each target device. In addition, we also consider the recent latency datasets [40, 64, 129], and confirm further that one proxy device is enough for hardware-aware NAS.

3.2 State of the Art and Limitations of Hardware-Aware NAS

In this section, we provide an overview of the existing (hardware-aware) NAS algorithms as well as SOTA approaches to reducing the performance evaluation cost, and highlight their limitations.

3.2.1 Overview

Neural architecture is a key design hyperparameter that affects the inference accuracy and latency of DNN models. In Fig. 3.2, we show an example architecture, which is found by searching over the possible layer-wise kernel sizes, expansion ratio, and block depth in the MobileNet-V2 search space using evolutionary search [23].

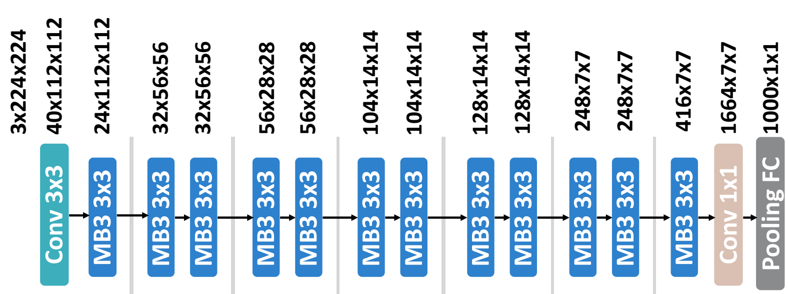


Figure 3.2: An example architecture in the MobileNet-V2 search space, which achieves 70.2% accuracy on ImageNet and 71ms average inference latency on S5e. The text "Z_1 x Z_2 x Z_3" the input size for each layer.

The available architecture space is exponentially large, often consisting of billions of or even more choices (e.g., $>10^{19}$ in [23]). To address the complexity challenge, NAS has recently been proposed to efficiently automate the discovery of neural architectures that exceed the performance of expert-designed architectures [133]. Next, we provide a summary of existing NAS algorithms.

NAS Without a Supernet

Many prior NAS algorithms can be broadly viewed as "NAS *without* a supernet", where the search process is entangled with the model training process [96, 107, 133]. Specifically, as illustrated in the left subfigure of Fig. 3.3, the NAS process is governed by a controller (e.g., a reinforcement learning agent): given each candidate architecture produced by the controller, the model is trained on the training dataset and then evaluated for its performance, based on which the controller produces another candidate architecture. This process repeats until convergence or the maximum search iteration is reached. Techniques to reduce the search cost include training on part of the training dataset, a small proxy dataset, using Bayesian optimization or reinforcement learning to reduce the number of sampled candidate architectures, parameterizing the architectures and using gradients of the loss to guide the search and training simultaneously, among others [15, 70, 101, 105, 107, 133]. Nonetheless, the search cost for even a single device can still take up to 100+ GPU hours, lacking scalability in the presence of numerous heterogeneous devices [24, 120].

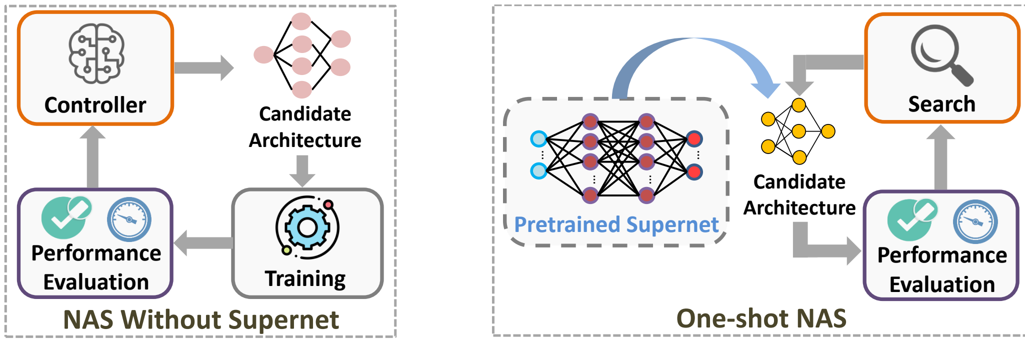


Figure 3.3: Overview of NAS algorithms. Left: NAS without a supernet. Right: One-shot NAS with a supernet.

One-shot NAS

In view of the extremely diverse devices and platforms for model deployment, one-shot NAS and its variants such as few-shot NAS have recently been proposed to reduce the search cost by exploiting the weight sharing mechanism [12, 13, 23, 33, 49, 97, 126, 130]. Concretely, as illustrated in the right subfigure of Fig. 3.3, the key idea of one-shot NAS is to decouple the training process from the search process: pre-train a super large model (called *supernet*) whose weight is shared among all the candidate architectures, and then use a separate search process to discover optimal architectures that inherit the weights from the supernet. For example, in SOTA algorithms such as APQ, ChamNet, BigNAS and FBNet-V3 [23, 33, 34, 115, 126], a supernet is pre-trained first, which is then followed by a search process based on evolutionary algorithms or reinforcement learning to find an optimal architecture.

While pre-training the supernet is more costly than training an individual network, the training cost is one-time[2] for each learning task and, when amortized over hundreds of

[2]With the optimal architecture found by NAS, additional model updates (e.g., by training over the entire dataset or fine-tuning the weights) may still be needed to further improve the accuracy, but this will typically not affect the accuracy rankings of different architectures [15] and is orthogonal to NAS whose goal is to decide an optimal neural architecture.

target devices, will be much more affordable. For example, with the recent once-for-all algorithm [23], the amortized training cost for each target device is around 12 hours given a modest size of 100 devices, and further less given more devices.

3.2.2 Current Practice for Reducing the Cost of Performance Evaluation

With a $\mathcal{O}(1)$ model training cost incurred by one-shot NAS, the cost of performance evaluation — accuracy and latency evaluation — increasingly becomes a bottleneck.

Accuracy evaluation. For each candidate architecture, the time needed to evaluate the inference accuracy (even on a small proxy/validation dataset) is in the order of minutes. Thus, to expedite the accuracy evaluation, SOTA NAS algorithms have leveraged an accuracy predictor: first measuring the accuracies of sample architectures (extracted from the supernet) and then building a machine learning model [33, 34, 115]. Therefore, the candidate architectures can be ranked based on their predicted accuracies, speeding up the runtime process for NAS. Since the inference accuracy is evaluated based on the testing dataset, the accuracy predictor is device-independent and can be re-used for different target devices, incurring a fixed one-time cost of $\mathcal{O}(1)$.

Latency evaluation. Similarly, latency evaluation of candidate architectures is also very costly. Concretely, measuring the actual latency for each candidate architecture takes about 20 seconds or more (to average out the random variations as per TensorFlow-Lite guideline [47] and also suggested by [24]). Meanwhile, the total number of candidate architectures sampled by a NAS algorithm is typically in the order of 10k or even more [28, 34, 107], thus settling the total latency evaluation time to be 50+ hours for just *one* target device.

Table 3.1: Cost Comparison of Hardware-aware NAS Algorithms for n Target Devices.

Algorithm	**Search Method**	**Model Training**	**Accuracy Evaluation**	**Latency Evaluation**	**Total Cost** (Machine-hours)
MNasNet [107]	RL	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$6912n$
FBNet [120]	Gradient	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$216n$
ProxylessNAS [24]	Gradient	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$(200 + c_L)n$
NetAdapt [125]	Loop	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$c_T + (c_A + c_L)n$
APQ [115]	Evolutionary	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$2400 + c_A + c_L n$
ChamNet [34]	Evolutionary	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$c_T + c_A + c_L n$
Once-for-All [23]	Evolutionary	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$1200 + c_A + c_L n$

Using the FLOPs as a device-agnostic proxy cannot accurately reflect the true latency rankings of different architectures on a target device [107]. Instead, to reduce the latency evaluation cost, SOTA hardware-aware NAS algorithms have most commonly used latency predictors — profiling/measuring the latencies for sample architectures in advance and then building a latency predictor (either a lookup table or machine learning model) [23, 24, 34, 129]. Then, the latency predictor is utilized to guide the NAS process, without measuring the actual latency on the target device.

3.2.3 Limitations

Despite the recent progress, SOTA hardware-aware NAS algorithms still cannot scale up in view of the extremely diverse target devices for model deployment.

Summary of total search cost. Given n target devices, we summarize in Table 3.1 the total search costs, measured in machine-hours, of a few representative hardware-aware NAS algorithms. If the quantitative evaluation cost is not reported for an algorithm, we use c_T, c_A, c_L to denote its model training cost, accuracy evaluation cost, and latency evaluation cost, respectively. Empirically, for each device, c_L is in the order of at least a few tens of hours [24, 40] or even hundreds of hours [64, 129]. Thus, we can see that the latency evaluation cost is a significant or even dominant part of the total search cost, especially when n increases.

While the actual execution time of NAS may be further reduced by parallel processing, the total cost in terms of machine-hours does not decrease. For example, latency measurements on multiple devices in parallel and assigning more GPUs for supernet training can both speed up the overall NAS process, but the total resources needed by NAS still remain unchanged (or possibly even higher due to communications overheads among GPUs for distributed training). For this reason, machine-hour is a more accurate and widely-used metric for total resource expenditure in NAS [23, 107, 115, 120].

Challenges. In the current practice, building a latency predictor for each target device requires significant engineering efforts and can be very slow, while it is often excluded from the total cost calculation [34, 68, 112, 115, 120, 125]. Moreover, the diverse set of target devices have different latency collection pipelines, programming environment, and/or hardware domain knowledge requirement, all of which add to the significant challenges of building a latency predictor [64].

The challenges of building latency predictors have been increasingly recognized and motivated some latest studies on latency predictors to facilitate hardware-aware NAS research. For example, [64] releases latency datasets/predictors for six devices on the NAS-Bench-201 space and FBNet space. Even by measuring latencies in parallel, the authors of [64] report on OpenReview that they spent *one month* to collect latency measurement. Another recent study [129] builds a kernel-level latency predictor, taking up 1–4.4 days for latency measurement on each device depending on how powerful the device is. Nonetheless, these approaches are not scalable, and the latency predictors built by these studies are all specific to their limited set of devices.

We can conclude that, in the presence of extremely diverse target devices, the combined cost of building latency predictors for hardware-aware NAS is prohibitively high at $\mathcal{O}(n)$. This has increasingly become a bottleneck for scalability.

3.3 Problem Formulation, Insights, and Practical Consideration

We present the problem formulation for hardware-aware NAS, show the key insights for when we can reduce the latency evaluation cost to $\mathcal{O}(1)$, and finally discuss practical considerations.

3.3.1 Problem Formulation

The general problem of hardware-aware NAS can be formulated as follows:

$$\max_{\mathbf{x}\in\mathcal{X}} \max_{\omega_{\mathbf{x}}} \ accuracy(\mathbf{x}, \omega_{\mathbf{x}}) \tag{3.1}$$

$$s.t., \ \ latency(\mathbf{x}; \mathbf{d}) \leq \overline{L}_{\mathbf{d}} \tag{3.2}$$

where $\mathbf{x}$ represents the architecture, $\mathcal{X}$ is the search space under consideration, $\omega_{\mathbf{x}}$ is the network weight given architecture $\mathbf{x}$, $\overline{L}_{\mathbf{d}}$ is the average inference latency constraint, and $\mathbf{d} \in \mathcal{D}$ denotes a device with $\mathcal{D}$ being the device set. Note that $accuracy(\mathbf{x}, \omega_{\mathbf{x}})$ is measured on a dataset independent of the device $\mathbf{d}$, and can also be replaced with a certain loss function (e.g., cross entropy). By varying $\overline{L}_{\mathbf{d}}$ between its feasible range $[\overline{L}_{\mathbf{d},\min}, \overline{L}_{\mathbf{d},\max}]$, we can obtain a set of *Pareto*-optimal architectures, denoted by $\mathcal{P}_{\mathbf{d}} = \{\mathbf{x}^*(\overline{L}_{\mathbf{d}}; \mathbf{d}),\ \text{for}\ \overline{L}_{\mathbf{d}} \in [\overline{L}_{\mathbf{d},\min}, \overline{L}_{\mathbf{d},\max}]\}$.

Remark. We offer the following remarks on the problem formulation. First, due to the non-convexity and combinatorial nature, the obtained architectures by using approximate methods (e.g., evolutionary search [115]) to solve Eqns. (4.1)(3.2) may not be globally Pareto-optimal in a strict sense; instead, the notation of Pareto-optimality (or simply, optimality) in the context of NAS usually means a satisfactory architecture that outperforms or is very close to SOTA results [2, 33, 107]. Second, as recently shown in [64], the inference latency and energy of an architecture on a device are very strongly correlated. That is, an energy constraint can be implicitly mapped to a corresponding latency constraint. Thus, like in [13, 30, 107, 120, 125], we only consider the inference latency constraint in our formulation for the convenience of presentation.

3.3.2 Key Insights

By observing the NAS problem in Eqns. (4.1)(3.2), achieving $\mathcal{O}(1)$ latency evaluation cost may seem very unlikely. The reason is that the inference latency $latency(\mathbf{x}; \mathbf{d})$ is highly device-specific — with a new device, the latency function will change in general, and so will the Pareto-optimal architectures accordingly. We notice, however, that the Pareto-optimal architectures for two different devices can actually be identical if their latency functions are *monotonic*, as formally defined and proved below.

Definition 1 (Latency Monotonicity) *Given two different devices* $\mathbf{d}_1 \in \mathcal{D}$ *and* $\mathbf{d}_2 \in \mathcal{D}$, *if latency*$(\mathbf{x}_1;\mathbf{d}_1) \geq$ *latency*$(\mathbf{x}_2;\mathbf{d}_1)$ *and latency*$(\mathbf{x}_1;\mathbf{d}_2) \geq$ *latency*$(\mathbf{x}_2;\mathbf{d}_2)$ *hold simultaneously for any two neural architectures* $\mathbf{x}_1 \in \mathcal{X}$ *and* $\mathbf{x}_2 \in \mathcal{X}$*, then the two devices* $\mathbf{d}_1$ *and* $\mathbf{d}_2$ *are said to satisfy latency monotonicity.* ■

Proposition 1 *If two devices* $\mathbf{d}_1 \in \mathcal{D}$ *and* $\mathbf{d}_2 \in \mathcal{D}$ *strictly satisfy latency monotonicity, then they have the same set of Pareto-optimal architectures, i.e.,* $\mathcal{P}_{\mathbf{d}_1} = \mathcal{P}_{\mathbf{d}_2}$*, where* $\mathcal{P}_{\mathbf{d}_i} = \{\mathbf{x}^*(\overline{L}_{\mathbf{d}_i};\mathbf{d}_i),\ \textit{for}\ \overline{L}_{\mathbf{d}_i} \in [\overline{L}_{\mathbf{d}_i,\min}, \overline{L}_{\mathbf{d}_i,\max}]\}$ *for* $i = 1, 2$*.*

Proof. Define $\mathcal{X}_{\overline{L}_{\mathbf{d}_1},\mathbf{d}_1}$ as the set of architectures satisfying *latency*$(\mathbf{x};\mathbf{d}_1) \leq \overline{L}_{\mathbf{d}_1}$. By latency monotonicity, we can find another constraint $\overline{L}_{\mathbf{d}_2}$ such that $\mathcal{X}_{\overline{L}_{\mathbf{d}_1},\mathbf{d}_1} = \mathcal{X}_{\overline{L}_{\mathbf{d}_2},\mathbf{d}_2}$. In other words, the latency constraint *latency*$(\mathbf{x};\mathbf{d}_1) \leq \overline{L}_{\mathbf{d}_1}$ is equivalent to *latency*$(\mathbf{x};\mathbf{d}_2) \leq \overline{L}_{\mathbf{d}_2}$. Therefore, device-aware NAS formulated in Eqns. (4.1)(3.2) for devices $\mathbf{d}_1$ and $\mathbf{d}_2$ are equivalent, sharing the same set of Pareto-optimal architectures. ■

Proposition 1 guarantees that, for any two devices satisfying latency monotonicity, we only need to run device-aware NAS on *one* device, avoiding the cost of numerous latency measurements and building a separate latency predictor for each device. The key reason is that in NAS, it is the architecture's accuracy and latency performance ranking that really matters for Pareto-optimality. Consequently, if latency monotonicity is satisfied among all the target devices, the latency evaluation cost can be kept as $\mathcal{O}(1)$.

3.3.3 Practical Consideration

To quantify the degree of latency monotonicity in practice, we use the metric of Spearman's Rank Correlation Coefficient (**SRCC**), which lies between -1 and 1 and assesses statistical dependence between the rankings of two variables using a monotonic function.

The greater the SRCC of CNN latencies on two devices, the better the latency monotonicity. SRCC of 0.9 to 1.0 is usually viewed as *strongly* dependent in terms of monotonicity [5].

While Proposition 1 does not strictly hold when the SRCC is less than 1.0, we note that a sufficiently high SRCC (e.g., around 0.9 in our experiments) is already good enough in practice. This is due in great part to imperfection/approximation in other aspects of the NAS process. Concretely, in SOTA hardware-aware NAS algorithms [34, 107, 115], the accuracy predictor (or the accuracy measured on a small proxy dataset) only has a SRCC value of around 0.9 with the true accuracy. Thus, given the imperfection of accuracy evaluation, strictly satisfying the latency monotonicity does not offer substantial benefits.

3.4 Latency Monotonicity in the Real World

We now investigate latency monotonicity in the real world and show that it commonly exists among devices, especially of the same platform.

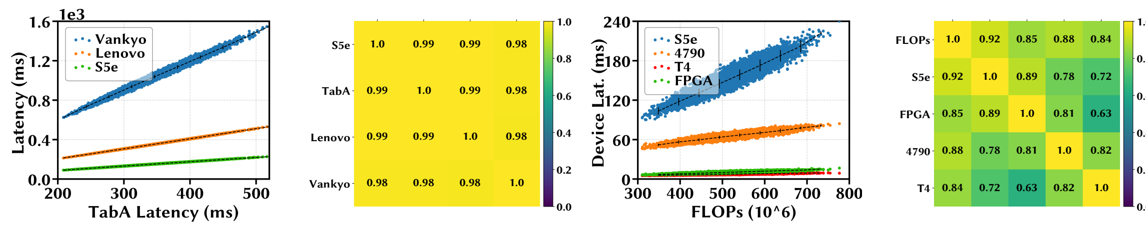
3.4.1 Intra-Platform Latency Monotonicity

We empirically show the existence of strong latency monotonicity among devices of the same platform, including mobile, FPGA, desktop GPU and CPU.

Mobile platform. We first empirically measure the actual latencies of CNN models on four mobile devices: Samsung Galaxy **S5e**, **TabA**, **Lenovo** Moto Tab, and **Vankyo** MatrixPad Z1 (a low-end device). The details of deveice specifications are listed in Table 3.2. We randomly sample 10k models from the MobileNet-V2 space [98] (details in Section 3.6). Then, we deploy these models on the four devices and calculate their average

Device	Abbrev.	Chipset	CPU (GHz)	Cores	RAM (GB)	RAM Freq. (MHz)	Peak Perf. (GFLOPs/sec)	Mem. Bandwidth (GB/sec)
Samsung Galaxy Tab S5e	**S5e**	Snapdragon 670	2	8	4	1866	40.6	14.93
Samsung Galaxy Tab A	**TabA**	Snapdragon 429	2	4	2	933	15.3	7.46
Lenovo Moto Tab	**Lenovo**	Snapdragon 625	2	8	2	933	26.5	7.5
Vankyo MatrixPad Z1	**Vankyo**	N/A	1.5	4	1	933	N/A	N/A

Table 3.2: Device specifications. Full details are not available for Vankyo.



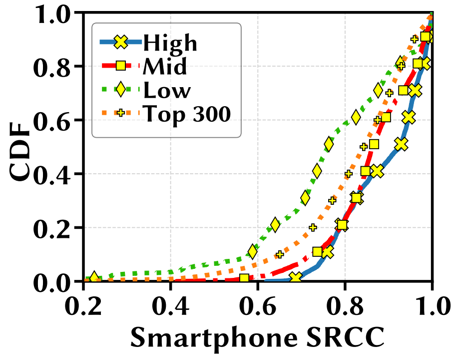
(a) (b) (c) (d)

Figure 3.4: Empirical measurement of latency monotonicity. (a)(c) Black vertical lines denote the standard deviation of latency data points within each bin, with the center denoting the average. (b)(d) SRCC of 10k sampled model latencies on different pairs of devices.

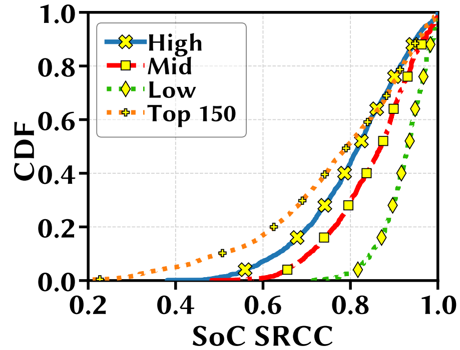
inference latencies. We show the actual latencies on S5e, Lenovo, Vankyo versus TabA in Fig. 3.4(a), where each dot represents one CNN model.

We see that when the sampled CNN models run faster on TabA, they also become faster on the other devices. In Fig. 3.4(a), the maximum standard deviation (denoted by the vertical line within each bin) is 1.3% for Vankyo, while it is negligibly 0.6% and 0.84% for Lenovo and S5e. Thus, latency monotonicity is well preserved on these devices. We further show the SRCC values of these 10k sampled model latencies on our four mobile device in Fig. 3.4(b) with heatmap. We see that SRCC between any pair of our mobile devices is larger than 0.98, implying strong latency monotonicity.

AI-Benchmark data. To examine latency monotonicity at scale, we resort to the AI-Benchmark dataset showing DNN inference latency measurements on diverse hardware [4]. Considering top-300 smartphones (ranging from Huawei Mate 40 Pro to Sony Xperia Z3)



(a) Top-300 Mobile Phones



(b) Top-150 Mobile Phones

Figure 3.5: CDF of SRCC values of DNN models on mobile phones and SoCs. The annotation “high/mid/low” represents the highest/middle/lowest 33.3% of the devices.

and top-150 mobile SoCs (ranging from HiSilicon Kirin 9000 to MediaTek Helio P10) ranked by the metric “AI-Score” [55], we show in Fig. 3.5 the SRCC values of latency rankings based on the 22 DNN models including both floating-point and quantized models (e.g., MobileNet-V2-INT8 and MobileNet-V2-FP16) listed in the dataset. We see that latency monotonicity is well preserved at scale. For example, among the top-100 mobile phones, SRCC values among 50+% of *any* device pairs are higher than 0.9 (a very strong ranking correlation). While the AI-Benchmark dataset is built for orthogonal purposes and includes models from different search spaces, the resulting SRCC values, along with our own experiments, still provide a good reference and show reasonable latency monotonicity for mobile devices at scale.

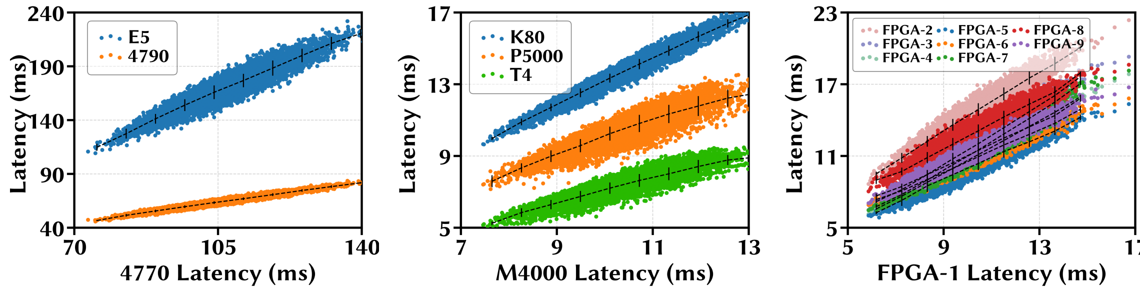
Other platforms. Going beyond the mobile platform, we also perform experiments to show latency monotonicity on other platforms: desktop CPU, desktop GPU, and FPGA.

Index	Computation Design			Communication Design		
	T_m	T_n	$T_m(d)$	I_p	O_p	W_p
1	160	12	576	11	8	13
2	160	12	576	5	5	22
3	160	12	576	12	13	17
4	160	12	576	10	10	10
5	130	12	832	10	10	10
6	100	16	832	10	10	10
7	220	8	704	10	10	10
8	100	16	832	6	14	10
9	100	18	704	10	10	10

Table 3.3: Nine FPGA specifications on Xilinx ZCU 102 board.

We build latency lookup tables for three desktop CPUs: Intel Core i7-**4790**, Intel Core i7-**4770** HQ, and **E5**-2673 v3. In addition, we consider four NVIDIA GPUs: Tesla **T4**, Tesla **K80**, Quadro **M4000**, and Quadro **P5000**. For the FPGA platform, we configure nine subsystems for an Xilinx ZCU 102 FPGA board to create nine different FPGAs following the hardware design space in [58]. The detailed configuration for FPGAs is shown in Table 3.3. "Computation Design" is the computation subsystem design, T_m, T_n are loop tiling parameters for input and output feature maps, and $T_m(d)$ denotes the parameter for depth-wise separable convolution. "Communication Design" represents the communication subsystem design, where I_p, O_p, and W_p are communication ports allocated for input feature maps, output feature maps and weights, respectively. We measure CNN model latency on nine Xilinx ZCU 102 boards shown in Table 3.3, using the performance model in [58].

We consider latencies for the same set of 10k models as in Fig. 3.4, and plot the results in Figs. 3.6 and 3.7, respectively. We see that *within* each platform, latency monotonicity is generally very well preserved, with most SRCC values close to or above 0.9+. In addition, we also show in Fig. 3.7 the SRCC between the model FLOPs and the actual inference latency, confirming the prior observation that FLOP may not accurately



(a) Desktop CPU (b) Desktop GPU (c) FPGA

Figure 3.6: Latency monotonicity on non-mobile platforms. Black vertical lines denote the standard deviation of latency data points within each bin, with the center denoting the average.

(a) Desktop CPU (b) Desktop GPU (c) FPGA

Figure 3.7: SRCC of 10k sampled model latencies on different pairs of non-mobile devices. Specification of nine FPGAs in Fig. 3.7(c) is listed in Table 3.3.

reflect the true latency performance [64, 120, 129].

Next, to complement our own measurement, we also examine latency monotonicity by leveraging third-party latency predictors and measurements results on other devices. The results are available in Appendix 3.10.1 and further corroborate our finding.

3.4.2 Inter-Platform Latency Monotonicity

We choose one FPGA (Xilinx ZCU 102), one desktop CPU (Intel Core i7-**4790**), and one desktop GPU (Tesla **T4**) as cross-platform devices. We show the latency monotonicity results and SRCC values for the same set of 10k models in Figs. 3.4(c) and 3.4(d), respectively. It can be seen that latency rankings are only moderately correlated for cross-platform devices. The SRCC values are lower than in the case of mobile device pairs (Fig. 3.4(b)), since mobile devices often differ significantly from desktops/FPGAs.

Our finding is also confirmed in the appendix by considering the six cross-platform devices on the NAS-Bench-201 [38] and FBNet [120], and four devices on MobileNet-V3 using nn-Meter [129].

3.4.3 Roofline Analysis

We now explain the empirically observed latency monotonicity based on roofline analysis, which is a methodology for visual representation of hardware platform's *peak* performance as a function of the operational intensity, which identifies the bottleneck of the system [118]

Fig. 3.8(a) shows the theoretical roofline model of two mobile devices (Samsung Galaxy S5e and TabA) plotted according to their reported hardware specification listed in Table. 3.2. When operational intensity is low (linear slope region in Fig. 3.8(a)), memory bandwidth is the limiting factor for program speed (i.e., memory-bound); when operational intensity is high (horizontal region), peak FLOPs rate becomes the bottleneck (i.e., compute-bound).

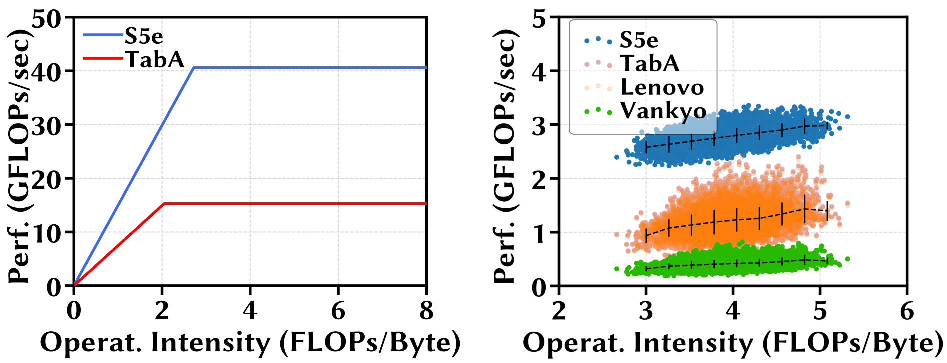
Suppose that we have two devices $\mathbf{d}_1$ and $\mathbf{d}_2$ with memory bandwidths $B_{\mathbf{d}_1}$ and $B_{\mathbf{d}_1}$, respectively, and two CNN models of architectures $\mathbf{x}_1$ and $\mathbf{x}_2$ with operational intensities $OI_{\mathbf{x}_1}$ and $OI_{\mathbf{x}_2}$, respectively. Next, we show that latency monotonicity is guaranteed to hold for two devices if CNN models are either memory-bound or compute-bound on both devices.

Memory-bound. In the memory-bound region, the slope in the roofline model of a device is the bandwidth, and the resulting performance is the bandwidth multiplied by the program's operational intensity. Assuming that $\mathbf{x}_1$ is slower than $\mathbf{x}_2$ on device $\mathbf{d}_1$ without loss of generality, we have $\frac{FLOP_{\mathbf{x}_1}}{OI_{\mathbf{x}_1} \cdot B_{\mathbf{d}_1}} > \frac{FLOP_{\mathbf{x}_2}}{OI_{\mathbf{x}_2} \cdot B_{\mathbf{d}_1}}$. Then, by multiplying both sides by $\frac{B_{\mathbf{d}_1}}{B_{\mathbf{d}_2}}$, we obtain $\frac{FLOP_{\mathbf{x}_1}}{OI_{\mathbf{x}_1} \cdot B_{\mathbf{d}_2}} > \frac{FLOP_{\mathbf{x}_2}}{OI_{\mathbf{x}_2} \cdot B_{\mathbf{d}_2}}$, i.e., $\mathbf{x}_1$ is also slower than $\mathbf{x}_2$ on device $\mathbf{d}_2$. Thus, latency monotonicity holds for the two devices $\mathbf{d}_1$ and $\mathbf{d}_2$.

Compute-bound. Likewise, if CNN models fall into the compute-bound region for two devices, then we can also establish latency monotonicity using a similar logic.

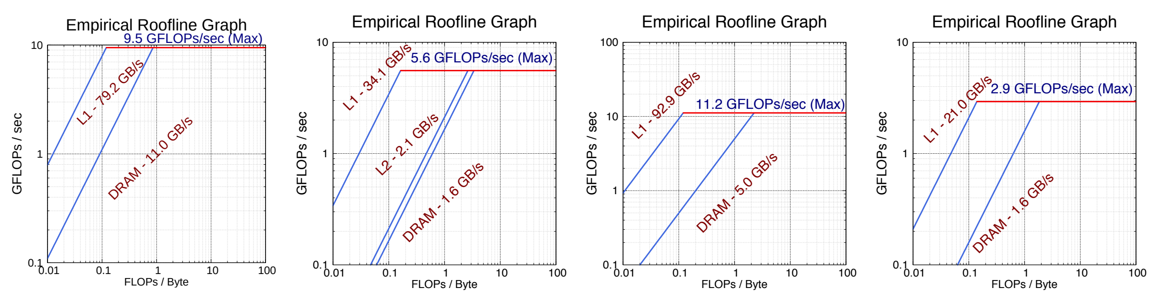
For search spaces with models that span across both memory-bound and compute-bound regions, the latency monotonicity may not be strong (which we shall address in this work). Moreover, the roofline analysis only provides a *sufficient* condition for latency monotonicity under the assumption that devices run at their peak performances (in terms of FLOPs/sec). Thus, we experimentally show the actual performance of CNN models on our four mobile devices shown in Table 3.2.

We measure the actual attainable peak performance of our four devices with the tool in [53], a roofline model specially for mobile SoCs. Our results show that the sampled CNN models (with 2.6 to 5.4 FLOPs/Byte) are all in the *compute*-bound region for the de-



(a) Theoretical roofline model (b) Sampled models

Figure 3.8: (a) Theoretical roofline model is plotted according to hardware specification of S5e and TabA. (b) Black vertical lines denote the standard deviation of data within each bin, with the center denoting the average.



(a) S5e (b) TabA (c) Lenovo (d) Vankyo

Figure 3.9: Empirical roofline models of devices in Table 3.2 measured with Gables [53].

vices. We randomly sample 10000 models from the MobileNet-V2 [98] (detailed experiment setup is presented in Section 3.6). The empirical roofline results are shown in Fig. 3.8(b). It can be seen that the operational intensity of the sampled models ranges from 2.6 to 5.4 FLOPs/Byte, while the devices' actual performances as shown in Fig. 3.9 are much lower than their peaks and vary for different models. Specifically, the ridge operational intensity of S5e, Lenovo, and Vankyo are less than or around 2 FLOPs/Byte, while TabA has a threshold of 3 FLOPs/Byte. Thus, most of our sampled models reside in the compute-bound region of these devices, except for those with operational intensity less than 3 FLOPs/Byte on

TabA. This partially explains the strong latency monotonicity that we empirically observe in Fig. 3.4(b).

3.5 Hardware-Aware NAS With One Proxy Device

Section 3.4 demonstrates good latency monotonicity among devices of the same platform, but this is not always the case, especially for devices across different platforms. To address the cases of low monotonicity, we propose efficient transfer learning based on the proxy device.

3.5.1 Necessity of Strong Latency Monotonicity

We first highlight the necessity of strong latency monotonicity for finding optimal architectures on the target device. An interesting and challenging case is when latency monotonicity is not satisfied, and this is not uncommon in practice as shown in Section 3.4. In such cases, the optimal architectures searched on one device can be far from optimality on another device. To see this point, we show in Fig. 3.11(a) the performance of architectures found on different devices using the MobileNet-V2 search space. All latencies are measured on S5e (Mobile), and the architectures directly found for S5e are Pareto-optimal ones. Nonetheless, when performing NAS on two other (proxy) devices — 4790 (Desktop CPU) and T4 (Desktop GPU) — which both have low SRCC values with S5e, the searched architectures are highly sub-optimal. Thus, given weak latency monotonicity, the Pareto optimality of $\mathcal{P}_{\mathbf{d}_0}$ on the proxy device $\mathbf{d}_0$ does not hold on the target device $\mathbf{d}$, calling for

Algorithm 5 Hardware-Aware NAS With One Proxy Device

Inputs: Target device $\mathbf{d}$, proxy device $\mathbf{d}_0$ with its latency predictor $L_{\mathbf{d}_0}(\mathbf{x})$ and Pareto-optimal architecture set $\mathcal{P}_{\mathbf{d}_0}$, small sample architecture set $\mathcal{A}$, SRCC threshold S_{th}
Output: Pareto-optimal architecture $\mathcal{P}_{\mathbf{d}}$
Measure $latency(\mathbf{x}; \mathbf{d})$ for $\mathbf{x} \in \mathcal{A}$;
Estimate SRCC $S_{\mathbf{d},d}$ for sample architectures in $\mathcal{A}$;
if $S_{\mathbf{d},\mathbf{d}_0} \geq S_{th}$ **then**
  Set $\mathcal{P}_{\mathbf{d}} = \mathcal{P}_{\mathbf{d}_0}$, or re-run NAS (e.g., evolutionary search) based on $L_{\mathbf{d}_0}(\mathbf{x})$ to obtain $\mathcal{P}_{\mathbf{d}}$;
else
  Use Eqn. (3.3) to obtain $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$ based on measured $latency(\mathbf{x}; \mathbf{d})$ for $\mathbf{x} \in \mathcal{A}$;
  Run NAS based on $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$ to obtain $\mathcal{P}_{\mathbf{d}}$;
end if
Measure latencies for architectures $\mathbf{x} \in \mathcal{P}_{\mathbf{d}}$ on device $\mathbf{d}$, and remove non-Pareto-optimal ones from $\mathcal{P}_{\mathbf{d}}$;

remedies to boost the latency monotonicity.

3.5.2 Overview

Our scalable hardware-aware NAS approach is illustrated in Fig. 3.10 and described in Algorithm 6.

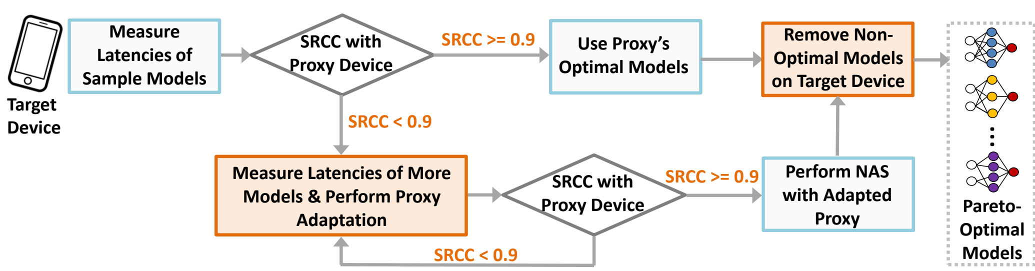


Figure 3.10: Overview of using one proxy device for hardware-aware NAS.

Prerequisite. The prerequisite step is to select a proxy device $\mathbf{d}_0$ and run SOTA hardware-aware NAS to find a set $\mathcal{P}_{\mathbf{d}_0}$ of Pareto-optimal architectures for the proxy.

Checking latency monotonicity. Given a new target device, we check whether strong latency monotonicity is satisfied between the proxy device and the target device, by estimating the SRCC based on a small set of sample architectures $\mathcal{A}$ and comparing it against a threshold.

- *When strong latency monotonicity holds.* With strong latency monotonicity, the target device's Pareto-optimal architecture set $\mathcal{P}_{\mathbf{d}}$ is also likely the same as proxy device's $\mathcal{P}_{\mathbf{d}_0}$. Alternatively, we can also re-run evolutionary search based on the proxy device's latency predictor to obtain more architectures, which are in turn also likely optimal ones for the target device.

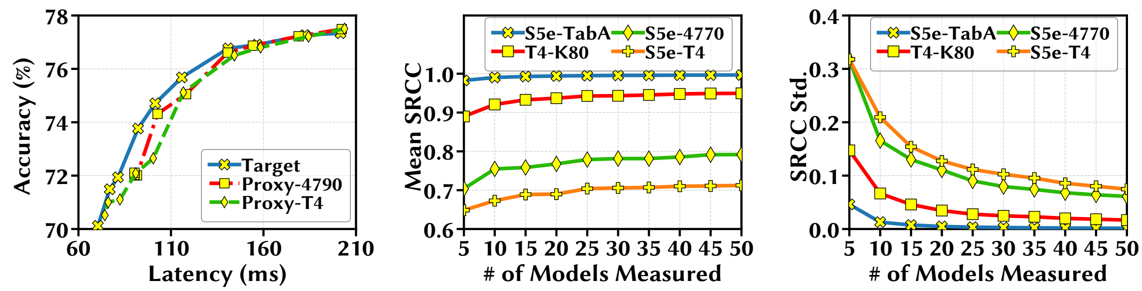
- *When strong latency monotonicity does not hold.* We propose an efficient transfer learning technique — adapting the proxy's latency predictor to the target device. By doing so, we can quickly find optimal architectures for the target device, yet *without* first measuring latencies of thousands of architectures and then building a latency predictor.

Removing non-Pareto-optimal architectures. We measure the actual latencies of Pareto-optimal architectures (obtained for either the paroxy or adapted proxy device) on the target device, and remove non-Pareto-optimal architectures.

3.5.3 Prerequisite and Checking Latency Monotonicity

Prerequisite

We first select a proxy device $\mathbf{d}_0$ that preferably has good latency monotonicity with other target devices. To do so, we can first measure the latencies of a small set $\mathcal{A}$ of sample architectures (e.g., 30-50 sample architectures in our experiments) on all the target devices,



(a) NAS results with low SRCC values (b) Average Estimated SRCC (c) Standard Deviation

Figure 3.11: (a) Architectures by evolutionary search in the MobileNet-V2 search space. All latencies are measured on S5e (Mobile). Architectures searched on 4790 (Desktop CPU) and T4 (Desktop GPU) are highly sub-optimal compared to those searched specifically on S5e. These two devices have SRCC of 0.78 and 0.72 with S5e, respectively. (b)(c) SRCC estimation. X-axis denotes the number of sample architectures we randomly select per run. We use 1000 runs to calculate the mean and standard deviation. "x-y" means the device pair is (x, y).

and calculate the resulting SRCC values for each pair of devices based on the measured latencies. We only need to measure the overall inference latency, unlike building latency predictors which typically needs profiling the latency of each operator/layer for thousands of architectures [68, 129].

Then, we can obtain a SRCC matrix like the one shown in Fig. 3.4(d). Latency measurement of a small set of sample architectures is also needed to check latency monotonicity (Section 3.5.3) and hence is not an extra step. Next, we can choose a proxy device that has high SRCCs with a good number of other devices. Note that proxy device selection does not need to be very precise; instead, even though we choose a proxy device that does not have high SRCCs with many other devices, our proposed proxy adaptation technique can still significantly boost the SRCC between the selected proxy device and target devices.

For the selected proxy device, we run SOTA hardware-aware NAS to find Pareto-optimal architectures. Specifically, following the one-shot NAS approach [23, 115], we first pre-train a supernet and build an accuracy predictor. We then build a latency predictor denoted by $L_{\mathbf{d}_0}(\mathbf{x})$ based on extensive latency profiling and SOTA methods for latency prediction [40, 129]. Finally, we apply evolutionary search [34, 115], which quickly produces the Pareto-optimal architecture set $\mathcal{P}_{\mathbf{d}_0}$ by varying different latency constraints. Once the accuracy predictor and latency predictor are built, running evolutionary search takes at most a few minutes and hence is negligible.

Checking latency monotonicity

To check whether strong latency monotonicity is satisfied between the selected proxy device and a target device, we estimate the SRCC based on a small set $\mathcal{A}$ of sample architectures and then compare it against a threshold. The latency measurement for the small set of sample architectures is already performed during the proxy selection process. In Figs. 3.11(b) and 3.11(c), we can see that latency measurement based on a few sample architectures is enough to reliably estimate the SRCC value: e.g., if we set 0.9 as the SRCC threshold, then 30-50 sample architectures are sufficient. Thus, the cost for measuring latencies for the small set $\mathcal{A}$ of sample architectures is negligible compared to building a device-specific latency predictor.

3.5.4 Increasing Latency Monotonicity by Adapting the Proxy Latency Predictor

As illustrated in Fig. 3.11(a), in case of weak latency monotonicity, we cannot re-use the same set of Pareto-optimal architectures found for the proxy device to a new target device. To address this issue, we propose an efficient transfer learning-based technique —

adapting the proxy's latency predictor to the target device — to drastically boost latency monotonicity.

A close look at SOTA latency predictors

We first review three major types of SOTA latency predictors used in hardware-aware NAS.

• **Operator-level latency predictor.** A straightforward approach is to first profile each operator [24, 34] (or each layer [20, 105]), and then sum all the operator-level latencies as the end-to-end latency of an architecture. Specifically, given K operators (e.g., each with a searchable kernel size and expansion ratio), we can represent each operator using one-hot encoding: 1 means the respective operator is included in an architecture, and 0 otherwise. Thus, an architecture can be represented as $\mathbf{x} \in \{0,1\}^K \cup \{1\}$, where the additional $\{1\}$ represents the non-searchable part, e.g., fully-connected layers in CNN, of the architecture. Accordingly, the latency predictor can be written as $l = \mathbf{w}^T\mathbf{x}$, where $\mathbf{w} \in \mathbb{R}^{K+1}$ is the operator-level latency vector. This approach needs a few thousands of latency measurement samples (taking up a few tens of hours) [24, 68].

• **GCN-based latency predictor.** To better capture the graph topology of different operators, a recent study [40] uses a graph convolutionary network (GCN) to predict the inference latency for a target device. Concretely, the latency predictor can be written as $l = GCN_{\Theta}(\mathbf{x})$, where Θ is the learnt GCN parameter learnt and $\mathbf{x}$ is the graph-based encoding of an architecture.

• **Kernel-level latency predictor.** Another recent latency predictor is to use a random forest to estimate the latency for each execution unit (called "kernel") that captures different compilers and execution flows, and then sum up all the involved execution units as the latency of the entire architecture [129]. This approach unifies different DNN frameworks, such as TensorFlow and Onnx, into a single model graph, and hence can predict latencies for models developed using different frameworks. By encoding an architecture based on the execution units, we can also transform the latency predictor into a linear one: $l = \mathbf{w}^T\mathbf{x}$ where $\mathbf{w}$ is the vector of latencies for different execution units and $\mathbf{x}$ denotes the number of each execution unit included in an architecture. Thus, an "execution unit" in [129] is conceptually equivalent to a searchable operator in the operator-level latency predictor [24].

Summary. The three SOTA latency predictors use different encodings/representations for an architecture: the encoding based on searchable operators in an operator-level predictor is the simplest, while the encoding based on fine-grained execution units in a kernel-based predictor has the most details of an architecture. Despite different prediction accuracies in terms of mean squared errors, they all reflect the latency rankings on an actual device very well and hence are sufficient for serving as the proxy predictor.

Adapting the proxy latency predictor

We propose efficient transfer learning to boost the otherwise possibly weak latency monotonicity for a target device.

Intuition. Even though two devices have weak latency monotonicity, it does not mean that their latencies for each searchable operator are uncorrelated; instead, for most operators, their latencies can still be roughly proportional. The reason is that a more complex operator with higher FLOPs that is slower (say, 2x slower than a reference operator) on one device is generally also slower on another device, although there may be some differences in the slow-down factor (say, 2x vs. 1.9x). This is also the reason why some NAS algorithms use the device-agnostic metric of architecture FLOPs as a rough approximation of the actual inference latency [107, 108]. If we view proxy adaptation as a new learning task, this task is highly correlated with the task of building the proxy device's latency predictor, and such correlation can greatly facilitate transfer learning.

Approach. To explain our transfer learning approach, we consider the proxy device's latency predictor in a linear form: $L_{\mathbf{d}_0}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$, where $\mathbf{w}$ is the weight and $\mathbf{x}$ is the architecture representation (e.g., one-hot encoding of the searchable operators, penultimate layer output in a neural network-based predictor,[3] or encoding of the execution units). We measure the latencies of a small set of sample architectures $\mathbf{x} \in \mathcal{A}$ on the target device, noting that this step is also needed to check the SRCC value and incurs a negligible overhead compared to SOTA approaches (i.e., tens of hours of latency measurement [64, 129]). Then, with the latency measurement samples denoted by $(\mathbf{x}_i, y_i)$, we quickly adapt the proxy device's latency predictor as $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x}) = \left[(\alpha\mathbf{I}^T + \mathbf{b}^T) \circ \mathbf{w}^T\right]\mathbf{x}$ tailored to the target device, by solving the following the problem:

$$\min_{\alpha,\mathbf{b}} \frac{1}{N} \sum_i \left| \left[(\alpha\mathbf{I}^T + \mathbf{b}^T) \circ \mathbf{w}^T\right]\mathbf{x} - y_i \right|^2 + \lambda|\mathbf{b}|, \tag{3.3}$$

where $\mathbf{I}$ is the identity vector with all the elements being 1, the operator "$\circ$" denotes the element-wise multiplication, and $\lambda \geq 0$ is a hyperparameter controlling the weight for the sparsity regularization term $|\mathbf{b}|$ and tuned based on a small validation set of architectures (20

[3] If the proxy device uses a neural network-based latency predictor, we can also fix the earlier layers while updating the weights in the last few layers, instead of only updating the last single layer.

architectures in our experiment) split from the sample architecture set $\mathcal{A}$. The interpretation of using Eqn. (3.3) is as follows. First, the scaling factor α reflects our intuition that a more complex operator that is slower on one device is generally also slower on another device. Second, the sparsity term $\mathbf{b}$ accounts for the fact that the slow-down factors for an operator on two devices are not necessarily the same.

With $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$, we essentially construct a new virtual proxy device (called adapted proxy or AdaProxy) whose latency is given by $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$. Here, our goal is to increase the latency monotonicity between the new virtual proxy and the target device; we do not need to create a new latency predictor that produces accurate estimates of the absolute latency values for the target device.

If strong latency monotonicity still does not hold between AdaProxy and the target device, we can incrementally measure the latencies of another small set of sample architectures on the target device and re-solve Eqn. (3.3). In the majority of our experiments, 50 latency measurements on the target device are enough to achieve a strong latency monotonicity. This is negligible compared to thousands of latency profiling and measurements used by SOTA algorithms [24, 129].

Next, with the adapted latency predictor $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$ that reflects the architecture latency rankings on the target device $\mathbf{d}$, we can run evolutionary search to find the set of Pareto-optimal architectures.

3.5.5 Remove non-Pareto-optimal architectures

Up to this point, we have obtained for the target device an architecture set $\mathcal{P}_{\mathbf{d}}$, which is the same as the proxy (or AdaProxy) device's Pareto-optimal set. While the latency monotonicity between the proxy (or AdaProxy) device and the target device is strong (e.g., SRCC around 0.9 or higher), it is not perfect. Thus, some architectures in $\mathcal{P}_{\mathbf{d}}$ may not be Pareto-optimal for the target device. We remove these architectures based on their actual latencies measured on the target device. Specifically, if an architecture $\mathbf{x}_1 \in \mathcal{P}_{\mathbf{d}}$ has a higher latency but the same or similar accuracy compared to another architecture $\mathbf{x}_2 \in \mathcal{P}_{\mathbf{d}}$, we can remove $\mathbf{x}_1$ from $\mathcal{P}_{\mathbf{d}}$.

Finally, if there is a specific latency constraint that is not satisfied by architectures in $\mathcal{P}_{\mathbf{d}}$, we can re-run evolutionary searches with the assistance of $L_{\mathbf{d}_0}(\mathbf{x})$, or adapted $L_{\mathbf{d}_0,\mathbf{d}}(\mathbf{x})$ if applicable, to further enlarge the set $\mathcal{P}_{\mathbf{d}}$. The key point is that we do not need to go through a very time-consuming process to build a new latency predictor specifically for the target device.

In summary, the cost for measuring latencies of a small sample set of architectures on the target device for checking latency monotonicity (and, if needed, adapting $L_{\mathbf{d}_0}(\mathbf{x})$) is negligible. Therefore, given n different devices, we achieve a total latency evaluation cost of $\mathcal{O}(1)$, which, when combined with SOTA NAS algorithms that have $\mathcal{O}(1)$ cost for model training and accuracy evaluation [23, 34], successfully keeps the entire NAS cost at $\mathcal{O}(1)$.

3.6 Experiment

We run experiments on multiple devices (including mobile phones, desktop GPU/CPU, ASIC, etc.) on different mainstream search spaces — MobileNet-V2, MobileNet-V3, NAS-Bench-201, and FBNet.

3.6.1 Results on MobileNet-V2

Setup

We now present the setup for our experiments on MobileNet-V2.

Search Space. As in [24], the backbone of our CNN architecture is MobileNet-V2 with multiplier 1.3, with the channel number in each block fixed. The search space consists of depth of each stage, kernel size of convolutional layers, and expansion ratio of each block. The depth can be chosen from "2, 3, 4", kernel size can be "3, 5, 7", and candidate expansion ratios are "3, 4, 6". There are five stages whose configurations can be searched.

NAS Method. We consider one-shot NAS and use the *Once-For-All* network [23] as a supernet that has the same search space as ours. We run evolutionary search to find optimal architectures for the proxy (or AdaProxy) device. Our parameter settings are: population size is 1000, parent ratio is 0.25, mutation probability is 0.1, mutation ratio is 0.25, and we search for 50 generations given each latency constraint. Evolutionary search takes less than 30 seconds for each run. To facilitate the readers' understanding, we provide a summary of evolutionary search in Appendix 3.9, while the full details can be found in [34, 115].

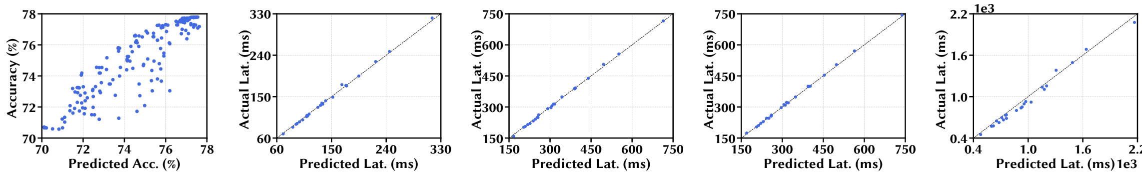
Accuracy Predictor. The evolutionary search is assisted with by an accuracy predictor for fast architecture performance evaluation [34, 115]. Our accuracy predictor is a neural network with four fully-connected layers and updated with 176 samples on top of the predictor used in [23]. The accuracy predictor takes a 128-dimensional feature vector (which is converted from a 21-dimensional architecture configuration within the search space) as input. Fig. 3.12(a) compares the actual and predicted accuracies, which have a SRCC of

0.903 and root mean squared error of 1.11%. The performance of our accuracy predictor is in line with the existing NAS literature for MobileNet-based models [23]. As a result, the imperfection in the accuracy predictor explains why a strong, but not perfect, latency monotonicity (e.g., SRCC>0.9) is enough for our one-proxy approach to find Pareto-optimal architectures for a new target device.

Latency Predictor. We build device-specific latency predictors in the MobileNet-V2 space for our four devices listed in Table 3.2. Specifically, for each sample architecture, we profile the average latency of 1000 runs. We use a single thread for running the TensorFlow Lite interpreter by default. To show the accuracy of our latency predictors, we sample a few additional models and measure their actual latency on our four mobile devices. The comparison between actual and predicted latency is shown in Fig. 3.12, with a root mean squared error of 2.88ms on S5e, 4.69ms on TabA, 3.72ms on Lenovo, and 59.18ms on the low-end Vankyo. As corroborated by prior studies [34, 115, 125], our result shows that the predicted average latency is almost identical to the actual value.

We choose S5e mobile phone as the proxy device. Our results of using other mobile devices as the proxy are nearly the same because S5e and the other mobile devices have SRCC close to 1.0 (Fig. 3.4(b)), i.e., these mobile devices are almost viewed *one* device based on Proposition 1.

Architecture Evaluation. For a searched architecture, the actual model performance is measured. We evaluate accuracies on the ImageNet validation dataset [35], which consists of 50000 images in 1000 classes. Accuracy evaluation is run on Google Colab equipped with Tesla T4.



(a) Accuracy (b) S5e (c) TabA (d) Lenovo (e) Vankyo

Figure 3.12: (a) Actual vs. predicted accuracy. The root mean squared error is 1.11%, and SRCC is 0.903. (b)(c)(d)(e) Measured average inference latency versus predicted latency based on latency lookup tables. The root mean squared errors for S5e, TabA, Lenovo, and Vankyo are 2.88ms, 4.69ms, 3.72ms, and 59.18ms respectively.

Baselines

We consider the following baselines for hardware-aware NAS.

#1: Building a Latency Predictor for Each Target Device [23,34,40,115]. For each device, we use the same evolutionary search described in Section 4.4. While the accuracy predictor is reusable across devices and evolutionary search is quick, measuring latencies of thousands of architectures to build a device-specific latency predictor (as done in the existing hardware-aware NAS [23, 40, 115]) is time-consuming. Thus, this approach has a total cost of $\mathcal{O}(n)$ for n devices [13, 34].

#2: Heuristic Model Scaling. There are different ways to scale a CNN to meet different latency constraints: e.g., adapt the network depth and/or width [107, 108]. Since the number of channels in our backbone network is fixed, we heuristically scale the depth of a Pareto-optimal architecture on the proxy device by increasing (for higher accuracy) or reducing (for smaller latency) the depth by up to two blocks, and transfer the scaled architecture to new target devices. This approach has $\mathcal{O}(1)$ complexity.

The two baselines highlight that the existing hardware-aware NAS either achieves Pareto optimality but has a $\mathcal{O}(n)$ latency evaluation cost (Baseline #1), or keeps the latency evaluation cost at $\mathcal{O}(1)$ but loses Pareto optimality (Baseline #2). By contrast, our approach has a $\mathcal{O}(1)$ latency evaluation cost in total, while preserving Pareto optimality.

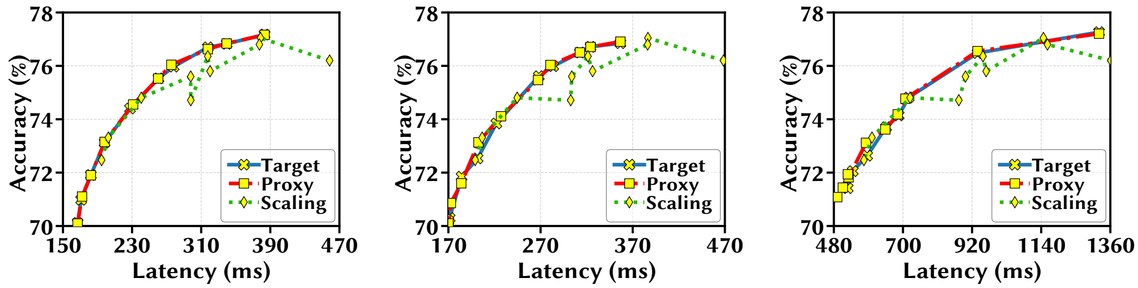
Performance of Searched Architectures

We compare the measured top-1 accuracy on ImageNet versus average inference latency of searched architectures on each target device.

Mobile Devices. Fig. 3.13 shows the result for three different target mobile devices, all using S5e as the proxy device. The SRCC values between S5e and the target devices are all greater than or equal to 0.98 (Fig. 3.4(b)). We see that the architectures searched on S5e can result in almost the same $(accuracy, latency)$ tradeoff as device-specific NAS, but the additional latency evaluation cost for each target device is negligible. Further, we see that despite its $\mathcal{O}(1)$ complexity, heuristic adaptation (baseline #2) can result in really bad architectures without performance guarantees.

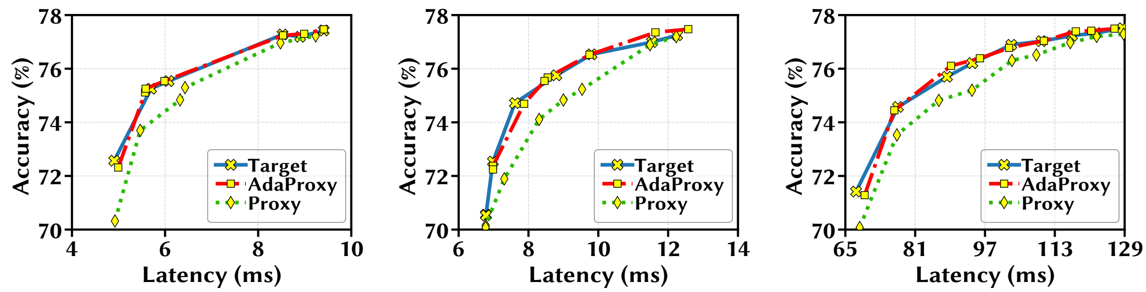
Non-Mobile Devices. We show the results in Fig. 3.14 for non-mobile devices. As these devices have low SRCC values with our S5e proxy, we use Eqn. (3.3) to create an AdaProxy device, which has SRCC of close to 0.9 or higher with the target devices. The details of the proxy adaptation process, including the SRCC values before and after proxy adaptation, are available in Appendix 3.10.2.

The top row shows the architectures found by evolutionary search. We see that with a low SRCC (around 0.7-0.8), the architectures searched on the proxy device are not Pareto-optimal on the target devices. With proxy adaptation, the SRCC increases significantly, and the architectures searched on the AdaProxy device are almost the same as those directly

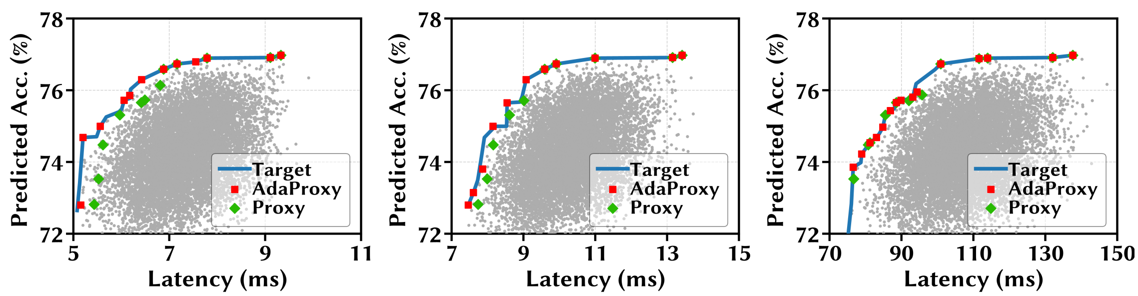


(a) TabA as target device (b) Lenovo as target device (c) Vankyo as target device

Figure 3.13: Results on three different mobile target devices, using S5e as proxy device. "Target" is the baseline #1, "Proxy" means using our approach with S5e as the proxy device, and "Scaling" means heuristic scaling applied to S5e's one Pareto-optimal architecture.



(a) Tesla T4 (b) Quadro M4000 (c) Intel i7-4770HQ



(d) Tesla T4 (e) Quadro M4000 (f) Intel i7-4770HQ

Figure 3.14: Results for non-mobile target devices with the default S5e proxy and AdaProxy. The top row shows the evolutionary search results with real measured accuracies, and the bottom row shows the exhaustive search results based on 10k random architectures and predicted accuracies.

searched on the target device. This highlights the need of strong latency monotonicity between the proxy and the target device, as well as the effectiveness of our proposed proxy adaptation technique to boost the latency monotonicity. The heuristic scaling approach

(Baseline #2) performs even worse than directly using the architectures searched on the proxy device, and hence are omitted.

The bottom row shows exhaustive search results out of 10k randomly selected architectures, using the predicted accuracies as the true values. This is essentially considering a semi-oracle NAS process (on a small space of 10k architectures) assuming a perfect accuracy predictor. As a result, compared to evolutionary search using an imperfect accuracy predictor, it may have a more stringent requirement on the SRCC between the target device and the proxy (or AdaProxy) device. We see that, due to the low SRCC, the architectures found by using the proxy device's latency predictor may not overlap with the oracle's Pareto-optimal boundary. In fact, some of the proxy's optimal architectures can perform very poorly on the target device. For example, Fig. 3.14(d) shows that S5e's optimal architectures are highly sub-optimal on Tesla T4. On the other hand, with improved SRCC, the architectures found by using the AdaProxy device's latency predictor preserves Pareto optimality very well on the target devices, again demonstrating the necessity and effectiveness of our proxy adaptation technique in the presence of weak latency monotonicity between the default proxy and target device.

Additional results, including settings for proxy adaptation and comparison of exhaustively searched architectures on other devices, can be found in Appendix 3.10.2.

3.6.2 Results on NAS-Bench-201, FBNet, and nn-Meter

We now evaluate our approach on the recently released latency datasets for six different devices on NAS-Bench-201 and FBNet spaces [64], additional devices on NAS-

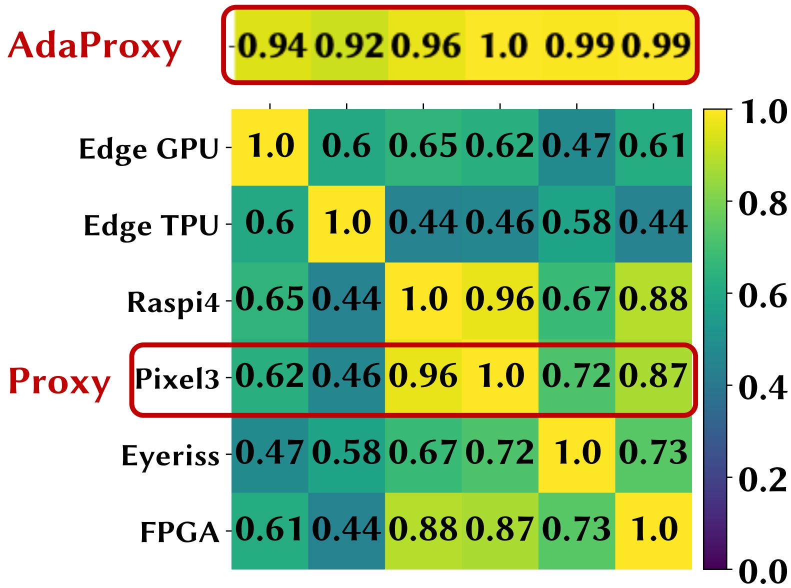
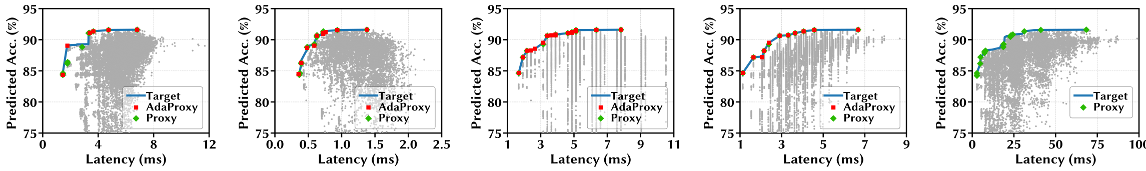


Figure 3.15: SRCC for various devices in the NAS-Bench-201 search space on CIFAR-10. Pixel3 is our proxy device. SRCC values boosted with AdaProxy are highlighted.

Bench-201 [40], as well as four devices on nine different search spaces [128].

We first consider the latency results on the NAS-Bench-201 search space using the CIFAR-10 dataset [64]. Since NAS-Bench-201 represents a simple architecture space with only around 15k architectures, we consider an oracle NAS process via exhaustive search. Thus, compared to evolutionary search using an imperfect accuracy predictor, the oracle NAS process can have a more stringent requirement on the SRCC between the target device and the proxy (or AdaProxy) device. We use Pixel3 as the default proxy which, as shown in Fig. 3.15, does not have strong latency monotonicity with the target devices (except for Raspi4). By proxy adaptation, we can significantly boost the latency monotonicity, increasing the SRCC values to 0.9 or higher.

Next, Fig. 3.16 shows the optimal architectures found by using the proxy device's latency predictor, the adapted latency predictor, and the oracle, respectively. We can see that due to the pre-adaptation low SRCC values between the proxy device Pixel3 and the



(a) Edge GPU (b) Edge TPU (c) Eyeriss (d) FPGA (e) Raspberry Pi 4

Figure 3.16: Exhaustive search results for different target devices on NAS-Bench-201 architectures (CIFAR-10 dataset) [38, 64]. Pixel3 is the proxy.

target devices, only a few architectures that are optimal for the proxy are still optimal for the target devices after architecture removal (Section 3.5.5). Moreover, even the proxy's remaining optimal architectures can be far from optimality on the target device. For example, Fig. 3.16(a) shows that some of Pixel3's optimal architectures deviate from the Pareto-optimal boundary on the edge GPU. By using proxy adaptation and increasing the SRCC values, the AdaProxy's optimal architectures can be efficiently transferred to target devices while preserving optimality. The proxy device Pixel3 has a high SRCC of 0.96 with Raspi4, even without proxy adaptation. Thus, as shown in Fig. 3.16(e), the optimality of Pixel3's architectures preserve very well on Raspi4. All these demonstrate the importance of strong monotonicity between the proxy and the target device, as well as the effectiveness of our proxy adaptation technique. for scalable hardware-aware NAS.

Additional results, including the details of proxy adaptation and results on other search spaces, are available in Appendix 3.10. These results further validate our approach and highlight the practical feasibility of using only one proxy device for scalable hardware-aware NAS.

3.7 Related Work

The huge search space for neural architectures presents significant challenges (see [42,68,79,107,108,120,133] and references therein). To minimize the cost of training numerous architectures, one-shot NAS uses a super net that includes all the weights for candidate architectures [12, 13, 15, 23, 49]. In recent years, transformer-based vision algorithms have also been emerging and inspired studies transformer search to optimize the performance [39], but it is orthogonal to NAS that we focus on.

Importantly, fast evaluation of accuracy and inference latency to rank different architectures is crucial for efficient hardware-aware NAS [42,68,79,107,108,120,133]. To reduce the cost of accuracy evaluation, the prior studies have considered reinforcement learning with accuracy evaluated based on a small proxy dataset [133], Bayesian optimization-based NAS (to reduce the number of sampled and evaluated architectures) [96], generative approaches [60], one-shot or few-shot NAS [12,23,130], and NAS assisted with an accuracy predictor [34, 115]. More recently, ranking architecture accuracies based on easily-computable proxy metrics has also been studied: e.g., computing a model score based on a small minibatch of training data [2], and analyzing the neural tangent kernel (NTK) as well as the number of linear regions in the input space [26].

To expedite inference latency evaluation, the SOTA hardware-aware NAS has mainly resorted to device-specific latency predictors [13, 15, 24, 30, 34, 40, 115, 125]. Nonetheless, building even one latency predictor incurs a non-trivial upfront cost. Thus, [40,64,129] have recently released latency datasets and predictors, but only for a few devices due to the prohibitive time cost.

Given many diverse devices, scalability of latency evaluation is critically important. A straightforward approach is to build a meta latency predictor that incorporates hardware features as additional input [63, 78]. Nonetheless, significant drawbacks exist for this approach: (1) numerous latency measurements on a large number of heterogeneous devices are required in advance for meta-training; (2) there is a fundamental challenge for provably-good generalization to new unseen target devices that deviate significantly from the training device pool (i.e., out-of-distribution); and (3) the process of meta-learning and adaptation to new devices involves complex hyperparameter tuning, adding considerable uncertainties to the latency prediction performance. For example, in order to cover 24 devices with good generalization performance in the experiment, up to 18 heterogeneous devices are used for meta-training in which 900/4000 architecture latencies are collected for each device on the NAS-Bench-201/FBNet search space, while only the remaining 6 devices are used for testing [63]. Crucially, these meta latency predictors [63, 78] aim at producing accurate latency prediction with low prediction errors, which adds further challenges to the prediction model but is *unnecessary* for hardware-aware NAS. By contrast, what matters most is the architecture latency ranking on a target device, for which sophisticated (meta) latency predictors may not offer substantial benefits. We show both theoretically and empirically that one proxy device that has strong latency monotonicity with target devices (after proxy adaptation if needed) is enough for hardware-aware NAS, truly keeping the total latency evaluation cost at $\mathcal{O}(1)$.

Considering a synthetic latency metric aggregated over a few devices, simultaneous multi-device NAS [30] may not meet the latency constraint or achieve Pareto optimality for any involved device. Heuristic scaling approaches, e.g., by changing the number of layers and/or channels [97, 107, 108], can limit the architecture space and hence reduce both accuracy and latency evaluation costs, but they may also miss Pareto-optimal architectures because of their coarse scaling granularity. Architecture FLOPs is a device-agnostic proxy metric, but it cannot accurately reflect the true latency ranking of architectures on real devices [34, 64, 107]. While various proxy metrics (e.g., NTK [26]) have been considered for accuracy evaluation, our approach of using one proxy device is the first to address a complementary challenge of fast latency evaluation in the presence of many diverse devices.

3.8 Conclusion

In this chapter, we efficiently scale up hardware-aware NAS for diverse target devices. Concretely, we demonstrate latency monotonicity among different devices, and propose to use just one proxy device's latency predictor for NAS. When latency monotonicity is not satisfied between the proxy device and the target device, we propose an efficient transfer learning technique — adapting the proxy's latency predictor to the target device — to boost latency monotonicity. Overall, our approach results in a much lower total cost of latency evaluation, yet without losing Pareto optimality. For evaluation, we conduct experiments with different devices of different platforms on mainstream search spaces, including MobileNet-V2, MobileNet-V3, NAS-Bench-201 and FBNet spaces.

Appendix

In the appendix, we provide a summary of evolutionary search used in our experiment and additional experimental results.

3.9 Summary of Evolutionary Search

3.9.1 Description

To facilitate the readers' understanding, we provide a summary of the widely-used evolutionary search process for NAS, taking the MobileNet-V2 search space as example. More details of using evolutionary search in hardware-aware NAS can be found in [34, 115].

In our experiment, the total number of searchable blocks is 21, divided into five stages plus the last convolutional layer. Thus, we can use two 21-dimension vectors to represent the kernel size and expansion ratio of each block, respectively, and one 5-dimension vector to denote the depth of each stage. The depth can be chosen from "2, 3, 4", the kernel size can be "3, 5, 7", and the candidate expansion ratios are "3, 4, 6". Each individual member in evolutionary search consists of these three vectors. Here is an example individual: {"kernel_size": [5, 3, 5, 7, 5, 3, 5, 3, 7, 7, 5, 7, 5, 3, 3, 5, 5, 3, 5, 5, 3], "expansion_ratio": [3, 3, 4, 6, 4, 3, 4, 6, 4, 3, 6, 4, 3, 4, 3, 4, 4, 3, 3, 4, 3], "depth": [2, 2, 2, 2, 3]}.

To run evolutionary search, we first randomly sample the initial population of individuals according to the population size. Next, we evaluate the *fitness* of each individual in the population, where the fitness function is defined as:

$$(t-1) \cdot accuracy + t \cdot latency \tag{3.4}$$

where $t \in [0, 1]$ is the weight parameter to balance the tradeoff between accuracy and latency, and *accuracy* and *latency* are predicted values given by the accuracy and latency predictors, respectively. By varying $t \in [0, 1]$, we can obtain a set of Pareto-optimal architectures.

For each evolutionary search iteration, we select the fittest individuals as parents for reproduction, which will survive in the next generation and also breed new individuals through crossover. For example, if our population size is 1000 and the parent ratio is 0.25, we have 250 fittest individuals as parents. Then, we randomly select a pair of parents each time for crossover and generate a child. Within the crossover process, each element in the child's vector is chosen randomly from one of the parents'. Also, based on the mutation ratio setting, part of the offsprings will further perform mutation operations. For example, with mutation ratio 0.25 and mutation probability 0.1, 250 out of 750 children have a possibility of 0.1 to mutate. If a child is chosen to mutate, its kernel size, expansion ratio, and depth will be randomly sampled out of all the possible values for exploration. After crossover and mutation, we have a new population consisting of parents, bred children, and mutated children. Next, the fittest individuals are selected as new parents for next iteration. The above crossover and mutation steps will be repeated for the maximum evolutionary search iteration number.

3.9.2 Evolutionary Search Hyperparameters

Typically, the evolutionary search is not very sensitive against different hyperparameter settings, provided that the population size and iteration number are large enough and that there is adequate exploration. In Section 3.6, our hyperparameter settings are: population size is 1000, parent ratio is 0.25, mutation probability is 0.1, mutation ratio is 0.25, and we search for 50 generations given each latency constraint. We denote these settings as "EA#1". In Fig. 3.17, we change the hyperparameters to "EA#2": population

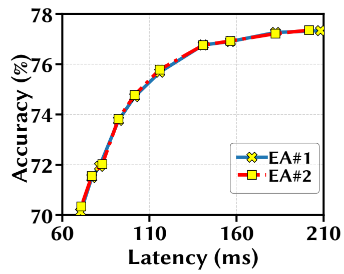


Figure 3.17: Pareto-optimal models searched on Samsung Galaxy S5e with different parameter settings for evolutionary search. "EA#1" denotes the parameter setting that population size is 1000, parent ratio is 0.25, mutation probability is 0.1 and mutation ratio is 0.25; while "EA#2" represents that population size is 500, parent ratio is 0.3, mutation probability is 0.2 and mutation ratio is 0.4.

size is 500, parent ratio is 0.3, mutation probability is 0.2, mutation ratio is 0.4, and run evolutionary search again on Samsuang Galaxy S5e. The results in Fig. 3.17 show that the searched Pareto-optimal models are almost identical to the original ones ("EA#1").

3.10 Additional Results

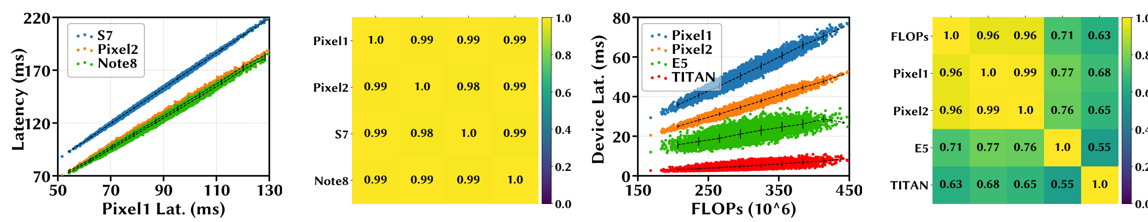
In this section, we present additional experimental results, including the demonstration of latency monotonicity based on third-party latency results and the effectiveness of our transfer learning technique in various mainstream search spaces.

3.10.1 Latency Monotonicity

To corroborate our own measurement and finding in Section 3.4, we examine latency monotonicity by leveraging third-party latency predictors and measurements for other devices.

Results on Predicted Latencies

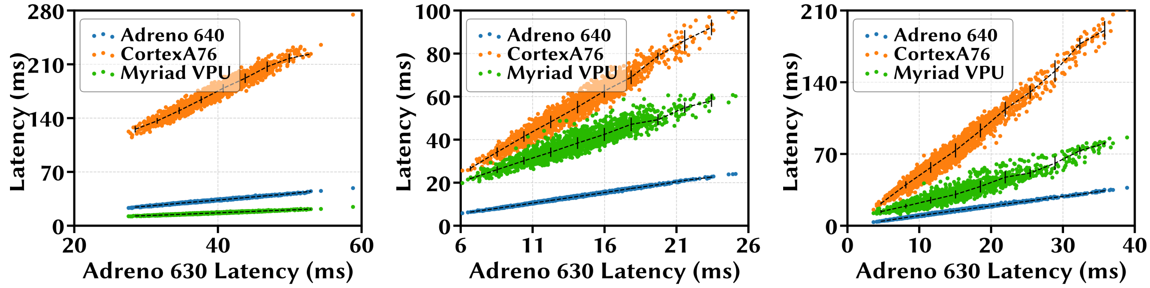
We obtain latency lookup tables for four mobile devices [21]: Google **Pixel1**, **Pixel2**, Samsung Galaxy **S7** edge, **Note8** in the MobileNet-V2 space with stage widths "32, 16, 24, 48, 80, 104, 192, 320, 1280". In addition, we obtain from [22] latency lookup tables for four cross-platform devices (used in [24]): Google **Pixel1**, **Pixel2**, **TITAN** Xp, **E5**-2640 v4 in the MobileNet-V2 space with different stage widths "32, 16, 24, 40, 80, 96, 192, 320, 1280" and measured with the MKL-DNN library. Note that latency predictors are very accurate (e.g., with an root mean squared error of less than 1% of the average) [13,24,34,125].



(a) (b) (c) (d)

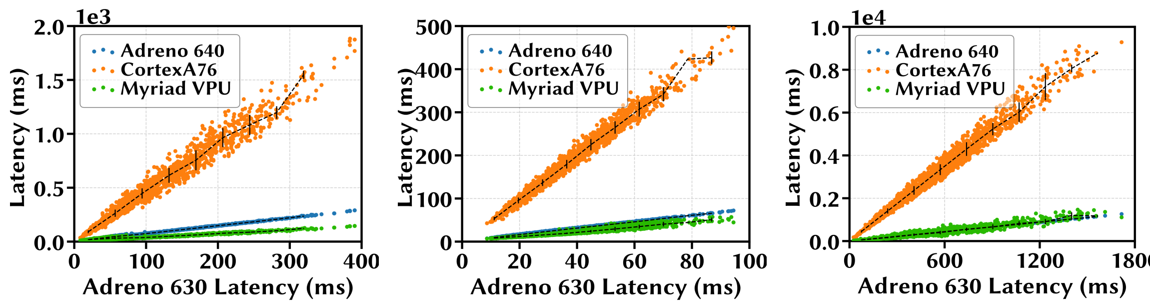
Figure 3.18: Latency monotonicity on third-party latency predictors [21, 22]. (a)(b) and (c)(d) use different search spaces and DNN acceleration libraries.

We randomly sample 10k models in each search space with variable depths of "2, 3, 4" in each stage, variable filter sizes of "3, 5, 7" in each convolutional layer, and variable expansion ratios of "3, 4, 6" in each block. We show the results in Fig. 3.18, which are in line with our experiments: latency monotonicity among mobile devices is strong (>0.95), while FLOP-latency ranking correlation for mobile devices is also quite strong but cross-platform latency monotonicity degrades.

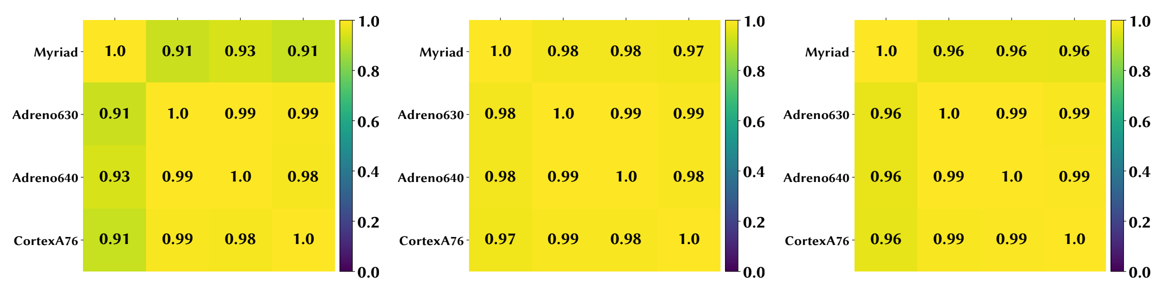


(a) GoogLeNet (b) MnasNet (c) MobileNet-V2

(d) GoogLeNet (e) MnasNet (f) MobileNet-V2



(g) ResNet (h) SqueezeNet (i) VGG



(j) ResNet (k) SqueezeNet (l) VGG

Figure 3.19: Latency results of 2000 models on CortexA76 CPU, Adreno 640 GPU, Adreno 630 GPU, and Myriad VPU, available in the dataset [128]. Search spaces: (a)(d) GoogLeNet, (b)(e) MnasNet, (c)(f) MobileNet-V2, (g)(j) ResNet, (h)(k) SqueezeNet, and (i)(l) VGG.

Results on Measured Latencies

We provide more evidence of latency monotonicity across different devices, and even across different DNN frameworks, using the nn-Meter results [128, 129]. Specifically, Fig. 3.19 shows the measured latencies and cross-device SRCCs in six different search spaces. We see that cross-device latency monotonicity strongly exists.

3.10.2 Results on MobileNet-V2

Search Space. Our backbone is MobileNet-V2 with multiplier 1.3, with the channel number in each block fixed. As shown in Fig. 3.20, The search space consists of depth of each stage, kernel size of convolutional layers, and expansion ratio of each block. The depth can be chosen from “2, 3, 4”, kernel size can be “3, 5, 7”, and candidate expansion ratios are “3, 4, 6”. There are five stages whose configurations can be searched, plus the kernel size and expansion ratio of the last inverted residual block.

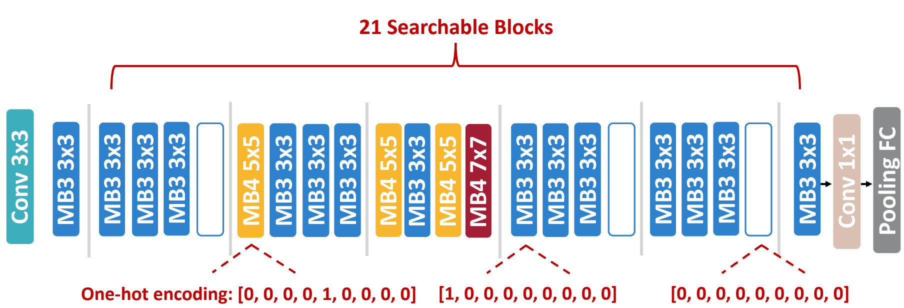


Figure 3.20: MobileNet-V2 search space and architectural encoding.

Proxy Adaptation. We use S5e as the default proxy device. Fig. 3.21 shows the original SRCC between S5e and desktop CPUs and GPUs, which are all below 0.8. We observe from Section 3.5 that SRCC of <0.8 is not enough to find Pareto-optimal architectures on the target device. Thus, in the absence of strong latency monotonicity between the default proxy device and the target device, proxy adaptation is necessary.

In the MobileNet-V2 search space, we have 21 searchable blocks in total, whose configurations can each be chosen out of nine kernel size and expansion ratio combinations or none (i.e., the block is not selected with a reduced stage depth). Thus, to represent an architecture, we simply use a 9-dimension one-hot vector $\mathbf{x}_b$ to encode the specification of each block. Given the proxy device's latency predictor as $L_{\mathbf{d}_0}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ built a priori, we collect the latencies of 80 sampled architectures on the target device, which are further split into 60 for training and 20 for validation. For i7-4790 and i7-4770HQ, we only need latencies of 30 sampled architectures for training. Next, by solving Eqn. (3.3), we obtain the AdaProxy device's latency predictor adapted to the target device, resulting in a significantly increased SRCC. Therefore, with the new latency predictor, we can quickly obtain Pareto-optimal architectures for the AdaProxy device, which are also very close to optimum for the target device (after removal of non-Pareto optimal architectures as specified in Section 3.5).

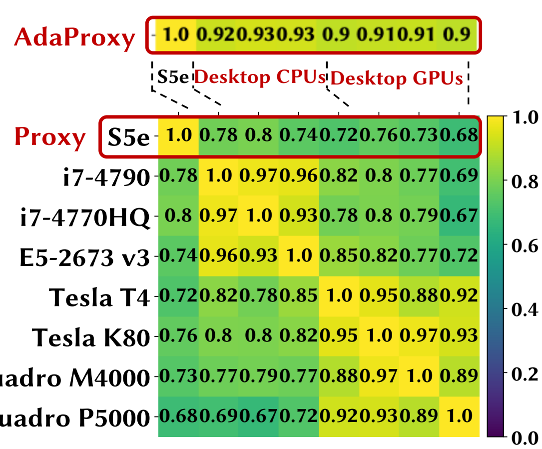
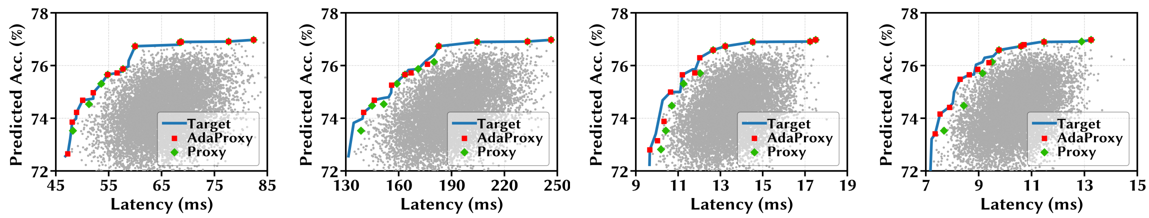


Figure 3.21: SRCC for various devices in the MobileNet-V2 space. S5e is the default proxy device. SRCC values boosted by AdaProxy are highlighted.



(a) Intel i7-4790 (b) E5-2673 v3 (c) Tesla K80 (d) Quadro P5000

Figure 3.22: Exhaustive search results based on 10k random architectures and predicted accuracies, for non-mobile target devices with the default S5e proxy and AdaProxy. SRCC values before and after proxy adaptation are shown in Fig. 3.21.

Results. Even without proxy adaptation, our results in Section 3.4 show that latency monotonicity among mobile devices (and between S5e and FPGA) is very strong. Here, we show the latency monotonicity between our proxy and desktop GPUs/CPUs, both with and without proxy adaptation. We see from Fig. 3.21 that the weak monotonicity can be significantly increased by using proxy adaptation. Thus, for a new target device that has a low SRCC with our default proxy device, we can simply use the AdaProxy device's latency predictor instead of profiling thousands of architectures and building a new one.

In Section 3.6, we already show the architecture performances for mobile target devices and some GPU/CPU devices. Now, we show the architecture performances for the remaining GPU/CPU devices in Fig. 3.22. We see that, due to the low SRCC, the architectures found by using the default proxy device's latency predictor may not overlap well with the oracle's Pareto-optimal boundary. On the other hand, with improved SRCC, the architectures found by using the AdaProxy device's latency predictor preserves Pareto optimality very well on the target devices. Again, this shows that our proposed transfer learning approach to boost the latency monotonicity is necessary and effective. For the devices in Fig. 3.22(d), we use 80 sampled architectures (50 for training, and 30 for validation and tuning λ) to construct AdaProxy. Note that the results are based on exhaustive search out of 10k randomly selected architectures, using the predicted accuracies as the true values. This is essentially considering a semi-oracle NAS process (on a small space of 10k architectures) assuming a perfect accuracy predictor. In other words, compared to evolutionary search (whose accuracy predictor itself is also not perfect), it has a more stringent requirement on the SRCC between the target device and the proxy (or AdaProxy) device. Thus, our approach works well even in this challenging case.

3.10.3 Results on NAS-Bench-201

Search Space. NAS-Bench-201 adopts a fixed cell search space [38]. Each searched cell is represented as a densely-connected directed acyclic graph (DAG), which is then stacked together with a pre-defined skeleton to construct an architecture. Specifically, as shown in Fig. 3.23, the search space considers four nodes and five representative operation candidates for the operation set, and varies the feature map sizes and dimensions of the final fully-connected layer to handle different datasets (i.e., CIFAR-10, CIFAR-100, and ImageNet16-120).

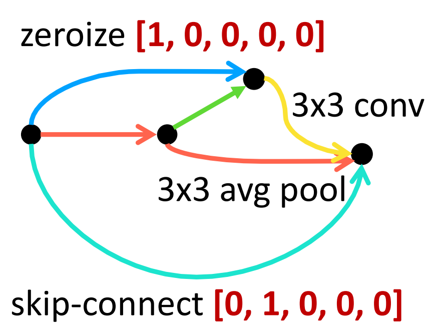


Figure 3.23: NAS-Bench-201 search space and architectural encoding.

Proxy Adaptation. We have four searchable nodes in total, the operation for each of which can be chosen from five candidates. Thus, we can use a 5-dimension one-hot vector to encode the specification of each node, although more advanced representation (e.g., graph-based [40]) is also applicable. Pixel3 is the default proxy device. Given the proxy device's latency predictor as $L_{\mathbf{d}_0}(\mathbf{x}) = \mathbf{w}^T\mathbf{x}$ built a priori, the training in transfer learning is based on measured latencies of 40 sampled architectures for the edge TPU and edge GPU, and 20 sampled architectures for Eyeriss and FPGA, respectively. In addition, validation uses another 20 sampled architectures for tuning the hyperparameter. Next, by solving Eqn. (3.3), we obtain the AdaProxy device's latency predictor adapted to the target device, resulting in a significantly increased SRCC. We show in Fig. 3.24 the latency monotonicity in terms of SRCC values, both with and without proxy adaptation. We see that the weak monotonicity can be significantly increased by using proxy adaptation.

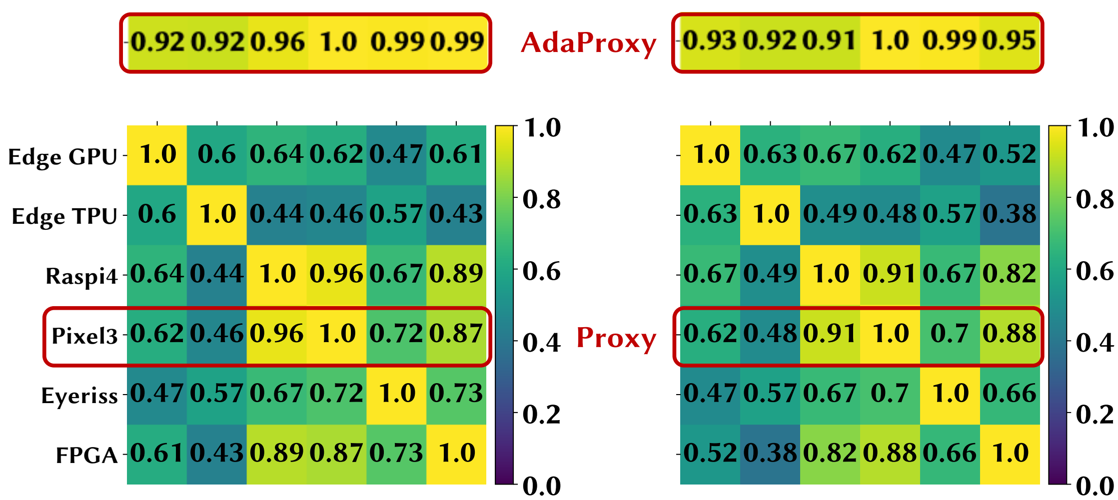
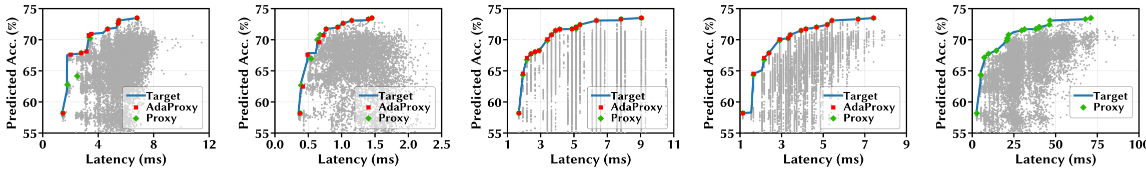
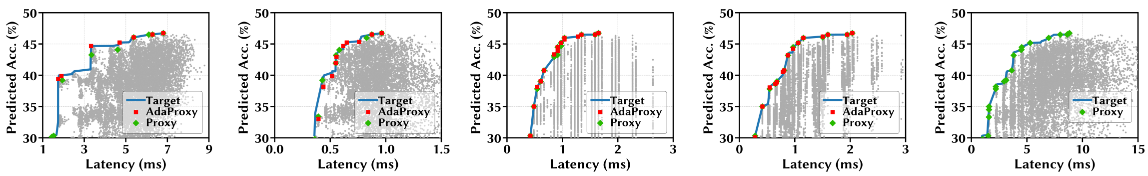


Figure 3.24: SRCC for various devices in the NAS-Bench-201 search space on CIFAR-100 (left) and ImageNet16-120 (right) datasets. Pixel3 is our proxy device. SRCC values boosted with AdaProxy are highlighted.



(a) Edge GPU (b) Edge TPU (c) Eyeriss (d) FPGA (e) Raspberry Pi 4

Figure 3.25: Exhaustive search results for different target devices on NAS-Bench-201 architectures (CIFAR-100 dataset) [38, 64]. Pixel3 is the proxy. SRCC values before and after proxy adaptation are shown in the left subfigure of Fig. 3.24.



(a) Edge GPU (b) Edge TPU (c) Eyeriss (d) FPGA (e) Raspberry Pi 4

Figure 3.26: Exhaustive search results for different target devices on NAS-Bench-201 architectures (ImageNet16-120 dataset) [38, 64]. Pixel3 is the proxy. SRCC values before and after proxy adaptation are shown in the right subfigure of Fig. 3.24.

Results. Considering the CIFAR-100 dataset, Fig. 3.25 shows optimal architectures found by using the proxy device's latency predictor, the adapted latency predictor, and the oracle, respectively. We can see that due to the pre-adaptation low SRCC values between the proxy device Pixel3 and the target devices, only a few architectures that are optimal for the proxy are still optimal for the target devices after architecture removal. Moreover, even the proxy's remaining optimal architectures can be far from optimality on the target device. For example, Fig. 3.25(a) shows that some of Pixel3's optimal architectures deviate from the Pareto-optimal boundary on the edge GPU. By using proxy adaptation and increasing the SRCC values, the AdaProxy's optimal architectures can be efficiently transferred to target devices while preserving optimality. The proxy device Pixel3 has a high SRCC of 0.96 with Raspi4, even without proxy adaptation. Thus, as shown in Fig. 3.25(e), the optimality of

Pixel3's architectures preserve very well on Raspi4. All these demonstrate the importance of strong monotonicity between the proxy and the target device, as well as the effectiveness of our proxy adaptation technique, for hardware-aware NAS with a total latency evaluation cost of $\mathcal{O}(1)$.

The same observation is also made in Fig. 3.26 for the ImageNet16-120 dataset.

3.10.4 Results on FBNet

Search Space. Similar to MobileNet-V2, the FBNet search space is also layer-wise with a fixed macro-architecture, which defines the number of layers and input/output dimensions of each layer and fixes the first and last three layers, with the remaining layers to be searched. As shown in Fig. 3.27, the overall search space consists of 22 searchable blocks: the first and last inverted residual blocks, and five stages within each of which there are at most four blocks. For each block, the kernel size can be chosen from "3, 5", and the expansion ratio can be "1, 3, 6". For the first and last 1x1 convolution layer, *group* convolution can be used to reduce the computation complexity. Also, each block can be skipped. Thus, there are nine candidate specification choice for each block (detailed configurations are shown in Table 2 of [120]).

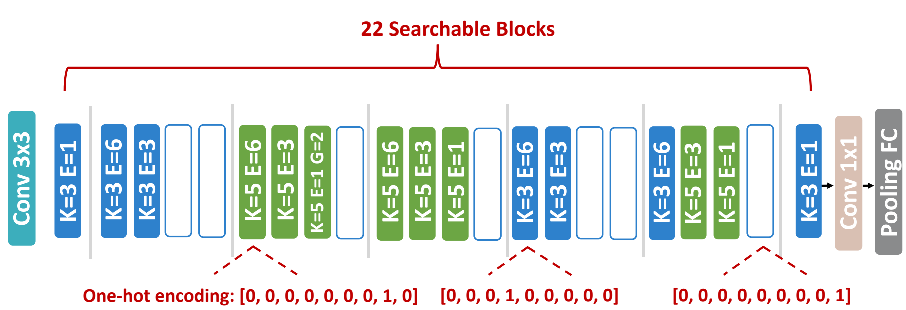


Figure 3.27: FBNet search space and architectural encoding.

Proxy Adaptation. We have 22 searchable blocks in total, the configuration for each of which can be chosen from the nine architecture candidates (including "Skip"). Then, we can still use a 9-dimension one-hot vector to encode each block. Using Pixel3 as the default proxy and the same approach as in Appendix 3.10.2, we can solve Eqn. (3.3) to create an AdaProxy device, which has SRCC of close to 0.9 or higher with the target device. In the transfer learning process, the numbers of sampled architectures for training are: 80 (Edge GPU), 40 (Raspi4), 30 (FPGA), 20 (Eyeriss). In addition, validation uses another 20 sampled architectures for tuning the hyperparameter.

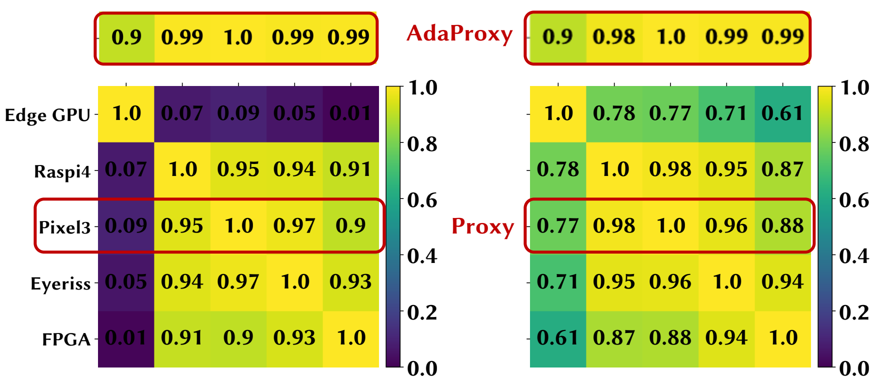


Figure 3.28: SRCC for various devices in the FBNet search spaces [64], on CIFAR-100 (left) and ImageNet16-120 (right) datasets respectively. Pixel3 is the proxy. SRCC values boosted by AdaProxy are highlighted.

Results. Our key focus is to achieve a high SRCC between the proxy (or AdaProxy) device and the target device, such that we can efficiently transfer the optimal architectures found on the proxy (or AdaProxy) device to the new target device without measuring latencies of thousands of architectures and building a new latency predictor. Since the accuracy results for architectures in the FBNet search space are not available [64], we only show in Fig. 3.28 the SRCC values instead, both with and without proxy adaptation. We can see that cross-platform SRCCs are greatly boosted (i.e., close to 1) with AdaProxy. By Theorem 1, the strong latency monotonicity ensures that the optimal architectures found on the proxy (or AdaProxy) device can be applied to new target devices.

3.10.5 Results on nn-Meter

The nn-Meter dataset released in [128, 129] includes measured inference latencies of 2000 models from 11 search spaces, including GoogLeNet, MnasNet and ProxylessNAS, etc on three mobile devices and one edge device: Pixel4 (Cortex A76 CPU), Mi9 (Adreno 640 GPU), Pixel3XL (Adreno 630 GPU), and Myriad VPU (Intel Movidius NCS2 edge device). Fig. 3.19 shows that the devices already have strong latency monotonicity with SRCC values greater than 0.9 on six search spaces. Among the remaining five search spaces, MobileNet-V1 and AlexNet are obsolete and phased out for hardware-ware NAS. Next, we apply our proxy adaptation technique on the other three search spaces: **MobileNet-V3**, **NAS-Bench-201**, and **ProxylessNAS**, which are mainstream and widely-used backbones in SOTA NAS algorithms.

MobileNet-V3

In our experiment, the number of searchable blocks in the MobileNet-V3 space is fixed as 12. For each block, the input, mid, and output channel number, and kernel size are variable from a set of candidates. Instead of directly using the kernel-based latency predictor in [129] that has a very large dimensionality for one-hot architectural encoding,

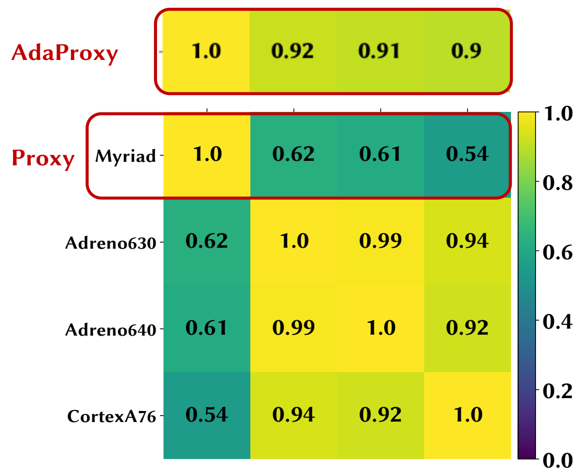


Figure 3.29: SRCC for various devices in the MobileNet-V3 search space [128, 129]. SRCC values boosted by AdaProxy are highlighted.

we use a simple block-level encoding method. Concretely, for each block, we use one-hot encodings for the input, mid, and output channel number and kernel size, respectively, and then concatenate these four one-hot vectors together to get the block-level encoding. After further concatenating the encoding vector of each block, we have a 530-dimension encoding for each architecture. Then, we build a simple 4-layer fully-connected neural network (with 500/250/100 neurons in each hidden layer) and train it on the latency data of the edge device Myriad VPU (used as the proxy), which has a low SRCC with the other three mobile devices. For the neural network training, we split the 1000 data samples (we use 1000 out of 2000 models for this experiment) into 800 for training and 200 for testing, set the learning rate as 0.01 and the batch size as 128, and train the network for 500 epoches. We also compress the network to 2 layers by fixing the first layer and appending it with another layer for the proxy device's latency predictor. Next, we apply the transfer learning method in Section 3.5.4 to the three target mobile devices. We use latencies of 150 architectures for transfer learning on Adreno 630/640 and 160 architectures for Cortex

A76, respectively, while using 20 architectures for validation. The relatively larger number of latency measurements needed for boosting the latency monotonicity is due in great part to two reasons: (1) MobileNet-V3 is a fairly complex search space, with many searchable operators; and (2) we intentionally address a challenging case where the proxy device has weak monotonicity with all the target devices. The results are shown in Fig. 3.29, where we can see that the SRCC values are significantly increased after proxy adaptation. despite the initially weak latency monotonicity.

ProxylessNAS

This search space is based on the MobileNet-V2 backbone, with variable expansion ratios, kernel sizes, inputs, and output channel numbers [24]. We apply a similar encoding approach as in the MobileNet-V3 space, and get a 783-dimension vector for each architecture in the nn-Meter dataset [128]. The Myriad VPU and Adreno 640 GPU is the only pair of devices with SRCC less than 0.9, with the pre-adaptation SRCC already being 0.87. We directly use the 783-dimension vector to perform transfer learning by updating the weights pre-trained on the proxy device (Adreno 640 GPU), with latencies of 30 sampled architectures for training and 20 architectures for validation. The results are shown in Fig. 3.30, demonstrating that the SRCC can be increased to over 0.9 after proxy adaptation.

NAS-Bench-201

For the NAS-Bench-201 space, we adopt the same encoding method as described in Appendix 3.10.3. We also consolidate the latency datasets released by three different

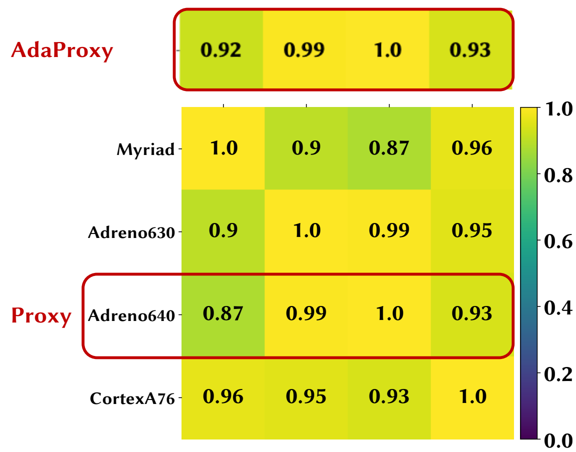


Figure 3.30: SRCC for various devices in the ProxylessNAS search space [128, 129]. SRCC values boosted by AdaProxy are highlighted. We only apply proxy adaptation for the Myriad VPU edge device, since the other target devices already have high SRCC of 0.9+ with the proxy device.

research studies [40, 64, 129] for the NAS-Bench-201 search space. The Myriad VPU edge device is the default proxy, while the target devices include FPGA, GPU, CPU, mobile, edge device, DSP, and TPU. Using the latencies of 20 sampled architectures for validation, the numbers of sampled architectures for training in the transfer learning process are: 30 for Edge GPU, Edge TPU, Eyeriss, FPGA, Raspi4, Adreno 630, Adreno 640, Cortex A76, CPU 855, GPU 855, 50 for DSP 855, 55 for Pixel3 and Jetson, 60 for GTX and i7, and 90 for Jetson 16. Note that the dataset in [128] only contains latencies for 2000 architectures in the NAS-Bench-201 space, and hence we only consider these 2000 architectures when calculating the cross-device SRCC values. We show the results in Fig. 3.31. While the latencies are measured by different research groups, on very different devices and using different deep learning frameworks, our proxy adaptation technique can still successfully increase the SRCC values to 0.9+, significantly boosting the otherwise weak latency monotonicity and keeping the total latency evaluation cost at $\mathcal{O}(1)$ for hardware-aware NAS.

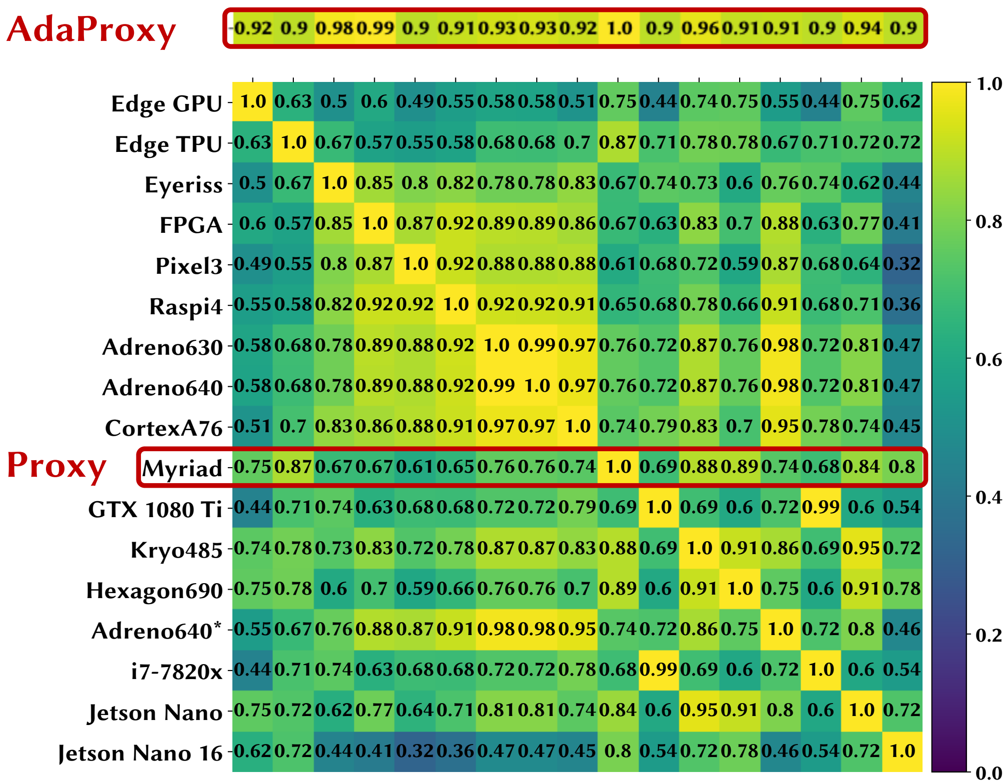


Figure 3.31: SRCC for various devices in the NAS-Bench-201 search space with latencies collected from [40, 64, 128, 129]. SRCC values boosted by AdaProxy are highlighted. "Adreno640" and "Adreno640*" denote model latencies measured by [129] and [40] respectively. "Jetson Nano" and "Jetson Nano 16" represent the latencies of FP32 and FP16 models correspondingly.

Chapter 4

A Semi-Decoupled Approach to Fast and Optimal Hardware-Software Co-Design of Neural Accelerators

4.1 Introduction

Neural architecture search (NAS) has been commonly used as a powerful tool to automate the design of efficient deep neural network (DNN) models [132]. As DNNs are being deployed on increasingly diverse devices and platforms, such as tiny Internet-of-Things devices and wearables, state-of-the-art (SOTA) NAS is turning hardware-aware by further taking into consideration the target hardware as a crucial factor that affects the resulting performance (e.g., inference latency and energy consumption) of NAS-designed models [14, 31, 64, 107, 108, 117, 120]

Likewise, optimizing hardware accelerators built on Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC), as well as the corresponding dataflows (e.g., scheduling DNN computations and mapping them on hardware), is also critical for speeding up DNN execution [3, 56, 123].

While both NAS and accelerator optimization can effectively improve the DNN performance (in terms of, e.g., accuracy and latency), they are traditionally performed in a siloed manner, without fully unleashing the potential of design flexibilities. As shown in recent studies [67, 79], such a decoupled approach does not explore potentially better combinations of architecture-accelerator designs, leading to highly sub-optimal DNN performance. As a result, co-design of neural architectures and accelerators (a.k.a., hardware-software co-design) has been emerging to discover jointly optimal architecture-accelerator designs [8, 56, 57, 79, 84, 123].

A common approach to hardware-software co-design is to use a nested loop: the outer loop searches over the hardware space while the inner loop searches for the optimal architecture given the hardware choice in the outer loop, or vice versa (i.e., outer loop for architectures and inner loops for hardware) [57,59]. Alternatively, one can also simultaneously search over the neural architecture and hardware spaces as a combined design choice [67].

While hardware-software co-design can further optimize DNN performance [124], it also exponentially enlarges the search space, presenting significant challenges. For example, the combination of architecture and accelerator design spaces can be up to 10^{861} [67]. Concretely, letting M and N be the sizes of the architecture space and hardware/accelerator space, respectively, the total search complexity is in the order of $\mathcal{O}(MN)$. By contrast, the fully-decoupled approach (i.e., separately performing NAS and accelerator optimization) has a total complexity of $\mathcal{O}(M+N)$, although it only results in sub-optimal designs.

Consequently, many studies have been focusing on speeding up the evaluation of co-design choices (e.g., using accuracy predictor and latency/energy simulation instead of actual measurement [23,79,115,123]), and/or improving the search efficiency (e.g., reinforcement learning or evolutionary search to co-optimize architecture and hardware [56,67,79]). Nonetheless, due to the $\mathcal{O}(MN)$ search space, the SOTA hardware-software co-design is still a time-consuming process, taking up a few or even tens of GPU hours for each new deployment scenario (e.g., changing the latency and/or energy constraints) [57,67].

Contributions. By settling in-between the fully-decoupled approach and the fully-coupled co-design approach, we propose a new *semi*-decoupled approach to reduce the size of the total co-search space $\mathcal{O}(MN)$ by orders of magnitude, yet without losing design optimality. Our approach builds on the *latency and energy monotonicity* — the architectures' ranking orders in terms of inference latency and energy consumption on different accelerators are highly correlated — and includes two stages. In **Stage 1**, we randomly choose a sample accelerator (a.k.a., a proxy accelerator), and then run hardware-aware NAS for K times to find a set $\mathcal{P}$ consisting of $K = |\mathcal{P}|$ optimal architectures for this proxy. Clearly, compared to M and N, the size of $\mathcal{P}$ is orders-of-magnitude smaller (e.g., 10-20 vs. 10^{18} [69]). Then, in **Stage 2**, instead of the entire architecture space as in the SOTA co-design, we only jointly search over the hardware space combined with the small set $\mathcal{P}$, which significantly reduces the total search space. Crucially, by latency and energy monotonicity, the set of optimal architectures is (approximately) the same for all accelerator designs, and hence selecting architectures out of $\mathcal{P}$ can still yield the optimal or very close-to-optimal architecture design.

We validate our approach by conducting experiments on a state-of-the-art neural accelerator simulator MAESTRO [62]. Our results confirm that strong latency and energy monotonicity exist among different accelerator designs. More importantly, by using one candidate accelerator as the proxy and obtaining its small set of optimal architectures, we can reuse the same architecture set for other accelerator candidates during the hardware search stage.

4.2 Problem Formulation

We focus on the design of a single neural architecture-accelerator pair. The main goal is to maximize the inference accuracy subject to a few design constraints such as inference latency, energy, and area [57]. Next, by denoting the neural architecture and hardware as a and h, respectively, we formulate the problem as follows:

$$\max_{a\in\mathcal{A},\, h\in\mathcal{H}} Accuracy(a) \tag{4.1}$$

$$s.t., \qquad Latency(a,h) \leq L \tag{4.2}$$

$$Energy(a,h) \leq E \tag{4.3}$$

$$HardwareResource(h) \leq H, \tag{4.4}$$

where the objective $Accuracy(a)$ depends on the architecture,[1] the first two constraints are set on the inference latency and energy consumption that depend on both the architecture and hardware choices, and the last constraint is on the hardware configuration itself (e.g., area) and hence independent of the architecture. We denote the optimal design as (a^*, h^*) which solves the optimization problem Eqns. (4.1)—(4.4). Note that, because of the combinatorial nature of the problem, *optimality* is not in a mathematically strict sense; instead, a design (a, h) is often considered as *optimal* if it is good enough in practice (e.g., better than or competitive with SOTA designs).

Suppose that the architecture space $\mathcal{A}$ and hardware space $\mathcal{H}$ have $M = |\mathcal{A}|$ and $N = |\mathcal{H}|$ design choices, respectively, which are both extremely large in practice. Thus, the co-design space $\mathcal{A} \times \mathcal{H}$ has a total of MN architecture-hardware combinations. This makes exhaustive search virtually impossible and adds significant challenges to co-design over the joint search space.

Remark. In our formulation, the notation of neural "architecture" $a \in \mathcal{A}$ can also broadly include other applicable design factors for the DNN model (e.g., weight quantization). Moreover, the hardware h implicitly includes the dataflow design, which is a downstream task based on the architecture and hardware choices. In the following, we also interchangeably use "accelerator" and "hardware" to refer to the hardware-dataflow combination unless otherwise specified. Thus, with different dataflows, the same hardware configuration will be considered as different $h \in \mathcal{H}$.

[1] The inference accuracy also depends on the network weight trained on a dataset, which is not a decision variable in hardware-software co-design and hence omitted.

4.3 A Semi-Decoupled Approach

In this section, we first review the existing architecture-accelerator design approaches, and then present our Aquaman approach.

4.3.1 Overview of Existing Approaches

Fully decoupled approach.

A straightforward approach is to separately optimize architectures and accelerators in a siloed manner by decoupling NAS from accelerator design [34, 115, 123]: first perform NAS to find *one* optimal architecture $\tilde{a} \in \mathcal{A}$, and then optimize the accelerator design for this particular architecture $\tilde{a}$; or, alteratively, first optimize the accelerator $\tilde{h} \in \mathcal{H}$, and then perform NAS to find the optimal architecture for this particular accelerator $\tilde{h}$. This approach has a total complexity in the order of $\mathcal{O}(M + N)$ where $M = |\mathcal{A}|$ and $N = |\mathcal{H}|$. But, the drawback is also significant: it does not fully exploit the flexibility of the co-design space and, as shown in several prior studies [57, 67, 79], can result in highly sub-optimal architecture-accelerator designs.

Fully coupled approach.

As can be seen in Eqns. (4.2) and (4.3), the inference latency and energy consumption is jointly determined by the architecture and hardware choices. Such entanglement of architecture and hardware is the key reason for the SOTA hardware-software co-design.

Concretely, a general co-design approach is to use a nested loop [79]. For example, the outer loop searches over the hardware space, whereas the inner loop searches for the optimal architecture given the hardware choice in the outer loop. Alternatively, another equivalent approach is to first search for neural architectures in the outer loop and then search for accelerators in the inner loop.

Here, we use "outer loop for hardware and inner loop for architecture" as an example. While the actual search method can differ from one study to another (e.g., reinforcement learning vs. evolutionary search [23, 79]), this nested search can be mathematically formulated as a bi-level optimization problem below:

$$\textbf{Outer:} \quad \max\nolimits_{h \in \mathcal{H}} Accuracy(a^*(h)) \tag{4.5}$$

$$s.t., \quad HardwareResource(h) \leq H, \tag{4.6}$$

where, given a choice of h, the architecture $a^*(h) = a^*(h; L, E)$ solves the inner hardware-aware NAS problem:

$$\textbf{Inner:} \quad \max\nolimits_{a \in \mathcal{A}} Accuracy(a) \tag{4.7}$$

$$s.t., \quad Latency(a, h) \leq L \tag{4.8}$$

$$Energy(a, h) \leq E. \tag{4.9}$$

In Eqn. (4.5), $Accuracy(\cdot)$ is still decided by the architecture, although we use $a^*(h) = a^*(h; L, E)$ to emphasize that the architecture is specifically optimized for the given hardware candidate h.

We see that, during the search for the optimal hardware h^* in the outer problem, the inner NAS problem is repeatedly solved as a subroutine and yields the optimal architecture

Table 4.1: Comparison of Different Approaches

Approach	Optimality	Complexity
Fully-decoupled separate design	No	$\mathcal{O}(M+N)$
Fullly-coupled co-design	Yes	$\mathcal{O}(MN)$
Semi-decoupled co-design	Yes	$\mathcal{O}(K(M+N))$

$a^*(h) = a^*(h; L, E)$ given each hardware choice h set by the outer search. For notational convenience, we also use $a^*(h)$ to represent $a^*(h; L, E)$ without causing ambiguity.

The focus of SOTA hardware-software co-design approaches have been primarily on speeding up the evaluation of architecture-hardware choices (e.g., using accuracy predictor and latency/energy simulation instead of actual measurement [23, 79, 115, 123]), and/or improving the search efficiency (e.g., reinforcement learning or evolutionary search to co-optimize architecture and hardware [56, 67, 79]). Nonetheless, evaluating one architecture-accelerator combination can still take up a few seconds in total (e.g., running MAESTRO to perform mapping/scheduling and estimate the latency and energy consumption takes 2-5 seconds on average [62]). Then, compounded by the exponentially large architecture and hardware space in the order of $\mathcal{O}(MN)$, the total hardware-software co-design cost is very high (e.g., a few or even tens of GPU hours for each deployment scenario [67, 79]).

4.3.2 Semi-Decoupled Co-Design

We propose a Aquaman approach — partially decoupling NAS from hardware search to reduce the total co-search cost from $\mathcal{O}(MN)$ to $\mathcal{O}(K(M+N))$ in a principled manner, where K is orders-of-magnitude less than M and N.

Performance monotonicity. The key intuition underlying our Aquaman approach is the latency and energy performance monotonicity — given different accelerators, the architectures' ranking orders in terms of both the inference latency and energy consumption are highly correlated. We can measure the ranking correlation in terms of the Spearman's rank correlation coefficient (SRCC), whose value lies within $[-1, 1]$ with "1" representing the identical ranking orders [6].

It has been shown in a recent hardware-aware NAS study [77] that the architectures' ranking orders in terms of inference latency are highly similar on different devices, with SRCCs often close to 0.9 or higher, especially among devices of the same platform (e.g., mobile phones). For example, if one architecture a_1 is faster than another architecture a_2 on one mobile phone, then it is very likely that a_1 is still faster than a_2 on another phone. One reason is that architectures are typically either computing-bound or memory-bound on devices of the same platform, which, by roofline analysis, results in similar rankings of their latencies [119]. Based on this property (a.k.a., latency monotonicity), it has been theoretically and empirically proved that the Pareto-optimal architectures on different devices are highly overlapping if not identical [77].

While the target hardware space chosen by the designer has many choices, it essentially covers one platform — neural accelerator under a set of hardware constraints. As a result, we expect latency monotonicity to be satisfied in our problem. Additionally, beyond the findings in [77], we observe in our experiments that *energy* monotonicity also holds: if one architecture a_1 is more energy-efficient than another architecture a_2 for one hardware choice, then it is very likely that a_1 is still more energy-efficient than a_2 for another hardware choice. Along with latency monotonicity, energy monotonicity will be later validated in our experiments. One reason for the energy monotonicity is that energy consumption is highly related to the inference latency with a strong correlation [64]. For simplicity, we use *performance* monotonicity to collectively refer to both latency and energy monotonicity.

Insights. The performance monotonicity leads to the following proposition, which generalizes the statement in [77] by considering both latency and energy monotonicity. We first note that, by solving the inner NAS problem under a set of latency and energy constraints in Eqns. (4.7)—(4.9), we can construct a set $\mathcal{P}(h) = (a_1^*(h; L_1, E_1), \cdots, a_K^*(h; L_K, E_K))$ of optimal architectures covering the architectures along the Pareto boundary. The size $K = |\mathcal{P}(h)|$ of the optimal architecture set depends on the granularity of latency and energy constraints we choose. In practice, K in the order of a few tens (e.g., $10 - 30$) is sufficient to cover a wide range of latency and energy constraints for our design target.

Proposition 2 *Given performance monotonicity, the set of optimal architectures* $\mathcal{P}(h) = (a_1^*(h; L_1, E_1), \cdots, a_K^*(h; L_K, E_K))$ *found by the inner hardware-aware NAS problem in Eqns.* (4.7)—(4.9) *is the same for all hardware choices, i.e.,* $\mathcal{P}(h_1) = \mathcal{P}(h_2)$*, for all* $h_1, h_2 \in \mathcal{H}$*.* ***Proof.*** *Consider two hardware choices* $h_1, h_2 \in \mathcal{H}$*. By performance monotonicity, we can replace the constraints* $Latency(a, h_1) \leq L_1$ *and* $Energy(a, h_2) \leq E_1$ *with another two equivalent constraints* $Latency(a, h_2) \leq L_1{}'$ *and* $Energy(a, h_2) \leq E_1{}'$*, respectively. By varying* E_1 *and* L_1 *over their feasible ranges, we obtain the optimal architecture set* $\mathcal{P}(h_1)$ *for* h_1*. Accordingly, due to the equivalent latency and energy constraints for* h_2*, we also obtain the optimal architecture set* $\mathcal{P}(h_2)$ *for* h_2*, thus completing the proof.* ■

Proposition 2 ensures that in the presence of performance monotonicity, the same set $\mathcal{P}(h)$ of optimal architectures apply to all $h \in \mathcal{H}$. Thus, we can also simply use $\mathcal{P}$ to denote the set of optimal architectures, which are essentially *shared* by $h \in \mathcal{H}$. Note

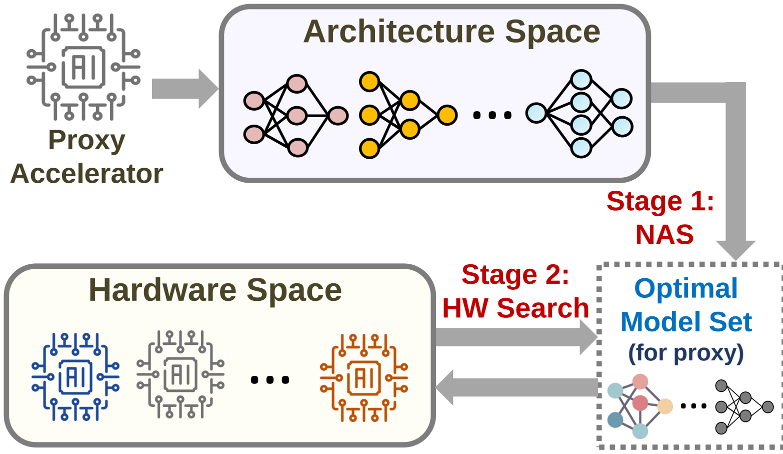


Figure 4.1: Overview of our Aquaman approach.

carefully that Proposition 2 does *not* mean that, given a specific pair of latency and energy constraints, we will have the same architecture $a^*(h_1; L, E) = a^*(h_2; L, E)$ for two hardware choices $h_1, h_2 \in \mathcal{H}$.

Nonetheless, once we have found $\mathcal{P} \subset \mathcal{A}$, there is no need to jointly search over the entire architecture-hardware space $\mathcal{A} \times \mathcal{H}$ any more. Instead, it is sufficient to merely search over the restricted architecture-hardware space $\mathcal{P} \times \mathcal{H}$. Importantly, the set $\mathcal{P}$ of optimal arachitectures is orders-of-magnitude smaller than the entire architecture space $\mathcal{A}$ (e.g., a few tens vs. 10^{18} in the DARTS architecture space [69]), thus significantly reducing the total hardware-software co-design cost without losing optimality.

Algorithm. Our Aquaman approach has two stages, as illustrated in Fig. 4.1 and summarized in Algorithm 6.

Stage 1: We randomly choose a sample accelerator $h_0 \in \mathcal{H}$, which we refer to as the *proxy* accelerator, and run hardware-aware NAS for K times to find a set of optimal architectures $\mathcal{P} = \mathcal{P}(h_0) = (a_1^*(h_0; L_1, E_1), \cdots, a_K^*(h_0; L_K, E_K))$. Specifically, $\mathcal{P}$ is con-

Algorithm 6 Semi-Decoupled Architecture-Accelerator Co-Design

Input: Architecture space $\mathcal{A}$, hardware space $\mathcal{H}$, sample hardware $h_0 \in \mathcal{H}$, and design constraints L, E, H in Eqns. (4.2), (4.3), (4.4)
Output: Optimal co-design (a^*, h^*)
Initilization: `Choose` K `latency and energy constraints` (L_k, E_k) `for` $k = 1, \cdots, K$`, set` $\mathcal{P} = \varnothing$`, and randomly choose` (a^*, h^*);
for $k = 1, \cdots, K$ **do**
  `For constraints` (L_k, E_k)`, run hardware-aware NAS to get optimal architecture` $a_k^*(h_0; L_k, E_k)$
  $\mathcal{P} = \mathcal{P} \bigcup \{a_k^*(h_0; L_k, E_k)\}$;
end for
for `each candidate hardware` $h \in \mathcal{H}$ **do**
  if $HardwareResource(h) \leq H$ **then**
    `Find optimal architecture` $a^*(h) \in \mathcal{P}$ `satisfying the latency and energy constraint` (L, E)
    if $Accuracy(a^*(h)) > Accuracy(a^*)$ **then**
      $(a^*, h^*) \leftarrow (a^*(h), h)$
    end if
  end if
end for

structed by setting K different latency and energy constraints and accordingly solving the inner NAS problem in Eqns. (4.7)—(4.9) for K times. Thus, the search cost in Stage 1 is $\mathcal{O}(KM)$ where $M = |\mathcal{A}|$.

Stage 2: We search for the optimal accelerator $h^* \in \mathcal{H}$. Specifically, given each candidate $h \in \mathcal{H}$ (selected by, e.g., reinforcement learning or evolutionary search [67, 79]), instead of searching over the entire architecture set $\mathcal{A}$, we obtain its corresponding optimal architecture $a^*(h)$ from the set $\mathcal{P} \subset \mathcal{A}$ constructed in Stage 1. Thus, the search cost in Stage 2 is $\mathcal{O}(KN)$ where $N = |\mathcal{H}|$.

4.3.3 Discussion

In practice, performance monotonicity may not be perfectly satisfied. Thus, the optimal architecture $a^*(h)$ corresponding to a candidate accelerator $h \in \mathcal{H}$ may not always strictly belong to the optimal architecture set $\mathcal{P}$ that is pre-constructed based on the proxy h_0. Nonetheless, by only searching over $\mathcal{P}$ for this candidate accelerator h, we can still find an architecture $a \in \mathcal{P}$ that is *close*-to-optimal. In fact, to speed up the NAS process and find competitive architectures, it is very common to use proxy/substitute metrics (such as accuracy predictor or the neural tangent kernel [26]) which only have SRCC of around 0.5–0.9 with the true performance. In our problem, we can also view the architectures' latency and energy performance on the proxy accelerator h_0 as the substitute performance on other accelerator candidates. Therefore, given the good albeit not necessarily close-to-perfect performance monotonicity, the architectures optimized specifically for the proxy are also sufficiently competitive ones for other accelerator candidates.

In [77], scalable hardware-aware NAS is proposed by utilizing latency monotonicity on various devices. Without considering energy consumption, a high SRCC (>0.9) for latency is needed to ensure that one proxy device's optimal architectures are still close to optimal on another device. In our problem, such high SRCC values are not necessarily needed, because we consider both energy and latency — moderate SRCC values on two performance metrics are enough. This is reflected in both our experiments and prior studies (e.g., two proxy metrics having moderate SRCC values with the true accuracy can estimate the accuracy performance very well [26]).

In the highly unlikely event of very low SRCCs (e.g., 0.2) between the proxy and other accelerator candidates, we can enlarge $\mathcal{P}$ by adding some approximately optimal architectures near the Pareto boundary (for the chosen proxy), such that they can be competitive choices for other candidate accelerators. Alternatively, we could use *a few* proxy accelerators, each having good latency and energy monotonicity with a subspace of accelerator design, and jointly construct an expanded set $\mathcal{P}$ of optimal architectures in Stage 1. In any case, the set $\mathcal{P}$ is orders-of-magnitude smaller than the entire architecture space or accelerator space.

Summary. The essence of our semi-decoupled approach is to use a proxy h_0 to find a small set of optimal architectures that also includes the actual optimal or *close-to-optimal* architectures for different accelerator candidates, thus reducing the total co-design complexity without losing optimality. This is significantly different from a typical fully-decoupled approach that pre-searches for *one* architecture and then find the matching accelerator, and also has a sharp contrast with a fully-coupled co-design approach that jointly searches over the entire architecture-accelerator space. The comparison of different approaches is also summarized in Table 4.1. Importantly, our approach focuses on reducing the search space complexity, and can be integrated with any actual NAS (Stage 1) and accelerator exploration techniques (Stage 2).

4.4 Experiment Setup

We provide details of our experiment setup as follows.

Accelerator hardware space. We employ an open-source tool MAESTRO [62] to simulate DNNs on the accelerator and measure inference metrics (e.g., latency and energy). MAESTRO supports a wide range of accelerators, including global shared scratchpad (i.e., L2 scratchapd), local PE scratchpad (i.e., L1 scratchpad), NoC, and a PE array organized into different hierarchies or dimensions.

DNN dataflow. Dataflow decides the DNN partitioning and scheduling strategies, which affects inference latency and energy performance. We consider three template dataflows: **KC-P** (motivated by NVDLA [88]), **YR-P** (motivated by Eyeriss [27]), and **X-P** (weight-stationary). Exhibiting different characteristics (e.g., temporal reuse of input activation and filter in YR-P vs. spatial reuse of input activation in KC-P), these representative dataflows are all supported by MAESTRO [62] and commonly used in SOTA hardware-software co-design [124].

Architecture space. We consider the following two spaces.

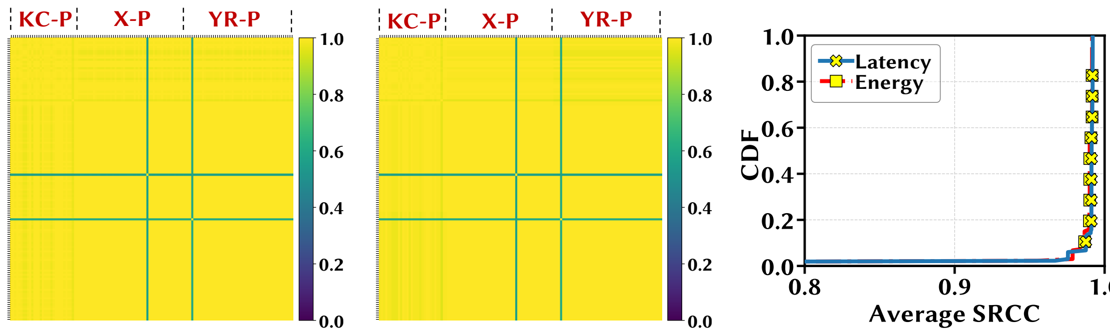
• *NAS-Bench-301*: It is a SOTA surrogate NAS benchmark built via deep ensembles and modeling uncertainty, which provides close-to-real predicted performances (i.e., accuracy and training time) of 10^{18} architectures on CIFAR-10 [103]. We consider the DARTS space [69], where each architecture is a stack of 20 convolutional cells, and each cell consists of seven nodes.

• *AlphaNet*: It is a new family of architectures on ImageNet discovered by applying a generalized α-divergence to supernet training [110]. Our search space is based on Table 7 of [110], with a slight variation that the channel width is fixed as "16, 16, 24, 32, 64, 112, 192, 216, 1792", and depth, kernel size, expansion ratio of the first and last inverted residual blocks are fixed as "1, 1", "3, 3", "1, 6", respectively. For other searchable inverted residual blocks, the candidate depth, kernel size, and expansion ratio are "2, 3, 4, 5, 6", "3, 5, 7", and "3, 4, 6", respectively.

Search strategy. Our approach can be integrated with any NAS and hardware search strategies. Here, we consider exhaustive search over a pre-sampled subspace. Specifically, for the NAS-Bench-301, we first sample 10k models. Then, based on the accuracy given by NAS-Bench-301 and FLOPs of these 10k models, we select 1017 models, including the Pareto-optimal front (in terms of predicted accuracy and FLOPs) and some random architectures. Similarly, for the AlphaNet space, we first sample 10k models and then select 1046 models based on the predicted accuracy given by the released accuracy predictor [43] and FLOPs. We consider a filtered space of 1k+ architectures (which include the Pareto-optimal ones out of the 10k sampled architectures), because using MAESTRO to measure the latency and energy of 10k models on thousands of different hardware-dataflow combinations is beyond our computational resource limit. For each of the three template dataflows, we sample 51 neural accelerators with different number of PEs, NoC bandwidth, and off-chip bandwidth per the MAESTRO document [89]. Specifically, the number of PEs can be chosen from "512, 256, 128, 64, 32, 16", candidate NoC bandwidths are from "300, 400, 500, 600, 700, 800, 900, 1000", and off-chip bandwidths are from "50, 100, 150, 200, 250, 275, 300, 325, 350". Note that some of our sampled hardware-dataflow pairs are not supported when running with KC-P and YR-P dataflows on MAESTRO. Thus, the actual numbers of sampled accelerators (i.e., hardware-dataflow combinations) are 133 for NAS-Bench-301 and 132 for AlphaNet, respectively. We also consider layer-wise mixture of different dataflows (Section 4.5.3) to create 5000 different hardware-dataflow combinations.

4.5 Experimental Results

In this section, we present our experimental results. We show that strong performance monotonicity exists in the hardware design space, and highlight that our Aquaman



(a) Latency SRCC (b) Energy SRCC (c) CDF of SRCC

Figure 4.2: Performance monotonicity. We test 1017 models sampled in DARTS search space on 133 accelerators.

approach can identify the optimal design at a much lower search complexity.

4.5.1 NAS-Bench-301

Performance monotonicity

We first validate that strong latency and energy performance monotonicity, quantified in SRCC, holds between different accelerators. The results are shown in Fig. 4.2. We see that, except for two accelerator choices that have SRCC less than 0.6 with others, all the other accelerators have almost perfect performance monotonicity with SRCC greater than 0.97. We also plot in Fig. 4.2(c) the cumulative distribution function (CDF) of the average SRCC values for all the sampled accelerators, where for each accelerator h the "average" is over the SRCC values of all the accelerator pairs that include h. We see that the vast majority of the accelerators have average SRCC close to 1.

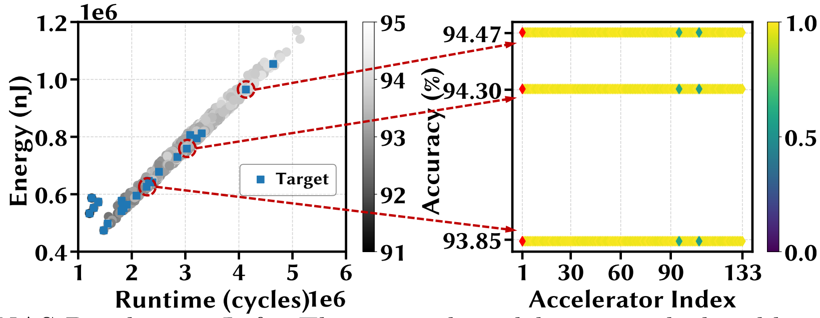


Figure 4.3: NAS-Bench-301. Left: The optimal models are marked in blue, and the grey scale indicates accuracy. Right: The accuracy of the model selected from the proxy's optimal model set. We test each accelerator as a different proxy. We also select two proxy accelerators (indexes 95 and 107) that have the lowest SRCCs with the target, and show the detailed results in Table 4.2.

Accelerator	SRCC		Hardware Config.				Model Performance		
Index	Latency	Energy	PEs	NoC	Off-chip	Dataflow	Latency (cycles)	Energy (nJ)	Accuracy (%)
1 (**target**)	1	1	512	900	350	KC-P	2279256	626090	93.85
107	0.556	0.567	64	400	250	YR-P	2279256	626090	93.85
95	0.595	0.595	256	800	350	X-P	2279256	626090	93.85
1 (**target**)	1	1	512	900	350	KC-P	3027992	758928	94.30
107	0.556	0.567	64	400	250	YR-P	3027992	758928	94.30
95	0.595	0.595	256	800	350	X-P	3027992	758928	94.30
1 (**target**)	1	1	512	900	350	KC-P	4130699	964783	94.47
107	0.556	0.567	64	400	250	YR-P	4130699	964783	94.47
95	0.595	0.595	256	800	350	X-P	4130699	964783	94.47

Table 4.2: Hardware configuration of the target and two proxy accelerators, and performance metrics of the selected optimal models on each of them. "Accelerator Index" corresponds to the x-axis in right of Fig. 4.3, the models on the target accelerator correspond to the circled ones in left of Fig. 4.3, while the models on the two proxy accelerators correspond to the diamond marks located on the accelerator indexes. The architecture configuration of the target models is further illustrated in Table 4.3.

Target Model	Model Architecture			
	Normal Cell Config.	Normal Cell Concat.	Reduce Cell Config.	Reduce Cell Concat.
#1	(skip_connect, 0), (skip_connect, 1), (skip_connect, 0), (skip_connect, 2), (sep_conv_5x5, 0), (skip_connect, 1), (dil_conv_5x5, 4), (skip_connect, 2)	[2, 3, 4, 5]	(sep_conv_3x3, 1), (sep_conv_3x3, 0), (dil_conv_3x3, 2), (skip_connect, 0), (sep_conv_5x5, 2), (avg_pool_3x3, 0), (dil_conv_3x3, 3), (sep_conv_3x3, 1)	[2, 3, 4, 5]
#2	(skip_connect, 0), (max_pool_3x3, 1), (sep_conv_3x3, 0), (skip_connect, 1), (skip_connect, 0), ('sep_conv_5x5, 3), (avg_pool_3x3, 4), (sep_conv_5x5, 1)	[2, 3, 4, 5]	(sep_conv_3x3, 1), (sep_conv_5x5, 0), (avg_pool_3x3, 0), (sep_conv_5x5, 1), (dil_conv_5x5, 3), (sep_conv_3x3, 2), (avg_pool_3x3, 4), (sep_conv_3x3, 0)	[2, 3, 4, 5]
#3	(dil_conv_5x5, 0), (skip_connect, 1), (max_pool_3x3, 0), (max_pool_3x3, 2), (sep_conv_5x5, 0), (dil_conv_3x3, 3), (dil_conv_5x5, 3), (dil_conv_5x5, 4)	[2, 3, 4, 5]	(skip_connect, 0), (dil_conv_3x3, 1), (sep_conv_3x3, 1), (sep_conv_5x5, 2), (skip_connect, 1), (max_pool_3x3, 0), (skip_connect, 1), (sep_conv_5x5, 2)	[2, 3, 4, 5]

Table 4.3: Architecture configuration of the target models in Fig. 4.3. The first row (i.e., target model #1) corresponds to the leftmost circled model in Fig. 4.3, and second row corresponds to the middle circled model, etc. These are the configurations for each convolutional cell constructing a complete model, which is a stack of 20 cells. For detailed explanation of the operations in the DARTS search space, please refer to [69] and [103].

Effectiveness

To demonstrate the effectiveness, suppose that we have an optimal architecture-accelerator pair (a^*, h^*) produced by the SOTA hardware-software co-design. We refer to the optimal accelerator as the "Target". By using our approach, in Stage 1, we first randomly choose a non-target accelerator h_0 as our proxy, and run hardware-aware NAS on this proxy to obtain the set $\mathcal{P}$ of optimal architectures. Next, in Stage 2, we will search over the accelerator space, retrieve the corresponding architecture a_0^* from $\mathcal{P}$ that best satisfies the latency and energy constraints, and keep the accelerator, whose corresponding architecture a_0^* has the highest accuracy, as the optimal accelerator. Thus, we prove the effectiveness of our approach if the architecture $a_0^* \in \mathcal{P}$ corresponding to the optimal accelerator found in Stage 2 produces (approximately) the same accuracy as a^* obtained using the SOTA co-design.

In our experiment, we consider a target optimal accelerator h^* as follows: 512 PEs, NoC bandwidth constraint 900, off-chip bandwidth constraint 350, and KC-P dataflow. In Fig. 4.3, we plot all the optimal architectures under various latency and energy constraints.[2] Then, we set three representative latency and energy consumption constraints, with their corresponding optimal models circled in red. Next, we test each of the other 132 accelerators as the proxy, and find the corresponding set $\mathcal{P}$, which includes about 20 optimal architectures for that proxy. Then, we select the architecture from $\mathcal{P}$ whose latency and energy are closest to the design constraints on the target accelerator. We see that by using *any* of the 132 accelerators as the proxy, our approach can still find the optimal architecture that

[2]MAESTRO returns the runtime cycles, instead of actual time, for the inference latency.

has (nearly) the same accuracy as that found by using SOTA hardware-software co-design. Importantly, even the proxy accelerator that has the lowest SRCC with the target can yield an competitive architecture with a good accuracy.

Total search cost

We now compare the total search cost incurred by exhaustive search over our sampled space. Using the coupled SOTA approach, the co-serach evaluates 133∗1017≈**135K** architecture-accelerator designs. In Stage 1 of our approach, we choose one proxy and evaluate 1017 architectures to obtain 20 optimal architectures for different latency and energy constraints. As we use exhaustive search, we do not need to run 20 times. In Stage 2, we evaluate the remaining 132 accelerators combined with the selected 20 architectures. Thus, the total search cost of our approach is 132∗20+1017≈**3.7K**, which is significantly less than 135K. While reinforcement learning or evolutionary search can improve the efficiency (especially on larger spaces), the order of the total cost remains the same. Moreover, when the architecture and accelerator spaces are larger, the relative advantage of our approach is even more significant.

4.5.2 AlphaNet

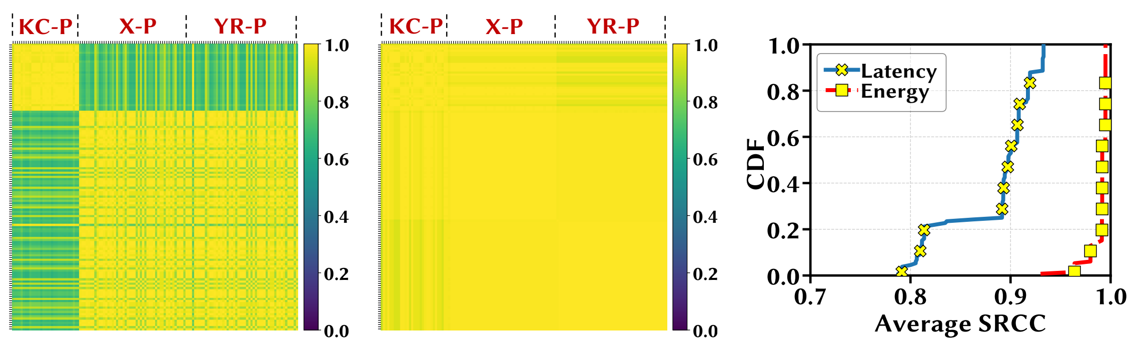
We now turn to the AlphaNet architecture space, and show the results in Fig. 4.4 and Fig. 4.5. While the SRCC values are lower than those in the NAS-Bench-301 case, they are still generally very high (e.g., mostly >0.9). Crucially, as shown in Fig. 4.5, our approach can successfully find an architecture that has (almost) the same accuracy as that obtained

Accelerator	SRCC		Hardware Config.				Model Config.		
Index	Latency	Energy	PEs	NoC	Off-chip	Dataflow	Latency (cycles)	Energy (nJ)	Accuracy (%)
1 (**target**)	1	1	512	900	350	KC-P	2061611	614779	69.60
64	0.638	0.945	512	400	350	X-P	2061611	602782	69.58
91	0.775	0.945	32	800	250	X-P	2046476	610891	69.60
1 (**target**)	1	1	512	900	350	KC-P	3367489	965462	71.18
64	0.638	0.945	512	400	350	X-P	3367489	965462	71.18
91	0.775	0.945	32	800	250	X-P	3367489	965462	71.18
1 (**target**)	1	1	512	900	350	KC-P	5923046	1858261	71.76
64	0.638	0.945	512	400	350	X-P	5923046	1858261	71.76
91	0.775	0.945	32	800	250	X-P	5923046	1858261	71.76

Table 4.4: Hardware configuration of the target and two proxy accelerators, and performance metrics of the selected optimal models on each of them. "Accelerator Index" corresponds to the x-axis in right of Fig. 4.5, models on the target accelerator correspond to the circled ones in left of Fig. 4.5, while the selected optimal models on proxy accelerators correspond to the diamond marks locating on the accelerator indexes. The architecture configuration of the target models is further illustrated in Table 4.5.

Target Model	Model Architecture				
	Resolution	Width	Kernel Size	Expansion Ratio	Depth
#1	224	16, 16, 24, 32, 64, 112, 192, 216, 1792	3, 3, 3, 3, 3, 3, 3	1, 4, 4, 6, 6, 5, 6	1, 3, 4, 3, 3, 3, 1
#2	288	16, 16, 24, 32, 64, 112, 192, 216, 1792	3, 3, 3, 3, 3, 7, 3	1, 4, 4, 5, 4, 5, 6	1, 3, 3, 3, 4, 4, 1
#3	288	16, 16, 24, 32, 64, 112, 192, 216, 1792	3, 3, 5, 7, 7, 7, 3	1, 6, 6, 6, 5, 5, 6	1, 6, 6, 3, 6, 6, 1

Table 4.5: Architecture configuration of target models in Fig. 4.5. The first row (i.e., target model #1) corresponds to the leftmost circled model in Fig. 4.5, and second row corresponds to the middle circled model, etc. For detailed explanation of the operations in AlphaNet search space, please refer to [111].



(a) SRCC of Latency

(b) SRCC of Energy Consumption

(c) CDF of SRCC

Figure 4.4: Performance monotonicity. We test 1046 models sampled in AlphaNet search space on 132 accelerators.

by using the SOTA coupled approach.

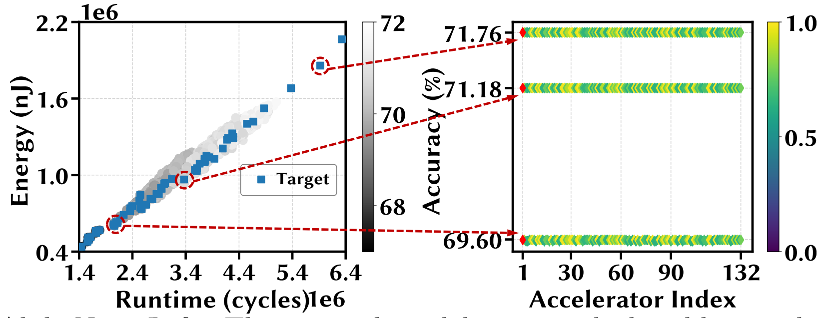
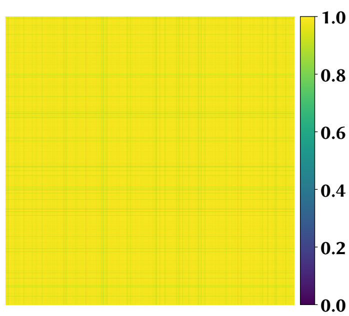
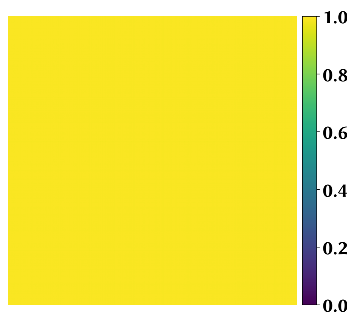


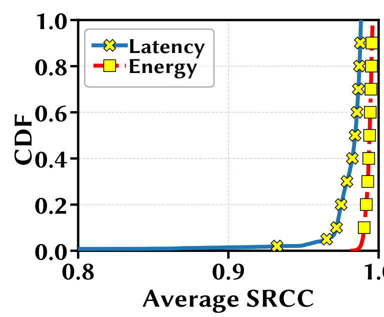
Figure 4.5: AlphaNet. Left: The optimal models are marked in blue, and the grey scale indicates accuracy. Right: The accuracy of the model selected from the proxy's optimal model set. We test each accelerator as a different proxy. We select two proxy accelerators (indexes 64 and 91) and show the detailed results in Table 4.4.



(a) Latency SRCC

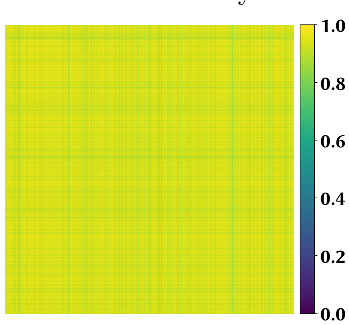


(b) Energy SRCC

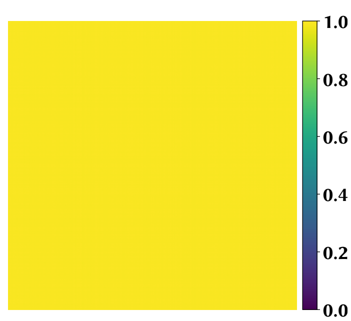


(c) CDF of SRCC

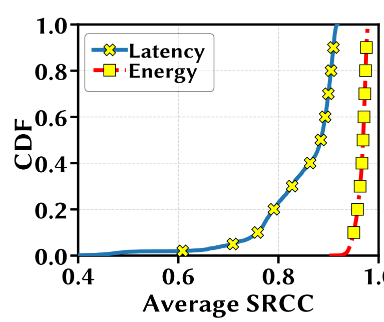
Figure 4.6: Performance monotonicity. We test 1017 models sampled in DARTS on 5000 accelerators with layer-wise mixed dataflows.



(a) Latency SRCC



(b) Energy SRCC



(c) CDF of SRCC

Figure 4.7: Performance monotonicity. We test 1046 models sampled in AlphaNet on 5000 accelerators with layer-wise mixed dataflows.

4.5.3 Layer-wise Mixed Dataflow

Ideally, each layer of a DNN model can be switched between accelerator hardware and dataflows to search for the best combination (especially in the multi-accelerator design case) [124]. To account for this, we divide each model into 22 parts: first and last convolutional layer, and evenly into 20 groups for all intermediate layers. For each part, it can be executed on any of our 51 sampled hardware configurations following any dataflow. We sample 5000 different mixtures for our models in NAS-Bench-301 and AlphaNet spaces, and report the SRCC results in Fig. 4.6 and 4.7, respectively. The results confirm again that strong performance monotonicity exists and ensures the effectiveness of our approach. We omit the optimal accuracy results due to the lack of space, while noting that they are similar to Figs. 4.3 and 4.5.

4.6 Related Work

NAS and accelerator design. Hardware-aware NAS has been actively studied to incorporate characteristics of target device and automate the design of optimal architectures subject to latency and/or energy constraints [14, 31, 64, 79, 94, 107, 120, 125]. These studies do not explore the hardware design space. A recent NAS study [77] explores latency monotonicity to scale up NAS across different devices, but it only considers latency constraints and, like other NAS studies, does explore the hardware design space. In parallel, there have also been studies on automating the design of accelerators for DNNs [123]. But, NAS and accelerator design have been traditionally studied in a siloed manner, resulting in sub-optimal designs.

Architecture-accelerator co-design. The studies on jointly optimizing architectures and accelerators have been quickly expanding. For example, [124] jointly optimizes neural architectures and ASIC accelerators using reinforcement learning, [59] performs a two-level (fast and slow) hardware exploration for each candidate neural architecture, [57] adopts a set of manually selected models as the hot start state for acceleration exploration, and [67] co-designs neural architecture, hardware configuration and dataflow, and employs evolutionary search to reduce the search cost. These studies primarily focus on improving the search efficiency given a certain search space. By contrast, we use a principled approach to reducing the total search space, without losing optimality.

4.7 Conclusion

In this chapter, we reduce the total hardware-software co-design cost by semi-decoupling NAS from accelerator design. Concretely, we demonstrate latency and energy monotonicity among different accelerators, and use just one proxy accelerator's optimal architecture set to avoid searching over the entire architecture space. Compared to the SOTA co-designs, our approach can reduce the total design complexity by orders of magnitude, without losing optimality. Finally, we validate our approach via experiments on two search spaces — NAS-Bench-301 and AlphaNet.

Chapter 5

Conclusions

To conclude this dissertation, we provide innovative scalable solutions to efficiently scale up DNN optimization for effcient inference on diverse edge devices. Previous device-unaware DNN optimization on a single target device cannot result in the optimal DNN model for all other devices, motivating the device-aware approach. While the existing approaches can produce an optimal DNN model for a given device, it lacks scalability facing a large number of heterogeneous edge devices. Specifically, even using prediction-assisted optimization, the often lengthy process of building an offline performance predictor is required for each target device.Therefore, scalable and automated approaches are crucial for effcient DNN optimization. This thesis can be summarized as two parts: part I automates DNN model selection for diverse edge devices to maximize users' QoE, using machine learning-based techniques, which is presented in Chapter 2; part II focuses on fully reaping the benefits of flexible design spaces – both neural architecture and hardware accelerators – to optimize neural network performance, which is investigated in Chapters 3 and 4.

The key challenge of user-centric DNN selection is that the QoE can be a very complicated function of the DNN performance metrics and is unkown *a priori*. In Chapter 2, we leverage a machine learning model to approximate the QoE function based on users' QoE feedback, which serves as "labels" for training the model. Based on it, an automated and user-centric DNN selection engine – Aquaman, is proposed to optimize DNN selection decisions and maximize users' QoE feedback. Aquaman consists of two integrated parts: QoE prediction and DNN model selection. To balance exploitation and exploration, Aquaman selects DNN models based on the QoE UCB, resulting in provably-efficient QoE performance compared to the oracle.

Going beyond selecting pre-existing DNN models, the key challenge of traversing the flexible neural architecture and accelerator design space is the exponentially large search space consisting of billions of or even more candidates. The reason is that evaluating and ranking the candidate architectures or architecture-accelerator pairs in terms of metrics of interest (e.g., accuracy and latency) can be extremely time-consuming. In Chapter 3, we focus on efficiently exploring the neural architecture search space for diverse target devices, namely hardware-aware NAS. We first demonstrate latency monotonicity among different devices, and propose to use just one proxy device's latency predictor for NAS on any target device. When latency monotonicity is not satisfied between the proxy device and the target device, we propose an efficient transfer learning technique – adapting the proxy's latency predictor to the target device – to boost latency monotonicity. Overall, our approach results in a much lower total cost of latency evaluation, yet without losing Pareto optimality. Our experiments with different devices of different platforms on mainstream neural architecture search spaces, including MobileNet-V2, MobileNet-V3, NAS-Bench-201 and FBNet spaces, prove the effectiveness of our **one proxy** approach.

To take into consideration the hardware design freedom, we further extend our analysis to hardware-software co-design of neural accelerators, which is investigated in Chapter 4. Based on the latency monotonicity of different architectures in Chapter 3, we additionally demonstrate latency and energy consumption monotonicity among different accelerators, and then propose to reduce the total hardware-software co-design cost by **semi-decoupling** NAS from accelerator design. Particularly, we use just one proxy accelerator's optimal architecture set to avoid searching over the entire architecture space. According to our evaluation on two search spaces – NAS-Bench-301 and AlphaNet, our approach can reduce the total design complexity by orders of magnitude, without losing optimality, compared to the SOTA co-designs.

Bibliography

[1] Yasin Abbasi-yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. In *NIPS*, Granada, Spain, December 2011.

[2] Mohamed S Abdelfattah, Abhinav Mehrotra, Łukasz Dudziak, and Nicholas Donald Lane. Zero-cost proxies for lightweight {nas}. In *International Conference on Learning Representations*, 2021.

[3] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, Virtually, April 2020.

[4] AI-Benchmark. Performance of mobile phones. `http://ai-benchmark.com/ranking_detailed.html`.

[5] Haldun Akoglu. User's guide to correlation coefficients. *Turkish Journal of Emergency Medicine*, 18(3):91 – 93, 2018.

[6] Haldun Akoglu. User's guide to correlation coefficients. *Turkish journal of emergency medicine*, 2018.

[7] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *MICRO*, 2016.

[8] Amazon. Amazon ec2 f1 instances. `https://aws.amazon.com/ec2/instance-types/f1/`, 2019.

[9] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. In *NIPS*, 2016.

[10] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. The power of optimization from samples. In *NIPS*, 2016.

[11] Eric Balkanski, Aviad Rubinstein, and Yaron Singer. The limitations of optimization from samples. In *STOC*, 2017.

[12] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In *ICML*, 2018.

[13] 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 *CVPR*, 2020.

[14] 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 *CVPR*, 2020.

[15] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search, 2021.

[16] Djallel Bouneffouf and Irina Rish. A survey on practical applications of multi-armed and contextual bandits. In *arXiv*, 2019.

[17] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[18] Leo Breiman. Random forests. *Machine learning*, 2001.

[19] Sébastien Bubeck, Nicolo Cesa-Bianchi, et al. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[20] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. *NeuralPower*: Predict and deploy energy-efficient convolutional neural networks. In *ACML*, 2017.

[21] Han Cai. Latency lookup tables of mobile devices. `https://file.lzhu.me/hancai/`.

[22] Han Cai. Latency lookup tables of mobile devices and GPUs. `https://file.lzhu.me/LatencyTools/tvm_lut/`.

[23] Han Cai, Chuang Gan, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *ICLR*, New Orleans, LA, USA, May 2019.

[24] Han Cai, Ligeng Zhu, and Song Han. ProxylessNas: Direct neural architecture search on target task and hardware. In *ICLR*, Long Beach, CA, USA, June 2019.

[25] Wei Chen, Yajun Wang, and Yang Yuan. Combinatorial multi-armed bandit: General framework, results and applications. In *ICML*, Atlanta, GA, USA, June 2013.

[26] Wuyang Chen, Xinyu Gong, and Zhangyang Wang. Neural architecture search on ImageNet in four GPU hours: A theoretically inspired perspective. In *International Conference on Learning Representations*, 2021.

[27] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News*, 2016.

[28] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Harris Teague, Yiran Chen, and Hai Li. MSNet: Structural wired neural architecture search for internet of things. In *ICCV Workshop*, 2019.

[29] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. A survey of model compression and acceleration for deep neural networks. 2017. Available at: `https://arxiv.org/abs/1710.09282`.

[30] Grace Chu, Okan Arikan, Gabriel Bender, Weijun Wang, Achille Brighton, Pieter-Jan Kindermans, Hanxiao Liu, Berkin Akin, Suyog Gupta, and Andrew Howard. Discovering multi-hardware mobile models via architecture search, 2020.

[31] Grace Chu, Okan Arikan, Gabriel Bender, Weijun Wang, Achille Brighton, Pieter-Jan Kindermans, Hanxiao Liu, Berkin Akin, Suyog Gupta, and Andrew Howard. Discovering multi-hardware mobile models via architecture search. In *CVPR*, 2021.

[32] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NeurIPS*, 2015.

[33] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu Yu, Peter Vajda, and Joseph E. Gonzalez. 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.

[34] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. ChamNet: Towards efficient network design through platform-aware model adaptation. In *CVPR*, Long Beach, CA, USA, June 2019.

[35] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[36] Emily L Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, and Rob Fergus. Exploiting linear structure within convolutional networks for efficient evaluation. In *NeurIPS*, Montreal, Quebec, Canada, December 2014.

[37] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. CirCNN: Accelerating and compressing deep neural networks using block-circulant weight matrices. In *MICRO*, 2017.

[38] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. In *International Conference on Learning Representations*, 2020.

[39] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *ICLR*, 2021.

[40] Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Brp-nas: Prediction-based nas using gcns, 2020.

[41] ELI5. Permutation importance. `https://eli5.readthedocs.io/en/latest/blackbox/permutation_importance.html`.

[42] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[43] Facebook. Alphanet: Improved training of supernet with alpha-divergence. `https://github.com/facebookresearch/AlphaNet`, 2021.

[44] Manuel López Galván. The multivariate bisection algorithm. *arXiv preprint arXiv:1702.05542*, 2017.

[45] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[46] Google. Android profiler. `https://developer.android.com/studio/profile/android-profiler`.

[47] Google. Tensorflow lite image classification app. `https://www.tensorflow.org/lite/models/image_classification/overview`.

[48] Google. Tensorflow lite image classification hosted models. `https://www.tensorflow.org/lite/guide/hosted_models`.

[49] 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 *ECCV*, 2020.

[50] 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*, 2016.

[51] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, San Juan, Puerto Rico, May 2016.

[52] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015.

[53] Mark Hill and Vijay Janapa Reddi. Gables: A roofline model for mobile SoCs. In *HPCA*, 2019.

[54] 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. 2017. Available at: `https://arxiv.org/abs/1704.04861`.

[55] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. Ai benchmark: All about deep learning on smartphones in 2019. In *ICCVW*, 2019.

[56] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving super-linear speedup across multi-fpga for real-time dnn inference. *ACM Trans. Embed. Comput. Syst.*, 18(5s):67:1–67:23, October 2019.

[57] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated CIrcuits and Systems*, 2020.

[58] Weiwen Jiang, Lei Yang, Sakyasingha Dasgupta, Jingtong Hu, and Yiyu Shi. Standing on the shoulders of giants: Hardware and neural architecture co-search with hot start. *IEEE Transactions on Computer-Aided Design of Integrated CIrcuits and Systems*, 2020.

[59] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[60] Sheng-Chun Kao, Arun Ramamurthy, and Tushar Krishna. Generative design of hardware-aware dnns. 2020.

[61] Johannes Kirschner, Ilija Bogunovic, Stefanie Jegelka, and Andreas Krause. Distributionally robust bayesian optimization. In *AISTATS*, Virtually, September 2020.

[62] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *MICRO*, 2019.

[63] Hayeon Lee, Sewoong Lee, Song Chong, and Sung Ju Hwang. HELP: hardware-adaptive efficient latency predictor for nas via meta-learning. In *NeurIPS*, 2021.

[64] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, Cong Hao, and Yingyan Lin. HW-NAS-Bench: Hardware-aware neural architecture search benchmark. In *International Conference on Learning Representations*, 2021.

[65] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, Raleigh, NC, USA, April 2010.

[66] F. Liang, C. Shen, W. Yu, and F. Wu. Towards optimal power control via ensembling deep neural networks. *IEEE Transactions on Communications*, 68(3):1760–1776, 2020.

[67] Yujun Lin, Mengtian Yang, and Song Han. NAAS: Neural Accelerator Architecture Search. In *2021 58th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2021.

[68] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018.

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

[70] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019.

[71] Ning Liu, Xiaolong Ma, Zhiyuan Xu, Yanzhi Wang, Jian Tang, and Jieping Ye. Auto-Compress: An automatic dnn structured pruning framework for ultra-high compression rates. In *AAAI*, New York, New York, USA., February 2020.

[72] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *MobiSys*, Munich, Germany, June 2018.

[73] Wei Liu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning. In *ASPLOS*, Virtually, March 2020.

[74] Yi Liu, Yuanshao Zhu, and JQ James. Resource-constrained federated learning with heterogeneous data: Formulation and analysis. *IEEE Transactions on Network Science and Engineering*, 2021.

[75] B. Lu, J. Yang, L. Y. Chen, and S. Ren. Automating deep neural network model selection for edge inference. In *CogMI*, Los Angeles, California, USA, December 2019.

[76] Bingqian Lu, Zheyu Yan, Yiyu Shi, and Shaolei Ren. A semi-decoupled approach to fast and optimal hardware-software co-design of neural accelerators. `https://arxiv.org/abs/2203.13921`.

[77] Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, and Shaolei Ren. One proxy device is enough for hardware-aware neural architecture search. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), dec 2021.

[78] Bingqian Lu, Jianyi Yang, and Shaolei Ren. Poster: Scaling up deep neural network optimization for edge inferenceâĂă. In *IEEE/ACM Symposium on Edge Computing (SEC)*, 2020.

[79] Qing Lu, Weiwen Jiang, Xiaowei Xu, Yiyu Shi, and Jingtong Hu. On neural architecture search for resource-constrained hardware platforms. In *ICCAD*, Westminster, Colorado, USA, November 2019.

[80] Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *NIPS*, 2018.

[81] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.

[82] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in DNN weight pruning for real-time execution on mobile device. In *AAAI*, New York, New York, USA, February 2020.

[83] Bradley McDanel, Surat Teerapittayanon, and HT Kung. Embedded binarized neural networks. 2017. Available at: `https://arxiv.org/abs/1709.02260`.

[84] Microsoft. Microsoft project brainwave. `https://www.microsoft.com/en-us/research/project/project-brainwave/`, 2019.

[85] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys Tutorials*, 20(4):2923–2960, Fourthquarter 2018.

[86] Arjun Mukherjee, Bing Liu, and Natalie Glance. Spotting fake reviewer groups in consumer reviews. In *WWW*, New York, New York, USA, April 2012.

[87] Xuefei Ning, Wenshuo Li, Zixuan Zhou, Tianchen Zhao, Yin Zheng, Shuang Liang, Huazhong Yang, and Yu Wang. A surgery of the neural architecture evaluators. *arXiv preprint arXiv:2008.03064*, 2020.

[88] NVIDIA. Nvdla deep learning accelerator. `http://nvdla.org`, 2018.

[89] Georgia Institute of Technology. Maestro's documentation. `http://maestro.ece.gatech.edu/docs/build/html/examples/running_maestro.html`, 2019.

[90] Samuel S. Ogden and Tian Guo. Characterizing the deep neural networks inference performance of mobile applications. In *arXiv*, 2019, `https://arxiv.org/abs/1909.04783`.

[91] Francesco Pase, Deniz Gunduz, and Michele Zorzi. Contextual multi-armed bandit with communication constraints. 2021.

[92] Basheer Qolomany, Ihab Mohammed, Ala Al-Fuqaha, Mohsen Guizani, and Junaid Qadir. Trust-based cloud machine learning model selection for industrial iot and smart city services. *IEEE Internet of Things Journal*, 2020.

[93] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *ECCV*, 2016.

[94] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *arXiv preprint arXiv:2006.02903*, 2020.

[95] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. Neural architecture search using bayesian optimisation with weisfeiler-lehman kernel. In *International Conference on Learning Representations*, 2021.

[96] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. Neural architecture search using bayesian optimisation with weisfeiler-lehman kernel. In *International Conference on Learning Representations*, 2021.

[97] Manas Sahni, Shreya Varshini, Alind Khare, and Alexey Tumanov. Comp{ofa} – compound once-for-all networks for faster multi-platform deployment. In *International Conference on Learning Representations*, 2021.

[98] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.

[99] Vidit Saxena, Joakim Jaldén, Joseph E. Gonzalez, Mats Bengtsson, Hugo Tullberg, and Ion Stoica. Contextual multi-armed bandits for link adaptation in cellular networks. In *Workshop on Network Meets AI & ML (NetAI)*, Beijing, China, August 2019.

[100] Ragini Sharma, Saman Biookaghazadeh, Baoxin Li, and Ming Zhao. Are existing knowledge transfer techniques effective for deep learning with edge devices? In *EDGE*, San Francisco, CA, USA, July 2018.

[101] Han Shi, Renjie Pi, Hang Xu, Zhenguo Li, James T. Kwok, and Tong Zhang. Multi-objective neural srchitecture search via predictive network performance optimization. *arXiv preprint arXiv:1911.09336*, 2019.

[102] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[103] Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. NAS-Bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777*, 2020.

[104] D. Stamoulis, E. Cai, D. Juan, and D. Marculescu. HyperPower: Power- and memory-constrained hyper-parameter optimization for neural networks. In *DATE*, 2018.

[105] 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 *ECML-PKDD*, 2019.

[106] 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 *CVPR*, Long Beach, CA, USA, June 2019.

[107] 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 *CVPR*, Long Beach, CA, USA, June 2019.

[108] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019.

[109] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. Philadelphia, PA, USA, June 2018.

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

[111] Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. Attentivenas: Improving neural architecture search via attentive sampling. In *CVPR*, 2021.

[112] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. HAT: Hardware-aware transformers for efficient natural language processing. In *ACL*, 2020.

[113] Haozhao Wang, Zhihao Qu, Qihua Zhou, Haobo Zhang, Boyuan Luo, Wenchao Xu, Song Guo, and Ruixuan Li. A comprehensive survey on training acceleration for large machine learning models in iots. *IEEE Internet of Things Journal*, 2021.

[114] Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. AlphaX: exploring neural architectures with deep neural networks and monte carlo tree search. *arXiv preprint arXiv:1903.11059*, 2019.

[115] Tianzhe Wang, Kuan Wang, Han Cai, Ji Lin, Zhijian Liu, Hanrui Wang, Yujun Lin, and Song Han. APQ: Joint search for network architecture, pruning and quantization policy. In *CVPR*, Virtually, June 2020.

[116] Yanzhi Wang. Towards ultra-efficient dnn inference acceleration on edge devices for wellbeing applications. In *HealthDL*, Toronto, Ontario, Canada, June 2020.

[117] Wei Wen, Hanxiao Liu, Hai Li, Yiran Chen, Gabriel Bender, and Pieter-Jan Kindermans. Neural predictor for neural architecture search. *arXiv preprint arXiv:1912.00848*, 2019.

[118] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009.

[119] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65âĂŞ76, apr 2009.

[120] 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 *CVPR*, 2019.

[121] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at Facebook: Understanding inference at the edge. In *HPCA*, Washington, DC, USA, February 2019.

[122] Liang Xiao, Yuzhen Ding, Donghua Jiang, Jinhao Huang, Dongming Wang, Jie Li, and H Vincent Poor. A reinforcement learning and blockchain-based trust mechanism for edge networks. *IEEE Transactions on Communications*, 2020.

[123] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *FPGA*, 2020.

[124] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. In *DAC*, 2020.

[125] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *ECCV*, 2018.

[126] 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 *ECCV*, 2020.

[127] Huaizheng Zhang, Linsen Dong, Guanyu Gao, Han Hu, Yonggang Wen, and Kyle Guan. Deepqoe: A multimodal learning framework for video quality of experience (qoe) prediction. *IEEE Transactions on Multimedia*, 2020.

[128] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. https://github.com/microsoft/nn-meter.

[129] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *MobiSys*, Virtually, July 2021.

[130] Yiyang Zhao, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. Few-shot neural architecture search. In *International Conference on Machine Learning*, 2021.

[131] Dongruo Zhou, Lihong Li, and Quanquan Gu. Neural contextual bandits with upper confidence bound-based exploration. In *ICML*, Virtually, July 2020.

[132] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

[133] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017.